

new/usr/src/Targetdirs

1

```
*****
69620 Thu Jun 28 15:57:05 2012
new/usr/src/Targetdirs
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 # CDDL HEADER START
2 #
3 # The contents of this file are subject to the terms of the
4 # Common Development and Distribution License (the "License").
5 # You may not use this file except in compliance with the License.
6 #
7 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
8 # or http://www.opensolaris.org/os/licensing.
9 # See the License for the specific language governing permissions
10 # and limitations under the License.
11 #
12 # When distributing Covered Code, include this CDDL HEADER in each
13 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
14 # If applicable, add the following below this CDDL HEADER, with the
15 # fields enclosed by brackets "[]" replaced with your own identifying
16 # information: Portions Copyright [yyyy] [name of copyright owner]
17 #
18 # CDDL HEADER END
19 #
20 #
21 #
22 # Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
23 # Copyright 2011, Richard Lowe
24 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25 # Copyright (c) 2012 by Delphix. All rights reserved.
26 #endif /* ! codereview */
27 #
28 #
29 #
30 # It is easier to think in terms of directory names without the ROOT macro
31 # prefix.  ROOTDIRS is TARGETDIRS with ROOT prefixes.  It is necessary
32 # to work with ROOT prefixes when controlling conditional assignments.
33 #
34 #
35 DIRLINKS=      $(SYM.DIRS)
36 $(BUILD64)    DIRLINKS += $(SYM.DIRS64)
37 #
38 FILELINKS= $(SYM.USRCCSLIB) $(SYM.USRLIB)
39 $(BUILD64)    FILELINKS += $(SYM.USRCCSLIB64) $(SYM.USRLIB64)
40 #
41 TARGETDIRS=    $(DIRS)
42 $(BUILD64)    TARGETDIRS += $(DIRS64)
43 #
44 TARGETDIRS    += $(FILELINKS) $(DIRLINKS)
45 #
46 i386_DIRS=    \
47 /boot/acpi    \
48 /boot/acpi/tables \
49 /boot/grub    \
50 /boot/grub/bin \
51 /platform/i86pc \
52 /usr/lib/xen  \
53 /usr/lib/xen/bin
```

new/usr/src/Targetdirs

2

```
55 sparc_DIRS=  \
56 /usr/lib/ldoms
57 #
58 # EXPORT DELETE START
59 XDIRS= \
60 /usr/lib/inet/wanboot
61 # EXPORT DELETE END
62 #
63 sparc_64ONLY= $(POUND_SIGN)
64 64ONLY=  $( $(MACH)_64ONLY)
65 #
66 $(64ONLY) MACH32_DIRS=/usr/ucb/$(MACH32)
67 #
68 DIRS= \
69 /boot \
70 /boot/solaris \
71 /boot/solaris/bin \
72 $( $(MACH)_DIRS) \
73 /dev \
74 /dev/dsk \
75 /dev/fd \
76 /dev/ipnet \
77 /dev/net \
78 /dev/rdisk \
79 /dev/rmt \
80 /dev/pts \
81 /dev/sad \
82 /dev/swap \
83 /dev/term \
84 /dev/vt \
85 /dev/zcons \
86 /devices \
87 /devices/pseudo \
88 /etc \
89 /etc/brand \
90 /etc/brand/solaris10 \
91 /etc/certs \
92 /etc/certs/CA \
93 /etc/openssl/certs/ \
94 /etc/cron.d \
95 /etc/crypto \
96 /etc/crypto/certs \
97 /etc/crypto/crls \
98 /etc/dbus-1 \
99 /etc/dbus-1/system.d \
100 /etc/default \
101 /etc/devices \
102 /etc/dev \
103 /etc/dfs \
104 /etc/dladm \
105 /etc/fs \
106 /etc/fs/nfs \
107 /etc/fs/zfs \
108 /etc/ftpd \
109 /etc/hal \
110 /etc/hal/fdi \
111 /etc/hal/fdi/information \
112 /etc/hal/fdi/information/10freedesktop \
113 /etc/hal/fdi/information/20thirdparty \
114 /etc/hal/fdi/information/30user \
115 /etc/hal/fdi/policy \
116 /etc/hal/fdi/policy/10osvendor \
117 /etc/hal/fdi/policy/20thirdparty \
118 /etc/hal/fdi/policy/30user \
119 /etc/hal/fdi/preprobe \
120 /etc/hal/fdi/preprobe/10osvendor \
```

```

121 /etc/hal/fdi/preprobe/20thirdparty \
122 /etc/hal/fdi/preprobe/30user \
123 /etc/ipadm \
124 /etc/iscsi \
125 /etc/pcsec \
126 /etc/security \
127 /etc/security/auth_attr.d \
128 /etc/security/exec_attr.d \
129 /etc/security/prof_attr.d \
130 /etc/security/tsol \
131 /etc/gss \
132 /etc/init.d \
133 /etc/dhcp \
134 /etc/lib \
135 /etc/mail \
136 /etc/mail/cf \
137 /etc/mail/cf/cf \
138 /etc/mail/cf/domain \
139 /etc/mail/cf/feature \
140 /etc/mail/cf/m4 \
141 /etc/mail/cf/mailer \
142 /etc/mail/cf/ostype \
143 /etc/mail/cf/sh \
144 /etc/net-snmp \
145 /etc/net-snmp/snmp \
146 /etc/opt \
147 /etc/rc0.d \
148 /etc/rc1.d \
149 /etc/rc2.d \
150 /etc/rc3.d \
151 /etc/rcS.d \
152 /etc/saf \
153 /etc/sasl \
154 /etc/sfw \
155 /etc/svc \
156 /etc/svc/profile \
157 /etc/svc/profile/site \
158 /etc/svc/volatile \
159 /etc/tm \
160 /etc/usb \
161 /etc/user_attr.d \
162 /etc/zfs \
163 /etc/zones \
164 /export \
165 /home \
166 /lib \
167 /lib/crypto \
168 /lib/inet \
169 /lib/fm \
170 /lib/secure \
171 /lib/svc \
172 /lib/svc/bin \
173 /lib/svc/capture \
174 /lib/svc/manifest \
175 /lib/svc/manifest/milestone \
176 /lib/svc/manifest/device \
177 /lib/svc/manifest/system \
178 /lib/svc/manifest/system/device \
179 /lib/svc/manifest/system/filesystem \
180 /lib/svc/manifest/system/security \
181 /lib/svc/manifest/system/svc \
182 /lib/svc/manifest/network \
183 /lib/svc/manifest/network/dns \
184 /lib/svc/manifest/network/ipsec \
185 /lib/svc/manifest/network/ldap \
186 /lib/svc/manifest/network/nfs \

```

```

187 /lib/svc/manifest/network/nis \
188 /lib/svc/manifest/network/rpc \
189 /lib/svc/manifest/network/security \
190 /lib/svc/manifest/network/shares \
191 /lib/svc/manifest/network/ssl \
192 /lib/svc/manifest/application \
193 /lib/svc/manifest/application/management \
194 /lib/svc/manifest/application/security \
195 /lib/svc/manifest/application/print \
196 /lib/svc/manifest/platform \
197 /lib/svc/manifest/platform/sun4u \
198 /lib/svc/manifest/platform/sun4v \
199 /lib/svc/manifest/site \
200 /lib/svc/method \
201 /lib/svc/monitor \
202 /lib/svc/seed \
203 /lib/svc/share \
204 /kernel \
205 /mnt \
206 /opt \
207 /platform \
208 /proc \
209 /root \
210 /sbin \
211 /system \
212 /system/contract \
213 /system/object \
214 /tmp \
215 /usr \
216 /usr/4lib \
217 /usr/ast \
218 /usr/ast/bin \
219 /usr/bin \
220 /usr/bin/$(MACH32) \
221 /usr/ccs \
222 /usr/ccs/bin \
223 /usr/ccs/lib \
224 /usr/demo \
225 /usr/demo/SOUND \
226 /usr/games \
227 /usr/has \
228 /usr/has/bin \
229 /usr/has/lib \
230 /usr/has/man \
231 /usr/has/man/manlhas \
232 /usr/include \
233 /usr/include/ast \
234 /usr/include/fm \
235 /usr/include/gssapi \
236 /usr/include/hal \
237 /usr/include/kerberosv5 \
238 /usr/include/libmilter \
239 /usr/include/libpolkit \
240 /usr/include/sasl \
241 /usr/include/scsi \
242 /usr/include/security \
243 /usr/include/sys/crypto \
244 /usr/include/tsol \
245 /usr/kernel \
246 /usr/kvm \
247 /usr/lib \
248 /usr/lib/abi \
249 /usr/lib/brand \
250 /usr/lib/brand/ipkg \
251 /usr/lib/brand/labeled \
252 /usr/lib/brand/shared \

```

```

253 /usr/lib/brand/sn1 \
254 /usr/lib/brand/solaris10 \
255 /usr/lib/class \
256 /usr/lib/class/FSS \
257 /usr/lib/class/FX \
258 /usr/lib/class/IA \
259 /usr/lib/class/RT \
260 /usr/lib/class/SDC \
261 /usr/lib/class/TS \
262 /usr/lib/crypto \
263 /usr/lib/drv \
264 /usr/lib/elfedit \
265 /usr/lib/fm \
266 /usr/lib/font \
267 /usr/lib/fs \
268 /usr/lib/fs/nfs \
269 /usr/lib/fs/proc \
270 /usr/lib/fs/smb \
271 /usr/lib/fs/zfs \
272 /usr/lib/gss \
273 /usr/lib/hal \
274 /usr/lib/inet \
275 /usr/lib/inet/dhcp \
276 /usr/lib/inet/dhcp/nsu \
277 /usr/lib/inet/dhcp/svc \
278 /usr/lib/inet/dhcp/svcadm \
279 /usr/lib/inet/ilb \
280 /usr/lib/inet/$(MACH32) \
281 $(XDIRS) \
282 /usr/lib/krb5 \
283 /usr/lib/link_audit \
284 /usr/lib/libp \
285 /usr/lib/lwp \
286 /usr/lib/mdb \
287 /usr/lib/mdb/kvm \
288 /usr/lib/mdb/proc \
289 /usr/lib/nfs \
290 /usr/net \
291 /usr/net/servers \
292 /usr/lib/pool \
293 /usr/lib/python2.6 \
294 /usr/lib/python2.6/vendor-packages \
295 /usr/lib/python2.6/vendor-packages/solaris \
296 /usr/lib/python2.6/vendor-packages/zfs \
297 /usr/lib/python2.6/vendor-packages/beam \
298 /usr/lib/rcap \
299 /usr/lib/rcap/$(MACH32) \
300 /usr/lib/sa \
301 /usr/lib/saf \
302 /usr/lib/sasl \
303 /usr/lib/scsi \
304 /usr/lib/secure \
305 /usr/lib/security \
306 /usr/lib/smberv \
307 /usr/lib/vscan \
308 /usr/lib/zfs \
309 /usr/lib/zones \
310 /usr/old \
311 /usr/platform \
312 /usr/proc \
313 /usr/proc/bin \
314 /usr/sadm \
315 /usr/sadm/install \
316 /usr/sadm/install/bin \
317 /usr/sadm/install/scripts \
318 /usr/sbin \

```

```

319 /usr/sbin/$(MACH32) \
320 /usr/share \
321 /usr/share/applications \
322 /usr/share/audio \
323 /usr/share/audio/samples \
324 /usr/share/audio/samples/au \
325 /usr/share/gnome \
326 /usr/share/gnome/autostart \
327 /usr/share/hwdata \
328 /usr/share/lib \
329 /usr/share/lib/ccs \
330 /usr/share/lib/tmac \
331 /usr/share/lib/ldif \
332 /usr/share/lib/xml \
333 /usr/share/lib/xml/dtd \
334 /usr/share/man \
335 /usr/share/man/man1 \
336 /usr/share/man/man1b \
337 /usr/share/man/man1c \
338 /usr/share/man/man1m \
339 /usr/share/man/man2 \
340 /usr/share/man/man3 \
341 /usr/share/man/man3bsm \
342 /usr/share/man/man3c \
343 /usr/share/man/man3c_db \
344 /usr/share/man/man3cfgadm \
345 /usr/share/man/man3commutil \
346 /usr/share/man/man3contract \
347 /usr/share/man/man3cpc \
348 /usr/share/man/man3curses \
349 /usr/share/man/man3dat \
350 /usr/share/man/man3devid \
351 /usr/share/man/man3devinfo \
352 /usr/share/man/man3dlpi \
353 /usr/share/man/man3dns_sd \
354 /usr/share/man/man3elf \
355 /usr/share/man/man3exacct \
356 /usr/share/man/man3ext \
357 /usr/share/man/man3fcoe \
358 /usr/share/man/man3fstyp \
359 /usr/share/man/man3gen \
360 /usr/share/man/man3gss \
361 /usr/share/man/man3head \
362 /usr/share/man/man3iscsit \
363 /usr/share/man/man3kstat \
364 /usr/share/man/man3kvm \
365 /usr/share/man/man3ldap \
366 /usr/share/man/man3lgrp \
367 /usr/share/man/man3lib \
368 /usr/share/man/man3libucb \
369 /usr/share/man/man3mail \
370 /usr/share/man/man3malloc \
371 /usr/share/man/man3mp \
372 /usr/share/man/man3mpapi \
373 /usr/share/man/man3nsl \
374 /usr/share/man/man3nvpair \
375 /usr/share/man/man3pam \
376 /usr/share/man/man3papi \
377 /usr/share/man/man3perl \
378 /usr/share/man/man3picl \
379 /usr/share/man/man3picltree \
380 /usr/share/man/man3pool \
381 /usr/share/man/man3proc \
382 /usr/share/man/man3project \
383 /usr/share/man/man3resolv \
384 /usr/share/man/man3rpc \

```

```

385 /usr/share/man/man3rsm \
386 /usr/share/man/man3sas1 \
387 /usr/share/man/man3scf \
388 /usr/share/man/man3sec \
389 /usr/share/man/man3secdb \
390 /usr/share/man/man3sip \
391 /usr/share/man/man3slp \
392 /usr/share/man/man3socket \
393 /usr/share/man/man3stmf \
394 /usr/share/man/man3sysevent \
395 /usr/share/man/man3tecla \
396 /usr/share/man/man3tnf \
397 /usr/share/man/man3tsol \
398 /usr/share/man/man3ucb \
399 /usr/share/man/man3uuid \
400 /usr/share/man/man3vo1mgt \
401 /usr/share/man/man3xcurses \
402 /usr/share/man/man3xnet \
403 /usr/share/man/man4 \
404 /usr/share/man/man5 \
405 /usr/share/man/man7 \
406 /usr/share/man/man7d \
407 /usr/share/man/man7fs \
408 /usr/share/man/man7i \
409 /usr/share/man/man7ipp \
410 /usr/share/man/man7m \
411 /usr/share/man/man7p \
412 /usr/share/man/man9 \
413 /usr/share/man/man9e \
414 /usr/share/man/man9f \
415 /usr/share/man/man9p \
416 /usr/share/man/man9s \
417 /usr/share/src \
418 /usr/snadm \
419 /usr/snadm/lib \
420 /usr/ucb \
421 $(MACH32_DIRS) \
422 /usr/ucblib \
423 /usr/xpg4 \
424 /usr/xpg4/bin \
425 /usr/xpg4/include \
426 /usr/xpg4/lib \
427 /usr/xpg6 \
428 /usr/xpg6/bin \
429 /var \
430 /var/adm \
431 /var/adm/exacct \
432 /var/adm/log \
433 /var/adm/pool \
434 /var/adm/sa \
435 /var/adm/sm.bin \
436 /var/adm/streams \
437 /var/cores \
438 /var/cron \
439 /var/db \
440 /var/db/ipf \
441 /var/games \
442 /var/idmap \
443 /var/krb5 \
444 /var/krb5/rcache \
445 /var/krb5/rcache/root \
446 /var/ld \
447 /var/log \
448 /var/log/pool \
449 /var/logadm \
450 /var/mail \

```

```

451 /var/news \
452 /var/opt \
453 /var/preserve \
454 /var/run \
455 /var/saf \
456 /var/sadm \
457 /var/sadm/install \
458 /var/sadm/install/admin \
459 /var/sadm/install/logs \
460 /var/sadm/pkg \
461 /var/sadm/security \
462 /var/smb \
463 /var/smb/cvol \
464 /var/smb/cvol/windows \
465 /var/smb/cvol/windows/system32 \
466 /var/smb/cvol/windows/system32/vss \
467 /var/spool \
468 /var/spool/cron \
469 /var/spool/cron/atjobs \
470 /var/spool/cron/crontabs \
471 /var/spool/lp \
472 /var/spool/pkg \
473 /var/spool/uucp \
474 /var/spool/uucppublic \
475 /var/svc \
476 /var/svc/log \
477 /var/svc/manifest \
478 /var/svc/manifest/milestone \
479 /var/svc/manifest/device \
480 /var/svc/manifest/system \
481 /var/svc/manifest/system/device \
482 /var/svc/manifest/system/filesystem \
483 /var/svc/manifest/system/security \
484 /var/svc/manifest/system/svc \
485 /var/svc/manifest/network \
486 /var/svc/manifest/network/dns \
487 /var/svc/manifest/network/ipsec \
488 /var/svc/manifest/network/ldap \
489 /var/svc/manifest/network/nfs \
490 /var/svc/manifest/network/nis \
491 /var/svc/manifest/network/rpc \
492 /var/svc/manifest/network/routing \
493 /var/svc/manifest/network/security \
494 /var/svc/manifest/network/shares \
495 /var/svc/manifest/network/ssl \
496 /var/svc/manifest/application \
497 /var/svc/manifest/application/management \
498 /var/svc/manifest/application/print \
499 /var/svc/manifest/application/security \
500 /var/svc/manifest/platform \
501 /var/svc/manifest/platform/sun4u \
502 /var/svc/manifest/platform/sun4v \
503 /var/svc/manifest/site \
504 /var/svc/profile \
505 /var/uucp \
506 /var/tmp \
507 /var/tsol \
508 /var/tsol/doors

510 sparcv9_DIRS64= \
511 /platform/sun4u \
512 /platform/sun4u/lib \
513 /platform/sun4u/lib/$(MACH64) \
514 /usr/platform/sun4u \
515 /usr/platform/sun4u/sbin \
516 /usr/platform/sun4u/lib \

```

```

517 /platform/sun4v/lib \
518 /platform/sun4v/lib/$(MACH64) \
519 /usr/platform/sun4v/sbin \
520 /usr/platform/sun4v/lib \
521 /usr/platform/sun4u-us3/lib \
522 /usr/platform/sun4u-opl/lib

524 amd64_DIRS64= \
525 /platform/i86pc/amd64

527 DIRS64= \
528 $( $(MACH64)_DIRS64) \
529 /lib/$(MACH64) \
530 /lib/crypto/$(MACH64) \
531 /lib/fm/$(MACH64) \
532 /lib/secure/$(MACH64) \
533 /usr/bin/$(MACH64) \
534 /usr/ccs/bin/$(MACH64) \
535 /usr/ccs/lib/$(MACH64) \
536 /usr/lib/$(MACH64) \
537 /usr/lib/$(MACH64)/gss \
538 /usr/lib/brand/sn1/$(MACH64) \
539 /usr/lib/brand/solaris10/$(MACH64) \
540 /usr/lib/elfedit/$(MACH64) \
541 /usr/lib/fm/$(MACH64) \
542 /usr/lib/fs/nfs/$(MACH64) \
543 /usr/lib/fs/smb/$(MACH64) \
544 /usr/lib/inet/$(MACH64) \
545 /usr/lib/krb5/$(MACH64) \
546 /usr/lib/libp/$(MACH64) \
547 /usr/lib/link_audit/$(MACH64) \
548 /usr/lib/lwp/$(MACH64) \
549 /usr/lib/mdb/kvm/$(MACH64) \
550 /usr/lib/mdb/proc/$(MACH64) \
551 /usr/lib/rcap/$(MACH64) \
552 /usr/lib/sasl/$(MACH64) \
553 /usr/lib/scsi/$(MACH64) \
554 /usr/lib/secure/$(MACH64) \
555 /usr/lib/security/$(MACH64) \
556 /usr/lib/smbsrv/$(MACH64) \
557 /usr/lib/abi/$(MACH64) \
558 /usr/sbin/$(MACH64) \
559 /usr/ucb/$(MACH64) \
560 /usr/ucb/lib/$(MACH64) \
561 /usr/xpg4/lib/$(MACH64) \
562 /var/ld/$(MACH64)

564 # /var/mail/:saved is built directly by the rootdirs target in
565 # /usr/src/Makefile because of the colon in its name.

567 # macros for symbolic links
568 SYM.DIRS= \
569 /bin \
570 /dev/stdin \
571 /dev/stdout \
572 /dev/stderr \
573 /etc/lib/ld.so.1 \
574 /etc/lib/libdl.so.1 \
575 /etc/lib/nss_files.so.1 \
576 /etc/log \
577 /lib/32 \
578 /lib/crypto/32 \
579 /lib/secure/32 \
580 /usr/adm \
581 /usr/spool \
582 /usr/lib/tmac \

```

```

583 /usr/ccs/lib/link_audit \
584 /usr/news \
585 /usr/preserve \
586 /usr/lib/32 \
587 /usr/lib/cron \
588 /usr/lib/elfedit/32 \
589 /usr/lib/libp/32 \
590 /usr/lib/lwp/32 \
591 /usr/lib/link_audit/32 \
592 /usr/lib/secure/32 \
593 /usr/mail \
594 /usr/man \
595 /usr/pub \
596 /usr/src \
597 /usr/tmp \
598 /usr/ucb/lib/32 \
599 /var/ld/32

601 sparc_SYM.DIRS64=

603 SYM.DIRS64= \
604 $( $(MACH)_SYM.DIRS64) \
605 /lib/64 \
606 /lib/crypto/64 \
607 /lib/secure/64 \
608 /usr/lib/64 \
609 /usr/lib/brand/sn1/64 \
610 /usr/lib/brand/solaris10/64 \
611 /usr/lib/elfedit/64 \
612 /usr/lib/libp/64 \
613 /usr/lib/link_audit/64 \
614 /usr/lib/lwp/64 \
615 /usr/lib/secure/64 \
616 /usr/lib/security/64 \
617 /usr/xpg4/lib/64 \
618 /var/ld/64 \
619 /usr/ucb/lib/64

621 # prepend the ROOT prefix

623 ROOTDIRS= $(TARGETDIRS:%=$(ROOT)%)

625 # conditional assignments
626 #
627 # Target directories with non-default values for owner and group must
628 # be referenced here, using their fully-prefixed names, and the non-
629 # default values assigned. If a directory is mentioned above and not
630 # mentioned below, it has default values for attributes.
631 #
632 # The default value for DIRMODE is specified in usr/src/Makefile.master.
633 #

635 $(ROOT)/var/adm \
636 $(ROOT)/var/adm/sa := DIRMODE= 775

638 $(ROOT)/var/spool/lp:= DIRMODE= 775

640 # file mode
641 #
642 $(ROOT)/tmp \
643 $(ROOT)/var/krb5/rcache \
644 $(ROOT)/var/preserve \
645 $(ROOT)/var/spool/pkg \
646 $(ROOT)/var/spool/uucppublic \
647 $(ROOT)/var/tmp:= DIRMODE= 1777

```

```

649 $(ROOT)/root:=          DIRMODE= 700

651 $(ROOT)/var/krb5/rcache/root:=  DIRMODE= 700

654 #
655 # These permissions must match those set
656 # in the package manifests.
657 #
658 $(ROOT)/var/sadm/pkg \
659 $(ROOT)/var/sadm/security \
660 $(ROOT)/var/sadm/install/logs :=      DIRMODE= 555

663 #
664 # These permissions must match the ones set
665 # internally by fdfs and autofs.
666 #
667 $(ROOT)/dev/fd \
668 $(ROOT)/home:=          DIRMODE= 555

670 $(ROOT)/var/mail:=      DIRMODE=1777

672 $(ROOT)/proc:=          DIRMODE= 555

674 $(ROOT)/system/contract:=      DIRMODE= 555
675 $(ROOT)/system/object:=        DIRMODE= 555

677 # symlink assignments, LINKDEST is the value of the symlink
678 #
679 $(ROOT)/usr/lib/cron:=          LINKDEST=../etc/cron.d
680 $(ROOT)/bin:=                  LINKDEST=usr/bin
681 $(ROOT)/lib/32:=              LINKDEST=.
682 $(ROOT)/lib/crypto/32:=       LINKDEST=.
683 $(ROOT)/lib/secure/32:=       LINKDEST=.
684 $(ROOT)/dev/stdin:=           LINKDEST=fd/0
685 $(ROOT)/dev/stdout:=          LINKDEST=fd/1
686 $(ROOT)/dev/stderr:=          LINKDEST=fd/2
687 $(ROOT)/usr/pub:=            LINKDEST=share/lib/pub
688 $(ROOT)/usr/man:=            LINKDEST=share/man
689 $(ROOT)/usr/src:=            LINKDEST=share/src
690 $(ROOT)/usr/adm:=            LINKDEST=../var/adm
691 $(ROOT)/etc/lib/ld.so.1:=      LINKDEST=../lib/ld.so.1
692 $(ROOT)/etc/lib/libdl.so.1:=  LINKDEST=../lib/libdl.so.1
693 $(ROOT)/etc/lib/nss_files.so.1:= LINKDEST=../lib/nss_files.so.1
694 $(ROOT)/etc/log:=            LINKDEST=../var/adm/log
695 $(ROOT)/usr/mail:=          LINKDEST=../var/mail
696 $(ROOT)/usr/news:=          LINKDEST=../var/news
697 $(ROOT)/usr/preserve:=       LINKDEST=../var/preserve
698 $(ROOT)/usr/spool:=          LINKDEST=../var/spool
699 $(ROOT)/usr/tmp:=            LINKDEST=../var/tmp
700 $(ROOT)/usr/lib/tmac:=        LINKDEST=../share/lib/tmac
701 $(ROOT)/usr/lib/32:=         LINKDEST=.
702 $(ROOT)/usr/lib/elfedit/32:=  LINKDEST=.
703 $(ROOT)/usr/lib/libp/32:=     LINKDEST=.
704 $(ROOT)/usr/lib/lwp/32:=      LINKDEST=.
705 $(ROOT)/usr/lib/link_audit/32:= LINKDEST=.
706 $(ROOT)/usr/lib/secure/32:=   LINKDEST=.
707 $(ROOT)/usr/ccs/lib/link_audit:= LINKDEST=../lib/link_audit
708 $(ROOT)/var/ld/32:=          LINKDEST=.
709 $(ROOT)/usr/ucblib/32:=       LINKDEST=.

712 $(BUILD64) $(ROOT)/lib/64:=   LINKDEST=$(MACH64)
713 $(BUILD64) $(ROOT)/lib/crypto/64:= LINKDEST=$(MACH64)
714 $(BUILD64) $(ROOT)/lib/secure/64:= LINKDEST=$(MACH64)

```

```

715 $(BUILD64) $(ROOT)/usr/lib/64:=   LINKDEST=$(MACH64)
716 $(BUILD64) $(ROOT)/usr/lib/elfedit/64:= LINKDEST=$(MACH64)
717 $(BUILD64) $(ROOT)/usr/lib/brand/snl/64:= LINKDEST=$(MACH64)
718 $(BUILD64) $(ROOT)/usr/lib/brand/solaris10/64:= LINKDEST=$(MACH64)
719 $(BUILD64) $(ROOT)/usr/lib/libp/64:= LINKDEST=$(MACH64)
720 $(BUILD64) $(ROOT)/usr/lib/lwp/64:= LINKDEST=$(MACH64)
721 $(BUILD64) $(ROOT)/usr/lib/link_audit/64:= LINKDEST=$(MACH64)
722 $(BUILD64) $(ROOT)/usr/lib/secure/64:= LINKDEST=$(MACH64)
723 $(BUILD64) $(ROOT)/usr/lib/security/64:= LINKDEST=$(MACH64)
724 $(BUILD64) $(ROOT)/usr/xpg4/lib/64:= LINKDEST=$(MACH64)
725 $(BUILD64) $(ROOT)/var/ld/64:=     LINKDEST=$(MACH64)
726 $(BUILD64) $(ROOT)/usr/ucblib/64:= LINKDEST=$(MACH64)

728 #
729 # Installing a directory symlink calls for overriding INS.dir to install
730 # a symlink.
731 #
732 $(DIRLINKS:%=$(ROOT)%):= \
733     INS.dir= -$(RM) -r $@; $(SYMLINK) $(LINKDEST) $@

735 # Special symlinks to populate usr/ccs/lib, whose objects
736 # have actually been moved to usr/lib
737 # Rather than adding another set of rules, we add usr/lib/lwp files here
738 $(ROOT)/usr/ccs/lib/libcurses.so:= REALPATH=../../../../lib/libcurses.so.1
739 $(ROOT)/usr/ccs/lib/llib-1curses:= REALPATH=../../../../lib/llib-1curses
740 $(ROOT)/usr/ccs/lib/llib-1curses.ln:= REALPATH=../../../../lib/llib-1curses.ln
741 $(ROOT)/usr/ccs/lib/libform.so:= REALPATH=../../../../lib/libform.so.1
742 $(ROOT)/usr/ccs/lib/llib-1form:= REALPATH=../../../../lib/llib-1form
743 $(ROOT)/usr/ccs/lib/llib-1form.ln:= REALPATH=../../../../lib/llib-1form.ln
744 $(ROOT)/usr/ccs/lib/libgen.so:= REALPATH=../../../../lib/libgen.so.1
745 $(ROOT)/usr/ccs/lib/llib-1gen:= REALPATH=../../../../lib/llib-1gen
746 $(ROOT)/usr/ccs/lib/llib-1gen.ln:= REALPATH=../../../../lib/llib-1gen.ln
747 $(ROOT)/usr/ccs/lib/libmalloc.so:= REALPATH=../../../../lib/libmalloc.so.1
748 $(ROOT)/usr/ccs/lib/libmenu.so:= REALPATH=../../../../lib/libmenu.so.1
749 $(ROOT)/usr/ccs/lib/llib-1menu:= REALPATH=../../../../lib/llib-1menu
750 $(ROOT)/usr/ccs/lib/llib-1menu.ln:= REALPATH=../../../../lib/llib-1menu.ln
751 $(ROOT)/usr/ccs/lib/libpanel.so:= REALPATH=../../../../lib/libpanel.so.1
752 $(ROOT)/usr/ccs/lib/llib-1panel:= REALPATH=../../../../lib/llib-1panel
753 $(ROOT)/usr/ccs/lib/llib-1panel.ln:= REALPATH=../../../../lib/llib-1panel.ln
754 $(ROOT)/usr/ccs/lib/libtermlib.so:= REALPATH=../../../../lib/libcurses.so.1
755 $(ROOT)/usr/ccs/lib/llib-1termlib:= REALPATH=../../../../lib/llib-1curses
756 $(ROOT)/usr/ccs/lib/llib-1termlib.ln:= REALPATH=../../../../lib/llib-1curses.ln
757 $(ROOT)/usr/ccs/lib/libtermcap.so:= REALPATH=../../../../lib/libtermcap.so.1
758 $(ROOT)/usr/ccs/lib/llib-1termcap:= REALPATH=../../../../lib/llib-1termcap
759 $(ROOT)/usr/ccs/lib/llib-1termcap.ln:= REALPATH=../../../../lib/llib-1termcap.ln
760 $(ROOT)/usr/ccs/lib/values-Xa.o:= REALPATH=../../../../lib/values-Xa.o
761 $(ROOT)/usr/ccs/lib/values-Xc.o:= REALPATH=../../../../lib/values-Xc.o
762 $(ROOT)/usr/ccs/lib/values-Xs.o:= REALPATH=../../../../lib/values-Xs.o
763 $(ROOT)/usr/ccs/lib/values-Xt.o:= REALPATH=../../../../lib/values-Xt.o
764 $(ROOT)/usr/ccs/lib/values-xpg4.o:= REALPATH=../../../../lib/values-xpg4.o
765 $(ROOT)/usr/ccs/lib/values-xpg6.o:= REALPATH=../../../../lib/values-xpg6.o
766 $(ROOT)/usr/ccs/lib/libl.so:= REALPATH=../../../../lib/libl.so.1
767 $(ROOT)/usr/ccs/lib/llib-1l.ln:= REALPATH=../../../../lib/llib-1l.ln
768 $(ROOT)/usr/ccs/lib/liby.so:= REALPATH=../../../../lib/liby.so.1
769 $(ROOT)/usr/ccs/lib/llib-1y.ln:= REALPATH=../../../../lib/llib-1y.ln
770 $(ROOT)/usr/lib/libp/libc.so.1:= REALPATH=../../../../lib/libc.so.1
771 $(ROOT)/usr/lib/lwp/libthread.so.1:= REALPATH=../../../../lib/libc.so.1
772 $(ROOT)/usr/lib/lwp/libthread_db.so.1:= REALPATH=../libthread_db.so.1

774 # symlinks to populate usr/ccs/lib/$(MACH64)
775 $(ROOT)/usr/ccs/lib/$(MACH64)/libcurses.so:= \
776     REALPATH=../../../../lib/$(MACH64)/libcurses.so.1
777 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-1curses.ln:= \
778     REALPATH=../../../../lib/$(MACH64)/llib-1curses.ln
779 $(ROOT)/usr/ccs/lib/$(MACH64)/libform.so:= \
780     REALPATH=../../../../lib/$(MACH64)/libform.so.1

```

```

781 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-lform.ln:= \
782   REALPATH=../../../../lib/$(MACH64)/llib-lform.ln
783 $(ROOT)/usr/ccs/lib/$(MACH64)/libgen.so:= \
784   REALPATH=../../../../lib/$(MACH64)/libgen.so.1
785 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-lgen.ln:= \
786   REALPATH=../../../../lib/$(MACH64)/llib-lgen.ln
787 $(ROOT)/usr/ccs/lib/$(MACH64)/libmalloc.so:= \
788   REALPATH=../../../../lib/$(MACH64)/libmalloc.so.1
789 $(ROOT)/usr/ccs/lib/$(MACH64)/libmenu.so:= \
790   REALPATH=../../../../lib/$(MACH64)/libmenu.so.1
791 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-lmenu.ln:= \
792   REALPATH=../../../../lib/$(MACH64)/llib-lmenu.ln
793 $(ROOT)/usr/ccs/lib/$(MACH64)/libpanel.so:= \
794   REALPATH=../../../../lib/$(MACH64)/libpanel.so.1
795 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-lpanel.ln:= \
796   REALPATH=../../../../lib/$(MACH64)/llib-lpanel.ln
797 $(ROOT)/usr/ccs/lib/$(MACH64)/libtermlib.so:= \
798   REALPATH=../../../../lib/$(MACH64)/libcurses.so.1
799 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-ltermlib.ln:= \
800   REALPATH=../../../../lib/$(MACH64)/llib-lcurses.ln
801 $(ROOT)/usr/ccs/lib/$(MACH64)/libtermcap.so:= \
802   REALPATH=../../../../lib/$(MACH64)/libtermcap.so.1
803 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-ltermcap.ln:= \
804   REALPATH=../../../../lib/$(MACH64)/llib-ltermcap.ln
805 $(ROOT)/usr/ccs/lib/$(MACH64)/values-Xa.o:= \
806   REALPATH=../../../../lib/$(MACH64)/values-Xa.o
807 $(ROOT)/usr/ccs/lib/$(MACH64)/values-Xc.o:= \
808   REALPATH=../../../../lib/$(MACH64)/values-Xc.o
809 $(ROOT)/usr/ccs/lib/$(MACH64)/values-Xs.o:= \
810   REALPATH=../../../../lib/$(MACH64)/values-Xs.o
811 $(ROOT)/usr/ccs/lib/$(MACH64)/values-Xt.o:= \
812   REALPATH=../../../../lib/$(MACH64)/values-Xt.o
813 $(ROOT)/usr/ccs/lib/$(MACH64)/values-xpg4.o:= \
814   REALPATH=../../../../lib/$(MACH64)/values-xpg4.o
815 $(ROOT)/usr/ccs/lib/$(MACH64)/values-xpg6.o:= \
816   REALPATH=../../../../lib/$(MACH64)/values-xpg6.o
817 $(ROOT)/usr/ccs/lib/$(MACH64)/libl.so:= \
818   REALPATH=../../../../lib/$(MACH64)/libl.so.1
819 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-ll.ln:= \
820   REALPATH=../../../../lib/$(MACH64)/llib-ll.ln
821 $(ROOT)/usr/ccs/lib/$(MACH64)/liby.so:= \
822   REALPATH=../../../../lib/$(MACH64)/liby.so.1
823 $(ROOT)/usr/ccs/lib/$(MACH64)/llib-ly.ln:= \
824   REALPATH=../../../../lib/$(MACH64)/llib-ly.ln
825 $(ROOT)/usr/lib/libp/$(MACH64)/libc.so.1:= \
826   REALPATH=../../../../lib/$(MACH64)/libc.so.1
827 $(ROOT)/usr/lib/lwp/$(MACH64)/libthread.so.1:= \
828   REALPATH=../../../../$(MACH64)/libthread.so.1
829 $(ROOT)/usr/lib/lwp/$(MACH64)/libthread_db.so.1:= \
830   REALPATH=../../../../$(MACH64)/libthread_db.so.1

832 SYM.USRCCSLIB= \
833   /usr/ccs/lib/libcurses.so \
834   /usr/ccs/lib/llib-lcurses \
835   /usr/ccs/lib/llib-lcurses.ln \
836   /usr/ccs/lib/libform.so \
837   /usr/ccs/lib/llib-lform \
838   /usr/ccs/lib/llib-lform.ln \
839   /usr/ccs/lib/libgen.so \
840   /usr/ccs/lib/llib-lgen \
841   /usr/ccs/lib/llib-lgen.ln \
842   /usr/ccs/lib/libmalloc.so \
843   /usr/ccs/lib/libmenu.so \
844   /usr/ccs/lib/llib-lmenu \
845   /usr/ccs/lib/llib-lmenu.ln \
846   /usr/ccs/lib/libpanel.so \

```

```

847   /usr/ccs/lib/llib-lpanel \
848   /usr/ccs/lib/llib-lpanel.ln \
849   /usr/ccs/lib/libtermlib.so \
850   /usr/ccs/lib/llib-ltermlib \
851   /usr/ccs/lib/llib-ltermlib.ln \
852   /usr/ccs/lib/libtermcap.so \
853   /usr/ccs/lib/llib-ltermcap \
854   /usr/ccs/lib/llib-ltermcap.ln \
855   /usr/ccs/lib/values-Xa.o \
856   /usr/ccs/lib/values-Xc.o \
857   /usr/ccs/lib/values-Xs.o \
858   /usr/ccs/lib/values-Xt.o \
859   /usr/ccs/lib/values-xpg4.o \
860   /usr/ccs/lib/values-xpg6.o \
861   /usr/ccs/lib/libl.so \
862   /usr/ccs/lib/llib-ll.ln \
863   /usr/ccs/lib/liby.so \
864   /usr/ccs/lib/llib-ly.ln \
865   /usr/lib/libp/libc.so.1 \
866   /usr/lib/lwp/libthread.so.1 \
867   /usr/lib/lwp/libthread_db.so.1

869 SYM.USRCCSLIB64= \
870   /usr/ccs/lib/$(MACH64)/libcurses.so \
871   /usr/ccs/lib/$(MACH64)/llib-lcurses.ln \
872   /usr/ccs/lib/$(MACH64)/libform.so \
873   /usr/ccs/lib/$(MACH64)/llib-lform.ln \
874   /usr/ccs/lib/$(MACH64)/libgen.so \
875   /usr/ccs/lib/$(MACH64)/llib-lgen.ln \
876   /usr/ccs/lib/$(MACH64)/libmalloc.so \
877   /usr/ccs/lib/$(MACH64)/libmenu.so \
878   /usr/ccs/lib/$(MACH64)/llib-lmenu.ln \
879   /usr/ccs/lib/$(MACH64)/libpanel.so \
880   /usr/ccs/lib/$(MACH64)/llib-lpanel.ln \
881   /usr/ccs/lib/$(MACH64)/libtermlib.so \
882   /usr/ccs/lib/$(MACH64)/llib-ltermlib.ln \
883   /usr/ccs/lib/$(MACH64)/libtermcap.so \
884   /usr/ccs/lib/$(MACH64)/llib-ltermcap.ln \
885   /usr/ccs/lib/$(MACH64)/values-Xa.o \
886   /usr/ccs/lib/$(MACH64)/values-Xc.o \
887   /usr/ccs/lib/$(MACH64)/values-Xs.o \
888   /usr/ccs/lib/$(MACH64)/values-Xt.o \
889   /usr/ccs/lib/$(MACH64)/values-xpg4.o \
890   /usr/ccs/lib/$(MACH64)/values-xpg6.o \
891   /usr/ccs/lib/$(MACH64)/libl.so \
892   /usr/ccs/lib/$(MACH64)/llib-ll.ln \
893   /usr/ccs/lib/$(MACH64)/liby.so \
894   /usr/ccs/lib/$(MACH64)/llib-ly.ln \
895   /usr/lib/libp/$(MACH64)/libc.so.1 \
896   /usr/lib/lwp/$(MACH64)/libthread.so.1 \
897   /usr/lib/lwp/$(MACH64)/libthread_db.so.1

899 # Special symlinks to direct libraries that have been moved
900 # from /usr/lib to /lib in order to live in the root filesystem.
901 $(ROOT)/lib/libposix4.so.1:= REALPATH=librt.so.1
902 $(ROOT)/lib/libposix4.so:= REALPATH=libposix4.so.1
903 $(ROOT)/lib/llib-lposix4:= REALPATH=llib-lrt
904 $(ROOT)/lib/llib-lposix4.ln:= REALPATH=llib-lrt.ln
905 $(ROOT)/lib/libthread_db.so.1:= REALPATH=libc_db.so.1
906 $(ROOT)/lib/libthread_db.so:= REALPATH=libc_db.so.1
907 $(ROOT)/usr/lib/ld.so.1:= REALPATH=../../../../lib/ld.so.1
908 $(ROOT)/usr/lib/libadm.so.1:= REALPATH=../../../../lib/libadm.so.1
909 $(ROOT)/usr/lib/libadm.so:= REALPATH=../../../../lib/libadm.so.1
910 $(ROOT)/usr/lib/libaio.so.1:= REALPATH=../../../../lib/libaio.so.1
911 $(ROOT)/usr/lib/libaio.so:= REALPATH=../../../../lib/libaio.so.1
912 $(ROOT)/usr/lib/libavl.so.1:= REALPATH=../../../../lib/libavl.so.1

```

```

913 $(ROOT)/usr/lib/libavl.so:= REALPATH=../lib/libavl.so.1
914 $(ROOT)/usr/lib/libbasm.so.1:= REALPATH=../lib/libbasm.so.1
915 $(ROOT)/usr/lib/libbasm.so:= REALPATH=../lib/libbasm.so.1
916 $(ROOT)/usr/lib/libbc.so.1:= REALPATH=../lib/libbc.so.1
917 $(ROOT)/usr/lib/libc.so:= REALPATH=../lib/libc.so.1
918 $(ROOT)/usr/lib/libc_db.so.1:= REALPATH=../lib/libc_db.so.1
919 $(ROOT)/usr/lib/libc_db.so:= REALPATH=../lib/libc_db.so.1
920 $(ROOT)/usr/lib/libcmdutils.so.1:= REALPATH=../lib/libcmdutils.so.1
921 $(ROOT)/usr/lib/libcmdutils.so:= REALPATH=../lib/libcmdutils.so.1
922 $(ROOT)/usr/lib/libcontract.so.1:= REALPATH=../lib/libcontract.so.1
923 $(ROOT)/usr/lib/libcontract.so:= REALPATH=../lib/libcontract.so.1
924 $(ROOT)/usr/lib/libcryptoutil.so.1:= REALPATH=../lib/libcryptoutil.so.1
925 $(ROOT)/usr/lib/libcryptoutil.so:= REALPATH=../lib/libcryptoutil.so.1
926 $(ROOT)/usr/lib/libctf.so.1:= REALPATH=../lib/libctf.so.1
927 $(ROOT)/usr/lib/libctf.so:= REALPATH=../lib/libctf.so.1
928 $(ROOT)/usr/lib/libcurses.so.1:= REALPATH=../lib/libcurses.so.1
929 $(ROOT)/usr/lib/libcurses.so:= REALPATH=../lib/libcurses.so.1
930 $(ROOT)/usr/lib/libdevice.so.1:= REALPATH=../lib/libdevice.so.1
931 $(ROOT)/usr/lib/libdevice.so:= REALPATH=../lib/libdevice.so.1
932 $(ROOT)/usr/lib/libdevvid.so.1:= REALPATH=../lib/libdevvid.so.1
933 $(ROOT)/usr/lib/libdevvid.so:= REALPATH=../lib/libdevvid.so.1
934 $(ROOT)/usr/lib/libdevinfo.so.1:= REALPATH=../lib/libdevinfo.so.1
935 $(ROOT)/usr/lib/libdevinfo.so:= REALPATH=../lib/libdevinfo.so.1
936 $(ROOT)/usr/lib/libdhcpcagent.so.1:= REALPATH=../lib/libdhcpcagent.so.1
937 $(ROOT)/usr/lib/libdhcpcagent.so:= REALPATH=../lib/libdhcpcagent.so.1
938 $(ROOT)/usr/lib/libdhcputil.so.1:= REALPATH=../lib/libdhcputil.so.1
939 $(ROOT)/usr/lib/libdhcputil.so:= REALPATH=../lib/libdhcputil.so.1
940 $(ROOT)/usr/lib/libddl.so.1:= REALPATH=../lib/libddl.so.1
941 $(ROOT)/usr/lib/libddl.so:= REALPATH=../lib/libddl.so.1
942 $(ROOT)/usr/lib/libdmpi.so.1:= REALPATH=../lib/libdmpi.so.1
943 $(ROOT)/usr/lib/libdmpi.so:= REALPATH=../lib/libdmpi.so.1
944 $(ROOT)/usr/lib/libdoor.so.1:= REALPATH=../lib/libdoor.so.1
945 $(ROOT)/usr/lib/libdoor.so:= REALPATH=../lib/libdoor.so.1
946 $(ROOT)/usr/lib/libefi.so.1:= REALPATH=../lib/libefi.so.1
947 $(ROOT)/usr/lib/libefi.so:= REALPATH=../lib/libefi.so.1
948 $(ROOT)/usr/lib/libelf.so.1:= REALPATH=../lib/libelf.so.1
949 $(ROOT)/usr/lib/libelf.so:= REALPATH=../lib/libelf.so.1
950 $(ROOT)/usr/lib/libfdisk.so.1:= REALPATH=../lib/libfdisk.so.1
951 $(ROOT)/usr/lib/libfdisk.so:= REALPATH=../lib/libfdisk.so.1
952 $(ROOT)/usr/lib/libgen.so.1:= REALPATH=../lib/libgen.so.1
953 $(ROOT)/usr/lib/libgen.so:= REALPATH=../lib/libgen.so.1
954 $(ROOT)/usr/lib/libinetutil.so.1:= REALPATH=../lib/libinetutil.so.1
955 $(ROOT)/usr/lib/libinetutil.so:= REALPATH=../lib/libinetutil.so.1
956 $(ROOT)/usr/lib/libintl.so.1:= REALPATH=../lib/libintl.so.1
957 $(ROOT)/usr/lib/libintl.so:= REALPATH=../lib/libintl.so.1
958 $(ROOT)/usr/lib/libkmf.so.1:= REALPATH=../lib/libkmf.so.1
959 $(ROOT)/usr/lib/libkmf.so:= REALPATH=../lib/libkmf.so.1
960 $(ROOT)/usr/lib/libkmfberder.so.1:= REALPATH=../lib/libkmfberder.so.1
961 $(ROOT)/usr/lib/libkmfberder.so:= REALPATH=../lib/libkmfberder.so.1
962 $(ROOT)/usr/lib/libkstat.so.1:= REALPATH=../lib/libkstat.so.1
963 $(ROOT)/usr/lib/libkstat.so:= REALPATH=../lib/libkstat.so.1
964 $(ROOT)/usr/lib/liblddbg.so.4:= REALPATH=../lib/liblddbg.so.4
965 $(ROOT)/usr/lib/libmd.so.1:= REALPATH=../lib/libmd.so.1
966 $(ROOT)/usr/lib/libmd.so:= REALPATH=../lib/libmd.so.1
967 $(ROOT)/usr/lib/libmd5.so.1:= REALPATH=../lib/libmd5.so.1
968 $(ROOT)/usr/lib/libmd5.so:= REALPATH=../lib/libmd5.so.1
969 $(ROOT)/usr/lib/libmeta.so.1:= REALPATH=../lib/libmeta.so.1
970 $(ROOT)/usr/lib/libmeta.so:= REALPATH=../lib/libmeta.so.1
971 $(ROOT)/usr/lib/libmp.so.1:= REALPATH=../lib/libmp.so.1
972 $(ROOT)/usr/lib/libmp.so.2:= REALPATH=../lib/libmp.so.2
973 $(ROOT)/usr/lib/libmp.so:= REALPATH=../lib/libmp.so.2
974 $(ROOT)/usr/lib/libnsl.so.1:= REALPATH=../lib/libnsl.so.1
975 $(ROOT)/usr/lib/libnsl.so:= REALPATH=../lib/libnsl.so.1
976 $(ROOT)/usr/lib/libnvpair.so.1:= REALPATH=../lib/libnvpair.so.1
977 $(ROOT)/usr/lib/libnvpair.so:= REALPATH=../lib/libnvpair.so.1
978 $(ROOT)/usr/lib/libpam.so.1:= REALPATH=../lib/libpam.so.1

```

```

979 $(ROOT)/usr/lib/libpam.so:= REALPATH=../lib/libpam.so.1
980 $(ROOT)/usr/lib/libposix4.so.1:= REALPATH=../lib/librt.so.1
981 $(ROOT)/usr/lib/libposix4.so:= REALPATH=../lib/librt.so.1
982 $(ROOT)/usr/lib/libproc.so.1:= REALPATH=../lib/libproc.so.1
983 $(ROOT)/usr/lib/libproc.so:= REALPATH=../lib/libproc.so.1
984 $(ROOT)/usr/lib/libpthread.so.1:= REALPATH=../lib/libpthread.so.1
985 $(ROOT)/usr/lib/libpthread.so:= REALPATH=../lib/libpthread.so.1
986 $(ROOT)/usr/lib/librcm.so.1:= REALPATH=../lib/librcm.so.1
987 $(ROOT)/usr/lib/librcm.so:= REALPATH=../lib/librcm.so.1
988 $(ROOT)/usr/lib/libresolv.so.1:= REALPATH=../lib/libresolv.so.1
989 $(ROOT)/usr/lib/libresolv.so.2:= REALPATH=../lib/libresolv.so.2
990 $(ROOT)/usr/lib/libresolv.so:= REALPATH=../lib/libresolv.so.2
991 $(ROOT)/usr/lib/librestart.so.1:= REALPATH=../lib/librestart.so.1
992 $(ROOT)/usr/lib/librestart.so:= REALPATH=../lib/librestart.so.1
993 $(ROOT)/usr/lib/librpcsvc.so.1:= REALPATH=../lib/librpcsvc.so.1
994 $(ROOT)/usr/lib/librpcsvc.so:= REALPATH=../lib/librpcsvc.so.1
995 $(ROOT)/usr/lib/librt.so.1:= REALPATH=../lib/librt.so.1
996 $(ROOT)/usr/lib/librt.so:= REALPATH=../lib/librt.so.1
997 $(ROOT)/usr/lib/librtld.so.1:= REALPATH=../lib/librtld.so.1
998 $(ROOT)/usr/lib/librtld_db.so.1:= REALPATH=../lib/librtld_db.so.1
999 $(ROOT)/usr/lib/librtld_db.so:= REALPATH=../lib/librtld_db.so.1
1000 $(ROOT)/usr/lib/libscf.so.1:= REALPATH=../lib/libscf.so.1
1001 $(ROOT)/usr/lib/libscf.so:= REALPATH=../lib/libscf.so.1
1002 $(ROOT)/usr/lib/libsec.so.1:= REALPATH=../lib/libsec.so.1
1003 $(ROOT)/usr/lib/libsec.so:= REALPATH=../lib/libsec.so.1
1004 $(ROOT)/usr/lib/libsecdb.so.1:= REALPATH=../lib/libsecdb.so.1
1005 $(ROOT)/usr/lib/libsecdb.so:= REALPATH=../lib/libsecdb.so.1
1006 $(ROOT)/usr/lib/libsendfile.so.1:= REALPATH=../lib/libsendfile.so.1
1007 $(ROOT)/usr/lib/libsendfile.so:= REALPATH=../lib/libsendfile.so.1
1008 $(ROOT)/usr/lib/libsocket.so.1:= REALPATH=../lib/libsocket.so.1
1009 $(ROOT)/usr/lib/libsocket.so:= REALPATH=../lib/libsocket.so.1
1010 $(ROOT)/usr/lib/libsysevent.so.1:= REALPATH=../lib/libsysevent.so.1
1011 $(ROOT)/usr/lib/libsysevent.so:= REALPATH=../lib/libsysevent.so.1
1012 $(ROOT)/usr/lib/libtermcap.so.1:= REALPATH=../lib/libtermcap.so.1
1013 $(ROOT)/usr/lib/libtermcap.so:= REALPATH=../lib/libtermcap.so.1
1014 $(ROOT)/usr/lib/libtermlib.so.1:= REALPATH=../lib/libcurses.so.1
1015 $(ROOT)/usr/lib/libtermlib.so:= REALPATH=../lib/libcurses.so.1
1016 $(ROOT)/usr/lib/libthread.so.1:= REALPATH=../lib/libthread.so.1
1017 $(ROOT)/usr/lib/libthread.so:= REALPATH=../lib/libthread.so.1
1018 $(ROOT)/usr/lib/libthread_db.so.1:= REALPATH=../lib/libc_db.so.1
1019 $(ROOT)/usr/lib/libthread_db.so:= REALPATH=../lib/libc_db.so.1
1020 $(ROOT)/usr/lib/libtsnet.so.1:= REALPATH=../lib/libtsnet.so.1
1021 $(ROOT)/usr/lib/libtsnet.so:= REALPATH=../lib/libtsnet.so.1
1022 $(ROOT)/usr/lib/libtsol.so.2:= REALPATH=../lib/libtsol.so.2
1023 $(ROOT)/usr/lib/libtsol.so:= REALPATH=../lib/libtsol.so.2
1024 $(ROOT)/usr/lib/libumem.so.1:= REALPATH=../lib/libumem.so.1
1025 $(ROOT)/usr/lib/libumem.so:= REALPATH=../lib/libumem.so.1
1026 $(ROOT)/usr/lib/libuuid.so.1:= REALPATH=../lib/libuuid.so.1
1027 $(ROOT)/usr/lib/libuuid.so:= REALPATH=../lib/libuuid.so.1
1028 $(ROOT)/usr/lib/libuutil.so.1:= REALPATH=../lib/libuutil.so.1
1029 $(ROOT)/usr/lib/libuutil.so:= REALPATH=../lib/libuutil.so.1
1030 $(ROOT)/usr/lib/libw.so.1:= REALPATH=../lib/libw.so.1
1031 $(ROOT)/usr/lib/libw.so:= REALPATH=../lib/libw.so.1
1032 $(ROOT)/usr/lib/libxnet.so.1:= REALPATH=../lib/libxnet.so.1
1033 $(ROOT)/usr/lib/libxnet.so:= REALPATH=../lib/libxnet.so.1
1034 $(ROOT)/usr/lib/libzfs.so.1:= REALPATH=../lib/libzfs.so.1
1035 $(ROOT)/usr/lib/libzfs.so:= REALPATH=../lib/libzfs.so.1
1036 $(ROOT)/usr/lib/libzfs_core.so.1:= REALPATH=../lib/libzfs_core.so.1
1037 $(ROOT)/usr/lib/libzfs_core.so:= REALPATH=../lib/libzfs_core.so.1
1038 #endif /* ! codereview */
1039 $(ROOT)/usr/lib/lib-ladm.ln:= REALPATH=../lib/lib-ladm.ln
1040 $(ROOT)/usr/lib/lib-ladm:= REALPATH=../lib/lib-ladm
1041 $(ROOT)/usr/lib/lib-laic.ln:= REALPATH=../lib/lib-laic.ln
1042 $(ROOT)/usr/lib/lib-laic:= REALPATH=../lib/lib-laic
1043 $(ROOT)/usr/lib/lib-lavl.ln:= REALPATH=../lib/lib-lavl.ln
1044 $(ROOT)/usr/lib/lib-lavl:= REALPATH=../lib/lib-lavl

```



```

1045 $(ROOT)/usr/lib/lib-lbasm.ln:= REALPATH=../lib/lib-lbasm.ln
1046 $(ROOT)/usr/lib/lib-lbasm:= REALPATH=../lib/lib-lbasm
1047 $(ROOT)/usr/lib/lib-lc.ln:= REALPATH=../lib/lib-lc.ln
1048 $(ROOT)/usr/lib/lib-lc:= REALPATH=../lib/lib-lc
1049 $(ROOT)/usr/lib/lib-lcmdutils.ln:= REALPATH=../lib/lib-lcmdutils.ln
1050 $(ROOT)/usr/lib/lib-lcmdutils:= REALPATH=../lib/lib-lcmdutils
1051 $(ROOT)/usr/lib/lib-lcontract.ln:= REALPATH=../lib/lib-lcontract.ln
1052 $(ROOT)/usr/lib/lib-lcontract:= REALPATH=../lib/lib-lcontract
1053 $(ROOT)/usr/lib/lib-lctf.ln:= REALPATH=../lib/lib-lctf.ln
1054 $(ROOT)/usr/lib/lib-lctf:= REALPATH=../lib/lib-lctf
1055 $(ROOT)/usr/lib/lib-lcurses.ln:= REALPATH=../lib/lib-lcurses.ln
1056 $(ROOT)/usr/lib/lib-lcurses:= REALPATH=../lib/lib-lcurses
1057 $(ROOT)/usr/lib/lib-ldevice.ln:= REALPATH=../lib/lib-ldevice.ln
1058 $(ROOT)/usr/lib/lib-ldevice:= REALPATH=../lib/lib-ldevice
1059 $(ROOT)/usr/lib/lib-ldevice.ln:= REALPATH=../lib/lib-ldevice.ln
1060 $(ROOT)/usr/lib/lib-ldevice:= REALPATH=../lib/lib-ldevice
1061 $(ROOT)/usr/lib/lib-ldevinfo.ln:= REALPATH=../lib/lib-ldevinfo.ln
1062 $(ROOT)/usr/lib/lib-ldevinfo:= REALPATH=../lib/lib-ldevinfo
1063 $(ROOT)/usr/lib/lib-ldhcpagent.ln:= REALPATH=../lib/lib-ldhcpagent.ln
1064 $(ROOT)/usr/lib/lib-ldhcpagent:= REALPATH=../lib/lib-ldhcpagent
1065 $(ROOT)/usr/lib/lib-ldhcputil.ln:= REALPATH=../lib/lib-ldhcputil.ln
1066 $(ROOT)/usr/lib/lib-ldhcputil:= REALPATH=../lib/lib-ldhcputil
1067 $(ROOT)/usr/lib/lib-ldl.ln:= REALPATH=../lib/lib-ldl.ln
1068 $(ROOT)/usr/lib/lib-ldl:= REALPATH=../lib/lib-ldl
1069 $(ROOT)/usr/lib/lib-ldoor.ln:= REALPATH=../lib/lib-ldoor.ln
1070 $(ROOT)/usr/lib/lib-ldoor:= REALPATH=../lib/lib-ldoor
1071 $(ROOT)/usr/lib/lib-lefi.ln:= REALPATH=../lib/lib-lefi.ln
1072 $(ROOT)/usr/lib/lib-lefi:= REALPATH=../lib/lib-lefi
1073 $(ROOT)/usr/lib/lib-lelf.ln:= REALPATH=../lib/lib-lelf.ln
1074 $(ROOT)/usr/lib/lib-lelf:= REALPATH=../lib/lib-lelf
1075 $(ROOT)/usr/lib/lib-lfdisk.ln:= REALPATH=../lib/lib-lfdisk.ln
1076 $(ROOT)/usr/lib/lib-lfdisk:= REALPATH=../lib/lib-lfdisk
1077 $(ROOT)/usr/lib/lib-lgen.ln:= REALPATH=../lib/lib-lgen.ln
1078 $(ROOT)/usr/lib/lib-lgen:= REALPATH=../lib/lib-lgen
1079 $(ROOT)/usr/lib/lib-linetutil.ln:= REALPATH=../lib/lib-linetutil.ln
1080 $(ROOT)/usr/lib/lib-linetutil:= REALPATH=../lib/lib-linetutil
1081 $(ROOT)/usr/lib/lib-lintl.ln:= REALPATH=../lib/lib-lintl.ln
1082 $(ROOT)/usr/lib/lib-lintl:= REALPATH=../lib/lib-lintl
1083 $(ROOT)/usr/lib/lib-lkstat.ln:= REALPATH=../lib/lib-lkstat.ln
1084 $(ROOT)/usr/lib/lib-lkstat:= REALPATH=../lib/lib-lkstat
1085 $(ROOT)/usr/lib/lib-lmd5.ln:= REALPATH=../lib/lib-lmd5.ln
1086 $(ROOT)/usr/lib/lib-lmd5:= REALPATH=../lib/lib-lmd5
1087 $(ROOT)/usr/lib/lib-lmeta.ln:= REALPATH=../lib/lib-lmeta.ln
1088 $(ROOT)/usr/lib/lib-lmeta:= REALPATH=../lib/lib-lmeta
1089 $(ROOT)/usr/lib/lib-lns1.ln:= REALPATH=../lib/lib-lns1.ln
1090 $(ROOT)/usr/lib/lib-lns1:= REALPATH=../lib/lib-lns1
1091 $(ROOT)/usr/lib/lib-lnvpair.ln:= REALPATH=../lib/lib-lnvpair.ln
1092 $(ROOT)/usr/lib/lib-lnvpair:= REALPATH=../lib/lib-lnvpair
1093 $(ROOT)/usr/lib/lib-lpam.ln:= REALPATH=../lib/lib-lpam.ln
1094 $(ROOT)/usr/lib/lib-lpam:= REALPATH=../lib/lib-lpam
1095 $(ROOT)/usr/lib/lib-lposix4.ln:= REALPATH=../lib/lib-lrt.ln
1096 $(ROOT)/usr/lib/lib-lposix4:= REALPATH=../lib/lib-lrt
1097 $(ROOT)/usr/lib/lib-lpthread.ln:= REALPATH=../lib/lib-lpthread.ln
1098 $(ROOT)/usr/lib/lib-lpthread:= REALPATH=../lib/lib-lpthread
1099 $(ROOT)/usr/lib/lib-lresolv.ln:= REALPATH=../lib/lib-lresolv.ln
1100 $(ROOT)/usr/lib/lib-lresolv:= REALPATH=../lib/lib-lresolv
1101 $(ROOT)/usr/lib/lib-lrpcsvc.ln:= REALPATH=../lib/lib-lrpcsvc.ln
1102 $(ROOT)/usr/lib/lib-lrpcsvc:= REALPATH=../lib/lib-lrpcsvc
1103 $(ROOT)/usr/lib/lib-lrt.ln:= REALPATH=../lib/lib-lrt.ln
1104 $(ROOT)/usr/lib/lib-lrt:= REALPATH=../lib/lib-lrt
1105 $(ROOT)/usr/lib/lib-lrtld_db.ln:= REALPATH=../lib/lib-lrtld_db.ln
1106 $(ROOT)/usr/lib/lib-lrtld_db:= REALPATH=../lib/lib-lrtld_db
1107 $(ROOT)/usr/lib/lib-lscf.ln:= REALPATH=../lib/lib-lscf.ln
1108 $(ROOT)/usr/lib/lib-lscf:= REALPATH=../lib/lib-lscf
1109 $(ROOT)/usr/lib/lib-lsec.ln:= REALPATH=../lib/lib-lsec.ln
1110 $(ROOT)/usr/lib/lib-lsec:= REALPATH=../lib/lib-lsec

```

```

1111 $(ROOT)/usr/lib/lib-lsecdb.ln:= REALPATH=../lib/lib-lsecdb.ln
1112 $(ROOT)/usr/lib/lib-lsecdb:= REALPATH=../lib/lib-lsecdb
1113 $(ROOT)/usr/lib/lib-lsendfile.ln:= REALPATH=../lib/lib-lsendfile.ln
1114 $(ROOT)/usr/lib/lib-lsendfile:= REALPATH=../lib/lib-lsendfile
1115 $(ROOT)/usr/lib/lib-lsocket.ln:= REALPATH=../lib/lib-lsocket.ln
1116 $(ROOT)/usr/lib/lib-lsocket:= REALPATH=../lib/lib-lsocket
1117 $(ROOT)/usr/lib/lib-lsysevent.ln:= REALPATH=../lib/lib-lsysevent.ln
1118 $(ROOT)/usr/lib/lib-lsysevent:= REALPATH=../lib/lib-lsysevent
1119 $(ROOT)/usr/lib/lib-ltermcap.ln:= REALPATH=../lib/lib-ltermcap.ln
1120 $(ROOT)/usr/lib/lib-ltermcap:= REALPATH=../lib/lib-ltermcap
1121 $(ROOT)/usr/lib/lib-ltermplib.ln:= REALPATH=../lib/lib-lcurses.ln
1122 $(ROOT)/usr/lib/lib-ltermplib:= REALPATH=../lib/lib-lcurses
1123 $(ROOT)/usr/lib/lib-lthread.ln:= REALPATH=../lib/lib-lthread.ln
1124 $(ROOT)/usr/lib/lib-lthread:= REALPATH=../lib/lib-lthread
1125 $(ROOT)/usr/lib/lib-lthread_db.ln:= REALPATH=../lib/lib-lc_db.ln
1126 $(ROOT)/usr/lib/lib-lthread_db:= REALPATH=../lib/lib-lc_db
1127 $(ROOT)/usr/lib/lib-ltsnet.ln:= REALPATH=../lib/lib-ltsnet.ln
1128 $(ROOT)/usr/lib/lib-ltsnet:= REALPATH=../lib/lib-ltsnet
1129 $(ROOT)/usr/lib/lib-ltsol.ln:= REALPATH=../lib/lib-ltsol.ln
1130 $(ROOT)/usr/lib/lib-ltsol:= REALPATH=../lib/lib-ltsol
1131 $(ROOT)/usr/lib/lib-lumem.ln:= REALPATH=../lib/lib-lumem.ln
1132 $(ROOT)/usr/lib/lib-lumem:= REALPATH=../lib/lib-lumem
1133 $(ROOT)/usr/lib/lib-luuid.ln:= REALPATH=../lib/lib-luuid.ln
1134 $(ROOT)/usr/lib/lib-luuid:= REALPATH=../lib/lib-luuid
1135 $(ROOT)/usr/lib/lib-lxnet.ln:= REALPATH=../lib/lib-lxnet.ln
1136 $(ROOT)/usr/lib/lib-lxnet:= REALPATH=../lib/lib-lxnet
1137 $(ROOT)/usr/lib/lib-lzfs.ln:= REALPATH=../lib/lib-lzfs.ln
1138 $(ROOT)/usr/lib/lib-lzfs:= REALPATH=../lib/lib-lzfs
1139 $(ROOT)/usr/lib/lib-lzfs_core.ln:= REALPATH=../lib/lib-lzfs_core.ln
1140 $(ROOT)/usr/lib/lib-lzfs_core:= REALPATH=../lib/lib-lzfs_core
1141 #endif /* ! codereview */
1142 $(ROOT)/usr/lib/nss_compat.so.1:= REALPATH=../lib/nss_compat.so.1
1143 $(ROOT)/usr/lib/nss_dns.so.1:= REALPATH=../lib/nss_dns.so.1
1144 $(ROOT)/usr/lib/nss_files.so.1:= REALPATH=../lib/nss_files.so.1
1145 $(ROOT)/usr/lib/nss_nis.so.1:= REALPATH=../lib/nss_nis.so.1
1146 $(ROOT)/usr/lib/nss_user.so.1:= REALPATH=../lib/nss_user.so.1
1147 $(ROOT)/usr/lib/fm/libfmevent.so.1:= REALPATH=../lib/fm/libfmevent.so.1
1148 $(ROOT)/usr/lib/fm/libfmevent.so:= REALPATH=../lib/fm/libfmevent.so.1
1149 $(ROOT)/usr/lib/fm/liblfmevent.ln:= REALPATH=../lib/fm/liblfmevent.ln
1150 $(ROOT)/usr/lib/fm/liblfmevent:= REALPATH=../lib/fm/liblfmevent

1152 $(ROOT)/lib/$(MACH64)/libposix4.so.1:= \
1153 REALPATH=librt.so.1
1154 $(ROOT)/lib/$(MACH64)/libposix4.so:= \
1155 REALPATH=libposix4.so.1
1156 $(ROOT)/lib/$(MACH64)/liblposix4.ln:= \
1157 REALPATH=liblrt.ln
1158 $(ROOT)/lib/$(MACH64)/libthread_db.so.1:= \
1159 REALPATH=libc_db.so.1
1160 $(ROOT)/lib/$(MACH64)/libthread_db.so:= \
1161 REALPATH=libc_db.so.1
1162 $(ROOT)/usr/lib/$(MACH64)/ld.so.1:= \
1163 REALPATH=../lib/$(MACH64)/ld.so.1
1164 $(ROOT)/usr/lib/$(MACH64)/libadm.so.1:= \
1165 REALPATH=../lib/$(MACH64)/libadm.so.1
1166 $(ROOT)/usr/lib/$(MACH64)/libadm.so:= \
1167 REALPATH=../lib/$(MACH64)/libadm.so.1
1168 $(ROOT)/usr/lib/$(MACH64)/libaio.so.1:= \
1169 REALPATH=../lib/$(MACH64)/libaio.so.1
1170 $(ROOT)/usr/lib/$(MACH64)/libaio.so:= \
1171 REALPATH=../lib/$(MACH64)/libaio.so.1
1172 $(ROOT)/usr/lib/$(MACH64)/libavl.so.1:= \
1173 REALPATH=../lib/$(MACH64)/libavl.so.1
1174 $(ROOT)/usr/lib/$(MACH64)/libavl.so:= \
1175 REALPATH=../lib/$(MACH64)/libavl.so.1
1176 $(ROOT)/usr/lib/$(MACH64)/libasm.so.1:= \

```

```

1177 REALPATH=../../../../lib/$(MACH64)/libbsm.so.1
1178 $(ROOT)/usr/lib/$(MACH64)/libbsm.so:= \
1179 REALPATH=../../../../lib/$(MACH64)/libbsm.so.1
1180 $(ROOT)/usr/lib/$(MACH64)/libc.so.1:= \
1181 REALPATH=../../../../lib/$(MACH64)/libc.so.1
1182 $(ROOT)/usr/lib/$(MACH64)/libc.so:= \
1183 REALPATH=../../../../lib/$(MACH64)/libc.so.1
1184 $(ROOT)/usr/lib/$(MACH64)/libc_db.so.1:= \
1185 REALPATH=../../../../lib/$(MACH64)/libc_db.so.1
1186 $(ROOT)/usr/lib/$(MACH64)/libc_db.so:= \
1187 REALPATH=../../../../lib/$(MACH64)/libc_db.so.1
1188 $(ROOT)/usr/lib/$(MACH64)/libcndutils.so.1:= \
1189 REALPATH=../../../../lib/$(MACH64)/libcndutils.so.1
1190 $(ROOT)/usr/lib/$(MACH64)/libcndutils.so:= \
1191 REALPATH=../../../../lib/$(MACH64)/libcndutils.so.1
1192 $(ROOT)/usr/lib/$(MACH64)/libcontract.so.1:= \
1193 REALPATH=../../../../lib/$(MACH64)/libcontract.so.1
1194 $(ROOT)/usr/lib/$(MACH64)/libcontract.so:= \
1195 REALPATH=../../../../lib/$(MACH64)/libcontract.so.1
1196 $(ROOT)/usr/lib/$(MACH64)/libctf.so.1:= \
1197 REALPATH=../../../../lib/$(MACH64)/libctf.so.1
1198 $(ROOT)/usr/lib/$(MACH64)/libctf.so:= \
1199 REALPATH=../../../../lib/$(MACH64)/libctf.so.1
1200 $(ROOT)/usr/lib/$(MACH64)/libcurses.so.1:= \
1201 REALPATH=../../../../lib/$(MACH64)/libcurses.so.1
1202 $(ROOT)/usr/lib/$(MACH64)/libcurses.so:= \
1203 REALPATH=../../../../lib/$(MACH64)/libcurses.so.1
1204 $(ROOT)/usr/lib/$(MACH64)/libdevice.so.1:= \
1205 REALPATH=../../../../lib/$(MACH64)/libdevice.so.1
1206 $(ROOT)/usr/lib/$(MACH64)/libdevice.so:= \
1207 REALPATH=../../../../lib/$(MACH64)/libdevice.so.1
1208 $(ROOT)/usr/lib/$(MACH64)/libdevid.so.1:= \
1209 REALPATH=../../../../lib/$(MACH64)/libdevid.so.1
1210 $(ROOT)/usr/lib/$(MACH64)/libdevid.so:= \
1211 REALPATH=../../../../lib/$(MACH64)/libdevid.so.1
1212 $(ROOT)/usr/lib/$(MACH64)/libdevinfo.so.1:= \
1213 REALPATH=../../../../lib/$(MACH64)/libdevinfo.so.1
1214 $(ROOT)/usr/lib/$(MACH64)/libdevinfo.so:= \
1215 REALPATH=../../../../lib/$(MACH64)/libdevinfo.so.1
1216 $(ROOT)/usr/lib/$(MACH64)/libdl.so.1:= \
1217 REALPATH=../../../../lib/$(MACH64)/libdl.so.1
1218 $(ROOT)/usr/lib/$(MACH64)/libdl.so:= \
1219 REALPATH=../../../../lib/$(MACH64)/libdl.so.1
1220 $(ROOT)/usr/lib/$(MACH64)/libdlpi.so.1:= \
1221 REALPATH=../../../../lib/$(MACH64)/libdlpi.so.1
1222 $(ROOT)/usr/lib/$(MACH64)/libdlpi.so:= \
1223 REALPATH=../../../../lib/$(MACH64)/libdlpi.so.1
1224 $(ROOT)/usr/lib/$(MACH64)/libdoor.so.1:= \
1225 REALPATH=../../../../lib/$(MACH64)/libdoor.so.1
1226 $(ROOT)/usr/lib/$(MACH64)/libdoor.so:= \
1227 REALPATH=../../../../lib/$(MACH64)/libdoor.so.1
1228 $(ROOT)/usr/lib/$(MACH64)/libefi.so.1:= \
1229 REALPATH=../../../../lib/$(MACH64)/libefi.so.1
1230 $(ROOT)/usr/lib/$(MACH64)/libefi.so:= \
1231 REALPATH=../../../../lib/$(MACH64)/libefi.so.1
1232 $(ROOT)/usr/lib/$(MACH64)/libelf.so.1:= \
1233 REALPATH=../../../../lib/$(MACH64)/libelf.so.1
1234 $(ROOT)/usr/lib/$(MACH64)/libelf.so:= \
1235 REALPATH=../../../../lib/$(MACH64)/libelf.so.1
1236 $(ROOT)/usr/lib/$(MACH64)/libgen.so.1:= \
1237 REALPATH=../../../../lib/$(MACH64)/libgen.so.1
1238 $(ROOT)/usr/lib/$(MACH64)/libgen.so:= \
1239 REALPATH=../../../../lib/$(MACH64)/libgen.so.1
1240 $(ROOT)/usr/lib/$(MACH64)/libinetutil.so.1:= \
1241 REALPATH=../../../../lib/$(MACH64)/libinetutil.so.1
1242 $(ROOT)/usr/lib/$(MACH64)/libinetutil.so:= \

```

```

1243 REALPATH=../../../../lib/$(MACH64)/libinetutil.so.1
1244 $(ROOT)/usr/lib/$(MACH64)/libintl.so.1:= \
1245 REALPATH=../../../../lib/$(MACH64)/libintl.so.1
1246 $(ROOT)/usr/lib/$(MACH64)/libintl.so:= \
1247 REALPATH=../../../../lib/$(MACH64)/libintl.so.1
1248 $(ROOT)/usr/lib/$(MACH64)/libkstat.so.1:= \
1249 REALPATH=../../../../lib/$(MACH64)/libkstat.so.1
1250 $(ROOT)/usr/lib/$(MACH64)/libkstat.so:= \
1251 REALPATH=../../../../lib/$(MACH64)/libkstat.so.1
1252 $(ROOT)/usr/lib/$(MACH64)/liblddbg.so.4:= \
1253 REALPATH=../../../../lib/$(MACH64)/liblddbg.so.4
1254 $(ROOT)/usr/lib/$(MACH64)/libmd.so.1:= \
1255 REALPATH=../../../../lib/$(MACH64)/libmd.so.1
1256 $(ROOT)/usr/lib/$(MACH64)/libmd.so:= \
1257 REALPATH=../../../../lib/$(MACH64)/libmd.so.1
1258 $(ROOT)/usr/lib/$(MACH64)/libmd5.so.1:= \
1259 REALPATH=../../../../lib/$(MACH64)/libmd5.so.1
1260 $(ROOT)/usr/lib/$(MACH64)/libmd5.so:= \
1261 REALPATH=../../../../lib/$(MACH64)/libmd5.so.1
1262 $(ROOT)/usr/lib/$(MACH64)/libmp.so.2:= \
1263 REALPATH=../../../../lib/$(MACH64)/libmp.so.2
1264 $(ROOT)/usr/lib/$(MACH64)/libmp.so:= \
1265 REALPATH=../../../../lib/$(MACH64)/libmp.so.2
1266 $(ROOT)/usr/lib/$(MACH64)/libnsl.so.1:= \
1267 REALPATH=../../../../lib/$(MACH64)/libnsl.so.1
1268 $(ROOT)/usr/lib/$(MACH64)/libnsl.so:= \
1269 REALPATH=../../../../lib/$(MACH64)/libnsl.so.1
1270 $(ROOT)/usr/lib/$(MACH64)/libnvpair.so.1:= \
1271 REALPATH=../../../../lib/$(MACH64)/libnvpair.so.1
1272 $(ROOT)/usr/lib/$(MACH64)/libnvpair.so:= \
1273 REALPATH=../../../../lib/$(MACH64)/libnvpair.so.1
1274 $(ROOT)/usr/lib/$(MACH64)/libpam.so.1:= \
1275 REALPATH=../../../../lib/$(MACH64)/libpam.so.1
1276 $(ROOT)/usr/lib/$(MACH64)/libpam.so:= \
1277 REALPATH=../../../../lib/$(MACH64)/libpam.so.1
1278 $(ROOT)/usr/lib/$(MACH64)/libposix4.so.1:= \
1279 REALPATH=../../../../lib/$(MACH64)/librt.so.1
1280 $(ROOT)/usr/lib/$(MACH64)/libposix4.so:= \
1281 REALPATH=../../../../lib/$(MACH64)/librt.so.1
1282 $(ROOT)/usr/lib/$(MACH64)/libproc.so.1:= \
1283 REALPATH=../../../../lib/$(MACH64)/libproc.so.1
1284 $(ROOT)/usr/lib/$(MACH64)/libproc.so:= \
1285 REALPATH=../../../../lib/$(MACH64)/libproc.so.1
1286 $(ROOT)/usr/lib/$(MACH64)/libpthread.so.1:= \
1287 REALPATH=../../../../lib/$(MACH64)/libpthread.so.1
1288 $(ROOT)/usr/lib/$(MACH64)/libpthread.so:= \
1289 REALPATH=../../../../lib/$(MACH64)/libpthread.so.1
1290 $(ROOT)/usr/lib/$(MACH64)/librcm.so.1:= \
1291 REALPATH=../../../../lib/$(MACH64)/librcm.so.1
1292 $(ROOT)/usr/lib/$(MACH64)/librcm.so:= \
1293 REALPATH=../../../../lib/$(MACH64)/librcm.so.1
1294 $(ROOT)/usr/lib/$(MACH64)/libresolv.so.2:= \
1295 REALPATH=../../../../lib/$(MACH64)/libresolv.so.2
1296 $(ROOT)/usr/lib/$(MACH64)/libresolv.so:= \
1297 REALPATH=../../../../lib/$(MACH64)/libresolv.so.2
1298 $(ROOT)/usr/lib/$(MACH64)/librestart.so.1:= \
1299 REALPATH=../../../../lib/$(MACH64)/librestart.so.1
1300 $(ROOT)/usr/lib/$(MACH64)/librestart.so:= \
1301 REALPATH=../../../../lib/$(MACH64)/librestart.so.1
1302 $(ROOT)/usr/lib/$(MACH64)/librpcsvc.so.1:= \
1303 REALPATH=../../../../lib/$(MACH64)/librpcsvc.so.1
1304 $(ROOT)/usr/lib/$(MACH64)/librpcsvc.so:= \
1305 REALPATH=../../../../lib/$(MACH64)/librpcsvc.so.1
1306 $(ROOT)/usr/lib/$(MACH64)/librt.so.1:= \
1307 REALPATH=../../../../lib/$(MACH64)/librt.so.1
1308 $(ROOT)/usr/lib/$(MACH64)/librt.so:= \

```

```

1309 REALPATH=../../../../lib/$(MACH64)/librt.so.1
1310 $(ROOT)/usr/lib/$(MACH64)/librtld.so.1:= \
1311 REALPATH=../../../../lib/$(MACH64)/librtld.so.1
1312 $(ROOT)/usr/lib/$(MACH64)/librtld_db.so.1:= \
1313 REALPATH=../../../../lib/$(MACH64)/librtld_db.so.1
1314 $(ROOT)/usr/lib/$(MACH64)/librtld_db.so:= \
1315 REALPATH=../../../../lib/$(MACH64)/librtld_db.so.1
1316 $(ROOT)/usr/lib/$(MACH64)/libscf.so.1:= \
1317 REALPATH=../../../../lib/$(MACH64)/libscf.so.1
1318 $(ROOT)/usr/lib/$(MACH64)/libscf.so:= \
1319 REALPATH=../../../../lib/$(MACH64)/libscf.so.1
1320 $(ROOT)/usr/lib/$(MACH64)/libsec.so.1:= \
1321 REALPATH=../../../../lib/$(MACH64)/libsec.so.1
1322 $(ROOT)/usr/lib/$(MACH64)/libsec.so:= \
1323 REALPATH=../../../../lib/$(MACH64)/libsec.so.1
1324 $(ROOT)/usr/lib/$(MACH64)/libsecdb.so.1:= \
1325 REALPATH=../../../../lib/$(MACH64)/libsecdb.so.1
1326 $(ROOT)/usr/lib/$(MACH64)/libsecdb.so:= \
1327 REALPATH=../../../../lib/$(MACH64)/libsecdb.so.1
1328 $(ROOT)/usr/lib/$(MACH64)/libsendfile.so.1:= \
1329 REALPATH=../../../../lib/$(MACH64)/libsendfile.so.1
1330 $(ROOT)/usr/lib/$(MACH64)/libsendfile.so:= \
1331 REALPATH=../../../../lib/$(MACH64)/libsendfile.so.1
1332 $(ROOT)/usr/lib/$(MACH64)/libsocket.so.1:= \
1333 REALPATH=../../../../lib/$(MACH64)/libsocket.so.1
1334 $(ROOT)/usr/lib/$(MACH64)/libsocket.so:= \
1335 REALPATH=../../../../lib/$(MACH64)/libsocket.so.1
1336 $(ROOT)/usr/lib/$(MACH64)/libsysevent.so.1:= \
1337 REALPATH=../../../../lib/$(MACH64)/libsysevent.so.1
1338 $(ROOT)/usr/lib/$(MACH64)/libsysevent.so:= \
1339 REALPATH=../../../../lib/$(MACH64)/libsysevent.so.1
1340 $(ROOT)/usr/lib/$(MACH64)/libtermcap.so.1:= \
1341 REALPATH=../../../../lib/$(MACH64)/libtermcap.so.1
1342 $(ROOT)/usr/lib/$(MACH64)/libtermcap.so:= \
1343 REALPATH=../../../../lib/$(MACH64)/libtermcap.so.1
1344 $(ROOT)/usr/lib/$(MACH64)/libtermplib.so.1:= \
1345 REALPATH=../../../../lib/$(MACH64)/libtermplib.so.1
1346 $(ROOT)/usr/lib/$(MACH64)/libtermplib.so:= \
1347 REALPATH=../../../../lib/$(MACH64)/libtermplib.so.1
1348 $(ROOT)/usr/lib/$(MACH64)/libthread.so.1:= \
1349 REALPATH=../../../../lib/$(MACH64)/libthread.so.1
1350 $(ROOT)/usr/lib/$(MACH64)/libthread.so:= \
1351 REALPATH=../../../../lib/$(MACH64)/libthread.so.1
1352 $(ROOT)/usr/lib/$(MACH64)/libthread_db.so.1:= \
1353 REALPATH=../../../../lib/$(MACH64)/libc_db.so.1
1354 $(ROOT)/usr/lib/$(MACH64)/libthread_db.so:= \
1355 REALPATH=../../../../lib/$(MACH64)/libc_db.so.1
1356 $(ROOT)/usr/lib/$(MACH64)/libtsnet.so.1:= \
1357 REALPATH=../../../../lib/$(MACH64)/libtsnet.so.1
1358 $(ROOT)/usr/lib/$(MACH64)/libtsnet.so:= \
1359 REALPATH=../../../../lib/$(MACH64)/libtsnet.so.1
1360 $(ROOT)/usr/lib/$(MACH64)/libtsol.so.2:= \
1361 REALPATH=../../../../lib/$(MACH64)/libtsol.so.2
1362 $(ROOT)/usr/lib/$(MACH64)/libtsol.so:= \
1363 REALPATH=../../../../lib/$(MACH64)/libtsol.so.2
1364 $(ROOT)/usr/lib/$(MACH64)/libumem.so.1:= \
1365 REALPATH=../../../../lib/$(MACH64)/libumem.so.1
1366 $(ROOT)/usr/lib/$(MACH64)/libumem.so:= \
1367 REALPATH=../../../../lib/$(MACH64)/libumem.so.1
1368 $(ROOT)/usr/lib/$(MACH64)/libuuid.so.1:= \
1369 REALPATH=../../../../lib/$(MACH64)/libuuid.so.1
1370 $(ROOT)/usr/lib/$(MACH64)/libuuid.so:= \
1371 REALPATH=../../../../lib/$(MACH64)/libuuid.so.1
1372 $(ROOT)/usr/lib/$(MACH64)/libuutil.so.1:= \
1373 REALPATH=../../../../lib/$(MACH64)/libuutil.so.1
1374 $(ROOT)/usr/lib/$(MACH64)/libuutil.so:= \

```

```

1375 REALPATH=../../../../lib/$(MACH64)/libuutil.so.1
1376 $(ROOT)/usr/lib/$(MACH64)/libw.so.1:= \
1377 REALPATH=../../../../lib/$(MACH64)/libw.so.1
1378 $(ROOT)/usr/lib/$(MACH64)/libw.so:= \
1379 REALPATH=../../../../lib/$(MACH64)/libw.so.1
1380 $(ROOT)/usr/lib/$(MACH64)/libxnet.so.1:= \
1381 REALPATH=../../../../lib/$(MACH64)/libxnet.so.1
1382 $(ROOT)/usr/lib/$(MACH64)/libxnet.so:= \
1383 REALPATH=../../../../lib/$(MACH64)/libxnet.so.1
1384 $(ROOT)/usr/lib/$(MACH64)/libzfs.so:= \
1385 REALPATH=../../../../lib/$(MACH64)/libzfs.so.1
1386 $(ROOT)/usr/lib/$(MACH64)/libzfs.so.1:= \
1387 REALPATH=../../../../lib/$(MACH64)/libzfs.so.1
1388 $(ROOT)/usr/lib/$(MACH64)/libzfs_core.so:= \
1389 REALPATH=../../../../lib/$(MACH64)/libzfs_core.so.1
1390 $(ROOT)/usr/lib/$(MACH64)/libzfs_core.so.1:= \
1391 REALPATH=../../../../lib/$(MACH64)/libzfs_core.so.1
1392 #endif /* ! codereview */
1393 $(ROOT)/usr/lib/$(MACH64)/libfdisk.so.1:= \
1394 REALPATH=../../../../lib/$(MACH64)/libfdisk.so.1
1395 $(ROOT)/usr/lib/$(MACH64)/libfdisk.so:= \
1396 REALPATH=../../../../lib/$(MACH64)/libfdisk.so.1
1397 $(ROOT)/usr/lib/$(MACH64)/llib-ladm.ln:= \
1398 REALPATH=../../../../lib/$(MACH64)/llib-ladm.ln
1399 $(ROOT)/usr/lib/$(MACH64)/llib-laio.ln:= \
1400 REALPATH=../../../../lib/$(MACH64)/llib-laio.ln
1401 $(ROOT)/usr/lib/$(MACH64)/llib-lavl.ln:= \
1402 REALPATH=../../../../lib/$(MACH64)/llib-lavl.ln
1403 $(ROOT)/usr/lib/$(MACH64)/llib-lbsm.ln:= \
1404 REALPATH=../../../../lib/$(MACH64)/llib-lbsm.ln
1405 $(ROOT)/usr/lib/$(MACH64)/llib-lc.ln:= \
1406 REALPATH=../../../../lib/$(MACH64)/llib-lc.ln
1407 $(ROOT)/usr/lib/$(MACH64)/llib-lcmdutils.ln:= \
1408 REALPATH=../../../../lib/$(MACH64)/llib-lcmdutils.ln
1409 $(ROOT)/usr/lib/$(MACH64)/llib-lcontract.ln:= \
1410 REALPATH=../../../../lib/$(MACH64)/llib-lcontract.ln
1411 $(ROOT)/usr/lib/$(MACH64)/llib-lctf.ln:= \
1412 REALPATH=../../../../lib/$(MACH64)/llib-lctf.ln
1413 $(ROOT)/usr/lib/$(MACH64)/llib-lcurses.ln:= \
1414 REALPATH=../../../../lib/$(MACH64)/llib-lcurses.ln
1415 $(ROOT)/usr/lib/$(MACH64)/llib-ldevice.ln:= \
1416 REALPATH=../../../../lib/$(MACH64)/llib-ldevice.ln
1417 $(ROOT)/usr/lib/$(MACH64)/llib-ldevvid.ln:= \
1418 REALPATH=../../../../lib/$(MACH64)/llib-ldevvid.ln
1419 $(ROOT)/usr/lib/$(MACH64)/llib-ldevinfo.ln:= \
1420 REALPATH=../../../../lib/$(MACH64)/llib-ldevinfo.ln
1421 $(ROOT)/usr/lib/$(MACH64)/llib-ldl.ln:= \
1422 REALPATH=../../../../lib/$(MACH64)/llib-ldl.ln
1423 $(ROOT)/usr/lib/$(MACH64)/llib-ldoor.ln:= \
1424 REALPATH=../../../../lib/$(MACH64)/llib-ldoor.ln
1425 $(ROOT)/usr/lib/$(MACH64)/llib-lefi.ln:= \
1426 REALPATH=../../../../lib/$(MACH64)/llib-lefi.ln
1427 $(ROOT)/usr/lib/$(MACH64)/llib-lelf.ln:= \
1428 REALPATH=../../../../lib/$(MACH64)/llib-lelf.ln
1429 $(ROOT)/usr/lib/$(MACH64)/llib-lgen.ln:= \
1430 REALPATH=../../../../lib/$(MACH64)/llib-lgen.ln
1431 $(ROOT)/usr/lib/$(MACH64)/llib-linetutil.ln:= \
1432 REALPATH=../../../../lib/$(MACH64)/llib-linetutil.ln
1433 $(ROOT)/usr/lib/$(MACH64)/llib-lintl.ln:= \
1434 REALPATH=../../../../lib/$(MACH64)/llib-lintl.ln
1435 $(ROOT)/usr/lib/$(MACH64)/llib-lkstat.ln:= \
1436 REALPATH=../../../../lib/$(MACH64)/llib-lkstat.ln
1437 $(ROOT)/usr/lib/$(MACH64)/llib-lmd5.ln:= \
1438 REALPATH=../../../../lib/$(MACH64)/llib-lmd5.ln
1439 $(ROOT)/usr/lib/$(MACH64)/llib-lns1.ln:= \
1440 REALPATH=../../../../lib/$(MACH64)/llib-lns1.ln

```

```

1441 $(ROOT)/usr/lib/$(MACH64)/llib-lnvpair.ln:= \
1442     REALPATH=../../../../lib/$(MACH64)/llib-lnvpair.ln
1443 $(ROOT)/usr/lib/$(MACH64)/llib-lpam.ln:= \
1444     REALPATH=../../../../lib/$(MACH64)/llib-lpam.ln
1445 $(ROOT)/usr/lib/$(MACH64)/llib-lposix4.ln:= \
1446     REALPATH=../../../../lib/$(MACH64)/llib-lrt.ln
1447 $(ROOT)/usr/lib/$(MACH64)/llib-lpthread.ln:= \
1448     REALPATH=../../../../lib/$(MACH64)/llib-lpthread.ln
1449 $(ROOT)/usr/lib/$(MACH64)/llib-lresolv.ln:= \
1450     REALPATH=../../../../lib/$(MACH64)/llib-lresolv.ln
1451 $(ROOT)/usr/lib/$(MACH64)/llib-lrpsvc.ln:= \
1452     REALPATH=../../../../lib/$(MACH64)/llib-lrpsvc.ln
1453 $(ROOT)/usr/lib/$(MACH64)/llib-lrt.ln:= \
1454     REALPATH=../../../../lib/$(MACH64)/llib-lrt.ln
1455 $(ROOT)/usr/lib/$(MACH64)/llib-lrtld_db.ln:= \
1456     REALPATH=../../../../lib/$(MACH64)/llib-lrtld_db.ln
1457 $(ROOT)/usr/lib/$(MACH64)/llib-lscf.ln:= \
1458     REALPATH=../../../../lib/$(MACH64)/llib-lscf.ln
1459 $(ROOT)/usr/lib/$(MACH64)/llib-lsec.ln:= \
1460     REALPATH=../../../../lib/$(MACH64)/llib-lsec.ln
1461 $(ROOT)/usr/lib/$(MACH64)/llib-lsecdb.ln:= \
1462     REALPATH=../../../../lib/$(MACH64)/llib-lsecdb.ln
1463 $(ROOT)/usr/lib/$(MACH64)/llib-lsendfile.ln:= \
1464     REALPATH=../../../../lib/$(MACH64)/llib-lsendfile.ln
1465 $(ROOT)/usr/lib/$(MACH64)/llib-lsocket.ln:= \
1466     REALPATH=../../../../lib/$(MACH64)/llib-lsocket.ln
1467 $(ROOT)/usr/lib/$(MACH64)/llib-lsysevent.ln:= \
1468     REALPATH=../../../../lib/$(MACH64)/llib-lsysevent.ln
1469 $(ROOT)/usr/lib/$(MACH64)/llib-ltermcap.ln:= \
1470     REALPATH=../../../../lib/$(MACH64)/llib-ltermcap.ln
1471 $(ROOT)/usr/lib/$(MACH64)/llib-ltermplib.ln:= \
1472     REALPATH=../../../../lib/$(MACH64)/llib-lcurses.ln
1473 $(ROOT)/usr/lib/$(MACH64)/llib-lthread.ln:= \
1474     REALPATH=../../../../lib/$(MACH64)/llib-lthread.ln
1475 $(ROOT)/usr/lib/$(MACH64)/llib-lthread_db.ln:= \
1476     REALPATH=../../../../lib/$(MACH64)/llib-lc_db.ln
1477 $(ROOT)/usr/lib/$(MACH64)/llib-ltsnet.ln:= \
1478     REALPATH=../../../../lib/$(MACH64)/llib-ltsnet.ln
1479 $(ROOT)/usr/lib/$(MACH64)/llib-ltsol.ln:= \
1480     REALPATH=../../../../lib/$(MACH64)/llib-ltsol.ln
1481 $(ROOT)/usr/lib/$(MACH64)/llib-lumem.ln:= \
1482     REALPATH=../../../../lib/$(MACH64)/llib-lumem.ln
1483 $(ROOT)/usr/lib/$(MACH64)/llib-luuid.ln:= \
1484     REALPATH=../../../../lib/$(MACH64)/llib-luuid.ln
1485 $(ROOT)/usr/lib/$(MACH64)/llib-lxnet.ln:= \
1486     REALPATH=../../../../lib/$(MACH64)/llib-lxnet.ln
1487 $(ROOT)/usr/lib/$(MACH64)/llib-lzfs.ln:= \
1488     REALPATH=../../../../lib/$(MACH64)/llib-lzfs.ln
1489 $(ROOT)/usr/lib/$(MACH64)/llib-lzfs_core.ln:= \
1490     REALPATH=../../../../lib/$(MACH64)/llib-lzfs_core.ln
1491 #endif /* ! codereview */
1492 $(ROOT)/usr/lib/$(MACH64)/llib-lfdisk.ln:= \
1493     REALPATH=../../../../lib/$(MACH64)/llib-lfdisk.ln
1494 $(ROOT)/usr/lib/$(MACH64)/nss_compat.so.1:= \
1495     REALPATH=../../../../lib/$(MACH64)/nss_compat.so.1
1496 $(ROOT)/usr/lib/$(MACH64)/nss_dns.so.1:= \
1497     REALPATH=../../../../lib/$(MACH64)/nss_dns.so.1
1498 $(ROOT)/usr/lib/$(MACH64)/nss_files.so.1:= \
1499     REALPATH=../../../../lib/$(MACH64)/nss_files.so.1
1500 $(ROOT)/usr/lib/$(MACH64)/nss_nis.so.1:= \
1501     REALPATH=../../../../lib/$(MACH64)/nss_nis.so.1
1502 $(ROOT)/usr/lib/$(MACH64)/nss_user.so.1:= \
1503     REALPATH=../../../../lib/$(MACH64)/nss_user.so.1
1504 $(ROOT)/usr/lib/fm/$(MACH64)/libfmevent.so.1:= \
1505     REALPATH=../../../../lib/fm/$(MACH64)/libfmevent.so.1
1506 $(ROOT)/usr/lib/fm/$(MACH64)/libfmevent.so:= \

```

```

1507     REALPATH=../../../../lib/fm/$(MACH64)/libfmevent.so.1
1508 $(ROOT)/usr/lib/fm/$(MACH64)/llib-lfmevent.ln:= \
1509     REALPATH=../../../../lib/fm/$(MACH64)/llib-lfmevent.ln
1511 i386_SYM.USRLIB= \
1512     /usr/lib/libfdisk.so \
1513     /usr/lib/libfdisk.so.1 \
1514     /usr/lib/llib-lfdisk \
1515     /usr/lib/llib-lfdisk.ln
1517 SYM.USRLIB= \
1518     $(MACH)_SYM.USRLIB \
1519     /lib/libposix4.so \
1520     /lib/libposix4.so.1 \
1521     /lib/llib-lposix4 \
1522     /lib/llib-lposix4.ln \
1523     /lib/libthread_db.so \
1524     /lib/libthread_db.so.1 \
1525     /usr/lib/ld.so.1 \
1526     /usr/lib/libadm.so \
1527     /usr/lib/libadm.so.1 \
1528     /usr/lib/libaio.so \
1529     /usr/lib/libaio.so.1 \
1530     /usr/lib/libavl.so \
1531     /usr/lib/libavl.so.1 \
1532     /usr/lib/libbsm.so \
1533     /usr/lib/libbsm.so.1 \
1534     /usr/lib/libc.so \
1535     /usr/lib/libc.so.1 \
1536     /usr/lib/libc_db.so \
1537     /usr/lib/libc_db.so.1 \
1538     /usr/lib/libcmdutils.so \
1539     /usr/lib/libcmdutils.so.1 \
1540     /usr/lib/libcontract.so \
1541     /usr/lib/libcontract.so.1 \
1542     /usr/lib/libctf.so \
1543     /usr/lib/libctf.so.1 \
1544     /usr/lib/libcurses.so \
1545     /usr/lib/libcurses.so.1 \
1546     /usr/lib/libdevice.so \
1547     /usr/lib/libdevice.so.1 \
1548     /usr/lib/libdevice.so \
1549     /usr/lib/libdevice.so.1 \
1550     /usr/lib/libdevinfo.so \
1551     /usr/lib/libdevinfo.so.1 \
1552     /usr/lib/libdhcpcagent.so \
1553     /usr/lib/libdhcpcagent.so.1 \
1554     /usr/lib/libdhcputil.so \
1555     /usr/lib/libdhcputil.so.1 \
1556     /usr/lib/libddl.so \
1557     /usr/lib/libddl.so.1 \
1558     /usr/lib/libdmpi.so \
1559     /usr/lib/libdmpi.so.1 \
1560     /usr/lib/libdoor.so \
1561     /usr/lib/libdoor.so.1 \
1562     /usr/lib/libefi.so \
1563     /usr/lib/libefi.so.1 \
1564     /usr/lib/libelf.so \
1565     /usr/lib/libelf.so.1 \
1566     /usr/lib/libgen.so \
1567     /usr/lib/libgen.so.1 \
1568     /usr/lib/libinetutil.so \
1569     /usr/lib/libinetutil.so.1 \
1570     /usr/lib/libintl.so \
1571     /usr/lib/libintl.so.1 \
1572     /usr/lib/libkstat.so \

```

```

1573 /usr/lib/libkstat.so.1 \
1574 /usr/lib/liblddbg.so.4 \
1575 /usr/lib/libmd.so \
1576 /usr/lib/libmd.so.1 \
1577 /usr/lib/libmd5.so \
1578 /usr/lib/libmd5.so.1 \
1579 /usr/lib/libmeta.so \
1580 /usr/lib/libmeta.so.1 \
1581 /usr/lib/libmp.so \
1582 /usr/lib/libmp.so.1 \
1583 /usr/lib/libmp.so.2 \
1584 /usr/lib/libnsl.so \
1585 /usr/lib/libnsl.so.1 \
1586 /usr/lib/libnvpair.so \
1587 /usr/lib/libnvpair.so.1 \
1588 /usr/lib/libpam.so \
1589 /usr/lib/libpam.so.1 \
1590 /usr/lib/libposix4.so \
1591 /usr/lib/libposix4.so.1 \
1592 /usr/lib/libproc.so \
1593 /usr/lib/libproc.so.1 \
1594 /usr/lib/libpthread.so \
1595 /usr/lib/libpthread.so.1 \
1596 /usr/lib/librcm.so \
1597 /usr/lib/librcm.so.1 \
1598 /usr/lib/libresolv.so \
1599 /usr/lib/libresolv.so.1 \
1600 /usr/lib/libresolv.so.2 \
1601 /usr/lib/librestart.so \
1602 /usr/lib/librestart.so.1 \
1603 /usr/lib/librpcsvc.so \
1604 /usr/lib/librpcsvc.so.1 \
1605 /usr/lib/librt.so \
1606 /usr/lib/librt.so.1 \
1607 /usr/lib/librtld.so.1 \
1608 /usr/lib/librtld_db.so \
1609 /usr/lib/librtld_db.so.1 \
1610 /usr/lib/libscf.so \
1611 /usr/lib/libscf.so.1 \
1612 /usr/lib/libsec.so \
1613 /usr/lib/libsec.so.1 \
1614 /usr/lib/libsecdb.so \
1615 /usr/lib/libsecdb.so.1 \
1616 /usr/lib/libsendfile.so \
1617 /usr/lib/libsendfile.so.1 \
1618 /usr/lib/libsocket.so \
1619 /usr/lib/libsocket.so.1 \
1620 /usr/lib/libsysevent.so \
1621 /usr/lib/libsysevent.so.1 \
1622 /usr/lib/libtermcap.so \
1623 /usr/lib/libtermcap.so.1 \
1624 /usr/lib/libtermplib.so \
1625 /usr/lib/libtermplib.so.1 \
1626 /usr/lib/libthread.so \
1627 /usr/lib/libthread.so.1 \
1628 /usr/lib/libthread_db.so \
1629 /usr/lib/libthread_db.so.1 \
1630 /usr/lib/libtsnet.so \
1631 /usr/lib/libtsnet.so.1 \
1632 /usr/lib/libtsol.so \
1633 /usr/lib/libtsol.so.2 \
1634 /usr/lib/libumem.so \
1635 /usr/lib/libumem.so.1 \
1636 /usr/lib/libuuid.so \
1637 /usr/lib/libuuid.so.1 \
1638 /usr/lib/libutil.so \

```

```

1639 /usr/lib/libuutil.so.1 \
1640 /usr/lib/libw.so \
1641 /usr/lib/libw.so.1 \
1642 /usr/lib/libxnet.so \
1643 /usr/lib/libxnet.so.1 \
1644 /usr/lib/libzfs.so \
1645 /usr/lib/libzfs.so.1 \
1646 /usr/lib/libzfs_core.so \
1647 /usr/lib/libzfs_core.so.1 \
1648 #endif /* ! codereview */
1649 /usr/lib/libl1ib-ladm \
1650 /usr/lib/libl1ib-ladm.ln \
1651 /usr/lib/libl1ib-laio \
1652 /usr/lib/libl1ib-laio.ln \
1653 /usr/lib/libl1ib-lavl \
1654 /usr/lib/libl1ib-lavl.ln \
1655 /usr/lib/libl1ib-lbsm \
1656 /usr/lib/libl1ib-lbsm.ln \
1657 /usr/lib/libl1ib-lc \
1658 /usr/lib/libl1ib-lc.ln \
1659 /usr/lib/libl1ib-lcmdutils \
1660 /usr/lib/libl1ib-lcmdutils.ln \
1661 /usr/lib/libl1ib-lcontract \
1662 /usr/lib/libl1ib-lcontract.ln \
1663 /usr/lib/libl1ib-lctf \
1664 /usr/lib/libl1ib-lctf.ln \
1665 /usr/lib/libl1ib-lcurses \
1666 /usr/lib/libl1ib-lcurses.ln \
1667 /usr/lib/libl1ib-ldevice \
1668 /usr/lib/libl1ib-ldevice.ln \
1669 /usr/lib/libl1ib-ldevid \
1670 /usr/lib/libl1ib-ldevid.ln \
1671 /usr/lib/libl1ib-ldevinfo \
1672 /usr/lib/libl1ib-ldevinfo.ln \
1673 /usr/lib/libl1ib-ldhcpagent \
1674 /usr/lib/libl1ib-ldhcpagent.ln \
1675 /usr/lib/libl1ib-ldhcputil \
1676 /usr/lib/libl1ib-ldhcputil.ln \
1677 /usr/lib/libl1ib-ldl \
1678 /usr/lib/libl1ib-ldl.ln \
1679 /usr/lib/libl1ib-ldoor \
1680 /usr/lib/libl1ib-ldoor.ln \
1681 /usr/lib/libl1ib-lefi \
1682 /usr/lib/libl1ib-lefi.ln \
1683 /usr/lib/libl1ib-lelf \
1684 /usr/lib/libl1ib-lelf.ln \
1685 /usr/lib/libl1ib-lgen \
1686 /usr/lib/libl1ib-lgen.ln \
1687 /usr/lib/libl1ib-linetutil \
1688 /usr/lib/libl1ib-linetutil.ln \
1689 /usr/lib/libl1ib-lint1 \
1690 /usr/lib/libl1ib-lint1.ln \
1691 /usr/lib/libl1ib-lkstat \
1692 /usr/lib/libl1ib-lkstat.ln \
1693 /usr/lib/libl1ib-lmd5 \
1694 /usr/lib/libl1ib-lmd5.ln \
1695 /usr/lib/libl1ib-lmeta \
1696 /usr/lib/libl1ib-lmeta.ln \
1697 /usr/lib/libl1ib-lnsl \
1698 /usr/lib/libl1ib-lnsl.ln \
1699 /usr/lib/libl1ib-lnvpair \
1700 /usr/lib/libl1ib-lnvpair.ln \
1701 /usr/lib/libl1ib-lpam \
1702 /usr/lib/libl1ib-lpam.ln \
1703 /usr/lib/libl1ib-lposix4 \
1704 /usr/lib/libl1ib-lposix4.ln \

```

```

1705 /usr/lib/libl1ib-lpthread \
1706 /usr/lib/libl1ib-lpthread.ln \
1707 /usr/lib/libl1ib-lresolv \
1708 /usr/lib/libl1ib-lresolv.ln \
1709 /usr/lib/libl1ib-lrpcsvc \
1710 /usr/lib/libl1ib-lrpcsvc.ln \
1711 /usr/lib/libl1ib-lrt \
1712 /usr/lib/libl1ib-lrt.ln \
1713 /usr/lib/libl1ib-lrtld_db \
1714 /usr/lib/libl1ib-lrtld_db.ln \
1715 /usr/lib/libl1ib-lscf \
1716 /usr/lib/libl1ib-lscf.ln \
1717 /usr/lib/libl1ib-lsec \
1718 /usr/lib/libl1ib-lsec.ln \
1719 /usr/lib/libl1ib-lsecdb \
1720 /usr/lib/libl1ib-lsecdb.ln \
1721 /usr/lib/libl1ib-lsendfile \
1722 /usr/lib/libl1ib-lsendfile.ln \
1723 /usr/lib/libl1ib-lsocket \
1724 /usr/lib/libl1ib-lsocket.ln \
1725 /usr/lib/libl1ib-lsysevent \
1726 /usr/lib/libl1ib-lsysevent.ln \
1727 /usr/lib/libl1ib-ltermcap \
1728 /usr/lib/libl1ib-ltermcap.ln \
1729 /usr/lib/libl1ib-ltermlib \
1730 /usr/lib/libl1ib-ltermlib.ln \
1731 /usr/lib/libl1ib-lthread \
1732 /usr/lib/libl1ib-lthread.ln \
1733 /usr/lib/libl1ib-lthread_db \
1734 /usr/lib/libl1ib-lthread_db.ln \
1735 /usr/lib/libl1ib-ltsnet \
1736 /usr/lib/libl1ib-ltsnet.ln \
1737 /usr/lib/libl1ib-ltsol \
1738 /usr/lib/libl1ib-ltsol.ln \
1739 /usr/lib/libl1ib-lumem \
1740 /usr/lib/libl1ib-lumem.ln \
1741 /usr/lib/libl1ib-luuid \
1742 /usr/lib/libl1ib-luuid.ln \
1743 /usr/lib/libl1ib-lxnet \
1744 /usr/lib/libl1ib-lxnet.ln \
1745 /usr/lib/libl1ib-lzfs \
1746 /usr/lib/libl1ib-lzfs.ln \
1747 /usr/lib/libl1ib-lzfs_core \
1748 /usr/lib/libl1ib-lzfs_core.ln \
1749 #endif /* ! codereview */
1750 /usr/lib/nss_compat.so.1 \
1751 /usr/lib/nss_dns.so.1 \
1752 /usr/lib/nss_files.so.1 \
1753 /usr/lib/nss_nis.so.1 \
1754 /usr/lib/nss_user.so.1 \
1755 /usr/lib/fm/libfmevent.so \
1756 /usr/lib/fm/libfmevent.so.1 \
1757 /usr/lib/fm/liblfmevent \
1758 /usr/lib/fm/liblfmevent.ln

1760 sparcv9_SYM.USRLIB64=

1762 amd64_SYM.USRLIB64= \
1763 /usr/lib/amd64/libfdisk.so \
1764 /usr/lib/amd64/libfdisk.so.1 \
1765 /usr/lib/amd64/libl1lib-lfdisk.ln

1768 SYM.USRLIB64= \
1769 $(MACH64)_SYM.USRLIB64 \
1770 /lib/$(MACH64)/libposix4.so \

```

```

1771 /lib/$(MACH64)/libposix4.so.1 \
1772 /lib/$(MACH64)/libl1lib-lposix4.ln \
1773 /lib/$(MACH64)/libthread_db.so \
1774 /lib/$(MACH64)/libthread_db.so.1 \
1775 /usr/lib/$(MACH64)/ld.so.1 \
1776 /usr/lib/$(MACH64)/libadm.so \
1777 /usr/lib/$(MACH64)/libadm.so.1 \
1778 /usr/lib/$(MACH64)/libaio.so \
1779 /usr/lib/$(MACH64)/libaio.so.1 \
1780 /usr/lib/$(MACH64)/libavl.so \
1781 /usr/lib/$(MACH64)/libavl.so.1 \
1782 /usr/lib/$(MACH64)/libbsm.so \
1783 /usr/lib/$(MACH64)/libbsm.so.1 \
1784 /usr/lib/$(MACH64)/libc.so \
1785 /usr/lib/$(MACH64)/libc.so.1 \
1786 /usr/lib/$(MACH64)/libc_db.so \
1787 /usr/lib/$(MACH64)/libc_db.so.1 \
1788 /usr/lib/$(MACH64)/libcmdutils.so \
1789 /usr/lib/$(MACH64)/libcmdutils.so.1 \
1790 /usr/lib/$(MACH64)/libcontract.so \
1791 /usr/lib/$(MACH64)/libcontract.so.1 \
1792 /usr/lib/$(MACH64)/libctf.so \
1793 /usr/lib/$(MACH64)/libctf.so.1 \
1794 /usr/lib/$(MACH64)/libcurses.so \
1795 /usr/lib/$(MACH64)/libcurses.so.1 \
1796 /usr/lib/$(MACH64)/libdevice.so \
1797 /usr/lib/$(MACH64)/libdevice.so.1 \
1798 /usr/lib/$(MACH64)/libdevvid.so \
1799 /usr/lib/$(MACH64)/libdevvid.so.1 \
1800 /usr/lib/$(MACH64)/libdevinfo.so \
1801 /usr/lib/$(MACH64)/libdevinfo.so.1 \
1802 /usr/lib/$(MACH64)/libdl.so \
1803 /usr/lib/$(MACH64)/libdl.so.1 \
1804 /usr/lib/$(MACH64)/libdmpi.so \
1805 /usr/lib/$(MACH64)/libdmpi.so.1 \
1806 /usr/lib/$(MACH64)/libdoor.so \
1807 /usr/lib/$(MACH64)/libdoor.so.1 \
1808 /usr/lib/$(MACH64)/libefi.so \
1809 /usr/lib/$(MACH64)/libefi.so.1 \
1810 /usr/lib/$(MACH64)/libelf.so \
1811 /usr/lib/$(MACH64)/libelf.so.1 \
1812 /usr/lib/$(MACH64)/libgen.so \
1813 /usr/lib/$(MACH64)/libgen.so.1 \
1814 /usr/lib/$(MACH64)/libinetutil.so \
1815 /usr/lib/$(MACH64)/libinetutil.so.1 \
1816 /usr/lib/$(MACH64)/libintl.so \
1817 /usr/lib/$(MACH64)/libintl.so.1 \
1818 /usr/lib/$(MACH64)/libkstat.so \
1819 /usr/lib/$(MACH64)/libkstat.so.1 \
1820 /usr/lib/$(MACH64)/libl1lib-lldbg.so.4 \
1821 /usr/lib/$(MACH64)/libmd.so \
1822 /usr/lib/$(MACH64)/libmd.so.1 \
1823 /usr/lib/$(MACH64)/libmd5.so \
1824 /usr/lib/$(MACH64)/libmd5.so.1 \
1825 /usr/lib/$(MACH64)/libmp.so \
1826 /usr/lib/$(MACH64)/libmp.so.2 \
1827 /usr/lib/$(MACH64)/libnsl.so \
1828 /usr/lib/$(MACH64)/libnsl.so.1 \
1829 /usr/lib/$(MACH64)/libnvpair.so \
1830 /usr/lib/$(MACH64)/libnvpair.so.1 \
1831 /usr/lib/$(MACH64)/libpam.so \
1832 /usr/lib/$(MACH64)/libpam.so.1 \
1833 /usr/lib/$(MACH64)/libposix4.so \
1834 /usr/lib/$(MACH64)/libposix4.so.1 \
1835 /usr/lib/$(MACH64)/libproc.so \
1836 /usr/lib/$(MACH64)/libproc.so.1 \

```

```

1837 /usr/lib/$(MACH64)/libpthread.so \
1838 /usr/lib/$(MACH64)/libpthread.so.1 \
1839 /usr/lib/$(MACH64)/librcm.so \
1840 /usr/lib/$(MACH64)/librcm.so.1 \
1841 /usr/lib/$(MACH64)/libresolv.so \
1842 /usr/lib/$(MACH64)/libresolv.so.2 \
1843 /usr/lib/$(MACH64)/librestart.so \
1844 /usr/lib/$(MACH64)/librestart.so.1 \
1845 /usr/lib/$(MACH64)/librpcsvc.so \
1846 /usr/lib/$(MACH64)/librpcsvc.so.1 \
1847 /usr/lib/$(MACH64)/librt.so \
1848 /usr/lib/$(MACH64)/librt.so.1 \
1849 /usr/lib/$(MACH64)/librtld.so.1 \
1850 /usr/lib/$(MACH64)/librtld_db.so \
1851 /usr/lib/$(MACH64)/librtld_db.so.1 \
1852 /usr/lib/$(MACH64)/libscf.so \
1853 /usr/lib/$(MACH64)/libscf.so.1 \
1854 /usr/lib/$(MACH64)/libsec.so \
1855 /usr/lib/$(MACH64)/libsec.so.1 \
1856 /usr/lib/$(MACH64)/libsecdb.so \
1857 /usr/lib/$(MACH64)/libsecdb.so.1 \
1858 /usr/lib/$(MACH64)/libsndfile.so \
1859 /usr/lib/$(MACH64)/libsndfile.so.1 \
1860 /usr/lib/$(MACH64)/libsocket.so \
1861 /usr/lib/$(MACH64)/libsocket.so.1 \
1862 /usr/lib/$(MACH64)/libsysevent.so \
1863 /usr/lib/$(MACH64)/libsysevent.so.1 \
1864 /usr/lib/$(MACH64)/libtermcap.so \
1865 /usr/lib/$(MACH64)/libtermcap.so.1 \
1866 /usr/lib/$(MACH64)/libtermplib.so \
1867 /usr/lib/$(MACH64)/libtermplib.so.1 \
1868 /usr/lib/$(MACH64)/libthread.so \
1869 /usr/lib/$(MACH64)/libthread.so.1 \
1870 /usr/lib/$(MACH64)/libthread_db.so \
1871 /usr/lib/$(MACH64)/libthread_db.so.1 \
1872 /usr/lib/$(MACH64)/libtsnet.so \
1873 /usr/lib/$(MACH64)/libtsnet.so.1 \
1874 /usr/lib/$(MACH64)/libtsol.so \
1875 /usr/lib/$(MACH64)/libtsol.so.2 \
1876 /usr/lib/$(MACH64)/libumem.so \
1877 /usr/lib/$(MACH64)/libumem.so.1 \
1878 /usr/lib/$(MACH64)/libuuid.so \
1879 /usr/lib/$(MACH64)/libuuid.so.1 \
1880 /usr/lib/$(MACH64)/libutil.so \
1881 /usr/lib/$(MACH64)/libutil.so.1 \
1882 /usr/lib/$(MACH64)/libw.so \
1883 /usr/lib/$(MACH64)/libw.so.1 \
1884 /usr/lib/$(MACH64)/libxnet.so \
1885 /usr/lib/$(MACH64)/libxnet.so.1 \
1886 /usr/lib/$(MACH64)/libzfs.so \
1887 /usr/lib/$(MACH64)/libzfs.so.1 \
1888 /usr/lib/$(MACH64)/libzfs_core.so \
1889 /usr/lib/$(MACH64)/libzfs_core.so.1 \
1890 #endif /* ! codereview */
1891 /usr/lib/$(MACH64)/llib-ladm.ln \
1892 /usr/lib/$(MACH64)/llib-laio.ln \
1893 /usr/lib/$(MACH64)/llib-lavl.ln \
1894 /usr/lib/$(MACH64)/llib-lbsm.ln \
1895 /usr/lib/$(MACH64)/llib-lc.ln \
1896 /usr/lib/$(MACH64)/llib-lcmdutils.ln \
1897 /usr/lib/$(MACH64)/llib-lcontract.ln \
1898 /usr/lib/$(MACH64)/llib-lctf.ln \
1899 /usr/lib/$(MACH64)/llib-lcurses.ln \
1900 /usr/lib/$(MACH64)/llib-ldevice.ln \
1901 /usr/lib/$(MACH64)/llib-ldevid.ln \
1902 /usr/lib/$(MACH64)/llib-ldevinfo.ln \

```

```

1903 /usr/lib/$(MACH64)/llib-ld1.ln \
1904 /usr/lib/$(MACH64)/llib-ldoor.ln \
1905 /usr/lib/$(MACH64)/llib-lefi.ln \
1906 /usr/lib/$(MACH64)/llib-lelf.ln \
1907 /usr/lib/$(MACH64)/llib-lgen.ln \
1908 /usr/lib/$(MACH64)/llib-linetutil.ln \
1909 /usr/lib/$(MACH64)/llib-lintl.ln \
1910 /usr/lib/$(MACH64)/llib-lkstat.ln \
1911 /usr/lib/$(MACH64)/llib-lmd5.ln \
1912 /usr/lib/$(MACH64)/llib-lns1.ln \
1913 /usr/lib/$(MACH64)/llib-lnvpair.ln \
1914 /usr/lib/$(MACH64)/llib-lpam.ln \
1915 /usr/lib/$(MACH64)/llib-lposix4.ln \
1916 /usr/lib/$(MACH64)/llib-lpthread.ln \
1917 /usr/lib/$(MACH64)/llib-lresolv.ln \
1918 /usr/lib/$(MACH64)/llib-lrpcsvc.ln \
1919 /usr/lib/$(MACH64)/llib-lrt.ln \
1920 /usr/lib/$(MACH64)/llib-lrtld_db.ln \
1921 /usr/lib/$(MACH64)/llib-lscf.ln \
1922 /usr/lib/$(MACH64)/llib-lsec.ln \
1923 /usr/lib/$(MACH64)/llib-lsecdb.ln \
1924 /usr/lib/$(MACH64)/llib-lsndfile.ln \
1925 /usr/lib/$(MACH64)/llib-lsocket.ln \
1926 /usr/lib/$(MACH64)/llib-lsysevent.ln \
1927 /usr/lib/$(MACH64)/llib-ltermcap.ln \
1928 /usr/lib/$(MACH64)/llib-ltermplib.ln \
1929 /usr/lib/$(MACH64)/llib-lthread.ln \
1930 /usr/lib/$(MACH64)/llib-lthread_db.ln \
1931 /usr/lib/$(MACH64)/llib-ltsnet.ln \
1932 /usr/lib/$(MACH64)/llib-ltsol.ln \
1933 /usr/lib/$(MACH64)/llib-lumem.ln \
1934 /usr/lib/$(MACH64)/llib-luuid.ln \
1935 /usr/lib/$(MACH64)/llib-lxnet.ln \
1936 /usr/lib/$(MACH64)/llib-lzfs.ln \
1937 /usr/lib/$(MACH64)/llib-lzfs_core.ln \
1938 #endif /* ! codereview */
1939 /usr/lib/$(MACH64)/nss_compat.so.1 \
1940 /usr/lib/$(MACH64)/nss_dns.so.1 \
1941 /usr/lib/$(MACH64)/nss_files.so.1 \
1942 /usr/lib/$(MACH64)/nss_nis.so.1 \
1943 /usr/lib/$(MACH64)/nss_user.so.1 \
1944 /usr/lib/fm/$(MACH64)/libfmevent.so \
1945 /usr/lib/fm/$(MACH64)/libfmevent.so.1 \
1946 /usr/lib/fm/$(MACH64)/llib-lfmevent.ln

1948 #
1949 # usr/src/Makefile uses INS.dir for any member of ROOTDIRS, the fact
1950 # these are symlinks to files has no bearing on this.
1951 #
1952 $(FILELINKS:%=$(ROOT)%):= \
1953     INS.dir= -$(RM) $@; $(SYMLINK) $(REALPATH) $@

```

 77848 Thu Jun 28 15:57:06 2012
 new/usr/src/cmd/truss/codes.c
 2882 implement libfs_core
 2883 changing "cannount" property to "on" should not always remount dataset
 2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
 Reviewed by: George Wilson <george.wilson@delphix.com>
 Reviewed by: Chris Siden <christopher.siden@delphix.com>
 Reviewed by: Garrett D'Amore <garrett@damore.org>
 Reviewed by: Bill Pijewski <wdp@joyent.com>
 Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```

unchanged portion omitted
331 (uint_t)TCGETA, "TCGETA", NULL },
332 (uint_t)TCSETA, "TCSETA", NULL },
333 (uint_t)TCSETAW, "TCSETAW", NULL },
334 (uint_t)TCSETAF, "TCSETAF", NULL },
335 (uint_t)TCFLSH, "TCFLSH", NULL },
336 (uint_t)TIOCKBON, "TIOCKBON", NULL },
337 (uint_t)TIOCKBOF, "TIOCKBOF", NULL },
338 (uint_t)KBENABLED, "KBENABLED", NULL },
339 (uint_t)TCGETS, "TCGETS", NULL },
340 (uint_t)TCSETS, "TCSETS", NULL },
341 (uint_t)TCSETSW, "TCSETSW", NULL },
342 (uint_t)TCSETSF, "TCSETSF", NULL },
343 (uint_t)TCXONC, "TCXONC", NULL },
344 (uint_t)TCSBRK, "TCSBRK", NULL },
345 (uint_t)TCDSET, "TCDSET", NULL },
346 (uint_t)RTS_TOG, "RTS_TOG", NULL },
347 (uint_t)TIOCSWINSZ, "TIOCSWINSZ", NULL },
348 (uint_t)TIOCGWINSZ, "TIOCGWINSZ", NULL },
349 (uint_t)TIOCGETD, "TIOCGETD", NULL },
350 (uint_t)TIOCSETD, "TIOCSETD", NULL },
351 (uint_t)TIOCHPCL, "TIOCHPCL", NULL },
352 (uint_t)TIOCGETP, "TIOCGETP", NULL },
353 (uint_t)TIOCSETP, "TIOCSETP", NULL },
354 (uint_t)TIOCSETN, "TIOCSETN", NULL },
355 (uint_t)TIOCEXCL, "TIOCEXCL", NULL },
356 (uint_t)TIOCNXCL, "TIOCNXCL", NULL },
357 (uint_t)TIOCFLUSH, "TIOCFLUSH", NULL },
358 (uint_t)TIOCSETC, "TIOCSETC", NULL },
359 (uint_t)TIOCGETC, "TIOCGETC", NULL },
360 (uint_t)TIOCGPRP, "TIOCGPRP", NULL },
361 (uint_t)TIOCSGRP, "TIOCSGRP", NULL },
362 (uint_t)TIOCGSID, "TIOCGSID", NULL },
363 (uint_t)TIOCSTI, "TIOCSTI", NULL },
364 (uint_t)TIOCMSET, "TIOCMSET", NULL },
365 (uint_t)TIOCLBIS, "TIOCLBIS", NULL },
366 (uint_t)TIOCMBIC, "TIOCMBIC", NULL },
367 (uint_t)TIOCMGET, "TIOCMGET", NULL },
368 (uint_t)TIOCREMOTE, "TIOCREMOTE", NULL },
369 (uint_t)TIOCSIGNAL, "TIOCSIGNAL", NULL },
370 (uint_t)TIOCSTART, "TIOCSTART", NULL },
371 (uint_t)TIOCSTOP, "TIOCSTOP", NULL },
372 (uint_t)TIOCNOTTY, "TIOCNOTTY", NULL },
373 (uint_t)TIOCSCTTY, "TIOCSCTTY", NULL },
374 (uint_t)TIOCOUQ, "TIOCOUQ", NULL },
375 (uint_t)TIOCLTC, "TIOCLTC", NULL },
376 (uint_t)TIOCLTC, "TIOCLTC", NULL },
377 (uint_t)TIOCDTR, "TIOCDTR", NULL },
378 (uint_t)TIOCDTR, "TIOCDTR", NULL },
379 (uint_t)TIOCCBRK, "TIOCCBRK", NULL },
380 (uint_t)TIOCSBRK, "TIOCSBRK", NULL },
381 (uint_t)TIOCLGET, "TIOCLGET", NULL },
382 (uint_t)TIOCLSET, "TIOCLSET", NULL },
383 (uint_t)TIOCLBIC, "TIOCLBIC", NULL },

```

```

384 { (uint_t)TIOCLBIS, "TIOCLBIS", NULL },
386 { (uint_t)TIOCSILOOP, "TIOCSILOOP", NULL },
387 { (uint_t)TIOCCILoop, "TIOCCILoop", NULL },
389 { (uint_t)TIOCGPPS, "TIOCGPPS", NULL },
390 { (uint_t)TIOCSPPS, "TIOCSPPS", NULL },
391 { (uint_t)TIOCGPPSEV, "TIOCGPPSEV", NULL },
393 { (uint_t)TIOCPKT, "TIOCPKT", NULL }, /* ptyvar.h */
394 { (uint_t)TIOCUCNTL, "TIOCUCNTL", NULL },
395 { (uint_t)TIOCTCNTL, "TIOCTCNTL", NULL },
396 { (uint_t)TIOCISPACE, "TIOCISPACE", NULL },
397 { (uint_t)TIOCISIZE, "TIOCISIZE", NULL },
398 { (uint_t)TIOCSSIZE, "TIOCSSIZE", "ttysize" },
399 { (uint_t)TIOCGSIZE, "TIOCGSIZE", "ttysize" },
401 /*
402 * Unfortunately, the DLIOC and LDIOC codes overlap. Since the LDIOC
403 * ioctls (for xenix compatibility) are far less likely to be used, we
404 * give preference to DLIOC.
405 */
406 { (uint_t)DLIOCRAW, "DLIOCRAW", NULL },
407 { (uint_t)DLIOCNATIVE, "DLIOCNATIVE", NULL },
408 { (uint_t)DLIOCIPNETINFO, "DLIOCIPNETINFO", NULL },
409 { (uint_t)DLIOCLINK, "DLIOCLINK", NULL },
411 { (uint_t)LDOPEN, "LDOPEN", NULL },
412 { (uint_t)LDCLOSE, "LDCLOSE", NULL },
413 { (uint_t)LDCHG, "LDCHG", NULL },
414 { (uint_t)LDGETT, "LDGETT", NULL },
415 { (uint_t)LDSETT, "LDSETT", NULL },
416 { (uint_t)LDSMAP, "LDSMAP", NULL },
417 { (uint_t)LDGMAP, "LDGMAP", NULL },
418 { (uint_t)LDNMAP, "LDNMAP", NULL },
419 { (uint_t)TCGETX, "TCGETX", NULL },
420 { (uint_t)TCSETX, "TCSETX", NULL },
421 { (uint_t)TCSETXW, "TCSETXW", NULL },
422 { (uint_t)TCSETXF, "TCSETXF", NULL },
423 { (uint_t)FIORDCHK, "FIORDCHK", NULL },
424 { (uint_t)FIOCLEX, "FIOCLEX", NULL },
425 { (uint_t)FIONCLEX, "FIONCLEX", NULL },
426 { (uint_t)FIONREAD, "FIONREAD", NULL },
427 { (uint_t)FIONBIO, "FIONBIO", NULL },
428 { (uint_t)FIOASYNC, "FIOASYNC", NULL },
429 { (uint_t)FIOSETOWN, "FIOSETOWN", NULL },
430 { (uint_t)FIOGETOWN, "FIOGETOWN", NULL },
431 #ifdef DIOCGETP
432 { (uint_t)DIOCGETP, "DIOCGETP", NULL },
433 { (uint_t)DIOCSETP, "DIOCSETP", NULL },
434 #endif
435 #ifdef DIOCGETC
436 { (uint_t)DIOCGETC, "DIOCGETC", NULL },
437 { (uint_t)DIOCGETB, "DIOCGETB", NULL },
438 { (uint_t)DIOCSETE, "DIOCSETE", NULL },
439 #endif
440 #ifdef IFFORMAT
441 { (uint_t)IFFORMAT, "IFFORMAT", NULL },
442 { (uint_t)IFBCHECK, "IFBCHECK", NULL },
443 { (uint_t)IFCONFIRM, "IFCONFIRM", NULL },
444 #endif
445 #ifdef LIOCGETP
446 { (uint_t)LIOCGETP, "LIOCGETP", NULL },
447 { (uint_t)LIOCSETP, "LIOCSETP", NULL },
448 { (uint_t)LIOCGETS, "LIOCGETS", NULL },
449 { (uint_t)LIOCSETS, "LIOCSETS", NULL },

```



```

450 #endif
451 #ifdef JBOOT
452     { (uint_t)JBOOT,      "JBOOT",      NULL },
453     { (uint_t)JTERM,     "JTERM",      NULL },
454     { (uint_t)JMPX,      "JMPX",      NULL },
455 #ifdef JTIMO
456     { (uint_t)JTIMO,     "JTIMO",      NULL },
457 #endif
458     { (uint_t)JWINSIZE,  "JWINSIZE",  NULL },
459     { (uint_t)JTIMOM,    "JTIMOM",    NULL },
460     { (uint_t)JZOMBOOT,  "JZOMBOOT",  NULL },
461     { (uint_t)JAGENT,    "JAGENT",    NULL },
462     { (uint_t)JTRUN,     "JTRUN",     NULL },
463     { (uint_t)JXTPROTO,  "JXTPROTO",  NULL },
464 #endif
465     { (uint_t)KSTAT_IOC_CHAIN_ID, "KSTAT_IOC_CHAIN_ID", NULL },
466     { (uint_t)KSTAT_IOC_READ,     "KSTAT_IOC_READ",     NULL },
467     { (uint_t)KSTAT_IOC_WRITE,    "KSTAT_IOC_WRITE",    NULL },
468     { (uint_t)STGET,              "STGET",              NULL },
469     { (uint_t)STSET,              "STSET",              NULL },
470     { (uint_t)STHROW,             "STHROW",            NULL },
471     { (uint_t)STWLINE,           "STWLINE",           NULL },
472     { (uint_t)STTSV,             "STTSV",             NULL },
473     { (uint_t)I_NREAD,            "I_NREAD",           NULL },
474     { (uint_t)I_PUSH,            "I_PUSH",            NULL },
475     { (uint_t)I_POP,              "I_POP",              NULL },
476     { (uint_t)I_LOOK,            "I_LOOK",            NULL },
477     { (uint_t)I_FLUSH,           "I_FLUSH",           NULL },
478     { (uint_t)I_SRDLOPT,         "I_SRDLOPT",        NULL },
479     { (uint_t)I_GRDOPT,         "I_GRDOPT",        NULL },
480     { (uint_t)I_STR,             "I_STR",             NULL },
481     { (uint_t)I_SETSIG,          "I_SETSIG",          NULL },
482     { (uint_t)I_GETSIG,          "I_GETSIG",          NULL },
483     { (uint_t)I_FIND,           "I_FIND",            NULL },
484     { (uint_t)I_LINK,           "I_LINK",            NULL },
485     { (uint_t)I_UNLINK,         "I_UNLINK",         NULL },
486     { (uint_t)I_PEEK,           "I_PEEK",            NULL },
487     { (uint_t)I_FDINSERT,       "I_FDINSERT",       NULL },
488     { (uint_t)I_SENDFD,         "I_SENDFD",         NULL },
489     { (uint_t)I_RECVFD,         "I_RECVFD",         NULL },
490     { (uint_t)I_SWROPT,         "I_SWROPT",         NULL },
491     { (uint_t)I_GWROPT,         "I_GWROPT",         NULL },
492     { (uint_t)I_LIST,           "I_LIST",            NULL },
493     { (uint_t)I_PLINK,          "I_PLINK",           NULL },
494     { (uint_t)I_PUNLINK,        "I_PUNLINK",        NULL },
495     { (uint_t)I_FLUSHBAND,      "I_FLUSHBAND",      NULL },
496     { (uint_t)I_CKBAND,         "I_CKBAND",         NULL },
497     { (uint_t)I_GETBAND,        "I_GETBAND",        NULL },
498     { (uint_t)I_ATMARK,         "I_ATMARK",         NULL },
499     { (uint_t)I_SETCLTIME,      "I_SETCLTIME",      NULL },
500     { (uint_t)I_GETCLTIME,      "I_GETCLTIME",      NULL },
501     { (uint_t)I_CANPUT,         "I_CANPUT",         NULL },
502     { (uint_t)I_ANCHOR,         "I_ANCHOR",         NULL },
503     { (uint_t)I_CMD,            "I_CMD",            NULL },
504 #ifdef TI_GETINFO
505     { (uint_t)TI_GETINFO,      "TI_GETINFO",      NULL },
506     { (uint_t)TI_OPTMGMT,      "TI_OPTMGMT",      NULL },
507     { (uint_t)TI_BIND,         "TI_BIND",         NULL },
508     { (uint_t)TI_UNBIND,       "TI_UNBIND",       NULL },
509 #endif
510 #ifdef TI_CAPABILITY
511     { (uint_t)TI_CAPABILITY,    "TI_CAPABILITY",    NULL },
512 #endif
513 #ifdef TI_GETMYNAME
514     { (uint_t)TI_GETMYNAME,     "TI_GETMYNAME",     NULL },
515     { (uint_t)TI_GETPEERNAME,   "TI_GETPEERNAME",   NULL },

```

```

516     { (uint_t)TI_SETMYNAME,     "TI_SETMYNAME",     NULL },
517     { (uint_t)TI_SETPEERNAME,   "TI_SETPEERNAME",   NULL },
518 #endif
519 #ifdef V_PREAD
520     { (uint_t)V_PREAD,          "V_PREAD",          NULL },
521     { (uint_t)V_PWRITE,         "V_PWRITE",         NULL },
522     { (uint_t)V_PDREAD,         "V_PDREAD",         NULL },
523     { (uint_t)V_PDWRITE,        "V_PDWRITE",        NULL },
524 #if !defined(__i386) && !defined(__amd64)
525     { (uint_t)V_GETSSZ,         "V_GETSSZ",         NULL },
526 #endif /* !__i386 */
527 #endif
528 /* audio */
529     { (uint_t)AUDIO_GETINFO,    "AUDIO_GETINFO",    NULL },
530     { (uint_t)AUDIO_SETINFO,    "AUDIO_SETINFO",    NULL },
531     { (uint_t)AUDIO_DRAIN,      "AUDIO_DRAIN",      NULL },
532     { (uint_t)AUDIO_GETDEV,     "AUDIO_GETDEV",     NULL },
533     { (uint_t)AUDIO_DIAG_LOOPBACK, "AUDIO_DIAG_LOOPBACK", NULL },
534     { (uint_t)AUDIO_GET_CH_NUMBER, "AUDIO_GET_CH_NUMBER", NULL },
535     { (uint_t)AUDIO_GET_CH_TYPE, "AUDIO_GET_CH_TYPE", NULL },
536     { (uint_t)AUDIO_GET_NUM_CHS, "AUDIO_GET_NUM_CHS", NULL },
537     { (uint_t)AUDIO_GET_AD_DEV,  "AUDIO_GET_AD_DEV",  NULL },
538     { (uint_t)AUDIO_GET_APM_DEV, "AUDIO_GET_APM_DEV", NULL },
539     { (uint_t)AUDIO_GET_AS_DEV,  "AUDIO_GET_AS_DEV",  NULL },
540     { (uint_t)AUDIO_MIXER_MULTIPLE_OPEN, "AUDIO_MIXER_MULTIPLE_OPEN", NULL },
541     { (uint_t)AUDIO_MIXER_SINGLE_OPEN, "AUDIO_MIXER_SINGLE_OPEN", NULL },
542     { (uint_t)AUDIO_MIXER_GET_SAMPLE_RATES, "AUDIO_MIXER_GET_SAMPLE_RATES", NULL },
543     { (uint_t)AUDIO_MIXERCTL_GETINFO, "AUDIO_MIXERCTL_GETINFO", NULL },
544     { (uint_t)AUDIO_MIXERCTL_SETINFO, "AUDIO_MIXERCTL_SETINFO", NULL },
545     { (uint_t)AUDIO_MIXERCTL_GET_CHINFO, "AUDIO_MIXERCTL_GET_CHINFO", NULL },
546     { (uint_t)AUDIO_MIXERCTL_SET_CHINFO, "AUDIO_MIXERCTL_SET_CHINFO", NULL },
547     { (uint_t)AUDIO_MIXERCTL_GET_MODE, "AUDIO_MIXERCTL_GET_MODE", NULL },
548     { (uint_t)AUDIO_MIXERCTL_SET_MODE, "AUDIO_MIXERCTL_SET_MODE", NULL },
549     /* new style Boomer (OSS) ioctls */
550     { (uint_t)SNDCCTL_SYSINFO,    "SNDCCTL_SYSINFO",    NULL },
551     { (uint_t)SNDCCTL_AUDIOINFO,  "SNDCCTL_AUDIOINFO",  NULL },
552     { (uint_t)SNDCCTL_AUDIOINFO_EX, "SNDCCTL_AUDIOINFO_EX", NULL },
553     { (uint_t)SNDCCTL_MIXERINFO,  "SNDCCTL_MIXERINFO",  NULL },
554     { (uint_t)SNDCCTL_CARDINFO,   "SNDCCTL_CARDINFO",   NULL },
555     { (uint_t)SNDCCTL_ENGINEINFO, "SNDCCTL_ENGINEINFO", NULL },
556     { (uint_t)SNDCCTL_MIX_NRMIX,  "SNDCCTL_MIX_NRMIX",  NULL },
557     { (uint_t)SNDCCTL_MIX_NREXT,  "SNDCCTL_MIX_NREXT",  NULL },
558     { (uint_t)SNDCCTL_MIX_EXTINFO, "SNDCCTL_MIX_EXTINFO", NULL },
559     { (uint_t)SNDCCTL_MIX_READ,   "SNDCCTL_MIX_READ",   NULL },
560     { (uint_t)SNDCCTL_MIX_WRITE,  "SNDCCTL_MIX_WRITE",  NULL },
561     { (uint_t)SNDCCTL_MIX_ENUMINFO, "SNDCCTL_MIX_ENUMINFO", NULL },
562     { (uint_t)SNDCCTL_MIX_DESCRIPTION, "SNDCCTL_MIX_DESCRIPTION", NULL },
563     { (uint_t)SNDCCTL_SETSONG,    "SNDCCTL_SETSONG",    NULL },
564     { (uint_t)SNDCCTL_GETSONG,    "SNDCCTL_GETSONG",    NULL },
565     { (uint_t)SNDCCTL_SETNAME,    "SNDCCTL_SETNAME",    NULL },
566     { (uint_t)SNDCCTL_SETLABEL,   "SNDCCTL_SETLABEL",   NULL },
567     { (uint_t)SNDCCTL_GETLABEL,   "SNDCCTL_GETLABEL",   NULL },
568     { (uint_t)SNDCCTL_DSP_HALT,   "SNDCCTL_DSP_HALT",   NULL },
569     { (uint_t)SNDCCTL_DSP_RESET,  "SNDCCTL_DSP_RESET",  NULL },
570     { (uint_t)SNDCCTL_DSP_SYNC,   "SNDCCTL_DSP_SYNC",   NULL },
571     { (uint_t)SNDCCTL_DSP_SPEED,  "SNDCCTL_DSP_SPEED",  NULL },

```

```

582 { (uint_t)SNDCTL_DSP_STEREO, "SNDCTL_DSP_STEREO", NULL },
583 { (uint_t)SNDCTL_DSP_GETBLKSIZE, "SNDCTL_DSP_GETBLKSIZE",
584 NULL },
585 { (uint_t)SNDCTL_DSP_SAMPLESIZE, "SNDCTL_DSP_SAMPLESIZE",
586 NULL },
587 { (uint_t)SNDCTL_DSP_CHANNELS, "SNDCTL_DSP_CHANNELS", NULL },
588 { (uint_t)SNDCTL_DSP_POST, "SNDCTL_DSP_POST", NULL },
589 { (uint_t)SNDCTL_DSP_SUBDIVIDE, "SNDCTL_DSP_SUBDIVIDE", NULL },
590 { (uint_t)SNDCTL_DSP_SETFRAGMENT, "SNDCTL_DSP_SETFRAGMENT",
591 NULL },
592 { (uint_t)SNDCTL_DSP_GETFMTS, "SNDCTL_DSP_GETFMTS", NULL },
593 { (uint_t)SNDCTL_DSP_SETFMT, "SNDCTL_DSP_SETFMT", NULL },
594 { (uint_t)SNDCTL_DSP_GETOSPACE, "SNDCTL_DSP_GETOSPACE", NULL },
595 { (uint_t)SNDCTL_DSP_GETISPACE, "SNDCTL_DSP_GETISPACE", NULL },
596 { (uint_t)SNDCTL_DSP_GETCAPS, "SNDCTL_DSP_CAPS", NULL },
597 { (uint_t)SNDCTL_DSP_GETTRIGGER, "SNDCTL_DSP_GETTRIGGER",
598 NULL },
599 { (uint_t)SNDCTL_DSP_SETTRIGGER, "SNDCTL_DSP_SETTRIGGER",
600 NULL },
601 { (uint_t)SNDCTL_DSP_GETIPTR, "SNDCTL_DSP_GETIPTR", NULL },
602 { (uint_t)SNDCTL_DSP_GETOPTH, "SNDCTL_DSP_GETOPTH", NULL },
603 { (uint_t)SNDCTL_DSP_SETSYNCR, "SNDCTL_DSP_SETSYNCR", NULL },
604 { (uint_t)SNDCTL_DSP_SETDUPLEX, "SNDCTL_DSP_SETDUPLEX", NULL },
605 { (uint_t)SNDCTL_DSP_PROFILE, "SNDCTL_DSP_PROFILE", NULL },
606 { (uint_t)SNDCTL_DSP_GETODELAY, "SNDCTL_DSP_GETODELAY", NULL },
607 { (uint_t)SNDCTL_DSP_GETPLAYVOL, "SNDCTL_DSP_GETPLAYVOL",
608 NULL },
609 { (uint_t)SNDCTL_DSP_SETPLAYVOL, "SNDCTL_DSP_SETPLAYVOL",
610 NULL },
611 { (uint_t)SNDCTL_DSP_GETERROR, "SNDCTL_DSP_GETERROR", NULL },
612 { (uint_t)SNDCTL_DSP_READCTL, "SNDCTL_DSP_READCTL", NULL },
613 { (uint_t)SNDCTL_DSP_WRITECTL, "SNDCTL_DSP_WRITECTL", NULL },
614 { (uint_t)SNDCTL_DSP_SYNCGROUP, "SNDCTL_DSP_SYNCGROUP", NULL },
615 { (uint_t)SNDCTL_DSP_SYNCSTART, "SNDCTL_DSP_SYNCSTART", NULL },
616 { (uint_t)SNDCTL_DSP_COOKEDMODE, "SNDCTL_DSP_COOKEDMODE",
617 NULL },
618 { (uint_t)SNDCTL_DSP_SILENCE, "SNDCTL_DSP_SILENCE", NULL },
619 { (uint_t)SNDCTL_DSP_SKIP, "SNDCTL_DSP_SKIP", NULL },
620 { (uint_t)SNDCTL_DSP_HALT_INPUT, "SNDCTL_DSP_HALT_INPUT",
621 NULL },
622 { (uint_t)SNDCTL_DSP_HALT_OUTPUT, "SNDCTL_DSP_HALT_OUTPUT",
623 NULL },
624 { (uint_t)SNDCTL_DSP_LOW_WATER, "SNDCTL_DSP_LOW_WATER", NULL },
625 { (uint_t)SNDCTL_DSP_CURRENT_OPTR, "SNDCTL_DSP_CURRENT_OPTR",
626 NULL },
627 { (uint_t)SNDCTL_DSP_CURRENT_IPTR, "SNDCTL_DSP_CURRENT_IPTR",
628 NULL },
629 { (uint_t)SNDCTL_DSP_GET_RECSRC_NAMES, "SNDCTL_DSP_GET_RECSRC_NAMES",
630 NULL },
631 { (uint_t)SNDCTL_DSP_GET_RECSRC, "SNDCTL_DSP_GET_RECSRC",
632 NULL },
633 { (uint_t)SNDCTL_DSP_SET_RECSRC, "SNDCTL_DSP_SET_RECSRC",
634 NULL },
635 { (uint_t)SNDCTL_DSP_GET_PLAYTGT_NAMES, "SNDCTL_DSP_GET_PLAYTGT_NAMES",
636 NULL },
637 { (uint_t)SNDCTL_DSP_GET_PLAYTGT, "SNDCTL_DSP_GET_PLAYTGT",
638 NULL },
639 { (uint_t)SNDCTL_DSP_SET_PLAYTGT, "SNDCTL_DSP_SET_PLAYTGT",
640 NULL },
641 { (uint_t)SNDCTL_DSP_GETRECVOL, "SNDCTL_DSP_GETRECVOL",
642 NULL },
643 { (uint_t)SNDCTL_DSP_SETRECVOL, "SNDCTL_DSP_SETRECVOL",
644 NULL },
645 { (uint_t)SNDCTL_DSP_GET_CHNORDER, "SNDCTL_DSP_GET_CHNORDER",
646 NULL },
647 { (uint_t)SNDCTL_DSP_SET_CHNORDER, "SNDCTL_DSP_SET_CHNORDER",

```

```

648 NULL },
649 { (uint_t)SNDCTL_DSP_GETIPEAKS, "SNDCTL_DSP_GETIPEAKS", NULL },
650 { (uint_t)SNDCTL_DSP_GETOPEAKS, "SNDCTL_DSP_GETOPEAKS", NULL },
651 { (uint_t)SNDCTL_DSP_POLICY, "SNDCTL_DSP_POLICY", NULL },
652 { (uint_t)SNDCTL_DSP_GETCHANNELMASK, "SNDCTL_DSP_GETCHANNELMASK",
653 NULL },
654 { (uint_t)SNDCTL_DSP_BIND_CHANNEL, "SNDCTL_DSP_BIND_CHANNEL",
655 NULL },
656 { (uint_t)SOUND_MIXER_READ_VOLUME, "SOUND_MIXER_READ_VOLUME",
657 NULL },
658 { (uint_t)SOUND_MIXER_READ_OGAIN, "SOUND_MIXER_READ_OGAIN",
659 NULL },
660 { (uint_t)SOUND_MIXER_READ_PCM, "SOUND_MIXER_READ_PCM", NULL },
661 { (uint_t)SOUND_MIXER_READ_IGAIN, "SOUND_MIXER_READ_IGAIN",
662 NULL },
663 { (uint_t)SOUND_MIXER_READ_RECLEV, "SOUND_MIXER_READ_RECLEV",
664 NULL },
665 { (uint_t)SOUND_MIXER_READ_RECSRC, "SOUND_MIXER_READ_RECSRC",
666 NULL },
667 { (uint_t)SOUND_MIXER_READ_DEVMASK, "SOUND_MIXER_READ_DEVMASK",
668 NULL },
669 { (uint_t)SOUND_MIXER_READ_RECMAK, "SOUND_MIXER_READ_RECMAK",
670 NULL },
671 { (uint_t)SOUND_MIXER_READ_CAPS, "SOUND_MIXER_READ_CAPS",
672 NULL },
673 { (uint_t)SOUND_MIXER_READ_STEREODEVS, "SOUND_MIXER_READ_STEREODEVS",
674 NULL },
675 { (uint_t)SOUND_MIXER_READ_RECGAIN, "SOUND_MIXER_READ_RECGAIN",
676 NULL },
677 { (uint_t)SOUND_MIXER_READ_MONGAIN, "SOUND_MIXER_READ_MONGAIN",
678 NULL },
679 { (uint_t)SOUND_MIXER_WRITE_VOLUME, "SOUND_MIXER_WRITE_VOLUME",
680 NULL },
681 { (uint_t)SOUND_MIXER_WRITE_OGAIN, "SOUND_MIXER_WRITE_OGAIN",
682 NULL },
683 { (uint_t)SOUND_MIXER_WRITE_PCM, "SOUND_MIXER_WRITE_PCM",
684 NULL },
685 { (uint_t)SOUND_MIXER_WRITE_IGAIN, "SOUND_MIXER_WRITE_IGAIN",
686 NULL },
687 { (uint_t)SOUND_MIXER_WRITE_RECLEV, "SOUND_MIXER_WRITE_RECLEV",
688 NULL },
689 { (uint_t)SOUND_MIXER_WRITE_RECSRC, "SOUND_MIXER_WRITE_RECSRC",
690 NULL },
691 { (uint_t)SOUND_MIXER_WRITE_RECGAIN, "SOUND_MIXER_WRITE_RECGAIN",
692 NULL },
693 { (uint_t)SOUND_MIXER_WRITE_MONGAIN, "SOUND_MIXER_WRITE_MONGAIN",
694 NULL },
695
696 /* STREAMS redirection ioctls */
697 { (uint_t)SRIOCREDIR, "SRIOCREDIR", NULL },
698 { (uint_t)SRIOCISREDIR, "SRIOCISREDIR", NULL },
699 { (uint_t)CPCIO_BIND, "CPCIO_BIND", NULL },
700 { (uint_t)CPCIO_SAMPLE, "CPCIO_SAMPLE", NULL },
701 { (uint_t)CPCIO_RELE, "CPCIO_RELE", NULL },
702 /* /dev/poll ioctl() control codes */
703 { (uint_t)DP_POLL, "DP_POLL", NULL },
704 { (uint_t)DP_ISPOLLED, "DP_ISPOLLED", NULL },
705 /* the old /proc ioctl() control codes */
706 #define PIOC ('q' << 8)
707 { (uint_t)(PIOC 1), "PIOCSTATUS", NULL },
708 { (uint_t)(PIOC 2), "PIOCSTOP", NULL },
709 { (uint_t)(PIOC 3), "PIOCWSTOP", NULL },
710 { (uint_t)(PIOC 4), "PIOCRUN", NULL },
711 { (uint_t)(PIOC 5), "PIOCTRACE", NULL },
712 { (uint_t)(PIOC 6), "PIOCTRACE", NULL },
713 { (uint_t)(PIOC 7), "PIOCSSIG", NULL },

```

```

714 (uint_t)(PIOC 8), "PIOCKILL", NULL },
715 (uint_t)(PIOC 9), "PIOCUNKILL", NULL },
716 (uint_t)(PIOC 10), "PIOCGHOLD", NULL },
717 (uint_t)(PIOC 11), "PIOCSSHOLD", NULL },
718 (uint_t)(PIOC 12), "PIOCMAXSIG", NULL },
719 (uint_t)(PIOC 13), "PIOCACTION", NULL },
720 (uint_t)(PIOC 14), "PIOCGFAULT", NULL },
721 (uint_t)(PIOC 15), "PIOCSFAULT", NULL },
722 (uint_t)(PIOC 16), "PIOCCFAULT", NULL },
723 (uint_t)(PIOC 17), "PIOCGENTRY", NULL },
724 (uint_t)(PIOC 18), "PIOCSENTRY", NULL },
725 (uint_t)(PIOC 19), "PIOCGEXIT", NULL },
726 (uint_t)(PIOC 20), "PIOCSEXIT", NULL },
727 (uint_t)(PIOC 21), "PIOCSFORK", NULL },
728 (uint_t)(PIOC 22), "PIOCRFORK", NULL },
729 (uint_t)(PIOC 23), "PIOCSRLC", NULL },
730 (uint_t)(PIOC 24), "PIOCRRLC", NULL },
731 (uint_t)(PIOC 25), "PIOCGREG", NULL },
732 (uint_t)(PIOC 26), "PIOCSREG", NULL },
733 (uint_t)(PIOC 27), "PIOCGFPREG", NULL },
734 (uint_t)(PIOC 28), "PIOCSFPREG", NULL },
735 (uint_t)(PIOC 29), "PIOCNICE", NULL },
736 (uint_t)(PIOC 30), "PIOCPSINFO", NULL },
737 (uint_t)(PIOC 31), "PIOCNMAP", NULL },
738 (uint_t)(PIOC 32), "PIOCMAP", NULL },
739 (uint_t)(PIOC 33), "PIOCOPENM", NULL },
740 (uint_t)(PIOC 34), "PIOCCRED", NULL },
741 (uint_t)(PIOC 35), "PIOCGROUPS", NULL },
742 (uint_t)(PIOC 36), "PIOCGETPR", NULL },
743 (uint_t)(PIOC 37), "PIOCGETU", NULL },
744 (uint_t)(PIOC 38), "PIOCSET", NULL },
745 (uint_t)(PIOC 39), "PIOCRESET", NULL },
746 (uint_t)(PIOC 43), "PIOCUSAGE", NULL },
747 (uint_t)(PIOC 44), "PIOCOPENPD", NULL },
748 (uint_t)(PIOC 45), "PIOCLWPIDS", NULL },
749 (uint_t)(PIOC 46), "PIOCOPENLWP", NULL },
750 (uint_t)(PIOC 47), "PIOCLSTATUS", NULL },
751 (uint_t)(PIOC 48), "PIOCLUSAGE", NULL },
752 (uint_t)(PIOC 49), "PIOCNAUXV", NULL },
753 (uint_t)(PIOC 50), "PIOCAUXV", NULL },
754 (uint_t)(PIOC 51), "PIOCGXREGSIZE", NULL },
755 (uint_t)(PIOC 52), "PIOCGXREG", NULL },
756 (uint_t)(PIOC 53), "PIOCSXREG", NULL },
757 (uint_t)(PIOC 101), "PIOCGWIN", NULL },
758 (uint_t)(PIOC 103), "PIOCNLDT", NULL },
759 (uint_t)(PIOC 104), "PIOCLDT", NULL },

761 /* ioctl's applicable on sockets */
762 (uint_t)SIOCShiwat, "SIOCShiwat", NULL },
763 (uint_t)SIOCGhiwat, "SIOCGhiwat", NULL },
764 (uint_t)SIOCSlowat, "SIOCSlowat", NULL },
765 (uint_t)SIOCGlowat, "SIOCGlowat", NULL },
766 (uint_t)SIOCAtmark, "SIOCAtmark", NULL },
767 (uint_t)SIOCSgrp, "SIOCSgrp", NULL },
768 (uint_t)SIOCGgrp, "SIOCGgrp", NULL },
769 (uint_t)SIOCAdrt, "SIOCAdrt", "rtentry" },
770 (uint_t)SIOCDELRT, "SIOCDELRT", "rtentry" },
771 (uint_t)SIOCGETVIFCNT, "SIOCGETVIFCNT", "sioc_vif_req" },
772 (uint_t)SIOCGETSGCNT, "SIOCGETSGCNT", "sioc_sg_req" },
773 (uint_t)SIOCGETLSGCNT, "SIOCGETLSGCNT", "sioc_lsg_req" },
774 (uint_t)SIOCSifaddr, "SIOCSifaddr", "ifreq" },
775 (uint_t)SIOCGifaddr, "SIOCGifaddr", "ifreq" },
776 (uint_t)SIOCSifdstaddr, "SIOCSifdstaddr", "ifreq" },
777 (uint_t)SIOCGifdstaddr, "SIOCGifdstaddr", "ifreq" },
778 (uint_t)SIOCSifflags, "SIOCSifflags", "ifreq" },
779 (uint_t)SIOCGifflags, "SIOCGifflags", "ifreq" },

```

```

780 (uint_t)SIOCSIFMEM, "SIOCSIFMEM", "ifreq" },
781 (uint_t)SIOCGIFMEM, "SIOCGIFMEM", "ifreq" },
782 (uint_t)SIOCGIFCONF, "SIOCGIFCONF", "ifconf" },
783 (uint_t)SIOCSIFMTU, "SIOCSIFMTU", "ifreq" },
784 (uint_t)SIOCGIFMTU, "SIOCGIFMTU", "ifreq" },
785 (uint_t)SIOCGIFBRDADDR, "SIOCGIFBRDADDR", "ifreq" },
786 (uint_t)SIOCSIFBRDADDR, "SIOCSIFBRDADDR", "ifreq" },
787 (uint_t)SIOCGIFNETMASK, "SIOCGIFNETMASK", "ifreq" },
788 (uint_t)SIOCSIFNETMASK, "SIOCSIFNETMASK", "ifreq" },
789 (uint_t)SIOCGIFMETRIC, "SIOCGIFMETRIC", "ifreq" },
790 (uint_t)SIOCSIFMETRIC, "SIOCSIFMETRIC", "ifreq" },
791 (uint_t)SIOCSARP, "SIOCSARP", "arpreq" },
792 (uint_t)SIOCGARP, "SIOCGARP", "arpreq" },
793 (uint_t)SIOCDAEP, "SIOCDAEP", "arpreq" },
794 (uint_t)SIOCUPPER, "SIOCUPPER", "ifreq" },
795 (uint_t)SIOCLOWER, "SIOCLOWER", "ifreq" },
796 (uint_t)SIOCSETSYPNC, "SIOCSETSYPNC", "ifreq" },
797 (uint_t)SIOCGSETSYPNC, "SIOCGSETSYPNC", "ifreq" },
798 (uint_t)SIOCSSDSTATS, "SIOCSSDSTATS", "ifreq" },
799 (uint_t)SIOCSSESTATS, "SIOCSSESTATS", "ifreq" },
800 (uint_t)SIOCSPROMISC, "SIOCSPROMISC", NULL },
801 (uint_t)SIOCADDMULTI, "SIOCADDMULTI", "ifreq" },
802 (uint_t)SIOCDELMULTI, "SIOCDELMULTI", "ifreq" },
803 (uint_t)SIOCGETNAME, "SIOCGETNAME", "sockaddr" },
804 (uint_t)SIOCGETPEER, "SIOCGETPEER", "sockaddr" },
805 (uint_t)IF_UNITSEL, "IF_UNITSEL", NULL },
806 (uint_t)SIOCXPROTO, "SIOCXPROTO", NULL },
807 (uint_t)SIOCIFDETACH, "SIOCIFDETACH", "ifreq" },
808 (uint_t)SIOCGENPSTATS, "SIOCGENPSTATS", "ifreq" },
809 (uint_t)SIOCX25XMT, "SIOCX25XMT", "ifreq" },
810 (uint_t)SIOCX25RCV, "SIOCX25RCV", "ifreq" },
811 (uint_t)SIOCX25TBL, "SIOCX25TBL", "ifreq" },
812 (uint_t)SIOCSLGETREQ, "SIOCSLGETREQ", "ifreq" },
813 (uint_t)SIOCSLSTAT, "SIOCSLSTAT", "ifreq" },
814 (uint_t)SIOCSIFNAME, "SIOCSIFNAME", "ifreq" },
815 (uint_t)SIOCGENADDR, "SIOCGENADDR", "ifreq" },
816 (uint_t)SIOCGIFNUM, "SIOCGIFNUM", NULL },
817 (uint_t)SIOCGIFMUXID, "SIOCGIFMUXID", "ifreq" },
818 (uint_t)SIOCSIFMUXID, "SIOCSIFMUXID", "ifreq" },
819 (uint_t)SIOCGIFINDEX, "SIOCGIFINDEX", "ifreq" },
820 (uint_t)SIOCSIFINDEX, "SIOCSIFINDEX", "ifreq" },
821 (uint_t)SIOCLIFREMOVEIF, "SIOCLIFREMOVEIF", "lifreq" },
822 (uint_t)SIOCLIFADDIF, "SIOCLIFADDIF", "lifreq" },
823 (uint_t)SIOCSLIFADDR, "SIOCSLIFADDR", "lifreq" },
824 (uint_t)SIOCGLIFADDR, "SIOCGLIFADDR", "lifreq" },
825 (uint_t)SIOCSLIFDSTADDR, "SIOCSLIFDSTADDR", "lifreq" },
826 (uint_t)SIOCGLIFDSTADDR, "SIOCGLIFDSTADDR", "lifreq" },
827 (uint_t)SIOCSLIFFLAGS, "SIOCSLIFFLAGS", "lifreq" },
828 (uint_t)SIOCGLIFFLAGS, "SIOCGLIFFLAGS", "lifreq" },
829 (uint_t)SIOCSLIFCONF, "SIOCSLIFCONF", "lifconf" },
830 (uint_t)SIOCSLIFMTU, "SIOCSLIFMTU", "lifreq" },
831 (uint_t)SIOCGLIFMTU, "SIOCGLIFMTU", "lifreq" },
832 (uint_t)SIOCGLIFBRDADDR, "SIOCGLIFBRDADDR", "lifreq" },
833 (uint_t)SIOCSLIFBRDADDR, "SIOCSLIFBRDADDR", "lifreq" },
834 (uint_t)SIOCSLIFNETMASK, "SIOCSLIFNETMASK", "lifreq" },
835 (uint_t)SIOCSLIFNETMASK, "SIOCSLIFNETMASK", "lifreq" },
836 (uint_t)SIOCSLIFMETRIC, "SIOCSLIFMETRIC", "lifreq" },
837 (uint_t)SIOCSLIFMETRIC, "SIOCSLIFMETRIC", "lifreq" },
838 (uint_t)SIOCSLIFNAME, "SIOCSLIFNAME", "lifreq" },
839 (uint_t)SIOCGLIFNUM, "SIOCGLIFNUM", "lifnum" },
840 (uint_t)SIOCGLIFMUXID, "SIOCGLIFMUXID", "lifreq" },
841 (uint_t)SIOCSLIFMUXID, "SIOCSLIFMUXID", "lifreq" },
842 (uint_t)SIOCGLIFINDEX, "SIOCGLIFINDEX", "lifreq" },
843 (uint_t)SIOCSLIFINDEX, "SIOCSLIFINDEX", "lifreq" },
844 (uint_t)SIOCSLIFTOKEN, "SIOCSLIFTOKEN", "lifreq" },
845 (uint_t)SIOCGLIFTOKEN, "SIOCGLIFTOKEN", "lifreq" },

```

```

846 { (uint_t)SIOCSLIFSUBNET, "SIOCSLIFSUBNET", "lifreq" },
847 { (uint_t)SIOCLIFSUBNET, "SIOCLIFSUBNET", "lifreq" },
848 { (uint_t)SIOCSLIFLNKINFO, "SIOCSLIFLNKINFO", "lifreq" },
849 { (uint_t)SIOCLIFLNKINFO, "SIOCLIFLNKINFO", "lifreq" },
850 { (uint_t)SIOCLIFDELND, "SIOCLIFDELND", "lifreq" },
851 { (uint_t)SIOCLIFGETND, "SIOCLIFGETND", "lifreq" },
852 { (uint_t)SIOCLIFSETND, "SIOCLIFSETND", "lifreq" },
853 { (uint_t)SIOCTMYADDR, "SIOCTMYADDR", "sioc_addrreq" },
854 { (uint_t)SIOCTONLINK, "SIOCTONLINK", "sioc_addrreq" },
855 { (uint_t)SIOCTMYSITE, "SIOCTMYSITE", "sioc_addrreq" },
856 { (uint_t)SIOCFIPSECONFIG, "SIOCFIPSECONFIG", NULL },
857 { (uint_t)SIOCSIPSECONFIG, "SIOCSIPSECONFIG", NULL },
858 { (uint_t)SIOCDIPSECONFIG, "SIOCDIPSECONFIG", NULL },
859 { (uint_t)SIOCLIPSECONFIG, "SIOCLIPSECONFIG", NULL },
860 { (uint_t)SIOCGLIFBINDING, "SIOCGLIFBINDING", "lifreq" },
861 { (uint_t)SIOCSLIFGROUPNAME, "SIOCSLIFGROUPNAME", "lifreq" },
862 { (uint_t)SIOCLIFGROUPNAME, "SIOCLIFGROUPNAME", "lifreq" },
863 { (uint_t)SIOCGLIFGROUPINFO, "SIOCGLIFGROUPINFO", "lifgroupinfo" },
864 { (uint_t)SIOCGDSTINFO, "SIOCGDSTINFO", NULL },
865 { (uint_t)SIOCGIP6ADDRPOLICY, "SIOCGIP6ADDRPOLICY", NULL },
866 { (uint_t)SIOCSIP6ADDRPOLICY, "SIOCSIP6ADDRPOLICY", NULL },
867 { (uint_t)SIOCSXARP, "SIOCSXARP", "xarpreq" },
868 { (uint_t)SIOCGXARP, "SIOCGXARP", "xarpreq" },
869 { (uint_t)SIOCDXARP, "SIOCDXARP", "xarpreq" },
870 { (uint_t)SIOCLIFZONE, "SIOCLIFZONE", "lifreq" },
871 { (uint_t)SIOCSLIFZONE, "SIOCSLIFZONE", "lifreq" },
872 { (uint_t)SIOCSCTPSOFT, "SIOCSCTPSOFT", NULL },
873 { (uint_t)SIOCSCTPGOPT, "SIOCSCTPGOPT", NULL },
874 { (uint_t)SIOCSCTPPEELOFF, "SIOCSCTPPEELOFF", "int" },
875 { (uint_t)SIOCLIFUSESRC, "SIOCLIFUSESRC", "lifreq" },
876 { (uint_t)SIOCSLIFUSESRC, "SIOCSLIFUSESRC", "lifreq" },
877 { (uint_t)SIOCLIFSRCOF, "SIOCLIFSRCOF", "lifsrcof" },
878 { (uint_t)SIOCGMSFILTER, "SIOCGMSFILTER", "group filter" },
879 { (uint_t)SIOCSMSFILTER, "SIOCSMSFILTER", "group filter" },
880 { (uint_t)SIOCGIPMSFILTER, "SIOCGIPMSFILTER", "ip_msfilter" },
881 { (uint_t)SIOCSIPMSFILTER, "SIOCSIPMSFILTER", "ip_msfilter" },
882 { (uint_t)SIOCLIFPADSTATE, "SIOCLIFPADSTATE", "lifreq" },
883 { (uint_t)SIOCSLIFPREFIX, "SIOCSLIFPREFIX", "lifreq" },
884 { (uint_t)SIOCGSTAMP, "SIOCGSTAMP", "timeval" },
885 { (uint_t)SIOCGIFHWADDR, "SIOCGIFHWADDR", "ifreq" },
886 { (uint_t)SIOCLIFHWADDR, "SIOCLIFHWADDR", "lifreq" },

888 /* DES encryption */
889 { (uint_t)DESIOCBLOCK, "DESIOCBLOCK", "desparams" },
890 { (uint_t)DESIOCQUICK, "DESIOCQUICK", "desparams" },

892 /* Printing system */
893 { (uint_t)PRNIOC_GET_IFCAP, "PRNIOC_GET_IFCAP", NULL },
894 { (uint_t)PRNIOC_SET_IFCAP, "PRNIOC_SET_IFCAP", NULL },
895 { (uint_t)PRNIOC_GET_IFINFO, "PRNIOC_GET_IFINFO",
896 "prn_interface_info" },
897 { (uint_t)PRNIOC_GET_STATUS, "PRNIOC_GET_STATUS", NULL },
898 { (uint_t)PRNIOC_GET_1284_DEVID, "PRNIOC_GET_1284_DEVID",
899 "prn_1284_device_id" },
900 { (uint_t)PRNIOC_GET_1284_STATUS, "PRNIOC_GET_1284_STATUS",
901 "PRNIOC_GET_IFCANIOC_GET_1284_STATUS", NULL },
902 { (uint_t)PRNIOC_GET_TIMEOUTS, "PRNIOC_GET_TIMEOUTS",
903 "prn_timeouts" },
904 { (uint_t)PRNIOC_SET_TIMEOUTS, "PRNIOC_SET_TIMEOUTS",
905 "prn_timeouts" },
906 { (uint_t)PRNIOC_RESET, "PRNIOC_RESET", NULL },

908 /* DTrace */
909 { (uint_t)DTRACEIOC_PROVIDER, "DTRACEIOC_PROVIDER", NULL },
910 { (uint_t)DTRACEIOC_PROBES, "DTRACEIOC_PROBES", NULL },
911 { (uint_t)DTRACEIOC_BUFSNAP, "DTRACEIOC_BUFSNAP", NULL },

```

```

912 { (uint_t)DTRACEIOC_PROBEMATCH, "DTRACEIOC_PROBEMATCH", NULL },
913 { (uint_t)DTRACEIOC_ENABLE, "DTRACEIOC_ENABLE", NULL },
914 { (uint_t)DTRACEIOC_AGGSNAP, "DTRACEIOC_AGGSNAP", NULL },
915 { (uint_t)DTRACEIOC_EPROBE, "DTRACEIOC_EPROBE", NULL },
916 { (uint_t)DTRACEIOC_PROBEARG, "DTRACEIOC_PROBEARG", NULL },
917 { (uint_t)DTRACEIOC_CONF, "DTRACEIOC_CONF", NULL },
918 { (uint_t)DTRACEIOC_STATUS, "DTRACEIOC_STATUS", NULL },
919 { (uint_t)DTRACEIOC_GO, "DTRACEIOC_GO", NULL },
920 { (uint_t)DTRACEIOC_STOP, "DTRACEIOC_STOP", NULL },
921 { (uint_t)DTRACEIOC_AGDESC, "DTRACEIOC_AGDESC", NULL },
922 { (uint_t)DTRACEIOC_FORMAT, "DTRACEIOC_FORMAT", NULL },
923 { (uint_t)DTRACEIOC_DOFGET, "DTRACEIOC_DOFGET", NULL },
924 { (uint_t)DTRACEIOC_REPLICATE, "DTRACEIOC_REPLICATE", NULL },

926 { (uint_t)DTRACEHIOC_ADD, "DTRACEHIOC_ADD", NULL },
927 { (uint_t)DTRACEHIOC_REMOVE, "DTRACEHIOC_REMOVE", NULL },
928 { (uint_t)DTRACEHIOC_ADDDOF, "DTRACEHIOC_ADDDOF", NULL },

930 /* /dev/cryptoadm ioctl() control codes */
931 { (uint_t)CRYPTO_GET_VERSION, "CRYPTO_GET_VERSION", NULL },
932 { (uint_t)CRYPTO_GET_DEV_LIST, "CRYPTO_GET_DEV_LIST", NULL },
933 { (uint_t)CRYPTO_GET_SOFT_LIST, "CRYPTO_GET_SOFT_LIST", NULL },
934 { (uint_t)CRYPTO_GET_DEV_INFO, "CRYPTO_GET_DEV_INFO", NULL },
935 { (uint_t)CRYPTO_GET_SOFT_INFO, "CRYPTO_GET_SOFT_INFO", NULL },
936 { (uint_t)CRYPTO_LOAD_DEV_DISABLED, "CRYPTO_LOAD_DEV_DISABLED",
937 NULL },
938 { (uint_t)CRYPTO_LOAD_SOFT_DISABLED, "CRYPTO_LOAD_SOFT_DISABLED",
939 NULL },
940 { (uint_t)CRYPTO_UNLOAD_SOFT_MODULE, "CRYPTO_UNLOAD_SOFT_MODULE",
941 NULL },
942 { (uint_t)CRYPTO_LOAD_SOFT_CONFIG, "CRYPTO_LOAD_SOFT_CONFIG",
943 NULL },
944 { (uint_t)CRYPTO_POOL_CREATE, "CRYPTO_POOL_CREATE", NULL },
945 { (uint_t)CRYPTO_POOL_WAIT, "CRYPTO_POOL_WAIT", NULL },
946 { (uint_t)CRYPTO_POOL_RUN, "CRYPTO_POOL_RUN", NULL },
947 { (uint_t)CRYPTO_LOAD_DOOR, "CRYPTO_LOAD_DOOR", NULL },

949 /* /dev/crypto ioctl() control codes */
950 { (uint_t)CRYPTO_GET_FUNCTION_LIST, "CRYPTO_GET_FUNCTION_LIST",
951 NULL },
952 { (uint_t)CRYPTO_GET_MECHANISM_NUMBER, "CRYPTO_GET_MECHANISM_NUMBER",
953 NULL },
954 { (uint_t)CRYPTO_OPEN_SESSION, "CRYPTO_OPEN_SESSION", NULL },
955 { (uint_t)CRYPTO_CLOSE_SESSION, "CRYPTO_CLOSE_SESSION", NULL },
956 { (uint_t)CRYPTO_CLOSE_ALL_SESSIONS, "CRYPTO_CLOSE_ALL_SESSIONS",
957 NULL },
958 { (uint_t)CRYPTO_LOGIN, "CRYPTO_LOGIN", NULL },
959 { (uint_t)CRYPTO_LOGOUT, "CRYPTO_LOGOUT", NULL },
960 { (uint_t)CRYPTO_ENCRYPT, "CRYPTO_ENCRYPT", NULL },
961 { (uint_t)CRYPTO_ENCRYPT_INIT, "CRYPTO_ENCRYPT_INIT", NULL },
962 { (uint_t)CRYPTO_ENCRYPT_UPDATE, "CRYPTO_ENCRYPT_UPDATE",
963 NULL },
964 { (uint_t)CRYPTO_ENCRYPT_FINAL, "CRYPTO_ENCRYPT_FINAL", NULL },
965 { (uint_t)CRYPTO_DECRYPT, "CRYPTO_DECRYPT", NULL },
966 { (uint_t)CRYPTO_DECRYPT_INIT, "CRYPTO_DECRYPT_INIT", NULL },
967 { (uint_t)CRYPTO_DECRYPT_UPDATE, "CRYPTO_DECRYPT_UPDATE",
968 NULL },
969 { (uint_t)CRYPTO_DECRYPT_FINAL, "CRYPTO_DECRYPT_FINAL", NULL },
970 { (uint_t)CRYPTO_DIGEST, "CRYPTO_DIGEST", NULL },
971 { (uint_t)CRYPTO_DIGEST_INIT, "CRYPTO_DIGEST_INIT", NULL },
972 { (uint_t)CRYPTO_DIGEST_UPDATE, "CRYPTO_DIGEST_UPDATE", NULL },
973 { (uint_t)CRYPTO_DIGEST_KEY, "CRYPTO_DIGEST_KEY", NULL },
974 { (uint_t)CRYPTO_DIGEST_FINAL, "CRYPTO_DIGEST_FINAL", NULL },
975 { (uint_t)CRYPTO_MAC, "CRYPTO_MAC", NULL },
976 { (uint_t)CRYPTO_MAC_INIT, "CRYPTO_MAC_INIT", NULL },
977 { (uint_t)CRYPTO_MAC_UPDATE, "CRYPTO_MAC_UPDATE", NULL },

```

```

978 { (uint_t)CRYPTO_MAC_FINAL, "CRYPTO_MAC_FINAL", NULL },
979 { (uint_t)CRYPTO_SIGN, "CRYPTO_SIGN", NULL },
980 { (uint_t)CRYPTO_SIGN_INIT, "CRYPTO_SIGN_INIT", NULL },
981 { (uint_t)CRYPTO_SIGN_UPDATE, "CRYPTO_SIGN_UPDATE", NULL },
982 { (uint_t)CRYPTO_SIGN_FINAL, "CRYPTO_SIGN_FINAL", NULL },
983 { (uint_t)CRYPTO_SIGN_RECOVER_INIT, "CRYPTO_SIGN_RECOVER_INIT",
984 NULL },
985 { (uint_t)CRYPTO_SIGN_RECOVER, "CRYPTO_SIGN_RECOVER", NULL },
986 { (uint_t)CRYPTO_VERIFY, "CRYPTO_VERIFY", NULL },
987 { (uint_t)CRYPTO_VERIFY_INIT, "CRYPTO_VERIFY_INIT", NULL },
988 { (uint_t)CRYPTO_VERIFY_UPDATE, "CRYPTO_VERIFY_UPDATE", NULL },
989 { (uint_t)CRYPTO_VERIFY_FINAL, "CRYPTO_VERIFY_FINAL", NULL },
990 { (uint_t)CRYPTO_VERIFY_RECOVER_INIT, "CRYPTO_VERIFY_RECOVER_INIT",
991 NULL },
992 { (uint_t)CRYPTO_VERIFY_RECOVER, "CRYPTO_VERIFY_RECOVER",
993 NULL },
994 { (uint_t)CRYPTO_DIGEST_ENCRYPT_UPDATE, "CRYPTO_DIGEST_ENCRYPT_UPDATE",
995 NULL },
996 { (uint_t)CRYPTO_DECRYPT_DIGEST_UPDATE, "CRYPTO_DECRYPT_DIGEST_UPDATE",
997 NULL },
998 { (uint_t)CRYPTO_SIGN_ENCRYPT_UPDATE, "CRYPTO_SIGN_ENCRYPT_UPDATE",
999 NULL },
1000 { (uint_t)CRYPTO_DECRYPT_VERIFY_UPDATE, "CRYPTO_DECRYPT_VERIFY_UPDATE",
1001 NULL },
1002 { (uint_t)CRYPTO_SEED_RANDOM, "CRYPTO_SEED_RANDOM", NULL },
1003 { (uint_t)CRYPTO_GENERATE_RANDOM, "CRYPTO_GENERATE_RANDOM",
1004 NULL },
1005 { (uint_t)CRYPTO_OBJECT_CREATE, "CRYPTO_OBJECT_CREATE", NULL },
1006 { (uint_t)CRYPTO_OBJECT_COPY, "CRYPTO_OBJECT_COPY", NULL },
1007 { (uint_t)CRYPTO_OBJECT_DESTROY, "CRYPTO_OBJECT_DESTROY",
1008 NULL },
1009 { (uint_t)CRYPTO_OBJECT_GET_ATTRIBUTE_VALUE,
1010 "CRYPTO_OBJECT_GET_ATTRIBUTE_VALUE", NULL },
1011 { (uint_t)CRYPTO_OBJECT_GET_SIZE, "CRYPTO_OBJECT_GET_SIZE", NULL },
1012 { (uint_t)CRYPTO_OBJECT_SET_ATTRIBUTE_VALUE,
1013 "CRYPTO_OBJECT_SET_ATTRIBUTE_VALUE", NULL },
1014 { (uint_t)CRYPTO_OBJECT_FIND_INIT, "CRYPTO_OBJECT_FIND_INIT",
1015 NULL },
1016 { (uint_t)CRYPTO_OBJECT_FIND_UPDATE, "CRYPTO_OBJECT_FIND_UPDATE",
1017 NULL },
1018 { (uint_t)CRYPTO_OBJECT_FIND_FINAL, "CRYPTO_OBJECT_FIND_FINAL",
1019 NULL },
1020 { (uint_t)CRYPTO_GENERATE_KEY, "CRYPTO_GENERATE_KEY", NULL },
1021 { (uint_t)CRYPTO_GENERATE_KEY_PAIR, "CRYPTO_GENERATE_KEY_PAIR",
1022 NULL },
1023 { (uint_t)CRYPTO_WRAP_KEY, "CRYPTO_WRAP_KEY", NULL },
1024 { (uint_t)CRYPTO_UNWRAP_KEY, "CRYPTO_UNWRAP_KEY", NULL },
1025 { (uint_t)CRYPTO_DERIVE_KEY, "CRYPTO_DERIVE_KEY", NULL },
1026 { (uint_t)CRYPTO_GET_PROVIDER_LIST, "CRYPTO_GET_PROVIDER_LIST",
1027 NULL },
1028 { (uint_t)CRYPTO_GET_PROVIDER_INFO, "CRYPTO_GET_PROVIDER_INFO",
1029 NULL },
1030 { (uint_t)CRYPTO_GET_PROVIDER_MECHANISMS,
1031 "CRYPTO_GET_PROVIDER_MECHANISMS", NULL },
1032 { (uint_t)CRYPTO_GET_PROVIDER_MECHANISM_INFO,
1033 "CRYPTO_GET_PROVIDER_MECHANISM_INFO", NULL },
1034 { (uint_t)CRYPTO_INIT_TOKEN, "CRYPTO_INIT_TOKEN", NULL },
1035 { (uint_t)CRYPTO_INIT_PIN, "CRYPTO_INIT_PIN", NULL },
1036 { (uint_t)CRYPTO_SET_PIN, "CRYPTO_SET_PIN", NULL },
1037 { (uint_t)CRYPTO_NOSTORE_GENERATE_KEY,
1038 "CRYPTO_NOSTORE_GENERATE_KEY", NULL },
1039 { (uint_t)CRYPTO_NOSTORE_GENERATE_KEY_PAIR,
1040 "CRYPTO_NOSTORE_GENERATE_KEY_PAIR", NULL },
1041 { (uint_t)CRYPTO_NOSTORE_DERIVE_KEY,
1042 "CRYPTO_NOSTORE_DERIVE_KEY", NULL },
1043 { (uint_t)CRYPTO_FIPS140_STATUS, "CRYPTO_FIPS140_STATUS", NULL },

```

```

1044 { (uint_t)CRYPTO_FIPS140_SET, "CRYPTO_FIPS140_SET", NULL },
1046 /* kbio ioctls */
1047 { (uint_t)KIOCTRANS, "KIOCTRANS", NULL },
1048 { (uint_t)KIOCGTRANS, "KIOCGTRANS", NULL },
1049 { (uint_t)KIOCTRANSABLE, "KIOCTRANSABLE", NULL },
1050 { (uint_t)KIOCGTRANSABLE, "KIOCGTRANSABLE", NULL },
1051 { (uint_t)KIOCSETKEY, "KIOCSETKEY", NULL },
1052 { (uint_t)KIOCGSETKEY, "KIOCGSETKEY", NULL },
1053 { (uint_t)KIOCCMD, "KIOCCMD", NULL },
1054 { (uint_t)KIOCTYPE, "KIOCTYPE", NULL },
1055 { (uint_t)KIOCSDIRECT, "KIOCSDIRECT", NULL },
1056 { (uint_t)KIOCGDIRECT, "KIOCGDIRECT", NULL },
1057 { (uint_t)KIOCSKEY, "KIOCSKEY", NULL },
1058 { (uint_t)KIOCGKEY, "KIOCGKEY", NULL },
1059 { (uint_t)KIOCSLED, "KIOCSLED", NULL },
1060 { (uint_t)KIOCGLED, "KIOCGLED", NULL },
1061 { (uint_t)KIOCSCOMPAT, "KIOCSCOMPAT", NULL },
1062 { (uint_t)KIOCGCOMPAT, "KIOCGCOMPAT", NULL },
1063 { (uint_t)KIOCSLAYOUT, "KIOCSLAYOUT", NULL },
1064 { (uint_t)KIOCLAYOUT, "KIOCLAYOUT", NULL },
1065 { (uint_t)KIOCSKABORTEN, "KIOCSKABORTEN", NULL },
1066 { (uint_t)KIOCRPTDELAY, "KIOCRPTDELAY", NULL },
1067 { (uint_t)KIOCSRPTDELAY, "KIOCSRPTDELAY", NULL },
1068 { (uint_t)KIOCRPTRATE, "KIOCRPTRATE", NULL },
1069 { (uint_t)KIOCSRPRATE, "KIOCSRPRATE", NULL },
1070 { (uint_t)KIOCSETFREQ, "KIOCSETFREQ", NULL },
1071 { (uint_t)KIOCMKTONE, "KIOCMKTONE", NULL },
1073 /* ptm/pts driver I_STR ioctls */
1074 { (uint_t)ISPTM, "ISPTM", NULL },
1075 { (uint_t)UNLKPT, "UNLKPT", NULL },
1076 { (uint_t)PTSSTTY, "PTSSTTY", NULL },
1077 { (uint_t)ZONEPT, "ZONEPT", NULL },
1078 { (uint_t)OWNERPT, "OWNERPT", NULL },
1080 /* aggr link aggregation pseudo driver ioctls */
1081 { (uint_t)LAIOC_CREATE, "LAIOC_CREATE", "laioc_create"},
1082 { (uint_t)LAIOC_DELETE, "LAIOC_DELETE", "laioc_delete"},
1083 { (uint_t)LAIOC_INFO, "LAIOC_INFO", "laioc_info"},
1084 { (uint_t)LAIOC_ADD, "LAIOC_ADD", "laioc_add"},
1085 { (uint_t)LAIOC_ADD_REM, "LAIOC_ADD_REM", "laioc_add_rem"},
1086 { (uint_t)LAIOC_REMOVE, "LAIOC_REMOVE", "laioc_remove"},
1087 { (uint_t)LAIOC_ADD_REM, "LAIOC_ADD_REM", "laioc_add_rem"},
1088 { (uint_t)LAIOC_MODIFY, "LAIOC_MODIFY", "laioc_modify"},
1090 /* dld data-link ioctls */
1091 { (uint_t)DLDIOC_ATTR, "DLDIOC_ATTR", "dld_ioc_attr"},
1092 { (uint_t)DLDIOC_PHYS_ATTR, "DLDIOC_PHYS_ATTR", "dld_ioc_phys_attr"},
1093 { (uint_t)DLDIOC_DOORSERVER, "DLDIOC_DOORSERVER", "dld_ioc_door"},
1094 { (uint_t)DLDIOC_RENAME, "DLDIOC_RENAME", "dld_ioc_rename"},
1095 { (uint_t)DLDIOC_SECOBJ_GET, "DLDIOC_SECOBJ_GET", "dld_ioc_secojb_get"},
1096 { (uint_t)DLDIOC_SECOBJ_GET, "DLDIOC_SECOBJ_GET", "dld_ioc_secojb_get"},
1097 { (uint_t)DLDIOC_SECOBJ_SET, "DLDIOC_SECOBJ_SET", "dld_ioc_secojb_set"},
1098 { (uint_t)DLDIOC_SECOBJ_UNSET, "DLDIOC_SECOBJ_UNSET", "dld_ioc_secojb_unset"},
1099 { (uint_t)DLDIOC_SECOBJ_UNSET, "DLDIOC_SECOBJ_UNSET", "dld_ioc_secojb_unset"},
1100 { (uint_t)DLDIOC_MACADDRGET, "DLDIOC_MACADDRGET", "dld_ioc_macaddrget"},
1101 { (uint_t)DLDIOC_MACADDRGET, "DLDIOC_MACADDRGET", "dld_ioc_macaddrget"},
1102 { (uint_t)DLDIOC_SETMACPROP, "DLDIOC_SETMACPROP", "dld_ioc_macprop_s"},
1103 { (uint_t)DLDIOC_SETMACPROP, "DLDIOC_SETMACPROP", "dld_ioc_macprop_s"},
1104 { (uint_t)DLDIOC_GETMACPROP, "DLDIOC_GETMACPROP", "dld_ioc_macprop_s"},
1105 { (uint_t)DLDIOC_GETMACPROP, "DLDIOC_GETMACPROP", "dld_ioc_macprop_s"},
1106 { (uint_t)DLDIOC_ADDFLOW, "DLDIOC_ADDFLOW", "dld_ioc_addflow"},
1107 { (uint_t)DLDIOC_ADDFLOW, "DLDIOC_ADDFLOW", "dld_ioc_addflow"},
1108 { (uint_t)DLDIOC_ADDFLOW, "DLDIOC_ADDFLOW", "dld_ioc_addflow"},
1109 { (uint_t)DLDIOC_ADDFLOW, "DLDIOC_ADDFLOW", "dld_ioc_addflow"},

```

```

1110 { (uint_t)DLDIIOC_REMOVEFLOW, "DLDIIOC_REMOVEFLOW",
1111 "dld_ioc_removeflow"},
1112 { (uint_t)DLDIIOC_MODIFYFLOW, "DLDIIOC_MODIFYFLOW",
1113 "dld_ioc_modifyflow"},
1114 { (uint_t)DLDIIOC_WALKFLOW, "DLDIIOC_WALKFLOW",
1115 "dld_ioc_walkflow"},
1116 { (uint_t)DLDIIOC_USAGELOG, "DLDIIOC_USAGELOG",
1117 "dld_ioc_usagelog"},

1119 /* simnet ioctls */
1120 { (uint_t)SIMNET_IOC_CREATE, "SIMNET_IOC_CREATE",
1121 "simnet_ioc_create"},
1122 { (uint_t)SIMNET_IOC_DELETE, "SIMNET_IOC_DELETE",
1123 "simnet_ioc_delete"},
1124 { (uint_t)SIMNET_IOC_INFO, "SIMNET_IOC_INFO",
1125 "simnet_ioc_info"},
1126 { (uint_t)SIMNET_IOC_MODIFY, "SIMNET_IOC_MODIFY",
1127 "simnet_ioc_info"},

1129 /* vnic ioctls */
1130 { (uint_t)VNIC_IOC_CREATE, "VNIC_IOC_CREATE",
1131 "vnic_ioc_create"},
1132 { (uint_t)VNIC_IOC_DELETE, "VNIC_IOC_DELETE",
1133 "vnic_ioc_delete"},
1134 { (uint_t)VNIC_IOC_INFO, "VNIC_IOC_INFO",
1135 "vnic_ioc_info"},

1137 /* ZFS ioctls */
1138 { (uint_t)ZFS_IOC_POOL_CREATE, "ZFS_IOC_POOL_CREATE",
1139 "zfs_cmd_t"},
1140 { (uint_t)ZFS_IOC_POOL_DESTROY, "ZFS_IOC_POOL_DESTROY",
1141 "zfs_cmd_t"},
1142 { (uint_t)ZFS_IOC_POOL_IMPORT, "ZFS_IOC_POOL_IMPORT",
1143 "zfs_cmd_t"},
1144 { (uint_t)ZFS_IOC_POOL_EXPORT, "ZFS_IOC_POOL_EXPORT",
1145 "zfs_cmd_t"},
1146 { (uint_t)ZFS_IOC_POOL_CONFIGS, "ZFS_IOC_POOL_CONFIGS",
1147 "zfs_cmd_t"},
1148 { (uint_t)ZFS_IOC_POOL_STATS, "ZFS_IOC_POOL_STATS",
1149 "zfs_cmd_t"},
1150 { (uint_t)ZFS_IOC_POOL_TRYIMPORT, "ZFS_IOC_POOL_TRYIMPORT",
1151 "zfs_cmd_t"},
1152 { (uint_t)ZFS_IOC_POOL_SCAN, "ZFS_IOC_POOL_SCAN",
1153 "zfs_cmd_t"},
1154 { (uint_t)ZFS_IOC_POOL_FREEZE, "ZFS_IOC_POOL_FREEZE",
1155 "zfs_cmd_t"},
1156 { (uint_t)ZFS_IOC_POOL_UPGRADE, "ZFS_IOC_POOL_UPGRADE",
1157 "zfs_cmd_t"},
1158 { (uint_t)ZFS_IOC_POOL_GET_HISTORY, "ZFS_IOC_POOL_GET_HISTORY",
1159 "zfs_cmd_t"},
1160 { (uint_t)ZFS_IOC_VDEV_ADD, "ZFS_IOC_VDEV_ADD",
1161 "zfs_cmd_t"},
1162 { (uint_t)ZFS_IOC_VDEV_REMOVE, "ZFS_IOC_VDEV_REMOVE",
1163 "zfs_cmd_t"},
1164 { (uint_t)ZFS_IOC_VDEV_SET_STATE, "ZFS_IOC_VDEV_SET_STATE",
1165 "zfs_cmd_t"},
1166 { (uint_t)ZFS_IOC_VDEV_ATTACH, "ZFS_IOC_VDEV_ATTACH",
1167 "zfs_cmd_t"},
1168 { (uint_t)ZFS_IOC_VDEV_DETACH, "ZFS_IOC_VDEV_DETACH",
1169 "zfs_cmd_t"},
1170 { (uint_t)ZFS_IOC_VDEV_SETPATH, "ZFS_IOC_VDEV_SETPATH",
1171 "zfs_cmd_t"},
1172 { (uint_t)ZFS_IOC_VDEV_SETFRU, "ZFS_IOC_VDEV_SETFRU",
1173 "zfs_cmd_t"},
1174 { (uint_t)ZFS_IOC_OBJSET_STATS, "ZFS_IOC_OBJSET_STATS",
1175 "zfs_cmd_t"},

```

```

1176 { (uint_t)ZFS_IOC_OBJSET_ZPLPROPS, "ZFS_IOC_OBJSET_ZPLPROPS",
1177 "zfs_cmd_t"},
1178 { (uint_t)ZFS_IOC_DATASET_LIST_NEXT, "ZFS_IOC_DATASET_LIST_NEXT",
1179 "zfs_cmd_t"},
1180 { (uint_t)ZFS_IOC_SNAPSHOT_LIST_NEXT, "ZFS_IOC_SNAPSHOT_LIST_NEXT",
1181 "zfs_cmd_t"},
1182 { (uint_t)ZFS_IOC_SET_PROP, "ZFS_IOC_SET_PROP",
1183 "zfs_cmd_t"},
1184 { (uint_t)ZFS_IOC_CREATE, "ZFS_IOC_CREATE",
1185 "zfs_cmd_t"},
1186 { (uint_t)ZFS_IOC_DESTROY, "ZFS_IOC_DESTROY",
1187 "zfs_cmd_t"},
1188 { (uint_t)ZFS_IOC_ROLLBACK, "ZFS_IOC_ROLLBACK",
1189 "zfs_cmd_t"},
1190 { (uint_t)ZFS_IOC_RENAME, "ZFS_IOC_RENAME",
1191 "zfs_cmd_t"},
1192 { (uint_t)ZFS_IOC_RECV, "ZFS_IOC_RECV",
1193 "zfs_cmd_t"},
1194 { (uint_t)ZFS_IOC_SEND, "ZFS_IOC_SEND",
1195 "zfs_cmd_t"},
1196 { (uint_t)ZFS_IOC_INJECT_FAULT, "ZFS_IOC_INJECT_FAULT",
1197 "zfs_cmd_t"},
1198 { (uint_t)ZFS_IOC_CLEAR_FAULT, "ZFS_IOC_CLEAR_FAULT",
1199 "zfs_cmd_t"},
1200 { (uint_t)ZFS_IOC_INJECT_LIST_NEXT, "ZFS_IOC_INJECT_LIST_NEXT",
1201 "zfs_cmd_t"},
1202 { (uint_t)ZFS_IOC_ERROR_LOG, "ZFS_IOC_ERROR_LOG",
1203 "zfs_cmd_t"},
1204 { (uint_t)ZFS_IOC_CLEAR, "ZFS_IOC_CLEAR",
1205 "zfs_cmd_t"},
1206 { (uint_t)ZFS_IOC_PROMOTE, "ZFS_IOC_PROMOTE",
1207 "zfs_cmd_t"},
1208 { (uint_t)ZFS_IOC_SNAPSHOT, "ZFS_IOC_SNAPSHOT",
1209 "zfs_cmd_t"},
1210 { (uint_t)ZFS_IOC_DSOBJ_TO_DSNAME, "ZFS_IOC_DSOBJ_TO_DSNAME",
1211 "zfs_cmd_t"},
1212 { (uint_t)ZFS_IOC_OBJ_TO_PATH, "ZFS_IOC_OBJ_TO_PATH",
1213 "zfs_cmd_t"},
1214 { (uint_t)ZFS_IOC_POOL_SET_PROPS, "ZFS_IOC_POOL_SET_PROPS",
1215 "zfs_cmd_t"},
1216 { (uint_t)ZFS_IOC_POOL_GET_PROPS, "ZFS_IOC_POOL_GET_PROPS",
1217 "zfs_cmd_t"},
1218 { (uint_t)ZFS_IOC_SET_FSACL, "ZFS_IOC_SET_FSACL",
1219 "zfs_cmd_t"},
1220 { (uint_t)ZFS_IOC_GET_FSACL, "ZFS_IOC_GET_FSACL",
1221 "zfs_cmd_t"},
1222 { (uint_t)ZFS_IOC_SHARE, "ZFS_IOC_SHARE",
1223 "zfs_cmd_t"},
1224 { (uint_t)ZFS_IOC_INHERIT_PROP, "ZFS_IOC_INHERIT_PROP",
1225 "zfs_cmd_t"},
1226 { (uint_t)ZFS_IOC_SMB_ACL, "ZFS_IOC_SMB_ACL",
1227 "zfs_cmd_t"},
1228 { (uint_t)ZFS_IOC_USERSPACE_ONE, "ZFS_IOC_USERSPACE_ONE",
1229 "zfs_cmd_t"},
1230 { (uint_t)ZFS_IOC_USERSPACE_MANY, "ZFS_IOC_USERSPACE_MANY",
1231 "zfs_cmd_t"},
1232 { (uint_t)ZFS_IOC_USERSPACE_UPGRADE, "ZFS_IOC_USERSPACE_UPGRADE",
1233 "zfs_cmd_t"},
1234 { (uint_t)ZFS_IOC_HOLD, "ZFS_IOC_HOLD",
1235 "zfs_cmd_t"},
1236 { (uint_t)ZFS_IOC_RELEASE, "ZFS_IOC_RELEASE",
1237 "zfs_cmd_t"},
1238 { (uint_t)ZFS_IOC_GET_HOLDS, "ZFS_IOC_GET_HOLDS",
1239 "zfs_cmd_t"},
1240 { (uint_t)ZFS_IOC_OBJSET_RECVD_PROPS, "ZFS_IOC_OBJSET_RECVD_PROPS",
1241 "zfs_cmd_t"},

```

```

1242 { (uint_t)ZFS_IOC_VDEV_SPLIT, "ZFS_IOC_VDEV_SPLIT",
1243 "zfs_cmd_t" },
1244 { (uint_t)ZFS_IOC_NEXT_OBJ, "ZFS_IOC_NEXT_OBJ",
1245 "zfs_cmd_t" },
1246 { (uint_t)ZFS_IOC_DIFF, "ZFS_IOC_DIFF",
1247 "zfs_cmd_t" },
1248 { (uint_t)ZFS_IOC_TMP_SNAPSHOT, "ZFS_IOC_TMP_SNAPSHOT",
1249 "zfs_cmd_t" },
1250 { (uint_t)ZFS_IOC_OBJ_TO_STATS, "ZFS_IOC_OBJ_TO_STATS",
1251 "zfs_cmd_t" },
1252 { (uint_t)ZFS_IOC_SPACE_WRITTEN, "ZFS_IOC_SPACE_WRITTEN",
1253 "zfs_cmd_t" },
1254 { (uint_t)ZFS_IOC_DESTROY_SNAPS, "ZFS_IOC_DESTROY_SNAPS",
1254 { (uint_t)ZFS_IOC_DESTROY_SNAPS_NVL, "ZFS_IOC_DESTROY_SNAPS_NVL",
1255 "zfs_cmd_t" },
1256 { (uint_t)ZFS_IOC_POOL_REGUID, "ZFS_IOC_POOL_REGUID",
1257 "zfs_cmd_t" },
1258 { (uint_t)ZFS_IOC_POOL_REOPEN, "ZFS_IOC_POOL_REOPEN",
1259 "zfs_cmd_t" },
1260 { (uint_t)ZFS_IOC_SEND_PROGRESS, "ZFS_IOC_SEND_PROGRESS",
1261 "zfs_cmd_t" },
1262 { (uint_t)ZFS_IOC_LOG_HISTORY, "ZFS_IOC_LOG_HISTORY",
1263 "zfs_cmd_t" },
1264 { (uint_t)ZFS_IOC_SEND_NEW, "ZFS_IOC_SEND_NEW",
1265 "zfs_cmd_t" },
1266 { (uint_t)ZFS_IOC_SEND_SPACE, "ZFS_IOC_SEND_SPACE",
1267 "zfs_cmd_t" },
1268 { (uint_t)ZFS_IOC_CLONE, "ZFS_IOC_CLONE",
1269 "zfs_cmd_t" },
1270 #endif /* ! codereview */

1272 /* kssl ioctls */
1273 { (uint_t)KSSL_ADD_ENTRY, "KSSL_ADD_ENTRY",
1274 "kssl_params_t"},
1275 { (uint_t)KSSL_DELETE_ENTRY, "KSSL_DELETE_ENTRY",
1276 "sockaddr_in"},

1278 /* disk ioctls - (0x04 << 8) - dkio.h */
1279 { (uint_t)DKIOCGGEO, "DKIOCGGEO",
1280 "struct dk_geom"},
1281 { (uint_t)DKIOCINFO, "DKIOCINFO",
1282 "struct dk_info"},
1283 { (uint_t)DKIOCEJECT, "DKIOCEJECT",
1284 NULL},
1285 { (uint_t)DKIOCGVTOC, "DKIOCGVTOC",
1286 "struct vtoc"},
1287 { (uint_t)DKIOCSVTOC, "DKIOCSVTOC",
1288 "struct vtoc"},
1289 { (uint_t)DKIOCGEXTVTOC, "DKIOCGEXTVTOC",
1290 "struct extvtoc"},
1291 { (uint_t)DKIOCSEXTVTOC, "DKIOCSEXTVTOC",
1292 "struct extvtoc"},
1293 { (uint_t)DKIOCFLUSHWRITECACHE, "DKIOCFLUSHWRITECACHE",
1294 NULL},
1295 { (uint_t)DKIOCGETWCE, "DKIOCGETWCE",
1296 NULL},
1297 { (uint_t)DKIOCSETWCE, "DKIOCSETWCE",
1298 NULL},
1299 { (uint_t)DKIOCSGEO, "DKIOCSGEO",
1300 "struct dk_geom"},
1301 { (uint_t)DKIOCSAPART, "DKIOCSAPART",
1302 "struct dk_allmap"},
1303 { (uint_t)DKIOCGAPART, "DKIOCGAPART",
1304 "struct dk_allmap"},
1305 { (uint_t)DKIOCG_PHYGEO, "DKIOCG_PHYGEO",
1306 "struct dk_geom"},

```

```

1307 { (uint_t)DKIOCG_VIRTGEO, "DKIOCG_VIRTGEO",
1308 "struct dk_geom"},
1309 { (uint_t)DKIOCLOCK, "DKIOCLOCK",
1310 NULL},
1311 { (uint_t)DKIOCUNLOCK, "DKIOCUNLOCK",
1312 NULL},
1313 { (uint_t)DKIOCSTATE, "DKIOCSTATE",
1314 NULL},
1315 { (uint_t)DKIOCREMOVABLE, "DKIOCREMOVABLE",
1316 NULL},
1317 { (uint_t)DKIOCHOTPLUGGABLE, "DKIOCHOTPLUGGABLE",
1318 NULL},
1319 { (uint_t)DKIOCADDBAD, "DKIOCADDBAD",
1320 NULL},
1321 { (uint_t)DKIOCGETDEF, "DKIOCGETDEF",
1322 NULL},
1323 { (uint_t)DKIOCPARTINFO, "DKIOCPARTINFO",
1324 "struct part_info"},
1325 { (uint_t)DKIOCEXTPARTINFO, "DKIOCEXTPARTINFO",
1326 "struct extpart_info"},
1327 { (uint_t)DKIOCGMEDIAINFO, "DKIOCGMEDIAINFO",
1328 "struct dk_minfo"},
1329 { (uint_t)DKIOCGMBOOT, "DKIOCGMBOOT",
1330 NULL},
1331 { (uint_t)DKIOCSMBOOT, "DKIOCSMBOOT",
1332 NULL},
1333 { (uint_t)DKIOCSETEFI, "DKIOCSETEFI",
1334 "struct dk_efi"},
1335 { (uint_t)DKIOCGETEFI, "DKIOCGETEFI",
1336 "struct dk_efi"},
1337 { (uint_t)DKIOCPARTITION, "DKIOCPARTITION",
1338 "struct partition64"},
1339 { (uint_t)DKIOCGETVOLCAP, "DKIOCGETVOLCAP",
1340 "struct volcap_t"},
1341 { (uint_t)DKIOCSETVOLCAP, "DKIOCSETVOLCAP",
1342 "struct volcap_t"},
1343 { (uint_t)DKIOCDMR, "DKIOCDMR",
1344 "struct vol_directed_rd"},
1345 { (uint_t)DKIOCDUMPINIT, "DKIOCDUMPINIT",
1346 NULL},
1347 { (uint_t)DKIOCDUMPFINI, "DKIOCDUMPFINI",
1348 NULL},
1349 { (uint_t)DKIOCREADONLY, "DKIOCREADONLY",
1350 NULL},

1352 /* disk ioctls - (0x04 << 8) - fdio.h */
1353 { (uint_t)FDIOGCHAR, "FDIOGCHAR",
1354 "struct fd_char"},
1355 { (uint_t)FDIOSCHAR, "FDIOSCHAR",
1356 "struct fd_char"},
1357 { (uint_t)FDEJECT, "FDEJECT",
1358 NULL},
1359 { (uint_t)FDGETCHANGE, "FDGETCHANGE",
1360 NULL},
1361 { (uint_t)FDGETDRIVECHAR, "FDGETDRIVECHAR",
1362 "struct fd_drive"},
1363 { (uint_t)FDSETDRIVECHAR, "FDSETDRIVECHAR",
1364 "struct fd_drive"},
1365 { (uint_t)FDGETSEARCH, "FDGETSEARCH",
1366 NULL},
1367 { (uint_t)FDSETSEARCH, "FDSETSEARCH",
1368 NULL},
1369 { (uint_t)FDIOCMD, "FDIOCMD",
1370 "struct fd_cmd"},
1371 { (uint_t)FDRAW, "FDRAW",
1372 "struct fd_raw"},

```

```

1373 { (uint_t)FDDEFGECHAR, "FDDEFGECHAR",
1374 NULL},
1376 /* disk ioctls - (0x04 << 8) - cdio.h */
1377 { (uint_t)CDROMPAUSE, "CDROMPAUSE",
1378 NULL},
1379 { (uint_t)CDROMRESUME, "CDROMRESUME",
1380 NULL},
1381 { (uint_t)CDROMPLAYMSF, "CDROMPLAYMSF",
1382 "struct cdrom_msf"},
1383 { (uint_t)CDROMPLAYTRKIND, "CDROMPLAYTRKIND",
1384 "struct cdrom_ti"},
1385 { (uint_t)CDROMREADTOCHDR, "CDROMREADTOCHDR",
1386 "struct cdrom_tochdr"},
1387 { (uint_t)CDROMREADTOCENTRY, "CDROMREADTOCENTRY",
1388 "struct cdrom_tocentry"},
1389 { (uint_t)CDROMSTOP, "CDROMSTOP",
1390 NULL},
1391 { (uint_t)CDROMSTART, "CDROMSTART",
1392 NULL},
1393 { (uint_t)CDROMEJECT, "CDROMEJECT",
1394 NULL},
1395 { (uint_t)CDROMVOLCTRL, "CDROMVOLCTRL",
1396 "struct cdrom_volctrl"},
1397 { (uint_t)CDROMSUBCHNL, "CDROMSUBCHNL",
1398 "struct cdrom_subchnl"},
1399 { (uint_t)CDROMREADMODE2, "CDROMREADMODE2",
1400 "struct cdrom_read"},
1401 { (uint_t)CDROMREADMODE1, "CDROMREADMODE1",
1402 "struct cdrom_read"},
1403 { (uint_t)CDROMREADOFFSET, "CDROMREADOFFSET",
1404 NULL},
1405 { (uint_t)CDROMGBLKMDE, "CDROMGBLKMDE",
1406 NULL},
1407 { (uint_t)CDROMSBLKMDE, "CDROMSBLKMDE",
1408 NULL},
1409 { (uint_t)CDROMCDDA, "CDROMCDDA",
1410 "struct cdrom_cdda"},
1411 { (uint_t)CDROMCDXA, "CDROMCDXA",
1412 "struct cdrom_cdx"},
1413 { (uint_t)CDROMSUBCODE, "CDROMSUBCODE",
1414 "struct cdrom_subcode"},
1415 { (uint_t)CDROMGDRVSPEED, "CDROMGDRVSPEED",
1416 NULL},
1417 { (uint_t)CDROMSDRVSPEED, "CDROMSDRVSPEED",
1418 NULL},
1419 { (uint_t)CDROMCLOSETRAY, "CDROMCLOSETRAY",
1420 NULL},
1422 /* disk ioctls - (0x04 << 8) - uscsi.h */
1423 { (uint_t)USCSICMD, "USCSICMD",
1424 "struct uscsi_cmd"},
1426 /* dumpadm ioctls - (0xdd << 8) */
1427 { (uint_t)DIOCGETDEV, "DIOCGETDEV",
1428 NULL},
1430 /* mntio ioctls - ('m' << 8) */
1431 { (uint_t)MNNTIOC_NMNTS, "MNNTIOC_NMNTS",
1432 NULL},
1433 { (uint_t)MNNTIOC_GETDEVLIST, "MNNTIOC_GETDEVLIST",
1434 NULL},
1435 { (uint_t)MNNTIOC_SETTAG, "MNNTIOC_SETTAG",
1436 "struct mnttagdesc"},
1437 { (uint_t)MNNTIOC_CLRTAG, "MNNTIOC_CLRTAG",
1438 "struct mnttagdesc"},

```

```

1439 { (uint_t)MNNTIOC_SHOWHIDDEN, "MNNTIOC_SHOWHIDDEN",
1440 NULL},
1441 { (uint_t)MNNTIOC_GETMNTENT, "MNNTIOC_GETMNTENT",
1442 "struct mnttab"},
1443 { (uint_t)MNNTIOC_GETEXMNTENT, "MNNTIOC_GETEXMNTENT",
1444 "struct extmnttab"},
1445 { (uint_t)MNNTIOC_GETMNTANY, "MNNTIOC_GETMNTANY",
1446 "struct mnttab"},
1448 /* devinfo ioctls - ('df' << 8) - devinfo_impl.h */
1449 { (uint_t)DINFOURLD, "DINFOURLD",
1450 NULL},
1451 { (uint_t)DINFOLODRV, "DINFOLODRV",
1452 NULL},
1453 { (uint_t)DINFOIDENT, "DINFOIDENT",
1454 NULL},
1456 { (uint_t)IPTUN_CREATE, "IPTUN_CREATE", "iptun_kparams_t"},
1457 { (uint_t)IPTUN_DELETE, "IPTUN_DELETE", "datalink_id_t"},
1458 { (uint_t)IPTUN_MODIFY, "IPTUN_MODIFY", "iptun_kparams_t"},
1459 { (uint_t)IPTUN_INFO, "IPTUN_INFO", NULL},
1460 { (uint_t)IPTUN_SET_6TO4RELAY, "IPTUN_SET_6TO4RELAY", NULL},
1461 { (uint_t)IPTUN_GET_6TO4RELAY, "IPTUN_GET_6TO4RELAY", NULL},
1463 /* zcons ioctls */
1464 { (uint_t)ZC_HOLDSSLAVE, "ZC_HOLDSSLAVE", NULL },
1465 { (uint_t)ZC_RELEASESSLAVE, "ZC_RELEASESSLAVE", NULL },
1467 /* hid ioctls - ('h' << 8) - hid.h */
1468 { (uint_t)HIDIOCKMGDIRECT, "HIDIOCKMGDIRECT", NULL },
1469 { (uint_t)HIDIOCKMSDIRECT, "HIDIOCKMSDIRECT", NULL },
1471 /* pm ioctls */
1472 { (uint_t)PM_SCHEDULE, "PM_SCHEDULE", NULL },
1473 { (uint_t)PM_GET_IDLE_TIME, "PM_GET_IDLE_TIME", NULL },
1474 { (uint_t)PM_GET_NUM_CMPTS, "PM_GET_NUM_CMPTS", NULL },
1475 { (uint_t)PM_GET_THRESHOLD, "PM_GET_THRESHOLD", NULL },
1476 { (uint_t)PM_SET_THRESHOLD, "PM_SET_THRESHOLD", NULL },
1477 { (uint_t)PM_GET_NORM_PWR, "PM_GET_NORM_PWR", NULL },
1478 { (uint_t)PM_SET_CUR_PWR, "PM_SET_CUR_PWR", NULL },
1479 { (uint_t)PM_GET_CUR_PWR, "PM_GET_CUR_PWR", NULL },
1480 { (uint_t)PM_GET_NUM_DEPS, "PM_GET_NUM_DEPS", NULL },
1481 { (uint_t)PM_GET_DEP, "PM_GET_DEP", NULL },
1482 { (uint_t)PM_ADD_DEP, "PM_ADD_DEP", NULL },
1483 { (uint_t)PM_REM_DEP, "PM_REM_DEP", NULL },
1484 { (uint_t)PM_REM_DEVICE, "PM_REM_DEVICE", NULL },
1485 { (uint_t)PM_REM_DEVICES, "PM_REM_DEVICES", NULL },
1486 { (uint_t)PM_DISABLE_AUTOPM, "PM_DISABLE_AUTOPM", NULL },
1487 { (uint_t)PM_REENABLE_AUTOPM, "PM_REENABLE_AUTOPM", NULL },
1488 { (uint_t)PM_SET_NORM_PWR, "PM_SET_NORM_PWR", NULL },
1489 { (uint_t)PM_GET_SYSTEM_THRESHOLD, "PM_GET_SYSTEM_THRESHOLD",
1490 NULL },
1491 { (uint_t)PM_GET_DEFAULT_SYSTEM_THRESHOLD,
1492 "PM_GET_DEFAULT_SYSTEM_THRESHOLD", NULL },
1493 { (uint_t)PM_SET_SYSTEM_THRESHOLD, "PM_SET_SYSTEM_THRESHOLD",
1494 NULL },
1495 { (uint_t)PM_START_PM, "PM_START_PM", NULL },
1496 { (uint_t)PM_STOP_PM, "PM_STOP_PM", NULL },
1497 { (uint_t)PM_RESET_PM, "PM_RESET_PM", NULL },
1498 { (uint_t)PM_GET_PM_STATE, "PM_GET_PM_STATE", NULL },
1499 { (uint_t)PM_GET_AUTOS3_STATE, "PM_GET_AUTOS3_STATE", NULL },
1500 { (uint_t)PM_GET_S3_SUPPORT_STATE, "PM_GET_S3_SUPPORT_STATE",
1501 NULL },
1502 { (uint_t)PM_IDLE_DOWN, "PM_IDLE_DOWN", NULL },
1503 { (uint_t)PM_START_CPUPM, "PM_START_CPUPM", NULL },
1504 { (uint_t)PM_START_CPUPM_EV, "PM_START_CPUPM_EV", NULL },

```



```

1505 { (uint_t)PM_START_CPUPM_POLL, "PM_START_CPUPM_POLL", NULL },
1506 { (uint_t)PM_STOP_CPUPM, "PM_STOP_CPUPM", NULL },
1507 { (uint_t)PM_GET_CPU_THRESHOLD, "PM_GET_CPU_THRESHOLD", NULL },
1508 { (uint_t)PM_SET_CPU_THRESHOLD, "PM_SET_CPU_THRESHOLD", NULL },
1509 { (uint_t)PM_GET_CPUPM_STATE, "PM_GET_CPUPM_STATE", NULL },
1510 { (uint_t)PM_START_AUTOS3, "PM_START_AUTOS3", NULL },
1511 { (uint_t)PM_STOP_AUTOS3, "PM_STOP_AUTOS3", NULL },
1512 { (uint_t)PM_ENABLE_S3, "PM_ENABLE_S3", NULL },
1513 { (uint_t)PM_DISABLE_S3, "PM_DISABLE_S3", NULL },
1514 { (uint_t)PM_ENTER_S3, "PM_ENTER_S3", NULL },
1515 { (uint_t)PM_DISABLE_CPU_DEEP_IDLE, "PM_DISABLE_CPU_DEEP_IDLE",
1516 NULL },
1517 { (uint_t)PM_ENABLE_CPU_DEEP_IDLE, "PM_START_CPU_DEEP_IDLE",
1518 NULL },
1519 { (uint_t)PM_DEFAULT_CPU_DEEP_IDLE, "PM_DFLT_CPU_DEEP_IDLE",
1520 NULL },
1521 #ifdef _SYSCALL32
1522 { (uint_t)PM_GET_STATE_CHANGE, "PM_GET_STATE_CHANGE",
1523 "pm_state_change32_t" },
1524 { (uint_t)PM_GET_STATE_CHANGE_WAIT, "PM_GET_STATE_CHANGE_WAIT",
1525 "pm_state_change32_t" },
1526 { (uint_t)PM_DIRECT_NOTIFY, "PM_DIRECT_NOTIFY",
1527 "pm_state_change32_t" },
1528 { (uint_t)PM_DIRECT_NOTIFY_WAIT, "PM_DIRECT_NOTIFY_WAIT",
1529 "pm_state_change32_t" },
1530 { (uint_t)PM_REPARSE_PM_PROPS, "PM_REPARSE_PM_PROPS",
1531 "pm_req32_t" },
1532 { (uint_t)PM_SET_DEVICE_THRESHOLD, "PM_SET_DEVICE_THRESHOLD",
1533 "pm_req32_t" },
1534 { (uint_t)PM_GET_STATS, "PM_GET_STATS",
1535 "pm_req32_t" },
1536 { (uint_t)PM_GET_DEVICE_THRESHOLD, "PM_GET_DEVICE_THRESHOLD",
1537 "pm_req32_t" },
1538 { (uint_t)PM_GET_POWER_NAME, "PM_GET_POWER_NAME",
1539 "pm_req32_t" },
1540 { (uint_t)PM_GET_POWER_LEVELS, "PM_GET_POWER_LEVELS",
1541 "pm_req32_t" },
1542 { (uint_t)PM_GET_NUM_COMPONENTS, "PM_GET_NUM_COMPONENTS",
1543 "pm_req32_t" },
1544 { (uint_t)PM_GET_COMPONENT_NAME, "PM_GET_COMPONENT_NAME",
1545 "pm_req32_t" },
1546 { (uint_t)PM_GET_NUM_POWER_LEVELS, "PM_GET_NUM_POWER_LEVELS",
1547 "pm_req32_t" },
1548 { (uint_t)PM_DIRECT_PM, "PM_DIRECT_PM",
1549 "pm_req32_t" },
1550 { (uint_t)PM_RELEASE_DIRECT_PM, "PM_RELEASE_DIRECT_PM",
1551 "pm_req32_t" },
1552 { (uint_t)PM_RESET_DEVICE_THRESHOLD, "PM_RESET_DEVICE_THRESHOLD",
1553 "pm_req32_t" },
1554 { (uint_t)PM_GET_DEVICE_TYPE, "PM_GET_DEVICE_TYPE",
1555 "pm_req32_t" },
1556 { (uint_t)PM_SET_COMPONENT_THRESHOLDS, "PM_SET_COMPONENT_THRESHOLDS",
1557 "pm_req32_t" },
1558 { (uint_t)PM_GET_COMPONENT_THRESHOLDS, "PM_GET_COMPONENT_THRESHOLDS",
1559 "pm_req32_t" },
1560 { (uint_t)PM_GET_DEVICE_THRESHOLD_BASIS, "PM_GET_DEVICE_THRESHOLD_BASIS",
1561 "pm_req32_t" },
1562 { (uint_t)PM_SET_CURRENT_POWER, "PM_SET_CURRENT_POWER",
1563 "pm_req32_t" },
1564 { (uint_t)PM_GET_CURRENT_POWER, "PM_GET_CURRENT_POWER",
1565 "pm_req32_t" },
1566 { (uint_t)PM_GET_FULL_POWER, "PM_GET_FULL_POWER",
1567 "pm_req32_t" },
1568 { (uint_t)PM_ADD_DEPENDENT, "PM_ADD_DEPENDENT",
1569 "pm_req32_t" },
1570 { (uint_t)PM_GET_TIME_IDLE, "PM_GET_TIME_IDLE",

```

```

1571 "pm_req32_t" },
1572 { (uint_t)PM_ADD_DEPENDENT_PROPERTY, "PM_ADD_DEPENDENT_PROPERTY",
1573 "pm_req32_t" },
1574 { (uint_t)PM_GET_CMD_NAME, "PM_GET_CMD_NAME",
1575 "pm_req32_t" },
1576 { (uint_t)PM_SEARCH_LIST, "PM_SEARCH_LIST",
1577 "pm_searchargs32_t" },
1578 #else /* _SYSCALL32 */
1579 { (uint_t)PM_GET_STATE_CHANGE, "PM_GET_STATE_CHANGE",
1580 "pm_state_change_t" },
1581 { (uint_t)PM_GET_STATE_CHANGE_WAIT, "PM_GET_STATE_CHANGE_WAIT",
1582 "pm_state_change_t" },
1583 { (uint_t)PM_DIRECT_NOTIFY, "PM_DIRECT_NOTIFY",
1584 "pm_state_change_t" },
1585 { (uint_t)PM_DIRECT_NOTIFY_WAIT, "PM_DIRECT_NOTIFY_WAIT",
1586 "pm_state_change_t" },
1587 { (uint_t)PM_REPARSE_PM_PROPS, "PM_REPARSE_PM_PROPS",
1588 "pm_req_t" },
1589 { (uint_t)PM_SET_DEVICE_THRESHOLD, "PM_SET_DEVICE_THRESHOLD",
1590 "pm_req_t" },
1591 { (uint_t)PM_GET_STATS, "PM_GET_STATS",
1592 "pm_req_t" },
1593 { (uint_t)PM_GET_DEVICE_THRESHOLD, "PM_GET_DEVICE_THRESHOLD",
1594 "pm_req_t" },
1595 { (uint_t)PM_GET_POWER_NAME, "PM_GET_POWER_NAME",
1596 "pm_req_t" },
1597 { (uint_t)PM_GET_POWER_LEVELS, "PM_GET_POWER_LEVELS",
1598 "pm_req_t" },
1599 { (uint_t)PM_GET_NUM_COMPONENTS, "PM_GET_NUM_COMPONENTS",
1600 "pm_req_t" },
1601 { (uint_t)PM_GET_COMPONENT_NAME, "PM_GET_COMPONENT_NAME",
1602 "pm_req_t" },
1603 { (uint_t)PM_GET_NUM_POWER_LEVELS, "PM_GET_NUM_POWER_LEVELS",
1604 "pm_req_t" },
1605 { (uint_t)PM_DIRECT_PM, "PM_DIRECT_PM",
1606 "pm_req_t" },
1607 { (uint_t)PM_RELEASE_DIRECT_PM, "PM_RELEASE_DIRECT_PM",
1608 "pm_req_t" },
1609 { (uint_t)PM_RESET_DEVICE_THRESHOLD, "PM_RESET_DEVICE_THRESHOLD",
1610 "pm_req_t" },
1611 { (uint_t)PM_GET_DEVICE_TYPE, "PM_GET_DEVICE_TYPE",
1612 "pm_req_t" },
1613 { (uint_t)PM_SET_COMPONENT_THRESHOLDS, "PM_SET_COMPONENT_THRESHOLDS",
1614 "pm_req_t" },
1615 { (uint_t)PM_GET_COMPONENT_THRESHOLDS, "PM_GET_COMPONENT_THRESHOLDS",
1616 "pm_req_t" },
1617 { (uint_t)PM_GET_DEVICE_THRESHOLD_BASIS, "PM_GET_DEVICE_THRESHOLD_BASIS",
1618 "pm_req_t" },
1619 { (uint_t)PM_SET_CURRENT_POWER, "PM_SET_CURRENT_POWER",
1620 "pm_req_t" },
1621 { (uint_t)PM_GET_CURRENT_POWER, "PM_GET_CURRENT_POWER",
1622 "pm_req_t" },
1623 { (uint_t)PM_GET_FULL_POWER, "PM_GET_FULL_POWER",
1624 "pm_req_t" },
1625 { (uint_t)PM_ADD_DEPENDENT, "PM_ADD_DEPENDENT",
1626 "pm_req_t" },
1627 { (uint_t)PM_GET_TIME_IDLE, "PM_GET_TIME_IDLE",
1628 "pm_req_t" },
1629 { (uint_t)PM_ADD_DEPENDENT_PROPERTY, "PM_ADD_DEPENDENT_PROPERTY",
1630 "pm_req_t" },
1631 { (uint_t)PM_GET_CMD_NAME, "PM_GET_CMD_NAME",
1632 "pm_req_t" },
1633 { (uint_t)PM_SEARCH_LIST, "PM_SEARCH_LIST",
1634 "pm_searchargs_t" },
1635 #endif /* _SYSCALL32 */

```

```

1637     { (uint_t)0, NULL, NULL }
1638 };

1640 void
1641 ioctl_ioccom(char *buf, size_t size, uint_t code, int nbytes, int x, int y)
1642 {
1643     const char *inoutstr;

1645     if (code & IOC_VOID)
1646         inoutstr = "";
1647     else if ((code & IOC_INOUT) == IOC_INOUT)
1648         inoutstr = "WR";
1649     else
1650         inoutstr = code & IOC_IN ? "W" : "R";

1652     if (isascii(x) && isprint(x))
1653         (void) snprintf(buf, size, "_IO%SN('%c', %d, %d)", inoutstr,
1654             x, y, nbytes);
1655     else
1656         (void) snprintf(buf, size, "_IO%SN(0x%x, %d, %d)", inoutstr,
1657             x, y, nbytes);
1658 }

```

```

1661 const char *
1662 ioctlname(private_t *pri, uint_t code)

```

```

1663 {
1664     const struct ioc *ip;
1665     const char *str = NULL;

1667     for (ip = &ioc[0]; ip->name; ip++) {
1668         if (code == ip->code) {
1669             str = ip->name;
1670             break;
1671         }
1672     }

```

```

1674     /*
1675     * Developers hide ascii ioctl names in the ioctl subcode; for example
1676     * 0x445210 should be printed 'D'<<16|R'<<8|10. We allow for all
1677     * three high order bytes (called hi, mid and lo) to contain ascii
1678     * characters.
1679     */
1680     if (str == NULL) {
1681         int c_hi = code >> 24;
1682         int c_mid = (code >> 16) & 0xff;
1683         int c_mid_nm = (code >> 16);
1684         int c_lo = (code >> 8) & 0xff;
1685         int c_lo_nm = code >> 8;

```

```

1687         if (isascii(c_lo) && isprint(c_lo) &&
1688             isascii(c_mid) && isprint(c_mid) &&
1689             isascii(c_hi) && isprint(c_hi))
1690             (void) sprintf(pri->code_buf,
1691                 "(( '%c'<<24)|('%c'<<16)|('%c'<<8)|%d)",
1692                 c_hi, c_mid, c_lo, code & 0xff);
1693         else if (isascii(c_lo) && isprint(c_lo) &&
1694             isascii(c_mid_nm) && isprint(c_mid_nm))
1695             (void) sprintf(pri->code_buf,
1696                 "(( '%c'<<16)|('%c'<<8)|%d)", c_mid, c_lo,
1697                 code & 0xff);
1698         else if (isascii(c_lo_nm) && isprint(c_lo_nm))
1699             (void) sprintf(pri->code_buf, "(( '%c'<<8)|%d)",
1700                 c_lo_nm, code & 0xff);
1701         else if (code & (IOC_VOID|IOC_INOUT))
1702             ioctl_ioccom(pri->code_buf, sizeof (pri->code_buf),

```

```

1703         code, c_mid, c_lo, code & 0xff);
1704     else
1705         (void) sprintf(pri->code_buf, "0x%.4X", code);
1706     str = (const char *)pri->code_buf;
1707 }

1709     return (str);
1710 }

```

```

1713 const char *
1714 ioctldatastruct(uint_t code)
1715 {
1716     const struct ioc *ip;
1717     const char *str = NULL;

1719     for (ip = &ioc[0]; ip->name != NULL; ip++) {
1720         if (code == ip->code) {
1721             str = ip->datastruct;
1722             break;
1723         }
1724     }
1725     return (str);
1726 }

```

```

1729 const char *
1730 fcctlname(int code)
1731 {
1732     const char *str = NULL;

1734     if (code >= FCNTLMIN && code <= FCNTLMAX)
1735         str = FCNTLname[code-FCNTLMIN];
1736     return (str);
1737 }

```

```

1739 const char *
1740 sfsname(int code)
1741 {
1742     const char *str = NULL;

1744     if (code >= SYSFSMIN && code <= SYSFSMAX)
1745         str = SYSFSname[code-SYSFSMIN];
1746     return (str);
1747 }

```

```

1749 /* ARGSUSED */
1750 const char *
1751 si86name(int code)
1752 {
1753     const char *str = NULL;

1755 #if defined(__i386) || defined(__amd64)
1756     switch (code) {
1757     case SI86SWPI:           str = "SI86SWPI";           break;
1758     case SI86SYM:           str = "SI86SYM";           break;
1759     case SI86CONF:         str = "SI86CONF";           break;
1760     case SI86BOOT:         str = "SI86BOOT";           break;
1761     case SI86AUTO:         str = "SI86AUTO";           break;
1762     case SI86EDT:          str = "SI86EDT";           break;
1763     case SI86SWAP:         str = "SI86SWAP";           break;
1764     case SI86FPHW:         str = "SI86FPHW";           break;
1765     case SI86FPSTART:      str = "SI86FPSTART";        break;
1766     case GRNON:            str = "GRNON";              break;
1767     case GRNFLASH:         str = "GRNFLASH";           break;
1768     case STIME:            str = "STIME";              break;

```

```

1769     case SETNAME:      str = "SETNAME";      break;
1770     case RNVR:         str = "RNVR";          break;
1771     case WNVR:         str = "WNVR";          break;
1772     case RTODC:        str = "RTODC";         break;
1773     case CHKSER:       str = "CHKSER";        break;
1774     case SI86NVPRT:    str = "SI86NVPRT";     break;
1775     case SANUPD:        str = "SANUPD";       break;
1776     case SI86KSTR:     str = "SI86KSTR";      break;
1777     case SI86MEM:      str = "SI86MEM";       break;
1778     case SI86TODEMON: str = "SI86TODEMON";    break;
1779     case SI86CCDEMON: str = "SI86CCDEMON";    break;
1780     case SI86CACHE:   str = "SI86CACHE";     break;
1781     case SI86DELMEM:  str = "SI86DELMEM";    break;
1782     case SI86ADDMEM:  str = "SI86ADDMEM";    break;
1783 /* 71 through 74 reserved for VPIX */
1784     case SI86V86:      str = "SI86V86";       break;
1785     case SI86SLTIME:  str = "SI86SLTIME";    break;
1786     case SI86DSCR:    str = "SI86DSCR";      break;
1787     case RDUBLK:      str = "RDUBLK";        break;
1788 /* NFA entry point */
1789     case SI86NFA:      str = "SI86NFA";       break;
1790     case SI86VM86:    str = "SI86VM86";      break;
1791     case SI86VMENABLE: str = "SI86VMENABLE";  break;
1792     case SI86LIMUSER: str = "SI86LIMUSER";   break;
1793     case SI86RDID:    str = "SI86RDID";      break;
1794     case SI86RDBOOT:  str = "SI86RDBOOT";    break;
1795 /* Merged Product defines */
1796     case SI86SHFIL:   str = "SI86SHFIL";     break;
1797     case SI86PCHRG:   str = "SI86PCHRG";     break;
1798     case SI86BADVISE: str = "SI86BADVISE";   break;
1799     case SI86SHRGN:   str = "SI86SHRGN";     break;
1800     case SI86CHIDT:   str = "SI86CHIDT";     break;
1801     case SI86EMULRDA: str = "SI86EMULRDA";   break;
1802 /* RTC commands */
1803     case WTODC:        str = "WTODC";         break;
1804     case SGMTL:        str = "SGMTL";         break;
1805     case GGMTL:        str = "GGMTL";         break;
1806     case RTCSYNC:     str = "RTCSYNC";        break;
1807     }
1808 #endif /* __i386 */

1810     return (str);
1811 }

1813 const char *
1814 utscode(int code)
1815 {
1816     const char *str = NULL;

1818     switch (code) {
1819     case UTS_UNAME:    str = "UNAME"; break;
1820     case UTS_USTAT:   str = "USTAT"; break;
1821     case UTS_FUSERS:  str = "FUSERS"; break;
1822     }

1824     return (str);
1825 }

1827 const char *
1828 rctlsyscode(int code)
1829 {
1830     const char *str = NULL;
1831     switch (code) {
1832     case 0:           str = "GETRCTL";      break;
1833     case 1:           str = "SETRCTL";      break;
1834     case 2:           str = "RCTLSYS_LST";  break;

```

```

1835     case 3:           str = "RCTLSYS_CTL";   break;
1836     case 4:           str = "RCTLSYS_SETPROJ"; break;
1837     default:         str = "UNKNOWN";       break;
1838     }
1839     return (str);
1840 }

1842 const char *
1843 rctl_local_action(private_t *pri, uint_t val)
1844 {
1845     uint_t action = val & (~RCTL_LOCAL_ACTION_MASK);

1847     char *s = pri->code_buf;

1849     *s = '\0';

1851     if (action & RCTL_LOCAL_NOACTION) {
1852         action ^= RCTL_LOCAL_NOACTION;
1853         (void) strcat(s, "|RCTL_LOCAL_NOACTION",
1854             sizeof (pri->code_buf));
1855     }
1856     if (action & RCTL_LOCAL_SIGNAL) {
1857         action ^= RCTL_LOCAL_SIGNAL;
1858         (void) strcat(s, "|RCTL_LOCAL_SIGNAL",
1859             sizeof (pri->code_buf));
1860     }
1861     if (action & RCTL_LOCAL_DENY) {
1862         action ^= RCTL_LOCAL_DENY;
1863         (void) strcat(s, "|RCTL_LOCAL_DENY",
1864             sizeof (pri->code_buf));
1865     }

1867     if ((action & (~RCTL_LOCAL_ACTION_MASK)) != 0)
1868         return (NULL);
1869     else if (*s != '\0')
1870         return (s+1);
1871     else
1872         return (NULL);
1873 }

1876 const char *
1877 rctl_local_flags(private_t *pri, uint_t val)
1878 {
1879     uint_t pval = val & RCTL_LOCAL_ACTION_MASK;
1880     char *s = pri->code_buf;

1882     *s = '\0';

1884     if (pval & RCTL_LOCAL_MAXIMAL) {
1885         pval ^= RCTL_LOCAL_MAXIMAL;
1886         (void) strcat(s, "|RCTL_LOCAL_MAXIMAL",
1887             sizeof (pri->code_buf));
1888     }

1890     if ((pval & RCTL_LOCAL_ACTION_MASK) != 0)
1891         return (NULL);
1892     else if (*s != '\0')
1893         return (s+1);
1894     else
1895         return (NULL);
1896 }

1899 const char *
1900 sconfname(int code)

```

```

1901 {
1902     const char *str = NULL;

1904     if (code >= SCONFMIN && code <= SCONFMAX)
1905         str = SCONFname[code-SCONFMIN];
1906     return (str);
1907 }

1909 const char *
1910 pathconfname(int code)
1911 {
1912     const char *str = NULL;

1914     if (code >= PATHCONFMIN && code <= PATHCONFMAX)
1915         str = PATHCONFname[code-PATHCONFMIN];
1916     return (str);
1917 }

1919 #define ALL_O_FLAGS \
1920 (O_NDELAY|O_APPEND|O_SYNC|O_DSYNC|O_NONBLOCK|O_CREAT|O_TRUNC\
1921 |O_EXCL|O_NOCTTY|O_LARGEFILE|O_RSYNC|O_XATTR|O_NOFOLLOW|O_NOLINKS\
1922 |FXATTRDIROPEN)

1924 const char *
1925 openarg(private_t *pri, int arg)
1926 {
1927     char *str = pri->code_buf;

1929     if ((arg & ~(O_ACCMODE | ALL_O_FLAGS)) != 0)
1930         return (NULL);

1932     switch (arg & O_ACCMODE) {
1933     default:
1934         return (NULL);
1935     case O_RDONLY:
1936         (void) strcpy(str, "O_RDONLY");
1937         break;
1938     case O_WRONLY:
1939         (void) strcpy(str, "O_WRONLY");
1940         break;
1941     case O_RDWR:
1942         (void) strcpy(str, "O_RDWR");
1943         break;
1944     case O_SEARCH:
1945         (void) strcpy(str, "O_SEARCH");
1946         break;
1947     case O_EXEC:
1948         (void) strcpy(str, "O_EXEC");
1949         break;
1950     }

1952     if (arg & O_NDELAY)
1953         (void) strcat(str, "|O_NDELAY", sizeof (pri->code_buf));
1954     if (arg & O_APPEND)
1955         (void) strcat(str, "|O_APPEND", sizeof (pri->code_buf));
1956     if (arg & O_SYNC)
1957         (void) strcat(str, "|O_SYNC", sizeof (pri->code_buf));
1958     if (arg & O_DSYNC)
1959         (void) strcat(str, "|O_DSYNC", sizeof (pri->code_buf));
1960     if (arg & O_NONBLOCK)
1961         (void) strcat(str, "|O_NONBLOCK", sizeof (pri->code_buf));
1962     if (arg & O_CREAT)
1963         (void) strcat(str, "|O_CREAT", sizeof (pri->code_buf));
1964     if (arg & O_TRUNC)
1965         (void) strcat(str, "|O_TRUNC", sizeof (pri->code_buf));
1966     if (arg & O_EXCL)

```

```

1967         (void) strcat(str, "|O_EXCL", sizeof (pri->code_buf));
1968     if (arg & O_NOCTTY)
1969         (void) strcat(str, "|O_NOCTTY", sizeof (pri->code_buf));
1970     if (arg & O_LARGEFILE)
1971         (void) strcat(str, "|O_LARGEFILE", sizeof (pri->code_buf));
1972     if (arg & O_RSYNC)
1973         (void) strcat(str, "|O_RSYNC", sizeof (pri->code_buf));
1974     if (arg & O_XATTR)
1975         (void) strcat(str, "|O_XATTR", sizeof (pri->code_buf));
1976     if (arg & O_NOFOLLOW)
1977         (void) strcat(str, "|O_NOFOLLOW", sizeof (pri->code_buf));
1978     if (arg & O_NOLINKS)
1979         (void) strcat(str, "|O_NOLINKS", sizeof (pri->code_buf));
1980     if (arg & FXATTRDIROPEN)
1981         (void) strcat(str, "|FXATTRDIROPEN", sizeof (pri->code_buf));

1983     return ((const char *)str);
1984 }

1986 const char *
1987 whencearg(int arg)
1988 {
1989     const char *str = NULL;

1991     switch (arg) {
1992     case SEEK_SET: str = "SEEK_SET"; break;
1993     case SEEK_CUR: str = "SEEK_CUR"; break;
1994     case SEEK_END: str = "SEEK_END"; break;
1995     case SEEK_DATA: str = "SEEK_DATA"; break;
1996     case SEEK_HOLE: str = "SEEK_HOLE"; break;
1997     }

1999     return (str);
2000 }

2002 #define IPC_FLAGS (IPC_ALLOC|IPC_CREAT|IPC_EXCL|IPC_NOWAIT)

2004 char *
2005 ipcflags(private_t *pri, int arg)
2006 {
2007     char *str = pri->code_buf;

2009     if (arg & 0777)
2010         (void) sprintf(str, "%0.3o", arg&0777);
2011     else
2012         *str = '\0';

2014     if (arg & IPC_ALLOC)
2015         (void) strcat(str, "|IPC_ALLOC");
2016     if (arg & IPC_CREAT)
2017         (void) strcat(str, "|IPC_CREAT");
2018     if (arg & IPC_EXCL)
2019         (void) strcat(str, "|IPC_EXCL");
2020     if (arg & IPC_NOWAIT)
2021         (void) strcat(str, "|IPC_NOWAIT");

2023     return (str);
2024 }

2026 const char *
2027 msgflags(private_t *pri, int arg)
2028 {
2029     char *str;

2031     if (arg == 0 || (arg & ~(IPC_FLAGS|MSG_NOERROR|0777)) != 0)
2032         return ((char *)NULL);

```

```

2034     str = ipcflags(pri, arg);
2036     if (arg & MSG_NOERROR)
2037         (void) strcat(str, "|MSG_NOERROR");
2039     if (*str == '|')
2040         str++;
2041     return ((const char *)str);
2042 }
2044 const char *
2045 semflags(private_t *pri, int arg)
2046 {
2047     char *str;
2049     if (arg == 0 || (arg & ~(IPC_FLAGS|SEM_UNDO|0777)) != 0)
2050         return ((char *)NULL);
2052     str = ipcflags(pri, arg);
2054     if (arg & SEM_UNDO)
2055         (void) strcat(str, "|SEM_UNDO");
2057     if (*str == '|')
2058         str++;
2059     return ((const char *)str);
2060 }
2062 const char *
2063 shmflags(private_t *pri, int arg)
2064 {
2065     char *str;
2067     if (arg == 0 || (arg & ~(IPC_FLAGS|SHM_RDONLY|SHM_RND|0777)) != 0)
2068         return ((char *)NULL);
2070     str = ipcflags(pri, arg);
2072     if (arg & SHM_RDONLY)
2073         (void) strcat(str, "|SHM_RDONLY");
2074     if (arg & SHM_RND)
2075         (void) strcat(str, "|SHM_RND");
2077     if (*str == '|')
2078         str++;
2079     return ((const char *)str);
2080 }
2082 #define MSGCMDMIN      0
2083 #define MSGCMDMAX      IPC_STAT64
2084 const char *const MSGCMDname[MSGCMDMAX+1] = {
2085     NULL, NULL, NULL, NULL, NULL,
2086     NULL, NULL, NULL, NULL, NULL,
2087     "IPC_RMID",      /* 10 */
2088     "IPC_SET",       /* 11 */
2089     "IPC_STAT",      /* 12 */
2090     "IPC_SET64",     /* 13 */
2091     "IPC_STAT64",   /* 14 */
2092 };
2094 #define SEMCMDMIN      0
2095 #define SEMCMDMAX      IPC_STAT64
2096 const char *const SEMCMDname[SEMCMDMAX+1] = {
2097     NULL,          /* 0 */
2098     NULL,          /* 1 */

```

```

2099     NULL,          /* 2 */
2100     "GETNCNT",     /* 3 */
2101     "GETPID",      /* 4 */
2102     "GETVAL",      /* 5 */
2103     "GETALL",      /* 6 */
2104     "GETZCNT",     /* 7 */
2105     "SETVAL",      /* 8 */
2106     "SETALL",      /* 9 */
2107     "IPC_RMID",    /* 10 */
2108     "IPC_SET",     /* 11 */
2109     "IPC_STAT",    /* 12 */
2110     "IPC_SET64",   /* 13 */
2111     "IPC_STAT64", /* 14 */
2112 };
2114 #define SHMCMDDMIN      0
2115 #define SHMCMDDMAX      IPC_STAT64
2116 const char *const SHMCMDDname[SHMCMDDMAX+1] = {
2117     NULL,          /* 0 */
2118     NULL,          /* 1 */
2119     NULL,          /* 2 */
2120     "SHM_LOCK",   /* 3 */
2121     "SHM_UNLOCK", /* 4 */
2122     NULL, NULL, NULL, NULL, NULL, /* 5 NULLS */
2123     "IPC_RMID",   /* 10 */
2124     "IPC_SET",    /* 11 */
2125     "IPC_STAT",   /* 12 */
2126     "IPC_SET64", /* 13 */
2127     "IPC_STAT64", /* 14 */
2128 };
2130 const char *
2131 msgcmd(int arg)
2132 {
2133     const char *str = NULL;
2135     if (arg >= MSGCMDMIN && arg <= MSGCMDMAX)
2136         str = MSGCMDname[arg-MSGCMDMIN];
2137     return (str);
2138 }
2140 const char *
2141 semcmd(int arg)
2142 {
2143     const char *str = NULL;
2145     if (arg >= SEMCMDMIN && arg <= SEMCMDMAX)
2146         str = SEMCMDname[arg-SEMCMDMIN];
2147     return (str);
2148 }
2150 const char *
2151 shmcmd(int arg)
2152 {
2153     const char *str = NULL;
2155     if (arg >= SHMCMDDMIN && arg <= SHMCMDDMAX)
2156         str = SHMCMDDname[arg-SHMCMDDMIN];
2157     return (str);
2158 }
2160 const char *
2161 strrdopt(int arg) /* streams read option (I_SRDOPT I_GRDOPT) */
2162 {
2163     const char *str = NULL;

```



```

2297     const char *str = NULL;
2299     switch (arg) {
2300     case F_FILE_ONLY:      str = "F_FILE_ONLY";      break;
2301     case F_CONTAINED:    str = "F_CONTAINED";      break;
2302     }
2304     return (str);
2305 }
2307 const char *
2308 fuflags(private_t *pri, int arg)      /* fusers() output flags */
2309 {
2310     char *str = pri->code_buf;
2312     if (arg & ~(F_CDIR|F_RDIR|F_TEXT|F_MAP|F_OPEN|F_TRACE|F_TTY)) {
2313         (void) sprintf(str, "0x%x", arg);
2314         return (str);
2315     }
2316     *str = '\0';
2317     if (arg & F_CDIR)
2318         (void) strcat(str, "|F_CDIR");
2319     if (arg & F_RDIR)
2320         (void) strcat(str, "|F_RDIR");
2321     if (arg & F_TEXT)
2322         (void) strcat(str, "|F_TEXT");
2323     if (arg & F_MAP)
2324         (void) strcat(str, "|F_MAP");
2325     if (arg & F_OPEN)
2326         (void) strcat(str, "|F_OPEN");
2327     if (arg & F_TRACE)
2328         (void) strcat(str, "|F_TRACE");
2329     if (arg & F_TTY)
2330         (void) strcat(str, "|F_TTY");
2331     if (*str == '\0')
2332         (void) strcat(str, "|0");
2333     return ((const char *) (str+1));
2334 }
2337 const char *
2338 ipprotos(int arg)      /* IP protocols cf. netinet/in.h */
2339 {
2340     switch (arg) {
2341     case IPPROTO_IP:      return ("IPPROTO_IP");
2342     case IPPROTO_ICMP:    return ("IPPROTO_ICMP");
2343     case IPPROTO_IGMP:    return ("IPPROTO_IGMP");
2344     case IPPROTO_GGP:     return ("IPPROTO_GGP");
2345     case IPPROTO_ENCAP:   return ("IPPROTO_ENCAP");
2346     case IPPROTO_TCP:     return ("IPPROTO_TCP");
2347     case IPPROTO_EGP:     return ("IPPROTO_EGP");
2348     case IPPROTO_PUP:     return ("IPPROTO_PUP");
2349     case IPPROTO_UDP:     return ("IPPROTO_UDP");
2350     case IPPROTO_IDP:     return ("IPPROTO_IDP");
2351     case IPPROTO_IPV6:    return ("IPPROTO_IPV6");
2352     case IPPROTO_ROUTING: return ("IPPROTO_ROUTING");
2353     case IPPROTO_FRAGMENT: return ("IPPROTO_FRAGMENT");
2354     case IPPROTO_RSVP:   return ("IPPROTO_RSVP");
2355     case IPPROTO_ESP:    return ("IPPROTO_ESP");
2356     case IPPROTO_AH:     return ("IPPROTO_AH");
2357     case IPPROTO_ICMPV6: return ("IPPROTO_ICMPV6");
2358     case IPPROTO_NONE:   return ("IPPROTO_NONE");
2359     case IPPROTO_DSTOPTS: return ("IPPROTO_DSTOPTS");
2360     case IPPROTO_HELLO:  return ("IPPROTO_HELLO");
2361     case IPPROTO_ND:     return ("IPPROTO_ND");
2362     case IPPROTO_EON:    return ("IPPROTO_EON");

```

```

2363     case IPPROTO_PIM:     return ("IPPROTO_PIM");
2364     case IPPROTO_SCTP:    return ("IPPROTO_SCTP");
2365     case IPPROTO_RAW:    return ("IPPROTO_RAW");
2366     default:              return (NULL);
2367     }
2368 }

```

new/usr/src/cmd/zdb/zdb.c

1

```
*****
83696 Thu Jun 28 15:57:06 2012
new/usr/src/cmd/zdb/zdb.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2012 by Delphix. All rights reserved.
25  */
26
27 #include <stdio.h>
28 #include <stdio_ext.h>
29 #include <stdlib.h>
30 #include <ctype.h>
31 #include <sys/zfs_context.h>
32 #include <sys/spa.h>
33 #include <sys/spa_impl.h>
34 #include <sys/dmu.h>
35 #include <sys/zap.h>
36 #include <sys/fs/zfs.h>
37 #include <sys/zfs_znode.h>
38 #include <sys/zfs_sa.h>
39 #include <sys/sa.h>
40 #include <sys/sa_impl.h>
41 #include <sys/vdev.h>
42 #include <sys/vdev_impl.h>
43 #include <sys/metaslab_impl.h>
44 #include <sys/dmu_objset.h>
45 #include <sys/dsl_dir.h>
46 #include <sys/dsl_dataset.h>
47 #include <sys/dsl_pool.h>
48 #include <sys/dbuf.h>
49 #include <sys/zil.h>
50 #include <sys/zil_impl.h>
51 #include <sys/stat.h>
52 #include <sys/resource.h>
53 #include <sys/dmu_traverse.h>
54 #include <sys/zio_checksum.h>
```

new/usr/src/cmd/zdb/zdb.c

2

```
55 #include <sys/zio_compress.h>
56 #include <sys/zfs_fuid.h>
57 #include <sys/arc.h>
58 #include <sys/ddt.h>
59 #include <sys/zfeature.h>
60 #include <zfs_comutil.h>
61 #endif /* ! codereview */
62 #undef ZFS_MAXNAMELEN
63 #undef verify
64 #include <libzfs.h>
65
66 #define ZDB_COMPRESS_NAME(idx) ((idx) < ZIO_COMPRESS_FUNCTIONS ? \
67     zio_compress_table[(idx)].ci_name : "UNKNOWN")
68 #define ZDB_CHECKSUM_NAME(idx) ((idx) < ZIO_CHECKSUM_FUNCTIONS ? \
69     zio_checksum_table[(idx)].ci_name : "UNKNOWN")
70 #define ZDB_OT_NAME(idx) ((idx) < DMU_OT_NUMTYPES ? \
71     dmu_ot[(idx)].ot_name : DMU_OT_IS_VALID(idx) ? \
72     dmu_ot_byteswap[DMU_OT_BYTESWAP(idx)].ob_name : "UNKNOWN")
73 #define ZDB_OT_TYPE(idx) ((idx) < DMU_OT_NUMTYPES ? (idx) : DMU_OT_NUMTYPES)
74
75 #ifndef lint
76 extern int zfs_recover;
77 #else
78 int zfs_recover;
79 #endif
80
81 const char cmdname[] = "zdb";
82 uint8_t dump_opt[256];
83
84 typedef void object_viewer_t(objset_t *, uint64_t, void *data, size_t size);
85
86 extern void dump_intent_log(zilog_t *);
87 uint64_t *zopt_object = NULL;
88 int zopt_objects = 0;
89 libzfs_handle_t *g_zfs;
90
91 /*
92  * These libumem hooks provide a reasonable set of defaults for the allocator's
93  * debugging facilities.
94  */
95 const char *
96 _umem_debug_init()
97 {
98     return ("default,verbose"); /* $UMEM_DEBUG setting */
99 }
100
101 const char *
102 _umem_logging_init(void)
103 {
104     return ("fail,contents"); /* $UMEM_LOGGING setting */
105 }
106
107 static void
108 usage(void)
109 {
110     (void) fprintf(stderr,
111         "Usage: %s [-CumdibcsDvhLXFPA] [-t txg] [-e [-p path...]] "\n"
112         "poolname [object...]\n"
113         "    %s [-divPA] [-e -p path...] dataset [object...]\n"
114         "    %s -m [-LXFPA] [-t txg] [-e [-p path...]] "\n"
115         "poolname [vdev [metaslab...]]\n"
116         "    %s -R [-A] [-e [-p path...]] poolname "\n"
117         "vdev:offset:size[:flags]\n"
118         "    %s -S [-PA] [-e [-p path...]] poolname\n"
119         "    %s -l [-uA] device\n"
120         "    %s -C [-A] [-U config]\n",

```



```

121     cmdname, cmdname, cmdname, cmdname, cmdname, cmdname);
122
123     (void) fprintf(stderr, "    Dataset name must include at least one "
124     "separator character '/' or '@'\n");
125     (void) fprintf(stderr, "    If dataset name is specified, only that "
126     "dataset is dumped\n");
127     (void) fprintf(stderr, "    If object numbers are specified, only "
128     "those objects are dumped\n\n");
129     (void) fprintf(stderr, "    Options to control amount of output:\n");
130     (void) fprintf(stderr, "        -u uberblock\n");
131     (void) fprintf(stderr, "        -d dataset(s)\n");
132     (void) fprintf(stderr, "        -i intent logs\n");
133     (void) fprintf(stderr, "        -C config (or cachefile if alone)\n");
134     (void) fprintf(stderr, "        -h pool history\n");
135     (void) fprintf(stderr, "        -b block statistics\n");
136     (void) fprintf(stderr, "        -m metaslabs\n");
137     (void) fprintf(stderr, "        -c checksum all metadata (twice for "
138     "all data) blocks\n");
139     (void) fprintf(stderr, "        -s report stats on zdb's I/O\n");
140     (void) fprintf(stderr, "        -D dedup statistics\n");
141     (void) fprintf(stderr, "        -S simulate dedup to measure effect\n");
142     (void) fprintf(stderr, "        -v verbose (applies to all others)\n");
143     (void) fprintf(stderr, "        -l dump label contents\n");
144     (void) fprintf(stderr, "        -L disable leak tracking (do not "
145     "load spacemaps)\n");
146     (void) fprintf(stderr, "        -R read and display block from a "
147     "device\n");
148     (void) fprintf(stderr, "    Below options are intended for use "
149     "with other options (except -l):\n");
150     (void) fprintf(stderr, "        -A ignore assertions (-A), enable "
151     "panic recovery (-AA) or both (-AAA)\n");
152     (void) fprintf(stderr, "        -F attempt automatic rewind within "
153     "safe range of transaction groups\n");
154     (void) fprintf(stderr, "        -U <cachefile_path> -- use alternate "
155     "cachefile\n");
156     (void) fprintf(stderr, "        -X attempt extreme rewind (does not "
157     "work with dataset)\n");
158     (void) fprintf(stderr, "        -e pool is exported/destroyed/"
159     "has altroot/not in a cachefile\n");
160     (void) fprintf(stderr, "        -p <path> -- use one or more with "
161     "-e to specify path to vdev dir\n");
162     (void) fprintf(stderr, "        -P print numbers in parseable form\n");
163     (void) fprintf(stderr, "        -t <txg> -- highest txg to use when "
164     "searching for uberblocks\n");
165     (void) fprintf(stderr, "Specify an option more than once (e.g. -bb "
166     "to make only that option verbose\n");
167     (void) fprintf(stderr, "Default is to dump everything non-verbosely\n");
168     exit(1);
169 }
170
171 /*
172 * Called for usage errors that are discovered after a call to spa_open(),
173 * dmu_bonus_hold(), or pool_match(). abort() is called for other errors.
174 */
175
176 static void
177 fatal(const char *fmt, ...)
178 {
179     va_list ap;
180
181     va_start(ap, fmt);
182     (void) fprintf(stderr, "%s: ", cmdname);
183     (void) vfprintf(stderr, fmt, ap);
184     va_end(ap);
185     (void) fprintf(stderr, "\n");

```

```

187     exit(1);
188 }
189
190 /* ARGSUSED */
191 static void
192 dump_packed_nvlist(objset_t *os, uint64_t object, void *data, size_t size)
193 {
194     nvlist_t *nv;
195     size_t nvsize = *(uint64_t *)data;
196     char *packed = umem_alloc(nvsize, UMEM_NOFAIL);
197
198     VERIFY(0 == dmu_read(os, object, 0, nvsize, packed, DMU_READ_PREFETCH));
199
200     VERIFY(nvlist_unpack(packed, nvsize, &nv, 0) == 0);
201
202     umem_free(packed, nvsize);
203
204     dump_nvlist(nv, 8);
205
206     nvlist_free(nv);
207 }
208
209 /* ARGSUSED */
210 static void
211 dump_history_offsets(objset_t *os, uint64_t object, void *data, size_t size)
212 {
213     spa_history_phys_t *shp = data;
214
215     if (shp == NULL)
216         return;
217
218     (void) printf("\t\tpool_create_len = %llu\n",
219     (u_longlong_t)shp->sh_pool_create_len);
220     (void) printf("\t\tphys_max_off = %llu\n",
221     (u_longlong_t)shp->sh_phys_max_off);
222     (void) printf("\t\tbof = %llu\n",
223     (u_longlong_t)shp->sh_bof);
224     (void) printf("\t\tteof = %llu\n",
225     (u_longlong_t)shp->sh_eof);
226     (void) printf("\t\trecords_lost = %llu\n",
227     (u_longlong_t)shp->sh_records_lost);
228 }
229
230 #endif /* ! codereview */
231 static void
232 zdb_nicenum(uint64_t num, char *buf)
233 {
234     if (dump_opt['P'])
235         (void) sprintf(buf, "%llu", (longlong_t)num);
236     else
237         nicenum(num, buf);
238 }
239
240 const char dump_zap_stars[] = "*****";
241 const int dump_zap_width = sizeof (dump_zap_stars) - 1;
242
243 static void
244 dump_zap_histogram(uint64_t histo[ZAP_HISTOGRAM_SIZE])
245 {
246     int i;
247     int minidx = ZAP_HISTOGRAM_SIZE - 1;
248     int maxidx = 0;
249     uint64_t max = 0;
250
251     for (i = 0; i < ZAP_HISTOGRAM_SIZE; i++) {
252         if (histo[i] > max)

```

```

253         max = histo[i];
254         if (histo[i] > 0 && i > maxidx)
255             maxidx = i;
256         if (histo[i] > 0 && i < minidx)
257             minidx = i;
258     }
259
260     if (max < dump_zap_width)
261         max = dump_zap_width;
262
263     for (i = minidx; i <= maxidx; i++)
264         (void) printf("\t\t\t%u: %6llu %s\n", i, (u_longlong_t)histo[i],
265             &dump_zap_stars[(max - histo[i]) * dump_zap_width / max]);
266 }
267
268 static void
269 dump_zap_stats(objset_t *os, uint64_t object)
270 {
271     int error;
272     zap_stats_t zs;
273
274     error = zap_get_stats(os, object, &zs);
275     if (error)
276         return;
277
278     if (zs.zs_ptrtbl_len == 0) {
279         ASSERT(zs.zs_num_blocks == 1);
280         (void) printf("\tmicrozap: %llu bytes, %llu entries\n",
281             (u_longlong_t)zs.zs_blocksize,
282             (u_longlong_t)zs.zs_num_entries);
283         return;
284     }
285
286     (void) printf("\tFat ZAP stats:\n");
287
288     (void) printf("\t\tPointer table:\n");
289     (void) printf("\t\t\t%llu elements\n",
290         (u_longlong_t)zs.zs_ptrtbl_len);
291     (void) printf("\t\t\tzblk: %llu\n",
292         (u_longlong_t)zs.zs_ptrtbl_zt_blk);
293     (void) printf("\t\t\tzblk numblks: %llu\n",
294         (u_longlong_t)zs.zs_ptrtbl_zt_numblks);
295     (void) printf("\t\t\tzblk shift: %llu\n",
296         (u_longlong_t)zs.zs_ptrtbl_zt_shift);
297     (void) printf("\t\t\tzblk blks copied: %llu\n",
298         (u_longlong_t)zs.zs_ptrtbl_blks_copied);
299     (void) printf("\t\t\tzblk nextblk: %llu\n",
300         (u_longlong_t)zs.zs_ptrtbl_nextblk);
301
302     (void) printf("\t\tZAP entries: %llu\n",
303         (u_longlong_t)zs.zs_num_entries);
304     (void) printf("\t\tLeaf blocks: %llu\n",
305         (u_longlong_t)zs.zs_num_leafs);
306     (void) printf("\t\tTotal blocks: %llu\n",
307         (u_longlong_t)zs.zs_num_blocks);
308     (void) printf("\t\tzap_block_type: 0x%llx\n",
309         (u_longlong_t)zs.zs_block_type);
310     (void) printf("\t\tzap_magic: 0x%llx\n",
311         (u_longlong_t)zs.zs_magic);
312     (void) printf("\t\tzap_salt: 0x%llx\n",
313         (u_longlong_t)zs.zs_salt);
314
315     (void) printf("\t\tLeafs with 2^n pointers:\n");
316     dump_zap_histogram(zs.zs_leafs_with_2n_pointers);
317
318     (void) printf("\t\tBlocks with n*5 entries:\n");

```

```

319     dump_zap_histogram(zs.zs_blocks_with_n5_entries);
320
321     (void) printf("\t\tBlocks n/10 full:\n");
322     dump_zap_histogram(zs.zs_blocks_n_tenths_full);
323
324     (void) printf("\t\tEntries with n chunks:\n");
325     dump_zap_histogram(zs.zs_entries_using_n_chunks);
326
327     (void) printf("\t\tBuckets with n entries:\n");
328     dump_zap_histogram(zs.zs_buckets_with_n_entries);
329 }
330
331 /*ARGSUSED*/
332 static void
333 dump_none(objset_t *os, uint64_t object, void *data, size_t size)
334 {
335 }
336
337 /*ARGSUSED*/
338 static void
339 dump_unknown(objset_t *os, uint64_t object, void *data, size_t size)
340 {
341     (void) printf("\tUNKNOWN OBJECT TYPE\n");
342 }
343
344 /*ARGSUSED*/
345 void
346 dump_uint8(objset_t *os, uint64_t object, void *data, size_t size)
347 {
348 }
349
350 /*ARGSUSED*/
351 static void
352 dump_uint64(objset_t *os, uint64_t object, void *data, size_t size)
353 {
354 }
355
356 /*ARGSUSED*/
357 static void
358 dump_zap(objset_t *os, uint64_t object, void *data, size_t size)
359 {
360     zap_cursor_t zc;
361     zap_attribute_t attr;
362     void *prop;
363     int i;
364
365     dump_zap_stats(os, object);
366     (void) printf("\n");
367
368     for (zap_cursor_init(&zc, os, object);
369         zap_cursor_retrieve(&zc, &attr) == 0;
370         zap_cursor_advance(&zc)) {
371         (void) printf("\t\t%s = ", attr.za_name);
372         if (attr.za_num_integers == 0) {
373             (void) printf("\n");
374             continue;
375         }
376         prop = umem_zalloc(attr.za_num_integers *
377             attr.za_integer_length, UMEM_NOFAIL);
378         (void) zap_lookup(os, object, attr.za_name,
379             attr.za_integer_length, attr.za_num_integers, prop);
380         if (attr.za_integer_length == 1) {
381             (void) printf("%s", (char *)prop);
382         } else {
383             for (i = 0; i < attr.za_num_integers; i++) {
384                 switch (attr.za_integer_length) {

```

```

385         case 2:
386             (void) printf("%u ",
387                 ((uint16_t *)prop)[i]);
388             break;
389         case 4:
390             (void) printf("%u ",
391                 ((uint32_t *)prop)[i]);
392             break;
393         case 8:
394             (void) printf("%lld ",
395                 (u_longlong_t)((int64_t *)prop)[i]);
396             break;
397     }
398 }
399 }
400 (void) printf("\n");
401 umem_free(prop, attr.za_num_integers * attr.za_integer_length);
402 }
403 zap_cursor_fini(&ztc);
404 }

406 /*ARGSUSED*/
407 static void
408 dump_ddt_zap(objset_t *os, uint64_t object, void *data, size_t size)
409 {
410     dump_zap_stats(os, object);
411     /* contents are printed elsewhere, properly decoded */
412 }

414 /*ARGSUSED*/
415 static void
416 dump_sa_attrs(objset_t *os, uint64_t object, void *data, size_t size)
417 {
418     zap_cursor_t zc;
419     zap_attribute_t attr;

421     dump_zap_stats(os, object);
422     (void) printf("\n");

424     for (zap_cursor_init(&ztc, os, object);
425          zap_cursor_retrieve(&ztc, &attr) == 0;
426          zap_cursor_advance(&ztc)) {
427         (void) printf("\t\t%s = ", attr.za_name);
428         if (attr.za_num_integers == 0) {
429             (void) printf("\n");
430             continue;
431         }
432         (void) printf(" %llx : [%d:%d:%d]\n",
433             (u_longlong_t)attr.za_first_integer,
434             (int)ATTR_LENGTH(attr.za_first_integer),
435             (int)ATTR_BSWAP(attr.za_first_integer),
436             (int)ATTR_NUM(attr.za_first_integer));
437     }
438     zap_cursor_fini(&ztc);
439 }

441 /*ARGSUSED*/
442 static void
443 dump_sa_layouts(objset_t *os, uint64_t object, void *data, size_t size)
444 {
445     zap_cursor_t zc;
446     zap_attribute_t attr;
447     uint16_t *layout_attrs;
448     int i;

449     dump_zap_stats(os, object);

```

```

451     (void) printf("\n");

453     for (zap_cursor_init(&ztc, os, object);
454          zap_cursor_retrieve(&ztc, &attr) == 0;
455          zap_cursor_advance(&ztc)) {
456         (void) printf("\t\t%s = [", attr.za_name);
457         if (attr.za_num_integers == 0) {
458             (void) printf("\n");
459             continue;
460         }

462         VERIFY(attr.za_integer_length == 2);
463         layout_attrs = umem_zalloc(attr.za_num_integers *
464             attr.za_integer_length, UMEM_NOFAIL);

466         VERIFY(zap_lookup(os, object, attr.za_name,
467             attr.za_integer_length,
468             attr.za_num_integers, layout_attrs) == 0);

470         for (i = 0; i != attr.za_num_integers; i++)
471             (void) printf(" %d ", (int)layout_attrs[i]);
472         (void) printf("]\n");
473         umem_free(layout_attrs,
474             attr.za_num_integers * attr.za_integer_length);
475     }
476     zap_cursor_fini(&ztc);
477 }

479 /*ARGSUSED*/
480 static void
481 dump_zpldir(objset_t *os, uint64_t object, void *data, size_t size)
482 {
483     zap_cursor_t zc;
484     zap_attribute_t attr;
485     const char *typenames[] = {
486         /* 0 */ "not specified",
487         /* 1 */ "FIFO",
488         /* 2 */ "Character Device",
489         /* 3 */ "3 (invalid)",
490         /* 4 */ "Directory",
491         /* 5 */ "5 (invalid)",
492         /* 6 */ "Block Device",
493         /* 7 */ "7 (invalid)",
494         /* 8 */ "Regular File",
495         /* 9 */ "9 (invalid)",
496         /* 10 */ "Symbolic Link",
497         /* 11 */ "11 (invalid)",
498         /* 12 */ "Socket",
499         /* 13 */ "Door",
500         /* 14 */ "Event Port",
501         /* 15 */ "15 (invalid)",
502     };

504     dump_zap_stats(os, object);
505     (void) printf("\n");

507     for (zap_cursor_init(&ztc, os, object);
508          zap_cursor_retrieve(&ztc, &attr) == 0;
509          zap_cursor_advance(&ztc)) {
510         (void) printf("\t\t%s = %lld (type: %s)\n",
511             attr.za_name, ZFS_DIRENT_OBJ(attr.za_first_integer),
512             typenames[ZFS_DIRENT_TYPE(attr.za_first_integer)]);
513     }
514     zap_cursor_fini(&ztc);
515 }

```

```

517 static void
518 dump_spacemap(objset_t *os, space_map_obj_t *smo, space_map_t *sm)
519 {
520     uint64_t alloc, offset, entry;
521     uint8_t mapshift = sm->sm_shift;
522     uint64_t mapstart = sm->sm_start;
523     char *ddata[] = { "ALLOC", "FREE", "CONDENSE", "INVALID",
524                     "INVALID", "INVALID", "INVALID", "INVALID" };
525
526     if (smo->smo_object == 0)
527         return;
528
529     /*
530      * Print out the freelist entries in both encoded and decoded form.
531      */
532     alloc = 0;
533     for (offset = 0; offset < smo->smo_objsize; offset += sizeof (entry)) {
534         VERIFY3U(0, ==, dmu_read(os, smo->smo_object, offset,
535             sizeof (entry), &entry, DMU_READ_PREFETCH));
536         if (SM_DEBUG_DECODE(entry)) {
537             (void) printf("\t [%6llu] %s: txg %llu, pass %llu\n",
538                 (u_longlong_t)(offset / sizeof (entry)),
539                 ddata[SM_DEBUG_ACTION_DECODE(entry)],
540                 (u_longlong_t)SM_DEBUG_TXG_DECODE(entry),
541                 (u_longlong_t)SM_DEBUG_SYNCPASS_DECODE(entry));
542         } else {
543             (void) printf("\t [%6llu] %c range:"
544                 " %010llx-%010llx size: %06llx\n",
545                 (u_longlong_t)(offset / sizeof (entry)),
546                 SM_TYPE_DECODE(entry) == SM_ALLOC ? 'A' : 'F',
547                 (u_longlong_t)((SM_OFFSET_DECODE(entry) <<
548                     mapshift) + mapstart),
549                 (u_longlong_t)((SM_OFFSET_DECODE(entry) <<
550                     mapshift) + mapstart + (SM_RUN_DECODE(entry) <<
551                     mapshift)),
552                 (u_longlong_t)(SM_RUN_DECODE(entry) << mapshift));
553             if (SM_TYPE_DECODE(entry) == SM_ALLOC)
554                 alloc += SM_RUN_DECODE(entry) << mapshift;
555             else
556                 alloc -= SM_RUN_DECODE(entry) << mapshift;
557         }
558     }
559     if (alloc != smo->smo_alloc) {
560         (void) printf("space_map_object alloc (%llu) INCONSISTENT "
561             "with space map summary (%llu)\n",
562             (u_longlong_t)smo->smo_alloc, (u_longlong_t)alloc);
563     }
564 }
565
566 static void
567 dump_metaslab_stats(metaslab_t *msp)
568 {
569     char maxbuf[32];
570     space_map_t *sm = &msp->ms_map;
571     avl_tree_t *t = sm->sm_pp_root;
572     int free_pct = sm->sm_space * 100 / sm->sm_size;
573
574     zdb_nicenum(space_map_maxsize(sm), maxbuf);
575
576     (void) printf("\t %25s %10lu %7s %6s %4s %4d%%\n",
577         "segments", avl_numnodes(t), "maxsize", maxbuf,
578         "freepct", free_pct);
579 }
580
581 static void
582 dump_metaslab(metaslab_t *msp)

```

```

583 {
584     vdev_t *vd = msp->ms_group->mg_vd;
585     spa_t *spa = vd->vdev_spa;
586     space_map_t *sm = &msp->ms_map;
587     space_map_obj_t *smo = &msp->ms_smo;
588     char freebuf[32];
589
590     zdb_nicenum(sm->sm_size - smo->smo_alloc, freebuf);
591
592     (void) printf(
593         "\tmetaslab %6llu offset %21lx spacemap %6llu free %5s\n",
594         (u_longlong_t)(sm->sm_start / sm->sm_size),
595         (u_longlong_t)sm->sm_start, (u_longlong_t)smo->smo_object, freebuf);
596
597     if (dump_opt['m'] > 1 && !dump_opt['L']) {
598         mutex_enter(&msp->ms_lock);
599         space_map_load_wait(sm);
600         if (!sm->sm_loaded)
601             VERIFY(space_map_load(sm, zfs_metaslab_ops,
602                 SM_FREE, smo, spa->spa_meta_objset) == 0);
603         dump_metaslab_stats(msp);
604         space_map_unload(sm);
605         mutex_exit(&msp->ms_lock);
606     }
607
608     if (dump_opt['d'] > 5 || dump_opt['m'] > 2) {
609         ASSERT(sm->sm_size == (LULL << vd->vdev_ms_shift));
610
611         mutex_enter(&msp->ms_lock);
612         dump_spacemap(spa->spa_meta_objset, smo, sm);
613         mutex_exit(&msp->ms_lock);
614     }
615 }
616
617 static void
618 print_vdev_metaslab_header(vdev_t *vd)
619 {
620     (void) printf("\tvdev %10llu\n\t%-10s%5llu %-19s %-15s %-10s\n",
621         (u_longlong_t)vd->vdev_id,
622         "metaslabs", (u_longlong_t)vd->vdev_ms_count,
623         "offset", "spacemap", "free");
624     (void) printf("\t%15s %19s %15s %10s\n",
625         "-----", "-----",
626         "-----", "-----");
627 }
628
629 static void
630 dump_metaslabs(spa_t *spa)
631 {
632     vdev_t *vd, *rvd = spa->spa_root_vdev;
633     uint64_t m, c = 0, children = rvd->vdev_children;
634
635     (void) printf("\nMetaslabs:\n");
636
637     if (!dump_opt['d'] && zopt_objects > 0) {
638         c = zopt_object[0];
639
640         if (c >= children)
641             (void) fatal("bad vdev id: %llu", (u_longlong_t)c);
642
643         if (zopt_objects > 1) {
644             vd = rvd->vdev_child[c];
645             print_vdev_metaslab_header(vd);
646
647             for (m = 1; m < zopt_objects; m++) {
648                 if (zopt_object[m] < vd->vdev_ms_count)

```

```

649         dump metaslab(
650             vd->vdev_ms[zopt_object[m]]);
651     else
652         (void) fprintf(stderr, "bad metaslab "
653             "number %llu\n",
654             (u_longlong_t)zopt_object[m]);
655     }
656     (void) printf("\n");
657     return;
658 }
659     children = c + 1;
660 }
661 for (; c < children; c++) {
662     vd = rvd->vdev_child[c];
663     print_vdev_metaslab_header(vd);
664
665     for (m = 0; m < vd->vdev_ms_count; m++)
666         dump_metaslab(vd->vdev_ms[m]);
667     (void) printf("\n");
668 }
669 }
670
671 static void
672 dump_dde(const ddt_t *ddt, const ddt_entry_t *dde, uint64_t index)
673 {
674     const ddt_phys_t *ddp = dde->dde_phys;
675     const ddt_key_t *ddk = &dde->dde_key;
676     char *types[4] = { "ditto", "single", "double", "triple" };
677     char blkbuf[BP_SPRINTF_LEN];
678     blkptr_t blk;
679
680     for (int p = 0; p < DDT_PHYS_TYPES; p++, ddp++) {
681         if (ddp->ddp_phys_birth == 0)
682             continue;
683         ddt_bp_create(ddt->ddt_checksum, ddk, ddp, &blk);
684         sprintf_blkbuf(blkbuf, &blk);
685         (void) printf("index %llx refcnt %llu %s %s\n",
686             (u_longlong_t)index, (u_longlong_t)ddp->ddp_refcnt,
687             types[p], blkbuf);
688     }
689 }
690
691 static void
692 dump_dedup_ratio(const ddt_stat_t *dds)
693 {
694     double rL, rP, rD, D, dedup, compress, copies;
695
696     if (dds->dds_blocks == 0)
697         return;
698
699     rL = (double)dds->dds_ref_lsize;
700     rP = (double)dds->dds_ref_psize;
701     rD = (double)dds->dds_ref_dsize;
702     D = (double)dds->dds_dsize;
703
704     dedup = rD / D;
705     compress = rL / rP;
706     copies = rD / rP;
707
708     (void) printf("dedup = %.2f, compress = %.2f, copies = %.2f, "
709         "dedup * compress / copies = %.2f\n\n",
710         dedup, compress, copies, dedup * compress / copies);
711 }
712
713 static void
714 dump_ddt(ddt_t *ddt, enum ddt_type type, enum ddt_class class)

```

```

715 {
716     char name[DDT_NAMELEN];
717     ddt_entry_t dde;
718     uint64_t walk = 0;
719     dmu_object_info_t doi;
720     uint64_t count, dspace, mspace;
721     int error;
722
723     error = ddt_object_info(ddt, type, class, &doi);
724
725     if (error == ENOENT)
726         return;
727     ASSERT(error == 0);
728
729     if ((count = ddt_object_count(ddt, type, class)) == 0)
730         return;
731
732     dspace = doi.doi_physical_blocks_512 << 9;
733     mspace = doi.doi_fill_count * doi.doi_data_block_size;
734
735     ddt_object_name(ddt, type, class, name);
736
737     (void) printf("%s: %llu entries, size %llu on disk, %llu in core\n",
738         name,
739         (u_longlong_t)count,
740         (u_longlong_t)(dspace / count),
741         (u_longlong_t)(mspace / count));
742
743     if (dump_opt['D'] < 3)
744         return;
745
746     zpool_dump_ddt(NULL, &ddt->ddt_histogram[type][class]);
747
748     if (dump_opt['D'] < 4)
749         return;
750
751     if (dump_opt['D'] < 5 && class == DDT_CLASS_UNIQUE)
752         return;
753
754     (void) printf("%s contents:\n\n", name);
755
756     while ((error = ddt_object_walk(ddt, type, class, &walk, &dde)) == 0)
757         dump_dde(ddt, &dde, walk);
758
759     ASSERT(error == ENOENT);
760
761     (void) printf("\n");
762 }
763
764 static void
765 dump_all_ddts(spa_t *spa)
766 {
767     ddt_histogram_t ddh_total = { 0 };
768     ddt_stat_t dds_total = { 0 };
769
770     for (enum zio_checksum c = 0; c < ZIO_CHECKSUM_FUNCTIONS; c++) {
771         ddt_t *ddt = spa->spa_ddt[c];
772         for (enum ddt_type type = 0; type < DDT_TYPES; type++) {
773             for (enum ddt_class class = 0; class < DDT_CLASSES;
774                 class++) {
775                 dump_ddt(ddt, type, class);
776             }
777         }
778     }
779
780     ddt_get_dedup_stats(spa, &dds_total);

```

```

782     if (dds_total.dds_blocks == 0) {
783         (void) printf("All DDTs are empty\n");
784         return;
785     }
787     (void) printf("\n");
789     if (dump_opt['D'] > 1) {
790         (void) printf("DDT histogram (aggregated over all DDTs):\n");
791         ddt_get_dedup_histogram(spa, &ddh_total);
792         zpool_dump_ddt(&dds_total, &ddh_total);
793     }
795     dump_dedup_ratio(&dds_total);
796 }
798 static void
799 dump_dtl_seg(space_map_t *sm, uint64_t start, uint64_t size)
800 {
801     char *prefix = (void *)sm;
803     (void) printf("%s [%llu,%llu] length %llu\n",
804                 prefix,
805                 (u_longlong_t)start,
806                 (u_longlong_t)(start + size),
807                 (u_longlong_t)(size));
808 }
810 static void
811 dump_dtl(vdev_t *vd, int indent)
812 {
813     spa_t *spa = vd->vdev_spa;
814     boolean_t required;
815     char *name[DTL_TYPES] = { "missing", "partial", "scrub", "outage" };
816     char prefix[256];
818     spa_vdev_state_enter(spa, SCL_NONE);
819     required = vdev_dtl_required(vd);
820     (void) spa_vdev_state_exit(spa, NULL, 0);
822     if (indent == 0)
823         (void) printf("\nDirty time logs:\n\n");
825     (void) printf("\t\t*s*s [%s]\n", indent, "",
826                 vd->vdev_path ? vd->vdev_path :
827                 vd->vdev_parent ? vd->vdev_ops->vdev_op_type : spa_name(spa),
828                 required ? "DTL-required" : "DTL-expendable");
830     for (int t = 0; t < DTL_TYPES; t++) {
831         space_map_t *sm = &vd->vdev_dtl[t];
832         if (sm->sm_space == 0)
833             continue;
834         (void) snprintf(prefix, sizeof (prefix), "\t\t*s*s",
835                         indent + 2, "", name[t]);
836         mutex_enter(sm->sm_lock);
837         space_map_walk(sm, dump_dtl_seg, (void *)prefix);
838         mutex_exit(sm->sm_lock);
839         if (dump_opt['d'] > 5 && vd->vdev_children == 0)
840             dump_spacemap(spa->spa_meta_objset,
841                         &vd->vdev_dtl_smo, sm);
842     }
844     for (int c = 0; c < vd->vdev_children; c++)
845         dump_dtl(vd->vdev_child[c], indent + 4);
846 }

```

```

848 static void
849 dump_history(spa_t *spa)
850 {
851     nvlist_t **events = NULL;
852     char buf[SPA_MAXBLOCKSIZE];
853     uint64_t resid, len, off = 0;
854     uint_t num = 0;
855     int error;
856     time_t tsec;
857     struct tm t;
858     char tbuf[30];
859     char internalstr[MAXPATHLEN];
861     do {
862         len = sizeof (buf);
864         if ((error = spa_history_get(spa, &off, &len, buf)) != 0) {
865             (void) fprintf(stderr, "Unable to read history: "
866                             "error %d\n", error);
867             return;
868         }
870         if (zpool_history_unpack(buf, len, &resid, &events, &num) != 0)
871             break;
873         off -= resid;
874     } while (len != 0);
876     (void) printf("\nHistory:\n");
877     for (int i = 0; i < num; i++) {
878         uint64_t time, txg, ievent;
879         char *cmd, *intstr;
880         boolean_t printed = B_FALSE;
881         #endif /* !codereview */
883         if (nvlist_lookup_uint64(events[i], ZPOOL_HIST_TIME,
884                                 &time) != 0)
885             goto next;
886         continue;
888         if (nvlist_lookup_string(events[i], ZPOOL_HIST_CMD,
889                                 &cmd) != 0) {
890             if (nvlist_lookup_uint64(events[i],
891                                     ZPOOL_HIST_INT_EVENT, &ievent) != 0)
892                 goto next;
893             continue;
894             verify(nvlist_lookup_uint64(events[i],
895                                         ZPOOL_HIST_TXG, &txg) == 0);
896             verify(nvlist_lookup_string(events[i],
897                                         ZPOOL_HIST_INT_STR, &intstr) == 0);
898             if (ievent >= ZFS_NUM_LEGACY_HISTORY_EVENTS)
899                 goto next;
900             if (ievent >= LOG_END)
901                 continue;
902         }
903         (void) snprintf(internalstr,
904                         sizeof (internalstr),
905                         "[internal %s txg:%lld] %s",
906                         zfs_history_event_names[ievent], txg,
907                         intstr);
908         cmd = internalstr;
909     }
910     tsec = time;
911     (void) localtime_r(&tsec, &t);
912     (void) strftime(tbuf, sizeof (tbuf), "%F.%T", &t);
913     (void) printf("%s %s\n", tbuf, cmd);

```

```

909         printed = B_TRUE;
910
911 next:
912         if (dump_opt['h'] > 1) {
913             if (!printed)
914                 (void) printf("unrecognized record:\n");
915             dump_nvlist(events[i], 2);
916         }
917 #endif /* ! codereview */
918     }
919 }
920
921 /*ARGSUSED*/
922 static void
923 dump_dnode(objset_t *os, uint64_t object, void *data, size_t size)
924 {
925 }
926
927 static uint64_t
928 blkid2offset(const dnode_phys_t *dnp, const blkptr_t *bp, const zbookmark_t *zb)
929 {
930     if (dnp == NULL) {
931         ASSERT(zb->zb_level < 0);
932         if (zb->zb_object == 0)
933             return (zb->zb_blkid);
934         return (zb->zb_blkid * BP_GET_LSIZE(bp));
935     }
936
937     ASSERT(zb->zb_level >= 0);
938
939     return ((zb->zb_blkid <<
940         (zb->zb_level * (dnp->dn_indblkshift - SPA_BLKPTRSHIFT))) *
941         dnp->dn_datablkssize << SPA_MINBLOCKSHIFT);
942 }
943
944 static void
945 sprintf_blkptr_compact(char *blkbuf, const blkptr_t *bp)
946 {
947     const dva_t *dva = bp->blk_dva;
948     int ndvas = dump_opt['d'] > 5 ? BP_GET_NDVAS(bp) : 1;
949
950     if (dump_opt['b'] >= 5) {
951         sprintf_blkptr(blkbuf, bp);
952         return;
953     }
954
955     blkbuf[0] = '\0';
956
957     for (int i = 0; i < ndvas; i++)
958         (void) sprintf(blkbuf + strlen(blkbuf), "%llu:%llx:%llx ",
959             (u_longlong_t)DVA_GET_VDEV(&dva[i]),
960             (u_longlong_t)DVA_GET_OFFSET(&dva[i]),
961             (u_longlong_t)DVA_GET_ASIZE(&dva[i]));
962
963     (void) sprintf(blkbuf + strlen(blkbuf),
964         "%llxL/%llxP F=%llu B=%llu/%llu",
965         (u_longlong_t)BP_GET_LSIZE(bp),
966         (u_longlong_t)BP_GET_PSIZE(bp),
967         (u_longlong_t)bp->blk_fill,
968         (u_longlong_t)bp->blk_birth,
969         (u_longlong_t)BP_PHYSICAL_BIRTH(bp));
970 }
971
972 static void
973 print_indirect(blkptr_t *bp, const zbookmark_t *zb,
974     const dnode_phys_t *dnp)

```

```

975 {
976     char blkbuf[BP_SPRINTF_LEN];
977     int l;
978
979     ASSERT3U(BP_GET_TYPE(bp), ==, dnp->dn_type);
980     ASSERT3U(BP_GET_LEVEL(bp), ==, zb->zb_level);
981
982     (void) printf("%16llx ", (u_longlong_t)blkid2offset(dnp, bp, zb));
983
984     ASSERT(zb->zb_level >= 0);
985
986     for (l = dnp->dn_nlevels - 1; l >= -1; l--) {
987         if (l == zb->zb_level) {
988             (void) printf("L%11lx", (u_longlong_t)zb->zb_level);
989         } else {
990             (void) printf(" ");
991         }
992     }
993
994     sprintf_blkptr_compact(blkbuf, bp);
995     (void) printf("%s\n", blkbuf);
996 }
997
998 static int
999 visit_indirect(spa_t *spa, const dnode_phys_t *dnp,
1000     blkptr_t *bp, const zbookmark_t *zb)
1001 {
1002     int err = 0;
1003
1004     if (bp->blk_birth == 0)
1005         return (0);
1006
1007     print_indirect(bp, zb, dnp);
1008
1009     if (BP_GET_LEVEL(bp) > 0) {
1010         uint32_t flags = ARC_WAIT;
1011         int i;
1012         blkptr_t *cbp;
1013         int epb = BP_GET_LSIZE(bp) >> SPA_BLKPTRSHIFT;
1014         arc_buf_t *buf;
1015         uint64_t fill = 0;
1016
1017         err = arc_read_nolock(NULL, spa, bp, arc_getbuf_func, &buf,
1018             ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL, &flags, zb);
1019         if (err)
1020             return (err);
1021         ASSERT(buf->b_data);
1022
1023         /* recursively visit blocks below this */
1024         cbp = buf->b_data;
1025         for (i = 0; i < epb; i++, cbp++) {
1026             zbookmark_t czb;
1027
1028             SET_BOOKMARK(&czb, zb->zb_objset, zb->zb_object,
1029                 zb->zb_level - 1,
1030                 zb->zb_blkid * epb + i);
1031             err = visit_indirect(spa, dnp, cbp, &czb);
1032             if (err)
1033                 break;
1034             fill += cbp->blk_fill;
1035         }
1036         if (!err)
1037             ASSERT3U(fill, ==, bp->blk_fill);
1038         (void) arc_buf_remove_ref(buf, &buf);
1039     }

```



```

1173 static int
1174 dump_bptree_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
1175 {
1176     char blkbuf[BP_SPRINTF_LEN];
1177
1178     if (bp->blk_birth != 0) {
1179         sprintf_blkptr(blkbuf, bp);
1180         (void) printf("\t%s\n", blkbuf);
1181     }
1182     return (0);
1183 }
1184
1185 static void
1186 dump_bptree(objset_t *os, uint64_t obj, char *name)
1187 {
1188     char bytes[32];
1189     bptree_phys_t *bt;
1190     dmu_buf_t *db;
1191
1192     if (dump_opt['d'] < 3)
1193         return;
1194
1195     VERIFY3U(0, ==, dmu_bonus_hold(os, obj, FTAG, &db));
1196     bt = db->db_data;
1197     zdb_nicenum(bt->bt_bytes, bytes);
1198     (void) printf("\n    %s: %llu datasets, %s\n",
1199         name, (unsigned long long)(bt->bt_end - bt->bt_begin), bytes);
1200     dmu_buf_rele(db, FTAG);
1201
1202     if (dump_opt['d'] < 5)
1203         return;
1204
1205     (void) printf("\n");
1206
1207     (void) bptree_iterate(os, obj, B_FALSE, dump_bptree_cb, NULL, NULL);
1208 }
1209
1210 /* ARGSUSED */
1211 static int
1212 dump_bpobj_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
1213 {
1214     char blkbuf[BP_SPRINTF_LEN];
1215
1216     ASSERT(bp->blk_birth != 0);
1217     sprintf_blkptr_compact(blkbuf, bp);
1218     (void) printf("\t%s\n", blkbuf);
1219     return (0);
1220 }
1221
1222 static void
1223 dump_bpobj(bpobj_t *bpo, char *name)
1224 {
1225     char bytes[32];
1226     char comp[32];
1227     char uncomp[32];
1228
1229     if (dump_opt['d'] < 3)
1230         return;
1231
1232     zdb_nicenum(bpo->bpo_phys->bpo_bytes, bytes);
1233     if (bpo->bpo_havesubobj) {
1234         zdb_nicenum(bpo->bpo_phys->bpo_comp, comp);
1235         zdb_nicenum(bpo->bpo_phys->bpo_uncomp, uncomp);
1236         (void) printf("\n    %s: %llu local blkptrs, %llu subobjs, "
1237             "%s (%s/%s comp)\n",
1238             name, (u_longlong_t)bpo->bpo_phys->bpo_num_blkptrs,

```

```

1239         (u_longlong_t)bpo->bpo_phys->bpo_num_subobjs,
1240         bytes, comp, uncomp);
1241     } else {
1242         (void) printf("\n    %s: %llu blkptrs, %s\n",
1243             name, (u_longlong_t)bpo->bpo_phys->bpo_num_blkptrs, bytes);
1244     }
1245
1246     if (dump_opt['d'] < 5)
1247         return;
1248
1249     (void) printf("\n");
1250
1251     (void) bpobj_iterate_nofree(bpo, dump_bpobj_cb, NULL, NULL);
1252 }
1253
1254 static void
1255 dump_deadlist(dsl_deadlist_t *dl)
1256 {
1257     dsl_deadlist_entry_t *dle;
1258     char bytes[32];
1259     char comp[32];
1260     char uncomp[32];
1261
1262     if (dump_opt['d'] < 3)
1263         return;
1264
1265     zdb_nicenum(dl->dl_phys->dl_used, bytes);
1266     zdb_nicenum(dl->dl_phys->dl_comp, comp);
1267     zdb_nicenum(dl->dl_phys->dl_uncomp, uncomp);
1268     (void) printf("\n    Deadlist: %s (%s/%s comp)\n",
1269         bytes, comp, uncomp);
1270
1271     if (dump_opt['d'] < 4)
1272         return;
1273
1274     (void) printf("\n");
1275
1276     for (dle = avl_first(&dl->dl_tree); dle;
1277         dle = AVL_NEXT(&dl->dl_tree, dle)) {
1278         (void) printf("        mintxg %llu -> obj %llu\n",
1279             (longlong_t)dle->dle_mintxg,
1280             (longlong_t)dle->dle_bpobj.bpo_object);
1281
1282         if (dump_opt['d'] >= 5)
1283             dump_bpobj(&dle->dle_bpobj, "");
1284     }
1285 }
1286
1287 static avl_tree_t idx_tree;
1288 static avl_tree_t domain_tree;
1289 static boolean_t fuid_table_loaded;
1290 static boolean_t sa_loaded;
1291 sa_attr_type_t *sa_attr_table;
1292
1293 static void
1294 fuid_table_destroy()
1295 {
1296     if (fuid_table_loaded) {
1297         zfs_fuid_table_destroy(&idx_tree, &domain_tree);
1298         fuid_table_loaded = B_FALSE;
1299     }
1300 }
1301
1302 /*
1303  * print uid or gid information.
1304  * For normal POSIX id just the id is printed in decimal format.

```

```

1305 * For CIFS files with FUID the fuid is printed in hex followed by
1306 * the domain-rid string.
1307 */
1308 static void
1309 print_idstr(uint64_t id, const char *id_type)
1310 {
1311     if (FUID_INDEX(id)) {
1312         char *domain;
1313
1314         domain = zfs_fuid_idx_domain(&idx_tree, FUID_INDEX(id));
1315         (void) printf("\t%s    %llx [%s-%d]\n", id_type,
1316             (u_longlong_t)id, domain, (int)FUID_RID(id));
1317     } else {
1318         (void) printf("\t%s    %llu\n", id_type, (u_longlong_t)id);
1319     }
1320 }
1321
1322 static void
1323 dump_uidgid(objset_t *os, uint64_t uid, uint64_t gid)
1324 {
1325     uint32_t uid_idx, gid_idx;
1326
1327     uid_idx = FUID_INDEX(uid);
1328     gid_idx = FUID_INDEX(gid);
1329
1330     /* Load domain table, if not already loaded */
1331     if (!fuid_table_loaded && (uid_idx || gid_idx)) {
1332         uint64_t fuid_obj;
1333
1334         /* first find the fuid object. It lives in the master node */
1335         VERIFY(zap_lookup(os, MASTER_NODE_OBJ, ZFS_FUID_TABLES,
1336             8, 1, &fuid_obj) == 0);
1337         zfs_fuid_avl_tree_create(&idx_tree, &domain_tree);
1338         (void) zfs_fuid_table_load(os, fuid_obj,
1339             &idx_tree, &domain_tree);
1340         fuid_table_loaded = B_TRUE;
1341     }
1342
1343     print_idstr(uid, "uid");
1344     print_idstr(gid, "gid");
1345 }
1346
1347 /*ARGSUSED*/
1348 static void
1349 dump_znode(objset_t *os, uint64_t object, void *data, size_t size)
1350 {
1351     char path[MAXPATHLEN * 2]; /* allow for xattr and failure prefix */
1352     sa_handle_t *hdl;
1353     uint64_t xattr, rdev, gen;
1354     uint64_t uid, gid, mode, fsize, parent, links;
1355     uint64_t pflags;
1356     uint64_t acctm[2], modtm[2], chgtm[2], crtm[2];
1357     time_t z_crtime, z_atime, z_mtime, z_ctime;
1358     sa_bulk_attr_t bulk[12];
1359     int idx = 0;
1360     int error;
1361
1362     if (!sa_loaded) {
1363         uint64_t sa_attrs = 0;
1364         uint64_t version;
1365
1366         VERIFY(zap_lookup(os, MASTER_NODE_OBJ, ZPL_VERSION_STR,
1367             8, 1, &version) == 0);
1368         if (version >= ZPL_VERSION_SA) {
1369             VERIFY(zap_lookup(os, MASTER_NODE_OBJ, ZFS_SA_ATTRS,

```

```

1371         8, 1, &sa_attrs) == 0);
1372     }
1373     if ((error = sa_setup(os, sa_attrs, zfs_attr_table,
1374         ZPL_END, &sa_attr_table)) != 0) {
1375         (void) printf("sa_setup failed errno %d, can't "
1376             "display znode contents\n", error);
1377         return;
1378     }
1379     sa_loaded = B_TRUE;
1380 }
1381
1382 if (sa_handle_get(os, object, NULL, SA_HDL_PRIVATE, &hdl)) {
1383     (void) printf("Failed to get handle for SA znode\n");
1384     return;
1385 }
1386
1387 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_UID], NULL, &uid, 8);
1388 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_GID], NULL, &gid, 8);
1389 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_LINKS], NULL,
1390     &links, 8);
1391 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_GEN], NULL, &gen, 8);
1392 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_MODE], NULL,
1393     &mode, 8);
1394 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_PARENT],
1395     NULL, &parent, 8);
1396 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_SIZE], NULL,
1397     &fsize, 8);
1398 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_ATIME], NULL,
1399     &acctm, 16);
1400 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_MTIME], NULL,
1401     &modtm, 16);
1402 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_CRTIME], NULL,
1403     &crtm, 16);
1404 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_CTIME], NULL,
1405     &chgtm, 16);
1406 SA_ADD_BULK_ATTR(bulk, idx, sa_attr_table[ZPL_FLAGS], NULL,
1407     &pflags, 8);
1408
1409 if (sa_bulk_lookup(hdl, bulk, idx)) {
1410     (void) sa_handle_destroy(hdl);
1411     return;
1412 }
1413
1414 error = zfs_obj_to_path(os, object, path, sizeof(path));
1415 if (error != 0) {
1416     (void) snprintf(path, sizeof(path), "?\??\?<object#%llu>",
1417         (u_longlong_t)object);
1418 }
1419 if (dump_opt['d'] < 3) {
1420     (void) printf("\t%s\n", path);
1421     (void) sa_handle_destroy(hdl);
1422     return;
1423 }
1424
1425 z_crtime = (time_t)crtm[0];
1426 z_atime = (time_t)acctm[0];
1427 z_mtime = (time_t)modtm[0];
1428 z_ctime = (time_t)chgtm[0];
1429
1430 (void) printf("\tpath %s\n", path);
1431 dump_uidgid(os, uid, gid);
1432 (void) printf("\tatime %s", ctime(&z_atime));
1433 (void) printf("\tmtime %s", ctime(&z_mtime));
1434 (void) printf("\tctime %s", ctime(&z_ctime));
1435 (void) printf("\tcrtime %s", ctime(&z_crtime));
1436 (void) printf("\tgen %llu\n", (u_longlong_t)gen);

```

```

1437 (void) printf("\tmode %llu\n", (u_longlong_t)mode);
1438 (void) printf("\tsize %llu\n", (u_longlong_t)fsize);
1439 (void) printf("\tparent %llu\n", (u_longlong_t)parent);
1440 (void) printf("\tlinks %llu\n", (u_longlong_t)links);
1441 (void) printf("\tflags %llx\n", (u_longlong_t)pfllags);
1442 if (sa_lookup(hdl, sa_attr_table[ZPL_XATTR], &xattr,
1443             sizeof (uint64_t)) == 0)
1444     (void) printf("\txattr %llu\n", (u_longlong_t)xattr);
1445 if (sa_lookup(hdl, sa_attr_table[ZPL_RDEV], &rdev,
1446             sizeof (uint64_t)) == 0)
1447     (void) printf("\trdev 0x%016llx\n", (u_longlong_t)rdev);
1448 sa_handle_destroy(hdl);
1449 }

1451 /*ARGSUSED*/
1452 static void
1453 dump_acl(objset_t *os, uint64_t object, void *data, size_t size)
1454 {
1455 }

1457 /*ARGSUSED*/
1458 static void
1459 dump_dmu_objset(objset_t *os, uint64_t object, void *data, size_t size)
1460 {
1461 }

1463 static object_viewer_t *object_viewer[DMU_OT_NUMTYPES + 1] = {
1464     dump_none, /* unallocated */
1465     dump_zap, /* object directory */
1466     dump_uint64, /* object array */
1467     dump_none, /* packed nvlist */
1468     dump_packed_nvlist, /* packed nvlist size */
1469     dump_none, /* bplist */
1470     dump_none, /* bplist header */
1471     dump_none, /* SPA space map header */
1472     dump_none, /* SPA space map */
1473     dump_none, /* ZIL intent log */
1474     dump_dnode, /* DMU dnode */
1475     dump_dmu_objset, /* DMU objset */
1476     dump_dsl_dir, /* DSL directory */
1477     dump_zap, /* DSL directory child map */
1478     dump_zap, /* DSL dataset snap map */
1479     dump_zap, /* DSL props */
1480     dump_dsl_dataset, /* DSL dataset */
1481     dump_znode, /* ZFS znode */
1482     dump_acl, /* ZFS V0 ACL */
1483     dump_uint8, /* ZFS plain file */
1484     dump_zpldir, /* ZFS directory */
1485     dump_zap, /* ZFS master node */
1486     dump_zap, /* ZFS delete queue */
1487     dump_uint8, /* zvol object */
1488     dump_zap, /* zvol prop */
1489     dump_uint8, /* other uint8[] */
1490     dump_uint64, /* other uint64[] */
1491     dump_zap, /* other ZAP */
1492     dump_zap, /* persistent error log */
1493     dump_uint8, /* SPA history */
1494     dump_history_offsets, /* SPA history offsets */
1495     84, /* SPA history offsets */
1496     dump_zap, /* Pool properties */
1497     dump_zap, /* DSL permissions */
1498     dump_acl, /* ZFS ACL */
1499     dump_uint8, /* ZFS SYSACL */
1500     dump_none, /* FUID nvlist */
1501     dump_packed_nvlist, /* FUID nvlist size */
1502     dump_zap, /* DSL dataset next clones */

```

```

1502     dump_zap, /* DSL scrub queue */
1503     dump_zap, /* ZFS user/group used */
1504     dump_zap, /* ZFS user/group quota */
1505     dump_zap, /* snapshot refcount tags */
1506     dump_ddt_zap, /* DDT ZAP object */
1507     dump_zap, /* DDT statistics */
1508     dump_znode, /* SA object */
1509     dump_zap, /* SA Master Node */
1510     dump_sa_attrs, /* SA attribute registration */
1511     dump_sa_layouts, /* SA attribute layouts */
1512     dump_zap, /* DSL scrub translations */
1513     dump_none, /* fake dedup BP */
1514     dump_zap, /* deadlist */
1515     dump_none, /* deadlist hdr */
1516     dump_zap, /* dsl clones */
1517     dump_none, /* bpojb subobjs */
1518     dump_unknown, /* Unknown type, must be last */
1519 };

```

unchanged_portion_omitted

```

*****
2476 Thu Jun 28 15:57:06 2012
new/usr/src/cmd/zfs/Makefile
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 # Copyright 2010 Sun Microsystems, Inc. All rights reserved.
22 # Use is subject to license terms.
23 #endif /* !codereview */
24 #
21 # Copyright 2010 Sun Microsystems, Inc. All rights reserved.
25 # Copyright 2010 Nexenta Systems, Inc. All rights reserved.
26 # Copyright (c) 2012 by Delphix. All rights reserved.
27 # Use is subject to license terms.
28 #

29 PROG=          zfs
30 OBJS=          zfs_main.o zfs_iter.o
31 SRCS=          $(OBJS:%.o=%.c)
32 POFILES=      zfs_main.po zfs_iter.po
33 POFILE=       zfs.po

35 include ../Makefile.cmd
36 include ../Makefile.ctf

38 FSTYPE=        zfs
39 LINKPROGS=     mount umount
40 ROOTETCFSTYPE= $(ROOTETC)/fs/$(FSTYPE)
41 USRLIBFSTYPE=  $(ROOTLIB)/fs/$(FSTYPE)

43 LDLIBS += -lzfs_core -lzfs -luutil -lumem -lnvpair -lsec -lidmap
40 LDLIBS += -lzfs -luutil -lumem -lnvpair -lsec -lidmap

45 INCS += -I../common/zfs

47 C99MODE=       -xc99=%all
48 C99LMODE=      -Xc99=%all

50 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT $(INCS)
51 $(NOT_RELEASE_BUILD)CPPFLAGS += -DDEBUG

```

```

53 # lint complains about unused _umem_* functions
54 LINTFLAGS += -xeroff=E_NAME_DEF_NOT_USED2
55 LINTFLAGS64 += -xeroff=E_NAME_DEF_NOT_USED2

57 ROOTUSRSBINLINKS = $(PROG:%=$(ROOTUSRSBIN)/%)
58 USRLIBFSTYPELINKS = $(LINKPROGS:%=$(USRLIBFSTYPE)/%)
59 ROOTETCFSTYPELINKS = $(LINKPROGS:%=$(ROOTETCFSTYPE)/%)

61 .KEEP_STATE:

63 .PARALLEL:

65 all: $(PROG)

67 $(PROG): $(OBJS)
68     $(LINK.c) -o $@ $(OBJS) $(LDLIBS)
69     $(POST_PROCESS)

71 install: all $(ROOTSBINPROG) $(ROOTUSRSBINLINKS) $(USRLIBFSTYPELINKS) \
72     $(ROOTETCFSTYPELINKS)

74 $(POFILE): $(POFILES)
75     $(RM) $@
76     cat $(POFILES) > $@

78 clean:
79     $(RM) $(OBJS)

81 lint: lint_SRCS

83 # Links from /usr/sbin to /sbin
84 $(ROOTUSRSBINLINKS):
85     -$(RM) $@; $(SYMLINK) ../../sbin/$(PROG) $@

87 # Links from /usr/lib/fs/zfs to /sbin
88 $(USRLIBFSTYPELINKS):
89     -$(RM) $@; $(SYMLINK) ../../../../sbin/$(PROG) $@

91 # Links from /etc/fs/zfs to /sbin
92 $(ROOTETCFSTYPELINKS):
93     -$(RM) $@; $(SYMLINK) ../../../../sbin/$(PROG) $@

95 FRC:

97 include ../Makefile.targ

```

new/usr/src/cmd/zfs/zfs_main.c

1

```
*****
160261 Thu Jun 28 15:57:07 2012
new/usr/src/cmd/zfs/zfs_main.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
25  * Copyright (c) 2012 by Delphix. All rights reserved.
26  * Copyright 2012 Milan Jurik. All rights reserved.
27  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
28  */
29
30 #include <assert.h>
31 #include <ctype.h>
32 #include <errno.h>
33 #include <libgen.h>
34 #include <libintl.h>
35 #include <libutil.h>
36 #include <libnvpair.h>
37 #include <locale.h>
38 #include <stddef.h>
39 #include <stdio.h>
40 #include <stdlib.h>
41 #include <strings.h>
42 #include <unistd.h>
43 #include <fcntl.h>
44 #include <zone.h>
45 #include <grp.h>
46 #include <pwd.h>
47 #include <signal.h>
48 #include <sys/list.h>
49 #include <sys/mkdev.h>
50 #include <sys/mntent.h>
51 #include <sys/mnttab.h>
52 #include <sys/mount.h>
53 #include <sys/stat.h>
54 #include <sys/fs/zfs.h>
```

new/usr/src/cmd/zfs/zfs_main.c

2

```
55 #include <sys/types.h>
56 #include <time.h>
57
58 #include <libzfs.h>
59 #include <libzfs_core.h>
60 #endif /* ! codereview */
61 #include <zfs_prop.h>
62 #include <zfs_deleg.h>
63 #include <libuutil.h>
64 #include <aclutils.h>
65 #include <directory.h>
66
67 #include "zfs_iter.h"
68 #include "zfs_util.h"
69 #include "zfs_comutil.h"
70
71 libzfs_handle_t *g_zfs;
72
73 static FILE *mnttab_file;
74 static char history_str[HIS_MAX_RECORD_LEN];
75 static boolean_t log_history = B_TRUE;
76 #endif /* ! codereview */
77
78 static int zfs_do_clone(int argc, char **argv);
79 static int zfs_do_create(int argc, char **argv);
80 static int zfs_do_destroy(int argc, char **argv);
81 static int zfs_do_get(int argc, char **argv);
82 static int zfs_do_inherit(int argc, char **argv);
83 static int zfs_do_list(int argc, char **argv);
84 static int zfs_do_mount(int argc, char **argv);
85 static int zfs_do_rename(int argc, char **argv);
86 static int zfs_do_rollback(int argc, char **argv);
87 static int zfs_do_set(int argc, char **argv);
88 static int zfs_do_upgrade(int argc, char **argv);
89 static int zfs_do_snapshot(int argc, char **argv);
90 static int zfs_do_unmount(int argc, char **argv);
91 static int zfs_do_share(int argc, char **argv);
92 static int zfs_do_unshare(int argc, char **argv);
93 static int zfs_do_send(int argc, char **argv);
94 static int zfs_do_receive(int argc, char **argv);
95 static int zfs_do_promote(int argc, char **argv);
96 static int zfs_do_userspace(int argc, char **argv);
97 static int zfs_do_allow(int argc, char **argv);
98 static int zfs_do_unallow(int argc, char **argv);
99 static int zfs_do_hold(int argc, char **argv);
100 static int zfs_do_holds(int argc, char **argv);
101 static int zfs_do_release(int argc, char **argv);
102 static int zfs_do_diff(int argc, char **argv);
103
104 /*
105  * Enable a reasonable set of defaults for libumem debugging on DEBUG builds.
106  */
107
108 #ifdef DEBUG
109 const char *
110 _umem_debug_init(void)
111 {
112     return ("default,verbose"); /* $UMEM_DEBUG setting */
113 }
114
115 const char *
116 _umem_logging_init(void)
117 {
118     return ("fail,contents"); /* $UMEM_LOGGING setting */
119 }
120 #endif
```

```

122 typedef enum {
123     HELP_CLONE,
124     HELP_CREATE,
125     HELP_DESTROY,
126     HELP_GET,
127     HELP_INHERIT,
128     HELP_UPGRADE,
129     HELP_LIST,
130     HELP_MOUNT,
131     HELP_PROMOTE,
132     HELP_RECEIVE,
133     HELP_RENAME,
134     HELP_ROLLBACK,
135     HELP_SEND,
136     HELP_SET,
137     HELP_SHARE,
138     HELP_SNAPSHOT,
139     HELP_UNMOUNT,
140     HELP_UNSHARE,
141     HELP_ALLOW,
142     HELP_UNALLOW,
143     HELP_USERSPACE,
144     HELP_GROUPSPACE,
145     HELP_HOLD,
146     HELP_HOLDS,
147     HELP_RELEASE,
148     HELP_DIFF,
149 } zfs_help_t;

151 typedef struct zfs_command {
152     const char    *name;
153     int           (*func)(int argc, char **argv);
154     zfs_help_t    usage;
155 } zfs_command_t;

157 /*
158  * Master command table.  Each ZFS command has a name, associated function, and
159  * usage message.  The usage messages need to be internationalized, so we have
160  * to have a function to return the usage message based on a command index.
161  *
162  * These commands are organized according to how they are displayed in the usage
163  * message.  An empty command (one with a NULL name) indicates an empty line in
164  * the generic usage message.
165  */
166 static zfs_command_t command_table[] = {
167     {"create",    zfs_do_create,    HELP_CREATE    },
168     {"destroy",  zfs_do_destroy,    HELP_DESTROY   },
169     {NULL},
170     {"snapshot", zfs_do_snapshot,   HELP_SNAPSHOT  },
171     {"rollback", zfs_do_rollback,   HELP_ROLLBACK  },
172     {"clone",    zfs_do_clone,      HELP_CLONE     },
173     {"promote",  zfs_do_promote,    HELP_PROMOTE   },
174     {"rename",   zfs_do_rename,     HELP_RENAME    },
175     {NULL},
176     {"list",     zfs_do_list,       HELP_LIST      },
177     {NULL},
178     {"set",      zfs_do_set,        HELP_SET       },
179     {"get",      zfs_do_get,        HELP_GET       },
180     {"inherit",  zfs_do_inherit,    HELP_INHERIT   },
181     {"upgrade",  zfs_do_upgrade,    HELP_UPGRADE   },
182     {"userspace", zfs_do_userspace,    HELP_USERSPACE },
183     {"groupspace", zfs_do_userspace,    HELP_GROUPSPACE },
184     {NULL},
185     {"mount",    zfs_do_mount,      HELP_MOUNT     },
186     {"unmount",  zfs_do_unmount,    HELP_UNMOUNT   },

```

```

187     {"share",    zfs_do_share,      HELP_SHARE     },
188     {"unshare",  zfs_do_unshare,    HELP_UNSHARE   },
189     {NULL},
190     {"send",     zfs_do_send,      HELP_SEND      },
191     {"receive",  zfs_do_receive,    HELP_RECEIVE   },
192     {NULL},
193     {"allow",    zfs_do_allow,     HELP_ALLOW     },
194     {NULL},
195     {"unallow",  zfs_do_unallow,    HELP_UNALLOW   },
196     {NULL},
197     {"hold",     zfs_do_hold,      HELP_HOLD      },
198     {"holds",    zfs_do_holds,     HELP_HOLDS     },
199     {"release",  zfs_do_release,    HELP_RELEASE   },
200     {"diff",     zfs_do_diff,      HELP_DIFF      },
201 };

203 #define NCOMMAND    (sizeof (command_table) / sizeof (command_table[0]))

205 zfs_command_t *current_command;

207 static const char *
208 get_usage(zfs_help_t idx)
209 {
210     switch (idx) {
211     case HELP_CLONE:
212         return (gettext("\tclone [-p] [-o property=value] ... "
213             "<snapshot> <filesystem|volume>\n"));
214     case HELP_CREATE:
215         return (gettext("\tcreate [-p] [-o property=value] ... "
216             "<filesystem>\n"
217             "\tcreate [-ps] [-b blocksize] [-o property=value] ... "
218             "-V <size> <volume>\n"));
219     case HELP_DESTROY:
220         return (gettext("\tdestroy [-fnpRrv] <filesystem|volume>\n"
221             "\tdestroy [-dnpRrv] "
222             "<filesystem|volume>@<snap>[%<snap>][,...]\n"));
223     case HELP_GET:
224         return (gettext("\tget [-rHp] [-d max] "
225             "[-o \"all\" | field,...] [-t type,...] "
226             "[-s source,...]\n"
227             "\t <\"all\" | property,...> "
228             "[filesystem|volume|snapshot] ... \n"));
229     case HELP_INHERIT:
230         return (gettext("\tinherit [-rS] <property> "
231             "<filesystem|volume|snapshot> ... \n"));
232     case HELP_UPGRADE:
233         return (gettext("\tupgrade [-v]\n"
234             "\tupgrade [-r] [-V version] <-a | filesystem ...>\n"));
235     case HELP_LIST:
236         return (gettext("\tlist [-rH][-d max] "
237             "[-o property,...] [-t type,...] [-s property] ... \n"
238             "\t [-S property] ... "
239             "[filesystem|volume|snapshot] ... \n"));
240     case HELP_MOUNT:
241         return (gettext("\tmount\n"
242             "\tmount [-vO] [-o opts] <-a | filesystem>\n"));
243     case HELP_PROMOTE:
244         return (gettext("\tpromote <clone-filesystem>\n"));
245     case HELP_RECEIVE:
246         return (gettext("\treceive [-vnFu] <filesystem|volume| "
247             "snapshot>\n"
248             "\treceive [-vnFu] [-d | -e] <filesystem>\n"));
249     case HELP_RENAME:
250         return (gettext("\trename [-f] <filesystem|volume|snapshot> "
251             "<filesystem|volume|snapshot>\n"
252             "\trename [-f] -p <filesystem|volume> <filesystem|volume>\n"));

```

```

253     "\trename -r <snapshot> <snapshot>"));
254 case HELP_ROLLBACK:
255     return (gettext("\trollback [-rRf] <snapshot>\n"));
256 case HELP_SEND:
257     return (gettext("\tsend [-DnPPrrv] [-iI] snapshot "
258     "<snapshot>\n"));
259 case HELP_SET:
260     return (gettext("\tset <property=value> "
261     "<filesystem|volume|snapshot> ... \n"));
262 case HELP_SHARE:
263     return (gettext("\tshare <-a | filesystem>\n"));
264 case HELP_SNAPSHOT:
265     return (gettext("\tsnapshot [-r] [-o property=value] ... "
266     "<filesystem@snapname|volume@snapname> ... \n"));
267     "\t<filesystem@snapname|volume@snapname>\n"));
268 case HELP_UNMOUNT:
269     return (gettext("\tunmount [-f] "
270     "<-a | filesystem|mountpoint>\n"));
271 case HELP_UNSHARE:
272     return (gettext("\tunshare "
273     "<-a | filesystem|mountpoint>\n"));
274 case HELP_ALLOW:
275     return (gettext("\tallow <filesystem|volume>\n"
276     "\tallow [-ldug] "
277     "<\\"everyone\"|user|group>[,...] <perm|@setname>[,...] \n"
278     "\t    <filesystem|volume>\n"
279     "\tallow [-ld] -e <perm|@setname>[,...] "
280     "<filesystem|volume>\n"
281     "\tallow -c <perm|@setname>[,...] <filesystem|volume>\n"
282     "\tallow -s @setname <perm|@setname>[,...] "
283     "<filesystem|volume>\n"));
284 case HELP_UNALLOW:
285     return (gettext("\tunallow [-rldug] "
286     "<\\"everyone\"|user|group>[,...] \n"
287     "\t    [<perm|@setname>[,...]] <filesystem|volume>\n"
288     "\tunallow [-rld] -e [<perm|@setname>[,...]] "
289     "<filesystem|volume>\n"
290     "\tunallow [-r] -c [<perm|@setname>[,...]] "
291     "<filesystem|volume>\n"
292     "\tunallow [-r] -s @setname [<perm|@setname>[,...]] "
293     "<filesystem|volume>\n"));
294 case HELP_USERSPACE:
295     return (gettext("\tuserspace [-hniHp] [-o field[,...]] "
296     "[-sS field] ... [-t type[,...]] \n"
297     "\t    <filesystem|snapshot>\n"));
298 case HELP_GROUPSPACE:
299     return (gettext("\tgroupspace [-hniHpU] [-o field[,...]] "
300     "[-sS field] ... [-t type[,...]] \n"
301     "\t    <filesystem|snapshot>\n"));
302 case HELP_HOLD:
303     return (gettext("\thold [-r] <tag> <snapshot> ... \n"));
304 case HELP_HOLDS:
305     return (gettext("\tholds [-r] <snapshot> ... \n"));
306 case HELP_RELEASE:
307     return (gettext("\trelease [-r] <tag> <snapshot> ... \n"));
308 case HELP_DIFF:
309     return (gettext("\tdiff [-FHT] <snapshot> "
310     "[snapshot|filesystem]\n"));
311 }
312 abort();
313 /* NOTREACHED */
314 }

```

unchanged_portion_omitted

868 /*

```

869 * zfs destroy [-rRf] <fs, vol>
870 * zfs destroy [-rRd] <snap>
871 *
872 *     -r       Recursively destroy all children
873 *     -R       Recursively destroy all dependents, including clones
874 *     -f       Force unmounting of any dependents
875 *     -d       If we can't destroy now, mark for deferred destruction
876 *
877 * Destroys the given dataset.  By default, it will unmount any filesystems,
878 * and refuse to destroy a dataset that has any dependents.  A dependent can
879 * either be a child, or a clone of a child.
880 */
881 typedef struct destroy_cbdata {
882     boolean_t    cb_first;
883     boolean_t    cb_force;
884     boolean_t    cb_recurse;
885     boolean_t    cb_error;
886     boolean_t    cb_doclones;
887     zfs_handle_t *cb_target;
888     boolean_t    cb_defer_destroy;
889     boolean_t    cb_verbose;
890     boolean_t    cb_parsable;
891     boolean_t    cb_dryrun;
892     nvlist_t     *cb_nvlist;
893
894     /* first snap in contiguous run */
895     char         *cb_firstsnap;
896     zfs_handle_t *cb_firstsnap;
897     /* previous snap in contiguous run */
898     char         *cb_prevsnap;
899     zfs_handle_t *cb_prevsnap;
900     int64_t      cb_snapused;
901     char         *cb_snapspec;
902 } destroy_cbdata_t;
903
904 /* unchanged_portion_omitted */
905
906 static int
907 destroy_print_cb(zfs_handle_t *zhp, void *arg)
908 {
909     destroy_cbdata_t *cb = arg;
910     const char *name = zfs_get_name(zhp);
911     int err = 0;
912
913     if (nvlist_exists(cb->cb_nvlist, name)) {
914         if (cb->cb_firstsnap == NULL)
915             cb->cb_firstsnap = strdup(name);
916         cb->cb_firstsnap = zfs_handle_dup(zhp);
917         if (cb->cb_prevsnap != NULL)
918             free(cb->cb_prevsnap);
919         zfs_close(cb->cb_prevsnap);
920         /* this snap continues the current range */
921         cb->cb_prevsnap = strdup(name);
922         if (cb->cb_firstsnap == NULL || cb->cb_prevsnap == NULL)
923             nomem();
924         cb->cb_prevsnap = zfs_handle_dup(zhp);
925         if (cb->cb_verbose) {
926             if (cb->cb_parsable) {
927                 (void) printf("destroy\t%s\n", name);
928             } else if (cb->cb_dryrun) {
929                 (void) printf(gettext("would destroy %s\n"),
930                     name);
931             } else {
932                 (void) printf(gettext("will destroy %s\n"),
933                     name);
934             }
935         }
936     }
937 }

```

```

1029     } else if (cb->cb_firstsnap != NULL) {
1030         /* end of this range */
1031         uint64_t used = 0;
1032         err = lzc_snaprange_space(cb->cb_firstsnap,
1033         err = zfs_get_snapused_int(cb->cb_firstsnap,
1034             cb->cb_prevsnap, &used);
1035         cb->cb_snapused += used;
1036         free(cb->cb_firstsnap);
1037         zfs_close(cb->cb_firstsnap);
1038         cb->cb_firstsnap = NULL;
1039         free(cb->cb_prevsnap);
1040         zfs_close(cb->cb_prevsnap);
1041         cb->cb_prevsnap = NULL;
1042     }
1043     zfs_close(zhp);
1044     return (err);
1045 }
1046
1047 static int
1048 destroy_print_snapshots(zfs_handle_t *fs_zhp, destroy_cbdata_t *cb)
1049 {
1050     int err = 0;
1051     assert(cb->cb_firstsnap == NULL);
1052     assert(cb->cb_prevsnap == NULL);
1053     err = zfs_iter_snapshots_sorted(fs_zhp, destroy_print_cb, cb);
1054     if (cb->cb_firstsnap != NULL) {
1055         uint64_t used = 0;
1056         if (err == 0) {
1057             err = lzc_snaprange_space(cb->cb_firstsnap,
1058             err = zfs_get_snapused_int(cb->cb_firstsnap,
1059                 cb->cb_prevsnap, &used);
1060             cb->cb_snapused += used;
1061             free(cb->cb_firstsnap);
1062             zfs_close(cb->cb_firstsnap);
1063             cb->cb_firstsnap = NULL;
1064             free(cb->cb_prevsnap);
1065             zfs_close(cb->cb_prevsnap);
1066             cb->cb_prevsnap = NULL;
1067         }
1068     }
1069     return (err);
1070 }
1071
1072 unchanged_portion_omitted

```

```

1878 static int
1879 upgrade_set_callback(zfs_handle_t *zhp, void *data)
1880 {
1881     upgrade_cbdata_t *cb = data;
1882     int version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1883     int needed_spa_version;
1884     int spa_version;
1885
1886     if (zfs_spa_version(zhp, &spa_version) < 0)
1887         return (-1);
1888
1889     needed_spa_version = zfs_spa_version_map(cb->cb_version);
1890
1891     if (needed_spa_version < 0)
1892         return (-1);
1893
1894     if (spa_version < needed_spa_version) {
1895         /* can't upgrade */
1896         (void) printf(gettext("%s: can not be "
1897             "upgraded; the pool version needs to first "
1898             "be upgraded\nto version %d\n"),
1899             zfs_get_name(zhp), needed_spa_version);

```

```

1900         cb->cb_numfailed++;
1901         return (0);
1902     }
1903
1904     /* upgrade */
1905     if (version < cb->cb_version) {
1906         char verstr[16];
1907         (void) snprintf(verstr, sizeof(verstr),
1908             "%llu", cb->cb_version);
1909         if (cb->cb_lastfs[0] && !same_pool(zhp, cb->cb_lastfs)) {
1910             /*
1911              * If they did "zfs upgrade -a", then we could
1912              * be doing ioctl's to different pools. We need
1913              * to log this history once to each pool, and bypass
1914              * the normal history logging that happens in main().
1915              * to log this history once to each pool.
1916              */
1917             (void) zpool_log_history(g_zfs, history_str);
1918             log_history = B_FALSE;
1919             verify(zpool_stage_history(g_zfs, history_str) == 0);
1920         }
1921         if (zfs_prop_set(zhp, "version", verstr) == 0)
1922             cb->cb_numupgraded++;
1923         else
1924             cb->cb_numfailed++;
1925         (void) strcpy(cb->cb_lastfs, zfs_get_name(zhp));
1926     } else if (version > cb->cb_version) {
1927         /* can't downgrade */
1928         (void) printf(gettext("%s: can not be downgraded; "
1929             "it is already at version %u\n"),
1930             zfs_get_name(zhp), version);
1931         cb->cb_numfailed++;
1932     } else {
1933         cb->cb_numsamegraded++;
1934     }
1935     return (0);
1936 }
1937
1938 unchanged_portion_omitted
1939
1940 typedef struct snap_cbdata {
1941     nvlist_t *sd_nvl;
1942     boolean_t sd_recursive;
1943     const char *sd_snapname;
1944 } snap_cbdata_t;
1945
1946 static int
1947 zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
1948 {
1949     snap_cbdata_t *sd = arg;
1950     char *name;
1951     int rv = 0;
1952     int error;
1953
1954     error = asprintf(&name, "%s%s", zfs_get_name(zhp), sd->sd_snapname);
1955     if (error == -1)
1956         nomem();
1957     fnvlist_add_boolean(sd->sd_nvl, name);
1958     free(name);
1959
1960     if (sd->sd_recursive)
1961         rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
1962     zfs_close(zhp);
1963     return (rv);
1964 }
1965
1966 #endif /* !codereview */

```



```

3462 /*
3463  * zfs snapshot [-r] [-o prop=value] ... <fs@snap>
3464  *
3465  * Creates a snapshot with the given name. While functionally equivalent to
3466  * 'zfs create', it is a separate command to differentiate intent.
3467  */
3468 static int
3469 zfs_do_snapshot(int argc, char **argv)
3470 {
3471     boolean_t recursive = B_FALSE;
3472     int ret = 0;
3473     char c;
3474     nvlist_t *props;
3475     snap_cbdata_t sd = { 0 };
3476     boolean_t multiple_snaps = B_FALSE;
3477 #endif /* ! codereview */
3478     if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
3479         nomem();
3480     if (nvlist_alloc(&sd.sd_nvlist, NV_UNIQUE_NAME, 0) != 0)
3481         nomem();
3482 #endif /* ! codereview */
3483
3484     /* check options */
3485     while ((c = getopt(argc, argv, "ro:")) != -1) {
3486         switch (c) {
3487             case 'o':
3488                 if (parseprop(props))
3489                     return (1);
3490                 break;
3491             case 'r':
3492                 sd.sd_recursive = B_TRUE;
3493                 multiple_snaps = B_TRUE;
3494                 recursive = B_TRUE;
3495                 break;
3496             case '?':
3497                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3498                     optopt);
3499                 goto usage;
3500         }
3501     }
3502
3503     argc -= optind;
3504     argv += optind;
3505
3506     /* check number of arguments */
3507     if (argc < 1) {
3508         (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3509         goto usage;
3510     }
3511
3512     if (argc > 1)
3513         multiple_snaps = B_TRUE;
3514     for (; argc > 0; argc--, argv++) {
3515         char *atp;
3516         zfs_handle_t *zhp;
3517
3518         atp = strchr(argv[0], '@');
3519         if (atp == NULL)
3520             goto usage;
3521         *atp = '\0';
3522         sd.sd_snapname = atp + 1;
3523         zhp = zfs_open(g_zfs, argv[0],
3524             ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3525         if (zhp == NULL)
3526             goto usage;

```

```

3527         if (zfs_snapshot_cb(zhp, &sd) != 0)
3528             if (argc > 1) {
3529                 (void) fprintf(stderr, gettext("too many arguments\n"));
3530                 goto usage;
3531             }
3532
3533     ret = zfs_snapshot_nvlist(g_zfs, sd.sd_nvlist, props);
3534     nvlist_free(sd.sd_nvlist);
3535     ret = zfs_snapshot(g_zfs, argv[0], recursive, props);
3536     nvlist_free(props);
3537     if (ret != 0 && multiple_snaps)
3538         if (ret && recursive)
3539             (void) fprintf(stderr, gettext("no snapshots were created\n"));
3540     return (ret != 0);
3541
3542 usage:
3543     nvlist_free(sd.sd_nvlist);
3544 #endif /* ! codereview */
3545     nvlist_free(props);
3546     usage(B_FALSE);
3547     return (-1);
3548 }
3549
3550 /*
3551  * Send a backup stream to stdout.
3552  */
3553 static int
3554 zfs_do_send(int argc, char **argv)
3555 {
3556     char *fromname = NULL;
3557     char *toname = NULL;
3558     char *cp;
3559     zfs_handle_t *zhp;
3560     sendflags_t flags = { 0 };
3561     int c, err;
3562     nvlist_t *dbgnav = NULL;
3563     boolean_t extraverbose = B_FALSE;
3564
3565     /* check options */
3566     while ((c = getopt(argc, argv, ":i:IRDpvnP")) != -1) {
3567         switch (c) {
3568             case 'i':
3569                 if (fromname)
3570                     usage(B_FALSE);
3571                 fromname = optarg;
3572                 break;
3573             case 'I':
3574                 if (fromname)
3575                     usage(B_FALSE);
3576                 fromname = optarg;
3577                 flags.doall = B_TRUE;
3578                 break;
3579             case 'R':
3580                 flags.replicate = B_TRUE;
3581                 break;
3582             case 'p':
3583                 flags.props = B_TRUE;
3584                 break;
3585             case 'P':
3586                 flags.parsable = B_TRUE;
3587                 flags.verbose = B_TRUE;
3588                 break;
3589             case 'v':
3590                 if (flags.verbose)
3591                     extraverbose = B_TRUE;
3592                 flags.verbose = B_TRUE;

```

```

3588         flags.progress = B_TRUE;
3589         break;
3590     case 'D':
3591         flags.dedup = B_TRUE;
3592         break;
3593     case 'n':
3594         flags.dryrun = B_TRUE;
3595         break;
3596     case ':':
3597         (void) fprintf(stderr, gettext("missing argument for "
3598         "%c' option\n"), optopt);
3599         usage(B_FALSE);
3600         break;
3601     case '?':
3602         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3603         optopt);
3604         usage(B_FALSE);
3605     }
3606 }
3608 argc -= optind;
3609 argv += optind;
3611 /* check number of arguments */
3612 if (argc < 1) {
3613     (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3614     usage(B_FALSE);
3615 }
3616 if (argc > 1) {
3617     (void) fprintf(stderr, gettext("too many arguments\n"));
3618     usage(B_FALSE);
3619 }
3621 if (!flags.dryrun && isatty(STDOUT_FILENO)) {
3622     (void) fprintf(stderr,
3623     gettext("Error: Stream can not be written to a terminal.\n"
3624     "You must redirect standard output.\n"));
3625     return (1);
3626 }
3628 cp = strchr(argv[0], '@');
3629 if (cp == NULL) {
3630     (void) fprintf(stderr,
3631     gettext("argument must be a snapshot\n"));
3632     usage(B_FALSE);
3633 }
3634 *cp = '\0';
3635 toname = cp + 1;
3636 zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3637 if (zhp == NULL)
3638     return (1);
3640 /*
3641  * If they specified the full path to the snapshot, chop off
3642  * everything except the short name of the snapshot, but special
3643  * case if they specify the origin.
3644  */
3645 if (fromname && (cp = strchr(fromname, '@')) != NULL) {
3646     char origin[ZFS_MAXNAMELEN];
3647     zprop_source_t src;
3649     (void) zfs_prop_get(zhp, ZFS_PROP_ORIGIN,
3650     origin, sizeof (origin), &src, NULL, 0, B_FALSE);
3652     if (strcmp(origin, fromname) == 0) {
3653         fromname = NULL;

```

```

3654         flags.fromorigin = B_TRUE;
3655     } else {
3656         *cp = '\0';
3657         if (cp != fromname && strcmp(argv[0], fromname)) {
3658             (void) fprintf(stderr,
3659             gettext("incremental source must be "
3660             "in same filesystem\n"));
3661             usage(B_FALSE);
3662         }
3663         fromname = cp + 1;
3664         if (strchr(fromname, '@') || strchr(fromname, '/')) {
3665             (void) fprintf(stderr,
3666             gettext("invalid incremental source\n"));
3667             usage(B_FALSE);
3668         }
3669     }
3670 }
3672 if (flags.replicate && fromname == NULL)
3673     flags.doall = B_TRUE;
3675 err = zfs_send(zhp, fromname, toname, &flags, STDOUT_FILENO, NULL, 0,
3676     extraverbose ? &dbgnv : NULL);
3678 if (extraverbose && dbgnv != NULL) {
3679     /*
3680     * dump_nvlist prints to stdout, but that's been
3681     * redirected to a file. Make it print to stderr
3682     * instead.
3683     */
3684     (void) dup2(STDERR_FILENO, STDOUT_FILENO);
3685     dump_nvlist(dbgnv, 0);
3686     nvlist_free(dbgnv);
3687 }
3688 zfs_close(zhp);
3690 return (err != 0);
3691 }
3693 /*
3694  * zfs receive [-vnFu] [-d | -e] <fs@snap>
3695  * Restore a backup stream from stdin.
3696  */
3697 static int
3698 zfs_do_receive(int argc, char **argv)
3699 {
3700     int c, err;
3701     recvflags_t flags = { 0 };
3702
3704     /* check options */
3705     while ((c = getopt(argc, argv, "denuvF")) != -1) {
3706         switch (c) {
3707             case 'd':
3708                 flags.isprefix = B_TRUE;
3709                 break;
3710             case 'e':
3711                 flags.isprefix = B_TRUE;
3712                 flags.istail = B_TRUE;
3713                 break;
3714             case 'n':
3715                 flags.dryrun = B_TRUE;
3716                 break;
3717             case 'u':
3718                 flags.nomount = B_TRUE;
3719                 break;

```

```

3720     case 'v':
3721         flags.verbose = B_TRUE;
3722         break;
3723     case 'F':
3724         flags.force = B_TRUE;
3725         break;
3726     case ':':
3727         (void) fprintf(stderr, gettext("missing argument for "
3728             "'%c' option\n"), optopt);
3729         usage(B_FALSE);
3730         break;
3731     case '?':
3732         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3733             optopt);
3734         usage(B_FALSE);
3735     }
3736 }
3737
3738 argc -= optind;
3739 argv += optind;
3740
3741 /* check number of arguments */
3742 if (argc < 1) {
3743     (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3744     usage(B_FALSE);
3745 }
3746 if (argc > 1) {
3747     (void) fprintf(stderr, gettext("too many arguments\n"));
3748     usage(B_FALSE);
3749 }
3750
3751 if (isatty(STDIN_FILENO)) {
3752     (void) fprintf(stderr,
3753         gettext("Error: Backup stream can not be read "
3754             "from a terminal.\n"
3755             "You must redirect standard input.\n"));
3756     return (1);
3757 }
3758
3759 err = zfs_receive(g_zfs, argv[0], &flags, STDIN_FILENO, NULL);
3760
3761 return (err != 0);
3762 }
3763
3764 /*
3765  * allow/unallow stuff
3766  */
3767 /* copied from zfs/sys/dsl_deleg.h */
3768 #define ZFS_DELEG_PERM_CREATE      "create"
3769 #define ZFS_DELEG_PERM_DESTROY    "destroy"
3770 #define ZFS_DELEG_PERM_SNAPSHOT   "snapshot"
3771 #define ZFS_DELEG_PERM_ROLLBACK   "rollback"
3772 #define ZFS_DELEG_PERM_CLONE      "clone"
3773 #define ZFS_DELEG_PERM_PROMOTE    "promote"
3774 #define ZFS_DELEG_PERM_RENAME     "rename"
3775 #define ZFS_DELEG_PERM_MOUNT     "mount"
3776 #define ZFS_DELEG_PERM_SHARE     "share"
3777 #define ZFS_DELEG_PERM_SEND       "send"
3778 #define ZFS_DELEG_PERM_RECEIVE    "receive"
3779 #define ZFS_DELEG_PERM_ALLOW      "allow"
3780 #define ZFS_DELEG_PERM_USERPROP   "userprop"
3781 #define ZFS_DELEG_PERM_VSCAN      "vscan" /* ??? */
3782 #define ZFS_DELEG_PERM_USERQUOTA  "userquota"
3783 #define ZFS_DELEG_PERM_GROUPQUOTA "groupquota"
3784 #define ZFS_DELEG_PERM_USERUSED   "userused"
3785 #define ZFS_DELEG_PERM_GROUPUSED  "groupused"

```

```

3786 #define ZFS_DELEG_PERM_HOLD        "hold"
3787 #define ZFS_DELEG_PERM_RELEASE    "release"
3788 #define ZFS_DELEG_PERM_DIFF       "diff"
3789
3790 #define ZFS_NUM_DELEG_NOTES ZFS_DELEG_NOTE_NONE
3791
3792 static zfs_deleg_perm_tab_t zfs_deleg_perm_tbl[] = {
3793     { ZFS_DELEG_PERM_ALLOW, ZFS_DELEG_NOTE_ALLOW },
3794     { ZFS_DELEG_PERM_CLONE, ZFS_DELEG_NOTE_CLONE },
3795     { ZFS_DELEG_PERM_CREATE, ZFS_DELEG_NOTE_CREATE },
3796     { ZFS_DELEG_PERM_DESTROY, ZFS_DELEG_NOTE_DESTROY },
3797     { ZFS_DELEG_PERM_DIFF, ZFS_DELEG_NOTE_DIFF },
3798     { ZFS_DELEG_PERM_HOLD, ZFS_DELEG_NOTE_HOLD },
3799     { ZFS_DELEG_PERM_MOUNT, ZFS_DELEG_NOTE_MOUNT },
3800     { ZFS_DELEG_PERM_PROMOTE, ZFS_DELEG_NOTE_PROMOTE },
3801     { ZFS_DELEG_PERM_RECEIVE, ZFS_DELEG_NOTE_RECEIVE },
3802     { ZFS_DELEG_PERM_RELEASE, ZFS_DELEG_NOTE_RELEASE },
3803     { ZFS_DELEG_PERM_RENAME, ZFS_DELEG_NOTE_RENAME },
3804     { ZFS_DELEG_PERM_ROLLBACK, ZFS_DELEG_NOTE_ROLLBACK },
3805     { ZFS_DELEG_PERM_SEND, ZFS_DELEG_NOTE_SEND },
3806     { ZFS_DELEG_PERM_SHARE, ZFS_DELEG_NOTE_SHARE },
3807     { ZFS_DELEG_PERM_SNAPSHOT, ZFS_DELEG_NOTE_SNAPSHOT },
3808
3809     { ZFS_DELEG_PERM_GROUPQUOTA, ZFS_DELEG_NOTE_GROUPQUOTA },
3810     { ZFS_DELEG_PERM_GROUPUSED, ZFS_DELEG_NOTE_GROUPUSED },
3811     { ZFS_DELEG_PERM_USERPROP, ZFS_DELEG_NOTE_USERPROP },
3812     { ZFS_DELEG_PERM_USERQUOTA, ZFS_DELEG_NOTE_USERQUOTA },
3813     { ZFS_DELEG_PERM_USERUSED, ZFS_DELEG_NOTE_USERUSED },
3814     { NULL, ZFS_DELEG_NOTE_NONE }
3815 };
3816
3817 /* permission structure */
3818 typedef struct deleg_perm {
3819     zfs_deleg_who_type_t    dp_who_type;
3820     const char              *dp_name;
3821     boolean_t               dp_local;
3822     boolean_t               dp_descend;
3823 } deleg_perm_t;
3824
3825 /* */
3826 typedef struct deleg_perm_node {
3827     deleg_perm_t            dpn_perm;
3828
3829     uu_avl_node_t          dpn_avl_node;
3830 } deleg_perm_node_t;
3831
3832 typedef struct fs_perm fs_perm_t;
3833
3834 /* permissions set */
3835 typedef struct who_perm {
3836     zfs_deleg_who_type_t    who_type;
3837     const char              *who_name; /* id */
3838     char                    who_ug_name[256]; /* user/group name */
3839     fs_perm_t               *who_fsperm; /* uplink */
3840
3841     uu_avl_t                *who_deleg_perm_avl; /* permissions */
3842 } who_perm_t;
3843
3844 /* */
3845 typedef struct who_perm_node {
3846     who_perm_t              who_perm;
3847     uu_avl_node_t          who_avl_node;
3848 } who_perm_node_t;
3849
3850 typedef struct fs_perm_set fs_perm_set_t;
3851 /* fs permissions */

```

```

3852 struct fs_perm {
3853     const char          *fsp_name;

3855     uu_avl_t            *fsp_sc_avl;    /* sets,create */
3856     uu_avl_t            *fsp_uge_avl;  /* user,group,everyone */

3858     fs_perm_set_t      *fsp_set;      /* uplink */
3859 };

3861 /* */
3862 typedef struct fs_perm_node {
3863     fs_perm_t           fspn_fsp;
3864     uu_avl_t           *fspn_avl;

3866     uu_list_node_t     fspn_list_node;
3867 } fs_perm_node_t;

3869 /* top level structure */
3870 struct fs_perm_set {
3871     uu_list_pool_t     *fspd_list_pool;
3872     uu_list_t          *fspd_list; /* list of fs_perms */

3874     uu_avl_pool_t     *fspd_named_set_avl_pool;
3875     uu_avl_pool_t     *fspd_who_perm_avl_pool;
3876     uu_avl_pool_t     *fspd_deleg_perm_avl_pool;
3877 };

3879 static inline const char *
3880 deleg_perm_type(zfs_deleg_note_t note)
3881 {
3882     /* subcommands */
3883     switch (note) {
3884         /* SUBCOMMANDS */
3885         /* OTHER */
3886         case ZFS_DELEG_NOTE_GROUPQUOTA:
3887         case ZFS_DELEG_NOTE_GROUPUSED:
3888         case ZFS_DELEG_NOTE_USERPROP:
3889         case ZFS_DELEG_NOTE_USERQUOTA:
3890         case ZFS_DELEG_NOTE_USERUSED:
3891             /* other */
3892             return (gettext("other"));
3893         default:
3894             return (gettext("subcommand"));
3895     }
3896 }

3898 static int inline
3899 who_type2weight(zfs_deleg_who_type_t who_type)
3900 {
3901     int res;
3902     switch (who_type) {
3903         case ZFS_DELEG_NAMED_SET_SETS:
3904         case ZFS_DELEG_NAMED_SET:
3905             res = 0;
3906             break;
3907         case ZFS_DELEG_CREATE_SETS:
3908         case ZFS_DELEG_CREATE:
3909             res = 1;
3910             break;
3911         case ZFS_DELEG_USER_SETS:
3912         case ZFS_DELEG_USER:
3913             res = 2;
3914             break;
3915         case ZFS_DELEG_GROUP_SETS:
3916         case ZFS_DELEG_GROUP:
3917             res = 3;

```

```

3918         break;
3919         case ZFS_DELEG_EVERYONE_SETS:
3920         case ZFS_DELEG_EVERYONE:
3921             res = 4;
3922             break;
3923         default:
3924             res = -1;
3925     }

3927     return (res);
3928 }

3930 /* ARGSUSED */
3931 static int
3932 who_perm_compare(const void *larg, const void *rarg, void *unused)
3933 {
3934     const who_perm_node_t *l = larg;
3935     const who_perm_node_t *r = rarg;
3936     zfs_deleg_who_type_t ltype = l->who_perm.who_type;
3937     zfs_deleg_who_type_t rtype = r->who_perm.who_type;
3938     int lweight = who_type2weight(ltype);
3939     int rweight = who_type2weight(rtype);
3940     int res = lweight - rweight;
3941     if (res == 0)
3942         res = strcmp(l->who_perm.who_name, r->who_perm.who_name,
3943             ZFS_MAX_DELEG_NAME-1);

3945     if (res == 0)
3946         return (0);
3947     if (res > 0)
3948         return (1);
3949     else
3950         return (-1);
3951 }

3953 /* ARGSUSED */
3954 static int
3955 deleg_perm_compare(const void *larg, const void *rarg, void *unused)
3956 {
3957     const deleg_perm_node_t *l = larg;
3958     const deleg_perm_node_t *r = rarg;
3959     int res = strcmp(l->dpn_perm.dp_name, r->dpn_perm.dp_name,
3960         ZFS_MAX_DELEG_NAME-1);

3962     if (res == 0)
3963         return (0);

3965     if (res > 0)
3966         return (1);
3967     else
3968         return (-1);
3969 }

3971 static inline void
3972 fs_perm_set_init(fs_perm_set_t *fspset)
3973 {
3974     bzero(fspset, sizeof (fs_perm_set_t));

3976     if ((fspset->fspd_list_pool = uu_list_pool_create("fspd_list_pool",
3977         sizeof (fs_perm_node_t), offsetof(fs_perm_node_t, fspn_list_node),
3978         NULL, UU_DEFAULT)) == NULL)
3979         nomem();
3980     if ((fspset->fspd_list = uu_list_create(fspset->fspd_list_pool, NULL,
3981         UU_DEFAULT)) == NULL)
3982         nomem();

```

```

3984     if ((fspset->fsps_named_set_avl_pool = uu_avl_pool_create(
3985         "named_set_avl_pool", sizeof (who_perm_node_t), offsetof(
3986         who_perm_node_t, who_avl_node), who_perm_compare,
3987         UU_DEFAULT)) == NULL)
3988         nomem();
3990     if ((fspset->fsps_who_perm_avl_pool = uu_avl_pool_create(
3991         "who_perm_avl_pool", sizeof (who_perm_node_t), offsetof(
3992         who_perm_node_t, who_avl_node), who_perm_compare,
3993         UU_DEFAULT)) == NULL)
3994         nomem();
3996     if ((fspset->fsps_deleg_perm_avl_pool = uu_avl_pool_create(
3997         "deleg_perm_avl_pool", sizeof (deleg_perm_node_t), offsetof(
3998         deleg_perm_node_t, dpm_avl_node), deleg_perm_compare, UU_DEFAULT))
3999         == NULL)
4000         nomem();
4001 }
4003 static inline void fs_perm_fini(fs_perm_t *);
4004 static inline void who_perm_fini(who_perm_t *);
4006 static inline void
4007 fs_perm_set_fini(fs_perm_set_t *fspset)
4008 {
4009     fs_perm_node_t *node = uu_list_first(fspset->fsps_list);
4011     while (node != NULL) {
4012         fs_perm_node_t *next_node =
4013             uu_list_next(fspset->fsps_list, node);
4014         fs_perm_t *fsperm = &node->fspn_fsperm;
4015         fs_perm_fini(fsperm);
4016         uu_list_remove(fspset->fsps_list, node);
4017         free(node);
4018         node = next_node;
4019     }
4021     uu_avl_pool_destroy(fspset->fsps_named_set_avl_pool);
4022     uu_avl_pool_destroy(fspset->fsps_who_perm_avl_pool);
4023     uu_avl_pool_destroy(fspset->fsps_deleg_perm_avl_pool);
4024 }
4026 static inline void
4027 deleg_perm_init(deleg_perm_t *deleg_perm, zfs_deleg_who_type_t type,
4028     const char *name)
4029 {
4030     deleg_perm->dp_who_type = type;
4031     deleg_perm->dp_name = name;
4032 }
4034 static inline void
4035 who_perm_init(who_perm_t *who_perm, fs_perm_t *fsperm,
4036     zfs_deleg_who_type_t type, const char *name)
4037 {
4038     uu_avl_pool_t *pool;
4039     pool = fsperm->fsp_set->fsps_deleg_perm_avl_pool;
4041     bzero(who_perm, sizeof (who_perm_t));
4043     if ((who_perm->who_deleg_perm_avl = uu_avl_create(pool, NULL,
4044         UU_DEFAULT)) == NULL)
4045         nomem();
4047     who_perm->who_type = type;
4048     who_perm->who_name = name;
4049     who_perm->who_fsperm = fsperm;

```

```

4050 }
4052 static inline void
4053 who_perm_fini(who_perm_t *who_perm)
4054 {
4055     deleg_perm_node_t *node = uu_avl_first(who_perm->who_deleg_perm_avl);
4057     while (node != NULL) {
4058         deleg_perm_node_t *next_node =
4059             uu_avl_next(who_perm->who_deleg_perm_avl, node);
4061         uu_avl_remove(who_perm->who_deleg_perm_avl, node);
4062         free(node);
4063         node = next_node;
4064     }
4066     uu_avl_destroy(who_perm->who_deleg_perm_avl);
4067 }
4069 static inline void
4070 fs_perm_init(fs_perm_t *fsperm, fs_perm_set_t *fspset, const char *fsname)
4071 {
4072     uu_avl_pool_t *nset_pool = fspset->fsps_named_set_avl_pool;
4073     uu_avl_pool_t *who_pool = fspset->fsps_who_perm_avl_pool;
4075     bzero(fsperm, sizeof (fs_perm_t));
4077     if ((fsperm->fsp_sc_avl = uu_avl_create(nset_pool, NULL, UU_DEFAULT))
4078         == NULL)
4079         nomem();
4081     if ((fsperm->fsp_uge_avl = uu_avl_create(who_pool, NULL, UU_DEFAULT))
4082         == NULL)
4083         nomem();
4085     fsperm->fsp_set = fspset;
4086     fsperm->fsp_name = fsname;
4087 }
4089 static inline void
4090 fs_perm_fini(fs_perm_t *fsperm)
4091 {
4092     who_perm_node_t *node = uu_avl_first(fsperm->fsp_sc_avl);
4093     while (node != NULL) {
4094         who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_sc_avl,
4095             node);
4096         who_perm_t *who_perm = &node->who_perm;
4097         who_perm_fini(who_perm);
4098         uu_avl_remove(fsperm->fsp_sc_avl, node);
4099         free(node);
4100         node = next_node;
4101     }
4103     node = uu_avl_first(fsperm->fsp_uge_avl);
4104     while (node != NULL) {
4105         who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_uge_avl,
4106             node);
4107         who_perm_t *who_perm = &node->who_perm;
4108         who_perm_fini(who_perm);
4109         uu_avl_remove(fsperm->fsp_uge_avl, node);
4110         free(node);
4111         node = next_node;
4112     }
4114     uu_avl_destroy(fsperm->fsp_sc_avl);
4115     uu_avl_destroy(fsperm->fsp_uge_avl);

```

```

4116 }
4118 static void inline
4119 set_deleg_perm_node(uu_avl_t *avl, deleg_perm_node_t *node,
4120     zfs_deleg_who_type_t who_type, const char *name, char locality)
4121 {
4122     uu_avl_index_t idx = 0;
4124     deleg_perm_node_t *found_node = NULL;
4125     deleg_perm_t *deleg_perm = &node->dpn_perm;
4127     deleg_perm_init(deleg_perm, who_type, name);
4129     if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4130         == NULL)
4131         uu_avl_insert(avl, node, idx);
4132     else {
4133         node = found_node;
4134         deleg_perm = &node->dpn_perm;
4135     }
4138     switch (locality) {
4139     case ZFS_DELEG_LOCAL:
4140         deleg_perm->dp_local = B_TRUE;
4141         break;
4142     case ZFS_DELEG_DESCENDENT:
4143         deleg_perm->dp_descend = B_TRUE;
4144         break;
4145     case ZFS_DELEG_NA:
4146         break;
4147     default:
4148         assert(B_FALSE); /* invalid locality */
4149     }
4150 }
4152 static inline int
4153 parse_who_perm(who_perm_t *who_perm, nvlist_t *nvl, char locality)
4154 {
4155     nvpair_t *nvp = NULL;
4156     fs_perm_set_t *fspset = who_perm->who_fspset->fsp_set;
4157     uu_avl_t *avl = who_perm->who_deleg_perm_avl;
4158     zfs_deleg_who_type_t who_type = who_perm->who_type;
4160     while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4161         const char *name = nvpair_name(nvp);
4162         data_type_t type = nvpair_type(nvp);
4163         uu_avl_pool_t *avl_pool = fspset->fsps_deleg_perm_avl_pool;
4164         deleg_perm_node_t *node =
4165             safe_malloc(sizeof (deleg_perm_node_t));
4167         assert(type == DATA_TYPE_BOOLEAN);
4169         uu_avl_node_init(node, &node->dpn_avl_node, avl_pool);
4170         set_deleg_perm_node(avl, node, who_type, name, locality);
4171     }
4173     return (0);
4174 }
4176 static inline int
4177 parse_fs_perm(fs_perm_t *fspset, nvlist_t *nvl)
4178 {
4179     nvpair_t *nvp = NULL;
4180     fs_perm_set_t *fspset = fspset->fsp_set;

```

```

4182     while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4183         nvlist_t *nvl2 = NULL;
4184         const char *name = nvpair_name(nvp);
4185         uu_avl_t *avl = NULL;
4186         uu_avl_pool_t *avl_pool;
4187         zfs_deleg_who_type_t perm_type = name[0];
4188         char perm_locality = name[1];
4189         const char *perm_name = name + 3;
4190         boolean_t is_set = B_TRUE;
4191         who_perm_t *who_perm = NULL;
4193         assert('$' == name[2]);
4195         if (nvpair_value_nvlist(nvp, &nvl2) != 0)
4196             return (-1);
4198         switch (perm_type) {
4199         case ZFS_DELEG_CREATE:
4200         case ZFS_DELEG_CREATE_SETS:
4201         case ZFS_DELEG_NAMED_SET:
4202         case ZFS_DELEG_NAMED_SET_SETS:
4203             avl_pool = fspset->fsps_named_set_avl_pool;
4204             avl = fspset->fsp_sc_avl;
4205             break;
4206         case ZFS_DELEG_USER:
4207         case ZFS_DELEG_USER_SETS:
4208         case ZFS_DELEG_GROUP:
4209         case ZFS_DELEG_GROUP_SETS:
4210         case ZFS_DELEG_EVERYONE:
4211         case ZFS_DELEG_EVERYONE_SETS:
4212             avl_pool = fspset->fsps_who_perm_avl_pool;
4213             avl = fspset->fsp_uge_avl;
4214             break;
4215         }
4217         if (is_set) {
4218             who_perm_node_t *found_node = NULL;
4219             who_perm_node_t *node = safe_malloc(
4220                 sizeof (who_perm_node_t));
4221             who_perm = &node->who_perm;
4222             uu_avl_index_t idx = 0;
4224             uu_avl_node_init(node, &node->who_avl_node, avl_pool);
4225             who_perm_init(who_perm, fspset, perm_type, perm_name);
4227             if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4228                 == NULL) {
4229                 if (avl == fspset->fsp_uge_avl) {
4230                     uid_t rid = 0;
4231                     struct passwd *p = NULL;
4232                     struct group *g = NULL;
4233                     const char *nice_name = NULL;
4235                     switch (perm_type) {
4236                     case ZFS_DELEG_USER_SETS:
4237                     case ZFS_DELEG_USER:
4238                         rid = atoi(perm_name);
4239                         p = getpwuid(rid);
4240                         if (p)
4241                             nice_name = p->pw_name;
4242                         break;
4243                     case ZFS_DELEG_GROUP_SETS:
4244                     case ZFS_DELEG_GROUP:
4245                         rid = atoi(perm_name);
4246                         g = getgrgid(rid);
4247                         if (g)

```



```

4380         break;
4381     case ZFS_DELEG_NOTE_USERPROP:
4382         str = gettext("Allows changing any user property");
4383         break;
4384     case ZFS_DELEG_NOTE_USERQUOTA:
4385         str = gettext("Allows accessing any userquota@... property");
4386         break;
4387     case ZFS_DELEG_NOTE_USERUSED:
4388         str = gettext("Allows reading any userused@... property");
4389         break;
4390         /* other */
4391     default:
4392         str = "";
4393     }
4395     return (str);
4396 }

4398 struct allow_opts {
4399     boolean_t local;
4400     boolean_t descend;
4401     boolean_t user;
4402     boolean_t group;
4403     boolean_t everyone;
4404     boolean_t create;
4405     boolean_t set;
4406     boolean_t recursive; /* unallow only */
4407     boolean_t prt_usage;

4409     boolean_t prt_perms;
4410     char *who;
4411     char *perms;
4412     const char *dataset;
4413 };

4415 static inline int
4416 prop_cmp(const void *a, const void *b)
4417 {
4418     const char *str1 = *(const char **)a;
4419     const char *str2 = *(const char **)b;
4420     return (strcmp(str1, str2));
4421 }

4423 static void
4424 allow_usage(boolean_t un, boolean_t requested, const char *msg)
4425 {
4426     const char *opt_desc[] = {
4427         "-h", gettext("show this help message and exit"),
4428         "-l", gettext("set permission locally"),
4429         "-d", gettext("set permission for descents"),
4430         "-u", gettext("set permission for user"),
4431         "-g", gettext("set permission for group"),
4432         "-e", gettext("set permission for everyone"),
4433         "-c", gettext("set create time permission"),
4434         "-s", gettext("define permission set"),
4435         /* unallow only */
4436         "-r", gettext("remove permissions recursively"),
4437     };
4438     size_t unallow_size = sizeof (opt_desc) / sizeof (char *);
4439     size_t allow_size = unallow_size - 2;
4440     const char *props[ZFS_NUM_PROPS];
4441     int i;
4442     size_t count = 0;
4443     FILE *fp = requested ? stdout : stderr;
4444     zprop_desc_t *pdtbl = zfs_prop_get_table();
4445     const char *fmt = gettext("%-16s %-14s\t%s\n");

```

```

4447     (void) fprintf(fp, gettext("Usage: %s\n"), get_usage(un ? HELP_UNALLOW :
4448     HELP_ALLOW));
4449     (void) fprintf(fp, gettext("Options:\n"));
4450     for (int i = 0; i < (un ? unallow_size : allow_size); i++) {
4451         const char *opt = opt_desc[i+1];
4452         const char *optdesc = opt_desc[i];
4453         (void) fprintf(fp, gettext(" %-10s %s\n"), opt, optdesc);
4454     }

4456     (void) fprintf(fp, gettext("\nThe following permissions are "
4457     "supported:\n\n"));
4458     (void) fprintf(fp, fmt, gettext("NAME"), gettext("TYPE"),
4459     gettext("NOTES"));
4460     for (i = 0; i < ZFS_NUM_DELEG_NOTES; i++) {
4461         const char *perm_name = zfs_deleg_perm_tbl[i].z_perm;
4462         zfs_deleg_note_t perm_note = zfs_deleg_perm_tbl[i].z_note;
4463         const char *perm_type = deleg_perm_type(perm_note);
4464         const char *perm_comment = deleg_perm_comment(perm_note);
4465         (void) fprintf(fp, fmt, perm_name, perm_type, perm_comment);
4466     }

4468     for (i = 0; i < ZFS_NUM_PROPS; i++) {
4469         zprop_desc_t *pd = &pdtbl[i];
4470         if (pd->pd_visible != B_TRUE)
4471             continue;

4473         if (pd->pd_attr == PROP_READONLY)
4474             continue;

4476         props[count++] = pd->pd_name;
4477     }
4478     props[count] = NULL;

4480     qsort(props, count, sizeof (char *), prop_cmp);

4482     for (i = 0; i < count; i++)
4483         (void) fprintf(fp, fmt, props[i], gettext("property"), "");

4485     if (msg != NULL)
4486         (void) fprintf(fp, gettext("\nzfs: error: %s"), msg);

4488     exit(requested ? 0 : 2);
4489 }

4491 static inline const char *
4492 munge_args(int argc, char **argv, boolean_t un, size_t expected_argc,
4493     char **permssp)
4494 {
4495     if (un && argc == expected_argc - 1)
4496         *permssp = NULL;
4497     else if (argc == expected_argc)
4498         *permssp = argv[argc - 2];
4499     else
4500         allow_usage(un, B_FALSE,
4501             gettext("wrong number of parameters\n"));

4503     return (argv[argc - 1]);
4504 }

4506 static void
4507 parse_allow_args(int argc, char **argv, boolean_t un, struct allow_opts *opts)
4508 {
4509     int uge_sum = opts->user + opts->group + opts->everyone;
4510     int csuge_sum = opts->create + opts->set + uge_sum;
4511     int ldcuge_sum = csuge_sum + opts->local + opts->descend;

```



```

4512     int all_sum = un ? ldcsuge_sum + opts->recursive : ldcsuge_sum;
4513
4514     if (uge_sum > 1)
4515         allow_usage(un, B_FALSE,
4516             gettext("-u, -g, and -e are mutually exclusive\n"));
4517
4518     if (opts->prt_usage)
4519         if (argc == 0 && all_sum == 0)
4520             allow_usage(un, B_TRUE, NULL);
4521         else
4522             usage(B_FALSE);
4523
4524     if (opts->set) {
4525         if (csuge_sum > 1)
4526             allow_usage(un, B_FALSE,
4527                 gettext("invalid options combined with -s\n"));
4528
4529         opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4530         if (argv[0][0] != '@')
4531             allow_usage(un, B_FALSE,
4532                 gettext("invalid set name: missing '@' prefix\n"));
4533         opts->who = argv[0];
4534     } else if (opts->create) {
4535         if (ldcsuge_sum > 1)
4536             allow_usage(un, B_FALSE,
4537                 gettext("invalid options combined with -c\n"));
4538         opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4539     } else if (opts->everyone) {
4540         if (csuge_sum > 1)
4541             allow_usage(un, B_FALSE,
4542                 gettext("invalid options combined with -e\n"));
4543         opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4544     } else if (uge_sum == 0 && argc > 0 && strcmp(argv[0], "everyone")
4545 == 0) {
4546         opts->everyone = B_TRUE;
4547         argc--;
4548         argv++;
4549         opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4550     } else if (argc == 1 && !un) {
4551         opts->prt_perms = B_TRUE;
4552         opts->dataset = argv[argc-1];
4553     } else {
4554         opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4555         opts->who = argv[0];
4556     }
4557
4558     if (!opts->local && !opts->descend) {
4559         opts->local = B_TRUE;
4560         opts->descend = B_TRUE;
4561     }
4562 }
4563
4564 static void
4565 store_allow_perm(zfs_deleg_who_type_t type, boolean_t local, boolean_t descend,
4566     const char *who, char *perms, nvlist_t *top_nvlist)
4567 {
4568     int i;
4569     char ld[2] = { '\0', '\0' };
4570     char who_buf[ZFS_MAXNAMELEN+32];
4571     char base_type;
4572     char set_type;
4573     nvlist_t *base_nvlist = NULL;
4574     nvlist_t *set_nvlist = NULL;
4575     nvlist_t *nvl;
4576
4577     if (nvlist_alloc(&base_nvlist, NV_UNIQUE_NAME, 0) != 0)

```

```

4578         nomem();
4579     if (nvlist_alloc(&set_nvlist, NV_UNIQUE_NAME, 0) != 0)
4580         nomem();
4581
4582     switch (type) {
4583     case ZFS_DELEG_NAMED_SET_SETS:
4584     case ZFS_DELEG_NAMED_SET:
4585         set_type = ZFS_DELEG_NAMED_SET_SETS;
4586         base_type = ZFS_DELEG_NAMED_SET;
4587         ld[0] = ZFS_DELEG_NA;
4588         break;
4589     case ZFS_DELEG_CREATE_SETS:
4590     case ZFS_DELEG_CREATE:
4591         set_type = ZFS_DELEG_CREATE_SETS;
4592         base_type = ZFS_DELEG_CREATE;
4593         ld[0] = ZFS_DELEG_NA;
4594         break;
4595     case ZFS_DELEG_USER_SETS:
4596     case ZFS_DELEG_USER:
4597         set_type = ZFS_DELEG_USER_SETS;
4598         base_type = ZFS_DELEG_USER;
4599         if (local)
4600             ld[0] = ZFS_DELEG_LOCAL;
4601         if (descend)
4602             ld[1] = ZFS_DELEG_DESCENDENT;
4603         break;
4604     case ZFS_DELEG_GROUP_SETS:
4605     case ZFS_DELEG_GROUP:
4606         set_type = ZFS_DELEG_GROUP_SETS;
4607         base_type = ZFS_DELEG_GROUP;
4608         if (local)
4609             ld[0] = ZFS_DELEG_LOCAL;
4610         if (descend)
4611             ld[1] = ZFS_DELEG_DESCENDENT;
4612         break;
4613     case ZFS_DELEG_EVERYONE_SETS:
4614     case ZFS_DELEG_EVERYONE:
4615         set_type = ZFS_DELEG_EVERYONE_SETS;
4616         base_type = ZFS_DELEG_EVERYONE;
4617         if (local)
4618             ld[0] = ZFS_DELEG_LOCAL;
4619         if (descend)
4620             ld[1] = ZFS_DELEG_DESCENDENT;
4621     }
4622
4623     if (perms != NULL) {
4624         char *curr = perms;
4625         char *end = curr + strlen(perms);
4626
4627         while (curr < end) {
4628             char *delim = strchr(curr, ',');
4629             if (delim == NULL)
4630                 delim = end;
4631             else
4632                 *delim = '\0';
4633
4634             if (curr[0] == '@')
4635                 nvl = set_nvlist;
4636             else
4637                 nvl = base_nvlist;
4638
4639             (void) nvlist_add_boolean(nvl, curr);
4640             if (delim != end)
4641                 *delim = ',';
4642             curr = delim + 1;
4643         }

```

```

4645     for (i = 0; i < 2; i++) {
4646         char locality = ld[i];
4647         if (locality == 0)
4648             continue;
4650         if (!nvlist_empty(base_nvl)) {
4651             if (who != NULL)
4652                 (void) snprintf(who_buf,
4653                     sizeof(who_buf), "%c%c%s",
4654                     base_type, locality, who);
4655             else
4656                 (void) snprintf(who_buf,
4657                     sizeof(who_buf), "%c%c%",
4658                     base_type, locality);
4660             (void) nvlist_add_nvlist(top_nvl, who_buf,
4661                 base_nvl);
4662         }
4665         if (!nvlist_empty(set_nvl)) {
4666             if (who != NULL)
4667                 (void) snprintf(who_buf,
4668                     sizeof(who_buf), "%c%c%s",
4669                     set_type, locality, who);
4670             else
4671                 (void) snprintf(who_buf,
4672                     sizeof(who_buf), "%c%c%",
4673                     set_type, locality);
4675             (void) nvlist_add_nvlist(top_nvl, who_buf,
4676                 set_nvl);
4677         }
4678     } else {
4679         for (i = 0; i < 2; i++) {
4680             char locality = ld[i];
4681             if (locality == 0)
4682                 continue;
4685             if (who != NULL)
4686                 (void) snprintf(who_buf, sizeof(who_buf),
4687                     "%c%c%s", base_type, locality, who);
4688             else
4689                 (void) snprintf(who_buf, sizeof(who_buf),
4690                     "%c%c%", base_type, locality);
4691             (void) nvlist_add_boolean(top_nvl, who_buf);
4693             if (who != NULL)
4694                 (void) snprintf(who_buf, sizeof(who_buf),
4695                     "%c%c%s", set_type, locality, who);
4696             else
4697                 (void) snprintf(who_buf, sizeof(who_buf),
4698                     "%c%c%", set_type, locality);
4699             (void) nvlist_add_boolean(top_nvl, who_buf);
4700         }
4701     }
4702 }
4704 static int
4705 construct_fsacl_list(boolean_t un, struct allow_opts *opts, nvlist_t **nvlp)
4706 {
4707     if (nvlist_alloc(nvlp, NV_UNIQUE_NAME, 0) != 0)
4708         nomem();

```

```

4710     if (opts->set) {
4711         store_allow_perm(ZFS_DELEG_NAMED_SET, opts->local,
4712             opts->descend, opts->who, opts->perms, *nvlp);
4713     } else if (opts->create) {
4714         store_allow_perm(ZFS_DELEG_CREATE, opts->local,
4715             opts->descend, NULL, opts->perms, *nvlp);
4716     } else if (opts->everyone) {
4717         store_allow_perm(ZFS_DELEG_EVERYONE, opts->local,
4718             opts->descend, NULL, opts->perms, *nvlp);
4719     } else {
4720         char *curr = opts->who;
4721         char *end = curr + strlen(curr);
4723         while (curr < end) {
4724             const char *who;
4725             zfs_deleg_who_type_t who_type;
4726             char *endch;
4727             char *delim = strchr(curr, ',');
4728             char errbuf[256];
4729             char id[64];
4730             struct passwd *p = NULL;
4731             struct group *g = NULL;
4733             uid_t rid;
4734             if (delim == NULL)
4735                 delim = end;
4736             else
4737                 *delim = '\\0';
4739             rid = (uid_t)strtol(curr, &endch, 0);
4740             if (opts->user) {
4741                 who_type = ZFS_DELEG_USER;
4742                 if (*endch != '\\0')
4743                     p = getpwnam(curr);
4744                 else
4745                     p = getpwuid(rid);
4747                 if (p != NULL)
4748                     rid = p->pw_uid;
4749                 else {
4750                     (void) snprintf(errbuf, 256, gettext(
4751                         "invalid user %s"), curr);
4752                     allow_usage(un, B_TRUE, errbuf);
4753                 }
4754             } else if (opts->group) {
4755                 who_type = ZFS_DELEG_GROUP;
4756                 if (*endch != '\\0')
4757                     g = getgrnam(curr);
4758                 else
4759                     g = getgrgid(rid);
4761                 if (g != NULL)
4762                     rid = g->gr_gid;
4763                 else {
4764                     (void) snprintf(errbuf, 256, gettext(
4765                         "invalid group %s"), curr);
4766                     allow_usage(un, B_TRUE, errbuf);
4767                 }
4768             } else {
4769                 if (*endch != '\\0') {
4770                     p = getpwnam(curr);
4771                 } else {
4772                     p = getpwuid(rid);
4773                 }
4775                 if (p == NULL)

```

```

4776         if (*endch != '\0') {
4777             g = getgrnam(curr);
4778         } else {
4779             g = getgrgid(rid);
4780         }
4782     if (p != NULL) {
4783         who_type = ZFS_DELEG_USER;
4784         rid = p->pw_uid;
4785     } else if (g != NULL) {
4786         who_type = ZFS_DELEG_GROUP;
4787         rid = g->gr_gid;
4788     } else {
4789         (void) snprintf(errbuf, 256, gettext(
4790             "invalid user/group %s"), curr);
4791         allow_usage(un, B_TRUE, errbuf);
4792     }
4793 }
4795 (void) sprintf(id, "%u", rid);
4796 who = id;
4798 store_allow_perm(who_type, opts->local,
4799                 opts->descend, who, opts->perms, *nvp);
4800 curr = delim + 1;
4801 }
4802 }
4804 return (0);
4805 }
4807 static void
4808 print_set_creat_perms(uu_avl_t *who_avl)
4809 {
4810     const char *sc_title[] = {
4811         gettext("Permission sets:\n"),
4812         gettext("Create time permissions:\n"),
4813         NULL
4814     };
4815     const char **title_ptr = sc_title;
4816     who_perm_node_t *who_node = NULL;
4817     int prev_weight = -1;
4819     for (who_node = uu_avl_first(who_avl); who_node != NULL;
4820          who_node = uu_avl_next(who_avl, who_node)) {
4821         uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4822         zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4823         const char *who_name = who_node->who_perm.who_name;
4824         int weight = who_type2weight(who_type);
4825         boolean_t first = B_TRUE;
4826         deleg_perm_node_t *deleg_node;
4828         if (prev_weight != weight) {
4829             (void) printf(*title_ptr++);
4830             prev_weight = weight;
4831         }
4833         if (who_name == NULL || strlen(who_name, 1) == 0)
4834             (void) printf("\t");
4835         else
4836             (void) printf("\t%s ", who_name);
4838         for (deleg_node = uu_avl_first(avl); deleg_node != NULL;
4839              deleg_node = uu_avl_next(avl, deleg_node)) {
4840             if (first) {
4841                 (void) printf("%s",

```

```

4842             deleg_node->dpn_perm.dp_name);
4843             first = B_FALSE;
4844         } else
4845             (void) printf("%s",
4846                 deleg_node->dpn_perm.dp_name);
4847         }
4849         (void) printf("\n");
4850     }
4851 }
4853 static void inline
4854 print_uge_deleg_perms(uu_avl_t *who_avl, boolean_t local, boolean_t descend,
4855                     const char *title)
4856 {
4857     who_perm_node_t *who_node = NULL;
4858     boolean_t prt_title = B_TRUE;
4859     uu_avl_walk_t *walk;
4861     if ((walk = uu_avl_walk_start(who_avl, UU_WALK_ROBUST)) == NULL)
4862         nomem();
4864     while ((who_node = uu_avl_walk_next(walk)) != NULL) {
4865         const char *who_name = who_node->who_perm.who_name;
4866         const char *nice_who_name = who_node->who_perm.who_ug_name;
4867         uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4868         zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4869         char delim = ' ';
4870         deleg_perm_node_t *deleg_node;
4871         boolean_t prt_who = B_TRUE;
4873         for (deleg_node = uu_avl_first(avl);
4874              deleg_node != NULL;
4875              deleg_node = uu_avl_next(avl, deleg_node)) {
4876             if (local != deleg_node->dpn_perm.dp_local ||
4877                 descend != deleg_node->dpn_perm.dp_descend)
4878                 continue;
4880             if (prt_who) {
4881                 const char *who = NULL;
4882                 if (prt_title) {
4883                     prt_title = B_FALSE;
4884                     (void) printf(title);
4885                 }
4887                 switch (who_type) {
4888                     case ZFS_DELEG_USER_SETS:
4889                         who = gettext("user");
4890                         if (nice_who_name)
4891                             who_name = nice_who_name;
4892                         break;
4893                     case ZFS_DELEG_GROUP_SETS:
4894                         who = gettext("group");
4895                         if (nice_who_name)
4896                             who_name = nice_who_name;
4897                         break;
4898                     case ZFS_DELEG_EVERYONE_SETS:
4899                         who = gettext("everyone");
4900                         who_name = NULL;
4901                     case ZFS_DELEG_EVERYONE:
4902                         who = gettext("everyone");
4903                         who_name = NULL;
4904                 }
4906                 prt_who = B_FALSE;
4907                 if (who_name == NULL)

```

```

4908         (void) printf("\t%s", who);
4909     else
4910         (void) printf("\t%s %s", who, who_name);
4911     }
4912
4913     (void) printf("%c%s", delim,
4914                 deleg_node->dpn_perm.dp_name);
4915     delim = ',';
4916 }
4917
4918     if (!prt_who)
4919         (void) printf("\n");
4920 }
4921
4922     uu_avl_walk_end(walk);
4923 }
4924
4925 static void
4926 print_fs_perms(fs_perm_set_t *fspset)
4927 {
4928     fs_perm_node_t *node = NULL;
4929     char buf[ZFS_MAXNAMELEN+32];
4930     const char *dsname = buf;
4931
4932     for (node = uu_list_first(fspset->fsps_list); node != NULL;
4933          node = uu_list_next(fspset->fsps_list, node)) {
4934         uu_avl_t *sc_avl = node->fspn_fspn.fsp_sc_avl;
4935         uu_avl_t *uge_avl = node->fspn_fspn.fsp_uge_avl;
4936         int left = 0;
4937
4938         (void) snprintf(buf, ZFS_MAXNAMELEN+32,
4939                       gettext("---- Permissions on %s"),
4940                           node->fspn_fspn.fsp_name);
4941         (void) printf(dsname);
4942         left = 70 - strlen(buf);
4943         while (left-- > 0)
4944             (void) printf("-");
4945         (void) printf("\n");
4946
4947         print_set_creat_perms(sc_avl);
4948         print_uge_deleg_perms(uge_avl, B_TRUE, B_FALSE,
4949                             gettext("Local permissions:\n"));
4950         print_uge_deleg_perms(uge_avl, B_FALSE, B_TRUE,
4951                             gettext("Descendent permissions:\n"));
4952         print_uge_deleg_perms(uge_avl, B_TRUE, B_TRUE,
4953                             gettext("Local+Descendent permissions:\n"));
4954     }
4955 }
4956
4957 static fs_perm_set_t fs_perm_set = { NULL, NULL, NULL, NULL };
4958
4959 struct deleg_perms {
4960     boolean_t un;
4961     nvlist_t *nvl;
4962 };
4963
4964 static int
4965 set_deleg_perms(zfs_handle_t *zhp, void *data)
4966 {
4967     struct deleg_perms *perms = (struct deleg_perms *)data;
4968     zfs_type_t zfs_type = zfs_get_type(zhp);
4969
4970     if (zfs_type != ZFS_TYPE_FILESYSTEM && zfs_type != ZFS_TYPE_VOLUME)
4971         return (0);
4972
4973     return (zfs_set_fsacl(zhp, perms->un, perms->nvl));

```

```

4974 }
4975
4976 static int
4977 zfs_do_allow_unallow_impl(int argc, char **argv, boolean_t un)
4978 {
4979     zfs_handle_t *zhp;
4980     nvlist_t *perm_nvl = NULL;
4981     nvlist_t *update_perm_nvl = NULL;
4982     int error = 1;
4983     int c;
4984     struct allow_opts opts = { 0 };
4985
4986     const char *optstr = un ? "ldugecsh" : "ldugecsh";
4987
4988     /* check opts */
4989     while ((c = getopt(argc, argv, optstr)) != -1) {
4990         switch (c) {
4991             case 'l':
4992                 opts.local = B_TRUE;
4993                 break;
4994             case 'd':
4995                 opts.descend = B_TRUE;
4996                 break;
4997             case 'u':
4998                 opts.user = B_TRUE;
4999                 break;
5000             case 'g':
5001                 opts.group = B_TRUE;
5002                 break;
5003             case 'e':
5004                 opts.everyone = B_TRUE;
5005                 break;
5006             case 's':
5007                 opts.set = B_TRUE;
5008                 break;
5009             case 'c':
5010                 opts.create = B_TRUE;
5011                 break;
5012             case 'r':
5013                 opts.recursive = B_TRUE;
5014                 break;
5015             case ':':
5016                 (void) fprintf(stderr, gettext("missing argument for "
5017                                                 "'%c' option\n"), optopt);
5018                 usage(B_FALSE);
5019                 break;
5020             case 'h':
5021                 opts.prt_usage = B_TRUE;
5022                 break;
5023             case '?':
5024                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5025                                 optopt);
5026                 usage(B_FALSE);
5027                 break;
5028         }
5029     }
5030
5031     argc -= optind;
5032     argv += optind;
5033
5034     /* check arguments */
5035     parse_allow_args(argc, argv, un, &opts);
5036
5037     /* try to open the dataset */
5038     if ((zhp = zfs_open(g_zfs, opts.dataset, ZFS_TYPE_FILESYSTEM |
5039                       ZFS_TYPE_VOLUME)) == NULL) {
5040         (void) fprintf(stderr, "Failed to open dataset: %s\n",

```

```

5040     opts.dataset);
5041     return (-1);
5042 }

5044 if (zfs_get_fsacl(zhp, &perm_nvlist) != 0)
5045     goto cleanup2;

5047 fs_perm_set_init(&fs_perm_set);
5048 if (parse_fs_perm_set(&fs_perm_set, perm_nvlist) != 0) {
5049     (void) fprintf(stderr, "Failed to parse fsacl permissions\n");
5050     goto cleanup1;
5051 }

5053 if (opts.prt_perms)
5054     print_fs_perms(&fs_perm_set);
5055 else {
5056     (void) construct_fs_acl_list(un, &opts, &update_perm_nvlist);
5057     if (zfs_set_fs_acl(zhp, un, update_perm_nvlist) != 0)
5058         goto cleanup0;

5060     if (un && opts.recursive) {
5061         struct deleg_perms data = { un, update_perm_nvlist };
5062         if (zfs_iter_filesystems(zhp, set_deleg_perms,
5063             &data) != 0)
5064             goto cleanup0;
5065     }
5066 }

5068 error = 0;

5070 cleanup0:
5071     nvlist_free(perm_nvlist);
5072     if (update_perm_nvlist != NULL)
5073         nvlist_free(update_perm_nvlist);
5074 cleanup1:
5075     fs_perm_set_fini(&fs_perm_set);
5076 cleanup2:
5077     zfs_close(zhp);

5079     return (error);
5080 }

5082 /*
5083  * zfs allow [-r] [-t] <tag> <snap> ...
5084  *
5085  *   -r   Recursively hold
5086  *   -t   Temporary hold (hidden option)
5087  *
5088  * Apply a user-hold with the given tag to the list of snapshots.
5089  */
5090 static int
5091 zfs_do_allow(int argc, char **argv)
5092 {
5093     return (zfs_do_allow_unallow_impl(argc, argv, B_FALSE));
5094 }

5096 /*
5097  * zfs unallow [-r] [-t] <tag> <snap> ...
5098  *
5099  *   -r   Recursively hold
5100  *   -t   Temporary hold (hidden option)
5101  *
5102  * Apply a user-hold with the given tag to the list of snapshots.
5103  */
5104 static int
5105 zfs_do_unallow(int argc, char **argv)

```

```

5106 {
5107     return (zfs_do_allow_unallow_impl(argc, argv, B_TRUE));
5108 }

5110 static int
5111 zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
5112 {
5113     int errors = 0;
5114     int i;
5115     const char *tag;
5116     boolean_t recursive = B_FALSE;
5117     boolean_t temphold = B_FALSE;
5118     const char *opts = holding ? "rt" : "r";
5119     int c;

5121     /* check options */
5122     while ((c = getopt(argc, argv, opts)) != -1) {
5123         switch (c) {
5124             case 'r':
5125                 recursive = B_TRUE;
5126                 break;
5127             case 't':
5128                 temphold = B_TRUE;
5129                 break;
5130             case '?':
5131                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5132                     optopt);
5133                 usage(B_FALSE);
5134         }
5135     }

5137     argc -= optind;
5138     argv += optind;

5140     /* check number of arguments */
5141     if (argc < 2)
5142         usage(B_FALSE);

5144     tag = argv[0];
5145     --argc;
5146     ++argv;

5148     if (holding && tag[0] == '.') {
5149         /* tags starting with '.' are reserved for libzfs */
5150         (void) fprintf(stderr, gettext("tag may not start with '.'\n"));
5151         usage(B_FALSE);
5152     }

5154     for (i = 0; i < argc; ++i) {
5155         zfs_handle_t *zhp;
5156         char parent[ZFS_MAXNAMELEN];
5157         const char *delim;
5158         char *path = argv[i];

5160         delim = strchr(path, '@');
5161         if (delim == NULL) {
5162             (void) fprintf(stderr,
5163                 gettext("'%' is not a snapshot\n"), path);
5164             ++errors;
5165             continue;
5166         }
5167         (void) strncpy(parent, path, delim - path);
5168         parent[delim - path] = '\0';

5170         zhp = zfs_open(g_zfs, parent,
5171             ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);

```

```

5172         if (zhp == NULL) {
5173             ++errors;
5174             continue;
5175         }
5176         if (holding) {
5177             if (zfs_hold(zhp, delim+1, tag, recursive,
5178                 temphold, B_FALSE, -1, 0, 0) != 0)
5179                 ++errors;
5180         } else {
5181             if (zfs_release(zhp, delim+1, tag, recursive) != 0)
5182                 ++errors;
5183         }
5184         zfs_close(zhp);
5185     }
5187     return (errors != 0);
5188 }
5190 /*
5191  * zfs hold [-r] [-t] <tag> <snap> ...
5192  *
5193  *   -r       Recursively hold
5194  *   -t       Temporary hold (hidden option)
5195  *
5196  * Apply a user-hold with the given tag to the list of snapshots.
5197  */
5198 static int
5199 zfs_do_hold(int argc, char **argv)
5200 {
5201     return (zfs_do_hold_rele_impl(argc, argv, B_TRUE));
5202 }
5204 /*
5205  * zfs release [-r] <tag> <snap> ...
5206  *
5207  *   -r       Recursively release
5208  *
5209  * Release a user-hold with the given tag from the list of snapshots.
5210  */
5211 static int
5212 zfs_do_release(int argc, char **argv)
5213 {
5214     return (zfs_do_hold_rele_impl(argc, argv, B_FALSE));
5215 }
5217 typedef struct holds_cbdata {
5218     boolean_t      cb_recursive;
5219     const char     *cb_snapname;
5220     nvlist_t       **cb_nvlp;
5221     size_t         cb_max_namelen;
5222     size_t         cb_max_taglen;
5223 } holds_cbdata_t;
5225 #define STRFTIME_FMT_STR "%a %b %e %k:%M %Y"
5226 #define DATETIME_BUF_LEN (32)
5227 /*
5228  *
5229  */
5230 static void
5231 print_holds(boolean_t scripted, size_t nwidth, size_t tagwidth, nvlist_t *nvl)
5232 {
5233     int i;
5234     nvpair_t *nvp = NULL;
5235     char *hdr_cols[] = { "NAME", "TAG", "TIMESTAMP" };
5236     const char *col;

```

```

5238         if (!scripted) {
5239             for (i = 0; i < 3; i++) {
5240                 col = gettext(hdr_cols[i]);
5241                 if (i < 2)
5242                     (void) printf("%-*s ", i ? tagwidth : nwidth,
5243                         col);
5244                 else
5245                     (void) printf("%s\n", col);
5246             }
5247         }
5249         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
5250             char *zname = nvpair_name(nvp);
5251             nvlist_t *nvl2;
5252             nvpair_t *nvp2 = NULL;
5253             (void) nvpair_value_nvlist(nvp, &nvl2);
5254             while ((nvp2 = nvlist_next_nvpair(nvl2, nvp2)) != NULL) {
5255                 char tsbuf[DATETIME_BUF_LEN];
5256                 char *tagname = nvpair_name(nvp2);
5257                 uint64_t val = 0;
5258                 time_t time;
5259                 struct tm t;
5260                 char sep = scripted ? '\t' : ' ';
5261                 size_t sepnum = scripted ? 1 : 2;
5263                 (void) nvpair_value_uint64(nvp2, &val);
5264                 time = (time_t)val;
5265                 (void) localtime_r(&time, &t);
5266                 (void) strftime(tsbuf, DATETIME_BUF_LEN,
5267                     gettext(STRFTIME_FMT_STR), &t);
5269                 (void) printf("%-*s%c%-*s%c%s\n", nwidth, zname,
5270                     sepnum, sep, tagwidth, tagname, sepnum, sep, tsbuf);
5271             }
5272         }
5273     }
5275 /*
5276  * Generic callback function to list a dataset or snapshot.
5277  */
5278 static int
5279 holds_callback(zfs_handle_t *zhp, void *data)
5280 {
5281     holds_cbdata_t *cbp = data;
5282     nvlist_t *top_nvl = *cbp->cb_nvlp;
5283     nvlist_t *nvl = NULL;
5284     nvpair_t *nvp = NULL;
5285     const char *zname = zfs_get_name(zhp);
5286     size_t znamelen = strlen(zname, ZFS_MAXNAMELEN);
5288     if (cbp->cb_recursive) {
5289         const char *snapname;
5290         char *delim = strchr(zname, '@');
5291         if (delim == NULL)
5292             return (0);
5294         snapname = delim + 1;
5295         if (strcmp(cbp->cb_snapname, snapname))
5296             return (0);
5297     }
5299     if (zfs_get_holds(zhp, &nvl) != 0)
5300         return (-1);
5302     if (znamelen > cbp->cb_max_namelen)
5303         cbp->cb_max_namelen = znamelen;

```

```

5305     while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
5306         const char *tag = nvpair_name(nvp);
5307         size_t taglen = strlen(tag, MAXNAMELEN);
5308         if (taglen > cbp->cb_max_taglen)
5309             cbp->cb_max_taglen = taglen;
5310     }
5312     return (nvlist_add_nvlist(top_nvl, zname, nvl));
5313 }
5315 /*
5316  * zfs holds [-r] <snap> ...
5317  *
5318  * -r      Recursively hold
5319  */
5320 static int
5321 zfs_do_holds(int argc, char **argv)
5322 {
5323     int errors = 0;
5324     int c;
5325     int i;
5326     boolean_t scripted = B_FALSE;
5327     boolean_t recursive = B_FALSE;
5328     const char *opts = "rH";
5329     nvlist_t *nvl;
5331     int types = ZFS_TYPE_SNAPSHOT;
5332     holds_cbdata_t cb = { 0 };
5334     int limit = 0;
5335     int ret = 0;
5336     int flags = 0;
5338     /* check options */
5339     while ((c = getopt(argc, argv, opts)) != -1) {
5340         switch (c) {
5341             case 'r':
5342                 recursive = B_TRUE;
5343                 break;
5344             case 'H':
5345                 scripted = B_TRUE;
5346                 break;
5347             case '?':
5348                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5349                     optopt);
5350                 usage(B_FALSE);
5351         }
5352     }
5354     if (recursive) {
5355         types |= ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME;
5356         flags |= ZFS_ITER_RECURSE;
5357     }
5359     argc -= optind;
5360     argv += optind;
5362     /* check number of arguments */
5363     if (argc < 1)
5364         usage(B_FALSE);
5366     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
5367         nomem();
5369     for (i = 0; i < argc; ++i) {

```

```

5370         char *snapshot = argv[i];
5371         const char *delim;
5372         const char *snapname;
5374         delim = strchr(snapshot, '@');
5375         if (delim == NULL) {
5376             (void) fprintf(stderr,
5377                 gettext("'%s' is not a snapshot\n"), snapshot);
5378             ++errors;
5379             continue;
5380         }
5381         snapname = delim + 1;
5382         if (recursive)
5383             snapshot[delim - snapshot] = '\0';
5385         cb.cb_recursive = recursive;
5386         cb.cb_snapname = snapname;
5387         cb.cb_nvlp = &nvl;
5389         /*
5390          * 1. collect holds data, set format options
5391          */
5392         ret = zfs_for_each(argc, argv, flags, types, NULL, NULL, limit,
5393             holds_callback, &cb);
5394         if (ret != 0)
5395             ++errors;
5396     }
5398     /*
5399      * 2. print holds data
5400      */
5401     print_holds(scripted, cb.cb_max_namelen, cb.cb_max_taglen, nvl);
5403     if (nvlist_empty(nvl))
5404         (void) printf(gettext("no datasets available\n"));
5406     nvlist_free(nvl);
5408     return (0 != errors);
5409 }
5411 #define CHECK_SPINNER 30
5412 #define SPINNER_TIME 3          /* seconds */
5413 #define MOUNT_TIME 5          /* seconds */
5415 static int
5416 get_one_dataset(zfs_handle_t *zhp, void *data)
5417 {
5418     static char *spin[] = { "-", "\\ ", "|", "/" };
5419     static int spinval = 0;
5420     static int spincheck = 0;
5421     static time_t last_spin_time = (time_t)0;
5422     get_all_cb_t *cbp = data;
5423     zfs_type_t type = zfs_get_type(zhp);
5425     if (cbp->cb_verbose) {
5426         if (--spincheck < 0) {
5427             time_t now = time(NULL);
5428             if (last_spin_time + SPINNER_TIME < now) {
5429                 update_progress(spin[spinval++ % 4]);
5430                 last_spin_time = now;
5431             }
5432             spincheck = CHECK_SPINNER;
5433         }
5434     }

```

```

5436     /*
5437     * Iterate over any nested datasets.
5438     */
5439     if (zfs_iter_filesystems(zhp, get_one_dataset, data) != 0) {
5440         zfs_close(zhp);
5441         return (1);
5442     }

5444     /*
5445     * Skip any datasets whose type does not match.
5446     */
5447     if ((type & ZFS_TYPE_FILESYSTEM) == 0) {
5448         zfs_close(zhp);
5449         return (0);
5450     }
5451     libzfs_add_handle(cbp, zhp);
5452     assert(cbp->cb_used <= cbp->cb_alloc);

5454     return (0);
5455 }

5457 static void
5458 get_all_datasets(zfs_handle_t ***dslist, size_t *count, boolean_t verbose)
5459 {
5460     get_all_cb_t cb = { 0 };
5461     cb.cb_verbose = verbose;
5462     cb.cb_getone = get_one_dataset;

5464     if (verbose)
5465         set_progress_header(gettext("Reading ZFS config"));
5466     (void) zfs_iter_root(g_zfs, get_one_dataset, &cb);

5468     *dslist = cb.cb_handles;
5469     *count = cb.cb_used;

5471     if (verbose)
5472         finish_progress(gettext("done.));
5473 }

5475 /*
5476 * Generic callback for sharing or mounting filesystems. Because the code is so
5477 * similar, we have a common function with an extra parameter to determine which
5478 * mode we are using.
5479 */
5480 #define OP_SHARE      0x1
5481 #define OP_MOUNT     0x2

5483 /*
5484 * Share or mount a dataset.
5485 */
5486 static int
5487 share_mount_one(zfs_handle_t *zhp, int op, int flags, char *protocol,
5488     boolean_t explicit, const char *options)
5489 {
5490     char mountpoint[ZFS_MAXPROPLEN];
5491     char shareopts[ZFS_MAXPROPLEN];
5492     char smbshareopts[ZFS_MAXPROPLEN];
5493     const char *cmdname = op == OP_SHARE ? "share" : "mount";
5494     struct mnttab mnt;
5495     uint64_t zoned, canmount;
5496     boolean_t shared_nfs, shared_smb;

5498     assert(zfs_get_type(zhp) & ZFS_TYPE_FILESYSTEM);

5500     /*
5501     * Check to make sure we can mount/share this dataset. If we

```

```

5502     * are in the global zone and the filesystem is exported to a
5503     * local zone, or if we are in a local zone and the
5504     * filesystem is not exported, then it is an error.
5505     */
5506     zoned = zfs_prop_get_int(zhp, ZFS_PROP_ZONED);

5508     if (zoned && getzoneid() == GLOBAL_ZONEID) {
5509         if (!explicit)
5510             return (0);

5512         (void) fprintf(stderr, gettext("cannot %s '%s': "
5513             "dataset is exported to a local zone\n"), cmdname,
5514             zfs_get_name(zhp));
5515         return (1);

5517     } else if (!zoned && getzoneid() != GLOBAL_ZONEID) {
5518         if (!explicit)
5519             return (0);

5521         (void) fprintf(stderr, gettext("cannot %s '%s': "
5522             "permission denied\n"), cmdname,
5523             zfs_get_name(zhp));
5524         return (1);
5525     }

5527     /*
5528     * Ignore any filesystems which don't apply to us. This
5529     * includes those with a legacy mountpoint, or those with
5530     * legacy share options.
5531     */
5532     verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
5533         sizeof(mountpoint), NULL, NULL, 0, B_FALSE) == 0);
5534     verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS, shareopts,
5535         sizeof(shareopts), NULL, NULL, 0, B_FALSE) == 0);
5536     verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB, smbshareopts,
5537         sizeof(smbshareopts), NULL, NULL, 0, B_FALSE) == 0);

5539     if (op == OP_SHARE && strcmp(shareopts, "off") == 0 &&
5540         strcmp(smbshareopts, "off") == 0) {
5541         if (!explicit)
5542             return (0);

5544         (void) fprintf(stderr, gettext("cannot share '%s': "
5545             "legacy share\n"), zfs_get_name(zhp));
5546         (void) fprintf(stderr, gettext("use share(1M) to "
5547             "share this filesystem, or set "
5548             "sharenfs property on\n"));
5549         return (1);
5550     }

5552     /*
5553     * We cannot share or mount legacy filesystems. If the
5554     * shareopts is non-legacy but the mountpoint is legacy, we
5555     * treat it as a legacy share.
5556     */
5557     if (strcmp(mountpoint, "legacy") == 0) {
5558         if (!explicit)
5559             return (0);

5561         (void) fprintf(stderr, gettext("cannot %s '%s': "
5562             "legacy mountpoint\n"), cmdname, zfs_get_name(zhp));
5563         (void) fprintf(stderr, gettext("use %s(1M) to "
5564             "%s this filesystem\n"), cmdname, cmdname);
5565         return (1);
5566     }

```



```

5568     if (strcmp(mountpoint, "none") == 0) {
5569         if (!explicit)
5570             return (0);
5571
5572         (void) fprintf(stderr, gettext("cannot %s '%s': no "
5573 "mountpoint set\n"), cmdname, zfs_get_name(zhp));
5574         return (1);
5575     }
5576
5577     /*
5578     * canmount      explicit      outcome
5579     * on            no            pass through
5580     * on            yes           pass through
5581     * off           no            return 0
5582     * off           yes           display error, return 1
5583     * noauto       no            return 0
5584     * noauto       yes           pass through
5585     */
5586     canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);
5587     if (canmount == ZFS_CANMOUNT_OFF) {
5588         if (!explicit)
5589             return (0);
5590
5591         (void) fprintf(stderr, gettext("cannot %s '%s': "
5592 "canmount' property is set to 'off'\n"), cmdname,
5593 zfs_get_name(zhp));
5594         return (1);
5595     } else if (canmount == ZFS_CANMOUNT_NOAUTO && !explicit) {
5596         return (0);
5597     }
5598
5599     /*
5600     * At this point, we have verified that the mountpoint and/or
5601     * shareopts are appropriate for auto management. If the
5602     * filesystem is already mounted or shared, return (failing
5603     * for explicit requests); otherwise mount or share the
5604     * filesystem.
5605     */
5606     switch (op) {
5607     case OP_SHARE:
5608
5609         shared_nfs = zfs_is_shared_nfs(zhp, NULL);
5610         shared_smb = zfs_is_shared_smb(zhp, NULL);
5611
5612         if (shared_nfs && shared_smb ||
5613             (shared_nfs && strcmp(shareopts, "on") == 0 &&
5614              strcmp(smbshareopts, "off") == 0) ||
5615             (shared_smb && strcmp(smbshareopts, "on") == 0 &&
5616              strcmp(shareopts, "off") == 0)) {
5617             if (!explicit)
5618                 return (0);
5619
5620             (void) fprintf(stderr, gettext("cannot share "
5621 "'%s': filesystem already shared\n"),
5622 zfs_get_name(zhp));
5623             return (1);
5624         }
5625
5626         if (!zfs_is_mounted(zhp, NULL) &&
5627             zfs_mount(zhp, NULL, 0) != 0)
5628             return (1);
5629
5630         if (protocol == NULL) {
5631             if (zfs_shareall(zhp) != 0)
5632                 return (1);
5633         } else if (strcmp(protocol, "nfs") == 0) {

```

```

5634         if (zfs_share_nfs(zhp))
5635             return (1);
5636     } else if (strcmp(protocol, "smb") == 0) {
5637         if (zfs_share_smb(zhp))
5638             return (1);
5639     } else {
5640         (void) fprintf(stderr, gettext("cannot share "
5641 "'%s': invalid share type '%s' "
5642 "specified\n"),
5643 zfs_get_name(zhp), protocol);
5644         return (1);
5645     }
5646
5647     break;
5648
5649     case OP_MOUNT:
5650         if (options == NULL)
5651             mnt.mnt_mntopts = "";
5652         else
5653             mnt.mnt_mntopts = (char *)options;
5654
5655         if (!hasmntopt(&mnt, MNTOPT_REMOUNT) &&
5656             zfs_is_mounted(zhp, NULL)) {
5657             if (!explicit)
5658                 return (0);
5659
5660             (void) fprintf(stderr, gettext("cannot mount "
5661 "'%s': filesystem already mounted\n"),
5662 zfs_get_name(zhp));
5663             return (1);
5664         }
5665
5666         if (zfs_mount(zhp, options, flags) != 0)
5667             return (1);
5668         break;
5669     }
5670
5671     return (0);
5672 }
5673
5674 /*
5675  * Reports progress in the form "(current/total)". Not thread-safe.
5676  */
5677 static void
5678 report_mount_progress(int current, int total)
5679 {
5680     static time_t last_progress_time = 0;
5681     time_t now = time(NULL);
5682     char info[32];
5683
5684     /* report 1..n instead of 0..n-1 */
5685     ++current;
5686
5687     /* display header if we're here for the first time */
5688     if (current == 1) {
5689         set_progress_header(gettext("Mounting ZFS filesystems"));
5690     } else if (current != total && last_progress_time + MOUNT_TIME >= now) {
5691         /* too soon to report again */
5692         return;
5693     }
5694
5695     last_progress_time = now;
5696
5697     (void) sprintf(info, "(%d/%d)", current, total);
5698
5699     if (current == total)

```

```

5700         finish_progress(info);
5701     else
5702         update_progress(info);
5703 }

5705 static void
5706 append_options(char *mntopts, char *newopts)
5707 {
5708     int len = strlen(mntopts);

5710     /* original length plus new string to append plus 1 for the comma */
5711     if (len + 1 + strlen(newopts) >= MNT_LINE_MAX) {
5712         (void) fprintf(stderr, gettext("the opts argument for "
5713             "'%c' option is too long (more than %d chars)\n"),
5714             "-o", MNT_LINE_MAX);
5715         usage(B_FALSE);
5716     }

5718     if (*mntopts)
5719         mntopts[len++] = ',';

5721     (void) strcpy(&mntopts[len], newopts);
5722 }

5724 static int
5725 share_mount(int op, int argc, char **argv)
5726 {
5727     int do_all = 0;
5728     boolean_t verbose = B_FALSE;
5729     int c, ret = 0;
5730     char *options = NULL;
5731     int flags = 0;

5733     /* check options */
5734     while ((c = getopt(argc, argv, op == OP_MOUNT ? ":avo:O" : "a"))
5735         != -1) {
5736         switch (c) {
5737             case 'a':
5738                 do_all = 1;
5739                 break;
5740             case 'v':
5741                 verbose = B_TRUE;
5742                 break;
5743             case 'o':
5744                 if (*optarg == '\0') {
5745                     (void) fprintf(stderr, gettext("empty mount "
5746                         "options (-o) specified\n"));
5747                     usage(B_FALSE);
5748                 }

5750                 if (options == NULL)
5751                     options = safe_malloc(MNT_LINE_MAX + 1);

5753                 /* option validation is done later */
5754                 append_options(options, optarg);
5755                 break;

5757             case 'O':
5758                 flags |= MS_OVERLAY;
5759                 break;
5760             case ':':
5761                 (void) fprintf(stderr, gettext("missing argument for "
5762                     "'%c' option\n"), optopt);
5763                 usage(B_FALSE);
5764                 break;
5765             case '?':

```

```

5766         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5767             optopt);
5768         usage(B_FALSE);
5769     }
5770 }

5772     argc -= optind;
5773     argv += optind;

5775     /* check number of arguments */
5776     if (do_all) {
5777         zfs_handle_t **dslist = NULL;
5778         size_t i, count = 0;
5779         char *protocol = NULL;

5781         if (op == OP_SHARE && argc > 0) {
5782             if (strcmp(argv[0], "nfs") != 0 &&
5783                 strcmp(argv[0], "smb") != 0) {
5784                 (void) fprintf(stderr, gettext("share type "
5785                     "must be 'nfs' or 'smb'\n"));
5786                 usage(B_FALSE);
5787             }
5788             protocol = argv[0];
5789             argc--;
5790             argv++;
5791         }

5793         if (argc != 0) {
5794             (void) fprintf(stderr, gettext("too many arguments\n"));
5795             usage(B_FALSE);
5796         }

5798         start_progress_timer();
5799         get_all_datasets(&dslist, &count, verbose);

5801         if (count == 0)
5802             return (0);

5804         qsort(dslist, count, sizeof (void *), libzfs_dataset_cmp);

5806         for (i = 0; i < count; i++) {
5807             if (verbose)
5808                 report_mount_progress(i, count);

5810             if (share_mount_one(dslist[i], op, flags, protocol,
5811                 B_FALSE, options) != 0)
5812                 ret = 1;
5813             zfs_close(dslist[i]);
5814         }

5816         free(dslist);
5817     } else if (argc == 0) {
5818         struct mnttab entry;

5820         if ((op == OP_SHARE) || (options != NULL)) {
5821             (void) fprintf(stderr, gettext("missing filesystem "
5822                 "argument (specify -a for all)\n"));
5823             usage(B_FALSE);
5824         }

5826         /*
5827          * When mount is given no arguments, go through /etc/mnttab and
5828          * display any active ZFS mounts. We hide any snapshots, since
5829          * they are controlled automatically.
5830          */
5831         rewind(mnttab_file);

```

```

5832     while (getmntent(mnttab_file, &entry) == 0) {
5833         if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0 ||
5834             strchr(entry.mnt_special, '@') != NULL)
5835             continue;
5837         (void) printf("%-30s %s\n", entry.mnt_special,
5838             entry.mnt_mountpt);
5839     }
5841 } else {
5842     zfs_handle_t *zhp;
5844     if (argc > 1) {
5845         (void) fprintf(stderr,
5846             gettext("too many arguments\n"));
5847         usage(B_FALSE);
5848     }
5850     if ((zhp = zfs_open(g_zfs, argv[0],
5851         ZFS_TYPE_FILESYSTEM)) == NULL) {
5852         ret = 1;
5853     } else {
5854         ret = share_mount_one(zhp, op, flags, NULL, B_TRUE,
5855             options);
5856         zfs_close(zhp);
5857     }
5858 }
5860 return (ret);
5861 }
5863 /*
5864  * zfs mount -a [nfs]
5865  * zfs mount filesystem
5866  *
5867  * Mount all filesystems, or mount the given filesystem.
5868  */
5869 static int
5870 zfs_do_mount(int argc, char **argv)
5871 {
5872     return (share_mount(OP_MOUNT, argc, argv));
5873 }
5875 /*
5876  * zfs share -a [nfs | smb]
5877  * zfs share filesystem
5878  *
5879  * Share all filesystems, or share the given filesystem.
5880  */
5881 static int
5882 zfs_do_share(int argc, char **argv)
5883 {
5884     return (share_mount(OP_SHARE, argc, argv));
5885 }
5887 typedef struct unshare_unmount_node {
5888     zfs_handle_t *un_zhp;
5889     char *un_mountpt;
5890     uu_avl_node_t un_avlnode;
5891 } unshare_unmount_node_t;
5893 /* ARGSUSED */
5894 static int
5895 unshare_unmount_compare(const void *larg, const void *rarg, void *unused)
5896 {
5897     const unshare_unmount_node_t *l1 = larg;

```

```

5898     const unshare_unmount_node_t *r = rarg;
5900     return (strcmp(l1->un_mountpt, r->un_mountpt));
5901 }
5903 /*
5904  * Convenience routine used by zfs_do_umount() and manual_unmount(). Given an
5905  * absolute path, find the entry /etc/mnttab, verify that its a ZFS filesystem,
5906  * and unmount it appropriately.
5907  */
5908 static int
5909 unshare_unmount_path(int op, char *path, int flags, boolean_t is_manual)
5910 {
5911     zfs_handle_t *zhp;
5912     int ret = 0;
5913     struct stat64 statbuf;
5914     struct extmnttab entry;
5915     const char *cmdname = (op == OP_SHARE) ? "unshare" : "umount";
5916     ino_t path_inode;
5918     /*
5919      * Search for the path in /etc/mnttab. Rather than looking for the
5920      * specific path, which can be fooled by non-standard paths (i.e. ".."
5921      * or "//"), we stat() the path and search for the corresponding
5922      * (major,minor) device pair.
5923      */
5924     if (stat64(path, &statbuf) != 0) {
5925         (void) fprintf(stderr, gettext("cannot %s '%s': %s\n"),
5926             cmdname, path, strerror(errno));
5927         return (1);
5928     }
5929     path_inode = statbuf.st_ino;
5931     /*
5932      * Search for the given (major,minor) pair in the mount table.
5933      */
5934     rewind(mnttab_file);
5935     while ((ret = getextmntent(mnttab_file, &entry, 0)) == 0) {
5936         if (entry.mnt_major == major(statbuf.st_dev) &&
5937             entry.mnt_minor == minor(statbuf.st_dev))
5938             break;
5939     }
5940     if (ret != 0) {
5941         if (op == OP_SHARE) {
5942             (void) fprintf(stderr, gettext("cannot %s '%s': not "
5943                 "currently mounted\n"), cmdname, path);
5944             return (1);
5945         }
5946         (void) fprintf(stderr, gettext("warning: %s not in mnttab\n"),
5947             path);
5948         if ((ret = umount2(path, flags)) != 0)
5949             (void) fprintf(stderr, gettext("%s: %s\n"), path,
5950                 strerror(errno));
5951         return (ret != 0);
5952     }
5954     if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0) {
5955         (void) fprintf(stderr, gettext("cannot %s '%s': not a ZFS "
5956             "filesystem\n"), cmdname, path);
5957         return (1);
5958     }
5960     if ((zhp = zfs_open(g_zfs, entry.mnt_special,
5961         ZFS_TYPE_FILESYSTEM)) == NULL)
5962         return (1);

```

```

5964     ret = 1;
5965     if (stat64(entry.mnt_mountp, &statbuf) != 0) {
5966         (void) fprintf(stderr, gettext("cannot %s '%s': %s\n"),
5967             cmdname, path, strerror(errno));
5968         goto out;
5969     } else if (statbuf.st_ino != path_inode) {
5970         (void) fprintf(stderr, gettext("cannot "
5971             "%s '%s': not a mountpoint\n"), cmdname, path);
5972         goto out;
5973     }

5975     if (op == OP_SHARE) {
5976         char nfs_mnt_prop[ZFS_MAXPROPLEN];
5977         char smbshare_prop[ZFS_MAXPROPLEN];

5979         verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS, nfs_mnt_prop,
5980             sizeof(nfs_mnt_prop), NULL, NULL, 0, B_FALSE) == 0);
5981         verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB, smbshare_prop,
5982             sizeof(smbshare_prop), NULL, NULL, 0, B_FALSE) == 0);

5984         if (strcmp(nfs_mnt_prop, "off") == 0 &&
5985             strcmp(smbshare_prop, "off") == 0) {
5986             (void) fprintf(stderr, gettext("cannot unshare "
5987                 "'%s': legacy share\n"), path);
5988             (void) fprintf(stderr, gettext("use "
5989                 "unshare(1M) to unshare this filesystem\n"));
5990         } else if (!zfs_is_shared(zhp)) {
5991             (void) fprintf(stderr, gettext("cannot unshare '%s': "
5992                 "not currently shared\n"), path);
5993         } else {
5994             ret = zfs_unshareall_bypath(zhp, path);
5995         }
5996     } else {
5997         char mtpt_prop[ZFS_MAXPROPLEN];

5999         verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mtpt_prop,
6000             sizeof(mtpt_prop), NULL, NULL, 0, B_FALSE) == 0);

6002         if (is_manual) {
6003             ret = zfs_unmount(zhp, NULL, flags);
6004         } else if (strcmp(mtpt_prop, "legacy") == 0) {
6005             (void) fprintf(stderr, gettext("cannot unmount "
6006                 "'%s': legacy mountpoint\n"),
6007                 zfs_get_name(zhp));
6008             (void) fprintf(stderr, gettext("use umount(1M) "
6009                 "to unmount this filesystem\n"));
6010         } else {
6011             ret = zfs_unmountall(zhp, flags);
6012         }
6013     }

6015 out:
6016     zfs_close(zhp);

6018     return (ret != 0);
6019 }

6021 /*
6022  * Generic callback for unsharing or unmounting a filesystem.
6023  */
6024 static int
6025 unshare_unmount(int op, int argc, char **argv)
6026 {
6027     int do_all = 0;
6028     int flags = 0;
6029     int ret = 0;

```

```

6030     int c;
6031     zfs_handle_t *zhp;
6032     char nfs_mnt_prop[ZFS_MAXPROPLEN];
6033     char sharesmb[ZFS_MAXPROPLEN];

6035     /* check options */
6036     while ((c = getopt(argc, argv, op == OP_SHARE ? "a" : "af")) != -1) {
6037         switch (c) {
6038             case 'a':
6039                 do_all = 1;
6040                 break;
6041             case 'f':
6042                 flags = MS_FORCE;
6043                 break;
6044             case '?':
6045                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6046                     optopt);
6047                 usage(B_FALSE);
6048             }
6049     }

6051     argc -= optind;
6052     argv += optind;

6054     if (do_all) {
6055         /*
6056          * We could make use of zfs_for_each() to walk all datasets in
6057          * the system, but this would be very inefficient, especially
6058          * since we would have to linearly search /etc/mnttab for each
6059          * one. Instead, do one pass through /etc/mnttab looking for
6060          * zfs entries and call zfs_unmount() for each one.
6061          *
6062          * Things get a little tricky if the administrator has created
6063          * mountpoints beneath other ZFS filesystems. In this case, we
6064          * have to unmount the deepest filesystems first. To accomplish
6065          * this, we place all the mountpoints in an AVL tree sorted by
6066          * the special type (dataset name), and walk the result in
6067          * reverse to make sure to get any snapshots first.
6068          */
6069         struct mnttab entry;
6070         uu_avl_pool_t *pool;
6071         uu_avl_t *tree;
6072         unshare_unmount_node_t *node;
6073         uu_avl_index_t idx;
6074         uu_avl_walk_t *walk;

6076         if (argc != 0) {
6077             (void) fprintf(stderr, gettext("too many arguments\n"));
6078             usage(B_FALSE);
6079         }

6081         if ((pool = uu_avl_pool_create("unmount_pool",
6082             sizeof(unshare_unmount_node_t),
6083             offsetof(unshare_unmount_node_t, un_avlnode),
6084             unshare_unmount_compare, UU_DEFAULT)) == NULL) ||
6085             ((tree = uu_avl_create(pool, NULL, UU_DEFAULT)) == NULL))
6086             nomem();

6088         rewind(mnttab_file);
6089         while (getmntent(mnttab_file, &entry) == 0) {

6091             /* ignore non-ZFS entries */
6092             if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
6093                 continue;

6095             /* ignore snapshots */

```

```

6096         if (strchr(entry.mnt_special, '@') != NULL)
6097             continue;
6099         if ((zhp = zfs_open(g_zfs, entry.mnt_special,
6100             ZFS_TYPE_FILESYSTEM)) == NULL) {
6101             ret = 1;
6102             continue;
6103         }
6105         switch (op) {
6106         case OP_SHARE:
6107             verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS,
6108                 nfs_mnt_prop,
6109                 sizeof (nfs_mnt_prop),
6110                 NULL, NULL, 0, B_FALSE) == 0);
6111             if (strcmp(nfs_mnt_prop, "off") != 0)
6112                 break;
6113             verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB,
6114                 nfs_mnt_prop,
6115                 sizeof (nfs_mnt_prop),
6116                 NULL, NULL, 0, B_FALSE) == 0);
6117             if (strcmp(nfs_mnt_prop, "off") == 0)
6118                 continue;
6119             break;
6120         case OP_MOUNT:
6121             /* Ignore legacy mounts */
6122             verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT,
6123                 nfs_mnt_prop,
6124                 sizeof (nfs_mnt_prop),
6125                 NULL, NULL, 0, B_FALSE) == 0);
6126             if (strcmp(nfs_mnt_prop, "legacy") == 0)
6127                 continue;
6128             /* Ignore canmount=noauto mounts */
6129             if (zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT) ==
6130                 ZFS_CANMOUNT_NOAUTO)
6131                 continue;
6132         default:
6133             break;
6134         }
6136         node = safe_malloc(sizeof (unshare_unmount_node_t));
6137         node->un_zhp = zhp;
6138         node->un_mountp = safe_strdup(entry.mnt_mountp);
6140         uu_avl_node_init(node, &node->un_avlnode, pool);
6142         if (uu_avl_find(tree, node, NULL, &idx) == NULL) {
6143             uu_avl_insert(tree, node, idx);
6144         } else {
6145             zfs_close(node->un_zhp);
6146             free(node->un_mountp);
6147             free(node);
6148         }
6149     }
6151     /*
6152     * Walk the AVL tree in reverse, unmounting each filesystem and
6153     * removing it from the AVL tree in the process.
6154     */
6155     if ((walk = uu_avl_walk_start(tree,
6156         UU_WALK_REVERSE | UU_WALK_ROBUST)) == NULL)
6157         nomem();
6159     while ((node = uu_avl_walk_next(walk)) != NULL) {
6160         uu_avl_remove(tree, node);

```

```

6162         switch (op) {
6163         case OP_SHARE:
6164             if (zfs_unshareall_bypath(node->un_zhp,
6165                 node->un_mountp) != 0)
6166                 ret = 1;
6167             break;
6169         case OP_MOUNT:
6170             if (zfs_unmount(node->un_zhp,
6171                 node->un_mountp, flags) != 0)
6172                 ret = 1;
6173             break;
6174         }
6176         zfs_close(node->un_zhp);
6177         free(node->un_mountp);
6178         free(node);
6179     }
6181     uu_avl_walk_end(walk);
6182     uu_avl_destroy(tree);
6183     uu_avl_pool_destroy(pool);
6185     } else {
6186         if (argc != 1) {
6187             if (argc == 0)
6188                 (void) fprintf(stderr,
6189                     gettext("missing filesystem argument\n"));
6189             else
6190                 (void) fprintf(stderr,
6191                     gettext("too many arguments\n"));
6192             usage(B_FALSE);
6193         }
6194     /*
6195     * We have an argument, but it may be a full path or a ZFS
6196     * filesystem. Pass full paths off to unmount_path() (shared by
6197     * manual_unmount), otherwise open the filesystem and pass to
6198     * zfs_unmount().
6199     */
6200     if (argv[0][0] == '/')
6201         return (unshare_unmount_path(op, argv[0],
6202             flags, B_FALSE));
6206     if ((zhp = zfs_open(g_zfs, argv[0],
6207         ZFS_TYPE_FILESYSTEM)) == NULL)
6208         return (1);
6210     verify(zfs_prop_get(zhp, op == OP_SHARE ?
6211         ZFS_PROP_SHARENFS : ZFS_PROP_MOUNTPOINT,
6212         nfs_mnt_prop, sizeof (nfs_mnt_prop), NULL,
6213         NULL, 0, B_FALSE) == 0);
6215     switch (op) {
6216     case OP_SHARE:
6217         verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS,
6218             nfs_mnt_prop,
6219             sizeof (nfs_mnt_prop),
6220             NULL, NULL, 0, B_FALSE) == 0);
6221         verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB,
6222             sharesmb, sizeof (sharesmb), NULL, NULL,
6223             0, B_FALSE) == 0);
6225         if (strcmp(nfs_mnt_prop, "off") == 0 &&
6226             strcmp(sharesmb, "off") == 0) {
6227             (void) fprintf(stderr, gettext("cannot "

```

```

6228         "unshare '%s': legacy share\n"),
6229         zfs_get_name(zhp));
6230         (void) fprintf(stderr, gettext("use "
6231         "unshare(1M) to unshare this "
6232         "filesystem\n"));
6233         ret = 1;
6234     } else if (!zfs_is_shared(zhp)) {
6235         (void) fprintf(stderr, gettext("cannot "
6236         "unshare '%s': not currently "
6237         "shared\n"), zfs_get_name(zhp));
6238         ret = 1;
6239     } else if (zfs_unshareall(zhp) != 0) {
6240         ret = 1;
6241     }
6242     break;

6244     case OP_MOUNT:
6245         if (strcmp(nfs_mnt_prop, "legacy") == 0) {
6246             (void) fprintf(stderr, gettext("cannot "
6247             "unmount '%s': legacy "
6248             "mountpoint\n"), zfs_get_name(zhp));
6249             (void) fprintf(stderr, gettext("use "
6250             "umount(1M) to unmount this "
6251             "filesystem\n"));
6252             ret = 1;
6253         } else if (!zfs_is_mounted(zhp, NULL)) {
6254             (void) fprintf(stderr, gettext("cannot "
6255             "unmount '%s': not currently "
6256             "mounted\n"),
6257             zfs_get_name(zhp));
6258             ret = 1;
6259         } else if (zfs_unmountall(zhp, flags) != 0) {
6260             ret = 1;
6261         }
6262         break;
6263     }

6265     zfs_close(zhp);
6266 }

6268     return (ret);
6269 }

6271 /*
6272  * zfs unmount -a
6273  * zfs unmount filesystem
6274  *
6275  * Unmount all filesystems, or a specific ZFS filesystem.
6276  */
6277 static int
6278 zfs_do_unmount(int argc, char **argv)
6279 {
6280     return (unshare_unmount(OP_MOUNT, argc, argv));
6281 }

6283 /*
6284  * zfs unshare -a
6285  * zfs unshare filesystem
6286  *
6287  * Unshare all filesystems, or a specific ZFS filesystem.
6288  */
6289 static int
6290 zfs_do_unshare(int argc, char **argv)
6291 {
6292     return (unshare_unmount(OP_SHARE, argc, argv));
6293 }

```

```

6295 /*
6296  * Called when invoked as /etc/fs/zfs/mount. Do the mount if the mountpoint is
6297  * 'legacy'. Otherwise, complain that use should be using 'zfs mount'.
6298  */
6299 static int
6300 manual_mount(int argc, char **argv)
6301 {
6302     zfs_handle_t *zhp;
6303     char mountpoint[ZFS_MAXPROPLEN];
6304     char mntopts[MNT_LINE_MAX] = { '\0' };
6305     int ret = 0;
6306     int c;
6307     int flags = 0;
6308     char *dataset, *path;

6310     /* check options */
6311     while ((c = getopt(argc, argv, "mo:O")) != -1) {
6312         switch (c) {
6313             case 'o':
6314                 (void) strlcpy(mntopts, optarg, sizeof (mntopts));
6315                 break;
6316             case 'O':
6317                 flags |= MS_OVERLAY;
6318                 break;
6319             case 'm':
6320                 flags |= MS_NOMNTTAB;
6321                 break;
6322             case ':':
6323                 (void) fprintf(stderr, gettext("missing argument for "
6324                 "'%c' option\n"), optarg);
6325                 usage(B_FALSE);
6326                 break;
6327             case '?':
6328                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6329                 optarg);
6330                 (void) fprintf(stderr, gettext("usage: mount [-o opts] "
6331                 "<path>\n"));
6332                 return (2);
6333             }
6334     }

6336     argc -= optind;
6337     argv += optind;

6339     /* check that we only have two arguments */
6340     if (argc != 2) {
6341         if (argc == 0)
6342             (void) fprintf(stderr, gettext("missing dataset "
6343             "argument\n"));
6344         else if (argc == 1)
6345             (void) fprintf(stderr,
6346             gettext("missing mountpoint argument\n"));
6347         else
6348             (void) fprintf(stderr, gettext("too many arguments\n"));
6349         (void) fprintf(stderr, "usage: mount <dataset> <mountpoint>\n");
6350         return (2);
6351     }

6353     dataset = argv[0];
6354     path = argv[1];

6356     /* try to open the dataset */
6357     if ((zhp = zfs_open(g_zfs, dataset, ZFS_TYPE_FILESYSTEM)) == NULL)
6358         return (1);

```

```

6360     (void) zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
6361         sizeof (mountpoint), NULL, NULL, 0, B_FALSE);

6363     /* check for legacy mountpoint and complain appropriately */
6364     ret = 0;
6365     if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
6366         if (mount(dataset, path, MS_OPTIONSTR | flags, MNTTYPE_ZFS,
6367             NULL, 0, mntopts, sizeof (mntopts)) != 0) {
6368             (void) fprintf(stderr, gettext("mount failed: %s\n"),
6369                 strerror(errno));
6370             ret = 1;
6371         }
6372     } else {
6373         (void) fprintf(stderr, gettext("filesystem '%s' cannot be "
6374             "mounted using 'mount -F zfs'\n"), dataset);
6375         (void) fprintf(stderr, gettext("Use 'zfs set mountpoint=%s' "
6376             "instead.\n"), path);
6377         (void) fprintf(stderr, gettext("If you must use 'mount -F zfs' "
6378             "or /etc/vfstab, use 'zfs set mountpoint=legacy'\n"));
6379         (void) fprintf(stderr, gettext("See zfs(1M) for more "
6380             "information.\n"));
6381         ret = 1;
6382     }

6384     return (ret);
6385 }

6387 /*
6388  * Called when invoked as /etc/fs/zfs/umount. Unlike a manual mount, we allow
6389  * unmounts of non-legacy filesystems, as this is the dominant administrative
6390  * interface.
6391  */
6392 static int
6393 manual_unmount(int argc, char **argv)
6394 {
6395     int flags = 0;
6396     int c;

6398     /* check options */
6399     while ((c = getopt(argc, argv, "f")) != -1) {
6400         switch (c) {
6401             case 'f':
6402                 flags = MS_FORCE;
6403                 break;
6404             case '?':
6405                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6406                     optopt);
6407                 (void) fprintf(stderr, gettext("usage: unmount [-f] "
6408                     "<path>\n"));
6409                 return (2);
6410             }
6411     }

6413     argc -= optind;
6414     argv += optind;

6416     /* check arguments */
6417     if (argc != 1) {
6418         if (argc == 0)
6419             (void) fprintf(stderr, gettext("missing path "
6420                 "argument\n"));
6421         else
6422             (void) fprintf(stderr, gettext("too many arguments\n"));
6423         (void) fprintf(stderr, gettext("usage: unmount [-f] <path>\n"));
6424         return (2);
6425     }

```

```

6427         return (unshare_unmount_path(OP_MOUNT, argv[0], flags, B_TRUE));
6428     }

6430 static int
6431 find_command_idx(char *command, int *idx)
6432 {
6433     int i;

6435     for (i = 0; i < NCOMMAND; i++) {
6436         if (command_table[i].name == NULL)
6437             continue;

6439         if (strcmp(command, command_table[i].name) == 0) {
6440             *idx = i;
6441             return (0);
6442         }
6443     }
6444     return (1);
6445 }

6447 static int
6448 zfs_do_diff(int argc, char **argv)
6449 {
6450     zfs_handle_t *zhp;
6451     int flags = 0;
6452     char *tosnap = NULL;
6453     char *fromsnap = NULL;
6454     char *atp, *copy;
6455     int err = 0;
6456     int c;

6458     while ((c = getopt(argc, argv, "FHT")) != -1) {
6459         switch (c) {
6460             case 'F':
6461                 flags |= ZFS_DIFF_CLASSIFY;
6462                 break;
6463             case 'H':
6464                 flags |= ZFS_DIFF_PARSEABLE;
6465                 break;
6466             case 't':
6467                 flags |= ZFS_DIFF_TIMESTAMP;
6468                 break;
6469             default:
6470                 (void) fprintf(stderr,
6471                     gettext("invalid option '%c'\n"), optopt);
6472                 usage(B_FALSE);
6473             }
6474         }

6476         argc -= optind;
6477         argv += optind;

6479         if (argc < 1) {
6480             (void) fprintf(stderr,
6481                 gettext("must provide at least one snapshot name\n"));
6482             usage(B_FALSE);
6483         }

6485         if (argc > 2) {
6486             (void) fprintf(stderr, gettext("too many arguments\n"));
6487             usage(B_FALSE);
6488         }

6490         fromsnap = argv[0];
6491         tosnap = (argc == 2) ? argv[1] : NULL;

```

```

6493     copy = NULL;
6494     if (*fromsnap != '@')
6495         copy = strdup(fromsnap);
6496     else if (tosnap)
6497         copy = strdup(tosnap);
6498     if (copy == NULL)
6499         usage(B_FALSE);

6501     if (atp = strchr(copy, '@'))
6502         *atp = '\0';

6504     if ((zhp = zfs_open(g_zfs, copy, ZFS_TYPE_FILESYSTEM)) == NULL)
6505         return (1);

6507     free(copy);

6509     /*
6510      * Ignore SIGPIPE so that the library can give us
6511      * information on any failure
6512      */
6513     (void) sigignore(SIGPIPE);

6515     err = zfs_show_diffs(zhp, STDOUT_FILENO, fromsnap, tosnap, flags);

6517     zfs_close(zhp);

6519     return (err != 0);
6520 }

6522 int
6523 main(int argc, char **argv)
6524 {
6525     int ret = 0;
6526     int i;
6527     char *progname;
6528     char *cmdname;

6530     (void) setlocale(LC_ALL, "");
6531     (void) textdomain(TEXT_DOMAIN);

6533     opterr = 0;

6535     if ((g_zfs = libzfs_init()) == NULL) {
6536         (void) fprintf(stderr, gettext("internal error: failed to "
6537             "initialize ZFS library\n"));
6538         return (1);
6539     }

6541     zfs_save_arguments(argc, argv, history_str, sizeof (history_str));
3257     zpool_set_history_str("zfs", argc, argv, history_str);
3258     verify(zpool_stage_history(g_zfs, history_str) == 0);

6543     libzfs_print_on_error(g_zfs, B_TRUE);

6545     if ((mnttab_file = fopen(MNTTAB, "r")) == NULL) {
6546         (void) fprintf(stderr, gettext("internal error: unable to "
6547             "open %s\n"), MNTTAB);
6548         return (1);
6549     }

6551     /*
6552      * This command also doubles as the /etc/fs mount and unmount program.
6553      * Determine if we should take this behavior based on argv[0].
6554      */
6555     progname = basename(argv[0]);

```

```

6556     if (strcmp(progname, "mount") == 0) {
6557         ret = manual_mount(argc, argv);
6558     } else if (strcmp(progname, "umount") == 0) {
6559         ret = manual_unmount(argc, argv);
6560     } else {
6561         /*
6562          * Make sure the user has specified some command.
6563          */
6564         if (argc < 2) {
6565             (void) fprintf(stderr, gettext("missing command\n"));
6566             usage(B_FALSE);
6567         }

6569         cmdname = argv[1];

6571         /*
6572          * The 'umount' command is an alias for 'unmount'
6573          */
6574         if (strcmp(cmdname, "umount") == 0)
6575             cmdname = "unmount";

6577         /*
6578          * The 'recv' command is an alias for 'receive'
6579          */
6580         if (strcmp(cmdname, "recv") == 0)
6581             cmdname = "receive";

6583         /*
6584          * Special case '-?'
6585          */
6586         if (strcmp(cmdname, "-?") == 0)
6587             usage(B_TRUE);

6589         /*
6590          * Run the appropriate command.
6591          */
6592         libzfs_mnttab_cache(g_zfs, B_TRUE);
6593         if (find_command_idx(cmdname, &i) == 0) {
6594             current_command = &command_table[i];
6595             ret = command_table[i].func(argc - 1, argv + 1);
6596         } else if (strchr(cmdname, '=') != NULL) {
6597             verify(find_command_idx("set", &i) == 0);
6598             current_command = &command_table[i];
6599             ret = command_table[i].func(argc, argv);
6600         } else {
6601             (void) fprintf(stderr, gettext("unrecognized "
6602                 "command '%s'\n"), cmdname);
6603             usage(B_FALSE);
6604         }
6605         libzfs_mnttab_cache(g_zfs, B_FALSE);
6606     }

6608     (void) fclose(mnttab_file);

6610     if (ret == 0 && log_history)
6611         (void) zpool_log_history(g_zfs, history_str);

6613 #endif /* ! codereview */
6614     libzfs_fini(g_zfs);

6616     /*
6617      * The 'ZFS_ABORT' environment variable causes us to dump core on exit
6618      * for the purposes of running ::findleaks.
6619      */
6620     if (getenv("ZFS_ABORT") != NULL) {
6621         (void) printf("dumping core by request\n");

```



```
6622         abort();
6623     }
6625     return (ret);
6626 }
```

```

*****
12734 Thu Jun 28 15:57:07 2012
new/usr/src/cmd/zhack/zhack.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

275 static void
276 feature_enable_sync(void *arg1, void *arg2, dmu_tx_t *tx)
277 {
278     spa_t *spa = arg1;
279     zfeature_info_t *feature = arg2;

281     spa_feature_enable(spa, feature, tx);
282     spa_history_log_internal(spa, "zhack enable feature", tx,
283         "name=%s can_readonly=%u",
284         feature->fi_guid, feature->fi_can_readonly);
285 #endif /* ! codereview */
286 }

288 static void
289 zhack_do_feature_enable(int argc, char **argv)
290 {
291     char c;
292     char *desc, *target;
293     spa_t *spa;
294     objset_t *mos;
295     zfeature_info_t feature;
296     zfeature_info_t *nodeps[] = { NULL };

298     /*
299      * Features are not added to the pool's label until their refcounts
300      * are incremented, so fi_mos can just be left as false for now.
301      */
302     desc = NULL;
303     feature.fi_undef = "zhack";
304     feature.fi_mos = B_FALSE;
305     feature.fi_can_readonly = B_FALSE;
306     feature.fi_depends = nodeps;

308     optind = 1;
309     while ((c = getopt(argc, argv, "rmd:")) != -1) {
310         switch (c) {
311             case 'r':
312                 feature.fi_can_readonly = B_TRUE;
313                 break;
314             case 'd':
315                 desc = strdup(optarg);
316                 break;
317             default:
318                 usage();
319                 break;
320         }
321     }

323     if (desc == NULL)
324         desc = strdup("zhack injected");
325     feature.fi_desc = desc;

```

```

327     argc -= optind;
328     argv += optind;

330     if (argc < 2) {
331         (void) fprintf(stderr, "error: missing feature or pool name\n");
332         usage();
333     }
334     target = argv[0];
335     feature.fi_guid = argv[1];

337     if (!zfeature_is_valid_guid(feature.fi_guid))
338         fatal("invalid feature guid: %s", feature.fi_guid);

340     zhack_spa_open(target, B_FALSE, FTAG, &spa);
341     mos = spa->spa_meta_objset;

343     if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
344         fatal("'s' is a real feature, will not enable");
345     if (0 == zap_contains(mos, spa->spa_feat_desc_obj, feature.fi_guid))
346         fatal("feature already enabled: %s", feature.fi_guid);

348     VERIFY3U(0, ==, dsl_sync_task_do(spa->spa_dsl_pool, NULL,
349         feature_enable_sync, spa, &feature, 5));

351     spa_close(spa, FTAG);

353     free(desc);
354 }

356 static void
357 feature_incr_sync(void *arg1, void *arg2, dmu_tx_t *tx)
358 {
359     spa_t *spa = arg1;
360     zfeature_info_t *feature = arg2;

362     spa_feature_incr(spa, feature, tx);
363     spa_history_log_internal(spa, "zhack feature incr", tx,
364         "name=%s", feature->fi_guid);
365 #endif /* ! codereview */
366 }

368 static void
369 feature_decr_sync(void *arg1, void *arg2, dmu_tx_t *tx)
370 {
371     spa_t *spa = arg1;
372     zfeature_info_t *feature = arg2;

374     spa_feature_decr(spa, feature, tx);
375     spa_history_log_internal(spa, "zhack feature decr", tx,
376         "name=%s", feature->fi_guid);
377 #endif /* ! codereview */
378 }

380 static void
381 zhack_do_feature_ref(int argc, char **argv)
382 {
383     char c;
384     char *target;
385     boolean_t decr = B_FALSE;
386     spa_t *spa;
387     objset_t *mos;
388     zfeature_info_t feature;
389     zfeature_info_t *nodeps[] = { NULL };

391     /*
392      * fi_desc does not matter here because it was written to disk

```

```

393     * when the feature was enabled, but we need to properly set the
394     * feature for read or write based on the information we read off
395     * disk later.
396     */
397     feature.fi_uname = "zhack";
398     feature.fi_mos = B_FALSE;
399     feature.fi_desc = NULL;
400     feature.fi_depends = nodeps;

402     optind = 1;
403     while ((c = getopt(argc, argv, "md")) != -1) {
404         switch (c) {
405             case 'm':
406                 feature.fi_mos = B_TRUE;
407                 break;
408             case 'd':
409                 decr = B_TRUE;
410                 break;
411             default:
412                 usage();
413                 break;
414         }
415     }
416     argc -= optind;
417     argv += optind;

419     if (argc < 2) {
420         (void) fprintf(stderr, "error: missing feature or pool name\n");
421         usage();
422     }
423     target = argv[0];
424     feature.fi_guid = argv[1];

426     if (!zfeature_is_valid_guid(feature.fi_guid))
427         fatal("invalid feature guid: %s", feature.fi_guid);

429     zhack_spa_open(target, B_FALSE, FTAG, &spa);
430     mos = spa->spa_meta_objset;

432     if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
433         fatal("'s' is a real feature, will not change refcount");

435     if (0 == zap_contains(mos, spa->spa_feat_for_read_obj,
436         feature.fi_guid)) {
437         feature.fi_can_readonly = B_FALSE;
438     } else if (0 == zap_contains(mos, spa->spa_feat_for_write_obj,
439         feature.fi_guid)) {
440         feature.fi_can_readonly = B_TRUE;
441     } else {
442         fatal("feature is not enabled: %s", feature.fi_guid);
443     }

445     if (decr && !spa_feature_is_active(spa, &feature))
446         fatal("feature refcount already 0: %s", feature.fi_guid);

448     VERIFY3U(0, ==, dsl_sync_task_do(spa->spa_dsl_pool, NULL,
449         decr ? feature_decr_sync : feature_incr_sync, spa, &feature, 5));

451     spa_close(spa, FTAG);
452 }

454 static int
455 zhack_do_feature(int argc, char **argv)
456 {
457     char *subcommand;

```

```

459     argc--;
460     argv++;
461     if (argc == 0) {
462         (void) fprintf(stderr,
463             "error: no feature operation specified\n");
464         usage();
465     }

467     subcommand = argv[0];
468     if (strcmp(subcommand, "stat") == 0) {
469         zhack_do_feature_stat(argc, argv);
470     } else if (strcmp(subcommand, "enable") == 0) {
471         zhack_do_feature_enable(argc, argv);
472     } else if (strcmp(subcommand, "ref") == 0) {
473         zhack_do_feature_ref(argc, argv);
474     } else {
475         (void) fprintf(stderr, "error: unknown subcommand: %s\n",
476             subcommand);
477         usage();
478     }

480     return (0);
481 }

483 #define MAX_NUM_PATHS 1024

485 int
486 main(int argc, char **argv)
487 {
488     extern void zfs_prop_init(void);

490     char *path[MAX_NUM_PATHS];
491     const char *subcommand;
492     int rv = 0;
493     char c;

495     g_importargs.path = path;

497     dprintf_setup(&argc, argv);
498     zfs_prop_init();

500     while ((c = getopt(argc, argv, "c:d:")) != -1) {
501         switch (c) {
502             case 'c':
503                 g_importargs.cachefile = optarg;
504                 break;
505             case 'd':
506                 assert(g_importargs.paths < MAX_NUM_PATHS);
507                 g_importargs.path[g_importargs.paths++] = optarg;
508                 break;
509             default:
510                 usage();
511                 break;
512         }
513     }

515     argc -= optind;
516     argv += optind;
517     optind = 1;

519     if (argc == 0) {
520         (void) fprintf(stderr, "error: no command specified\n");
521         usage();
522     }

524     subcommand = argv[0];

```

```
526     if (strcmp(subcommand, "feature") == 0) {
527         rv = zhack_do_feature(argc, argv);
528     } else {
529         (void) fprintf(stderr, "error: unknown subcommand: %s\n",
530             subcommand);
531         usage();
532     }
533
534     if (!g_readonly && spa_export(g_pool, NULL, B_TRUE, B_TRUE) != 0) {
535         fatal("pool export failed; "
536             "changes may not be committed to disk\n");
537     }
538
539     libzfs_fini(g_zfs);
540     kernel_fini();
541
542     return (rv);
543 }
```

```

*****
123099 Thu Jun 28 15:57:08 2012
new/usr/src/cmd/zpool/zpool_main.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
unchanged_portion_omitted_

186 #define NCOMMAND      (sizeof (command_table) / sizeof (command_table[0]))

188 static zpool_command_t *current_command;
188 zpool_command_t *current_command;
189 static char history_str[HIS_MAX_RECORD_LEN];
190 static boolean_t log_history = B_TRUE;

191 static uint_t timestamp_fmt = NODATE;

193 static const char *
194 get_usage(zpool_help_t idx) {
195     switch (idx) {
196     case HELP_ADD:
197         return (gettext("\tadd [-fn] <pool> <vdev> ... \n"));
198     case HELP_ATTACH:
199         return (gettext("\tattach [-f] <pool> <device> "
200             "<new-device>\n"));
201     case HELP_CLEAR:
202         return (gettext("\tclear [-nF] <pool> [device]\n"));
203     case HELP_CREATE:
204         return (gettext("\tcreate [-fnd] [-o property=value] ... \n"
205             "\t\t [-O file-system-property=value] ... \n"
206             "\t\t [-m mountpoint] [-R root] <pool> <vdev> ... \n"));
207     case HELP_DESTROY:
208         return (gettext("\tdestroy [-f] <pool>\n"));
209     case HELP_DETACH:
210         return (gettext("\tdetach <pool> <device>\n"));
211     case HELP_EXPORT:
212         return (gettext("\texport [-f] <pool> ... \n"));
213     case HELP_HISTORY:
214         return (gettext("\thistory [-il] [<pool>] ... \n"));
215     case HELP_IMPORT:
216         return (gettext("\timport [-d dir] [-D]\n"
217             "\t\timport [-d dir | -c cachefile] [-F [-n]] <pool | id>\n"
218             "\t\timport [-o mntopts] [-o property=value] ... \n"
219             "\t\t [-d dir | -c cachefile] [-D] [-f] [-m] [-N] "
220             "[-R root] [-F [-n]] -a\n"
221             "\t\timport [-o mntopts] [-o property=value] ... \n"
222             "\t\t [-d dir | -c cachefile] [-D] [-f] [-m] [-N] "
223             "[-R root] [-F [-n]]\n"
224             "\t\t <pool | id> [newpool]\n"));
225     case HELP_IOSTAT:
226         return (gettext("\tiostat [-v] [-T d|u] [pool] ... [interval "
227             "[count]]\n"));
228     case HELP_LIST:
229         return (gettext("\tlist [-H] [-o property[,...]] "
230             "[-T d|u] [pool] ... [interval [count]]\n"));
231     case HELP_OFFLINE:
232         return (gettext("\toffline [-t] <pool> <device> ... \n"));
233     case HELP_ONLINE:
234         return (gettext("\tonline <pool> <device> ... \n"));
235     case HELP_REPLACE:

```

```

236         return (gettext("\treplace [-f] <pool> <device> "
237             "[new-device]\n"));
238     case HELP_REMOVE:
239         return (gettext("\tremove <pool> <device> ... \n"));
240     case HELP_REOPEN:
241         return (""); /* Undocumented command */
242     case HELP_SCRUB:
243         return (gettext("\tscrub [-s] <pool> ... \n"));
244     case HELP_STATUS:
245         return (gettext("\tstatus [-vx] [-T d|u] [pool] ... [interval "
246             "[count]]\n"));
247     case HELP_UPGRADE:
248         return (gettext("\tupgrade\n"
249             "\t\tupgrade -v\n"
250             "\t\tupgrade [-V version] <-a | pool ...>\n"));
251     case HELP_GET:
252         return (gettext("\tget <\"all\" | property[,...]> "
253             "<pool> ... \n"));
254     case HELP_SET:
255         return (gettext("\tset <property=value> <pool> \n"));
256     case HELP_SPLIT:
257         return (gettext("\tsplit [-n] [-R altroot] [-o mntopts]\n"
258             "\t\t [-o property=value] <pool> <newpool> "
259             "[<device> ...]\n"));
260     case HELP_REGUID:
261         return (gettext("\treguid <pool>\n"));
262     }

264     abort();
265     /* NOTREACHED */
266 }
unchanged_portion_omitted_

877 /*
878 * zpool destroy <pool>
879 *
880 * -f Forcefully unmount any datasets
881 *
882 * Destroy the given pool. Automatically unmounts any datasets in the pool.
883 */
884 int
885 zpool_do_destroy(int argc, char **argv)
886 {
887     boolean_t force = B_FALSE;
888     int c;
889     char *pool;
890     zpool_handle_t *zhp;
891     int ret;

893     /* check options */
894     while ((c = getopt(argc, argv, "f")) != -1) {
895         switch (c) {
896             case 'f':
897                 force = B_TRUE;
898                 break;
899             case '?':
900                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
901                     optopt);
902                 usage(B_FALSE);
903             }
904     }

906     argc -= optind;
907     argv += optind;

909     /* check arguments */

```

```

910     if (argc < 1) {
911         (void) fprintf(stderr, gettext("missing pool argument\n"));
912         usage(B_FALSE);
913     }
914     if (argc > 1) {
915         (void) fprintf(stderr, gettext("too many arguments\n"));
916         usage(B_FALSE);
917     }
918
919     pool = argv[0];
920
921     if ((zhp = zpool_open_canfail(g_zfs, pool)) == NULL) {
922         /*
923          * As a special case, check for use of '/' in the name, and
924          * direct the user to use 'zfs destroy' instead.
925          */
926         if (strchr(pool, '/') != NULL)
927             (void) fprintf(stderr, gettext("use 'zfs destroy' to "
928                 "destroy a dataset\n"));
929         return (1);
930     }
931
932     if (zpool_disable_datasets(zhp, force) != 0) {
933         (void) fprintf(stderr, gettext("could not destroy '%s': "
934             "could not unmount datasets\n"), zpool_get_name(zhp));
935         return (1);
936     }
937
938     /* The history must be logged as part of the export */
939     log_history = B_FALSE;
940
941     ret = (zpool_destroy(zhp, history_str) != 0);
942     ret = (zpool_destroy(zhp) != 0);
943
944     zpool_close(zhp);
945
946     return (ret);
947 }
948
949 /*
950 * zpool export [-f] <pool> ...
951 *
952 * -f Forcefully unmount datasets
953 *
954 * Export the given pools. By default, the command will attempt to cleanly
955 * unmount any active datasets within the pool. If the '-f' flag is specified,
956 * then the datasets will be forcefully unmounted.
957 */
958 int
959 zpool_do_export(int argc, char **argv)
960 {
961     boolean_t force = B_FALSE;
962     boolean_t hardforce = B_FALSE;
963     int c;
964     zpool_handle_t *zhp;
965     int ret;
966     int i;
967
968     /* check options */
969     while ((c = getopt(argc, argv, "fF")) != -1) {
970         switch (c) {
971             case 'f':
972                 force = B_TRUE;
973                 break;
974             case 'F':
975                 hardforce = B_TRUE;

```

```

976         break;
977     case '?':
978         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
979             optopt);
980         usage(B_FALSE);
981     }
982
983     argc -= optind;
984     argv += optind;
985
986     /* check arguments */
987     if (argc < 1) {
988         (void) fprintf(stderr, gettext("missing pool argument\n"));
989         usage(B_FALSE);
990     }
991
992     ret = 0;
993     for (i = 0; i < argc; i++) {
994         if ((zhp = zpool_open_canfail(g_zfs, argv[i])) == NULL) {
995             ret = 1;
996             continue;
997         }
998
999         if (zpool_disable_datasets(zhp, force) != 0) {
1000             ret = 1;
1001             zpool_close(zhp);
1002             continue;
1003         }
1004
1005         /* The history must be logged as part of the export */
1006         log_history = B_FALSE;
1007
1008         #endif /* ! codereview */
1009         if (hardforce) {
1010             if (zpool_export_force(zhp, history_str) != 0)
1011                 if (zpool_export_force(zhp) != 0)
1012                     ret = 1;
1013             } else if (zpool_export(zhp, force, history_str) != 0) {
1014                 if (zpool_export(zhp, force) != 0)
1015                     ret = 1;
1016             }
1017         }
1018         zpool_close(zhp);
1019     }
1020     return (ret);
1021 }
1022
1023 unchanged_portion_omitted
1024
1025 static int
1026 upgrade_cb(zpool_handle_t *zhp, void *arg)
1027 {
1028     upgrade_cbdata_t *cbp = arg;
1029     nvlist_t *config;
1030     uint64_t version;
1031     int ret = 0;
1032
1033     config = zpool_get_config(zhp, NULL);
1034     verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_VERSION,
1035         &version) == 0);
1036
1037     if (!cbp->cb_newer && SPA_VERSION_IS_SUPPORTED(version) &&
1038         version != SPA_VERSION) {
1039         if (!cbp->cb_all) {
1040             if (cbp->cb_first) {

```

```

4260         (void) printf(gettext("The following pools are "
4261             "out of date, and can be upgraded. After "
4262             "being\nupgraded, these pools will no "
4263             "longer be accessible by older software "
4264             "versions.\n\n"));
4265         (void) printf(gettext("VER POOL\n"));
4266         (void) printf(gettext("--- -----\n"));
4267         cbp->cb_first = B_FALSE;
4268     }

4270     (void) printf("%2llu  %s\n", (u_longlong_t)version,
4271         zpool_get_name(zhp));
4272 } else {
4273     cbp->cb_first = B_FALSE;
4274     ret = zpool_upgrade(zhp, cbp->cb_version);
4275     if (!ret) {
4276         (void) printf(gettext("Successfully upgraded "
4277             "'%s'\n\n"), zpool_get_name(zhp));
4278     }
4279     /*
4280     * If they did "zpool upgrade -a", then we could
4281     * be doing ioctl's to different pools. We need
4282     * to log this history once to each pool, and bypass
4283     * the normal history logging that happens in main().
4284     */
4285     (void) zpool_log_history(g_zfs, history_str);
4286     log_history = B_FALSE;
4287 #endif /* ! codereview */
4288 }
4289 } else if (cbp->cb_newer && !SPA_VERSION_IS_SUPPORTED(version)) {
4290     assert(!cbp->cb_all);

4292     if (cbp->cb_first) {
4293         (void) printf(gettext("The following pools are "
4294             "formatted using an unsupported software version "
4295             "and\ncannot be accessed on the current "
4296             "system.\n\n"));
4297         (void) printf(gettext("VER POOL\n"));
4298         (void) printf(gettext("--- -----\n"));
4299         cbp->cb_first = B_FALSE;
4300     }

4302     (void) printf("%2llu  %s\n", (u_longlong_t)version,
4303         zpool_get_name(zhp));
4304 }

4306     zpool_close(zhp);
4307     return (ret);
4308 }

4310 /* ARGSUSED */
4311 static int
4312 upgrade_one(zpool_handle_t *zhp, void *data)
4313 {
4314     upgrade_cbdata_t *cbp = data;
4315     uint64_t cur_version;
4316     int ret;

4318     if (strcmp("log", zpool_get_name(zhp)) == 0) {
4319         (void) printf(gettext("'log' is now a reserved word\n"
4320             "Pool 'log' must be renamed using export and import "
4321             "to upgrade.\n"));
4322         return (1);
4323     }

4325     cur_version = zpool_get_prop_int(zhp, ZPOOL_PROP_VERSION, NULL);

```

```

4326     if (cur_version > cbp->cb_version) {
4327         (void) printf(gettext("Pool '%s' is already formatted "
4328             "using more current version '%llu'.\n"),
4329             zpool_get_name(zhp), cur_version);
4330         return (0);
4331     }
4332     if (cur_version == cbp->cb_version) {
4333         (void) printf(gettext("Pool '%s' is already formatted "
4334             "using the current version.\n"), zpool_get_name(zhp));
4335         return (0);
4336     }

4338     ret = zpool_upgrade(zhp, cbp->cb_version);

4340     if (!ret) {
4341         (void) printf(gettext("Successfully upgraded '%s' "
4342             "from version %llu to version %llu\n\n"),
4343             zpool_get_name(zhp), (u_longlong_t)cur_version,
4344             (u_longlong_t)cbp->cb_version);
4345     }

4347     return (ret != 0);
4348 }

4350 /*
4351  * zpool upgrade
4352  * zpool upgrade -v
4353  * zpool upgrade [-V version] <-a | pool ...>
4354  *
4355  * With no arguments, display downrev'd ZFS pool available for upgrade.
4356  * Individual pools can be upgraded by specifying the pool, and '-a' will
4357  * upgrade all pools.
4358  */
4359 int
4360 zpool_do_upgrade(int argc, char **argv)
4361 {
4362     int c;
4363     upgrade_cbdata_t cb = { 0 };
4364     int ret = 0;
4365     boolean_t showversions = B_FALSE;
4366     char *end;

4369     /* check options */
4370     while ((c = getopt(argc, argv, "avV:")) != -1) {
4371         switch (c) {
4372             case 'a':
4373                 cb.cb_all = B_TRUE;
4374                 break;
4375             case 'v':
4376                 showversions = B_TRUE;
4377                 break;
4378             case 'V':
4379                 cb.cb_version = strtoll(optarg, &end, 10);
4380                 if (*end != '\0' ||
4381                     !SPA_VERSION_IS_SUPPORTED(cb.cb_version)) {
4382                     (void) fprintf(stderr,
4383                         gettext("invalid version '%s'\n"), optarg);
4384                     usage(B_FALSE);
4385                 }
4386                 break;
4387             case ':':
4388                 (void) fprintf(stderr, gettext("missing argument for "
4389                     "'%c' option\n"), optopt);
4390                 usage(B_FALSE);
4391                 break;

```

```

4392     case '?':
4393         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
4394             optarg);
4395         usage(B_FALSE);
4396     }
4397 }
4399 cb.cb_argc = argc;
4400 cb.cb_argv = argv;
4401 argc -= optind;
4402 argv += optind;
4404 if (cb.cb_version == 0) {
4405     cb.cb_version = SPA_VERSION;
4406 } else if (!cb.cb_all && argc == 0) {
4407     (void) fprintf(stderr, gettext("-V option is "
4408         "incompatible with other arguments\n"));
4409     usage(B_FALSE);
4410 }
4412 if (showversions) {
4413     if (cb.cb_all || argc != 0) {
4414         (void) fprintf(stderr, gettext("-v option is "
4415             "incompatible with other arguments\n"));
4416         usage(B_FALSE);
4417     }
4418 } else if (cb.cb_all) {
4419     if (argc != 0) {
4420         (void) fprintf(stderr, gettext("-a option should not "
4421             "be used along with a pool name\n"));
4422         usage(B_FALSE);
4423     }
4424 }
4426 (void) printf(gettext("This system supports ZFS pool feature "
4427     "flags.\n\n"));
4428 cb.cb_first = B_TRUE;
4429 if (showversions) {
4430     (void) printf(gettext("The following versions are "
4431         "supported:\n\n"));
4432     (void) printf(gettext("VER  DESCRIPTION\n"));
4433     (void) printf("-----\n");
4434     (void) printf("-----\n");
4435     (void) printf(gettext(" 1  Initial ZFS version\n"));
4436     (void) printf(gettext(" 2  Ditto blocks "
4437         "(replicated metadata)\n"));
4438     (void) printf(gettext(" 3  Hot spares and double parity "
4439         "RAID-Z\n"));
4440     (void) printf(gettext(" 4  zpool history\n"));
4441     (void) printf(gettext(" 5  Compression using the gzip "
4442         "algorithm\n"));
4443     (void) printf(gettext(" 6  bootfs pool property\n"));
4444     (void) printf(gettext(" 7  Separate intent log devices\n"));
4445     (void) printf(gettext(" 8  Delegated administration\n"));
4446     (void) printf(gettext(" 9  refquota and reservation "
4447         "properties\n"));
4448     (void) printf(gettext("10  Cache devices\n"));
4449     (void) printf(gettext("11  Improved scrub performance\n"));
4450     (void) printf(gettext("12  Snapshot properties\n"));
4451     (void) printf(gettext("13  snapused property\n"));
4452     (void) printf(gettext("14  passthrough-x aclinherit\n"));
4453     (void) printf(gettext("15  user/group space accounting\n"));
4454     (void) printf(gettext("16  stmf property support\n"));
4455     (void) printf(gettext("17  Triple-parity RAID-Z\n"));
4456     (void) printf(gettext("18  Snapshot user holds\n"));
4457     (void) printf(gettext("19  Log device removal\n"));

```

```

4458     (void) printf(gettext(" 20 Compression using zle "
4459         "(zero-length encoding)\n"));
4460     (void) printf(gettext(" 21 Deduplication\n"));
4461     (void) printf(gettext(" 22 Received properties\n"));
4462     (void) printf(gettext(" 23 Slim ZIL\n"));
4463     (void) printf(gettext(" 24 System attributes\n"));
4464     (void) printf(gettext(" 25 Improved scrub stats\n"));
4465     (void) printf(gettext(" 26 Improved snapshot deletion "
4466         "performance\n"));
4467     (void) printf(gettext(" 27 Improved snapshot creation "
4468         "performance\n"));
4469     (void) printf(gettext(" 28 Multiple vdev replacements\n"));
4470     (void) printf(gettext("\nFor more information on a particular "
4471         "version, including supported releases,\n"));
4472     (void) printf(gettext("see the ZFS Administration Guide.\n\n"));
4473 } else if (argc == 0) {
4474     int notfound;
4476     ret = zpool_iter(g_zfs, upgrade_cb, &cb);
4477     notfound = cb.cb_first;
4479     if (!cb.cb_all && ret == 0) {
4480         if (!cb.cb_first)
4481             (void) printf("\n");
4482         cb.cb_first = B_TRUE;
4483         cb.cb_newer = B_TRUE;
4484         ret = zpool_iter(g_zfs, upgrade_cb, &cb);
4485         if (!cb.cb_first) {
4486             notfound = B_FALSE;
4487             (void) printf("\n");
4488         }
4489     }
4491     if (ret == 0) {
4492         if (notfound)
4493             (void) printf(gettext("All pools are formatted "
4494                 "using this version.\n"));
4495         else if (!cb.cb_all)
4496             (void) printf(gettext("Use 'zpool upgrade -v' "
4497                 "for a list of available versions and "
4498                 "their associated\nfeatures.\n"));
4499     }
4500 } else {
4501     ret = for_each_pool(argc, argv, B_FALSE, NULL,
4502         upgrade_one, &cb);
4503 }
4505 return (ret);
4506 }
4508 typedef struct hist_cbdata {
4509     boolean_t first;
4510     boolean_t longfmt;
4511     boolean_t internal;
4512     int longfmt;
4513     int internal;
4514 } hist_cbdata_t;
4514 /*
4515  * Print out the command history for a specific pool.
4516  */
4517 static int
4518 get_history_one(zpool_handle_t *zhp, void *data)
4519 {
4520     nvlist_t *nvhis;
4521     nvlist_t **records;

```



```

4522     uint_t numrecords;
4284     char *cmdstr;
4285     char *pathstr;
4286     uint64_t dst_time;
4287     time_t tsec;
4288     struct tm t;
4289     char tbuf[30];
4523     int ret, i;
4291     uint64_t who;
4292     struct passwd *pwd;
4293     char *hostname;
4294     char *zonename;
4295     char internalstr[MAXPATHLEN];
4524     hist_cbdata_t *cb = (hist_cbdata_t *)data;
4297     uint64_t txg;
4298     uint64_t ievent;

4526     cb->first = B_FALSE;

4528     (void) printf(gettext("History for '%s':\n"), zpool_get_name(zhp));

4530     if ((ret = zpool_get_history(zhp, &nvhis)) != 0)
4531         return (ret);

4533     verify(nvlist_lookup_nvlist_array(nvhis, ZPOOL_HIST_RECORD,
4534         &records, &numrecords) == 0);
4535     for (i = 0; i < numrecords; i++) {
4536         nvlist_t *rec = records[i];
4537         char tbuf[30] = "";

4539         if (nvlist_exists(rec, ZPOOL_HIST_TIME)) {
4540             time_t tsec;
4541             struct tm t;

4543             tsec = fnvlist_lookup_uint64(records[i],
4544                 ZPOOL_HIST_TIME);
4545             (void) localtime_r(&tsec, &t);
4546             (void) strftime(tbuf, sizeof (tbuf), "%F.%T", &t);
4547         }

4549         if (nvlist_exists(rec, ZPOOL_HIST_CMD)) {
4550             (void) printf("%s %s", tbuf,
4551                 fnvlist_lookup_string(rec, ZPOOL_HIST_CMD));
4552         } else if (nvlist_exists(rec, ZPOOL_HIST_INT_EVENT)) {
4553             int ievent =
4554                 fnvlist_lookup_uint64(rec, ZPOOL_HIST_INT_EVENT);
4555             if (!cb->internal)
4556                 if (nvlist_lookup_uint64(records[i], ZPOOL_HIST_TIME,
4557                     &dst_time) != 0)
4558                     continue;
4559             if (ievent >= ZFS_NUM_LEGACY_HISTORY_EVENTS) {
4560                 (void) printf("%s unrecognized record:\n",
4561                     tbuf);
4562                 dump_nvlist(rec, 4);
4563                 continue;
4564             }
4565             (void) printf("%s [internal %s txg:%lld] %s", tbuf,
4566                 zfs_history_event_names[ievent],
4567                 fnvlist_lookup_uint64(rec, ZPOOL_HIST_TXG),
4568                 fnvlist_lookup_string(rec, ZPOOL_HIST_INT_STR));
4569         } else if (nvlist_exists(rec, ZPOOL_HIST_INT_NAME)) {
4570             if (!cb->internal)

```

```

4314     /* is it an internal event or a standard event? */
4315     if (nvlist_lookup_string(records[i], ZPOOL_HIST_CMD,
4316         &cmdstr) != 0) {

```

```

4317         if (cb->internal == 0)
4318             continue;
4319         (void) printf("%s [txg:%lld] %s", tbuf,
4320             fnvlist_lookup_uint64(rec, ZPOOL_HIST_TXG),
4321             fnvlist_lookup_string(rec, ZPOOL_HIST_INT_NAME));
4322         if (nvlist_exists(rec, ZPOOL_HIST_DSNAME)) {
4323             (void) printf(" %s (%llu)",
4324                 fnvlist_lookup_string(rec,
4325                     ZPOOL_HIST_DSNAME),
4326                 fnvlist_lookup_uint64(rec,
4327                     ZPOOL_HIST_DSID));
4328         }
4329         (void) printf(" %s", fnvlist_lookup_string(rec,
4330             ZPOOL_HIST_INT_STR));
4331     } else if (nvlist_exists(rec, ZPOOL_HIST_IOCTL)) {
4332         if (!cb->internal)
4333             if (nvlist_lookup_uint64(records[i],
4334                 ZPOOL_HIST_INT_EVENT, &ievent) != 0)
4335                 continue;
4336         (void) printf("%s ioctl %s\n", tbuf,
4337             fnvlist_lookup_string(rec, ZPOOL_HIST_IOCTL));
4338         if (nvlist_exists(rec, ZPOOL_HIST_INPUT_NVLIST)) {
4339             (void) printf("   input:\n");
4340             dump_nvlist(fnvlist_lookup_nvlist(rec,
4341                 ZPOOL_HIST_INPUT_NVLIST), 8);
4342         }
4343         if (nvlist_exists(rec, ZPOOL_HIST_OUTPUT_NVLIST)) {
4344             (void) printf("   output:\n");
4345             dump_nvlist(fnvlist_lookup_nvlist(rec,
4346                 ZPOOL_HIST_OUTPUT_NVLIST), 8);
4347         }
4348     } else {
4349         if (!cb->internal)
4350             verify(nvlist_lookup_uint64(records[i],
4351                 ZPOOL_HIST_TXG, &txg) == 0);
4352             verify(nvlist_lookup_string(records[i],
4353                 ZPOOL_HIST_INT_STR, &pathstr) == 0);
4354             if (ievent >= LOG_END)
4355                 continue;
4356             (void) printf("%s unrecognized record:\n", tbuf);
4357             dump_nvlist(rec, 4);
4358             (void) snprintf(internalstr,
4359                 sizeof (internalstr),
4360                 "[internal %s txg:%lld] %s",
4361                 zfs_history_event_names[ievent], txg,
4362                 pathstr);
4363             cmdstr = internalstr;
4364         }
4365         tsec = dst_time;
4366         (void) localtime_r(&tsec, &t);
4367         (void) strftime(tbuf, sizeof (tbuf), "%F.%T", &t);
4368         (void) printf("%s %s", tbuf, cmdstr);

4370         if (!cb->longfmt) {
4371             (void) printf("\n");
4372             continue;
4373         }
4374         (void) printf(" [");
4375         if (nvlist_exists(rec, ZPOOL_HIST_WHO)) {
4376             uid_t who = fnvlist_lookup_uint64(rec, ZPOOL_HIST_WHO);
4377             struct passwd *pwd = getpwuid(who);
4378             (void) printf("user %d ", (int)who);
4379             if (pwd != NULL)
4380                 (void) printf("(%s) ", pwd->pw_name);
4381         }

```

```

4616     if (nvlist_exists(rec, ZPOOL_HIST_HOST)) {
4617         (void) printf("on %s",
4618             fnvlist_lookup_string(rec, ZPOOL_HIST_HOST));
4619     }
4620     if (nvlist_exists(rec, ZPOOL_HIST_ZONE)) {
4621         (void) printf(":%s",
4622             fnvlist_lookup_string(rec, ZPOOL_HIST_ZONE));
4623     }
4624     if (nvlist_lookup_uint64(records[i],
4625         ZPOOL_HIST_WHO, &who) == 0) {
4626         pwd = getpwuid((uid_t)who);
4627         if (pwd)
4628             (void) printf("user %s on",
4629                 pwd->pw_name);
4630     } else
4631         (void) printf("user %d on",
4632             (int)who);
4633     } else {
4634         (void) printf(gettext("no info\n"));
4635         continue;
4636     }
4637     if (nvlist_lookup_string(records[i],
4638         ZPOOL_HIST_HOST, &hostname) == 0) {
4639         (void) printf(" %s", hostname);
4640     }
4641     if (nvlist_lookup_string(records[i],
4642         ZPOOL_HIST_ZONE, &zonename) == 0) {
4643         (void) printf(":%s", zonename);
4644     }
4645     (void) printf("]");
4646     (void) printf("\n");
4647 }
4648 (void) printf("\n");
4649 nvlist_free(nvhis);
4650
4651 return (ret);
4652 }
4653
4654 /*
4655  * zpool history <pool>
4656  * Displays the history of commands that modified pools.
4657  */
4658
4659 int
4660 zpool_do_history(int argc, char **argv)
4661 {
4662     hist_cbdata_t cbdata = { 0 };
4663     int ret;
4664     int c;
4665
4666     cbdata.first = B_TRUE;
4667     /* check options */
4668     while ((c = getopt(argc, argv, "li")) != -1) {
4669         switch (c) {
4670             case 'l':
4671                 cbdata.longfmt = B_TRUE;
4672                 cbdata.longfmt = 1;
4673                 break;
4674             case 'i':
4675                 cbdata.internal = B_TRUE;
4676                 cbdata.internal = 1;
4677                 break;
4678             case '?':
4679                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
4680                     c);

```

```

4681         optopt);
4682         usage(B_FALSE);
4683     }
4684     argc -= optind;
4685     argv += optind;
4686
4687     ret = for_each_pool(argc, argv, B_FALSE, NULL, get_history_one,
4688         &cbdata);
4689
4690     if (argc == 0 && cbdata.first == B_TRUE) {
4691         (void) printf(gettext("no pools available\n"));
4692         return (0);
4693     }
4694
4695     return (ret);
4696 }
4697
4698 unchanged portion omitted
4699
4700 int
4701 main(int argc, char **argv)
4702 {
4703     int ret;
4704     int i;
4705     char *cmdname;
4706
4707     (void) setlocale(LC_ALL, "");
4708     (void) textdomain(TEXT_DOMAIN);
4709
4710     if ((g_zfs = libzfs_init()) == NULL) {
4711         (void) fprintf(stderr, gettext("internal error: failed to "
4712             "initialize ZFS library\n"));
4713         return (1);
4714     }
4715
4716     libzfs_print_on_error(g_zfs, B_TRUE);
4717
4718     opterr = 0;
4719
4720     /*
4721      * Make sure the user has specified some command.
4722      */
4723     if (argc < 2) {
4724         (void) fprintf(stderr, gettext("missing command\n"));
4725         usage(B_FALSE);
4726     }
4727
4728     cmdname = argv[1];
4729
4730     /*
4731      * Special case '--?'
4732      */
4733     if (strcmp(cmdname, "--?") == 0)
4734         usage(B_TRUE);
4735
4736     zfs_save_arguments(argc, argv, history_str, sizeof(history_str));
4737     zpool_set_history_str("zpool", argc, argv, history_str);
4738     verify(zpool_stage_history(g_zfs, history_str) == 0);
4739
4740     /*
4741      * Run the appropriate command.
4742      */
4743     if (find_command_idx(cmdname, &i) == 0) {
4744         current_command = &command_table[i];
4745         ret = command_table[i].func(argc - 1, argv + 1);
4746     } else if (strchr(cmdname, '=') {

```

```
4887         verify(find_command_idx("set", &i) == 0);
4888         current_command = &command_table[i];
4889         ret = command_table[i].func(argc, argv);
4890     } else if (strcmp(cmdname, "freeze") == 0 && argc == 3) {
4891         /*
4892          * 'freeze' is a vile debugging abomination, so we treat
4893          * it as such.
4894          */
4895         char buf[16384];
4896         int fd = open(ZFS_DEV, O_RDWR);
4897         (void) strcpy((void *)buf, argv[2]);
4898         return (!!ioctl(fd, ZFS_IOC_POOL_FREEZE, buf));
4899     } else {
4900         (void) fprintf(stderr, gettext("unrecognized "
4901         "command '%s'\n"), cmdname);
4902         usage(B_FALSE);
4903     }
4904
4905     if (ret == 0 && log_history)
4906         (void) zpool_log_history(g_zfs, history_str);
4907
4908 #endif /* ! codereview */
4909     libzfs_fini(g_zfs);
4910
4911     /*
4912     * The 'ZFS_ABORT' environment variable causes us to dump core on exit
4913     * for the purposes of running ::findleaks.
4914     */
4915     if (getenv("ZFS_ABORT") != NULL) {
4916         (void) printf("dumping core by request\n");
4917         abort();
4918     }
4919
4920     return (ret);
4921 }
```

new/usr/src/cmd/ztest/ztest.c

1

```
*****
153311 Thu Jun 28 15:57:08 2012
new/usr/src/cmd/ztest/ztest.c
2882 implement libfs_core
2883 changing "cannount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
unchanged_portion_omitted_

2238 /*
2239  * Verify that we can't destroy an active pool, create an existing pool,
2240  * or create a pool with a bad vdev spec.
2241  */
2242 /* ARGSUSED */
2243 void
2244 ztest_spa_create_destroy(ztest_ds_t *zd, uint64_t id)
2245 {
2246     ztest_shared_opts_t *zo = &ztest_opts;
2247     spa_t *spa;
2248     nvlist_t *nvroot;

2250     /*
2251      * Attempt to create using a bad file.
2252      */
2253     nvroot = make_vdev_root("/dev/bogus", NULL, 0, 0, 0, 0, 1);
2254     VERIFY3U(ENOENT, ==,
2255             spa_create("ztest_bad_file", nvroot, NULL, NULL));
2256     spa_create("ztest_bad_file", nvroot, NULL, NULL, NULL));
2257     nvlist_free(nvroot);

2258     /*
2259      * Attempt to create using a bad mirror.
2260      */
2261     nvroot = make_vdev_root("/dev/bogus", NULL, 0, 0, 0, 0, 2, 1);
2262     VERIFY3U(ENOENT, ==,
2263             spa_create("ztest_bad_mirror", nvroot, NULL, NULL));
2264     spa_create("ztest_bad_mirror", nvroot, NULL, NULL, NULL));
2265     nvlist_free(nvroot);

2266     /*
2267      * Attempt to create an existing pool. It shouldn't matter
2268      * what's in the nvroot; we should fail with EEXIST.
2269      */
2270     (void) rw_rdlock(&ztest_name_lock);
2271     nvroot = make_vdev_root("/dev/bogus", NULL, 0, 0, 0, 0, 1);
2272     VERIFY3U(EEXIST, ==, spa_create(zo->zo_pool, nvroot, NULL, NULL));
2273     VERIFY3U(EEXIST, ==, spa_create(zo->zo_pool, nvroot, NULL, NULL, NULL));
2274     nvlist_free(nvroot);
2275     VERIFY3U(0, ==, spa_open(zo->zo_pool, &spa, FTAG));
2276     VERIFY3U(EBUSY, ==, spa_destroy(zo->zo_pool));
2277     spa_close(spa, FTAG);

2278     (void) rw_unlock(&ztest_name_lock);
2279 }
unchanged_portion_omitted_

3050 static boolean_t
3051 ztest_snapshot_create(char *osname, uint64_t id)
3052 {
3053     char snapname[MAXNAMELEN];
3054     int error;
```

new/usr/src/cmd/ztest/ztest.c

2

```
3056     (void) snprintf(snapname, MAXNAMELEN, "%s@%llu", osname,
3057                    (u_longlong_t)id);

3059     error = dmu_objset_snapshot_one(osname, strchr(snapname, '@') + 1);
3060     error = dmu_objset_snapshot(osname, strchr(snapname, '@') + 1,
3061                                NULL, NULL, B_FALSE, B_FALSE, -1);
3062     if (error == ENOSPC) {
3063         ztest_record_enospc(FTAG);
3064         return (B_FALSE);
3065     }
3066     if (error != 0 && error != EEXIST)
3067         fatal(0, "ztest_snapshot_create(%s) = %d", snapname, error);
3068     return (B_TRUE);
3069 }
unchanged_portion_omitted_

3233 /*
3234  * Verify dsl_dataset_promote handles EBUSY
3235  */
3236 void
3237 ztest_dsl_dataset_promote_busy(ztest_ds_t *zd, uint64_t id)
3238 {
3239     objset_t *clone;
3240     dsl_dataset_t *ds;
3241     char snap1name[MAXNAMELEN];
3242     char clone1name[MAXNAMELEN];
3243     char snap2name[MAXNAMELEN];
3244     char clone2name[MAXNAMELEN];
3245     char snap3name[MAXNAMELEN];
3246     char *osname = zd->zid_name;
3247     int error;

3249     (void) rw_rdlock(&ztest_name_lock);

3251     ztest_dsl_dataset_cleanup(osname, id);

3253     (void) snprintf(snap1name, MAXNAMELEN, "%s@s1_%llu", osname, id);
3254     (void) snprintf(clone1name, MAXNAMELEN, "%s/cl_%llu", osname, id);
3255     (void) snprintf(snap2name, MAXNAMELEN, "%s@s2_%llu", clone1name, id);
3256     (void) snprintf(clone2name, MAXNAMELEN, "%s/c2_%llu", osname, id);
3257     (void) snprintf(snap3name, MAXNAMELEN, "%s@s3_%llu", clone1name, id);

3259     error = dmu_objset_snapshot_one(osname, strchr(snap1name, '@')+1,
3260                                     NULL, NULL, B_FALSE, B_FALSE, -1);
3261     if (error && error != EEXIST) {
3262         if (error == ENOSPC) {
3263             ztest_record_enospc(FTAG);
3264             goto out;
3265         }
3266         fatal(0, "dmu_take_snapshot(%s) = %d", snap1name, error);

3268     error = dmu_objset_hold(snap1name, FTAG, &clone);
3269     if (error)
3270         fatal(0, "dmu_open_snapshot(%s) = %d", snap1name, error);

3272     error = dmu_objset_clone(clone1name, dmu_objset_ds(clone), 0);
3273     dmu_objset_rele(clone, FTAG);
3274     if (error) {
3275         if (error == ENOSPC) {
3276             ztest_record_enospc(FTAG);
3277             goto out;
3278         }
3279         fatal(0, "dmu_objset_create(%s) = %d", clone1name, error);
```

```

3280     }
3282     error = dmu_objset_snapshot_one(clone1name, strchr(snap2name, '@') + 1);
3284     error = dmu_objset_snapshot(clone1name, strchr(snap2name, '@')+1,
3285     NULL, NULL, B_FALSE, B_FALSE, -1);
3283     if (error && error != EEXIST) {
3284         if (error == ENOSPC) {
3285             ztest_record_enospc(FTAG);
3286             goto out;
3287         }
3288         fatal(0, "dmu_open_snapshot(%s) = %d", snap2name, error);
3289     }
3291     error = dmu_objset_snapshot_one(clone1name, strchr(snap3name, '@') + 1);
3294     error = dmu_objset_snapshot(clone1name, strchr(snap3name, '@')+1,
3295     NULL, NULL, B_FALSE, B_FALSE, -1);
3292     if (error && error != EEXIST) {
3293         if (error == ENOSPC) {
3294             ztest_record_enospc(FTAG);
3295             goto out;
3296         }
3297         fatal(0, "dmu_open_snapshot(%s) = %d", snap3name, error);
3298     }
3300     error = dmu_objset_hold(snap3name, FTAG, &clone);
3301     if (error)
3302         fatal(0, "dmu_open_snapshot(%s) = %d", snap3name, error);
3304     error = dmu_objset_clone(clone2name, dmu_objset_ds(clone), 0);
3305     dmu_objset_rele(clone, FTAG);
3306     if (error) {
3307         if (error == ENOSPC) {
3308             ztest_record_enospc(FTAG);
3309             goto out;
3310         }
3311         fatal(0, "dmu_objset_create(%s) = %d", clone2name, error);
3312     }
3314     error = dsl_dataset_own(snap2name, B_FALSE, FTAG, &ds);
3315     if (error)
3316         fatal(0, "dsl_dataset_own(%s) = %d", snap2name, error);
3317     error = dsl_dataset_promote(clone2name, NULL);
3318     if (error != EBUSY)
3319         fatal(0, "dsl_dataset_promote(%s), %d, not EBUSY", clone2name,
3320         error);
3321     dsl_dataset_disown(ds, FTAG);
3323 out:
3324     ztest_dsl_dataset_cleanup(osname, id);
3326     (void) rw_unlock(&ztest_name_lock);
3327 }
    unchanged portion omitted
4444 /*
4445  * Test snapshot hold/release and deferred destroy.
4446  */
4447 void
4448 ztest_dmu_snapshot_hold(ztest_ds_t *zd, uint64_t id)
4449 {
4450     int error;
4451     objset_t *os = zd->zd_os;
4452     objset_t *origin;
4453     char snapname[100];
4454     char fullname[100];
4455     char clonename[100];

```

```

4456     char tag[100];
4457     char osname[MAXNAMELEN];
4459     (void) rw_rdlock(&ztest_name_lock);
4461     dmu_objset_name(os, osname);
4463     (void) snprintf(snapname, 100, "sh1_%llu", id);
4464     (void) snprintf(fullname, 100, "%s%s", osname, snapname);
4465     (void) snprintf(clonename, 100, "%s/ch1_%llu", osname, id);
4466     (void) snprintf(tag, 100, "%tag_%llu", id);
4468     /*
4469     * Clean up from any previous run.
4470     */
4471     (void) dmu_objset_destroy(clonename, B_FALSE);
4472     (void) dsl_dataset_user_release(osname, snapname, tag, B_FALSE);
4473     (void) dmu_objset_destroy(fullname, B_FALSE);
4475     /*
4476     * Create snapshot, clone it, mark snap for deferred destroy,
4477     * destroy clone, verify snap was also destroyed.
4478     */
4479     error = dmu_objset_snapshot_one(osname, snapname);
4483     error = dmu_objset_snapshot(osname, snapname, NULL, NULL, FALSE,
4484     FALSE, -1);
4480     if (error) {
4481         if (error == ENOSPC) {
4482             ztest_record_enospc("dmu_objset_snapshot");
4483             goto out;
4484         }
4485         fatal(0, "dmu_objset_snapshot(%s) = %d", fullname, error);
4486     }
4488     error = dmu_objset_hold(fullname, FTAG, &origin);
4489     if (error)
4490         fatal(0, "dmu_objset_hold(%s) = %d", fullname, error);
4492     error = dmu_objset_clone(clonename, dmu_objset_ds(origin), 0);
4493     dmu_objset_rele(origin, FTAG);
4494     if (error) {
4495         if (error == ENOSPC) {
4496             ztest_record_enospc("dmu_objset_clone");
4497             goto out;
4498         }
4499         fatal(0, "dmu_objset_clone(%s) = %d", clonename, error);
4500     }
4502     error = dmu_objset_destroy(fullname, B_TRUE);
4503     if (error) {
4504         fatal(0, "dmu_objset_destroy(%s, B_TRUE) = %d",
4505         fullname, error);
4506     }
4508     error = dmu_objset_destroy(clonename, B_FALSE);
4509     if (error)
4510         fatal(0, "dmu_objset_destroy(%s) = %d", clonename, error);
4512     error = dmu_objset_hold(fullname, FTAG, &origin);
4513     if (error != ENOENT)
4514         fatal(0, "dmu_objset_hold(%s) = %d", fullname, error);
4516     /*
4517     * Create snapshot, add temporary hold, verify that we can't
4518     * destroy a held snapshot, mark for deferred destroy,
4519     * release hold, verify snapshot was destroyed.

```

```

4520      */
4521      error = dmu_objset_snapshot_one(osname, snapname);
4522      error = dmu_objset_snapshot(osname, snapname, NULL, NULL, FALSE,
4523      FALSE, -1);
4524      if (error) {
4525          if (error == ENOSPC) {
4526              ztest_record_enospc("dmu_objset_snapshot");
4527              goto out;
4528          }
4529          fatal(0, "dmu_objset_snapshot(%s) = %d", fullname, error);
4530      }
4531      error = dsl_dataset_user_hold(osname, snapname, tag, B_FALSE,
4532      B_TRUE, -1);
4533      if (error)
4534          fatal(0, "dsl_dataset_user_hold(%s)", fullname, tag);
4535      error = dmu_objset_destroy(fullname, B_FALSE);
4536      if (error != EBUSY) {
4537          fatal(0, "dmu_objset_destroy(%s, B_FALSE) = %d",
4538          fullname, error);
4539      }
4540      error = dmu_objset_destroy(fullname, B_TRUE);
4541      if (error) {
4542          fatal(0, "dmu_objset_destroy(%s, B_TRUE) = %d",
4543          fullname, error);
4544      }
4545      error = dsl_dataset_user_release(osname, snapname, tag, B_FALSE);
4546      if (error)
4547          fatal(0, "dsl_dataset_user_release(%s)", fullname, tag);
4548      VERIFY(dmu_objset_hold(fullname, FTAG, &origin) == ENOENT);
4549
4550 out:
4551      (void) rw_unlock(&ztest_name_lock);
4552 }
4553 _____unchanged_portion_omitted_____
4554
4555 /*
4556  * Create a storage pool with the given name and initial vdev size.
4557  * Then test spa_freeze() functionality.
4558  */
4559 static void
4560 ztest_init(ztest_shared_t *zs)
4561 {
4562     spa_t *spa;
4563     nvlist_t *nvroot, *props;
4564
4565     VERIFY(_mutex_init(&ztest_vdev_lock, USYNC_THREAD, NULL) == 0);
4566     VERIFY(rwlock_init(&ztest_name_lock, USYNC_THREAD, NULL) == 0);
4567
4568     kernel_init(FREAD | FWRITE);
4569
4570     /*
4571      * Create the storage pool.
4572      */
4573     (void) spa_destroy(ztest_opts.zo_pool);
4574     ztest_shared->zsvdev_next_leaf = 0;
4575     zs->zsvdev_splits = 0;
4576     zs->zsvdev_mirrors = ztest_opts.zo_mirrors;
4577     nvroot = make_vdev_root(NULL, NULL, ztest_opts.zo_vdev_size, 0,
4578     0, ztest_opts.zo_raidz, zs->zsvdev_mirrors, 1);
4579     props = make_random_props();
4580     for (int i = 0; i < SPA_FEATURES; i++) {

```

```

5604         char buf[1024];
5605         (void) snprintf(buf, sizeof (buf), "feature@%s",
5606         spa_feature_table[i].fi_uname);
5607         VERIFY3U(0, ==, nvlist_add_uint64(props, buf, 0));
5608     }
5609     VERIFY3U(0, ==, spa_create(ztest_opts.zo_pool, nvroot, props, NULL));
5610     VERIFY3U(0, ==, spa_create(ztest_opts.zo_pool, nvroot, props,
5611     NULL, NULL));
5612     nvlist_free(nvroot);
5613
5614     VERIFY3U(0, ==, spa_open(ztest_opts.zo_pool, &spa, FTAG));
5615     zs->zsvdev metaslab_sz =
5616     1ULL << spa->spa_root_vdev->vdev_child[0]->vdev_ms_shift;
5617
5618     spa_close(spa, FTAG);
5619
5620     kernel_fini();
5621
5622     ztest_run_zdb(ztest_opts.zo_pool);
5623
5624     ztest_freeze();
5625
5626     ztest_run_zdb(ztest_opts.zo_pool);
5627
5628     (void) rwlock_destroy(&ztest_name_lock);
5629     (void) _mutex_destroy(&ztest_vdev_lock);
5630 }
5631 _____unchanged_portion_omitted_____

```

```

*****
5091 Thu Jun 28 15:57:08 2012
new/usr/src/common/zfs/zfs_comutil.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */
27 /*
28 * This file is intended for functions that ought to be common between user
29 * land (libzfs) and the kernel. When many common routines need to be shared
30 * then a separate file should to be created.
31 */
33 #if defined( KERNEL)
34 #include <sys/system.h>
35 #else
36 #include <string.h>
37 #endif
39 #include <sys/types.h>
40 #include <sys/fs/zfs.h>
41 #include <sys/int_limits.h>
42 #include <sys/nvpair.h>
43 #include "zfs_comutil.h"
45 /*
46 * Are there allocatable vdevs?
47 */
48 boolean_t
49 zfs_allocatable_devs(nvlist_t *nv)
50 {
51     uint64_t is_log;
52     uint_t c;
53     nvlist_t **child;
54     uint_t children;

```

```

56     if (nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
57         &child, &children) != 0) {
58         return (B_FALSE);
59     }
60     for (c = 0; c < children; c++) {
61         is_log = 0;
62         (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_LOG,
63             &is_log);
64         if (!is_log)
65             return (B_TRUE);
66     }
67     return (B_FALSE);
68 }
70 void
71 zpool_get_rewind_policy(nvlist_t *nvl, zpool_rewind_policy_t *zrpp)
72 {
73     nvlist_t *policy;
74     nvpair_t *elem;
75     char *nm;
77     /* Defaults */
78     zrpp->zrp_request = ZPOOL_NO_REWIND;
79     zrpp->zrp_maxmeta = 0;
80     zrpp->zrp_maxdata = UINT64_MAX;
81     zrpp->zrp_tsg = UINT64_MAX;
83     if (nvl == NULL)
84         return;
86     elem = NULL;
87     while ((elem = nvlist_next_nvpair(nvl, elem)) != NULL) {
88         nm = nvpair_name(elem);
89         if (strcmp(nm, ZPOOL_REWIND_POLICY) == 0) {
90             if (nvpair_value_nvlist(elem, &policy) == 0)
91                 zpool_get_rewind_policy(policy, zrpp);
92             return;
93         } else if (strcmp(nm, ZPOOL_REWIND_REQUEST) == 0) {
94             if (nvpair_value_uint32(elem, &zrpp->zrp_request) == 0)
95                 if (zrpp->zrp_request & ~ZPOOL_REWIND_POLICIES)
96                     zrpp->zrp_request = ZPOOL_NO_REWIND;
97         } else if (strcmp(nm, ZPOOL_REWIND_REQUEST_TXG) == 0) {
98             (void) nvpair_value_uint64(elem, &zrpp->zrp_tsg);
99         } else if (strcmp(nm, ZPOOL_REWIND_META_THRESH) == 0) {
100             (void) nvpair_value_uint64(elem, &zrpp->zrp_maxmeta);
101         } else if (strcmp(nm, ZPOOL_REWIND_DATA_THRESH) == 0) {
102             (void) nvpair_value_uint64(elem, &zrpp->zrp_maxdata);
103         }
104     }
105     if (zrpp->zrp_request == 0)
106         zrpp->zrp_request = ZPOOL_NO_REWIND;
107 }
109 typedef struct zfs_version_spa_map {
110     int     version_zpl;
111     int     version_spa;
112 } zfs_version_spa_map_t;
114 /*
115 * Keep this table in monotonically increasing version number order.
116 */
117 static zfs_version_spa_map_t zfs_version_table[] = {
118     {ZPL_VERSION_INITIAL, SPA_VERSION_INITIAL},
119     {ZPL_VERSION_DIRENT_TYPE, SPA_VERSION_INITIAL},
120     {ZPL_VERSION_FUID, SPA_VERSION_FUID},

```

```

121     {ZPL_VERSION_USERSPACE, SPA_VERSION_USERSPACE},
122     {ZPL_VERSION_SA, SPA_VERSION_SA},
123     {0, 0}
124 };

126 /*
127  * Return the max zpl version for a corresponding spa version
128  * -1 is returned if no mapping exists.
129  */
130 int
131 zfs_zpl_version_map(int spa_version)
132 {
133     int i;
134     int version = -1;

136     for (i = 0; zfs_version_table[i].version_spa; i++) {
137         if (spa_version >= zfs_version_table[i].version_spa)
138             version = zfs_version_table[i].version_zpl;
139     }

141     return (version);
142 }

144 /*
145  * Return the min spa version for a corresponding spa version
146  * -1 is returned if no mapping exists.
147  */
148 int
149 zfs_spa_version_map(int zpl_version)
150 {
151     int i;
152     int version = -1;

154     for (i = 0; zfs_version_table[i].version_zpl; i++) {
155         if (zfs_version_table[i].version_zpl >= zpl_version)
156             return (zfs_version_table[i].version_spa);
157     }

159     return (version);
160 }

162 /*
163  * This is the table of legacy internal event names; it should not be modified.
164  * The internal events are now stored in the history log as strings.
165  */
166 const char *zfs_history_event_names[ZFS_NUM_LEGACY_HISTORY_EVENTS] = {
167     const char *zfs_history_event_names[LOG_END] = {
168         "invalid event",
169         "pool create",
170         "vdev add",
171         "pool remove",
172         "pool destroy",
173         "pool export",
174         "pool import",
175         "vdev attach",
176         "vdev replace",
177         "vdev detach",
178         "vdev online",
179         "vdev offline",
180         "vdev upgrade",
181         "pool clear",
182         "pool scrub",
183         "pool property set",
184         "create",
185         "clone",
186         "destroy",

```

```

186         "destroy_begin_sync",
187         "inherit",
188         "property set",
189         "quota set",
190         "permission update",
191         "permission remove",
192         "permission who remove",
193         "promote",
194         "receive",
195         "rename",
196         "reservation set",
197         "replay_inc_sync",
198         "replay_full_sync",
199         "rollback",
200         "snapshot",
201         "filesystem version upgrade",
202         "refquota set",
203         "refreservation set",
204         "pool scrub done",
205         "user hold",
206         "user release",
207         "pool split",
208     };
    unchanged_portion_omitted

```


new/usr/src/common/zfs/zfs_comutil.h

1

1496 Thu Jun 28 15:57:09 2012

new/usr/src/common/zfs/zfs_comutil.h

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */

27 #ifndef _ZFS_COMUTIL_H
28 #define _ZFS_COMUTIL_H

30 #include <sys/fs/zfs.h>
31 #include <sys/types.h>

33 #ifdef __cplusplus
34 extern "C" {
35 #endif

37 extern boolean_t zfs_allocatable_devs(nvlist_t *);
38 extern void zpool_get_rewind_policy(nvlist_t *, zpool_rewind_policy_t *);

40 extern int zfs_zpl_version_map(int spa_version);
41 extern int zfs_spa_version_map(int zpl_version);
42 #define ZFS_NUM_LEGACY_HISTORY_EVENTS 41
43 extern const char *zfs_history_event_names[ZFS_NUM_LEGACY_HISTORY_EVENTS];
44 extern const char *zfs_history_event_names[LOG_END];

45 #ifdef __cplusplus
46 }

```

unchanged_portion_omitted

```

*****
10003 Thu Jun 28 15:57:09 2012
new/usr/src/common/zfs/zprop_common.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 /*
26 * Copyright (c) 2012 by Delphix. All rights reserved.
27 */
28 #endif /* !codereview */

30 /*
31  * Common routines used by zfs and zpool property management.
32  */

34 #include <sys/zio.h>
35 #include <sys/spa.h>
36 #include <sys/zfs_acl.h>
37 #include <sys/zfs_ioctl.h>
38 #include <sys/zfs_znode.h>
39 #include <sys/fs/zfs.h>

41 #include "zfs_prop.h"
42 #include "zfs_deleg.h"

44 #if defined(_KERNEL)
45 #include <sys/system.h>
46 #include <util/qsorth.h>
47 #else
48 #include <stdlib.h>
49 #include <string.h>
50 #include <ctype.h>
51 #endif

53 static zprop_desc_t *
54 zprop_get_proptable(zfs_type_t type)

```

```

55 {
56     if (type == ZFS_TYPE_POOL)
57         return (zpool_prop_get_table());
58     else
59         return (zfs_prop_get_table());
60 }

62 static int
63 zprop_get_numprops(zfs_type_t type)
64 {
65     if (type == ZFS_TYPE_POOL)
66         return (ZPOOL_NUM_PROPS);
67     else
68         return (ZFS_NUM_PROPS);
69 }

71 void
72 zprop_register_impl(int prop, const char *name, zprop_type_t type,
73 uint64_t numdefault, const char *strdefault, zprop_attr_t attr,
74 int objset_types, const char *values, const char *colname,
75 boolean_t rightalign, boolean_t visible, const zprop_index_t *idx_tbl)
76 {
77     zprop_desc_t *prop_tbl = zprop_get_proptable(objset_types);
78     zprop_desc_t *pd;

80     pd = &prop_tbl[prop];

82     ASSERT(pd->pd_name == NULL || pd->pd_name == name);
83     ASSERT(name != NULL);
84     ASSERT(colname != NULL);

86     pd->pd_name = name;
87     pd->pd_propnum = prop;
88     pd->pd_proptype = type;
89     pd->pd_numdefault = numdefault;
90     pd->pd_strdefault = strdefault;
91     pd->pd_attr = attr;
92     pd->pd_types = objset_types;
93     pd->pd_values = values;
94     pd->pd_colname = colname;
95     pd->pd_rightalign = rightalign;
96     pd->pd_visible = visible;
97     pd->pd_table = idx_tbl;
98     pd->pd_table_size = 0;
99     while (idx_tbl && (idx_tbl++)->pi_name != NULL)
100         pd->pd_table_size++;
101 }

103 void
104 zprop_register_string(int prop, const char *name, const char *def,
105 zprop_attr_t attr, int objset_types, const char *values,
106 const char *colname)
107 {
108     zprop_register_impl(prop, name, PROP_TYPE_STRING, 0, def, attr,
109 objset_types, values, colname, B_FALSE, B_TRUE, NULL);
110 }

111 }

113 void
114 zprop_register_number(int prop, const char *name, uint64_t def,
115 zprop_attr_t attr, int objset_types, const char *values,
116 const char *colname)
117 {
118     zprop_register_impl(prop, name, PROP_TYPE_NUMBER, def, NULL, attr,
119 objset_types, values, colname, B_TRUE, B_TRUE, NULL);
120 }

```

```
122 void
123 zprop_register_index(int prop, const char *name, uint64_t def,
124     zprop_attr_t attr, int objset_types, const char *values,
125     const char *colname, const zprop_index_t *idx_tbl)
126 {
127     zprop_register_impl(prop, name, PROP_TYPE_INDEX, def, NULL, attr,
128     objset_types, values, colname, B_TRUE, B_TRUE, idx_tbl);
129 }

131 void
132 zprop_register_hidden(int prop, const char *name, zprop_type_t type,
133     zprop_attr_t attr, int objset_types, const char *colname)
134 {
135     zprop_register_impl(prop, name, type, 0, NULL, attr,
136     objset_types, NULL, colname,
137     type == PROP_TYPE_NUMBER, B_FALSE, NULL);
138     objset_types, NULL, colname, B_FALSE, B_FALSE, NULL);
138 }
unchanged_portion_omitted
```

```

*****
13721 Thu Jun 28 15:57:09 2012
new/usr/src/lib/Makefile
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
23 # Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright (c) 2012 by Delphix. All rights reserved.
25 #endif /* ! codereview */
26 #
27 include ../Makefile.master
28 #
29 # Note that libcurses installs commands along with its library.
30 # This is a minor bug which probably should be fixed.
31 # Note also that a few extra libraries are kept in cmd source.
32 #
33 # Certain libraries are linked with, hence depend on, other libraries.
34 #
35 # Although we have historically used .WAIT to express dependencies, it
36 # reduces the amount of parallelism and thus lengthens the time it
37 # takes to build the libraries. Thus, we now require that any new
38 # libraries explicitly call out their dependencies. Eventually, all
39 # the library dependencies will be called out explicitly. See
40 # "Library interdependencies" near the end of this file.
41 #
42 # Aside from explicit dependencies (and legacy .WAITs), all libraries
43 # are built in parallel.
44 #
45 .PARALLEL:
46 #
47 #
48 # The $(CLOSED_BUILD) additions to SUBDIRS & MSGSUBDIRS are unfortunate,
49 # but required due to the "dependencies" of using .WAIT to barrier the
50 # parallel dmake builds. once 4631488 has been fixed, they can be
51 # consolidated into one $(CLOSED_BUILD)SUBDIRS += (all closed libs) as
52 # shown in HDRSUBDIRS
53 #
54 SUBDIRS= \

```

```

55 common .WAIT \
56 ../cmd/sgs/libconv \
57 ../cmd/sgs/libdl .WAIT \
58 \
59 SUBDIRS += \
60 libc .WAIT \
61 ../cmd/sgs/libelf .WAIT \
62 c_synonyms \
63 libmd \
64 libmd5 \
65 librm \
66 libmp .WAIT \
67 libns1 \
68 libsecdb .WAIT \
69 librpcsvc \
70 libsocket .WAIT \
71 libsctp \
72 libsip \
73 libcommputil \
74 libresolv \
75 libresolv2 .WAIT \
76 libw .WAIT \
77 libintl .WAIT \
78 ../cmd/sgs/librtld_db \
79 libaio \
80 libast \
81 libdll \
82 libcmd \
83 libshell \
84 libsum \
85 librt \
86 libadm \
87 libctf \
88 libdtrace \
89 libdtrace_jni \
90 libcurses \
91 libtermcap \
92 libgen \
93 libgss \
94 libpam \
95 libuuid \
96 libthread \
97 libpthread .WAIT \
98 libslp \
99 libbsdmalloc \
100 libdoor \
101 libdevinfo \
102 libldadm \
103 libdlpi \
104 libeti \
105 libcrypt \
106 libdns_sd \
107 libefi \
108 libfstyp \
109 libwanboot \
110 libwanbootutil \
111 libcryptoutil \
112 libinetutil \
113 libipadm \
114 libipmp \
115 libiscsit \
116 libkmf \
117 libkstat \
118 libkvm \
119 liblm \
120 libmalloc \

```

```

121 libmapmalloc \
122 libmtmalloc \
123 libnls \
124 libnwam \
125 libsembios \
126 libtecla \
127 libumem \
128 libnvpair .WAIT \
129 libxacct \
130 libsas1 \
131 libldap5 \
132 libslldap .WAIT \
133 libbsm \
134 libsys \
135 libsysevent \
136 libnisdb \
137 libpool \
138 libpp \
139 libproc \
140 libproject \
141 libsendfile \
142 nametoaddr \
143 ncad_addr \
144 hbaapi \
145 smhba \
146 sun_fc \
147 sun_sas \
148 gss_mechs/mech_krb5 .WAIT \
149 libkrb5 .WAIT \
150 krb5 .WAIT \
151 libsmbs \
152 libfcoe \
153 libsrpt \
154 libstmf \
155 libstmfproxy \
156 libnsctl \
157 libunistat \
158 libdscfg \
159 librdc \
160 libinstzones \
161 libpkg

163 SUBDIRS += \
164 passwdutil \
165 pam_modules \
166 crypt_modules \
167 libadt_jni \
168 abi \
169 auditd_plugins \
170 libvolmgt \
171 libdevice \
172 libdevvid \
173 libdhcpsvc \
174 libc_db \
175 libndmp \
176 libsec \
177 libtnfprobe \
178 libtnf \
179 libtnfctl \
180 libdhcpagent \
181 libdhcpdu \
182 libdhcputil \
183 libxnet \
184 libipseutil
185 $(CLOSED_BUILD)SUBDIRS += \
186 $(CLOSED)/lib/libike

```

```

187 SUBDIRS += \
188 nsswitch \
189 print \
190 libuutil \
191 libscf \
192 libinetsvc \
193 librestart \
194 libsched \
195 libelfsign \
196 pkcs11 .WAIT \
197 libpctx .WAIT \
198 libcpc \
199 watchmalloc \
200 extendedFILE \
201 madv \
202 mpss \
203 libdisasm \
204 libwrap \
205 libxcurses \
206 libxcurses2 \
207 libbrand .WAIT \
208 libzonecfg \
209 libzoneinfo \
210 libzonestat \
211 libtsnet \
212 libtsol \
213 gss_mechs/mech_spnego \
214 gss_mechs/mech_dummy \
215 gss_mechs/mech_dh \
216 rpcsec_gss \
217 libraidcfg .WAIT \
218 librcm .WAIT \
219 libcfgadm .WAIT \
220 libpicl .WAIT \
221 libpicltree .WAIT \
222 raidcfg_plugins \
223 cfgadm_plugins \
224 libmail \
225 lvm \
226 libsmmedia \
227 libipp \
228 libdiskmgt \
229 liblgrp \
230 libfsmgt \
231 fm \
232 libavl \
233 libcmdutils \
234 libcontract \
235 ../cmd/sendmail/libmilter \
236 sasl_plugins \
237 udapl \
238 libzpool \
239 libzfs_core \
240 #endif /* ! codereview */
241 libzfs \
242 libbe \
243 pylibbe \
244 libzfs_jni \
245 pyzfs \
246 pysolaris \
247 libmapid \
248 brand \
249 policykit \
250 hal \
251 libshare \
252 libsqlite

```

```

253 libidmap \
254 libadutils \
255 libipmi \
256 libexacct/demo \
257 libvrrpadm \
258 libvscan \
259 libgrubmgmt \
260 smbstrv \
261 libilb \
262 scsi \
263 libima \
264 libsun_ima \
265 mpapi \
266 librstp \
267 librepase \
268 libhotplug \
269 libfruutils .WAIT \
270 libfru \
271 $(MACH)_SUBDIRS)

273 i386_SUBDIRS= \
274 libntfs \
275 libparted \
276 libfdisk

278 sparc_SUBDIRS= .WAIT \
279 efcodes \
280 libds \
281 libdscp \
282 libprtdiag .WAIT \
283 libprtdiag_psr \
284 libpri \
285 librsd \
286 storage \
287 libpcp \
288 libtsalarm \
289 libv12n

291 FM_sparc_DEPLIBS= libpri

293 fm: \
294 libexacct \
295 libipmi \
296 libzfs \
297 scsi \
298 $(FM_$(MACH)_DEPLIBS)

300 #
301 # Create a special version of $(SUBDIRS) with no .WAIT's, for use with the
302 # clean and clobber targets (for more information, see those targets, below).
303 #
304 NOWAIT_SUBDIRS= $(SUBDIRS:.WAIT=)

306 DCSSUBDIRS = \
307 lvm

309 MSGSUBDIRS= \
310 abi \
311 auditd_plugins \
312 brand \
313 cfsadm_plugins \
314 gss_mechs/mech_dh \
315 gss_mechs/mech_krb5 \
316 krb5 \
317 libast \
318 libbsm \

```

```

319 libc \
320 libcfgadm \
321 libcmd \
322 libcontract \
323 libcurses \
324 libdhcpcsvc \
325 libdhcputil \
326 libipsecutil \
327 libdiskmgmt \
328 libdladm \
329 libdll \
330 libgrubmgmt \
331 libgss \
332 libidmap \
333 libipmp \
334 libilb \
335 libinetutil \
336 libinstzones \
337 libipadm \
338 libnsd \
339 libnwam \
340 libpam \
341 libpicl \
342 libpool \
343 libpkg \
344 libppp \
345 libscf \
346 libsas1 \
347 libldap5 \
348 libsecdb \
349 libshare \
350 libshell \
351 libslldap \
352 libslp \
353 libsmbsfs \
354 libsmmedia \
355 libsum \
356 libtsol \
357 libuutil \
358 libvrrpadm \
359 libvscan \
360 libwanboot \
361 libwanbootutil \
362 libzfs \
363 libzonecfg \
364 lvm \
365 madv \
366 mpss \
367 pam_modules \
368 pyzfs \
369 pysolaris \
370 rpcsec_gss \
371 libreparse

372 MSGSUBDIRS += \
373 $(MACH)_MSGSUBDIRS)

375 sparc_MSGSUBDIRS= \
376 libprtdiag \
377 libprtdiag_psr

379 i386_MSGSUBDIRS= libfdisk

381 HDRSUBDIRS= \
382 auditd_plugins \
383 libast \
384 libbrand \

```

```

385 libbsm \
386 libc \
387 libcmd \
388 libcmdutils \
389 libcommputil \
390 libcontract \
391 libcpc \
392 libctf \
393 libcurses \
394 libtermcap \
395 libcryptoutil \
396 libdevice \
397 libdevic \
398 libdevinfo \
399 libdiskmgt \
400 libdladm \
401 libdll \
402 libdlpi \
403 libdhcpagent \
404 libdhcpsvc \
405 libdhcputil \
406 libdisasm \
407 libdns_sd \
408 libdscfg \
409 libdtrace \
410 libdtrace_jni \
411 libelfsign \
412 libeti \
413 libfru \
414 libfstyp \
415 libgen \
416 libipadm \
417 libipsecutil \
418 libinetsvc \
419 libinetutil \
420 libinstzones \
421 libipmi \
422 libipmp \
423 libipp \
424 libiscsit \
425 libkstat \
426 libkvm \
427 libmail \
428 libmd \
429 libmtmalloc \
430 libndmp \
431 libnvpair \
432 libnsctl \
433 libnsl \
434 libnwam \
435 libpam \
436 libpctx \
437 libpicl \
438 libpicltree \
439 libpool \
440 libpp \
441 libproc \
442 libraidcfg \
443 librcm \
444 librdc \
445 libscf \
446 libsip \
447 libsbios \
448 librestart \
449 librpcsvc \
450 librsn \

```

```

451 librstp \
452 libsas1 \
453 libsec \
454 libshell \
455 libslp \
456 libsmmedia \
457 libsocket \
458 libsqlite \
459 libfcoe \
460 libsrpt \
461 libstmf \
462 libstmfproxy \
463 libsum \
464 libsysevent \
465 libtecla \
466 libtnf \
467 libtnfctl \
468 libtnfprobe \
469 libtsnet \
470 libtsol \
471 libvrrpadm \
472 libvolmgt \
473 libumem \
474 libunistat \
475 libuutil \
476 libwanboot \
477 libwanbootutil \
478 libwrap \
479 libxcurses2 \
480 libzfs \
481 libzfs_core \
482 #endif /* ! codereview */
483 libzfs_jni \
484 libzoneinfo \
485 libzonestat \
486 hal \
487 policykit \
488 lvm \
489 pkcs11 \
490 passwdutil \
491 ../cmd/sendmail/libmilter \
492 fm \
493 udapl \
494 libmapid \
495 libkrb5 \
496 libsbmfs \
497 libshare \
498 libidmap \
499 libvscan \
500 libgrubmgmt \
501 smbsrv \
502 libilb \
503 scsi \
504 hbaapi \
505 smhba \
506 libima \
507 libsun_ima \
508 mpapi \
509 librepase \
510 $(MACH)_HDRSUBDIRS)

512 $(CLOSED_BUILD)HDRSUBDIRS += \
513 $(CLOSED)/lib/libike

515 i386_HDRSUBDIRS= \
516 libparted \

```

```

517 libfdisk
519 sparc_HDRSUBDIRS= \
520 libds \
521 libdscp \
522 libpri \
523 libv12n \
524 storage

526 all := TARGET= all
527 check := TARGET= check
528 clean := TARGET= clean
529 clobber := TARGET= clobber
530 install := TARGET= install
531 install_h := TARGET= install_h
532 lint := TARGET= lint
533 _dc := TARGET= _dc
534 _msg := TARGET= _msg

536 .KEEP_STATE:

538 #
539 # For the all and install targets, we clearly must respect library
540 # dependencies so that the libraries link correctly. However, for
541 # the remaining targets (check, clean, clobber, install_h, lint, _dc
542 # and _msg), libraries do not have any dependencies on one another
543 # and thus respecting dependencies just slows down the build.
544 # As such, for these rules, we use pattern replacement to explicitly
545 # avoid triggering the dependency information. Note that for clean,
546 # clobber and lint, we must use $(NOWAIT_SUBDIRS) rather than
547 # $(SUBDIRS), to prevent '.WAIT' from expanding to '.WAIT-nodepend'.
548 #

550 all: $(SUBDIRS)

552 install: $(SUBDIRS) .WAIT install_extra

554 # extra libraries kept in other source areas
555 install_extra:
556 @cd ../cmd/sgs; pwd; $(MAKE) install_lib
557 @pwd

559 clean clobber lint: $(NOWAIT_SUBDIRS:%=-nodepend)

561 install_h check: $(HDRSUBDIRS:%=-nodepend)

563 _msg: $(MSGSUBDIRS:%=-nodepend) .WAIT _dc

565 _dc: $(DCSUBDIRS:%=-nodepend)

567 #
568 # Library interdependencies are called out explicitly here
569 #
570 auditd_plugins: libbsm libns1 libsecdb
571 gss_mechs/mech_krb5: libgss libns1 libsocket libresolv pkcs11
572 libadt_jni: libbsm
573 libast: libsocket
574 libadutils: libldap5 libresolv libsocket libns1
575 nsswitch: libadutils libidmap
576 libbe: libzfs
577 libbsm: libtsol
578 libcmd: libsum libast libsocket libns1
579 libcmdutils: libavl
580 libcontract: libnvpair
581 libdevid: libdevinfo
582 libdevinfo: libnvpair libsec

```

```

583 libdhcpageant: libsocket libdhcputil libuuid libdlpi libcontract
584 libdhcpcsvc: libinetutil
585 libdhcputil: libns1 libgen libinetutil libdlpi
586 libdladm: libdevinfo libinetutil libsocket libscf librcm libnvpair \
587 libexacct libns1 libkstat libcurses
588 libdll: libast
589 libdlpi: libinetutil libdladm
590 libds: libsysevent
591 libdscfg: libnsctl libumistat libsocket libns1
592 libdtrace: libproc libgen libctf
593 libdtrace_jni: libuutil libdtrace
594 libefi: libuuid
595 libfstyp: libnvpair
596 libelfsign: libcryptoutil libkrmf
597 libidmap: libadutils libldap5 libavl libsldap libuutil
598 libipadm: libns1 libinetutil libsocket libdlpi libnvpair libdhcpageant \
599 libdladm libsecdb
600 libiscsit: libc libnvpair libstmf libuuid libns1
601 libkrmf: libcryptoutil pkcs11
602 libns1: libmd5
603 libmapid: libresolv
604 librdc: libsocket libns1 libnsctl libumistat libdscfg
605 libuuid: libdlpi
606 $(CLOSED_BUILD)libike: libipsecutil libxnet libcryptoutil
607 libinetutil: libsocket
608 libipsecutil: libtecla libsocket
609 libinstzones: libzonecfg libcontract
610 libpkg: libwanboot libscf libadm
611 libnwam: libscf
612 libsecdb: libns1
613 libsasl: libgss libsocket pkcs11 libmd
614 sasl_plugins: pkcs11 libgss libsocket libsasl
615 libscfp: libsocket
616 libshell: libast libcmd libdll libsocket libsecdb
617 libsip: libmd5
618 libsmbsfs: libcmdutils libsocket libns1 libkrb5
619 libsocket: libns1
620 libstmfproxy: libstmf libsocket libns1 libpthread
621 libsum: libast
622 libsysevent: libsecdb
623 libldap5: libsasl libsocket libns1 libmd
624 libsldap: libldap5 libtsol libns1 libc libscf libresolv
625 libpool: libnvpair libexacct
626 libpp: libast
627 libzonecfg: libc libsocket libns1 libuuid libnvpair libsysevent libsec \
628 libbrand libpool libscf
629 libproc: ../cmd/sgs/librtld_db ../cmd/sgs/libelf libctf
630 libproject: libpool libproc libsecdb
631 libtermcap: libcurses
632 libtsnet: libns1 libtsol libsecdb
633 libwrap: libns1 libsocket
634 libwanboot: libnvpair libresolv libns1 libsocket libdevinfo libinetutil \
635 libdhcputil
636 libwanbootutil: libns1
637 pam_modules: libproject passwdutil smbsrv
638 libscf: libuutil libmd libgen libsmbios libns1
639 libinetsvc: libscf
640 librestart: libuutil libscf
641 ../cmd/sgs/libdl: ../cmd/sgs/libconv
642 ../cmd/sgs/libelf: ../cmd/sgs/libconv
643 pkcs11: libcryptoutil
644 print: libldap5
645 udapl/udapl_tavor: udapl/libdat
646 libzfs: libdevid libgen libnvpair libuutil \
647 libadm libavl libefi libidmap libmd libzfs_core
648 libzfs_core: libnvpair

```



```
24          libadm libavl libefi libidmap libmd
649 libzfs_jni:    libdiskmgt libnvpair libzfs
650 libzpool:     libavl libumem libnvpair libcmdutils
651 libsec:       libavl libidmap
652 brand:        libc libsocket
653 libshare:     libscf libzfs libuuid libfsmgt libsecdb libumem libsmbfs
654 libexacct/demo: libexacct libproject libsocket libnsl
655 libtsalarm:   libpcp
656 smbsrv:       libsocket libnsl libmd libxnet libpthread librt \
657               libshare libidmap pkcs11 libsqlite libcryptoutil \
658               librepase libcmdutils
659 libv12n:      libds libuuid
660 libvrrpadm:   libsocket libdladm libscf
661 libvscan:     libscf
662 libfru:       libfruutils
663 scsi:         libnvpair libfru
664 mpapi:        libpthread libdevinfo libsysevent libnvpair
665 sun_fc:       libdevinfo libsysevent libnvpair
666 libsun_ima:   libdevinfo libsysevent libnsl
667 sun_sas:      libdevinfo libsysevent libnvpair libkstat libdevid
668 libgrubmgmt: libdevinfo libzfs libfstyp
669 pylibbe:      libbe libzfs
670 pyzfs:        libnvpair libzfs
671 pysolaris:   libsec libidmap
672 libreparse:   libnvpair
673 libhotplug:   libnvpair
674 cfgadm_plugins: libhotplug
675 libilb:       libsocket
676 $(INTEL_BUILD)libdiskmgt:libfdisk

678 #
679 # The reason this rule checks for the existence of the
680 # Makefile is that some of the directories do not exist
681 # in certain situations (e.g., exportable source builds,
682 # OpenSolaris).
683 #
684 $(SUBDIRS): FRC
685     @if [ -f $@/Makefile ]; then \
686         cd $@; pwd; $(MAKE) $(TARGET); \
687     else \
688         true; \
689     fi

691 $(SUBDIRS:%=%-nodepend):
692     @if [ -f $@:%-nodepend=%/Makefile ]; then \
693         cd $@:%-nodepend=%; pwd; $(MAKE) $(TARGET); \
694     else \
695         true; \
696     fi

698 FRC:
```

new/usr/src/lib/libzfs/Makefile.com

1

```
*****
2178 Thu Jun 28 15:57:09 2012
new/usr/src/lib/libzfs/Makefile.com
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 # Copyright (c) 2012 by Delphix. All rights reserved.
24 #
25 #
26 LIBRARY= libzfs.a
27 VERS= .1
28 #
29 OBJS_SHARED= \
30     zfeature_common.o \
31     zfs_comutil.o \
32     zfs_deleg.o \
33     zfs_fletcher.o \
34     zfs_namecheck.o \
35     zfs_prop.o \
36     zpool_prop.o \
37     zprop_common.o
38 #
39 OBJS_COMMON= \
40     libzfs_changelist.o \
41     libzfs_config.o \
42     libzfs_dataset.o \
43     libzfs_diff.o \
44     libzfs_fru.o \
45     libzfs_import.o \
46     libzfs_iter.o \
47     libzfs_mount.o \
48     libzfs_pool.o \
49     libzfs_sendrecv.o \
50     libzfs_status.o \
51     libzfs_util.o
52 #
53 OBJECTS= $(OBJS_COMMON) $(OBJS_SHARED)
```

new/usr/src/lib/libzfs/Makefile.com

2

```
55 include ../../Makefile.lib
56 #
57 # libzfs must be installed in the root filesystem for mount(1M)
58 include ../../Makefile.rootfs
59 #
60 LIBS= $(DYNLIB) $(LINTLIB)
61 #
62 SRCDIR = ../../common
63 #
64 INCS += -I$(SRCDIR)
65 INCS += -I../../uts/common/fs/zfs
66 INCS += -I../../common/zfs
67 INCS += -I../../libc/inc
68 #
69 C99MODE= -xc99=%all
70 C99LMODE= -Xc99=%all
71 LDLIBS += -lc -lm -ldevd -lgen -lnvpair -luutil -lavl -lefi \
72     -ladm -lidmap -ltsol -lmd -lumem -lzfs_core
73 CPPFLAGS += $(INCS) -D_LARGEFILE64_SOURCE=1 -D_REENTRANT
74 #
75 SRCS= $(OBJS_COMMON:%.o=$(SRCDIR)/%.c) \
76     $(OBJS_SHARED:%.o=$(SRC)/common/zfs/%.c)
77 $(LINTLIB) := SRCS= $(SRCDIR)/$(LINTSRC)
78 #
79 .KEEP_STATE:
80 #
81 all: $(LIBS)
82 #
83 lint: lintcheck
84 #
85 pics/%.o: ../../common/zfs/%.c
86     $(COMPILE.c) -o $@ $<
87     $(POST_PROCESS_O)
88 #
89 include ../../Makefile.targ
```

```

*****
26825 Thu Jun 28 15:57:10 2012
new/usr/src/lib/libzfs/common/libzfs.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25  * Copyright (c) 2012 by Delphix. All rights reserved.
26  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
27 */
28
29 #ifndef _LIBZFS_H
30 #define _LIBZFS_H
31
32 #include <assert.h>
33 #include <libnvpair.h>
34 #include <sys/mnttab.h>
35 #include <sys/param.h>
36 #include <sys/types.h>
37 #include <sys/varargs.h>
38 #include <sys/fs/zfs.h>
39 #include <sys/avl.h>
40 #include <sys/ucred.h>
41
42 #ifdef __cplusplus
43 extern "C" {
44 #endif
45
46 /*
47  * Miscellaneous ZFS constants
48  */
49 #define ZFS_MAXNAMELEN      MAXNAMELEN
50 #define ZPOOL_MAXNAMELEN   MAXNAMELEN
51 #define ZFS_MAXPROPLEN     MAXPATHLEN
52 #define ZPOOL_MAXPROPLEN  MAXPATHLEN
53
54 /*

```

```

55 * libzfs errors
56 */
57 typedef enum zfs_error {
58     EZFS_SUCCESS = 0, /* no error -- success */
59     enum {
60         EZFS_NOMEM = 2000, /* out of memory */
61         EZFS_BADPROP, /* invalid property value */
62         EZFS_PROPREADONLY, /* cannot set readonly property */
63         EZFS_PROPTYPE, /* property does not apply to dataset type */
64         EZFS_PROPNONINHERIT, /* property is not inheritable */
65         EZFS_PROPSPACE, /* bad quota or reservation */
66         EZFS_BADTYPE, /* dataset is not of appropriate type */
67         EZFS_BUSY, /* pool or dataset is busy */
68         EZFS_EXISTS, /* pool or dataset already exists */
69         EZFS_NOENT, /* no such pool or dataset */
70         EZFS_BADSTREAM, /* bad backup stream */
71         EZFS_DSREADONLY, /* dataset is readonly */
72         EZFS_VOLTOOBIG, /* volume is too large for 32-bit system */
73         EZFS_INVALIDNAME, /* invalid dataset name */
74         EZFS_BADRESTORE, /* unable to restore to destination */
75         EZFS_BADBACKUP, /* backup failed */
76         EZFS_BADTARGET, /* bad attach/detach/replace target */
77         EZFS_NODEVICE, /* no such device in pool */
78         EZFS_BADDEV, /* invalid device to add */
79         EZFS_NOREPLICAS, /* no valid replicas */
80         EZFS_RESILVERING, /* currently resilvering */
81         EZFS_BADVERSION, /* unsupported version */
82         EZFS_POOLUNAVAIL, /* pool is currently unavailable */
83         EZFS_DEVOVERFLOW, /* too many devices in one vdev */
84         EZFS_BADPATH, /* must be an absolute path */
85         EZFS_CROSSTARGET, /* rename or clone across pool or dataset */
86         EZFS_ZONED, /* used improperly in local zone */
87         EZFS_MOUNTFAILED, /* failed to mount dataset */
88         EZFS_UMOUNTFAILED, /* failed to unmount dataset */
89         EZFS_UNSHARENFSAILED, /* unshare(1M) failed */
90         EZFS_SHARENFSAILED, /* share(1M) failed */
91         EZFS_PERM, /* permission denied */
92         EZFS_NOSPC, /* out of space */
93         EZFS_FAULT, /* bad address */
94         EZFS_IO, /* I/O error */
95         EZFS_INTR, /* signal received */
96         EZFS_ISSPARE, /* device is a hot spare */
97         EZFS_INVALIDCONFIG, /* invalid vdev configuration */
98         EZFS_RECURSIVE, /* recursive dependency */
99         EZFS_NOHISTORY, /* no history object */
100        EZFS_POOLPROPS, /* couldn't retrieve pool props */
101        EZFS_POOL_NOTSUP, /* ops not supported for this type of pool */
102        EZFS_POOL_INVALIDARG, /* invalid argument for this pool operation */
103        EZFS_NAMETOOLONG, /* dataset name is too long */
104        EZFS_OPENFAILED, /* open of device failed */
105        EZFS_NOCAP, /* couldn't get capacity */
106        EZFS_LABELFAILED, /* write of label failed */
107        EZFS_BADWHO, /* invalid permission who */
108        EZFS_BADPERM, /* invalid permission */
109        EZFS_BADPERMSET, /* invalid permission set name */
110        EZFS_NODELEGATION, /* delegated administration is disabled */
111        EZFS_UNSHARESMBFAILED, /* failed to unshare over smb */
112        EZFS_SHARESMBFAILED, /* failed to share over smb */
113        EZFS_BADCACHE, /* bad cache file */
114        EZFS_ISL2CACHE, /* device is for the level 2 ARC */
115        EZFS_VDEVNOTSUP, /* unsupported vdev type */
116        EZFS_NOTSUP, /* ops not supported on this dataset */
117        EZFS_ACTIVE_SPARE, /* pool has active shared spare devices */
118        EZFS_UNPLAYED_LOGS, /* log device has unplayed logs */
119        EZFS_REFTAG_RELE, /* snapshot release: tag not found */
120        EZFS_REFTAG_HOLD, /* snapshot hold: tag already exists */

```

```

120     EZFS_TAGTOOLONG,          /* snapshot hold/rele: tag too long */
121     EZFS_PIPEFAILED,         /* pipe create failed */
122     EZFS_THREADCREATEFAILED, /* thread create failed */
123     EZFS_POSTSPLIT_ONLINE,   /* onlining a disk after splitting it */
124     EZFS_SCRUBBING,          /* currently scrubbing */
125     EZFS_NO_SCRUB,           /* no active scrub */
126     EZFS_DIFF,               /* general failure of zfs diff */
127     EZFS_DIFFDATA,           /* bad zfs diff data */
128     EZFS_POOLREADONLY,      /* pool is in read-only mode */
129     EZFS_UNKNOWN
130 } zfs_error_t;
129 };

132 /*
133  * The following data structures are all part
134  * of the zfs_allow_t data structure which is
135  * used for printing 'allow' permissions.
136  * It is a linked list of zfs_allow_t's which
137  * then contain avl tree's for user/group/sets/...
138  * and each one of the entries in those trees have
139  * avl tree's for the permissions they belong to and
140  * whether they are local,descendent or local+descendent
141  * permissions. The AVL trees are used primarily for
142  * sorting purposes, but also so that we can quickly find
143  * a given user and or permission.
144  */
145 typedef struct zfs_perm_node {
146     avl_node_t z_node;
147     char z_pname[MAXPATHLEN];
148 } zfs_perm_node_t;
149 unchanged_portion_omitted

168 /*
169  * Basic handle types
170  */
171 typedef struct zfs_handle zfs_handle_t;
172 typedef struct zpool_handle zpool_handle_t;
173 typedef struct libzfs_handle libzfs_handle_t;

175 /*
176  * Library initialization
177  */
178 extern libzfs_handle_t *libzfs_init(void);
179 extern void libzfs_fini(libzfs_handle_t *);

181 extern libzfs_handle_t *zpool_get_handle(zpool_handle_t *);
182 extern libzfs_handle_t *zfs_get_handle(zfs_handle_t *);

184 extern void libzfs_print_on_error(libzfs_handle_t *, boolean_t);

186 extern void zfs_save_arguments(int argc, char **, char *, int);
187 extern int zpool_log_history(libzfs_handle_t *, const char *);

189 #endif /* ! codereview */
190 extern int libzfs_errno(libzfs_handle_t *);
191 extern const char *libzfs_error_action(libzfs_handle_t *);
192 extern const char *libzfs_error_description(libzfs_handle_t *);
193 extern void libzfs_mnttab_init(libzfs_handle_t *);
194 extern void libzfs_mnttab_fini(libzfs_handle_t *);
195 extern void libzfs_mnttab_cache(libzfs_handle_t *, boolean_t);
196 extern int libzfs_mnttab_find(libzfs_handle_t *, const char *,
197     struct mnttab *);
198 extern void libzfs_mnttab_add(libzfs_handle_t *, const char *,
199     const char *, const char *);
200 extern void libzfs_mnttab_remove(libzfs_handle_t *, const char *);

```

```

202 /*
203  * Basic handle functions
204  */
205 extern zpool_handle_t *zpool_open(libzfs_handle_t *, const char *);
206 extern zpool_handle_t *zpool_open_canfail(libzfs_handle_t *, const char *);
207 extern void zpool_close(zpool_handle_t *);
208 extern const char *zpool_get_name(zpool_handle_t *);
209 extern int zpool_get_state(zpool_handle_t *);
210 extern char *zpool_state_to_name(vdev_state_t, vdev_aux_t);
211 extern void zpool_free_handles(libzfs_handle_t *);

213 /*
214  * Iterate over all active pools in the system.
215  */
216 typedef int (*zpool_iter_f)(zpool_handle_t *, void *);
217 extern int zpool_iter(libzfs_handle_t *, zpool_iter_f, void *);

219 /*
220  * Functions to create and destroy pools
221  */
222 extern int zpool_create(libzfs_handle_t *, const char *, nvlist_t *,
223     nvlist_t *, nvlist_t *);
224 extern int zpool_destroy(zpool_handle_t *, const char *);
225 extern int zpool_add(zpool_handle_t *, nvlist_t *);

227 typedef struct splitflags {
228     /* do not split, but return the config that would be split off */
229     int dryrun : 1;

231     /* after splitting, import the pool */
232     int import : 1;
233 } splitflags_t;
234 unchanged_portion_omitted

331 extern zpool_status_t zpool_get_status(zpool_handle_t *, char **);
332 extern zpool_status_t zpool_import_status(nvlist_t *, char **);
333 extern void zpool_dump_ddt(const ddt_stat_t *dds, const ddt_histogram_t *ddh);

335 /*
336  * Statistics and configuration functions.
337  */
338 extern nvlist_t *zpool_get_config(zpool_handle_t *, nvlist_t **);
339 extern nvlist_t *zpool_get_features(zpool_handle_t *);
340 extern int zpool_refresh_stats(zpool_handle_t *, boolean_t *);
341 extern int zpool_get_errlog(zpool_handle_t *, nvlist_t **);

343 /*
344  * Import and export functions
345  */
346 extern int zpool_export(zpool_handle_t *, boolean_t, const char *);
347 extern int zpool_export_force(zpool_handle_t *, const char *);
348 extern int zpool_export_force(zpool_handle_t *);
349 extern int zpool_import(libzfs_handle_t *, nvlist_t *, const char *,
350     char *altroot);
351 extern int zpool_import_props(libzfs_handle_t *, nvlist_t *, const char *,
352     nvlist_t *, int);
353 extern void zpool_print_unsup_feat(nvlist_t *config);

354 /*
355  * Search for pools to import
356  */

358 typedef struct importargs {
359     char **path;          /* a list of paths to search */

```

```

360     int paths;          /* number of paths to search */
361     char *poolname;     /* name of a pool to find */
362     uint64_t guid;      /* guid of a pool to find */
363     char *cachefile;    /* cachefile to use for import */
364     int can_be_active : 1; /* can the pool be active? */
365     int unique : 1;     /* does 'poolname' already exist? */
366     int exists : 1;     /* set on return if pool already exists */
367 } importargs_t;

369 extern nvlist_t *zpool_search_import(libzfs_handle_t *, importargs_t *);

371 /* legacy pool search routines */
372 extern nvlist_t *zpool_find_import(libzfs_handle_t *, int, char **);
373 extern nvlist_t *zpool_find_import_cached(libzfs_handle_t *, const char *,
374     char *, uint64_t);

376 /*
377  * Miscellaneous pool functions
378  */
379 struct zfs_cmd;

381 extern const char *zfs_history_event_names[];
382 extern const char *zfs_history_event_names[LOG_END];

383 extern char *zpool_vdev_name(libzfs_handle_t *, zpool_handle_t *, nvlist_t *,
384     boolean_t verbose);
385 extern int zpool_upgrade(zpool_handle_t *, uint64_t);
386 extern int zpool_get_history(zpool_handle_t *, nvlist_t **);
387 extern int zpool_history_unpack(char *, uint64_t, uint64_t *,
388     nvlist_t ***, uint_t *);
389 extern void zpool_set_history_str(const char *subcommand, int argc,
390     char **argv, char *history_str);
391 extern int zpool_stage_history(libzfs_handle_t *, const char *);
392 extern void zpool_obj_to_path(zpool_handle_t *, uint64_t, uint64_t, char *,
393     size_t len);
394 extern int zfs_ioctl(libzfs_handle_t *, int, struct zfs_cmd *);
395 extern int zpool_get_physpath(zpool_handle_t *, char *, size_t);
396 extern void zpool_explain_recover(libzfs_handle_t *, const char *, int,
397     nvlist_t *);

399 /*
400  * Basic handle manipulations.  These functions do not create or destroy the
401  * underlying datasets, only the references to them.
402  */
403 extern zfs_handle_t *zfs_open(libzfs_handle_t *, const char *, int);
404 extern zfs_handle_t *zfs_handle_dup(zfs_handle_t *);
405 extern void zfs_close(zfs_handle_t *);
406 extern int zfs_get_type(const zfs_handle_t *);
407 extern const char *zfs_get_name(const zfs_handle_t *);
408 extern zpool_handle_t *zfs_get_pool_handle(const zfs_handle_t *);

410 /*
411  * Property management functions.  Some functions are shared with the kernel,
412  * and are found in sys/fs/zfs.h.
413  */

414 /*
415  * zfs dataset property management
416  */
417 extern const char *zfs_prop_default_string(zfs_prop_t);
418 extern uint64_t zfs_prop_default_numeric(zfs_prop_t);
419 extern const char *zfs_prop_column_name(zfs_prop_t);
420 extern boolean_t zfs_prop_align_right(zfs_prop_t);

421 extern nvlist_t *zfs_valid_proplist(libzfs_handle_t *, zfs_type_t,
422     nvlist_t *, uint64_t, zfs_handle_t *, const char *);

```

```

423 extern const char *zfs_prop_to_name(zfs_prop_t);
424 extern int zfs_prop_set(zfs_handle_t *, const char *, const char *);
425 extern int zfs_prop_get(zfs_handle_t *, zfs_prop_t, char *, size_t,
426     zprop_source_t *, char *, size_t, boolean_t);
427 extern int zfs_prop_get_recvd(zfs_handle_t *, const char *, char *, size_t,
428     boolean_t);
429 extern int zfs_prop_get_numeric(zfs_handle_t *, zfs_prop_t, uint64_t *,
430     zprop_source_t *, char *, size_t);
431 extern int zfs_prop_get_userquota_int(zfs_handle_t *zhp, const char *propname,
432     uint64_t *propvalue);
433 extern int zfs_prop_get_userquota(zfs_handle_t *zhp, const char *propname,
434     char *propbuf, int proplen, boolean_t literal);
435 extern int zfs_prop_get_written_int(zfs_handle_t *zhp, const char *propname,
436     uint64_t *propvalue);
437 extern int zfs_prop_get_written(zfs_handle_t *zhp, const char *propname,
438     char *propbuf, int proplen, boolean_t literal);
439 extern int zfs_prop_get_feature(zfs_handle_t *zhp, const char *propname,
440     char *buf, size_t len);
441 extern int zfs_get_snapused_int(zfs_handle_t *firstsnap, zfs_handle_t *lastsnap,
442     uint64_t *usedp);
443 extern uint64_t zfs_prop_get_int(zfs_handle_t *, zfs_prop_t);
444 extern int zfs_prop_inherit(zfs_handle_t *, const char *, boolean_t);
445 extern const char *zfs_prop_values(zfs_prop_t);
446 extern int zfs_prop_is_string(zfs_prop_t prop);
447 extern nvlist_t *zfs_get_user_props(zfs_handle_t *);
448 extern nvlist_t *zfs_get_recvd_props(zfs_handle_t *);
449 extern nvlist_t *zfs_get_clones_nvl(zfs_handle_t *);

450 typedef struct zprop_list {
451     int         pl_prop;
452     char        *pl_user_prop;
453     struct zprop_list *pl_next;
454     boolean_t   pl_all;
455     size_t      pl_width;
456     size_t      pl_recvd_width;
457     boolean_t   pl_fixed;
458 } zprop_list_t;

459 #ifndef zfs_prop_list_t
460 #define zprop_list_t zfs_prop_list_t
461 #endif

462 void libzfs_add_handle(get_all_cb_t *, zfs_handle_t *);
463 int libzfs_dataset_cmp(const void *, const void *);

464 /*
465  * Functions to create and destroy datasets.
466  */
467 extern int zfs_create(libzfs_handle_t *, const char *, zfs_type_t,
468     nvlist_t *);
469 extern int zfs_create_ancestors(libzfs_handle_t *, const char *);
470 extern int zfs_destroy(zfs_handle_t *, boolean_t);
471 extern int zfs_destroy_snaps(zfs_handle_t *, char *, boolean_t);
472 extern int zfs_destroy_snaps_nvl(zfs_handle_t *, nvlist_t *, boolean_t);
473 extern int zfs_clone(zfs_handle_t *, const char *, nvlist_t *);
474 extern int zfs_snapshot(libzfs_handle_t *, const char *, boolean_t, nvlist_t *);
475 extern int zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps,
476     nvlist_t *props);
477 #endif /* ! codereview */
478 extern int zfs_rollback(zfs_handle_t *, zfs_handle_t *, boolean_t);
479 extern int zfs_rename(zfs_handle_t *, const char *, boolean_t, boolean_t);

480 typedef struct sendflags {
481     /* print informational messages (ie, -v was specified) */
482     boolean_t verbose;
483 } sendflags_t;

484 /* recursive send (ie, -R) */

```

```

567     boolean_t replicate;

569     /* for incrementals, do all intermediate snapshots */
570     boolean_t doall;

572     /* if dataset is a clone, do incremental from its origin */
573     boolean_t fromorigin;

575     /* do deduplication */
576     boolean_t dedup;

578     /* send properties (ie, -p) */
579     boolean_t props;

581     /* do not send (no-op, ie. -n) */
582     boolean_t dryrun;

584     /* parsable verbose output (ie. -P) */
585     boolean_t parsable;

587     /* show progress (ie. -v) */
588     boolean_t progress;
589 } sendflags_t;

591 typedef boolean_t (snapfilter_cb_t)(zfs_handle_t *, void *);

593 extern int zfs_send(zfs_handle_t *, const char *, const char *,
594     sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);

596 extern int zfs_promote(zfs_handle_t *);
597 extern int zfs_hold(zfs_handle_t *, const char *, const char *, boolean_t,
598     boolean_t, boolean_t, int, uint64_t, uint64_t);
599 extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
600 extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
601 extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

603 typedef int (*zfs_userspace_cb_t)(void *arg, const char *domain,
604     uid_t rid, uint64_t space);

606 extern int zfs_userspace(zfs_handle_t *, zfs_userquota_prop_t,
607     zfs_userspace_cb_t, void *);

609 extern int zfs_get_fsacl(zfs_handle_t *, nvlist_t **);
610 extern int zfs_set_fsacl(zfs_handle_t *, boolean_t, nvlist_t *);

612 typedef struct recvflags {
613     /* print informational messages (ie, -v was specified) */
614     boolean_t verbose;

616     /* the destination is a prefix, not the exact fs (ie, -d) */
617     boolean_t isprefix;

619     /*
620      * Only the tail of the sent snapshot path is appended to the
621      * destination to determine the received snapshot name (ie, -e).
622      */
623     boolean_t istail;

625     /* do not actually do the recv, just check if it would work (ie, -n) */
626     boolean_t dryrun;

628     /* rollback/destroy filesystems as necessary (eg, -F) */
629     boolean_t force;

631     /* set "canmount=off" on all modified filesystems */
632     boolean_t canmountoff;

```

```

634     /* byteswap flag is used internally; callers need not specify */
635     boolean_t byteswap;

637     /* do not mount file systems as they are extracted (private) */
638     boolean_t nomount;
639 } recvflags_t;

641 extern int zfs_receive(libzfs_handle_t *, const char *, recvflags_t *,
642     int, avl_tree_t *);

644 typedef enum diff_flags {
645     ZFS_DIFF_PARSEABLE = 0x1,
646     ZFS_DIFF_TIMESTAMP = 0x2,
647     ZFS_DIFF_CLASSIFY = 0x4
648 } diff_flags_t;

650 extern int zfs_show_diffs(zfs_handle_t *, int, const char *, const char *,
651     int);

653 /*
654  * Miscellaneous functions.
655  */
656 extern const char *zfs_type_to_name(zfs_type_t);
657 extern void zfs_refresh_properties(zfs_handle_t *);
658 extern int zfs_name_valid(const char *, zfs_type_t);
659 extern zfs_handle_t *zfs_path_to_zhandle(libzfs_handle_t *, char *, zfs_type_t);
660 extern boolean_t zfs_dataset_exists(libzfs_handle_t *, const char *,
661     zfs_type_t);
662 extern int zfs_spa_version(zfs_handle_t *, int *);

664 /*
665  * Mount support functions.
666  */
667 extern boolean_t is_mounted(libzfs_handle_t *, const char *special, char **);
668 extern boolean_t zfs_is_mounted(zfs_handle_t *, char **);
669 extern int zfs_mount(zfs_handle_t *, const char *, int);
670 extern int zfs_unmount(zfs_handle_t *, const char *, int);
671 extern int zfs_unmountall(zfs_handle_t *, int);

673 /*
674  * Share support functions.
675  */
676 extern boolean_t zfs_is_shared(zfs_handle_t *);
677 extern int zfs_share(zfs_handle_t *);
678 extern int zfs_unshare(zfs_handle_t *);

680 /*
681  * Protocol-specific share support functions.
682  */
683 extern boolean_t zfs_is_shared_nfs(zfs_handle_t *, char **);
684 extern boolean_t zfs_is_shared_smb(zfs_handle_t *, char **);
685 extern int zfs_share_nfs(zfs_handle_t *);
686 extern int zfs_share_smb(zfs_handle_t *);
687 extern int zfs_shareall(zfs_handle_t *);
688 extern int zfs_unshare_nfs(zfs_handle_t *, const char *);
689 extern int zfs_unshare_smb(zfs_handle_t *, const char *);
690 extern int zfs_unshareall_nfs(zfs_handle_t *);
691 extern int zfs_unshareall_smb(zfs_handle_t *);
692 extern int zfs_unshareall_bypath(zfs_handle_t *, const char *);
693 extern int zfs_unshareall(zfs_handle_t *);
694 extern int zfs_deleg_share_nfs(libzfs_handle_t *, char *, char *, char *,
695     void *, void *, int, zfs_share_op_t);

697 /*
698  * When dealing with nvlists, verify() is extremely useful

```

```
699 */
700 #ifdef NDEBUG
701 #define verify(EX)      ((void)(EX))
702 #else
703 #define verify(EX)      assert(EX)
704 #endif
706 /*
707  * Utility function to convert a number to a human-readable form.
708  */
709 extern void zfs_nicenum(uint64_t, char *, size_t);
710 extern int zfs_nicestrtonum(libzfs_handle_t *, const char *, uint64_t *);
712 /*
713  * Given a device or file, determine if it is part of a pool.
714  */
715 extern int zpool_in_use(libzfs_handle_t *, int, pool_state_t *, char **,
716     boolean_t *);
718 /*
719  * Label manipulation.
720  */
721 extern int zpool_read_label(int, nvlist_t **);
722 extern int zpool_clear_label(int);
724 /* is this zvol valid for use as a dump device? */
725 extern int zvol_check_dump_config(char *);
727 /*
728  * Management interfaces for SMB ACL files
729  */
731 int zfs_smb_acl_add(libzfs_handle_t *, char *, char *, char *);
732 int zfs_smb_acl_remove(libzfs_handle_t *, char *, char *, char *);
733 int zfs_smb_acl_purge(libzfs_handle_t *, char *, char *);
734 int zfs_smb_acl_rename(libzfs_handle_t *, char *, char *, char *, char *);
736 /*
737  * Enable and disable datasets within a pool by mounting/unmounting and
738  * sharing/unsharing them.
739  */
740 extern int zpool_enable_datasets(zpool_handle_t *, const char *, int);
741 extern int zpool_disable_datasets(zpool_handle_t *, boolean_t);
743 /*
744  * Mappings between vdev and FRU.
745  */
746 extern void libzfs_fru_refresh(libzfs_handle_t *);
747 extern const char *libzfs_fru_lookup(libzfs_handle_t *, const char *);
748 extern const char *libzfs_fru_devpath(libzfs_handle_t *, const char *);
749 extern boolean_t libzfs_fru_compare(libzfs_handle_t *, const char *,
750     const char *);
751 extern boolean_t libzfs_fru_notself(libzfs_handle_t *, const char *);
752 extern int zpool_fru_set(zpool_handle_t *, uint64_t, const char *);
754 #ifdef __cplusplus
755 }
756 #endif
758 #endif /* _LIBZFS_H */
```

```

*****
109395 Thu Jun 28 15:57:10 2012
new/usr/src/lib/libzfs/common/libzfs_dataset.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

1397 /*
1398  * Given a property name and value, set the property for the given dataset.
1399  */
1400 int
1401 zfs_prop_set(zfs_handle_t *zhp, const char *propname, const char *propval)
1402 {
1403     zfs_cmd_t zc = { 0 };
1404     int ret = -1;
1405     prop_changelist_t *cl = NULL;
1406     char errbuf[1024];
1407     libzfs_handle_t *hdl = zhp->zfs_hdl;
1408     nvlist_t *nvl = NULL, *realprops;
1409     zfs_prop_t prop;
1410     boolean_t do_prefix = B_TRUE;
1411     boolean_t do_prefix;
1412     uint64_t idx;
1413     int added_resv;
1414
1415     (void) snprintf(errbuf, sizeof(errbuf),
1416         dgettext(TEXT_DOMAIN, "cannot set property for '%s'",
1417             zhp->zfs_name));
1418
1419     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0 ||
1420         nvlist_add_string(nvl, propname, propval) != 0) {
1421         (void) no_memory(hdl);
1422         goto error;
1423     }
1424
1425     if ((realprops = zfs_valid_proplist(hdl, zhp->zfs_type, nvl,
1426         zfs_prop_get_int(zhp, ZFS_PROP_ZONED), zhp, errbuf)) == NULL)
1427         goto error;
1428
1429     nvlist_free(nvl);
1430     nvl = realprops;
1431
1432     prop = zfs_name_to_prop(propname);
1433
1434     if (prop == ZFS_PROP_VOLSIZE) {
1435         if ((added_resv = zfs_add_synthetic_resv(zhp, nvl)) == -1)
1436             goto error;
1437     }
1438
1439     if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1440         goto error;
1441
1442     if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1443         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1444             "child dataset with inherited mountpoint is used "
1445             "in a non-global zone"));
1446         ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1447         goto error;
1448     }

```

```

1448     /*
1449     * We don't want to unmount & remount the dataset when changing
1450     * its canmount property to 'on' or 'noauto'. We only use
1451     * the changelist logic to unmount when setting canmount=off.
1452     * If the dataset's canmount property is being set to noauto,
1453     * then we want to prevent unmounting & remounting it.
1454     */
1455     if (prop == ZFS_PROP_CANMOUNT) {
1456         uint64_t idx;
1457         int err = zprop_string_to_index(prop, propval, &idx,
1458             ZFS_TYPE_DATASET);
1459         if (err == 0 && idx != ZFS_CANMOUNT_OFF)
1460             do_prefix = B_FALSE;
1461     }
1462     do_prefix = !((prop == ZFS_PROP_CANMOUNT) &&
1463         (zprop_string_to_index(prop, propval, &idx,
1464             ZFS_TYPE_DATASET) == 0) && (idx == ZFS_CANMOUNT_NOAUTO));
1465
1466     if (do_prefix && (ret = changelist_prefix(cl)) != 0)
1467         goto error;
1468
1469     /*
1470     * Execute the corresponding ioctl() to set this property.
1471     */
1472     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
1473
1474     if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1475         goto error;
1476
1477     ret = zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1478
1479     if (ret != 0) {
1480         zfs_setprop_error(hdl, prop, errno, errbuf);
1481         if (added_resv && errno == ENOSPC) {
1482             /* clean up the volsize property we tried to set */
1483             uint64_t old_volsize = zfs_prop_get_int(zhp,
1484                 ZFS_PROP_VOLSIZE);
1485             nvlist_free(nvl);
1486             zcmd_free_nvlists(&zc);
1487             if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
1488                 goto error;
1489             if (nvlist_add_uint64(nvl,
1490                 zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1491                 old_volsize) != 0)
1492                 goto error;
1493             if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1494                 goto error;
1495             (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1496         }
1497     } else {
1498         if (do_prefix)
1499             ret = changelist_postfix(cl);
1500     }
1501
1502     /*
1503     * Refresh the statistics so the new property value
1504     * is reflected.
1505     */
1506     if (ret == 0)
1507         (void) get_stats(zhp);
1508
1509 error:
1510     nvlist_free(nvl);
1511     zcmd_free_nvlists(&zc);
1512     if (cl)

```



```

1508         changelist_free(cl);
1509         return (ret);
1510     }
    _____unchanged_portion_omitted_____

2644 int
2645 zfs_get_snapused_int(zfs_handle_t *firstsnap, zfs_handle_t *lastsnap,
2646                     uint64_t *usedp)
2647 {
2648     int err;
2649     zfs_cmd_t zc = { 0 };

2651     (void) strncpy(zc.zc_name, lastsnap->zfs_name, sizeof (zc.zc_name));
2652     (void) strncpy(zc.zc_value, firstsnap->zfs_name, sizeof (zc.zc_value));

2654     err = ioctl(lastsnap->zfs_hdl->libzfs_fd, ZFS_IOC_SPACE_SNAPS, &zc);
2655     if (err)
2656         return (err);

2658     *usedp = zc.zc_cookie;

2660     return (0);
2661 }

2648 /*
2649  * Returns the name of the given zfs handle.
2650  */
2651 const char *
2652 zfs_get_name(const zfs_handle_t *zhp)
2653 {
2654     return (zhp->zfs_name);
2655 }
    _____unchanged_portion_omitted_____

2816 /*
2817  * Given a path to 'target', create all the ancestors between
2818  * the prefixlen portion of the path, and the target itself.
2819  * Fail if the initial prefixlen-ancestor does not already exist.
2820  */
2821 int
2822 create_parents(libzfs_handle_t *hdl, char *target, int prefixlen)
2823 {
2824     zfs_handle_t *h;
2825     char *cp;
2826     const char *opname;

2828     /* make sure prefix exists */
2829     cp = target + prefixlen;
2830     if (*cp != '/') {
2831         assert(strchr(cp, '/') == NULL);
2832         h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2833     } else {
2834         *cp = '\0';
2835         h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2836         *cp = '/';
2837     }
2838     if (h == NULL)
2839         return (-1);
2840     zfs_close(h);

2842     /*
2843      * Attempt to create, mount, and share any ancestor filesystems,
2844      * up to the prefixlen-long one.
2845      */
2846     for (cp = target + prefixlen + 1;
2847          cp = strchr(cp, '/'); *cp = '/', cp++) {

```

```

2863         char *logstr;

2849         *cp = '\0';

2851         h = make_dataset_handle(hdl, target);
2852         if (h) {
2853             /* it already exists, nothing to do here */
2854             zfs_close(h);
2855             continue;
2856         }

2874         logstr = hdl->libzfs_log_str;
2875         hdl->libzfs_log_str = NULL;
2858         if (zfs_create(hdl, target, ZFS_TYPE_FILESYSTEM,
2859                     NULL) != 0) {
2878             hdl->libzfs_log_str = logstr;
2860             opname = dgettext(TEXT_DOMAIN, "create");
2861             goto ancestorerr;
2862         }

2883         hdl->libzfs_log_str = logstr;
2864         h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2865         if (h == NULL) {
2866             opname = dgettext(TEXT_DOMAIN, "open");
2867             goto ancestorerr;
2868         }

2870         if (zfs_mount(h, NULL, 0) != 0) {
2871             opname = dgettext(TEXT_DOMAIN, "mount");
2872             goto ancestorerr;
2873         }

2875         if (zfs_share(h) != 0) {
2876             opname = dgettext(TEXT_DOMAIN, "share");
2877             goto ancestorerr;
2878         }

2880         zfs_close(h);
2881     }

2883     return (0);

2885 ancestorerr:
2886     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2887         "failed to %s ancestor '%s'", opname, target);
2888     return (-1);
2889 }
    _____unchanged_portion_omitted_____

2914 /*
2915  * Create a new filesystem or volume.
2916  */
2917 int
2918 zfs_create(libzfs_handle_t *hdl, const char *path, zfs_type_t type,
2919            nvlist_t *props)
2920 {
2941     zfs_cmd_t zc = { 0 };
2921     int ret;
2922     uint64_t size = 0;
2923     uint64_t blocksize = zfs_prop_default_numeric(ZFS_PROP_VOLBLOCKSIZE);
2924     char errbuf[1024];
2925     uint64_t zoned;
2926     dmu_objset_type_t ost;
2927     #ifdef /* !codereview */

2929     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,

```



```

3051             /* FALLTHROUGH */
3052         default:
3053             return (zfs_standard_error(hdl, errno, errbuf));
3054     }
3055 }
3057     return (0);
3058 }
    unchanged_portion_omitted_

3140 /*
3141  * Destroys all the snapshots named in the nvlist.  They must be underneath
3142  * the zhp (either snapshots of it, or snapshots of its descendants).
3143  */
3144 int
3145 zfs_destroy_snaps_nvlist(zfs_handle_t *zhp, nvlist_t *snaps, boolean_t defer)
3146 {
3147     int ret;
3148     nvlist_t *errlist;
3154     zfs_cmd_t zc = { 0 };

3150     ret = lzc_destroy_snaps(snaps, defer, &errlist);
3156     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3157     if (zcmd_write_src_nvlist(zhp->zfs_hdl, &zc, snaps) != 0)
3158         return (-1);
3159     zc.zc_defer_destroy = defer;

3161     ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_DESTROY_SNAPS_NVLIST, &zc);
3162     if (ret != 0) {
3163         for (nvpair_t *pair = nvlist_next_nvpair(errlist, NULL);
3164              pair != NULL; pair = nvlist_next_nvpair(errlist, pair)) {
3165             #endif /* ! codereview */
3166             char errbuf[1024];
3167             (void) snprintf(errbuf, sizeof (errbuf),
3168                 dgettext(TEXT_DOMAIN, "cannot destroy snapshot %s"),
3169                 nvpair_name(pair));
3170             #endif /* ! codereview */

3162             switch (fnvpair_value_int32(pair)) {
3163             (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3164                 "cannot destroy snapshots in %s"), zc.zc_name);

3166             switch (errno) {
3167             case EEXIST:
3168                 zfs_error_aux(zhp->zfs_hdl,
3169                     dgettext(TEXT_DOMAIN,
3170                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3171                             "snapshot is cloned"));
3172                 ret = zfs_error(zhp->zfs_hdl, EZFS_EXISTS,
3173                     errbuf);
3174                 break;
3175             return (zfs_error(zhp->zfs_hdl, EZFS_EXISTS, errbuf));

3170         default:
3171             ret = zfs_standard_error(zhp->zfs_hdl, errno,
3172                 errbuf);
3173             break;
3174         }
3175     }
3176     return (zfs_standard_error(zhp->zfs_hdl, errno,
3177         errbuf));
3178 }
3179 }
3178     return (ret);
3179     return (0);
3179 }

```

```

3181 /*
3182  * Clones the given dataset.  The target must be of the same type as the source.
3183  */
3184 int
3185 zfs_clone(zfs_handle_t *zhp, const char *target, nvlist_t *props)
3186 {
3187     zfs_cmd_t zc = { 0 };
3188     char parent[ZFS_MAXNAMELEN];
3189     int ret;
3190     char errbuf[1024];
3191     libzfs_handle_t *hdl = zhp->zfs_hdl;
3192     zfs_type_t type;
3193     uint64_t zoned;

3193     assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);

3195     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3196         "cannot create '%s'", target));

3198     /* validate the target/clone name */
3199     if (!zfs_validate_name(hdl, target, ZFS_TYPE_FILESYSTEM, B_TRUE))
3200         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3202     /* validate parents exist */
3203     if (check_parents(hdl, target, &zoned, B_FALSE, NULL) != 0)
3204         return (-1);

3206     (void) parent_name(target, parent, sizeof (parent));

3208     /* do the clone */

3210     if (props) {
3211         zfs_type_t type;
3212         #endif /* ! codereview */
3213         if (ZFS_IS_VOLUME(zhp)) {
3214             zc.zc_objset_type = DMU_OST_ZVOL;
3215             type = ZFS_TYPE_VOLUME;
3216         } else {
3217             zc.zc_objset_type = DMU_OST_ZFS;
3218             type = ZFS_TYPE_FILESYSTEM;
3219         }

3218     if (props) {
3219         if ((props = zfs_valid_proplist(hdl, type, props, zoned,
3220             zhp, errbuf)) == NULL)
3221             return (-1);

3223         if (zcmd_write_src_nvlist(hdl, &zc, props) != 0) {
3224             nvlist_free(props);
3225             return (-1);
3226         }

3223         ret = lzc_clone(target, zhp->zfs_name, props);
3224     #endif /* ! codereview */
3225     nvlist_free(props);
3226     }

3230     (void) strlcpy(zc.zc_name, target, sizeof (zc.zc_name));
3231     (void) strlcpy(zc.zc_value, zhp->zfs_name, sizeof (zc.zc_value));
3232     ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_CREATE, &zc);

3234     zcmd_free_nvlists(&zc);

3227     if (ret != 0) {
3228         switch (errno) {

```

```

3230     case ENOENT:
3231         /*
3232          * The parent doesn't exist. We should have caught this
3233          * above, but there may a race condition that has since
3234          * destroyed the parent.
3235          *
3236          * At this point, we don't know whether it's the source
3237          * that doesn't exist anymore, or whether the target
3238          * dataset doesn't exist.
3239          */
3240         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3241         "no such parent '%s'", parent));
3242         return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
3243
3244     case EXDEV:
3245         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3246         "source and target pools differ"));
3247         return (zfs_error(zhp->zfs_hdl, EZFS_CROSSTARGET,
3248         errbuf));
3249
3250     default:
3251         return (zfs_standard_error(zhp->zfs_hdl, errno,
3252         errbuf));
3253     }
3254 }
3255
3256 return (ret);
3257 }

```

unchanged portion omitted

```

3310 typedef struct snapdata {
3311     nvlist_t *sd_nvl;
3312     const char *sd_snapname;
3313 } snapdata_t;
3314
3315 static int
3316 zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
3317 {
3318     snapdata_t *sd = arg;
3319     char name[ZFS_MAXNAMELEN];
3320     int rv = 0;
3321
3322     (void) snprintf(name, sizeof (name),
3323     "%s@%s", zfs_get_name(zhp), sd->sd_snapname);
3324
3325     fnvlist_add_boolean(sd->sd_nvl, name);
3326
3327     rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
3328     zfs_close(zhp);
3329     return (rv);
3330 }
3331
3332 #endif /* ! codereview */
3333 /*
3334  * Creates snapshots. The keys in the snaps nvlist are the snapshots to be
3335  * created.
3336  * Takes a snapshot of the given dataset.
3337  */
3338 int
3339 zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, nvlist_t *props)
3340 zfs_snapshot(libzfs_handle_t *hdl, const char *path, boolean_t recursive,
3341 nvlist_t *props)
3342 {
3343     const char *delim;
3344     char parent[ZFS_MAXNAMELEN];

```

```

3327     zfs_handle_t *zhp;
3328     zfs_cmd_t zc = { 0 };
3329     int ret;
3330     char errbuf[1024];
3331     nvpair_t *elem;
3332     nvlist_t *errors;
3333 #endif /* ! codereview */
3334
3335     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3336     "cannot create snapshots "));
3337
3338     elem = NULL;
3339     while ((elem = nvlist_next_nvpair(snaps, elem)) != NULL) {
3340         const char *snapname = nvpair_name(elem);
3341         "cannot snapshot '%s'", path);
3342
3343         /* validate the target name */
3344         if (!zfs_validate_name(hdl, snapname, ZFS_TYPE_SNAPSHOT,
3345         B_TRUE)) {
3346             (void) snprintf(errbuf, sizeof (errbuf),
3347             dgettext(TEXT_DOMAIN,
3348             "cannot create snapshot '%s'", snapname));
3349             if (!zfs_validate_name(hdl, path, ZFS_TYPE_SNAPSHOT, B_TRUE))
3350                 return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3351         }
3352     }
3353 #endif /* ! codereview */
3354
3355     if (props != NULL &&
3356         (props = zfs_valid_proplist(hdl, ZFS_TYPE_SNAPSHOT,
3357         props, B_FALSE, NULL, errbuf)) == NULL) {
3358         if (props) {
3359             if ((props = zfs_valid_proplist(hdl, ZFS_TYPE_SNAPSHOT,
3360             props, B_FALSE, NULL, errbuf)) == NULL)
3361                 return (-1);
3362         }
3363     }
3364 #endif /* ! codereview */
3365
3366     ret = lzfs_snapshot(snaps, props, &errors);
3367
3368     if (ret != 0) {
3369         boolean_t printed = B_FALSE;
3370         for (elem = nvlist_next_nvpair(errors, NULL);
3371             elem != NULL;
3372             elem = nvlist_next_nvpair(errors, elem)) {
3373             (void) snprintf(errbuf, sizeof (errbuf),
3374             dgettext(TEXT_DOMAIN,
3375             "cannot create snapshot '%s'", nvpair_name(elem));
3376             (void) zfs_standard_error(hdl,
3377             fnvpair_value_int32(elem), errbuf);
3378             printed = B_TRUE;
3379         }
3380         if (!printed) {
3381             switch (ret) {
3382             case EXDEV:
3383                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3384                 "multiple snapshots of same "
3385                 "fs not allowed"));
3386                 (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3387             default:
3388                 (void) zfs_standard_error(hdl, ret, errbuf);
3389             }
3390         }
3391         if (zcmd_write_src_nvlist(hdl, &zc, props) != 0) {

```

```

3341         nvlist_free(props);
3342         return (-1);
3398     }

3400     nvlist_free(props);
3401     nvlist_free(errors);
3402     return (ret);
3403 }

3405 int
3406 zfs_snapshot(libzfs_handle_t *hdl, const char *path, boolean_t recursive,
3407             nvlist_t *props)
3408 {
3409     int ret;
3410     snapdata_t sd = { 0 };
3411     char fsname[ZFS_MAXNAMELEN];
3412     char *cp;
3413     zfs_handle_t *zhp;
3414     char errbuf[1024];

3416     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3417             "cannot snapshot %s"), path);

3419     if (!zfs_validate_name(hdl, path, ZFS_TYPE_SNAPSHOT, B_TRUE))
3420         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3421 }

3422     (void) strncpy(fsname, path, sizeof (fsname));
3423     cp = strchr(fsname, '@');
3424     *cp = '\0';
3425     sd.sd_snapname = cp + 1;
3426     /* make sure the parent exists and is of the appropriate type */
3427     delim = strchr(path, '@');
3428     (void) strncpy(parent, path, delim - path);
3429     parent[delim - path] = '\0';

3432     if ((zhp = zfs_open(hdl, fsname, ZFS_TYPE_FILESYSTEM |
3433         if ((zhp = zfs_open(hdl, parent, ZFS_TYPE_FILESYSTEM |
3434             ZFS_TYPE_VOLUME)) == NULL) {
3435         zcmd_free_nvlists(&zc);
3436         return (-1);
3437     }

3438     verify(nvlist_alloc(&sd.sd_nvlist, NV_UNIQUE_NAME, 0) == 0);
3439     if (recursive) {
3440         (void) zfs_snapshot_cb(zfs_handle_dup(zhp), &sd);
3441     } else {
3442         fnvlist_add_boolean(sd.sd_nvlist, path);
3443         (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3444         (void) strncpy(zc.zc_value, delim+1, sizeof (zc.zc_value));
3445         if (ZFS_IS_VOLUME(zhp))
3446             zc.zc_objset_type = DMU_OST_ZVOL;
3447     else
3448         zc.zc_objset_type = DMU_OST_ZFS;
3449     zc.zc_cookie = recursive;
3450     ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SNAPSHOT, &zc);

3453     zcmd_free_nvlists(&zc);

3456     /*
3457      * if it was recursive, the one that actually failed will be in
3458      * zc.zc_name.
3459      */
3460     if (ret != 0) {
3461         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3462             "cannot create snapshot '%s@%s'"), zc.zc_name, zc.zc_value);

```

```

3377         (void) zfs_standard_error(hdl, errno, errbuf);
3437     }

3439     ret = zfs_snapshot_nvlist(hdl, sd.sd_nvlist, props);
3440     nvlist_free(sd.sd_nvlist);
3441 #endif /* !codereview */
3442     zfs_close(zhp);

3443     return (ret);
3444 }
_____ unchanged_portion_omitted _____

3460 static int
3461 rollback_destroy(zfs_handle_t *zhp, void *data)
3462 {
3463     rollback_data_t *cbp = data;

3465     if (!cbp->cb_dependent) {
3466         if (strcmp(zhp->zfs_name, cbp->cb_target) != 0 &&
3467             zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
3468             zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) >
3469             cbp->cb_create) {
3470             char *logstr;

3471             cbp->cb_dependent = B_TRUE;
3472             cbp->cb_error |= zfs_iter_dependents(zhp, B_FALSE,
3473                 rollback_destroy, cbp);
3474             cbp->cb_dependent = B_FALSE;

3475             logstr = zhp->zfs_hdl->libzfs_log_str;
3476             zhp->zfs_hdl->libzfs_log_str = NULL;
3477             cbp->cb_error |= zfs_destroy(zhp, B_FALSE);
3478             zhp->zfs_hdl->libzfs_log_str = logstr;
3479         } else {
3480             /* We must destroy this clone; first unmount it */
3481             prop_changelist_t *clp;

3482             clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3483                 cbp->cb_force ? MS_FORCE : 0);
3484             if (clp == NULL || changelist_prefix(clp) != 0) {
3485                 cbp->cb_error = B_TRUE;
3486                 zfs_close(zhp);
3487                 return (0);
3488             }
3489             if (zfs_destroy(zhp, B_FALSE) != 0)
3490                 cbp->cb_error = B_TRUE;
3491             else
3492                 changelist_remove(clp, zhp->zfs_name);
3493             (void) changelist_postfix(clp);
3494             changelist_free(clp);
3495         }

3497         zfs_close(zhp);
3498         return (0);
3499 }
_____ unchanged_portion_omitted _____

```

new/usr/src/lib/libzfs/common/libzfs_impl.h

1

```
*****
6440 Thu Jun 28 15:57:10 2012
new/usr/src/lib/libzfs/common/libzfs_impl.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER SART
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2012 by Delphix. All rights reserved.
25  * Copyright (c) 2011 by Delphix. All rights reserved.
26 */
27 #ifndef LIBZFS_IMPL_H
28 #define LIBZFS_IMPL_H
29 #ifndef LIBFS_IMPL_H
30 #define LIBFS_IMPL_H
31
32 #include <sys/dmu.h>
33 #include <sys/fs/zfs.h>
34 #include <sys/zfs_ioctl.h>
35 #include <sys/spa.h>
36 #include <sys/nvpair.h>
37
38 #include <libuutil.h>
39 #include <libzfs.h>
40 #include <libshare.h>
41 #include <libzfs_core.h>
42 #endif /* ! codereview */
43
44 #include <fm/libtopo.h>
45
46 #ifdef __cplusplus
47 extern "C" {
48 #endif
49
50 #ifdef VERIFY
51 #define VERIFY verify
```

new/usr/src/lib/libzfs/common/libzfs_impl.h

2

```
53 typedef struct libzfs_fru {
54     char *zf_device;
55     char *zf_fru;
56     struct libzfs_fru *zf_chain;
57     struct libzfs_fru *zf_next;
58 } libzfs_fru_t;
59
60 struct libzfs_handle {
61     int libzfs_error;
62     int libzfs_fd;
63     FILE *libzfs_mnttab;
64     FILE *libzfs_sharetab;
65     zpool_handle_t *libzfs_pool_handles;
66     uu_avl_pool_t *libzfs_ns_avlpool;
67     uu_avl_t *libzfs_ns_avl;
68     uint64_t libzfs_ns_gen;
69     int libzfs_desc_active;
70     char libzfs_action[1024];
71     char libzfs_desc[1024];
72     char *libzfs_log_str;
73     int libzfs_printerr;
74     int libzfs_storeerr; /* stuff error messages into buffer */
75     void *libzfs_sharehdl; /* libshare handle */
76     uint_t libzfs_shareflags;
77     boolean_t libzfs_mnttab_enable;
78     avl_tree_t libzfs_mnttab_cache;
79     int libzfs_pool_iter;
80     topo_hdl_t *libzfs_topo_hdl;
81     libzfs_fru_t **libzfs_fru_hash;
82     libzfs_fru_t *libzfs_fru_list;
83     char libzfs_chassis_id[256];
84 };
85
86 unchanged portion omitted
87 #endif
88
89 #endif /* _LIBZFS_IMPL_H */
90 #endif /* _LIBFS_IMPL_H */
```

new/usr/src/lib/libzfs/common/libzfs_iter.c

1

```
*****
10758 Thu Jun 28 15:57:11 2012
new/usr/src/lib/libzfs/common/libzfs_iter.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24 * Copyright 2010 Nexenta Systems, Inc. All rights reserved.
25 * Copyright (c) 2012 by Delphix. All rights reserved.
26 * Copyright (c) 2011 by Delphix. All rights reserved.
27 */
28 #include <stdio.h>
29 #include <stdlib.h>
30 #include <strings.h>
31 #include <unistd.h>
32 #include <stddef.h>
33 #include <libintl.h>
34 #include <libzfs.h>
35
36 #include "libzfs_impl.h"
37
38 int
39 zfs_iter_clones(zfs_handle_t *zhp, zfs_iter_f func, void *data)
40 {
41     nvlist_t *nvl = zfs_get_clones_nvl(zhp);
42     nvpair_t *pair;
43
44     if (nvl == NULL)
45         return (0);
46
47     for (pair = nvlist_next_nvpair(nvl, NULL); pair != NULL;
48          pair = nvlist_next_nvpair(nvl, pair)) {
49         zfs_handle_t *clone = zfs_open(zhp->zfs_hdl, nvpair_name(pair),
50                                       ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
51         if (clone != NULL) {
52             int err = func(clone, data);
53             if (err != 0)

```

new/usr/src/lib/libzfs/common/libzfs_iter.c

2

```
54         return (err);
55     }
56 }
57 return (0);
58 }
59
60 _____ unchanged_portion_omitted _____
61
62 286 /*
63 287  * spec is a string like "A,B%C,D"
64 288  *
65 289  * <snaps>, where <snaps> can be:
66 290  *     <snap>           (single snapshot)
67 291  *     <snap>%<snap>   (range of snapshots, inclusive)
68 292  *     %<snap>         (range of snapshots, starting with earliest)
69 293  *     <snap>%        (range of snapshots, ending with last)
70 294  *     %               (all snapshots)
71 295  *     <snaps>[,...]  (comma separated list of the above)
72 296  *
73 297  * If a snapshot can not be opened, continue trying to open the others, but
74 298  * return ENOENT at the end.
75 299  */
76 300 int
77 301 zfs_iter_snapspec(zfs_handle_t *fs_zhp, const char *spec_orig,
78 302                  zfs_iter_f func, void *arg)
79 303 {
80 304     char *buf, *comma_separated, *cp;
81 305     char buf[ZFS_MAXNAMELEN];
82 306     char *comma_separated, *cp;
83 307     int err = 0;
84 308     int ret = 0;
85
86 309     buf = zfs_strdup(fs_zhp->zfs_hdl, spec_orig);
87 310     (void) strncpy(buf, spec_orig, sizeof (buf));
88 311     cp = buf;
89
90 312     while ((comma_separated = strsep(&cp, ",")) != NULL) {
91 313         char *pct = strchr(comma_separated, '%');
92 314         if (pct != NULL) {
93 315             snapspec_arg_t ssa = { 0 };
94 316             ssa.ssa_func = func;
95             ssa.ssa_arg = arg;
96
97 317             if (pct == comma_separated)
98 318                 ssa.ssa_seenfirst = B_TRUE;
99 319             else
100 320                 ssa.ssa_first = comma_separated;
101 321             *pct = '\0';
102 322             ssa.ssa_last = pct + 1;
103
104 323
105 324             /*
106 325              * If there is a lastname specified, make sure it
107 326              * exists.
108 327              */
109 328             if (ssa.ssa_last[0] != '\0') {
110 329                 char snapname[ZFS_MAXNAMELEN];
111 330                 (void) snprintf(snapname, sizeof (snapname),
112 331                                "%s@%s", zfs_get_name(fs_zhp),
113 332                                ssa.ssa_last);
114 333                 if (!zfs_dataset_exists(fs_zhp->zfs_hdl,
115 334                                        snapname, ZFS_TYPE_SNAPSHOT)) {
116 335                     ret = ENOENT;
117 336                     continue;
118 337                 }
119 338             }
120 339         }
121
122 340     }
123
124 341     err = zfs_iter_snapshots_sorted(fs_zhp,

```

```

342         snapspec_cb, &ssa);
343     if (ret == 0)
344         ret = err;
345     if (ret == 0 && (!ssa.ssa_seenfirst ||
346         (ssa.ssa_last[0] != '\0' && !ssa.ssa_seenlast))) {
347         ret = ENOENT;
348     }
349 } else {
350     char snapname[ZFS_MAXNAMELEN];
351     zfs_handle_t *snap_zhp;
352     (void) snprintf(snapname, sizeof(snapname), "%s@%s",
353         zfs_get_name(fs_zhp), comma_separated);
354     snap_zhp = make_dataset_handle(fs_zhp->zfs_hdl,
355         snapname);
356     if (snap_zhp == NULL) {
357         ret = ENOENT;
358         continue;
359     }
360     err = func(snap_zhp, arg);
361     if (ret == 0)
362         ret = err;
363 }
364 }

366     free(buf);
367 #endif /* !codereview */
368     return (ret);
369 }

371 /*
372  * Iterate over all children, snapshots and filesystems
373  */
374 int
375 zfs_iter_children(zfs_handle_t *zhp, zfs_iter_f func, void *data)
376 {
377     int ret;

379     if ((ret = zfs_iter_filesystems(zhp, func, data)) != 0)
380         return (ret);

382     return (zfs_iter_snapshots(zhp, func, data));
383 }

386 typedef struct iter_stack_frame {
387     struct iter_stack_frame *next;
388     zfs_handle_t *zhp;
389 } iter_stack_frame_t;

391 typedef struct iter_dependents_arg {
392     boolean_t first;
393     boolean_t allowrecursion;
394     iter_stack_frame_t *stack;
395     zfs_iter_f func;
396     void *data;
397 } iter_dependents_arg_t;

399 static int
400 iter_dependents_cb(zfs_handle_t *zhp, void *arg)
401 {
402     iter_dependents_arg_t *ida = arg;
403     int err;
404     boolean_t first = ida->first;
405     ida->first = B_FALSE;

407     if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {

```

```

408         err = zfs_iter_clones(zhp, iter_dependents_cb, ida);
409     } else {
410         iter_stack_frame_t isf;
411         iter_stack_frame_t *f;

413         /*
414          * check if there is a cycle by seeing if this fs is already
415          * on the stack.
416          */
417         for (f = ida->stack; f != NULL; f = f->next) {
418             if (f->zhp->zfs_dmustats.dds_guid ==
419                 zhp->zfs_dmustats.dds_guid) {
420                 if (ida->allowrecursion) {
421                     zfs_close(zhp);
422                     return (0);
423                 } else {
424                     zfs_error_aux(zhp->zfs_hdl,
425                         dgettext(TEXT_DOMAIN,
426                             "recursive dependency at '%s'",
427                             zfs_get_name(zhp)));
428                     err = zfs_error(zhp->zfs_hdl,
429                         EZFS_RECURSIVE,
430                         dgettext(TEXT_DOMAIN,
431                             "cannot determine dependent "
432                             "datasets"));
433                     zfs_close(zhp);
434                     return (err);
435                 }
436             }
437         }

439         isf.zhp = zhp;
440         isf.next = ida->stack;
441         ida->stack = &isf;
442         err = zfs_iter_filesystems(zhp, iter_dependents_cb, ida);
443         if (err == 0)
444             err = zfs_iter_snapshots(zhp, iter_dependents_cb, ida);
445         ida->stack = isf.next;
446     }
447     if (!first && err == 0)
448         err = ida->func(zhp, ida->data);
449     return (err);
450 }

452 int
453 zfs_iter_dependents(zfs_handle_t *zhp, boolean_t allowrecursion,
454     zfs_iter_f func, void *data)
455 {
456     iter_dependents_arg_t ida;
457     ida.allowrecursion = allowrecursion;
458     ida.stack = NULL;
459     ida.func = func;
460     ida.data = data;
461     ida.first = B_TRUE;
462     return (iter_dependents_cb(zfs_handle_dup(zhp), &ida));
463 }

```



```

*****
103406 Thu Jun 28 15:57:11 2012
new/usr/src/lib/libzfs/common/libzfs_pool.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25  * Copyright (c) 2012 by Delphix. All rights reserved.
26 */

28 #include <ctype.h>
29 #include <errno.h>
30 #include <devid.h>
31 #include <fcntl.h>
32 #include <libintl.h>
33 #include <stdio.h>
34 #include <stdlib.h>
35 #include <strings.h>
36 #include <unistd.h>
37 #include <libgen.h>
38 #endif /* ! codereview */
39 #include <sys/efi_partition.h>
40 #include <sys/vtoc.h>
41 #include <sys/zfs_ioctl.h>
42 #include <dlfcn.h>

44 #include "zfs_namecheck.h"
45 #include "zfs_prop.h"
46 #include "libzfs_impl.h"
47 #include "zfs_comutil.h"
48 #include "zfeature_common.h"

50 static int read_efi_label(nvlist_t *config, diskaddr_t *sb);

52 #define DISK_ROOT      "/dev/dsk"
53 #define RDISK_ROOT    "/dev/rdisk"
54 #define BACKUP_SLICE "s2"

```

```

56 typedef struct prop_flags {
57     int create:1; /* Validate property on creation */
58     int import:1; /* Validate property on import */
59 } prop_flags_t;

61 /*
62  * =====
63  * zpool property functions
64  * =====
65 */

67 static int
68 zpool_get_all_props(zpool_handle_t *zhp)
69 {
70     zfs_cmd_t zc = { 0 };
71     libzfs_handle_t *hdl = zhp->zpool_hdl;

73     (void) strncpy(zc.zc_name, zhp->zpool_name, sizeof (zc.zc_name));

75     if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0)
76         return (-1);

78     while (ioctl(hdl->libzfs_fd, ZFS_IOC_POOL_GET_PROPS, &zc) != 0) {
79         if (errno == ENOMEM) {
80             if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
81                 zcmd_free_nvlists(&zc);
82                 return (-1);
83             }
84         } else {
85             zcmd_free_nvlists(&zc);
86             return (-1);
87         }
88     }

90     if (zcmd_read_dst_nvlist(hdl, &zc, &zhp->zpool_props) != 0) {
91         zcmd_free_nvlists(&zc);
92         return (-1);
93     }

95     zcmd_free_nvlists(&zc);

97     return (0);
98 }

100 static int
101 zpool_props_refresh(zpool_handle_t *zhp)
102 {
103     nvlist_t *old_props;

105     old_props = zhp->zpool_props;

107     if (zpool_get_all_props(zhp) != 0)
108         return (-1);

110     nvlist_free(old_props);
111     return (0);
112 }

114 static char *
115 zpool_get_prop_string(zpool_handle_t *zhp, zpool_prop_t prop,
116                     zprop_source_t *src)
117 {
118     nvlist_t *nv, *nvl;
119     uint64_t ival;
120     char *value;

```

```

121     zprop_source_t source;
122
123     nvl = zhp->zpool_props;
124     if (nvlist_lookup_nvlist(nvl, zpool_prop_to_name(prop), &nv) == 0) {
125         verify(nvlist_lookup_uint64(nv, ZPROP_SOURCE, &ival) == 0);
126         source = ival;
127         verify(nvlist_lookup_string(nv, ZPROP_VALUE, &value) == 0);
128     } else {
129         source = ZPROP_SRC_DEFAULT;
130         if ((value = (char *)zpool_prop_default_string(prop)) == NULL)
131             value = "-";
132     }
133
134     if (src)
135         *src = source;
136
137     return (value);
138 }
139
140 uint64_t
141 zpool_get_prop_int(zpool_handle_t *zhp, zpool_prop_t prop, zprop_source_t *src)
142 {
143     nvlist_t *nv, *nvl;
144     uint64_t value;
145     zprop_source_t source;
146
147     if (zhp->zpool_props == NULL && zpool_get_all_props(zhp)) {
148         /*
149          * zpool_get_all_props() has most likely failed because
150          * the pool is faulted, but if all we need is the top level
151          * vdev's guid then get it from the zhp config nvlist.
152          */
153         if ((prop == ZPOOL_PROP_GUID) &&
154             (nvlist_lookup_nvlist(zhp->zpool_config,
155                                  ZPOOL_CONFIG_VDEV_TREE, &nv) == 0) &&
156             (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &value)
157              == 0)) {
158             return (value);
159         }
160         return (zpool_prop_default_numeric(prop));
161     }
162
163     nvl = zhp->zpool_props;
164     if (nvlist_lookup_nvlist(nvl, zpool_prop_to_name(prop), &nv) == 0) {
165         verify(nvlist_lookup_uint64(nv, ZPROP_SOURCE, &value) == 0);
166         source = value;
167         verify(nvlist_lookup_uint64(nv, ZPROP_VALUE, &value) == 0);
168     } else {
169         source = ZPROP_SRC_DEFAULT;
170         value = zpool_prop_default_numeric(prop);
171     }
172
173     if (src)
174         *src = source;
175
176     return (value);
177 }
178
179 /*
180  * Map VDEV STATE to printed strings.
181  */
182 char *
183 zpool_state_to_name(vdev_state_t state, vdev_aux_t aux)
184 {
185     switch (state) {
186     case VDEV_STATE_CLOSED:

```

```

187     case VDEV_STATE_OFFLINE:
188         return (gettext("OFFLINE"));
189     case VDEV_STATE_REMOVED:
190         return (gettext("REMOVED"));
191     case VDEV_STATE_CANT_OPEN:
192         if (aux == VDEV_AUX_CORRUPT_DATA || aux == VDEV_AUX_BAD_LOG)
193             return (gettext("FAULTED"));
194         else if (aux == VDEV_AUX_SPLIT_POOL)
195             return (gettext("SPLIT"));
196         else
197             return (gettext("UNAVAIL"));
198     case VDEV_STATE_FAULTED:
199         return (gettext("FAULTED"));
200     case VDEV_STATE_DEGRADED:
201         return (gettext("DEGRADED"));
202     case VDEV_STATE_HEALTHY:
203         return (gettext("ONLINE"));
204     }
205
206     return (gettext("UNKNOWN"));
207 }
208
209 /*
210  * Get a zpool property value for 'prop' and return the value in
211  * a pre-allocated buffer.
212  */
213 int
214 zpool_get_prop(zpool_handle_t *zhp, zpool_prop_t prop, char *buf, size_t len,
215               zprop_source_t *srctype)
216 {
217     uint64_t intval;
218     const char *strval;
219     zprop_source_t src = ZPROP_SRC_NONE;
220     nvlist_t *nvroot;
221     vdev_stat_t *vs;
222     uint_t vsc;
223
224     if (zpool_get_state(zhp) == POOL_STATE_UNAVAIL) {
225         switch (prop) {
226         case ZPOOL_PROP_NAME:
227             (void) strncpy(buf, zpool_get_name(zhp), len);
228             break;
229
230         case ZPOOL_PROP_HEALTH:
231             (void) strncpy(buf, "FAULTED", len);
232             break;
233
234         case ZPOOL_PROP_GUID:
235             intval = zpool_get_prop_int(zhp, prop, &src);
236             (void) snprintf(buf, len, "%llu", intval);
237             break;
238
239         case ZPOOL_PROP_ALTROOT:
240         case ZPOOL_PROP_CACHEFILE:
241         case ZPOOL_PROP_COMMENT:
242             if (zhp->zpool_props != NULL ||
243                 zpool_get_all_props(zhp) == 0) {
244                 (void) strncpy(buf,
245                                 zpool_get_prop_string(zhp, prop, &src),
246                                 len);
247                 if (srctype != NULL)
248                     *srctype = src;
249                 return (0);
250             }
251             /* FALLTHROUGH */
252         default:

```



```

385 /*
386  * Given an nvlist of zpool properties to be set, validate that they are
387  * correct, and parse any numeric properties (index, boolean, etc) if they are
388  * specified as strings.
389  */
390 static nvlist_t *
391 zpool_valid_proplist(libzfs_handle_t *hdl, const char *poolname,
392                     nvlist_t *props, uint64_t version, prop_flags_t flags, char *errbuf)
393 {
394     nvpair_t *elem;
395     nvlist_t *retprops;
396     zpool_prop_t prop;
397     char *strval;
398     uint64_t intval;
399     char *slash, *check;
400     struct stat64 statbuf;
401     zpool_handle_t *zhp;
402     nvlist_t *nvroot;
403
404     if (nvlist_alloc(&retprops, NV_UNIQUE_NAME, 0) != 0) {
405         (void) no_memory(hdl);
406         return (NULL);
407     }
408
409     elem = NULL;
410     while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
411         const char *propname = nvpair_name(elem);
412
413         prop = zpool_name_to_prop(propname);
414         if (prop == ZPROP_INVALID && zpool_prop_feature(propname)) {
415             int err;
416             zfeature_info_t *feature;
417             char *fname = strchr(propname, '@') + 1;
418
419             err = zfeature_lookup_name(fname, &feature);
420             if (err != 0) {
421                 ASSERT3U(err, ==, ENOENT);
422                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
423                     "invalid feature '%s'", fname));
424                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
425                 goto error;
426             }
427
428             if (nvpair_type(elem) != DATA_TYPE_STRING) {
429                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
430                     "'%s' must be a string"), propname);
431                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
432                 goto error;
433             }
434
435             (void) nvpair_value_string(elem, &strval);
436             if (strcmp(strval, ZFS_FEATURE_ENABLED) != 0) {
437                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
438                     "property '%s' can only be set to "
439                     "'enabled'"), propname);
440                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
441                 goto error;
442             }
443
444             if (nvlist_add_uint64(retprops, propname, 0) != 0) {
445                 (void) no_memory(hdl);
446                 goto error;
447             }
448             continue;
449         }

```

```

451     /*
452     * Make sure this property is valid and applies to this type.
453     */
454     if (prop == ZPROP_INVALID) {
455         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
456             "invalid property '%s'", propname));
457         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
458         goto error;
459     }
460
461     if (zpool_prop_readonly(prop)) {
462         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "'%s' "
463             "is readonly"), propname);
464         (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
465         goto error;
466     }
467
468     if (zprop_parse_value(hdl, elem, prop, ZFS_TYPE_POOL, retprops,
469         &strval, &intval, errbuf) != 0)
470         goto error;
471
472     /*
473     * Perform additional checking for specific properties.
474     */
475     switch (prop) {
476     case ZPOOL_PROP_VERSION:
477         if (intval < version ||
478             !SPA_VERSION_IS_SUPPORTED(intval)) {
479             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
480                 "property '%s' number %d is invalid."),
481                 propname, intval);
482             (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
483             goto error;
484         }
485         break;
486
487     case ZPOOL_PROP_BOOTFS:
488         if (flags.create || flags.import) {
489             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
490                 "property '%s' cannot be set at creation "
491                 "or import time"), propname);
492             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
493             goto error;
494         }
495
496         if (version < SPA_VERSION_BOOTFS) {
497             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
498                 "pool must be upgraded to support "
499                 "'%s' property"), propname);
500             (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
501             goto error;
502         }
503
504         /*
505         * bootfs property value has to be a dataset name and
506         * the dataset has to be in the same pool as it sets to.
507         */
508         if (strval[0] != '\0' && !bootfs_name_valid(poolname,
509             strval)) {
510             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "'%s' "
511                 "is an invalid name"), strval);
512             (void) zfs_error(hdl, EZFS_INVALIDIDNAME, errbuf);
513             goto error;
514         }
515
516         if ((zhp = zpool_open_canfail(hdl, poolname)) == NULL) {

```

```

517         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
518             "could not open pool '%s'", poolname);
519         (void) zfs_error(hdl, EZFS_OPENFAILED, errbuf);
520         goto error;
521     }
522     verify(nvlist_lookup_nvlist(zpool_get_config(zhp, NULL),
523         ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
524
525     /*
526     * bootfs property cannot be set on a disk which has
527     * been EFI labeled.
528     */
529     if (pool_uses_efi(nvroot)) {
530         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
531             "property '%s' not supported on "
532             "EFI labeled devices"), propname);
533         (void) zfs_error(hdl, EZFS_POOL_NOTSUP, errbuf);
534         zpool_close(zhp);
535         goto error;
536     }
537     zpool_close(zhp);
538     break;
539
540     case ZPOOL_PROP_ALTROOT:
541         if (!flags.create && !flags.import) {
542             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
543                 "property '%s' can only be set during pool "
544                 "creation or import"), propname);
545             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
546             goto error;
547         }
548
549         if (strval[0] != '/') {
550             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
551                 "bad alternate root '%s'", strval);
552             (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
553             goto error;
554         }
555         break;
556
557     case ZPOOL_PROP_CACHEFILE:
558         if (strval[0] == '\0')
559             break;
560
561         if (strcmp(strval, "none") == 0)
562             break;
563
564         if (strval[0] != '/') {
565             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
566                 "property '%s' must be empty, an "
567                 "absolute path, or 'none'"), propname);
568             (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
569             goto error;
570         }
571
572         slash = strrchr(strval, '/');
573
574         if (slash[1] == '\0' || strcmp(slash, "/.") == 0 ||
575             strcmp(slash, "/..") == 0) {
576             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
577                 "'%s' is not a valid file"), strval);
578             (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
579             goto error;
580         }
581
582         *slash = '\0';

```

```

584         if (strval[0] != '\0' &&
585             (stat64(strval, &statbuf) != 0 ||
586             !S_ISDIR(statbuf.st_mode))) {
587             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
588                 "'%s' is not a valid directory"),
589                 strval);
590             (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
591             goto error;
592         }
593
594         *slash = '/';
595         break;
596
597     case ZPOOL_PROP_COMMENT:
598         for (check = strval; *check != '\0'; check++) {
599             if (!isprint(*check)) {
600                 zfs_error_aux(hdl,
601                     dgettext(TEXT_DOMAIN,
602                         "comment may only have printable "
603                         "characters"));
604                 (void) zfs_error(hdl, EZFS_BADPROP,
605                     errbuf);
606                 goto error;
607             }
608         }
609         if (strlen(strval) > ZPROP_MAX_COMMENT) {
610             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
611                 "comment must not exceed %d characters"),
612                 ZPROP_MAX_COMMENT);
613             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
614             goto error;
615         }
616         break;
617     case ZPOOL_PROP_READONLY:
618         if (!flags.import) {
619             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
620                 "property '%s' can only be set at "
621                 "import time"), propname);
622             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
623             goto error;
624         }
625         break;
626     }
627 }
628
629     return (retprops);
630 error:
631     nvlist_free(retprops);
632     return (NULL);
633 }
634
635 /*
636  * Set zpool property : propname=propval.
637  */
638 int
639 zpool_set_prop(zpool_handle_t *zhp, const char *propname, const char *propval)
640 {
641     zfs_cmd_t zc = { 0 };
642     int ret = -1;
643     char errbuf[1024];
644     nvlist_t *nvl = NULL;
645     nvlist_t *realprops;
646     uint64_t version;
647     prop_flags_t flags = { 0 };

```

```

649     (void) snprintf(errbuf, sizeof (errbuf),
650         dgettext(TEXT_DOMAIN, "cannot set property for '%s'"),
651         zhp->zpool_name);

653     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
654         return (no_memory(zhp->zpool_hdl));

656     if (nvlist_add_string(nvl, propname, propval) != 0) {
657         nvlist_free(nvl);
658         return (no_memory(zhp->zpool_hdl));
659     }

661     version = zpool_get_prop_int(zhp, ZPOOL_PROP_VERSION, NULL);
662     if ((realprops = zpool_valid_proplist(zhp->zpool_hdl,
663         zhp->zpool_name, nvl, version, flags, errbuf)) == NULL) {
664         nvlist_free(nvl);
665         return (-1);
666     }

668     nvlist_free(nvl);
669     nvl = realprops;

671     /*
672     * Execute the corresponding ioctl() to set this property.
673     */
674     (void) strncpy(zc.zc_name, zhp->zpool_name, sizeof (zc.zc_name));

676     if (zcmd_write_src_nvlist(zhp->zpool_hdl, &zc, nvl) != 0) {
677         nvlist_free(nvl);
678         return (-1);
679     }

681     ret = zfs_ioctl(zhp->zpool_hdl, ZFS_IOC_POOL_SET_PROPS, &zc);

683     zcmd_free_nvlists(&zc);
684     nvlist_free(nvl);

686     if (ret)
687         (void) zpool_standard_error(zhp->zpool_hdl, errno, errbuf);
688     else
689         (void) zpool_props_refresh(zhp);

691     return (ret);
692 }

694 int
695 zpool_expand_proplist(zpool_handle_t *zhp, zprop_list_t **plp)
696 {
697     libzfs_handle_t *hdl = zhp->zpool_hdl;
698     zprop_list_t *entry;
699     char buf[ZFS_MAXPROPLEN];
700     nvlist_t *features = NULL;
701     zprop_list_t **last;
702     boolean_t firstexpand = (NULL == *plp);

704     if (zprop_expand_list(hdl, plp, ZFS_TYPE_POOL) != 0)
705         return (-1);

707     last = plp;
708     while (*last != NULL)
709         last = &(*last)->pl_next;

711     if ((*plp)->pl_all)
712         features = zpool_get_features(zhp);

714     if ((*plp)->pl_all && firstexpand) {

```

```

715         for (int i = 0; i < SPA_FEATURES; i++) {
716             zprop_list_t *entry = zfs_alloc(hdl,
717                 sizeof (zprop_list_t));
718             entry->pl_prop = ZPROP_INVALID;
719             entry->pl_user_prop = zfs_asprintf(hdl, "feature@%s",
720                 spa_feature_table[i].fi_uname);
721             entry->pl_width = strlen(entry->pl_user_prop);
722             entry->pl_all = B_TRUE;

724             *last = entry;
725             last = &entry->pl_next;
726         }
727     }

729     /* add any unsupported features */
730     for (nvpair_t *nvp = nvlist_next_nvpair(features, NULL);
731         nvp != NULL; nvp = nvlist_next_nvpair(features, nvp)) {
732         char *propname;
733         boolean_t found;
734         zprop_list_t *entry;

736         if (zfeature_is_supported(nvpair_name(nvp)))
737             continue;

739         propname = zfs_asprintf(hdl, "unsupported@%s",
740             nvpair_name(nvp));

742         /*
743         * Before adding the property to the list make sure that no
744         * other pool already added the same property.
745         */
746         found = B_FALSE;
747         entry = *plp;
748         while (entry != NULL) {
749             if (entry->pl_user_prop != NULL &&
750                 strcmp(propname, entry->pl_user_prop) == 0) {
751                 found = B_TRUE;
752                 break;
753             }
754             entry = entry->pl_next;
755         }
756         if (found) {
757             free(propname);
758             continue;
759         }

761         entry = zfs_alloc(hdl, sizeof (zprop_list_t));
762         entry->pl_prop = ZPROP_INVALID;
763         entry->pl_user_prop = propname;
764         entry->pl_width = strlen(entry->pl_user_prop);
765         entry->pl_all = B_TRUE;

767         *last = entry;
768         last = &entry->pl_next;
769     }

771     for (entry = *plp; entry != NULL; entry = entry->pl_next) {
773         if (entry->pl_fixed)
774             continue;

776         if (entry->pl_prop != ZPROP_INVALID &&
777             zpool_get_prop(zhp, entry->pl_prop, buf, sizeof (buf),
778                 NULL) == 0) {
779             if (strlen(buf) > entry->pl_width)
780                 entry->pl_width = strlen(buf);

```

```

781     }
782 }
783
784     return (0);
785 }
786
787 /*
788  * Get the state for the given feature on the given ZFS pool.
789  */
790 int
791 zpool_prop_get_feature(zpool_handle_t *zhp, const char *propname, char *buf,
792     size_t len)
793 {
794     uint64_t refcount;
795     boolean_t found = B_FALSE;
796     nvlist_t *features = zpool_get_features(zhp);
797     boolean_t supported;
798     const char *feature = strchr(propname, '@') + 1;
799
800     supported = zpool_prop_feature(propname);
801     ASSERT(supported || zfs_prop_unsupported(propname));
802
803     /*
804      * Convert from feature name to feature guid. This conversion is
805      * unnecessary for unsupported@... properties because they already
806      * use guids.
807      */
808     if (supported) {
809         int ret;
810         zfeature_info_t *fi;
811
812         ret = zfeature_lookup_name(feature, &fi);
813         if (ret != 0) {
814             (void) strlcpy(buf, "-", len);
815             return (ENOTSUP);
816         }
817         feature = fi->fi_guid;
818     }
819
820     if (nvlist_lookup_uint64(features, feature, &refcount) == 0)
821         found = B_TRUE;
822
823     if (supported) {
824         if (!found) {
825             (void) strlcpy(buf, ZFS_FEATURE_DISABLED, len);
826         } else {
827             if (refcount == 0)
828                 (void) strlcpy(buf, ZFS_FEATURE_ENABLED, len);
829             else
830                 (void) strlcpy(buf, ZFS_FEATURE_ACTIVE, len);
831         }
832     } else {
833         if (found) {
834             if (refcount == 0) {
835                 (void) strlcpy(buf, ZFS_UNSUPPORTED_INACTIVE);
836             } else {
837                 (void) strlcpy(buf, ZFS_UNSUPPORTED_READONLY);
838             }
839         } else {
840             (void) strlcpy(buf, "-", len);
841             return (ENOTSUP);
842         }
843     }
844
845     return (0);
846 }

```

```

848 /*
849  * Don't start the slice at the default block of 34; many storage
850  * devices will use a stripe width of 128k, so start there instead.
851  */
852 #define NEW_START_BLOCK 256
853
854 /*
855  * Validate the given pool name, optionally putting an extended error message in
856  * 'buf'.
857  */
858 boolean_t
859 zpool_name_valid(libzfs_handle_t *hdl, boolean_t isopen, const char *pool)
860 {
861     namecheck_err_t why;
862     char what;
863     int ret;
864
865     ret = pool_namecheck(pool, &why, &what);
866
867     /*
868      * The rules for reserved pool names were extended at a later point.
869      * But we need to support users with existing pools that may now be
870      * invalid. So we only check for this expanded set of names during a
871      * create (or import), and only in userland.
872      */
873     if (ret == 0 && !isopen &&
874         (strncmp(pool, "mirror", 6) == 0 ||
875          strncmp(pool, "raidz", 5) == 0 ||
876          strncmp(pool, "spare", 5) == 0 ||
877          strcmp(pool, "log") == 0)) {
878         if (hdl != NULL)
879             zfs_error_aux(hdl,
880                 dgettext(TEXT_DOMAIN, "name is reserved"));
881         return (B_FALSE);
882     }
883
884     if (ret != 0) {
885         if (hdl != NULL) {
886             switch (why) {
887                 case NAME_ERR_TOOLONG:
888                     zfs_error_aux(hdl,
889                         dgettext(TEXT_DOMAIN, "name is too long"));
890                     break;
891
892                 case NAME_ERR_INVALIDCHAR:
893                     zfs_error_aux(hdl,
894                         dgettext(TEXT_DOMAIN, "invalid character "
895                             "'%c' in pool name"), what);
896                     break;
897
898                 case NAME_ERR_NOLETTER:
899                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
900                         "name must begin with a letter"));
901                     break;
902
903                 case NAME_ERR_RESERVED:
904                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
905                         "name is reserved"));
906                     break;
907
908                 case NAME_ERR_DISKLIKE:
909                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
910                         "pool name is reserved"));
911                     break;
912             }
913         }
914     }
915 }

```

```

914         case NAME_ERR_LEADING_SLASH:
915             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
916                 "leading slash in name"));
917             break;
918
919         case NAME_ERR_EMPTY_COMPONENT:
920             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
921                 "empty component in name"));
922             break;
923
924         case NAME_ERR_TRAILING_SLASH:
925             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
926                 "trailing slash in name"));
927             break;
928
929         case NAME_ERR_MULTIPLE_AT:
930             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
931                 "multiple '@' delimiters in name"));
932             break;
933     }
934     }
935     return (B_FALSE);
936 }
937
938 return (B_TRUE);
939 }
940
941 /*
942  * Open a handle to the given pool, even if the pool is currently in the FAULTED
943  * state.
944  */
945 zpool_handle_t *
946 zpool_open_canfail(libzfs_handle_t *hdl, const char *pool)
947 {
948     zpool_handle_t *zhp;
949     boolean_t missing;
950
951     /*
952      * Make sure the pool name is valid.
953      */
954     if (!zpool_name_valid(hdl, B_TRUE, pool)) {
955         (void) zfs_error_fmt(hdl, EZFS_INVALIDNAME,
956             dgettext(TEXT_DOMAIN, "cannot open '%s'"),
957             pool);
958         return (NULL);
959     }
960
961     if ((zhp = zfs_alloc(hdl, sizeof (zpool_handle_t))) == NULL)
962         return (NULL);
963
964     zhp->zpool_hdl = hdl;
965     (void) strncpy(zhp->zpool_name, pool, sizeof (zhp->zpool_name));
966
967     if (zpool_refresh_stats(zhp, &missing) != 0) {
968         zpool_close(zhp);
969         return (NULL);
970     }
971
972     if (missing) {
973         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "no such pool"));
974         (void) zfs_error_fmt(hdl, EZFS_NOENT,
975             dgettext(TEXT_DOMAIN, "cannot open '%s'"), pool);
976         zpool_close(zhp);
977         return (NULL);
978     }

```

```

979     }
980
981     return (zhp);
982 }
983
984 /*
985  * Like the above, but silent on error. Used when iterating over pools (because
986  * the configuration cache may be out of date).
987  */
988 int
989 zpool_open_silent(libzfs_handle_t *hdl, const char *pool, zpool_handle_t **ret)
990 {
991     zpool_handle_t *zhp;
992     boolean_t missing;
993
994     if ((zhp = zfs_alloc(hdl, sizeof (zpool_handle_t))) == NULL)
995         return (-1);
996
997     zhp->zpool_hdl = hdl;
998     (void) strncpy(zhp->zpool_name, pool, sizeof (zhp->zpool_name));
999
1000     if (zpool_refresh_stats(zhp, &missing) != 0) {
1001         zpool_close(zhp);
1002         return (-1);
1003     }
1004
1005     if (missing) {
1006         zpool_close(zhp);
1007         *ret = NULL;
1008         return (0);
1009     }
1010
1011     *ret = zhp;
1012     return (0);
1013 }
1014
1015 /*
1016  * Similar to zpool_open_canfail(), but refuses to open pools in the faulted
1017  * state.
1018  */
1019 zpool_handle_t *
1020 zpool_open(libzfs_handle_t *hdl, const char *pool)
1021 {
1022     zpool_handle_t *zhp;
1023
1024     if ((zhp = zpool_open_canfail(hdl, pool)) == NULL)
1025         return (NULL);
1026
1027     if (zhp->zpool_state == POOL_STATE_UNAVAIL) {
1028         (void) zfs_error_fmt(hdl, EZFS_POOLUNAVAIL,
1029             dgettext(TEXT_DOMAIN, "cannot open '%s'"), zhp->zpool_name);
1030         zpool_close(zhp);
1031         return (NULL);
1032     }
1033
1034     return (zhp);
1035 }
1036
1037 /*
1038  * Close the handle. Simply frees the memory associated with the handle.
1039  */
1040 void
1041 zpool_close(zpool_handle_t *zhp)
1042 {
1043     if (zhp->zpool_config)
1044         nvlist_free(zhp->zpool_config);

```



```

1045     if (zhp->zpool_old_config)
1046         nvlist_free(zhp->zpool_old_config);
1047     if (zhp->zpool_props)
1048         nvlist_free(zhp->zpool_props);
1049     free(zhp);
1050 }

1052 /*
1053  * Return the name of the pool.
1054  */
1055 const char *
1056 zpool_get_name(zpool_handle_t *zhp)
1057 {
1058     return (zhp->zpool_name);
1059 }

1062 /*
1063  * Return the state of the pool (ACTIVE or UNAVAILABLE)
1064  */
1065 int
1066 zpool_get_state(zpool_handle_t *zhp)
1067 {
1068     return (zhp->zpool_state);
1069 }

1071 /*
1072  * Create the named pool, using the provided vdev list. It is assumed
1073  * that the consumer has already validated the contents of the nvlist, so we
1074  * don't have to worry about error semantics.
1075  */
1076 int
1077 zpool_create(libzfs_handle_t *hdl, const char *pool, nvlist_t *nvroot,
1078             nvlist_t *props, nvlist_t *fsprops)
1079 {
1080     zfs_cmd_t zc = { 0 };
1081     nvlist_t *zc_fsprops = NULL;
1082     nvlist_t *zc_props = NULL;
1083     char msg[1024];
1084     char *altroot;
1085     int ret = -1;

1087     (void) snprintf(msg, sizeof (msg), dgettext(TEXT_DOMAIN,
1088         "cannot create '%s'", pool);

1090     if (!zpool_name_valid(hdl, B_FALSE, pool))
1091         return (zfs_error(hdl, EZFS_INVALIDNAME, msg));

1093     if (zcmd_write_conf_nvlist(hdl, &zc, nvroot) != 0)
1094         return (-1);

1096     if (props) {
1097         prop_flags_t flags = { .create = B_TRUE, .import = B_FALSE };

1099         if ((zc_props = zpool_valid_proplist(hdl, pool, props,
1100             SPA_VERSION_1, flags, msg)) == NULL) {
1101             goto create_failed;
1102         }
1103     }

1105     if (fsprops) {
1106         uint64_t zoned;
1107         char *zonestr;

1109         zoned = ((nvlist_lookup_string(fsprops,
1110             zfs_prop_to_name(ZFS_PROP_ZONED), &zonestr) == 0) &&

```

```

1111         strcmp(zonestr, "on") == 0);

1113     if ((zc_fsprops = zfs_valid_proplist(hdl,
1114         ZFS_TYPE_FILESYSTEM, fsprops, zoned, NULL, msg)) == NULL) {
1115         goto create_failed;
1116     }
1117     if (!zc_props &&
1118         (nvlist_alloc(&zc_props, NV_UNIQUE_NAME, 0) != 0)) {
1119         goto create_failed;
1120     }
1121     if (nvlist_add_nvlist(zc_props,
1122         ZPOOL_ROOTFS_PROPS, zc_fsprops) != 0) {
1123         goto create_failed;
1124     }
1125 }

1127     if (zc_props && zcmd_write_src_nvlist(hdl, &zc, zc_props) != 0)
1128         goto create_failed;

1130     (void) strncpy(zc.zc_name, pool, sizeof (zc.zc_name));

1132     if ((ret = zfs_ioctl(hdl, ZFS_IOC_POOL_CREATE, &zc)) != 0) {

1134         zcmd_free_nvlists(&zc);
1135         nvlist_free(zc_props);
1136         nvlist_free(zc_fsprops);

1138         switch (errno) {
1139             case EBUSY:
1140                 /*
1141                  * This can happen if the user has specified the same
1142                  * device multiple times. We can't reliably detect this
1143                  * until we try to add it and see we already have a
1144                  * label.
1145                  */
1146                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1147                     "one or more vdevs refer to the same device"));
1148                 return (zfs_error(hdl, EZFS_BADDEV, msg));

1150             case EOVERFLOW:
1151                 /*
1152                  * This occurs when one of the devices is below
1153                  * SPA_MINDEVSIZE. Unfortunately, we can't detect which
1154                  * device was the problem device since there's no
1155                  * reliable way to determine device size from userland.
1156                  */
1157                 {
1158                     char buf[64];

1160                     zfs_nicenum(SPA_MINDEVSIZE, buf, sizeof (buf));

1162                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1163                         "one or more devices is less than the "
1164                         "minimum size (%s)", buf);
1165                 }
1166                 return (zfs_error(hdl, EZFS_BADDEV, msg));

1168             case ENOSPC:
1169                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1170                     "one or more devices is out of space"));
1171                 return (zfs_error(hdl, EZFS_BADDEV, msg));

1173             case ENOTBLK:
1174                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1175                     "cache device must be a disk or disk slice"));
1176                 return (zfs_error(hdl, EZFS_BADDEV, msg));

```

```

1178         default:
1179             return (zpool_standard_error(hdl, errno, msg));
1180     }
1181 }
1182
1183 /*
1184  * If this is an alternate root pool, then we automatically set the
1185  * mountpoint of the root dataset to be '/'.
1186  */
1187 if (nvlist_lookup_string(props, zpool_prop_to_name(ZPOOL_PROP_ALTROOT),
1188     &altroot) == 0) {
1189     zfs_handle_t *zhp;
1190
1191     verify((zhp = zfs_open(hdl, pool, ZFS_TYPE_DATASET)) != NULL);
1192     verify(zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1193         "/") == 0);
1194
1195     zfs_close(zhp);
1196 }
1197
1198 create_failed:
1199     zcmd_free_nvlists(&zc);
1200     nvlist_free(zc_props);
1201     nvlist_free(zc_fsprops);
1202     return (ret);
1203 }
1204
1205 /*
1206  * Destroy the given pool. It is up to the caller to ensure that there are no
1207  * datasets left in the pool.
1208  */
1209 int
1210 zpool_destroy(zpool_handle_t *zhp, const char *log_str)
1211 {
1212     zfs_cmd_t zc = { 0 };
1213     zfs_handle_t *zfp = NULL;
1214     libzfs_handle_t *hdl = zhp->zpool_hdl;
1215     char msg[1024];
1216
1217     if (zhp->zpool_state == POOL_STATE_ACTIVE &&
1218         (zfp = zfs_open(hdl, zhp->zpool_name, ZFS_TYPE_FILESYSTEM)) == NULL)
1219         return (-1);
1220
1221     (void) strncpy(zc.zc_name, zhp->zpool_name, sizeof (zc.zc_name));
1222     zc.zc_history = (uint64_t)(uintptr_t)log_str;
1223 #endif /* !codereview */
1224
1225     if (zfs_ioctl(hdl, ZFS_IOC_POOL_DESTROY, &zc) != 0) {
1226         (void) snprintf(msg, sizeof (msg), dgettext(TEXT_DOMAIN,
1227             "cannot destroy '%s'", zhp->zpool_name));
1228
1229         if (errno == EROFS) {
1230             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1231                 "one or more devices is read only"));
1232             (void) zfs_error(hdl, EZFS_BADDEV, msg);
1233         } else {
1234             (void) zpool_standard_error(hdl, errno, msg);
1235         }
1236     }
1237
1238     if (zfp)
1239         zfs_close(zfp);
1240     return (-1);

```

```

1242     if (zfp) {
1243         remove_mountpoint(zfp);
1244         zfs_close(zfp);
1245     }
1246
1247     return (0);
1248 }
1249
1250 /*
1251  * Add the given vdevs to the pool. The caller must have already performed the
1252  * necessary verification to ensure that the vdev specification is well-formed.
1253  */
1254 int
1255 zpool_add(zpool_handle_t *zhp, nvlist_t *nvroot)
1256 {
1257     zfs_cmd_t zc = { 0 };
1258     int ret;
1259     libzfs_handle_t *hdl = zhp->zpool_hdl;
1260     char msg[1024];
1261     nvlist_t **spares, **l2cache;
1262     uint_t nspares, nl2cache;
1263
1264     (void) snprintf(msg, sizeof (msg), dgettext(TEXT_DOMAIN,
1265         "cannot add to '%s'", zhp->zpool_name));
1266
1267     if (zpool_get_prop_int(zhp, ZPOOL_PROP_VERSION, NULL) <
1268         SPA_VERSION_SPARES &&
1269         nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
1270             &spares, &nspares) == 0) {
1271         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "pool must be "
1272             "upgraded to add hot spares"));
1273         return (zfs_error(hdl, EZFS_BADVERSION, msg));
1274     }
1275
1276     if (zpool_is_bootable(zhp) && nvlist_lookup_nvlist_array(nvroot,
1277         ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0) {
1278         uint64_t s;
1279
1280         for (s = 0; s < nspares; s++) {
1281             char *path;
1282
1283             if (nvlist_lookup_string(spares[s], ZPOOL_CONFIG_PATH,
1284                 &path) == 0 && pool_uses_efi(spares[s])) {
1285                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1286                     "device '%s' contains an EFI label and "
1287                     "cannot be used on root pools."),
1288                     zpool_vdev_name(hdl, NULL, spares[s],
1289                         B_FALSE));
1289                 return (zfs_error(hdl, EZFS_POOL_NOTSUP, msg));
1290             }
1291         }
1292     }
1293
1294     if (zpool_get_prop_int(zhp, ZPOOL_PROP_VERSION, NULL) <
1295         SPA_VERSION_L2CACHE &&
1296         nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
1297             &l2cache, &nl2cache) == 0) {
1298         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "pool must be "
1299             "upgraded to add cache devices"));
1300         return (zfs_error(hdl, EZFS_BADVERSION, msg));
1301     }
1302
1303     if (zcmd_write_conf_nvlist(hdl, &zc, nvroot) != 0)
1304         return (-1);
1305     (void) strncpy(zc.zc_name, zhp->zpool_name, sizeof (zc.zc_name));

```

```

1308     if (zfs_ioctl(hdl, ZFS_IOC_VDEV_ADD, &zc) != 0) {
1309         switch (errno) {
1310             case EBUSY:
1311                 /*
1312                  * This can happen if the user has specified the same
1313                  * device multiple times. We can't reliably detect this
1314                  * until we try to add it and see we already have a
1315                  * label.
1316                  */
1317                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1318                     "one or more vdevs refer to the same device"));
1319                 (void) zfs_error(hdl, EZFS_BADDEV, msg);
1320                 break;
1321
1322             case EOVERFLOW:
1323                 /*
1324                  * This occurs when one of the devices is below
1325                  * SPA_MINDEVSZ. Unfortunately, we can't detect which
1326                  * device was the problem device since there's no
1327                  * reliable way to determine device size from userland.
1328                  */
1329                 {
1330                     char buf[64];
1331
1332                     zfs_nicenum(SPA_MINDEVSZ, buf, sizeof (buf));
1333
1334                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1335                         "device is less than the minimum "
1336                         "size (%s)"), buf);
1337                 }
1338                 (void) zfs_error(hdl, EZFS_BADDEV, msg);
1339                 break;
1340
1341             case ENOTSUP:
1342                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1343                     "pool must be upgraded to add these vdevs"));
1344                 (void) zfs_error(hdl, EZFS_BADVERSION, msg);
1345                 break;
1346
1347             case EDOM:
1348                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1349                     "root pool can not have multiple vdevs"
1350                     " or separate logs"));
1351                 (void) zfs_error(hdl, EZFS_POOL_NOTSUP, msg);
1352                 break;
1353
1354             case ENOTBLK:
1355                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1356                     "cache device must be a disk or disk slice"));
1357                 (void) zfs_error(hdl, EZFS_BADDEV, msg);
1358                 break;
1359
1360             default:
1361                 (void) zpool_standard_error(hdl, errno, msg);
1362         }
1363
1364         ret = -1;
1365     } else {
1366         ret = 0;
1367     }
1368
1369     zcmd_free_nvlists(&zc);
1370
1371     return (ret);
1372 }

```

```

1374 /*
1375  * Exports the pool from the system. The caller must ensure that there are no
1376  * mounted datasets in the pool.
1377  */
1378 static int
1379 zpool_export_common(zpool_handle_t *zhp, boolean_t force, boolean_t hardforce,
1380     const char *log_str)
1381 {
1382     49 int
1383     50 zpool_export_common(zpool_handle_t *zhp, boolean_t force, boolean_t hardforce)
1384     {
1385         zfs_cmd_t zc = { 0 };
1386         char msg[1024];
1387
1388         (void) snprintf(msg, sizeof (msg), dgettext(TEXT_DOMAIN,
1389             "cannot export '%s'"), zhp->zpool_name);
1390
1391         (void) strncpy(zc.zc_name, zhp->zpool_name, sizeof (zc.zc_name));
1392         zc.zc_cookie = force;
1393         zc.zc_guid = hardforce;
1394         zc.zc_history = (uint64_t)(uintptr_t)log_str;
1395     #endif /* !codereview */
1396
1397     if (zfs_ioctl(zhp->zpool_hdl, ZFS_IOC_POOL_EXPORT, &zc) != 0) {
1398         switch (errno) {
1399             case EXDEV:
1400                 zfs_error_aux(zhp->zpool_hdl, dgettext(TEXT_DOMAIN,
1401                     "use '-f' to override the following errors:\n"
1402                     "'%s' has an active shared spare which could be"
1403                     " used by other pools once '%s' is exported."),
1404                     zhp->zpool_name, zhp->zpool_name);
1405                 return (zfs_error(zhp->zpool_hdl, EZFS_ACTIVE_SPARE,
1406                     msg));
1407             default:
1408                 return (zpool_standard_error_fmt(zhp->zpool_hdl, errno,
1409                     msg));
1410         }
1411     }
1412     return (0);
1413 }
1414
1415 int
1416 zpool_export(zpool_handle_t *zhp, boolean_t force, const char *log_str)
1417 {
1418     61 zpool_export(zpool_handle_t *zhp, boolean_t force)
1419     {
1420         return (zpool_export_common(zhp, force, B_FALSE, log_str));
1421     }
1422     63     return (zpool_export_common(zhp, force, B_FALSE));
1423 }
1424
1425 int
1426 zpool_export_force(zpool_handle_t *zhp, const char *log_str)
1427 {
1428     67 zpool_export_force(zpool_handle_t *zhp)
1429     {
1430         return (zpool_export_common(zhp, B_TRUE, B_TRUE, log_str));
1431     }
1432     69     return (zpool_export_common(zhp, B_TRUE, B_TRUE));
1433 }
1434
1435 _____unchanged_portion_omitted_____
1436
1437 void
1438 zfs_save_arguments(int argc, char **argv, char *string, int len)
1439 {
1440     zpool_set_history_str(const char *subcommand, int argc, char **argv,
1441         char *history_str)
1442     {
1443         (void) strncpy(string, basename(argv[0]), len);
1444         for (int i = 1; i < argc; i++) {
1445             (void) strncat(string, " ", len);

```

```
3589         (void) strlcat(string, argv[i], len);
2234     int i;

2236     (void) strcpy(history_str, subcommand, HIS_MAX_RECORD_LEN);
2237     for (i = 1; i < argc; i++) {
2238         if (strlen(history_str) + 1 + strlen(argv[i]) >
2239             HIS_MAX_RECORD_LEN)
2240             break;
2241         (void) strlcat(history_str, " ", HIS_MAX_RECORD_LEN);
2242         (void) strlcat(history_str, argv[i], HIS_MAX_RECORD_LEN);
3590     }
3591 }

2246 /*
2247  * Stage command history for logging.
2248  */
3593 int
3594 zpool_log_history(libzfs_handle_t *hdl, const char *message)
2250 zpool_stage_history(libzfs_handle_t *hdl, const char *history_str)
3595 {
3596     zfs_cmd_t zc = { 0 };
3597     nvlist_t *args;
3598     int err;
2252     if (history_str == NULL)
2253         return (EINVAL);

3600     args = fnvlist_alloc();
3601     fnvlist_add_string(args, "message", message);
3602     err = zcmd_write_src_nvlist(hdl, &zc, args);
3603     if (err == 0)
3604         err = ioctl(hdl->libzfs_fd, ZFS_IOC_LOG_HISTORY, &zc);
3605     nvlist_free(args);
3606     zcmd_free_nvlists(&zc);
3607     return (err);
2255     if (strlen(history_str) > HIS_MAX_RECORD_LEN)
2256         return (EINVAL);

2258     if (hdl->libzfs_log_str != NULL)
2259         free(hdl->libzfs_log_str);

2261     if ((hdl->libzfs_log_str = strdup(history_str)) == NULL)
2262         return (no_memory(hdl));

2264     return (0);
3608 }

    unchanged_portion_omitted_
```

```

*****
36060 Thu Jun 28 15:57:11 2012
new/usr/src/lib/libzfs/common/libzfs_util.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2012 by Delphix. All rights reserved.
25  */
26
27 /*
28  * Internal utility routines for the ZFS library.
29  */
30
31 #include <errno.h>
32 #include <fcntl.h>
33 #include <libintl.h>
34 #include <stdarg.h>
35 #include <stdio.h>
36 #include <stdlib.h>
37 #include <strings.h>
38 #include <unistd.h>
39 #include <ctype.h>
40 #include <math.h>
41 #include <sys/mnttab.h>
42 #include <sys/mntent.h>
43 #include <sys/types.h>
44
45 #include <libzfs.h>
46 #include <libzfs_core.h>
47 #endif /* ! codereview */
48
49 #include "libzfs_impl.h"
50 #include "zfs_prop.h"
51 #include "zfeature_common.h"
52
53 int
54 libzfs_errno(libzfs_handle_t *hdl)

```

```

55 {
56     return (hdl->libzfs_error);
57 }
58
59 const char *
60 libzfs_error_action(libzfs_handle_t *hdl)
61 {
62     return (hdl->libzfs_action);
63 }
64
65 const char *
66 libzfs_error_description(libzfs_handle_t *hdl)
67 {
68     if (hdl->libzfs_desc[0] != '\0')
69         return (hdl->libzfs_desc);
70
71     switch (hdl->libzfs_error) {
72     case EZFS_NOMEM:
73         return (dgettext(TEXT_DOMAIN, "out of memory"));
74     case EZFS_BADPROP:
75         return (dgettext(TEXT_DOMAIN, "invalid property value"));
76     case EZFS_PROPREADONLY:
77         return (dgettext(TEXT_DOMAIN, "read-only property"));
78     case EZFS_PROPTYPE:
79         return (dgettext(TEXT_DOMAIN, "property doesn't apply to "
80             "datasets of this type"));
81     case EZFS_PROPNONINHERIT:
82         return (dgettext(TEXT_DOMAIN, "property cannot be inherited"));
83     case EZFS_PROPSPACE:
84         return (dgettext(TEXT_DOMAIN, "invalid quota or reservation"));
85     case EZFS_BADTYPE:
86         return (dgettext(TEXT_DOMAIN, "operation not applicable to "
87             "datasets of this type"));
88     case EZFS_BUSY:
89         return (dgettext(TEXT_DOMAIN, "pool or dataset is busy"));
90     case EZFS_EXISTS:
91         return (dgettext(TEXT_DOMAIN, "pool or dataset exists"));
92     case EZFS_NOENT:
93         return (dgettext(TEXT_DOMAIN, "no such pool or dataset"));
94     case EZFS_BADSTREAM:
95         return (dgettext(TEXT_DOMAIN, "invalid backup stream"));
96     case EZFS_DSREADONLY:
97         return (dgettext(TEXT_DOMAIN, "dataset is read-only"));
98     case EZFS_VOLTOOBIG:
99         return (dgettext(TEXT_DOMAIN, "volume size exceeds limit for "
100             "this system"));
101     case EZFS_INVALIDNAME:
102         return (dgettext(TEXT_DOMAIN, "invalid name"));
103     case EZFS_BADRESTORE:
104         return (dgettext(TEXT_DOMAIN, "unable to restore to "
105             "destination"));
106     case EZFS_BADBACKUP:
107         return (dgettext(TEXT_DOMAIN, "backup failed"));
108     case EZFS_BADTARGET:
109         return (dgettext(TEXT_DOMAIN, "invalid target vdev"));
110     case EZFS_NODEVICE:
111         return (dgettext(TEXT_DOMAIN, "no such device in pool"));
112     case EZFS_BADDEV:
113         return (dgettext(TEXT_DOMAIN, "invalid device"));
114     case EZFS_NOREPLICAS:
115         return (dgettext(TEXT_DOMAIN, "no valid replicas"));
116     case EZFS_RESILVERING:
117         return (dgettext(TEXT_DOMAIN, "currently resilvering"));
118     case EZFS_BADVERSION:
119         return (dgettext(TEXT_DOMAIN, "unsupported version or "
120             "feature"));

```

```

121 case EZFS_POOLUNAVAIL:
122     return (dgettext(TEXT_DOMAIN, "pool is unavailable"));
123 case EZFS_DEVOVERFLOW:
124     return (dgettext(TEXT_DOMAIN, "too many devices in one vdev"));
125 case EZFS_BADPATH:
126     return (dgettext(TEXT_DOMAIN, "must be an absolute path"));
127 case EZFS_CROSSTARGET:
128     return (dgettext(TEXT_DOMAIN, "operation crosses datasets or "
129 "pools"));
130 case EZFS_ZONED:
131     return (dgettext(TEXT_DOMAIN, "dataset in use by local zone"));
132 case EZFS_MOUNTFAILED:
133     return (dgettext(TEXT_DOMAIN, "mount failed"));
134 case EZFS_UMOUNTFAILED:
135     return (dgettext(TEXT_DOMAIN, "umount failed"));
136 case EZFS_UNSHARENFSFAILED:
137     return (dgettext(TEXT_DOMAIN, "unshare(1M) failed"));
138 case EZFS_SHARENFSFAILED:
139     return (dgettext(TEXT_DOMAIN, "share(1M) failed"));
140 case EZFS_UNSHARESMBFAILED:
141     return (dgettext(TEXT_DOMAIN, "smb remove share failed"));
142 case EZFS_SHARESMBFAILED:
143     return (dgettext(TEXT_DOMAIN, "smb add share failed"));
144 case EZFS_PERM:
145     return (dgettext(TEXT_DOMAIN, "permission denied"));
146 case EZFS_NOSPC:
147     return (dgettext(TEXT_DOMAIN, "out of space"));
148 case EZFS_FAULT:
149     return (dgettext(TEXT_DOMAIN, "bad address"));
150 case EZFS_IO:
151     return (dgettext(TEXT_DOMAIN, "I/O error"));
152 case EZFS_INTR:
153     return (dgettext(TEXT_DOMAIN, "signal received"));
154 case EZFS_ISSPARE:
155     return (dgettext(TEXT_DOMAIN, "device is reserved as a hot "
156 "spare"));
157 case EZFS_INVALIDCONFIG:
158     return (dgettext(TEXT_DOMAIN, "invalid vdev configuration"));
159 case EZFS_RECURSIVE:
160     return (dgettext(TEXT_DOMAIN, "recursive dataset dependency"));
161 case EZFS_NOHISTORY:
162     return (dgettext(TEXT_DOMAIN, "no history available"));
163 case EZFS_POOLPROPS:
164     return (dgettext(TEXT_DOMAIN, "failed to retrieve "
165 "pool properties"));
166 case EZFS_POOL_NOTSUP:
167     return (dgettext(TEXT_DOMAIN, "operation not supported "
168 "on this type of pool"));
169 case EZFS_POOL_INVALIDARG:
170     return (dgettext(TEXT_DOMAIN, "invalid argument for "
171 "this pool operation"));
172 case EZFS_NAMETOOLONG:
173     return (dgettext(TEXT_DOMAIN, "dataset name is too long"));
174 case EZFS_OPENFAILED:
175     return (dgettext(TEXT_DOMAIN, "open failed"));
176 case EZFS_NOCAP:
177     return (dgettext(TEXT_DOMAIN,
178 "disk capacity information could not be retrieved"));
179 case EZFS_LABELFAILED:
180     return (dgettext(TEXT_DOMAIN, "write of label failed"));
181 case EZFS_BADWHO:
182     return (dgettext(TEXT_DOMAIN, "invalid user/group"));
183 case EZFS_BADPERM:
184     return (dgettext(TEXT_DOMAIN, "invalid permission"));
185 case EZFS_BADPERMSET:
186     return (dgettext(TEXT_DOMAIN, "invalid permission set name"));

```

```

187 case EZFS_NODELEGATION:
188     return (dgettext(TEXT_DOMAIN, "delegated administration is "
189 "disabled on pool"));
190 case EZFS_BADCACHE:
191     return (dgettext(TEXT_DOMAIN, "invalid or missing cache file"));
192 case EZFS_ISL2CACHE:
193     return (dgettext(TEXT_DOMAIN, "device is in use as a cache"));
194 case EZFS_VDEVNOTSUP:
195     return (dgettext(TEXT_DOMAIN, "vdev specification is not "
196 "supported"));
197 case EZFS_NOTSUP:
198     return (dgettext(TEXT_DOMAIN, "operation not supported "
199 "on this dataset"));
200 case EZFS_ACTIVE_SPARE:
201     return (dgettext(TEXT_DOMAIN, "pool has active shared spare "
202 "device"));
203 case EZFS_UNPLAYED_LOGS:
204     return (dgettext(TEXT_DOMAIN, "log device has unplayed intent "
205 "logs"));
206 case EZFS_REFTAG_RELE:
207     return (dgettext(TEXT_DOMAIN, "no such tag on this dataset"));
208 case EZFS_REFTAG_HOLD:
209     return (dgettext(TEXT_DOMAIN, "tag already exists on this "
210 "dataset"));
211 case EZFS_TAGTOOLONG:
212     return (dgettext(TEXT_DOMAIN, "tag too long"));
213 case EZFS_PIPEFAILED:
214     return (dgettext(TEXT_DOMAIN, "pipe create failed"));
215 case EZFS_THREADCREATEFAILED:
216     return (dgettext(TEXT_DOMAIN, "thread create failed"));
217 case EZFS_POSTSPLIT_ONLINE:
218     return (dgettext(TEXT_DOMAIN, "disk was split from this pool "
219 "into a new one"));
220 case EZFS_SCRUBBING:
221     return (dgettext(TEXT_DOMAIN, "currently scrubbing; "
222 "use 'zpool scrub -s' to cancel current scrub"));
223 case EZFS_NO_SCRUB:
224     return (dgettext(TEXT_DOMAIN, "there is no active scrub"));
225 case EZFS_DIFF:
226     return (dgettext(TEXT_DOMAIN, "unable to generate diffs"));
227 case EZFS_DIFFDATA:
228     return (dgettext(TEXT_DOMAIN, "invalid diff data"));
229 case EZFS_POOLREADONLY:
230     return (dgettext(TEXT_DOMAIN, "pool is read-only"));
231 case EZFS_UNKNOWN:
232     return (dgettext(TEXT_DOMAIN, "unknown error"));
233 default:
234     assert(hdl->libzfs_error == 0);
235     return (dgettext(TEXT_DOMAIN, "no error"));
236 }
237 }
238
239 /*PRINTFLIKE2*/
240 void
241 zfs_error_aux(libzfs_handle_t *hdl, const char *fmt, ...)
242 {
243     va_list ap;
244
245     va_start(ap, fmt);
246
247     (void) vsnprintf(hdl->libzfs_desc, sizeof(hdl->libzfs_desc),
248 fmt, ap);
249     hdl->libzfs_desc_active = 1;
250
251     va_end(ap);
252 }

```

```

254 static void
255 zfs_verror(libzfs_handle_t *hdl, int error, const char *fmt, va_list ap)
256 {
257     (void) vsnprintf(hdl->libzfs_action, sizeof (hdl->libzfs_action),
258                     fmt, ap);
259     hdl->libzfs_error = error;
260
261     if (hdl->libzfs_desc_active)
262         hdl->libzfs_desc_active = 0;
263     else
264         hdl->libzfs_desc[0] = '\0';
265
266     if (hdl->libzfs_printerr) {
267         if (error == EZFS_UNKNOWN) {
268             (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "internal "
269             "error: %s\n"), libzfs_error_description(hdl));
270             abort();
271         }
272
273         (void) fprintf(stderr, "%s: %s\n", hdl->libzfs_action,
274                         libzfs_error_description(hdl));
275         if (error == EZFS_NOMEM)
276             exit(1);
277     }
278 }
279
280 int
281 zfs_error(libzfs_handle_t *hdl, int error, const char *msg)
282 {
283     return (zfs_error_fmt(hdl, error, "%s", msg));
284 }
285
286 /*PRINTFLIKE3*/
287 int
288 zfs_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
289 {
290     va_list ap;
291
292     va_start(ap, fmt);
293
294     zfs_verror(hdl, error, fmt, ap);
295
296     va_end(ap);
297
298     return (-1);
299 }
300
301 static int
302 zfs_common_error(libzfs_handle_t *hdl, int error, const char *fmt,
303                 va_list ap)
304 {
305     switch (error) {
306     case EPERM:
307     case EACCES:
308         zfs_verror(hdl, EZFS_PERM, fmt, ap);
309         return (-1);
310
311     case ECANCELED:
312         zfs_verror(hdl, EZFS_NODELEGATION, fmt, ap);
313         return (-1);
314
315     case EIO:
316         zfs_verror(hdl, EZFS_IO, fmt, ap);
317         return (-1);

```

```

319     case EFAULT:
320         zfs_verror(hdl, EZFS_FAULT, fmt, ap);
321         return (-1);
322
323     case EINTR:
324         zfs_verror(hdl, EZFS_INTR, fmt, ap);
325         return (-1);
326     }
327
328     return (0);
329 }
330
331 int
332 zfs_standard_error(libzfs_handle_t *hdl, int error, const char *msg)
333 {
334     return (zfs_standard_error_fmt(hdl, error, "%s", msg));
335 }
336
337 /*PRINTFLIKE3*/
338 int
339 zfs_standard_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
340 {
341     va_list ap;
342
343     va_start(ap, fmt);
344
345     if (zfs_common_error(hdl, error, fmt, ap) != 0) {
346         va_end(ap);
347         return (-1);
348     }
349
350     switch (error) {
351     case ENXIO:
352     case ENODEV:
353     case EPIPE:
354         zfs_verror(hdl, EZFS_IO, fmt, ap);
355         break;
356
357     case ENOENT:
358         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
359         "dataset does not exist"));
360         zfs_verror(hdl, EZFS_NOENT, fmt, ap);
361         break;
362
363     case ENOSPC:
364     case EDQUOT:
365         zfs_verror(hdl, EZFS_NOSPC, fmt, ap);
366         return (-1);
367
368     case EEXIST:
369         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
370         "dataset already exists"));
371         zfs_verror(hdl, EZFS_EXISTS, fmt, ap);
372         break;
373
374     case EBUSY:
375         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
376         "dataset is busy"));
377         zfs_verror(hdl, EZFS_BUSY, fmt, ap);
378         break;
379
380     case EROFS:
381         zfs_verror(hdl, EZFS_POOLREADONLY, fmt, ap);
382         break;
383
384     case ENAMETOOLONG:
385         zfs_verror(hdl, EZFS_NAMETOOLONG, fmt, ap);
386         break;

```

```

385     case ENOTSUP:
386         zfs_verror(hdl, EZFS_BADVERSION, fmt, ap);
387         break;
388     case EAGAIN:
389         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
390             "pool I/O is currently suspended"));
391         zfs_verror(hdl, EZFS_POOLUNAVAIL, fmt, ap);
392         break;
393     default:
394         zfs_error_aux(hdl, strerror(error));
395         zfs_verror(hdl, EZFS_UNKNOWN, fmt, ap);
396         break;
397     }
399     va_end(ap);
400     return (-1);
401 }

403 int
404 zpool_standard_error(libzfs_handle_t *hdl, int error, const char *msg)
405 {
406     return (zpool_standard_error_fmt(hdl, error, "%s", msg));
407 }

409 /*PRINTFLIKE3*/
410 int
411 zpool_standard_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
412 {
413     va_list ap;
414
415     va_start(ap, fmt);
416
417     if (zfs_common_error(hdl, error, fmt, ap) != 0) {
418         va_end(ap);
419         return (-1);
420     }
421
422     switch (error) {
423     case ENODEV:
424         zfs_verror(hdl, EZFS_NODEVICE, fmt, ap);
425         break;
426
427     case ENOENT:
428         zfs_error_aux(hdl,
429             dgettext(TEXT_DOMAIN, "no such pool or dataset"));
430         zfs_verror(hdl, EZFS_NOENT, fmt, ap);
431         break;
432
433     case EEXIST:
434         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
435             "pool already exists"));
436         zfs_verror(hdl, EZFS_EXISTS, fmt, ap);
437         break;
438
439     case EBUSY:
440         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "pool is busy"));
441         zfs_verror(hdl, EZFS_BUSY, fmt, ap);
442         break;
443
444     case ENXIO:
445         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
446             "one or more devices is currently unavailable"));
447         zfs_verror(hdl, EZFS_BADDEV, fmt, ap);
448         break;
449
450     case ENAMETOOLONG:

```

```

451         zfs_verror(hdl, EZFS_DEVOVERFLOW, fmt, ap);
452         break;
453
454     case ENOTSUP:
455         zfs_verror(hdl, EZFS_POOL_NOTSUP, fmt, ap);
456         break;
457
458     case EINVAL:
459         zfs_verror(hdl, EZFS_POOL_INVALIDARG, fmt, ap);
460         break;
461
462     case ENOSPC:
463     case EDQUOT:
464         zfs_verror(hdl, EZFS_NOSPC, fmt, ap);
465         return (-1);
466
467     case EAGAIN:
468         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
469             "pool I/O is currently suspended"));
470         zfs_verror(hdl, EZFS_POOLUNAVAIL, fmt, ap);
471         break;
472
473     case EROFS:
474         zfs_verror(hdl, EZFS_POOLREADONLY, fmt, ap);
475         break;
476
477     default:
478         zfs_error_aux(hdl, strerror(error));
479         zfs_verror(hdl, EZFS_UNKNOWN, fmt, ap);
480     }
481
482     va_end(ap);
483     return (-1);
484 }

486 /*
487  * Display an out of memory error message and abort the current program.
488  */
489 int
490 no_memory(libzfs_handle_t *hdl)
491 {
492     return (zfs_error(hdl, EZFS_NOMEM, "internal error"));
493 }

495 /*
496  * A safe form of malloc() which will die if the allocation fails.
497  */
498 void *
499 zfs_alloc(libzfs_handle_t *hdl, size_t size)
500 {
501     void *data;
502
503     if ((data = calloc(1, size)) == NULL)
504         (void) no_memory(hdl);
505
506     return (data);
507 }

509 /*
510  * A safe form of asprintf() which will die if the allocation fails.
511  */
512 /*PRINTFLIKE2*/
513 char *
514 zfs_asprintf(libzfs_handle_t *hdl, const char *fmt, ...)
515 {
516     va_list ap;

```



```

517     char *ret;
518     int err;

520     va_start(ap, fmt);

522     err = vasprintf(&ret, fmt, ap);

524     va_end(ap);

526     if (err < 0)
527         (void) no_memory(hdl);

529     return (ret);
530 }

532 /*
533  * A safe form of realloc(), which also zeroes newly allocated space.
534  */
535 void *
536 zfs_realloc(libzfs_handle_t *hdl, void *ptr, size_t oldsize, size_t newsize)
537 {
538     void *ret;

540     if ((ret = realloc(ptr, newsize)) == NULL) {
541         (void) no_memory(hdl);
542         return (NULL);
543     }

545     bzero((char *)ret + oldsize, (newsize - oldsize));
546     return (ret);
547 }

549 /*
550  * A safe form of strdup() which will die if the allocation fails.
551  */
552 char *
553 zfs_strdup(libzfs_handle_t *hdl, const char *str)
554 {
555     char *ret;

557     if ((ret = strdup(str)) == NULL)
558         (void) no_memory(hdl);

560     return (ret);
561 }

563 /*
564  * Convert a number to an appropriately human-readable output.
565  */
566 void
567 zfs_nicenum(uint64_t num, char *buf, size_t buflen)
568 {
569     uint64_t n = num;
570     int index = 0;
571     char u;

573     while (n >= 1024) {
574         n /= 1024;
575         index++;
576     }

578     u = " KMGTPe"[index];

580     if (index == 0) {
581         (void) snprintf(buf, buflen, "%llu", n);
582     } else if ((num & ((LULL << 10 * index) - 1)) == 0) {

```

```

583         /*
584          * If this is an even multiple of the base, always display
585          * without any decimal precision.
586          */
587         (void) snprintf(buf, buflen, "%llu%c", n, u);
588     } else {
589         /*
590          * We want to choose a precision that reflects the best choice
591          * for fitting in 5 characters. This can get rather tricky when
592          * we have numbers that are very close to an order of magnitude.
593          * For example, when displaying 10239 (which is really 9.999K),
594          * we want only a single place of precision for 10.0K. We could
595          * develop some complex heuristics for this, but it's much
596          * easier just to try each combination in turn.
597          */
598         int i;
599         for (i = 2; i >= 0; i--) {
600             if (snprintf(buf, buflen, "%.*f%c", i,
601                 (double)num / (LULL << 10 * index), u) <= 5)
602                 break;
603         }
604     }
605 }

607 void
608 libzfs_print_on_error(libzfs_handle_t *hdl, boolean_t printerr)
609 {
610     hdl->libzfs_printerr = printerr;
611 }

613 libzfs_handle_t *
614 libzfs_init(void)
615 {
616     libzfs_handle_t *hdl;

618     if ((hdl = calloc(1, sizeof (libzfs_handle_t))) == NULL) {
619         return (NULL);
620     }

622     if ((hdl->libzfs_fd = open(ZFS_DEV, O_RDWR)) < 0) {
623         free(hdl);
624         return (NULL);
625     }

627     if ((hdl->libzfs_mnttab = fopen(MNTTAB, "r")) == NULL) {
628         (void) close(hdl->libzfs_fd);
629         free(hdl);
630         return (NULL);
631     }

633     hdl->libzfs_sharetab = fopen("/etc/dfs/sharetab", "r");

635     if (libzfs_core_init() != 0) {
636         (void) close(hdl->libzfs_fd);
637         (void) fclose(hdl->libzfs_mnttab);
638         (void) fclose(hdl->libzfs_sharetab);
639         free(hdl);
640         return (NULL);
641     }

643 #endif /* ! codereview */
644     zfs_prop_init();
645     zpool_prop_init();
646     zpool_feature_init();
647     libzfs_mnttab_init(hdl);

```

```

649     return (hdl);
650 }

652 void
653 libzfs_fini(libzfs_handle_t *hdl)
654 {
655     (void) close(hdl->libzfs_fd);
656     if (hdl->libzfs_mnttab)
657         (void) fclose(hdl->libzfs_mnttab);
658     if (hdl->libzfs_sharetab)
659         (void) fclose(hdl->libzfs_sharetab);
660     zfs_uninit_libshare(hdl);
661     if (hdl->libzfs_log_str)
662         (void) free(hdl->libzfs_log_str);
663     zpool_free_handles(hdl);
664     libzfs_fru_clear(hdl, B_TRUE);
665     namespace_clear(hdl);
666     libzfs_mnttab_fini(hdl);
667     libzfs_core_fini();
668 #endif /* ! codereview */
669     free(hdl);
670 }

671 libzfs_handle_t *
672 zpool_get_handle(zpool_handle_t *zhp)
673 {
674     return (zhp->zpool_hdl);
675 }

676 libzfs_handle_t *
677 zfs_get_handle(zfs_handle_t *zhp)
678 {
679     return (zhp->zfs_hdl);
680 }

681 zpool_handle_t *
682 zfs_get_pool_handle(const zfs_handle_t *zhp)
683 {
684     return (zhp->zpool_hdl);
685 }

686 /*
687  * Given a name, determine whether or not it's a valid path
688  * (starts with '/' or "/"). If so, walk the mnttab trying
689  * to match the device number. If not, treat the path as an
690  * fs/vol/snap name.
691  */
692 zfs_handle_t *
693 zfs_path_to_zhandle(libzfs_handle_t *hdl, char *path, zfs_type_t argtype)
694 {
695     struct stat64 statbuf;
696     struct extmnttab entry;
697     int ret;

698     if (path[0] != '/' && strcmp(path, "./", strlen("./")) != 0) {
699         /*
700          * It's not a valid path, assume it's a name of type 'argtype'.
701          */
702         return (zfs_open(hdl, path, argtype));
703     }

704     if (stat64(path, &statbuf) != 0) {
705         (void) fprintf(stderr, "%s: %s\n", path, strerror(errno));
706         return (NULL);
707     }
708 }

```

```

713     rewind(hdl->libzfs_mnttab);
714     while ((ret = getextmntent(hdl->libzfs_mnttab, &entry, 0)) == 0) {
715         if (makedevice(entry.mnt_major, entry.mnt_minor) ==
716             statbuf.st_dev) {
717             break;
718         }
719     }
720     if (ret != 0) {
721         return (NULL);
722     }

723     if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0) {
724         (void) fprintf(stderr, gettext("%s': not a ZFS filesystem\n"),
725             path);
726         return (NULL);
727     }

728     return (zfs_open(hdl, entry.mnt_special, ZFS_TYPE_FILESYSTEM));
729 }

730 /*
731  * Initialize the zc_nvlist_dst member to prepare for receiving an nvlist from
732  * an ioctl().
733  */
734 int
735 zcmd_alloc_dst_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, size_t len)
736 {
737     if (len == 0)
738         len = 16 * 1024;
739     zc->zc_nvlist_dst_size = len;
740     if ((zc->zc_nvlist_dst = (uint64_t)(uintptr_t)
741         zfs_alloc(hdl, zc->zc_nvlist_dst_size)) == NULL)
742         return (-1);

743     return (0);
744 }

745 /*
746  * Called when an ioctl() which returns an nvlist fails with ENOMEM. This will
747  * expand the nvlist to the size specified in 'zc_nvlist_dst_size', which was
748  * filled in by the kernel to indicate the actual required size.
749  */
750 int
751 zcmd_expand_dst_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc)
752 {
753     free((void *) (uintptr_t) zc->zc_nvlist_dst);
754     if ((zc->zc_nvlist_dst = (uint64_t)(uintptr_t)
755         zfs_alloc(hdl, zc->zc_nvlist_dst_size)) == NULL)
756         return (-1);

757     return (0);
758 }

759 /*
760  * Called to free the src and dst nvlists stored in the command structure.
761  */
762 void
763 zcmd_free_nvlists(zfs_cmd_t *zc)
764 {
765     free((void *) (uintptr_t) zc->zc_nvlist_conf);
766     free((void *) (uintptr_t) zc->zc_nvlist_src);
767     free((void *) (uintptr_t) zc->zc_nvlist_dst);
768 }

769 static int

```

```

779 zcmd_write_nvlist_com(libzfs_handle_t *hdl, uint64_t *outnv, uint64_t *outlen,
780     nvlist_t *nvl)
781 {
782     char *packed;
783     size_t len;
784
785     verify(nvlist_size(nvl, &len, NV_ENCODE_NATIVE) == 0);
786
787     if ((packed = zfs_alloc(hdl, len)) == NULL)
788         return (-1);
789
790     verify(nvlist_pack(nvl, &packed, &len, NV_ENCODE_NATIVE, 0) == 0);
791
792     *outnv = (uint64_t)(uintptr_t)packed;
793     *outlen = len;
794
795     return (0);
796 }
797
798 int
799 zcmd_write_conf_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, nvlist_t *nvl)
800 {
801     return (zcmd_write_nvlist_com(hdl, &zc->zc_nvlist_conf,
802     &zc->zc_nvlist_conf_size, nvl));
803 }
804
805 int
806 zcmd_write_src_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, nvlist_t *nvl)
807 {
808     return (zcmd_write_nvlist_com(hdl, &zc->zc_nvlist_src,
809     &zc->zc_nvlist_src_size, nvl));
810 }
811
812 /*
813  * Unpacks an nvlist from the ZFS ioctl command structure.
814  */
815 int
816 zcmd_read_dst_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, nvlist_t **nvlp)
817 {
818     if (nvlist_unpack((void *) (uintptr_t) zc->zc_nvlist_dst,
819     zc->zc_nvlist_dst_size, nvlp, 0) != 0)
820         return (no_memory(hdl));
821
822     return (0);
823 }
824
825 int
826 zfs_ioctl(libzfs_handle_t *hdl, int request, zfs_cmd_t *zc)
827 {
828     return (ioctl(hdl->libzfs_fd, request, zc));
829     int error;
830
831     zc->zc_history = (uint64_t)(uintptr_t)hdl->libzfs_log_str;
832     error = ioctl(hdl->libzfs_fd, request, zc);
833     if (hdl->libzfs_log_str) {
834         free(hdl->libzfs_log_str);
835         hdl->libzfs_log_str = NULL;
836     }
837     zc->zc_history = 0;
838
839     return (error);
840 }

```

_____unchanged_portion_omitted_____

new/usr/src/lib/libzfs/common/l1ib-lzfs

1

1312 Thu Jun 28 15:57:12 2012

new/usr/src/lib/libzfs/common/l1ib-lzfs

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Copyright 2010 Nexenta Systems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */
27 /*
28 * Copyright (c) 2012 by Delphix. All rights reserved.
29 */
31 /*LINTLIBRARY*/
32 /*PROTOLIB1*/
34 #include <libzfs.h>
35 #include <libzfs_core.h>
36 #endif /* ! codereview */
37 #include "../common/zfs/zfs_comutil.h"
38 #include "../common/zfs/zfs_fletcher.h"
39 #include "../common/zfs/zfs_prop.h"
40 #include "../common/zfs/zfeature_common.h"
```

```

*****
5362 Thu Jun 28 15:57:12 2012
new/usr/src/lib/libzfs/common/mapfile-vers
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 # Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
22 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
23 # Copyright (c) 2012 by Delphix. All rights reserved.
24 #
25 # MAPFILE HEADER START
26 #
27 # WARNING: STOP NOW. DO NOT MODIFY THIS FILE.
28 # Object versioning must comply with the rules detailed in
29 #
30 #     usr/src/lib/README.mapfiles
31 #
32 # You should not be making modifications here until you've read the most current
33 # copy of that file. If you need help, contact a gatekeeper for guidance.
34 #
35 # MAPFILE HEADER END
36 #

38 $mapfile_version 2

40 SYMBOL_VERSION SUNWprivate_1.1 {
41     global:
42         fletcher_2_native;
43         fletcher_2_byteswap;
44         fletcher_4_native;
45         fletcher_4_byteswap;
46         fletcher_4_incremental_native;
47         fletcher_4_incremental_byteswap;
48         libzfs_add_handle;
49         libzfs_dataset_cmp;
50         libzfs_errno;
51         libzfs_error_action;
52         libzfs_error_description;
53         libzfs_fini;
54         libzfs_fru_compare;

```

```

55     libzfs_fru_devpath;
56     libzfs_fru_lookup;
57     libzfs_fru_notself;
58     libzfs_fru_refresh;
59     libzfs_init;
60     libzfs_mnttab_cache;
61     libzfs_print_on_error;
62     spa_feature_table;
63     zfs_allocatable_devs;
64     zfs_asprintf;
65     zfs_clone;
66     zfs_close;
67     zfs_create;
68     zfs_create_ancestors;
69     zfs_dataset_exists;
70     zfs_deleg_share_nfs;
71     zfs_destroy;
72     zfs_destroy_snaps;
73     zfs_destroy_snaps_nvli;
74     zfs_expand_proplist;
75     zfs_get_handle;
76     zfs_get_holds;
77     zfs_get_name;
78     zfs_get_pool_handle;
79     zfs_get_snapused_int;
80     zfs_get_user_props;
81     zfs_get_type;
82     zfs_handle_dup;
83     zfs_history_event_names;
84     zfs_hold;
85     zfs_is_mounted;
86     zfs_is_shared;
87     zfs_is_shared_nfs;
88     zfs_is_shared_smb;
89     zfs_iter_children;
90     zfs_iter_dependents;
91     zfs_iter_filesystems;
92     zfs_iter_root;
93     zfs_iter_snapshots;
94     zfs_iter_snapshots_sorted;
95     zfs_iter_snapspec;
96     zfs_mount;
97     zfs_name_to_prop;
98     zfs_name_valid;
99     zfs_nicenum;
100    zfs_nicestrtonum;
101    zfs_open;
102    zfs_path_to_zhandle;
103    zfs_promote;
104    zfs_prop_align_right;
105    zfs_prop_column_name;
106    zfs_prop_default_numeric;
107    zfs_prop_default_string;
108    zfs_prop_get;
109    zfs_prop_get_int;
110    zfs_prop_get_numeric;
111    zfs_prop_get_recvd;
112    zfs_prop_get_table;
113    zfs_prop_get_userquota_int;
114    zfs_prop_get_userquota;
115    zfs_prop_get_written_int;
116    zfs_prop_get_written;
117    zfs_prop_inherit;
118    zfs_prop_inheritable;
119    zfs_prop_init;
120    zfs_prop_is_string;

```

```

120     zfs_prop_readonly;
121     zfs_prop_set;
122     zfs_prop_string_to_index;
123     zfs_prop_to_name;
124     zfs_prop_user;
125     zfs_prop_userquota;
126     zfs_prop_valid_for_type;
127     zfs_prop_values;
128     zfs_prop_written;
129     zfs_prune_proplist;
130     zfs_receive;
131     zfs_refresh_properties;
132     zfs_release;
133     zfs_rename;
134     zfs_rollback;
135     zfs_save_arguments;
136 #endif /* ! codereview */
137     zfs_send;
138     zfs_share;
139     zfs_shareall;
140     zfs_share_nfs;
141     zfs_share_smb;
142     zfs_show_diffs;
143     zfs_smb_acl_add;
144     zfs_smb_acl_purge;
145     zfs_smb_acl_remove;
146     zfs_smb_acl_rename;
147     zfs_snapshot;
148     zfs_snapshot_nvlist;
149 #endif /* ! codereview */
150     zfs_spa_version;
151     zfs_spa_version_map;
152     zfs_type_to_name;
153     zfs_unmount;
154     zfs_unmountall;
155     zfs_unshare;
156     zfs_unshare_nfs;
157     zfs_unshare_smb;
158     zfs_unshareall;
159     zfs_unshareall_bypath;
160     zfs_unshareall_nfs;
161     zfs_unshareall_smb;
162     zfs_userspace;
163     zfs_get_fsacl;
164     zfs_set_fsacl;
165     zfs_userquota_prop_prefixes;
166     zfs_zpl_version_map;
167     zpool_add;
168     zpool_clear;
169     zpool_clear_label;
170     zpool_close;
171     zpool_create;
172     zpool_destroy;
173     zpool_disable_datasets;
174     zpool_dump_ddt;
175     zpool_enable_datasets;
176     zpool_expand_proplist;
177     zpool_explain_recover;
178     zpool_export;
179     zpool_export_force;
180     zpool_find_import;
181     zpool_find_import_cached;
182     zpool_find_vdev;
183     zpool_find_vdev_by_physpath;
184     zpool_fru_set;
185     zpool_get_config;

```

```

186     zpool_get_errlog;
187     zpool_get_handle;
188     zpool_get_history;
189     zpool_get_name;
190     zpool_get_physpath;
191     zpool_get_prop;
192     zpool_get_prop_int;
193     zpool_get_state;
194     zpool_get_status;
195     zpool_history_unpack;
196     zpool_import;
197     zpool_import_props;
198     zpool_import_status;
199     zpool_in_use;
200     zpool_is_bootable;
201     zpool_iter;
202     zpool_label_disk;
203     zpool_log_history;
204 #endif /* ! codereview */
205     zpool_mount_datasets;
206     zpool_name_to_prop;
207     zpool_obj_to_path;
208     zpool_open;
209     zpool_open_canfail;
210     zpool_print_unsup_feat;
211     zpool_prop_align_right;
212     zpool_prop_column_name;
213     zpool_prop_feature;
214     zpool_prop_get_feature;
215     zpool_prop_readonly;
216     zpool_prop_to_name;
217     zpool_prop_unsupported;
218     zpool_prop_values;
219     zpool_read_label;
220     zpool_refresh_stats;
221     zpool_reguid;
222     zpool_reopen;
223     zpool_scan;
224     zpool_search_import;
225     zpool_set_history_str;
226     zpool_set_prop;
227     zpool_stage_history;
228     zpool_state_to_name;
229     zpool_unmount_datasets;
230     zpool_upgrade;
231     zpool_vdev_attach;
232     zpool_vdev_clear;
233     zpool_vdev_degrade;
234     zpool_vdev_detach;
235     zpool_vdev_fault;
236     zpool_vdev_name;
237     zpool_vdev_offline;
238     zpool_vdev_online;
239     zpool_vdev_remove;
240     zpool_vdev_split;
241     zprop_free_list;
242     zprop_get_list;
243     zprop_iter;
244     zprop_print_one_property;
245     zprop_width;
246     zvol_check_dump_config;
247     zvol_volsize_to_reservation;
248 };
    local:
    *;
    unchanged_portion_omitted

```

new/usr/src/lib/libzfs_core/Makefile

1

```
*****
1534 Thu Jun 28 15:57:12 2012
new/usr/src/lib/libzfs_core/Makefile
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
```

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright 2007 Sun Microsystems, Inc. All rights reserved.
23 # Use is subject to license terms.
24 #
25 # Copyright (c) 2012 by Delphix. All rights reserved.
26 #

28 include      ../Makefile.lib

30 HDRS=        libzfs_core.h

32 HDRDIR=      common

34 SUBDIRS=     $(MACH)
35 $(BUILD64)SUBDIRS += $(MACH64)

37 all :=       TARGET= all
38 clean :=     TARGET= clean
39 clobber :=   TARGET= clobber
40 install :=   TARGET= install
41 lint :=      TARGET= lint

43 MSGFILES =   '$(GREP) -l gettext $(HDRDIR)/*.ch\'
44 POFILE =     libzfs_core.po

46 .KEEP_STATE:

48 all clean clobber install lint: $(SUBDIRS)

50 $(POFILE):   pofile_MSGFILES

52 install_h:  $(ROOTHDRS)

54 check:      $(CHECKHDRS)
```

new/usr/src/lib/libzfs_core/Makefile

2

```
56 _msg: $(MSGDOMAINPOFILE)

58 $(SUBDIRS): FRC
59     @cd $@; pwd; $(MAKE) $(TARGET)

61 FRC:

63 include ../Makefile.targ
64 include ../../Makefile.msg.targ
65 #endif /* !codereview */
```

new/usr/src/lib/libzfs_core/Makefile.com

1

1743 Thu Jun 28 15:57:12 2012
new/usr/src/lib/libzfs_core/Makefile.com
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 # Copyright (c) 2012 by Delphix. All rights reserved.
24 #

26 LIBRARY= libzfs_core.a
27 VERS= .1

29 OBJS_SHARED=

31 OBJS_COMMON= \
32     libzfs_core.o

34 OBJECTS= $(OBJS_COMMON) $(OBJS_SHARED)

36 include ../../Makefile.lib

38 # libzfs_core must be installed in the root filesystem for mount(1M)
39 include ../../Makefile.rootfs

41 LIBS= $(DYNLIB) $(LINTLIB)

43 SRCDIR = ../common

45 INCS += -I$(SRCDIR)
46 INCS += -I../../uts/common/fs/zfs
47 INCS += -I../../common/zfs
48 INCS += -I../../libc/inc

50 C99MODE= -xc99=%all
51 C99LMODE= -Xc99=%all
52 LDLIBS += -lc -lnvpair
53 CPPFLAGS += $(INCS) -D_LARGEFILE64_SOURCE=1 -D_REENTRANT
```

new/usr/src/lib/libzfs_core/Makefile.com

2

```
55 SRCS= $(OBJS_COMMON:.o=$(SRCDIR)/%.c) \
56     $(OBJS_SHARED:.o=$(SRC)/common/zfs/%.c)
57 $(LINTLIB) := SRCS= $(SRCDIR)/$(LINTSRC)

59 .KEEP_STATE:

61 all: $(LIBS)

63 lint: lintcheck

65 pics%.o: ../../common/zfs/%.c
66     $(COMPILE.c) -o $@ $<
67     $(POST_PROCESS_O)

69 include ../../Makefile.targ
70 #endif /* !codereview */
```


new/usr/src/lib/libzfs_core/amd64/Makefile

1

1019 Thu Jun 28 15:57:12 2012

new/usr/src/lib/libzfs_core/amd64/Makefile

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License, Version 1.0 only
6 # (the "License"). You may not use this file except in compliance
7 # with the License.
8 #
9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 # or http://www.opensolaris.org/os/licensing.
11 # See the License for the specific language governing permissions
12 # and limitations under the License.
13 #
14 # When distributing Covered Code, include this CDDL HEADER in each
15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 # If applicable, add the following below this CDDL HEADER, with the
17 # fields enclosed by brackets "[]" replaced with your own identifying
18 # information: Portions Copyright [yyyy] [name of copyright owner]
19 #
20 # CDDL HEADER END
21 #
22 #
23 # Copyright 2005 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #

27 include ../Makefile.com
28 include ../../Makefile.lib.64

30 install: all $(ROOTLIBS64) $(ROOTLINKS64)
31 #endif /* ! codereview */
```

new/usr/src/lib/libzfs_core/common/libzfs_core.c

1

```
*****
13111 Thu Jun 28 15:57:12 2012
new/usr/src/lib/libzfs_core/common/libzfs_core.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 */
25
26 /*
27 * LibZFS_Core (lzc) is intended to replace most functionality in libzfs.
28 * It has the following characteristics:
29 *
30 * - Thread Safe. libzfs_core is accessible concurrently from multiple
31 * threads. This is accomplished primarily by avoiding global data
32 * (e.g. caching). Since it's thread-safe, there is no reason for a
33 * process to have multiple libzfs "instances". Therefore, we store
34 * our few pieces of data (e.g. the file descriptor) in global
35 * variables. The fd is reference-counted so that the libzfs_core
36 * library can be "initialized" multiple times (e.g. by different
37 * consumers within the same process).
38 *
39 * - Committed Interface. The libzfs_core interface will be committed,
40 * therefore consumers can compile against it and be confident that
41 * their code will continue to work on future releases of this code.
42 * Currently, the interface is Evolving (not Committed), but we intend
43 * to commit to it once it is more complete and we determine that it
44 * meets the needs of all consumers.
45 *
46 * - Programatic Error Handling. libzfs_core communicates errors with
47 * defined error numbers, and doesn't print anything to stdout/stderr.
48 *
49 * - Thin Layer. libzfs_core is a thin layer, marshaling arguments
50 * to/from the kernel ioctls. There is generally a 1:1 correspondence
51 * between libzfs_core functions and ioctls to /dev/zfs.
52 *
53 * - Clear Atomicity. Because libzfs_core functions are generally 1:1
54 * with kernel ioctls, and kernel ioctls are general atomic, each
```

new/usr/src/lib/libzfs_core/common/libzfs_core.c

2

```
55 * libzfs_core function is atomic. For example, creating multiple
56 * snapshots with a single call to lzc_snapshot() is atomic -- it
57 * can't fail with only some of the requested snapshots created, even
58 * in the event of power loss or system crash.
59 *
60 * - Continued libzfs Support. Some higher-level operations (e.g.
61 * support for "zfs send -R") are too complicated to fit the scope of
62 * libzfs_core. This functionality will continue to live in libzfs.
63 * Where appropriate, libzfs will use the underlying atomic operations
64 * of libzfs_core. For example, libzfs may implement "zfs send -R |
65 * zfs receive" by using individual "send one snapshot", rename,
66 * destroy, and "receive one snapshot" operations in libzfs_core.
67 * /sbin/zfs and /sbin/zpool will link with both libzfs and
68 * libzfs_core. Other consumers should aim to use only libzfs_core,
69 * since that will be the supported, stable interface going forwards.
70 */
71
72 #include <libzfs_core.h>
73 #include <ctype.h>
74 #include <unistd.h>
75 #include <stdlib.h>
76 #include <string.h>
77 #include <errno.h>
78 #include <fcntl.h>
79 #include <pthread.h>
80 #include <sys/nvpair.h>
81 #include <sys/param.h>
82 #include <sys/types.h>
83 #include <sys/stat.h>
84 #include <sys/zfs_ioctl.h>
85
86 static int g_fd;
87 static pthread_mutex_t g_lock = PTHREAD_MUTEX_INITIALIZER;
88 static int g_refcount;
89
90 int
91 libzfs_core_init(void)
92 {
93     (void) pthread_mutex_lock(&g_lock);
94     if (g_refcount == 0) {
95         g_fd = open("/dev/zfs", O_RDWR);
96         if (g_fd < 0) {
97             (void) pthread_mutex_unlock(&g_lock);
98             return (errno);
99         }
100     }
101     g_refcount++;
102     (void) pthread_mutex_unlock(&g_lock);
103     return (0);
104 }
105
106 void
107 libzfs_core_fini(void)
108 {
109     (void) pthread_mutex_lock(&g_lock);
110     ASSERT3S(g_refcount, >, 0);
111     g_refcount--;
112     if (g_refcount == 0)
113         (void) close(g_fd);
114     (void) pthread_mutex_unlock(&g_lock);
115 }
116
117 static int
118 lzc_ioctl(zfs_ioc_t ioc, const char *name,
119          nvlist_t *source, nvlist_t **resultp)
120 {
```

```

121     zfs_cmd_t zc = { 0 };
122     int error = 0;
123     char *packed;
124     size_t size;

126     ASSERT3S(g_refcount, >, 0);

128     (void) strncpy(zc.zc_name, name, sizeof (zc.zc_name));

130     packed = fnvlist_pack(source, &size);
131     zc.zc_nvlist_src = (uint64_t)(uintptr_t)packed;
132     zc.zc_nvlist_src_size = size;

134     if (resultp != NULL) {
135         zc.zc_nvlist_dst_size = MAX(size * 2, 128 * 1024);
136         zc.zc_nvlist_dst = (uint64_t)(uintptr_t)
137             malloc(zc.zc_nvlist_dst_size);
138         if (zc.zc_nvlist_dst == NULL) {
139             error = ENOMEM;
140             goto out;
141         }
142     }

144     while (ioctl(g_fd, ioc, &zc) != 0) {
145         if (errno == ENOMEM && resultp != NULL) {
146             free((void *) (uintptr_t)zc.zc_nvlist_dst);
147             zc.zc_nvlist_dst_size *= 2;
148             zc.zc_nvlist_dst = (uint64_t)(uintptr_t)
149                 malloc(zc.zc_nvlist_dst_size);
150             if (zc.zc_nvlist_dst == NULL) {
151                 error = ENOMEM;
152                 goto out;
153             }
154         } else {
155             error = errno;
156             break;
157         }
158     }
159     if (zc.zc_nvlist_dst_filled) {
160         *resultp = fnvlist_unpack((void *) (uintptr_t)zc.zc_nvlist_dst,
161             zc.zc_nvlist_dst_size);
162     } else if (resultp != NULL) {
163         *resultp = NULL;
164     }

166 out:
167     fnvlist_pack_free(packed, size);
168     free((void *) (uintptr_t)zc.zc_nvlist_dst);
169     return (error);
170 }

172 int
173 lzc_create(const char *fsname, dmu_objset_type_t type, nvlist_t *props)
174 {
175     int error;
176     nvlist_t *args = fnvlist_alloc();
177     fnvlist_add_int32(args, "type", type);
178     if (props != NULL)
179         fnvlist_add_nvlist(args, "props", props);
180     error = lzc_ioctl(ZFS_IOC_CREATE, fsname, args, NULL);
181     nvlist_free(args);
182     return (error);
183 }

185 int
186 lzc_clone(const char *fsname, const char *origin,

```

```

187     nvlist_t *props)
188 {
189     int error;
190     nvlist_t *args = fnvlist_alloc();
191     fnvlist_add_string(args, "origin", origin);
192     if (props != NULL)
193         fnvlist_add_nvlist(args, "props", props);
194     error = lzc_ioctl(ZFS_IOC_CLONE, fsname, args, NULL);
195     nvlist_free(args);
196     return (error);
197 }

199 /*
200  * Creates snapshots.
201  *
202  * The keys in the snaps nvlist are the snapshots to be created.
203  * They must all be in the same pool.
204  *
205  * The props nvlist is properties to set. Currently only user properties
206  * are supported. { user:prop_name -> string value }
207  *
208  * The returned results nvlist will have an entry for each snapshot that failed.
209  * The value will be the (int32) error code.
210  *
211  * The return value will be 0 if all snapshots were created, otherwise it will
212  * be the errno of a (undetermined) snapshot that failed.
213  */
214 int
215 lzc_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t **errlist)
216 {
217     nvpair_t *elem;
218     nvlist_t *args;
219     int error;
220     char pool[MAXNAMELEN];

222     *errlist = NULL;

224     /* determine the pool name */
225     elem = nvlist_next_nvpair(snaps, NULL);
226     if (elem == NULL)
227         return (0);
228     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
229     pool[strcspn(pool, "@*")] = '\0';

231     args = fnvlist_alloc();
232     fnvlist_add_nvlist(args, "snaps", snaps);
233     if (props != NULL)
234         fnvlist_add_nvlist(args, "props", props);

236     error = lzc_ioctl(ZFS_IOC_SNAPSHOT, pool, args, errlist);
237     nvlist_free(args);

239     return (error);
240 }

242 /*
243  * Destroys snapshots.
244  *
245  * The keys in the snaps nvlist are the snapshots to be destroyed.
246  * They must all be in the same pool.
247  *
248  * Snapshots that do not exist will be silently ignored.
249  *
250  * If 'defer' is not set, and a snapshot has user holds or clones, the
251  * destroy operation will fail and none of the snapshots will be
252  * destroyed.

```

```

253 *
254 * If 'defer' is set, and a snapshot has user holds or clones, it will be
255 * marked for deferred destruction, and will be destroyed when the last hold
256 * or clone is removed/destroyed.
257 *
258 * The return value will be 0 if all snapshots were destroyed (or marked for
259 * later destruction if 'defer' is set) or didn't exist to begin with.
260 *
261 * Otherwise the return value will be the errno of a (undetermined) snapshot
262 * that failed, no snapshots will be destroyed, and the errlist will have an
263 * entry for each snapshot that failed. The value in the errlist will be
264 * the (int32) error code.
265 */
266 int
267 lzc_destroy_snaps(nvlist_t *snaps, boolean_t defer, nvlist_t **errlist)
268 {
269     nvpair_t *elem;
270     nvlist_t *args;
271     int error;
272     char pool[MAXNAMELEN];

274     /* determine the pool name */
275     elem = nvlist_next_nvpair(snaps, NULL);
276     if (elem == NULL)
277         return (0);
278     (void) strlcpy(pool, nvpair_name(elem), sizeof (pool));
279     pool[strcspn(pool, "/"@)] = '\0';

281     args = fnvlist_alloc();
282     fnvlist_add_nvlist(args, "snaps", snaps);
283     if (defer)
284         fnvlist_add_boolean(args, "defer");

286     error = lzc_ioctl(ZFS_IOC_DESTROY_SNAPS, pool, args, errlist);
287     nvlist_free(args);

289     return (error);

291 }

293 int
294 lzc_snaprange_space(const char *firstsnap, const char *lastsnap,
295                    uint64_t *usedp)
296 {
297     nvlist_t *args;
298     nvlist_t *result;
299     int err;
300     char fs[MAXNAMELEN];
301     char *atp;

303     /* determine the fs name */
304     (void) strlcpy(fs, firstsnap, sizeof (fs));
305     atp = strchr(fs, '@');
306     if (atp == NULL)
307         return (EINVAL);
308     *atp = '\0';

310     args = fnvlist_alloc();
311     fnvlist_add_string(args, "firstsnap", firstsnap);

313     err = lzc_ioctl(ZFS_IOC_SPACE_SNAPS, lastsnap, args, &result);
314     nvlist_free(args);
315     if (err == 0)
316         *usedp = fnvlist_lookup_uint64(result, "used");
317     fnvlist_free(result);

```

```

319         return (err);
320     }

322 boolean_t
323 lzc_exists(const char *dataset)
324 {
325     /*
326      * The objset_stats ioctl is still legacy, so we need to construct our
327      * own zfs_cmd_t rather than using zfsc_ioctl().
328      */
329     zfs_cmd_t zc = { 0 };

331     (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
332     return (ioctl(g_fd, ZFS_IOC_OBJSET_STATS, &zc) == 0);
333 }

335 /*
336 * If fromsnap is NULL, a full (non-incremental) stream will be sent.
337 */
338 int
339 lzc_send(const char *snapname, const char *fromsnap, int fd)
340 {
341     nvlist_t *args;
342     int err;

344     args = fnvlist_alloc();
345     fnvlist_add_int32(args, "fd", fd);
346     if (fromsnap != NULL)
347         fnvlist_add_string(args, "fromsnap", fromsnap);
348     err = lzc_ioctl(ZFS_IOC_SEND_NEW, snapname, args, NULL);
349     nvlist_free(args);
350     return (err);
351 }

353 /*
354 * If fromsnap is NULL, a full (non-incremental) stream will be estimated.
355 */
356 int
357 lzc_send_space(const char *snapname, const char *fromsnap, uint64_t *spacep)
358 {
359     nvlist_t *args;
360     nvlist_t *result;
361     int err;

363     args = fnvlist_alloc();
364     if (fromsnap != NULL)
365         fnvlist_add_string(args, "fromsnap", fromsnap);
366     err = lzc_ioctl(ZFS_IOC_SEND_SPACE, snapname, args, &result);
367     nvlist_free(args);
368     if (err == 0)
369         *spacep = fnvlist_lookup_uint64(result, "space");
370     nvlist_free(result);
371     return (err);
372 }

374 static int
375 recv_read(int fd, void *buf, int ilen)
376 {
377     char *cp = buf;
378     int rv;
379     int len = ilen;

381     do {
382         rv = read(fd, cp, len);
383         cp += rv;
384         len -= rv;

```

```

385     } while (rv > 0);
387     if (rv < 0 || len != 0)
388         return (EIO);
390     return (0);
391 }

393 /*
394  * The simplest receive case: receive from the specified fd, creating the
395  * specified snapshot. Apply the specified properties a "received" properties
396  * (which can be overridden by locally-set properties). If the stream is a
397  * clone, its origin snapshot must be specified by 'origin'. The 'force'
398  * flag will cause the target filesystem to be rolled back or destroyed if
399  * necessary to receive.
400  *
401  * Return 0 on success or an errno on failure.
402  *
403  * Note: this interface does not work on dedup'd streams
404  * (those with DMU_BACKUP_FEATURE_DEDUP).
405  */
406 int
407 lzc_receive(const char *snapname, nvlist_t *props, const char *origin,
408             boolean_t force, int fd)
409 {
410     /*
411      * The receive ioctl is still legacy, so we need to construct our own
412      * zfs_cmd_t rather than using zfs_ioctl().
413      */
414     zfs_cmd_t zc = { 0 };
415     char *atp;
416     char *packed = NULL;
417     size_t size;
418     dmuf_replay_record_t drr;
419     int error;

421     ASSERT3S(g_refcount, >, 0);

423     /* zc_name is name of containing filesystem */
424     (void) strncpy(zc.zc_name, snapname, sizeof (zc.zc_name));
425     atp = strchr(zc.zc_name, '@');
426     if (atp == NULL)
427         return (EINVAL);
428     *atp = '\0';

430     /* if the fs does not exist, try its parent. */
431     if (!lzc_exists(zc.zc_name)) {
432         char *slashp = strrchr(zc.zc_name, '/');
433         if (slashp == NULL)
434             return (ENOENT);
435         *slashp = '\0';
437     }

439     /* zc_value is full name of the snapshot to create */
440     (void) strncpy(zc.zc_value, snapname, sizeof (zc.zc_value));

442     if (props != NULL) {
443         /* zc_nvlist_src is props to set */
444         packed = fnvlist_pack(props, &size);
445         zc.zc_nvlist_src = (uint64_t)(uintptr_t)packed;
446         zc.zc_nvlist_src_size = size;
447     }

449     /* zc_string is name of clone origin (if DRR_FLAG_CLONE) */
450     if (origin != NULL)

```

```

451         (void) strncpy(zc.zc_string, origin, sizeof (zc.zc_string));

453     /* zc_begin_record is non-byteswapped BEGIN record */
454     error = recv_read(fd, &drr, sizeof (drr));
455     if (error != 0)
456         goto out;
457     zc.zc_begin_record = drr.drr_u.drr_begin;

459     /* zc_cookie is fd to read from */
460     zc.zc_cookie = fd;

462     /* zc_guid is force flag */
463     zc.zc_guid = force;

465     /* zc_cleanup_fd is unused */
466     zc.zc_cleanup_fd = -1;

468     error = ioctl(g_fd, ZFS_IOC_RECV, &zc);
469     if (error != 0)
470         error = errno;

472 out:
473     if (packed != NULL)
474         fnvlist_pack_free(packed, size);
475     free((void*)(uintptr_t)zc.zc_nvlist_dst);
476     return (error);
477 }
478 #endif /* ! codereview */

```

new/usr/src/lib/libzfs_core/common/libzfs_core.h

1

1874 Thu Jun 28 15:57:12 2012

new/usr/src/lib/libzfs_core/common/libzfs_core.h

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2012 by Delphix. All rights reserved.
24 */
25
26 #ifndef _LIBZFS_CORE_H
27 #define _LIBZFS_CORE_H
28
29 #include <libnvpair.h>
30 #include <sys/param.h>
31 #include <sys/types.h>
32 #include <sys/fs/zfs.h>
33
34 #ifdef __cplusplus
35 extern "C" {
36 #endif
37
38 int libzfs_core_init(void);
39 void libzfs_core_fini(void);
40
41 int lzc_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t **errlist);
42 int lzc_create(const char *fsname, dmu_objset_type_t type, nvlist_t *props);
43 int lzc_clone(const char *fsname, const char *origin, nvlist_t *props);
44 int lzc_destroy_snaps(nvlist_t *snaps, boolean_t defer, nvlist_t **errlist);
45
46 int lzc_snaprange_space(const char *firstsnap, const char *lastsnap,
47     uint64_t *usedp);
48
49 int lzc_send(const char *snapname, const char *fromsnap, int fd);
50 int lzc_receive(const char *snapname, nvlist_t *props, const char *origin,
51     boolean_t force, int fd);
52 int lzc_send_space(const char *snapname, const char *fromsnap,
53     uint64_t *result);
```

new/usr/src/lib/libzfs_core/common/libzfs_core.h

2

```
55 boolean_t lzc_exists(const char *dataset);
```

```
58 #ifdef __cplusplus
```

```
59 }
```

```
60 #endif
```

```
62 #endif /* _LIBZFS_CORE_H */
```

```
63 #endif /* !codereview */
```

new/usr/src/lib/libzfs_core/common/l1ib-lzfs_core

1

939 Thu Jun 28 15:57:13 2012

new/usr/src/lib/libzfs_core/common/l1ib-lzfs_core

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2012 by Delphix. All rights reserved.
23 */

25 /*LINTLIBRARY*/
26 /*PROTOLIB1*/

28 #include <libzfs_core.h>
29 #endif /* ! codereview */
```

```

*****
1537 Thu Jun 28 15:57:13 2012
new/usr/src/lib/libzfs_core/common/mapfile-vers
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 # Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
22 # Copyright (c) 2012 by Delphix. All rights reserved.
23 #
24 # MAPFILE HEADER START
25 #
26 # WARNING: STOP NOW. DO NOT MODIFY THIS FILE.
27 # Object versioning must comply with the rules detailed in
28 #
29 #     usr/src/lib/README.mapfiles
30 #
31 # You should not be making modifications here until you've read the most current
32 # copy of that file. If you need help, contact a gatekeeper for guidance.
33 #
34 # MAPFILE HEADER END
35 #

37 $mapfile_version 2

39 SYMBOL_VERSION ILLUMOS_0.1 {
40     global:
41
42         libzfs_core_fini;
43         libzfs_core_init;
44         lzc_clone;
45         lzc_create;
46         lzc_destroy_snaps;
47         lzc_exists;
48         lzc_receive;
49         lzc_send;
50         lzc_send_space;
51         lzc_snaprange_space;
52         lzc_snapshot;

54     local:

```

```

55         *;
56     };
57 #endif /* ! codereview */

```


new/usr/src/lib/libzfs_core/i386/Makefile

1

997 Thu Jun 28 15:57:13 2012

new/usr/src/lib/libzfs_core/i386/Makefile

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License, Version 1.0 only
6 # (the "License"). You may not use this file except in compliance
7 # with the License.
8 #
9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 # or http://www.opensolaris.org/os/licensing.
11 # See the License for the specific language governing permissions
12 # and limitations under the License.
13 #
14 # When distributing Covered Code, include this CDDL HEADER in each
15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 # If applicable, add the following below this CDDL HEADER, with the
17 # fields enclosed by brackets "[]" replaced with your own identifying
18 # information: Portions Copyright [yyyy] [name of copyright owner]
19 #
20 # CDDL HEADER END
21 #
22 #
23 # Copyright 2005 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #

27 include ../Makefile.com

29 install: all $(ROOTLIBS) $(ROOTLINKS) $(ROOTLINT)
30 #endif /* ! codereview */
```

new/usr/src/lib/libzfs_core/sparc/Makefile

1

997 Thu Jun 28 15:57:13 2012

new/usr/src/lib/libzfs_core/sparc/Makefile

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License, Version 1.0 only
6 # (the "License"). You may not use this file except in compliance
7 # with the License.
8 #
9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 # or http://www.opensolaris.org/os/licensing.
11 # See the License for the specific language governing permissions
12 # and limitations under the License.
13 #
14 # When distributing Covered Code, include this CDDL HEADER in each
15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 # If applicable, add the following below this CDDL HEADER, with the
17 # fields enclosed by brackets "[]" replaced with your own identifying
18 # information: Portions Copyright [yyyy] [name of copyright owner]
19 #
20 # CDDL HEADER END
21 #
22 #
23 # Copyright 2005 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #

27 include ../Makefile.com

29 install: all $(ROOTLIBS) $(ROOTLINKS) $(ROOTLINT)
30 #endif /* ! codereview */
```

new/usr/src/lib/libzfs_core/sparcv9/Makefile

1

1047 Thu Jun 28 15:57:13 2012

new/usr/src/lib/libzfs_core/sparcv9/Makefile

2882 implement libzfs_core

2883 changing "canmount" property to "on" should not always remount dataset

2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on

Reviewed by: George Wilson <george.wilson@delphix.com>

Reviewed by: Chris Siden <christopher.siden@delphix.com>

Reviewed by: Garrett D'Amore <garrett@damore.org>

Reviewed by: Bill Pijewski <wdp@joyent.com>

Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License, Version 1.0 only
6 # (the "License"). You may not use this file except in compliance
7 # with the License.
8 #
9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 # or http://www.opensolaris.org/os/licensing.
11 # See the License for the specific language governing permissions
12 # and limitations under the License.
13 #
14 # When distributing Covered Code, include this CDDL HEADER in each
15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 # If applicable, add the following below this CDDL HEADER, with the
17 # fields enclosed by brackets "[]" replaced with your own identifying
18 # information: Portions Copyright [yyyy] [name of copyright owner]
19 #
20 # CDDL HEADER END
21 #
22 #
23 # Copyright 2006 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #

27 include ../Makefile.com
28 include ../../Makefile.lib.64

30 sparcv9_C_PICFLAGS= -K PIC

32 install: all $(ROOTLIBS64) $(ROOTLINKS64)
33 #endif /* ! codereview */
```

```

*****
20010 Thu Jun 28 15:57:13 2012
new/usr/src/lib/libzpool/common/kernel.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

```

```

874 uid_t
875 crgetruid(cred_t *cr)
876 {
877     return (0);
878 }

880 #endif /* ! codereview */
881 gid_t
882 crgetgid(cred_t *cr)
883 {
884     return (0);
885 }

887 int
888 crgetngroups(cred_t *cr)
889 {
890     return (0);
891 }

893 gid_t *
894 crgetgroups(cred_t *cr)
895 {
896     return (NULL);
897 }

899 int
900 zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr)
901 {
902     return (0);
903 }

905 int
906 zfs_secpolicy_rename_perms(const char *from, const char *to, cred_t *cr)
907 {
908     return (0);
909 }

911 int
912 zfs_secpolicy_destroy_perms(const char *name, cred_t *cr)
913 {
914     return (0);
915 }

917 ksiddomain_t *
918 ksiddomain_lookup(const char *dom)
919 {
920     ksiddomain_t *kd;

922     kd = umem_zalloc(sizeof (ksiddomain_t), UMEM_NOFAIL);
923     kd->kd_name = spa_strdup(dom);
924     return (kd);
925 }

```

```

927 void
928 ksiddomain_rele(ksiddomain_t *ksid)
929 {
930     spa_strfree(ksid->kd_name);
931     umem_free(ksid, sizeof (ksiddomain_t));
932 }

934 /*
935  * Do not change the length of the returned string; it must be freed
936  * with strfree().
937  */
938 char *
939 kmem_asprintf(const char *fmt, ...)
940 {
941     int size;
942     va_list adx;
943     char *buf;

945     va_start(adx, fmt);
946     size = vsnprintf(NULL, 0, fmt, adx) + 1;
947     va_end(adx);

949     buf = kmem_alloc(size, KM_SLEEP);

951     va_start(adx, fmt);
952     size = vsnprintf(buf, size, fmt, adx);
953     va_end(adx);

955     return (buf);
956 }

958 /* ARGSUSED */
959 int
960 zfs_onexit_fd_hold(int fd, minor_t *minorp)
961 {
962     *minorp = 0;
963     return (0);
964 }

966 /* ARGSUSED */
967 void
968 zfs_onexit_fd_rele(int fd)
969 {
970 }

972 /* ARGSUSED */
973 int
974 zfs_onexit_add_cb(minor_t minor, void (*func)(void *), void *data,
975                  uint64_t *action_handle)
976 {
977     return (0);
978 }

980 /* ARGSUSED */
981 int
982 zfs_onexit_del_cb(minor_t minor, uint64_t action_handle, boolean_t fire)
983 {
984     return (0);
985 }

987 /* ARGSUSED */
988 int
989 zfs_onexit_cb_data(minor_t minor, uint64_t action_handle, void **data)
990 {
991     return (0);

```

992 }

new/usr/src/lib/libzpool/common/sys/zfs_context.h

1

```
*****
16923 Thu Jun 28 15:57:13 2012
new/usr/src/lib/libzpool/common/sys/zfs_context.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

268 typedef int krw_t;

270 #define RW_READER      0
271 #define RW_WRITER      1
272 #define RW_DEFAULT     USYNC_THREAD

274 #undef RW_READ_HELD
275 #define RW_READ_HELD(x)      _rw_read_held(&(x)->rw_lock)

277 #undef RW_WRITE_HELD
278 #define RW_WRITE_HELD(x)     _rw_write_held(&(x)->rw_lock)

280 extern void rw_init(krwlock_t *rwlp, char *name, int type, void *arg);
281 extern void rw_destroy(krwlock_t *rwlp);
282 extern void rw_enter(krwlock_t *rwlp, krw_t rw);
283 extern int rw_tryenter(krwlock_t *rwlp, krw_t rw);
284 extern int rw_tryupgrade(krwlock_t *rwlp);
285 extern void rw_exit(krwlock_t *rwlp);
286 #define rw_downgrade(rwlp) do { } while (0)

288 extern uid_t crgetuid(cred_t *cr);
289 extern uid_t crgetruid(cred_t *cr);
290 #endif /* ! codereview */
291 extern gid_t crgetgid(cred_t *cr);
292 extern int crgetngroups(cred_t *cr);
293 extern gid_t *crgetgroups(cred_t *cr);

295 /*
296  * Condition variables
297  */
298 typedef cond_t kcondvar_t;

300 #define CV_DEFAULT      USYNC_THREAD

302 extern void cv_init(kcondvar_t *cv, char *name, int type, void *arg);
303 extern void cv_destroy(kcondvar_t *cv);
304 extern void cv_wait(kcondvar_t *cv, kmutex_t *mp);
305 extern clock_t cv_timedwait(kcondvar_t *cv, kmutex_t *mp, clock_t abstime);
306 extern void cv_signal(kcondvar_t *cv);
307 extern void cv_broadcast(kcondvar_t *cv);

309 /*
310  * kstat creation, installation and deletion
311  */
312 extern kstat_t *kstat_create(char *, int,
313     char *, char *, uchar_t, ulong_t, uchar_t);
314 extern void kstat_install(kstat_t *);
315 extern void kstat_delete(kstat_t *);

317 /*
318  * Kernel memory
319  */
```

new/usr/src/lib/libzpool/common/sys/zfs_context.h

2

```
320 #define KM_SLEEP      UMEM_NOFAIL
321 #define KM_PUSHPAGE   KM_SLEEP
322 #define KM_NOSLEEP    UMEM_DEFAULT
323 #define KMC_NODEBUG   UMC_NODEBUG
324 #define KMC_NOTOUCH   0 /* not needed for userland caches */
325 #define kmem_alloc(_s, _f)      umem_alloc(_s, _f)
326 #define kmem_zalloc(_s, _f)     umem_zalloc(_s, _f)
327 #define kmem_free(_b, _s)       umem_free(_b, _s)
328 #define kmem_cache_create(_a, _b, _c, _d, _e, _f, _g, _h, _i) \
329     umem_cache_create(_a, _b, _c, _d, _e, _f, _g, _h, _i)
330 #define kmem_cache_destroy(_c)  umem_cache_destroy(_c)
331 #define kmem_cache_alloc(_c, _f) umem_cache_alloc(_c, _f)
332 #define kmem_cache_free(_c, _b) umem_cache_free(_c, _b)
333 #define kmem_debugging()        0
334 #define kmem_cache_reap_now(_c) /* nothing */
335 #define kmem_cache_set_move(_c, _cb) /* nothing */
336 #define vmem_qcache_reap(_v) /* nothing */
337 #define POINTER_INVALIDATE(_pp) /* nothing */
338 #define POINTER_IS_VALID(_p)    0

340 extern vmem_t *zio_arena;

342 typedef umem_cache_t kmem_cache_t;

344 typedef enum kmem_cbrc {
345     KMEM_CBRC_YES,
346     KMEM_CBRC_NO,
347     KMEM_CBRC_LATER,
348     KMEM_CBRC_DONT_NEED,
349     KMEM_CBRC_DONT_KNOW
350 } kmem_cbrc_t;

352 /*
353  * Task queues
354  */
355 typedef struct taskq taskq_t;
356 typedef uintptr_t taskqid_t;
357 typedef void (task_func_t)(void *);

359 typedef struct taskq_ent {
360     struct taskq_ent *tqent_next;
361     struct taskq_ent *tqent_prev;
362     task_func_t *tqent_func;
363     void *tqent_arg;
364     uintptr_t tqent_flags;
365 } taskq_ent_t;

367 #define TQENT_FLAG_PREALLOC 0x1 /* taskq_dispatch_ent used */

369 #define TASKQ_PREPOPULATE 0x0001
370 #define TASKQ_CPR_SAFE 0x0002 /* Use CPR safe protocol */
371 #define TASKQ_DYNAMIC 0x0004 /* Use dynamic thread scheduling */
372 #define TASKQ_THREADS_CPU_PCT 0x0008 /* Scale # threads by # cpus */
373 #define TASKQ_DC_BATCH 0x0010 /* Mark threads as batch */

375 #define TQ_SLEEP      KM_SLEEP /* Can block for memory */
376 #define TQ_NOSLEEP    KM_NOSLEEP /* cannot block for memory; may fail */
377 #define TQ_NOQUEUE    0x02 /* Do not enqueue if can't dispatch */
378 #define TQ_FRONT      0x08 /* Queue in front */

381 extern taskq_t *system_taskq;

383 extern taskq_t *taskq_create(const char *, int, pri_t, int, int, uint_t);
384 #define taskq_create_proc(a, b, c, d, e, p, f) \
385     (taskq_create(a, b, c, d, e, f))
```

```

386 #define taskq_create_sysdc(a, b, d, e, p, dc, f) \
387     (taskq_create(a, b, maxclsypri, d, e, f))
388 extern taskqid_t taskq_dispatch(taskq_t *, task_func_t, void *, uint_t);
389 extern void taskq_dispatch_ent(taskq_t *, task_func_t, void *, uint_t,
390     taskq_ent_t *);
391 extern void taskq_destroy(taskq_t *);
392 extern void taskq_wait(taskq_t *);
393 extern int taskq_member(taskq_t *, void *);
394 extern void system_taskq_init(void);
395 extern void system_taskq_fini(void);

397 #define XVA_MAPSIZE      3
398 #define XVA_MAGIC        0x78766174

400 /*
401  * vnodes
402  */
403 typedef struct vnode {
404     uint64_t      v_size;
405     int           v_fd;
406     char          *v_path;
407 } vnode_t;

409 #define AV_SCANSTAMP_SZ 32          /* length of anti-virus scanstamp */

411 typedef struct xoptattr {
412     timestruc_t   xoa_createtime; /* Create time of file */
413     uint8_t       xoa_archive;
414     uint8_t       xoa_system;
415     uint8_t       xoa_readonly;
416     uint8_t       xoa_hidden;
417     uint8_t       xoa_nounlink;
418     uint8_t       xoa_immutable;
419     uint8_t       xoa_appendonly;
420     uint8_t       xoa_nodump;
421     uint8_t       xoa_settable;
422     uint8_t       xoa_opaque;
423     uint8_t       xoa_av_quarantined;
424     uint8_t       xoa_av_modified;
425     uint8_t       xoa_av_scanstamp[AV_SCANSTAMP_SZ];
426     uint8_t       xoa_reparse;
427     uint8_t       xoa_offline;
428     uint8_t       xoa_sparse;
429 } xoptattr_t;

431 typedef struct vattr {
432     uint_t         va_mask;        /* bit-mask of attributes */
433     u_offset_t     va_size;        /* file size in bytes */
434 } vattr_t;

437 typedef struct xvattr {
438     vattr_t        xva_vattr;      /* Embedded vattr structure */
439     uint32_t       xva_magic;      /* Magic Number */
440     uint32_t       xva_mapsize;    /* Size of attr bitmap (32-bit words) */
441     uint32_t       *xva_rtnattrmap; /* Ptr to xva_rtnattrmap[] */
442     uint32_t       xva_reqattrmap[XVA_MAPSIZE]; /* Requested attrs */
443     uint32_t       xva_rtnattrmap[XVA_MAPSIZE]; /* Returned attrs */
444     xoptattr_t     xva_xoptattrs; /* Optional attributes */
445 } xvattr_t;

447 typedef struct vsecattr {
448     uint_t         vsa_mask;        /* See below */
449     int            vsa_aclcnt;      /* ACL entry count */
450     void           *vsa_aclentp;   /* pointer to ACL entries */
451     int            vsa_dfaclcnt;    /* default ACL entry count */

```

```

452     void           *vsa_dfaclentp; /* pointer to default ACL entries */
453     size_t         vsa_aclentsz;   /* ACE size in bytes of vsa_aclentp */
454 } vsecattr_t;

456 #define AT_TYPE          0x00001
457 #define AT_MODE          0x00002
458 #define AT_UID           0x00004
459 #define AT_GID           0x00008
460 #define AT_FSID         0x00010
461 #define AT_NODEID       0x00020
462 #define AT_NLINK        0x00040
463 #define AT_SIZE          0x00080
464 #define AT_ATIME         0x00100
465 #define AT_MTIME         0x00200
466 #define AT_CTIME         0x00400
467 #define AT_RDEV          0x00800
468 #define AT_BLKSIZE       0x01000
469 #define AT_NBLOCKS       0x02000
470 #define AT_SEQ           0x08000
471 #define AT_XVATTR        0x10000

473 #define CRCREAT          0

475 extern int fop_getattr(vnode_t *vp, vattr_t *vap);

477 #define VOP_CLOSE(vp, f, c, o, cr, ct) 0
478 #define VOP_PUTPAGE(vp, of, sz, fl, cr, ct) 0
479 #define VOP_GETATTR(vp, vap, fl, cr, ct) fop_getattr((vp), (vap));

481 #define VOP_FSYNC(vp, f, cr, ct)       fsync((vp)->v_fd)

483 #define VN_RELE(vp)                   vn_close(vp)

485 extern int vn_open(char *path, int x1, int oflags, int mode, vnode_t **vpp,
486     int x2, int x3);
487 extern int vn_openat(char *path, int x1, int oflags, int mode, vnode_t **vpp,
488     int x2, int x3, vnode_t *vp, int fd);
489 extern int vn_rdwv(int uio, vnode_t *vp, void *addr, ssize_t len,
490     offset_t offset, int x1, int x2, rlim64_t x3, void *x4, ssize_t *residp);
491 extern void vn_close(vnode_t *vp);

493 #define vn_remove(path, x1, x2)         remove(path)
494 #define vn_rename(from, to, seg)       rename((from), (to))
495 #define vn_is_readonly(vp)             B_FALSE

497 extern vnode_t *rootdir;

499 #include <sys/file.h>          /* for FREAD, FWRITE, etc */

501 /*
502  * Random stuff
503  */
504 #define ddi_get_lbolt()                (gethrtime() >> 23)
505 #define ddi_get_lbolt64()              (gethrtime() >> 23)
506 #define hz                             119 /* frequency when using gethrtime() >> 23 for lbolt */

508 extern void delay(clock_t ticks);

510 #define gethrestime_sec() time(NULL)
511 #define gethrestime(t) \
512     do { \
513         (t)->tv_sec = gethrestime_sec(); \
514         (t)->tv_nsec = 0; \
515     } while (0);

517 #define max_ncpus                       64

```

```

519 #define minclsyspri      60
520 #define maxclsyspri      99

522 #define CPU_SEQID        (thr_self() & (max_ncpus - 1))

524 #define kcred            NULL
525 #define CRED()           NULL

527 #define ptob(x)          ((x) * PAGE_SIZE)

529 extern uint64_t physmem;

531 extern int highbit(ulong_t i);
532 extern int random_get_bytes(uint8_t *ptr, size_t len);
533 extern int random_get_pseudo_bytes(uint8_t *ptr, size_t len);

535 extern void kernel_init(int);
536 extern void kernel_fini(void);

538 struct spa;
539 extern void nicenum(uint64_t num, char *buf);
540 extern void show_pool_stats(struct spa *);

542 typedef struct callb_cpr {
543     kmutex_t      *cc_lockp;
544 } callb_cpr_t;

546 #define CALLB_CPR_INIT(cp, lockp, func, name) { \
547     (cp)->cc_lockp = lockp; \
548 }

550 #define CALLB_CPR_SAFE_BEGIN(cp) { \
551     ASSERT(MUTEX_HELD((cp)->cc_lockp)); \
552 }

554 #define CALLB_CPR_SAFE_END(cp, lockp) { \
555     ASSERT(MUTEX_HELD((cp)->cc_lockp)); \
556 }

558 #define CALLB_CPR_EXIT(cp) { \
559     ASSERT(MUTEX_HELD((cp)->cc_lockp)); \
560     mutex_exit((cp)->cc_lockp); \
561 }

563 #define zone_dataset_visible(x, y)      (1)
564 #define INGLOBALZONE(z)                (1)

566 extern char *kmem_asprintf(const char *fmt, ...);
567 #define strfree(str) kmem_free((str), strlen(str)+1)

569 /*
570  * Hostname information
571  */
572 extern char hw_serial[];          /* for userland-emulated hostid access */
573 extern int ddi_strtoul(const char *str, char **nptr, int base,
574     unsigned long *result);

576 extern int ddi_strtoull(const char *str, char **nptr, int base,
577     u_longlong_t *result);

579 /* ZFS Boot Related stuff. */

581 struct _buf {
582     intptr_t      _fd;
583 };

```

```

585 struct bootstat {
586     uint64_t st_size;
587 };

589 typedef struct ace_object {
590     uid_t      a_who;
591     uint32_t    a_access_mask;
592     uint16_t    a_flags;
593     uint16_t    a_type;
594     uint8_t     a_obj_type[16];
595     uint8_t     a_inherit_obj_type[16];
596 } ace_object_t;

599 #define ACE_ACCESS_ALLOWED_OBJECT_ACE_TYPE      0x05
600 #define ACE_ACCESS_DENIED_OBJECT_ACE_TYPE      0x06
601 #define ACE_SYSTEM_AUDIT_OBJECT_ACE_TYPE      0x07
602 #define ACE_SYSTEM_ALARM_OBJECT_ACE_TYPE      0x08

604 extern struct _buf *kobj_open_file(char *name);
605 extern int kobj_read_file(struct _buf *file, char *buf, unsigned size,
606     unsigned off);
607 extern void kobj_close_file(struct _buf *file);
608 extern int kobj_get_filesize(struct _buf *file, uint64_t *size);
609 extern int zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr);
610 extern int zfs_secpolicy_rename_perms(const char *from, const char *to,
611     cred_t *cr);
612 extern int zfs_secpolicy_destroy_perms(const char *name, cred_t *cr);
613 extern zoneid_t getzoneid(void);

615 /* SID stuff */
616 typedef struct ksiddomain {
617     uint_t kd_ref;
618     uint_t kd_len;
619     char *kd_name;
620 } ksiddomain_t;

622 ksiddomain_t *ksid_lookupdomain(const char *);
623 void ksiddomain_rele(ksiddomain_t *);

625 #define DDI_SLEEP      KM_SLEEP
626 #define ddi_log_sysevent(_a, _b, _c, _d, _e, _f, _g) \
627     sysevent_post_event(_c, _d, _b, "libzpool", _e, _f)

629 #ifdef __cplusplus
630 }
631 #endif

633 #endif /* _SYS_ZFS_CONTEXT_H */

```



```

*****
105813 Thu Jun 28 15:57:14 2012
new/usr/src/man/man1m/zfs.1m
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 \" te
2 .\" Copyright (c) 2009 Sun Microsystems, Inc. All Rights Reserved.
3 .\" Copyright (c) 2012 by Delphix. All rights reserved.
4 .\" Copyright (c) 2012 Nexenta Systems, Inc. All Rights Reserved.
5 .\" Copyright (c) 2012, Joyent, Inc. All rights reserved.
6 .\" The contents of this file are subject to the terms of the Common Development
7 .\" See the License for the specific language governing permissions and limitat
8 .\" the fields enclosed by brackets \"[]\" replaced with your own identifying info
9 .\" Copyright 2011 Joshua M. Clulow <josh@sysmgr.org>
10 .TH ZFS 1M \"28 Jul 2011\"
11 .SH NAME
12 zfs - configures ZFS file systems
13 .SH SYNOPSIS
14 .LP
15 .nf
16 \fBzfs\fR [\fB-?\fR]
17 .fi

19 .LP
20 .nf
21 \fBzfs\fR \fBcreate\fR [\fB-p\fR] [\fB-o\fR \fIproperty\fR=\fIvalue\fR] ... \fIfi
22 .fi

24 .LP
25 .nf
26 \fBzfs\fR \fBcreate\fR [\fB-ps\fR] [\fB-b\fR \fIblocksize\fR] [\fB-o\fR \fIprope
27 .fi

29 .LP
30 .nf
31 \fBzfs\fR \fBdestroy\fR [\fB-fnpRrv\fR] \fIfilesystem\fR|\fIvolume\fR
32 .fi

34 .LP
35 .nf
36 \fBzfs\fR \fBdestroy\fR [\fB-dnpRrv\fR] \fIfilesystem\fR|\fIvolume\fR@\fIsnap\fR
37 .fi

39 .LP
40 .nf
41 \fBzfs\fR \fBsnapshot\fR [\fB-r\fR] [\fB-o\fR \fIproperty\fR=\fIvalue\fR]...
42 \fIfilesystem@snapname\fR|\fIvolume@snapname\fR...
42 \fIfilesystem@snapname\fR|\fIvolume@snapname\fR
43 .fi

45 .LP
46 .nf
47 \fBzfs\fR \fBrollback\fR [\fB-rf\fR] \fIsnapshot\fR
48 .fi

50 .LP
51 .nf
52 \fBzfs\fR \fBclone\fR [\fB-p\fR] [\fB-o\fR \fIproperty\fR=\fIvalue\fR] ... \fIisn
53 .fi

```

```

55 .LP
56 .nf
57 \fBzfs\fR \fBpromote\fR \fIclone-filesystem\fR
58 .fi

60 .LP
61 .nf
62 \fBzfs\fR \fBrename\fR [\fB-f\fR] \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR
63 \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR
64 .fi

66 .LP
67 .nf
68 \fBzfs\fR \fBrename\fR [\fB-fp\fR] \fIfilesystem\fR|\fIvolume\fR \fIfilesystem\f
69 .fi

71 .LP
72 .nf
73 \fBzfs\fR \fBrename\fR \fB-r\fR \fIsnapshot\fR \fIsnapshot\fR
74 .fi

76 .LP
77 .nf
78 \fBzfs\fR \fBlist\fR [\fB-r\fR]|\fB-d\fR \fIdepth\fR][\fB-H\fR][\fB-o\fR \fIprope
79 [\fB-s\fR \fIproperty\fR] ... [\fB-S\fR \fIproperty\fR] ... \fIfilesystem\
80 .fi

82 .LP
83 .nf
84 \fBzfs\fR \fBset\fR \fIproperty\fR=\fIvalue\fR \fIfilesystem\fR|\fIvolume\fR|\fI
85 .fi

87 .LP
88 .nf
89 \fBzfs\fR \fBget\fR [\fB-r\fR]|\fB-d\fR \fIdepth\fR][\fB-Hp\fR][\fB-o\fR \fIfield
90 [\fB-s\fR \fIsource\fR[,...]] \"fIall\fR\" | \fIproperty\fR[,...]] \fIfilesyst
91 .fi

93 .LP
94 .nf
95 \fBzfs\fR \fBinherit\fR [\fB-r\fR] \fIproperty\fR \fIfilesystem\fR|\fIvolume|sna
96 .fi

98 .LP
99 .nf
100 \fBzfs\fR \fBupgrade\fR [\fB-v\fR]
101 .fi

103 .LP
104 .nf
105 \fBzfs\fR \fBupgrade\fR [\fB-r\fR] [\fB-V\fR \fIversion\fR] \fB-a\fR | \fIfilesy
106 .fi

108 .LP
109 .nf
110 \fBzfs\fR \fBuserspace\fR [\fB-niHp\fR] [\fB-o\fR \fIfield\fR[,...]] [\fB-sS\fR
111 [\fB-t\fR \fItype\fR[,...]] \fIfilesystem\fR|\fIsnapshot\fR
112 .fi

114 .LP
115 .nf
116 \fBzfs\fR \fBgroupspace\fR [\fB-niHp\fR] [\fB-o\fR \fIfield\fR[,...]] [\fB-sS\fR
117 [\fB-t\fR \fItype\fR[,...]] \fIfilesystem\fR|\fIsnapshot\fR
118 .fi

```

```

120 .LP
121 .nf
122 \fBzfs\fR \fBmount\fR
123 .fi

125 .LP
126 .nf
127 \fBzfs\fR \fBmount\fR [\fB-vO\fR] [\fB-o \fIoptions\fR\fR] \fB-a\fR | \fIfilesys
128 .fi

130 .LP
131 .nf
132 \fBzfs\fR \fBunmount\fR [\fB-f\fR] \fB-a\fR | \fIfilesystem\fR|\fImountpoint\fR
133 .fi

135 .LP
136 .nf
137 \fBzfs\fR \fBshare\fR \fB-a\fR | \fIfilesystem\fR
138 .fi

140 .LP
141 .nf
142 \fBzfs\fR \fBunshare\fR \fB-a\fR \fIfilesystem\fR|\fImountpoint\fR
143 .fi

145 .LP
146 .nf
147 \fBzfs\fR \fBsend\fR [\fB-DnppRrv\fR] [\fB-\fR[\fBiI\fR] \fIsnapshot\fR] \fIsnap
148 .fi

150 .LP
151 .nf
152 \fBzfs\fR \fBreceive\fR [\fB-vnFu\fR] \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\
153 .fi

155 .LP
156 .nf
157 \fBzfs\fR \fBreceive\fR [\fB-vnFu\fR] [\fB-d\fR|\fB-e\fR] \fIfilesystem\fR
158 .fi

160 .LP
161 .nf
162 \fBzfs\fR \fBallow\fR \fIfilesystem\fR|\fIvolume\fR
163 .fi

165 .LP
166 .nf
167 \fBzfs\fR \fBallow\fR [\fB-ldug\fR] "\fIeveryone\fR"|\fIuser\fR|\fIgroup\fR[,...
168 \fIfilesystem\fR|\fIvolume\fR
169 .fi

171 .LP
172 .nf
173 \fBzfs\fR \fBallow\fR [\fB-ld\fR] \fB-e\fR \fIperm\fR|\fIsetname\fR[,...] \fIfi
174 .fi

176 .LP
177 .nf
178 \fBzfs\fR \fBallow\fR \fB-c\fR \fIperm\fR|\fIsetname\fR[,...] \fIfilesystem\fR|
179 .fi

181 .LP
182 .nf
183 \fBzfs\fR \fBallow\fR \fB-s\fR @\fIsetname\fR \fIperm\fR|\fIsetname\fR[,...] \f
184 .fi

```

```

186 .LP
187 .nf
188 \fBzfs\fR \fBunallow\fR [\fB-rldug\fR] "\fIeveryone\fR"|\fIuser\fR|\fIgroup\fR[ ,
189 \fIfilesystem\fR|\fIvolume\fR
190 .fi

192 .LP
193 .nf
194 \fBzfs\fR \fBunallow\fR [\fB-rld\fR] \fB-e\fR [\fIperm\fR|\fIsetname\fR[,... ]]
195 .fi

197 .LP
198 .nf
199 \fBzfs\fR \fBunallow\fR [\fB-r\fR] \fB-c\fR [\fIperm\fR|\fIsetname\fR[ ... ]] \
200 .fi

202 .LP
203 .nf
204 \fBzfs\fR \fBunallow\fR [\fB-r\fR] \fB-s\fR @\fIsetname\fR [\fIperm\fR|\fIsetna
205 .fi

207 .LP
208 .nf
209 \fBzfs\fR \fBhold\fR [\fB-r\fR] \fItag\fR \fIsnapshot\fR...
210 .fi

212 .LP
213 .nf
214 \fBzfs\fR \fBholds\fR [\fB-r\fR] \fIsnapshot\fR...
215 .fi

217 .LP
218 .nf
219 \fBzfs\fR \fBrelease\fR [\fB-r\fR] \fItag\fR \fIsnapshot\fR...
220 .fi

222 .SH DESCRIPTION
223 .sp
224 .LP
225 The \fBzfs\fR command configures \fBZFS\fR datasets within a \fBZFS\fR storage
226 pool, as described in \fBzpool\fR(1M). A dataset is identified by a unique path
227 within the \fBZFS\fR namespace. For example:
228 .sp
229 .in +2
230 .nf
231 pool/{filesystem,volume,snapshot}
232 .fi
233 .in -2
234 .sp

236 .sp
237 .LP
238 where the maximum length of a dataset name is \fBMAXNAMELEN\fR (256 bytes).
239 .sp
240 .LP
241 A dataset can be one of the following:
242 .sp
243 .ne 2
244 .na
245 \fBfile system\fR
246 .ad
247 .sp .6
248 .RS 4n
249 A \fBZFS\fR dataset of type \fBfilesystem\fR can be mounted within the standard
250 system namespace and behaves like other file systems. While \fBZFS\fR file
251 systems are designed to be \fBPOSIX\fR compliant, known issues exist that

```

252 prevent compliance in some cases. Applications that depend on standards
 253 conformance might fail due to nonstandard behavior when checking file system
 254 free space.
 255 .RE

257 .sp
 258 .ne 2
 259 .na
 260 \fB\fIvolume\fR\fR
 261 .ad
 262 .sp .6
 263 .RS 4n
 264 A logical volume exported as a raw or block device. This type of dataset should
 265 only be used under special circumstances. File systems are typically used in
 266 most environments.
 267 .RE

269 .sp
 270 .ne 2
 271 .na
 272 \fB\fIsnapshot\fR\fR
 273 .ad
 274 .sp .6
 275 .RS 4n
 276 A read-only version of a file system or volume at a given point in time. It is
 277 specified as \fIfilesystem@name\fR or \fIvolume@name\fR.
 278 .RE

280 .SS "ZFS File System Hierarchy"
 281 .sp
 282 .LP
 283 A \fBZFS\fR storage pool is a logical collection of devices that provide space
 284 for datasets. A storage pool is also the root of the \fBZFS\fR file system
 285 hierarchy.
 286 .sp
 287 .LP
 288 The root of the pool can be accessed as a file system, such as mounting and
 289 unmounting, taking snapshots, and setting properties. The physical storage
 290 characteristics, however, are managed by the \fBzpool\fR(1M) command.
 291 .sp
 292 .LP
 293 See \fBzpool\fR(1M) for more information on creating and administering pools.
 294 .SS "Snapshots"
 295 .sp
 296 .LP
 297 A snapshot is a read-only copy of a file system or volume. Snapshots can be
 298 created extremely quickly, and initially consume no additional space within the
 299 pool. As data within the active dataset changes, the snapshot consumes more
 300 data than would otherwise be shared with the active dataset.
 301 .sp
 302 .LP
 303 Snapshots can have arbitrary names. Snapshots of volumes can be cloned or
 304 rolled back, but cannot be accessed independently.
 305 .sp
 306 .LP
 307 File system snapshots can be accessed under the \fB&.zfs/snapshot\fR directory
 308 in the root of the file system. Snapshots are automatically mounted on demand
 309 and may be unmounted at regular intervals. The visibility of the \fB&.zfs\fR
 310 directory can be controlled by the \fBsnapdir\fR property.
 311 .SS "Clones"
 312 .sp
 313 .LP
 314 A clone is a writable volume or file system whose initial contents are the same
 315 as another dataset. As with snapshots, creating a clone is nearly
 316 instantaneous, and initially consumes no additional space.
 317 .sp

318 .LP
 319 Clones can only be created from a snapshot. When a snapshot is cloned, it
 320 creates an implicit dependency between the parent and child. Even though the
 321 clone is created somewhere else in the dataset hierarchy, the original snapshot
 322 cannot be destroyed as long as a clone exists. The \fBborigin\fR property
 323 exposes this dependency, and the \fBdestroy\fR command lists any such
 324 dependencies, if they exist.
 325 .sp
 326 .LP
 327 The clone parent-child dependency relationship can be reversed by using the
 328 \fBpromote\fR subcommand. This causes the "origin" file system to become a
 329 clone of the specified file system, which makes it possible to destroy the file
 330 system that the clone was created from.
 331 .SS "Mount Points"
 332 .sp
 333 .LP
 334 Creating a \fBZFS\fR file system is a simple operation, so the number of file
 335 systems per system is likely to be numerous. To cope with this, \fBZFS\fR
 336 automatically manages mounting and unmounting file systems without the need to
 337 edit the \fB/etc/vfstab\fR file. All automatically managed file systems are
 338 mounted by \fBZFS\fR at boot time.
 339 .sp
 340 .LP
 341 By default, file systems are mounted under \fB/\fIpath\fR\fR, where \fIpath\fR
 342 is the name of the file system in the \fBZFS\fR namespace. Directories are
 343 created and destroyed as needed.
 344 .sp
 345 .LP
 346 A file system can also have a mount point set in the \fBmountpoint\fR property.
 347 This directory is created as needed, and \fBZFS\fR automatically mounts the
 348 file system when the \fBzfs mount -a\fR command is invoked (without editing
 349 \fB/etc/vfstab\fR). The \fBmountpoint\fR property can be inherited, so if
 350 \fBpool/home\fR has a mount point of \fB/export/stuff\fR, then
 351 \fBpool/home/user\fR automatically inherits a mount point of
 352 \fB/export/stuff/user\fR.
 353 .sp
 354 .LP
 355 A file system \fBmountpoint\fR property of \fBnone\fR prevents the file system
 356 from being mounted.
 357 .sp
 358 .LP
 359 If needed, \fBZFS\fR file systems can also be managed with traditional tools
 360 (\fBmount\fR, \fBumount\fR, \fB/etc/vfstab\fR). If a file system's mount point
 361 is set to \fBlegacy\fR, \fBZFS\fR makes no attempt to manage the file system,
 362 and the administrator is responsible for mounting and unmounting the file
 363 system.
 364 .SS "Zones"
 365 .sp
 366 .LP
 367 A \fBZFS\fR file system can be added to a non-global zone by using the
 368 \fBzonecfg\fR \fBadd fs\fR subcommand. A \fBZFS\fR file system that is added to
 369 a non-global zone must have its \fBmountpoint\fR property set to \fBlegacy\fR.
 370 .sp
 371 .LP
 372 The physical properties of an added file system are controlled by the global
 373 administrator. However, the zone administrator can create, modify, or destroy
 374 files within the added file system, depending on how the file system is
 375 mounted.
 376 .sp
 377 .LP
 378 A dataset can also be delegated to a non-global zone by using the \fBzonecfg\fR
 379 \fBadd dataset\fR subcommand. You cannot delegate a dataset to one zone and the
 380 children of the same dataset to another zone. The zone administrator can change
 381 properties of the dataset or any of its children. However, the \fBquota\fR
 382 property is controlled by the global administrator.
 383 .sp

```

384 .LP
385 A \fBZFS\fR volume can be added as a device to a non-global zone by using the
386 \fBzonecfg\fR \fBadd device\fR subcommand. However, its physical properties can
387 be modified only by the global administrator.
388 .sp
389 .LP
390 For more information about \fBzonecfg\fR syntax, see \fBzonecfg\fR(1M).
391 .sp
392 .LP
393 After a dataset is delegated to a non-global zone, the \fBzoned\fR property is
394 automatically set. A zoned file system cannot be mounted in the global zone,
395 since the zone administrator might have to set the mount point to an
396 unacceptable value.
397 .sp
398 .LP
399 The global administrator can forcibly clear the \fBzoned\fR property, though
400 this should be done with extreme care. The global administrator should verify
401 that all the mount points are acceptable before clearing the property.
402 .SS "Native Properties"
403 .sp
404 .LP
405 Properties are divided into two types, native properties and user-defined (or
406 "user") properties. Native properties either export internal statistics or
407 control \fBZFS\fR behavior. In addition, native properties are either editable
408 or read-only. User properties have no effect on \fBZFS\fR behavior, but you can
409 use them to annotate datasets in a way that is meaningful in your environment.
410 For more information about user properties, see the "User Properties" section,
411 below.
412 .sp
413 .LP
414 Every dataset has a set of properties that export statistics about the dataset
415 as well as control various behaviors. Properties are inherited from the parent
416 unless overridden by the child. Some properties apply only to certain types of
417 datasets (file systems, volumes, or snapshots).
418 .sp
419 .LP
420 The values of numeric properties can be specified using human-readable suffixes
421 (for example, \fBk\fR, \fBKB\fR, \fBMB\fR, \fBGB\fR, and so forth, up to \fBZ\fR
422 for zettabyte). The following are all valid (and equal) specifications:
423 .sp
424 .in +2
425 .nf
426 1536M, 1.5g, 1.50GB
427 .fi
428 .in -2
429 .sp
431 .sp
432 .LP
433 The values of non-numeric properties are case sensitive and must be lowercase,
434 except for \fBmountpoint\fR, \fBsharename\fR, and \fBsharesmb\fR.
435 .sp
436 .LP
437 The following native properties consist of read-only statistics about the
438 dataset. These properties can be neither set, nor inherited. Native properties
439 apply to all dataset types unless otherwise noted.
440 .sp
441 .ne 2
442 .na
443 \fBbavail\fR
444 .ad
445 .sp .6
446 .RS 4n
447 The amount of space available to the dataset and all its children, assuming
448 that there is no other activity in the pool. Because space is shared within a
449 pool, availability can be limited by any number of factors, including physical

```

```

450 pool size, quotas, reservations, or other datasets within the pool.
451 .sp
452 This property can also be referred to by its shortened column name,
453 \fBavail\fR.
454 .RE
456 .sp
457 .ne 2
458 .na
459 \fBbcompressratio\fR
460 .ad
461 .sp .6
462 .RS 4n
463 For non-snapshots, the compression ratio achieved for the \fBused\fR
464 space of this dataset, expressed as a multiplier. The \fBused\fR
465 property includes descendant datasets, and, for clones, does not include
466 the space shared with the origin snapshot. For snapshots, the
467 \fBcompressratio\fR is the same as the \fBrefcompressratio\fR property.
468 Compression can be turned on by running: \fBzfs set compression=on
469 \fRdataset\fR. The default value is \fBoff\fR.
470 .RE
472 .sp
473 .ne 2
474 .na
475 \fBbcreation\fR
476 .ad
477 .sp .6
478 .RS 4n
479 The time this dataset was created.
480 .RE
482 .sp
483 .ne 2
484 .na
485 \fBbclones\fR
486 .ad
487 .sp .6
488 .RS 4n
489 For snapshots, this property is a comma-separated list of filesystems or
490 volumes which are clones of this snapshot. The clones' \fBorigin\fR property
491 is this snapshot. If the \fBclones\fR property is not empty, then this
492 snapshot can not be destroyed (even with the \fB-r\fR or \fB-f\fR options).
493 .RE
495 .sp
496 .ne 2
497 .na
498 \fBbdefer_destroy\fR
499 .ad
500 .sp .6
501 .RS 4n
502 This property is \fBon\fR if the snapshot has been marked for deferred destroy
503 by using the \fBzfs destroy\fR \fB-d\fR command. Otherwise, the property is
504 \fBoff\fR.
505 .RE
507 .sp
508 .ne 2
509 .na
510 \fBbmounted\fR
511 .ad
512 .sp .6
513 .RS 4n
514 For file systems, indicates whether the file system is currently mounted. This
515 property can be either \fByes\fR or \fBno\fR.

```

```

516 .RE
518 .sp
519 .ne 2
520 .na
521 \fB\fBorigin\fR\fR
522 .ad
523 .sp .6
524 .RS 4n
525 For cloned file systems or volumes, the snapshot from which the clone was
526 created. See also the \fB\ clones\fR property.
527 .RE
529 .sp
530 .ne 2
531 .na
532 \fB\fBreferenced\fR\fR
533 .ad
534 .sp .6
535 .RS 4n
536 The amount of data that is accessible by this dataset, which may or may not be
537 shared with other datasets in the pool. When a snapshot or clone is created, it
538 initially references the same amount of space as the file system or snapshot it
539 was created from, since its contents are identical.
540 .sp
541 This property can also be referred to by its shortened column name,
542 \fB\ refer\fR.
543 .RE
545 .sp
546 .ne 2
547 .na
548 \fB\fBrefcompressratio\fR\fR
549 .ad
550 .sp .6
551 .RS 4n
552 The compression ratio achieved for the \fB\ referenced\fR space of this
553 dataset, expressed as a multiplier. See also the \fB\ compressratio\fR
554 property.
555 .RE
557 .sp
558 .ne 2
559 .na
560 \fB\fBtype\fR\fR
561 .ad
562 .sp .6
563 .RS 4n
564 The type of dataset: \fB\ filesystem\fR, \fB\ volume\fR, or \fB\ snapshot\fR.
565 .RE
567 .sp
568 .ne 2
569 .na
570 \fB\fBused\fR\fR
571 .ad
572 .sp .6
573 .RS 4n
574 The amount of space consumed by this dataset and all its descendents. This is
575 the value that is checked against this dataset's quota and reservation. The
576 space used does not include this dataset's reservation, but does take into
577 account the reservations of any descendent datasets. The amount of space that a
578 dataset consumes from its parent, as well as the amount of space that are freed
579 if this dataset is recursively destroyed, is the greater of its space used and
580 its reservation.
581 .sp

```

```

582 When snapshots (see the "Snapshots" section) are created, their space is
583 initially shared between the snapshot and the file system, and possibly with
584 previous snapshots. As the file system changes, space that was previously
585 shared becomes unique to the snapshot, and counted in the snapshot's space
586 used. Additionally, deleting snapshots can increase the amount of space unique
587 to (and used by) other snapshots.
588 .sp
589 The amount of space used, available, or referenced does not take into account
590 pending changes. Pending changes are generally accounted for within a few
591 seconds. Committing a change to a disk using \fB\ fsync\fR(3c) or \fB\ sync\fR
592 does not necessarily guarantee that the space usage information is updated
593 immediately.
594 .RE
596 .sp
597 .ne 2
598 .na
599 \fB\fBusedby*\fR\fR
600 .ad
601 .sp .6
602 .RS 4n
603 The \fB\ usedby*\fR properties decompose the \fB\ used\fR properties into the
604 various reasons that space is used. Specifically, \fB\ used\fR =
605 \fB\ usedbychildren\fR + \fB\ usedbydataset\fR + \fB\ usedbyreservation\fR +
606 \fB\ usedbysnapshots\fR. These properties are only available for datasets created
607 on \fB\ zpools\fR "version 13" pools.
608 .RE
610 .sp
611 .ne 2
612 .na
613 \fB\fBusedbychildren\fR\fR
614 .ad
615 .sp .6
616 .RS 4n
617 The amount of space used by children of this dataset, which would be freed if
618 all the dataset's children were destroyed.
619 .RE
621 .sp
622 .ne 2
623 .na
624 \fB\fBusedbydataset\fR\fR
625 .ad
626 .sp .6
627 .RS 4n
628 The amount of space used by this dataset itself, which would be freed if the
629 dataset were destroyed (after first removing any \fB\ reservation\fR and
630 destroying any necessary snapshots or descendents).
631 .RE
633 .sp
634 .ne 2
635 .na
636 \fB\fBusedbyreservation\fR\fR
637 .ad
638 .sp .6
639 .RS 4n
640 The amount of space used by a \fB\ reservation\fR set on this dataset, which
641 would be freed if the \fB\ reservation\fR was removed.
642 .RE
644 .sp
645 .ne 2
646 .na
647 \fB\fBusedbysnapshots\fR\fR

```

```

648 .ad
649 .sp .6
650 .RS 4n
651 The amount of space consumed by snapshots of this dataset. In particular, it is
652 the amount of space that would be freed if all of this dataset's snapshots were
653 destroyed. Note that this is not simply the sum of the snapshots' \fBused\fR
654 properties because space can be shared by multiple snapshots.
655 .RE

657 .sp
658 .ne 2
659 .na
660 \fB\fBuserused@\fR\fR\fR user's space usage. The root user, or a
661 user who has been granted the \fBzfs allow\fR privilege with \fBzfs allow\fR,
662 can access everyone's usage.
663 .RS 4n
664 The amount of space consumed by the specified user in this dataset. Space is
665 charged to the owner of each file, as displayed by \fBls\fR \fB-l\fR. The
666 amount of space charged is displayed by \fBdu\fR and \fBls\fR \fB-s\fR. See the
667 \fBzfs userspace\fR subcommand for more information.
668 .sp
669 Unprivileged users can access only their own space usage. The root user, or a
670 user who has been granted the \fBzfs allow\fR privilege with \fBzfs allow\fR,
671 can access everyone's usage.
672 .sp
673 The \fBuserused@\fR... properties are not displayed by \fBzfs get all\fR. The
674 user's name must be appended after the \fB@\fR symbol, using one of the
675 following forms:
676 .RS +4
677 .TP
678 .ie t \(\bu
679 .el o
680 \fB\fIPOSIX name\fR (for example, \fBjoe\fR)
681 .RE
682 .RS +4
683 .TP
684 .ie t \(\bu
685 .el o
686 \fB\fIPOSIX numeric ID\fR (for example, \fB789\fR)
687 .RE
688 .RS +4
689 .TP
690 .ie t \(\bu
691 .el o
692 \fB\fISID name\fR (for example, \fBjoe.smith@mydomain\fR)
693 .RE
694 .RS +4
695 .TP
696 .ie t \(\bu
697 .el o
698 \fB\fISID numeric ID\fR (for example, \fBS-1-123-456-789\fR)
699 .RE
700 .RE

702 .sp
703 .ne 2
704 .na
705 \fB\fBuserrefs\fR user's space usage. The root user, or a
706 user who has been granted the \fBzfs allow\fR privilege with \fBzfs allow\fR,
707 can access everyone's usage.
708 .RS 4n
709 This property is set to the number of user holds on this snapshot. User holds
710 are set by using the \fBzfs hold\fR command.
711 .RE

713 .sp

```

```

714 .ne 2
715 .na
716 \fB\fBgroupused@\fR\fR\fR group's space usage. The root user, or a
717 user who has been granted the \fBzfs allow\fR privilege with \fBzfs allow\fR,
718 can access everyone's usage.
719 .RS 4n
720 The amount of space consumed by the specified group in this dataset. Space is
721 charged to the group of each file, as displayed by \fBls\fR \fB-l\fR. See the
722 \fBzfs groupused@\fR\fR\fR property for more information.
723 .sp
724 Unprivileged users can only access their own groups' space usage. The root
725 user, or a user who has been granted the \fBzfs allow\fR privilege with \fBzfs
726 allow\fR, can access all groups' usage.
727 .RE

729 .sp
730 .ne 2
731 .na
732 \fB\fBvolblocksize\fR=\fR block size of the volume. The \fBblocksize\fR cannot
733 be changed once the volume has been written, so it should be set at volume
734 creation time. The default \fBblocksize\fR for volumes is 8 Kbytes. Any power
735 of 2 from 512 bytes to 128 Kbytes is valid.
736 .RS 4n
737 For volumes, specifies the block size of the volume. The \fBblocksize\fR cannot
738 be changed once the volume has been written, so it should be set at volume
739 creation time. The default \fBblocksize\fR for volumes is 8 Kbytes. Any power
740 of 2 from 512 bytes to 128 Kbytes is valid.
741 .sp
742 This property can also be referred to by its shortened column name,
743 \fBvolblock\fR.
744 .RE

745 .sp
746 .ne 2
747 .na
748 \fB\fBwritten\fR space written to this dataset since the
749 previous snapshot.
750 .RS 4n
751 The amount of \fBreferenced\fR space written to this dataset since the
752 previous snapshot.
753 .RE

755 .sp
756 .ne 2
757 .na
758 \fB\fBwritten@\fR\fR\fR space written to this dataset since the
759 previous snapshot. This is the space that is referenced by this dataset
760 but was not referenced by the specified snapshot.
761 .RS 4n
762 The amount of \fBreferenced\fR space written to this dataset since the
763 specified snapshot. This is the space that is referenced by this dataset
764 but was not referenced by the specified snapshot.
765 .sp
766 The \fBsnapshot\fR may be specified as a short snapshot name (just the part
767 after the \fB@\fR), in which case it will be interpreted as a snapshot in
768 the same filesystem as this dataset.
769 The \fBsnapshot\fR be a full snapshot name (\fBfilesystem\fR@\fBsnapshot\fR),
770 which for clones may be a snapshot in the origin's filesystem (or the origin
771 of the origin's filesystem, etc).
772 .RE

775 .sp
776 .LP
777 The following native properties can be used to change the behavior of a
778 \fBZFS\fR dataset.
779 .sp

```

```

780 .ne 2
781 .na
782 \fB\fBaclinherit\fR=\fBdiscard\fR | \fBnoallow\fR | \fBrestricted\fR |
783 \fBpassthrough\fR | \fBpassthrough-x\fR\fR
784 .ad
785 .sp .6
786 .RS 4n
787 Controls how \fBACL\fR entries are inherited when files and directories are
788 created. A file system with an \fBaclinherit\fR property of \fBdiscard\fR does
789 not inherit any \fBACL\fR entries. A file system with an \fBaclinherit\fR
790 property value of \fBnoallow\fR only inherits inheritable \fBACL\fR entries
791 that specify "deny" permissions. The property value \fBrestricted\fR (the
792 default) removes the \fBwrite_acl\fR and \fBwrite_owner\fR permissions when the
793 \fBACL\fR entry is inherited. A file system with an \fBaclinherit\fR property
794 value of \fBpassthrough\fR inherits all inheritable \fBACL\fR entries without
795 any modifications made to the \fBACL\fR entries when they are inherited. A file
796 system with an \fBaclinherit\fR property value of \fBpassthrough-x\fR has the
797 same meaning as \fBpassthrough\fR, except that the \fBowner@\fR, \fBgroup@\fR,
798 and \fBeveryone@\fR \fBACE\fRs inherit the execute permission only if the file
799 creation mode also requests the execute bit.
800 .sp
801 When the property value is set to \fBpassthrough\fR, files are created with a
802 mode determined by the inheritable \fBACE\fRs. If no inheritable \fBACE\fRs
803 exist that affect the mode, then the mode is set in accordance to the requested
804 mode from the application.
805 .RE

807 .sp
808 .ne 2
809 .na
810 \fB\fBaclmode\fR=\fBdiscard\fR | \fBgroupmask\fR | \fBpassthrough\fR\fR
811 .ad
812 .sp .6
813 .RS 4n
814 Controls how an \fBACL\fR is modified during \fBchmod\fR(2). A file system with
815 an \fBaclmode\fR property of \fBdiscard\fR (the default) deletes all \fBACL\fR
816 entries that do not represent the mode of the file. An \fBaclmode\fR property
817 of \fBgroupmask\fR reduces permissions granted in all \fBALLLOW\fR entries found
818 in the \fBACL\fR such that they are no greater than the group permissions
819 specified by \fBchmod\fR. A file system with an \fBaclmode\fR property of
820 \fBpassthrough\fR indicates that no changes are made to the \fBACL\fR other
821 than creating or updating the necessary \fBACL\fR entries to
822 represent the new mode of the file or directory.
823 .RE

825 .sp
826 .ne 2
827 .na
828 \fB\fBbatime\fR=\fBon\fR | \fBoff\fR\fR
829 .ad
830 .sp .6
831 .RS 4n
832 Controls whether the access time for files is updated when they are read.
833 Turning this property off avoids producing write traffic when reading files and
834 can result in significant performance gains, though it might confuse mailers
835 and other similar utilities. The default value is \fBon\fR.
836 .RE

838 .sp
839 .ne 2
840 .na
841 \fB\fBcanmount\fR=\fBon\fR | \fBoff\fR | \fBnoauto\fR\fR
842 .ad
843 .sp .6
844 .RS 4n
845 If this property is set to \fBoff\fR, the file system cannot be mounted, and is

```

```

846 ignored by \fBzfs mount -a\fR. Setting this property to \fBoff\fR is similar to
847 setting the \fBmountpoint\fR property to \fBnone\fR, except that the dataset
848 still has a normal \fBmountpoint\fR property, which can be inherited. Setting
849 this property to \fBoff\fR allows datasets to be used solely as a mechanism to
850 inherit properties. One example of setting \fBcanmount=\fR\fBoff\fR is to have
851 two datasets with the same \fBmountpoint\fR, so that the children of both
852 datasets appear in the same directory, but might have different inherited
853 characteristics.
854 .sp
855 When the \fBnoauto\fR option is set, a dataset can only be mounted and
856 unmounted explicitly. The dataset is not mounted automatically when the dataset
857 is created or imported, nor is it mounted by the \fBzfs mount -a\fR command or
858 unmounted by the \fBzfs unmount -a\fR command.
859 .sp
860 This property is not inherited.
861 .RE

863 .sp
864 .ne 2
865 .na
866 \fB\fBchecksum\fR=\fBon\fR | \fBoff\fR | \fBfletcher2,\fR | \fBfletcher4\fR |
867 \fBsha256\fR\fR
868 .ad
869 .sp .6
870 .RS 4n
871 Controls the checksum used to verify data integrity. The default value is
872 \fBon\fR, which automatically selects an appropriate algorithm (currently,
873 \fBfletcher2\fR, but this may change in future releases). The value \fBoff\fR
874 disables integrity checking on user data. Disabling checksums is \fBNOT\fR a
875 recommended practice.
876 .sp
877 Changing this property affects only newly-written data.
878 .RE

880 .sp
881 .ne 2
882 .na
883 \fB\fBcompression\fR=\fBon\fR | \fBoff\fR | \fBlzjb\fR | \fBgzip\fR |
884 \fBgzip-\fR\fIN\fR | \fBzle\fR\fR
885 .ad
886 .sp .6
887 .RS 4n
888 Controls the compression algorithm used for this dataset. The \fBlzjb\fR
889 compression algorithm is optimized for performance while providing decent data
890 compression. Setting compression to \fBon\fR uses the \fBlzjb\fR compression
891 algorithm. The \fBgzip\fR compression algorithm uses the same compression as
892 the \fBgzip\fR(1) command. You can specify the \fBgzip\fR level by using the
893 value \fBgzip-\fR\fIN\fR where \fIN\fR is an integer from 1 (fastest) to 9
894 (best compression ratio). Currently, \fBgzip\fR is equivalent to \fBgzip-6\fR
895 (which is also the default for \fBgzip\fR(1)). The \fBzle\fR compression
896 algorithm compresses runs of zeros.
897 .sp
898 This property can also be referred to by its shortened column name
899 \fBcompress\fR. Changing this property affects only newly-written data.
900 .RE

902 .sp
903 .ne 2
904 .na
905 \fB\fBcopies\fR=\fB1\fR | \fB2\fR | \fB3\fR\fR
906 .ad
907 .sp .6
908 .RS 4n
909 Controls the number of copies of data stored for this dataset. These copies are
910 in addition to any redundancy provided by the pool, for example, mirroring or
911 RAID-Z. The copies are stored on different disks, if possible. The space used

```

```

912 by multiple copies is charged to the associated file and dataset, changing the
913 \fBused\fR property and counting against quotas and reservations.
914 .sp
915 Changing this property only affects newly-written data. Therefore, set this
916 property at file system creation time by using the \fB-o\fR
917 \fBcopies=\fR\fIIN\fR option.
918 .RE

920 .sp
921 .ne 2
922 .na
923 \fB\bdevices\fR=\fBon\fR | \fBoff\fR\fR
924 .ad
925 .sp .6
926 .RS 4n
927 Controls whether device nodes can be opened on this file system. The default
928 value is \fBon\fR.
929 .RE

931 .sp
932 .ne 2
933 .na
934 \fB\bexec\fR=\fBon\fR | \fBoff\fR\fR
935 .ad
936 .sp .6
937 .RS 4n
938 Controls whether processes can be executed from within this file system. The
939 default value is \fBon\fR.
940 .RE

942 .sp
943 .ne 2
944 .na
945 \fB\bmountpoint\fR=\fIpath\fR | \fBnone\fR | \fBlegacy\fR\fR
946 .ad
947 .sp .6
948 .RS 4n
949 Controls the mount point used for this file system. See the "Mount Points"
950 section for more information on how this property is used.
951 .sp
952 When the \fBmountpoint\fR property is changed for a file system, the file
953 system and any children that inherit the mount point are unmounted. If the new
954 value is \fBlegacy\fR, then they remain unmounted. Otherwise, they are
955 automatically remounted in the new location if the property was previously
956 \fBlegacy\fR or \fBnone\fR, or if they were mounted before the property was
957 changed. In addition, any shared file systems are unshared and shared in the
958 new location.
959 .RE

961 .sp
962 .ne 2
963 .na
964 \fB\bmband\fR=\fBon\fR | \fBoff\fR\fR
965 .ad
966 .sp .6
967 .RS 4n
968 Controls whether the file system should be mounted with \fBmband\fR (Non
969 Blocking mandatory locks). This is used for \fBCIFS\fR clients. Changes to this
970 property only take effect when the file system is unmounted and remounted. See
971 \fBmount\fR(1M) for more information on \fBmband\fR mounts.
972 .RE

974 .sp
975 .ne 2
976 .na
977 \fB\bprimarycache\fR=\fBall\fR | \fBnone\fR | \fBmetadata\fR\fR

```

```

978 .ad
979 .sp .6
980 .RS 4n
981 Controls what is cached in the primary cache (ARC). If this property is set to
982 \fBall\fR, then both user data and metadata is cached. If this property is set
983 to \fBnone\fR, then neither user data nor metadata is cached. If this property
984 is set to \fBmetadata\fR, then only metadata is cached. The default value is
985 \fBall\fR.
986 .RE

988 .sp
989 .ne 2
990 .na
991 \fB\bquota\fR=\fIsize\fR | \fBnone\fR\fR
992 .ad
993 .sp .6
994 .RS 4n
995 Limits the amount of space a dataset and its descendants can consume. This
996 property enforces a hard limit on the amount of space used. This includes all
997 space consumed by descendants, including file systems and snapshots. Setting a
998 quota on a descendent of a dataset that already has a quota does not override
999 the ancestor's quota, but rather imposes an additional limit.
1000 .sp
1001 Quotas cannot be set on volumes, as the \fBvolsize\fR property acts as an
1002 implicit quota.
1003 .RE

1005 .sp
1006 .ne 2
1007 .na
1008 \fB\buserquota@\fR\fIuser\fR=\fIsize\fR | \fBnone\fR\fR
1009 .ad
1010 .sp .6
1011 .RS 4n
1012 Limits the amount of space consumed by the specified user. User space
1013 consumption is identified by the \fBuserspace@\fR\fIuser\fR property.
1014 .sp
1015 Enforcement of user quotas may be delayed by several seconds. This delay means
1016 that a user might exceed their quota before the system notices that they are
1017 over quota and begins to refuse additional writes with the \fBEDQUOT\fR error
1018 message . See the \fBzfs userspace\fR subcommand for more information.
1019 .sp
1020 Unprivileged users can only access their own groups' space usage. The root
1021 user, or a user who has been granted the \fBuserquota\fR privilege with \fBzfs
1022 allow\fR, can get and set everyone's quota.
1023 .sp
1024 This property is not available on volumes, on file systems before version 4, or
1025 on pools before version 15. The \fBuserquota@\fR... properties are not
1026 displayed by \fBzfs get all\fR. The user's name must be appended after the
1027 \fB@\fR symbol, using one of the following forms:
1028 .RS +4
1029 .TP
1030 .ie t \(\bu
1031 .el o
1032 \fIPOSIX name\fR (for example, \fBjoe\fR)
1033 .RE
1034 .RS +4
1035 .TP
1036 .ie t \(\bu
1037 .el o
1038 \fIPOSIX numeric ID\fR (for example, \fB789\fR)
1039 .RE
1040 .RS +4
1041 .TP
1042 .ie t \(\bu
1043 .el o

```



```

1044 \fISID name\fR (for example, \fBjoe.smith@mydomain\fR)
1045 .RE
1046 .RS +4
1047 .TP
1048 .ie t \(\bu
1049 .el o
1050 \fISID numeric ID\fR (for example, \fBS-1-123-456-789\fR)
1051 .RE
1052 .RE

1054 .sp
1055 .ne 2
1056 .na
1057 \fB\fBgroupquota@\fR\fIgroup\fR=\fIsize\fR | \fBNone\fR\fR
1058 .ad
1059 .sp .6
1060 .RS 4n
1061 Limits the amount of space consumed by the specified group. Group space
1062 consumption is identified by the \fBuserquota@\fR\fIuser\fR property.
1063 .sp
1064 Unprivileged users can access only their own groups' space usage. The root
1065 user, or a user who has been granted the \fBgroupquota\fR privilege with \fBzfs
1066 allow\fR, can get and set all groups' quotas.
1067 .RE

1069 .sp
1070 .ne 2
1071 .na
1072 \fB\fBreadonly\fR=\fBon\fR | \fBoff\fR\fR
1073 .ad
1074 .sp .6
1075 .RS 4n
1076 Controls whether this dataset can be modified. The default value is \fBoff\fR.
1077 .sp
1078 This property can also be referred to by its shortened column name,
1079 \fBrdonly\fR.
1080 .RE

1082 .sp
1083 .ne 2
1084 .na
1085 \fB\fBrecordsize\fR=\fIsize\fR\fR
1086 .ad
1087 .sp .6
1088 .RS 4n
1089 Specifies a suggested block size for files in the file system. This property is
1090 designed solely for use with database workloads that access files in fixed-size
1091 records. \fBZFS\fR automatically tunes block sizes according to internal
1092 algorithms optimized for typical access patterns.
1093 .sp
1094 For databases that create very large files but access them in small random
1095 chunks, these algorithms may be suboptimal. Specifying a \fBrecordsize\fR
1096 greater than or equal to the record size of the database can result in
1097 significant performance gains. Use of this property for general purpose file
1098 systems is strongly discouraged, and may adversely affect performance.
1099 .sp
1100 The size specified must be a power of two greater than or equal to 512 and less
1101 than or equal to 128 Kbytes.
1102 .sp
1103 Changing the file system's \fBrecordsize\fR affects only files created
1104 afterward; existing files are unaffected.
1105 .sp
1106 This property can also be referred to by its shortened column name,
1107 \fBrecsize\fR.
1108 .RE

```

```

1110 .sp
1111 .ne 2
1112 .na
1113 \fB\fBrefquota\fR=\fIsize\fR | \fBNone\fR\fR
1114 .ad
1115 .sp .6
1116 .RS 4n
1117 Limits the amount of space a dataset can consume. This property enforces a hard
1118 limit on the amount of space used. This hard limit does not include space used
1119 by descendants, including file systems and snapshots.
1120 .RE

1122 .sp
1123 .ne 2
1124 .na
1125 \fB\fBrefreservation\fR=\fIsize\fR | \fBNone\fR\fR
1126 .ad
1127 .sp .6
1128 .RS 4n
1129 The minimum amount of space guaranteed to a dataset, not including its
1130 descendants. When the amount of space used is below this value, the dataset is
1131 treated as if it were taking up the amount of space specified by
1132 \fBrefreservation\fR. The \fBrefreservation\fR reservation is accounted for in
1133 the parent datasets' space used, and counts against the parent datasets' quotas
1134 and reservations.
1135 .sp
1136 If \fBrefreservation\fR is set, a snapshot is only allowed if there is enough
1137 free pool space outside of this reservation to accommodate the current number
1138 of "referenced" bytes in the dataset.
1139 .sp
1140 This property can also be referred to by its shortened column name,
1141 \fBrefreserv\fR.
1142 .RE

1144 .sp
1145 .ne 2
1146 .na
1147 \fB\fBreservation\fR=\fIsize\fR | \fBNone\fR\fR
1148 .ad
1149 .sp .6
1150 .RS 4n
1151 The minimum amount of space guaranteed to a dataset and its descendants. When
1152 the amount of space used is below this value, the dataset is treated as if it
1153 were taking up the amount of space specified by its reservation. Reservations
1154 are accounted for in the parent datasets' space used, and count against the
1155 parent datasets' quotas and reservations.
1156 .sp
1157 This property can also be referred to by its shortened column name,
1158 \fBreserv\fR.
1159 .RE

1161 .sp
1162 .ne 2
1163 .na
1164 \fB\fBsecondarycache\fR=\fBall\fR | \fBNone\fR | \fBmetadata\fR\fR
1165 .ad
1166 .sp .6
1167 .RS 4n
1168 Controls what is cached in the secondary cache (L2ARC). If this property is set
1169 to \fBall\fR, then both user data and metadata is cached. If this property is
1170 set to \fBNone\fR, then neither user data nor metadata is cached. If this
1171 property is set to \fBmetadata\fR, then only metadata is cached. The default
1172 value is \fBall\fR.
1173 .RE

1175 .sp

```

1176 .ne 2
 1177 .na
 1178 \fB\fBsetuid\fR=\fBon\fR | \fBoff\fR
 1179 .ad
 1180 .sp .6
 1181 .RS 4n
 1182 Controls whether the set-\fBUID\fR bit is respected for the file system. The
 1183 default value is \fBon\fR.
 1184 .RE

1186 .sp
 1187 .ne 2
 1188 .na
 1189 \fB\fBshareiscsi\fR=\fBon\fR | \fBoff\fR
 1190 .ad
 1191 .sp .6
 1192 .RS 4n
 1193 Like the \fBsharenfs\fR property, \fBshareiscsi\fR indicates whether a
 1194 \fBZFS\fR volume is exported as an \fBiSCSI\fR target. The acceptable values
 1195 for this property are \fBon\fR, \fBoff\fR, and \fBtype=disk\fR. The default
 1196 value is \fBoff\fR. In the future, other target types might be supported. For
 1197 example, \fBtape\fR.
 1198 .sp
 1199 You might want to set \fBshareiscsi=on\fR for a file system so that all
 1200 \fBZFS\fR volumes within the file system are shared by default. However,
 1201 setting this property on a file system has no direct effect.
 1202 .RE

1204 .sp
 1205 .ne 2
 1206 .na
 1207 \fB\fBsharesmb\fR=\fBon\fR | \fBoff\fR | \fBopts\fR
 1208 .ad
 1209 .sp .6
 1210 .RS 4n
 1211 Controls whether the file system is shared by using the Solaris \fBCIFS\fR
 1212 service, and what options are to be used. A file system with the \fBsharesmb\fR
 1213 property set to \fBoff\fR is managed through traditional tools such as
 1214 \fBsharemgr\fR(1M). Otherwise, the file system is automatically shared and
 1215 unshared with the \fBzfs share\fR and \fBzfs unshare\fR commands. If the
 1216 property is set to \fBon\fR, the \fBsharemgr\fR(1M) command is invoked with no
 1217 options. Otherwise, the \fBsharemgr\fR(1M) command is invoked with options
 1218 equivalent to the contents of this property.
 1219 .sp
 1220 Because \fBSMB\fR shares requires a resource name, a unique resource name is
 1221 constructed from the dataset name. The constructed name is a copy of the
 1222 dataset name except that the characters in the dataset name, which would be
 1223 illegal in the resource name, are replaced with underscore (\fB_\fR)
 1224 characters. A pseudo property "name" is also supported that allows you to
 1225 replace the data set name with a specified name. The specified name is then
 1226 used to replace the prefix dataset in the case of inheritance. For example, if
 1227 the dataset \fBdata/home/john\fR is set to \fBname=john\fR, then
 1228 \fBdata/home/john\fR has a resource name of \fBjohn\fR. If a child dataset of
 1229 \fBdata/home/john/backups\fR, it has a resource name of \fBjohn_backups\fR.
 1230 .sp
 1231 When SMB shares are created, the SMB share name appears as an entry in the
 1232 \fB&.zfs/shares\fR directory. You can use the \fBls\fR or \fBchmod\fR command
 1233 to display the share-level ACLs on the entries in this directory.
 1234 .sp
 1235 When the \fBsharesmb\fR property is changed for a dataset, the dataset and any
 1236 children inheriting the property are re-shared with the new options, only if
 1237 the property was previously set to \fBoff\fR, or if they were shared before the
 1238 property was changed. If the new property is set to \fBoff\fR, the file systems
 1239 are unshared.
 1240 .RE

1242 .sp
 1243 .ne 2
 1244 .na
 1245 \fB\fBsharenfs\fR=\fBon\fR | \fBoff\fR | \fBopts\fR
 1246 .ad
 1247 .sp .6
 1248 .RS 4n
 1249 Controls whether the file system is shared via \fBNFS\fR, and what options are
 1250 used. A file system with a \fBsharenfs\fR property of \fBoff\fR is managed
 1251 through traditional tools such as \fBshare\fR(1M), \fBunshare\fR(1M), and
 1252 \fBdfstab\fR(4). Otherwise, the file system is automatically shared and
 1253 unshared with the \fBzfs share\fR and \fBzfs unshare\fR commands. If the
 1254 property is set to \fBon\fR, the \fBshare\fR(1M) command is invoked with no
 1255 options. Otherwise, the \fBshare\fR(1M) command is invoked with options
 1256 equivalent to the contents of this property.
 1257 .sp
 1258 When the \fBsharenfs\fR property is changed for a dataset, the dataset and any
 1259 children inheriting the property are re-shared with the new options, only if
 1260 the property was previously \fBoff\fR, or if they were shared before the
 1261 property was changed. If the new property is \fBoff\fR, the file systems are
 1262 unshared.
 1263 .RE

1265 .sp
 1266 .ne 2
 1267 .na
 1268 \fB\fBblogbias\fR = \fBlatency\fR | \fBthroughput\fR
 1269 .ad
 1270 .sp .6
 1271 .RS 4n
 1272 Provide a hint to ZFS about handling of synchronous requests in this dataset.
 1273 If \fBblogbias\fR is set to \fBlatency\fR (the default), ZFS will use pool log
 1274 devices (if configured) to handle the requests at low latency. If \fBblogbias\fR
 1275 is set to \fBthroughput\fR, ZFS will not use configured pool log devices. ZFS
 1276 will instead optimize synchronous operations for global pool throughput and
 1277 efficient use of resources.
 1278 .RE

1280 .sp
 1281 .ne 2
 1282 .na
 1283 \fB\fBsnapdir\fR=\fBhidden\fR | \fBvisible\fR
 1284 .ad
 1285 .sp .6
 1286 .RS 4n
 1287 Controls whether the \fB&.zfs\fR directory is hidden or visible in the root of
 1288 the file system as discussed in the "Snapshots" section. The default value is
 1289 \fBhidden\fR.
 1290 .RE

1292 .sp
 1293 .ne 2
 1294 .na
 1295 \fB\fBsync\fR=\fBdefault\fR | \fBalways\fR | \fBdisabled\fR
 1296 .ad
 1297 .sp .6
 1298 .RS 4n
 1299 Controls the behavior of synchronous requests (e.g. \fBsync\fR, \fBDSYNC\fR).
 1300 \fBdefault\fR is the POSIX specified behavior of ensuring all synchronous
 1301 requests are written to stable storage and all devices are flushed to ensure
 1302 data is not cached by device controllers (this is the default). \fBalways\fR
 1303 causes every file system transaction to be written and flushed before its
 1304 system call returns. This has a large performance penalty. \fBdisabled\fR
 1305 disables synchronous requests. File system transactions are only committed to
 1306 stable storage periodically. This option will give the highest performance.
 1307 However, it is very dangerous as ZFS would be ignoring the synchronous

1308 transaction demands of applications such as databases or NFS. Administrators
1309 should only use this option when the risks are understood.
1310 .RE

1312 .sp
1313 .ne 2
1314 .na
1315 \fB\fBversion\fR=\fB1\fR | \fB2\fR | \fBcurrent\fR\fR
1316 .ad
1317 .sp .6
1318 .RS 4n

1319 The on-disk version of this file system, which is independent of the pool
1320 version. This property can only be set to later supported versions. See the
1321 \fBzfs upgrade\fR command.
1322 .RE

1324 .sp
1325 .ne 2
1326 .na
1327 \fB\fBvolsize\fR=\fBsize\fR\fR
1328 .ad
1329 .sp .6
1330 .RS 4n

1331 For volumes, specifies the logical size of the volume. By default, creating a
1332 volume establishes a reservation of equal size. For storage pools with a
1333 version number of 9 or higher, a \fBrefreservation\fR is set instead. Any
1334 changes to \fBvolsize\fR are reflected in an equivalent change to the
1335 reservation (or \fBrefreservation\fR). The \fBvolsize\fR can only be set to a
1336 multiple of \fBvolblocksize\fR, and cannot be zero.
1337 .sp
1338 The reservation is kept equal to the volume's logical size to prevent
1339 unexpected behavior for consumers. Without the reservation, the volume could
1340 run out of space, resulting in undefined behavior or data corruption, depending
1341 on how the volume is used. These effects can also occur when the volume size is
1342 changed while it is in use (particularly when shrinking the size). Extreme care
1343 should be used when adjusting the volume size.
1344 .sp
1345 Though not recommended, a "sparse volume" (also known as "thin provisioning")
1346 can be created by specifying the \fB-s\fR option to the \fBzfs create -V\fR
1347 command, or by changing the reservation after the volume has been created. A
1348 "sparse volume" is a volume where the reservation is less than the volume size.
1349 Consequently, writes to a sparse volume can fail with \fBENOSPC\fR when the
1350 pool is low on space. For a sparse volume, changes to \fBvolsize\fR are not
1351 reflected in the reservation.
1352 .RE

1354 .sp
1355 .ne 2
1356 .na
1357 \fB\fBvscan\fR=\fBon\fR | \fBoff\fR\fR
1358 .ad
1359 .sp .6
1360 .RS 4n
1361 Controls whether regular files should be scanned for viruses when a file is
1362 opened and closed. In addition to enabling this property, the virus scan
1363 service must also be enabled for virus scanning to occur. The default value is
1364 \fBoff\fR.
1365 .RE

1367 .sp
1368 .ne 2
1369 .na
1370 \fB\fBxattr\fR=\fBon\fR | \fBoff\fR\fR
1371 .ad
1372 .sp .6
1373 .RS 4n

1374 Controls whether extended attributes are enabled for this file system. The
1375 default value is \fBon\fR.
1376 .RE

1378 .sp
1379 .ne 2
1380 .na
1381 \fB\fBzoned\fR=\fBon\fR | \fBoff\fR\fR
1382 .ad
1383 .sp .6
1384 .RS 4n

1385 Controls whether the dataset is managed from a non-global zone. See the "Zones"
1386 section for more information. The default value is \fBoff\fR.
1387 .RE

1389 .sp
1390 .LP
1391 The following three properties cannot be changed after the file system is
1392 created, and therefore, should be set when the file system is created. If the
1393 properties are not set with the \fBzfs create\fR or \fBzpool create\fR
1394 commands, these properties are inherited from the parent dataset. If the parent
1395 dataset lacks these properties due to having been created prior to these
1396 features being supported, the new file system will have the default values for
1397 these properties.

1398 .sp
1399 .ne 2
1400 .na
1401 \fB\fBcasesensitivity\fR=\fBensitive\fR | \fBinsensitive\fR | \fBmixed\fR\fR
1402 .ad
1403 .sp .6
1404 .RS 4n
1405 Indicates whether the file name matching algorithm used by the file system
1406 should be case-sensitive, case-insensitive, or allow a combination of both
1407 styles of matching. The default value for the \fBcasesensitivity\fR property is
1408 \fBensitive\fR. Traditionally, UNIX and POSIX file systems have case-sensitive
1409 file names.
1410 .sp
1411 The \fBmixed\fR value for the \fBcasesensitivity\fR property indicates that the
1412 file system can support requests for both case-sensitive and case-insensitive
1413 matching behavior. Currently, case-insensitive matching behavior on a file
1414 system that supports mixed behavior is limited to the Solaris CIFS server
1415 product. For more information about the \fBmixed\fR value behavior, see the
1416 \fBISolaris ZFS Administration Guide\fR.
1417 .RE

1419 .sp
1420 .ne 2
1421 .na
1422 \fB\fBnormalization\fR = \fBnone\fR | \fBformC\fR | \fBformD\fR | \fBformKC\fR
1423 | \fBformKD\fR\fR
1424 .ad
1425 .sp .6
1426 .RS 4n
1427 Indicates whether the file system should perform a \fBunicode\fR normalization
1428 of file names whenever two file names are compared, and which normalization
1429 algorithm should be used. File names are always stored unmodified, names are
1430 normalized as part of any comparison process. If this property is set to a
1431 legal value other than \fBnone\fR, and the \fButf8only\fR property was left
1432 unspecified, the \fButf8only\fR property is automatically set to \fBon\fR. The
1433 default value of the \fBnormalization\fR property is \fBnone\fR. This property
1434 cannot be changed after the file system is created.
1435 .RE

1437 .sp
1438 .ne 2
1439 .na

1440 \fB\fButf8only\fR=\fBon\fR | \fBoff\fR
 1441 .ad
 1442 .sp .6
 1443 .RS 4n
 1444 Indicates whether the file system should reject file names that include
 1445 characters that are not present in the \fBUTF-8\fR character code set. If this
 1446 property is explicitly set to \fBoff\fR, the normalization property must either
 1447 not be explicitly set or be set to \fBnone\fR. The default value for the
 1448 \fButf8only\fR property is \fBoff\fR. This property cannot be changed after the
 1449 file system is created.
 1450 .RE

1452 .sp
 1453 .LP
 1454 The \fBcasesensitivity\fR, \fBnormalization\fR, and \fButf8only\fR properties
 1455 are also new permissions that can be assigned to non-privileged users by using
 1456 the \fBZFS\fR delegated administration feature.
 1457 .SS "Temporary Mount Point Properties"
 1458 .sp
 1459 .LP
 1460 When a file system is mounted, either through \fBmount\fR(1M) for legacy mounts
 1461 or the \fBzfs mount\fR command for normal file systems, its mount options are
 1462 set according to its properties. The correlation between properties and mount
 1463 options is as follows:
 1464 .sp
 1465 .in +2
 1466 .nf

PROPERTY	MOUNT OPTION
devices	devices/nodevices
exec	exec/noexec
readonly	ro/rw
setuid	setuid/nosetuid
xattr	xattr/noxattr

1473 .fi
 1474 .in -2
 1475 .sp

1477 .sp
 1478 .LP
 1479 In addition, these options can be set on a per-mount basis using the \fB-o\fR
 1480 option, without affecting the property that is stored on disk. The values
 1481 specified on the command line override the values stored in the dataset. The
 1482 \fB-nosuid\fR option is an alias for \fBnodevices,nosetuid\fR. These properties
 1483 are reported as "temporary" by the \fBzfs get\fR command. If the properties are
 1484 changed while the dataset is mounted, the new setting overrides any temporary
 1485 settings.
 1486 .SS "User Properties"
 1487 .sp
 1488 .LP
 1489 In addition to the standard native properties, \fBZFS\fR supports arbitrary
 1490 user properties. User properties have no effect on \fBZFS\fR behavior, but
 1491 applications or administrators can use them to annotate datasets (file systems,
 1492 volumes, and snapshots).
 1493 .sp
 1494 .LP
 1495 User property names must contain a colon (\fB:\fR) character to distinguish
 1496 them from native properties. They may contain lowercase letters, numbers, and
 1497 the following punctuation characters: colon (\fB:\fR), dash (\fB-\fR), period
 1498 (\fB\&.\fR), and underscore (\fB_\fR). The expected convention is that the
 1499 property name is divided into two portions such as
 1500 \fBmodule\fB:\fB\fR\fBproperty\fR, but this namespace is not enforced by
 1501 \fBZFS\fR. User property names can be at most 256 characters, and cannot begin
 1502 with a dash (\fB-\fR).
 1503 .sp
 1504 .LP
 1505 When making programmatic use of user properties, it is strongly suggested to

1506 use a reversed \fBDNS\fR domain name for the \fBmodule\fR component of property
 1507 names to reduce the chance that two independently-developed packages use the
 1508 same property name for different purposes. Property names beginning with
 1509 \fBcom.sun\fR. are reserved for use by Sun Microsystems.
 1510 .sp
 1511 .LP
 1512 The values of user properties are arbitrary strings, are always inherited, and
 1513 are never validated. All of the commands that operate on properties (\fBzfs
 1514 list\fR, \fBzfs get\fR, \fBzfs set\fR, and so forth) can be used to manipulate
 1515 both native properties and user properties. Use the \fBzfs inherit\fR command
 1516 to clear a user property. If the property is not defined in any parent
 1517 dataset, it is removed entirely. Property values are limited to 1024
 1518 characters.
 1519 .SS "ZFS Volumes as Swap or Dump Devices"
 1520 .sp
 1521 .LP
 1522 During an initial installation a swap device and dump device are created on
 1523 \fBZFS\fR volumes in the \fBZFS\fR root pool. By default, the swap area size is
 1524 based on 1/2 the size of physical memory up to 2 Gbytes. The size of the dump
 1525 device depends on the kernel's requirements at installation time. Separate
 1526 \fBZFS\fR volumes must be used for the swap area and dump devices. Do not swap
 1527 to a file on a \fBZFS\fR file system. A \fBZFS\fR swap file configuration is
 1528 not supported.
 1529 .sp
 1530 .LP
 1531 If you need to change your swap area or dump device after the system is
 1532 installed or upgraded, use the \fBswap\fR(1M) and \fBdumpadm\fR(1M) commands.
 1533 If you need to change the size of your swap area or dump device, see the
 1534 \fBISolaris ZFS Administration Guide\fR.
 1535 .SH SUBCOMMANDS
 1536 .sp
 1537 .LP
 1538 All subcommands that modify state are logged persistently to the pool in their
 1539 original form.
 1540 .sp
 1541 .ne 2
 1542 .na
 1543 \fB\fBzfs ?\fR
 1544 .ad
 1545 .sp .6
 1546 .RS 4n
 1547 Displays a help message.
 1548 .RE

1550 .sp
 1551 .ne 2
 1552 .na
 1553 \fB\fBzfs create\fR [\fB-p\fR] [\fB-o\fR \fBproperty\fR=\fBvalue\fR] ...
 1554 \fBifilesystem\fR
 1555 .ad
 1556 .sp .6
 1557 .RS 4n
 1558 Creates a new \fBZFS\fR file system. The file system is automatically mounted
 1559 according to the \fBmountpoint\fR property inherited from the parent.
 1560 .sp
 1561 .ne 2
 1562 .na
 1563 \fB\fB-p\fR
 1564 .ad
 1565 .sp .6
 1566 .RS 4n
 1567 Creates all the non-existing parent datasets. Datasets created in this manner
 1568 are automatically mounted according to the \fBmountpoint\fR property inherited
 1569 from their parent. Any property specified on the command line using the
 1570 \fB-o\fR option is ignored. If the target filesystem already exists, the
 1571 operation completes successfully.

```

1572 .RE

1574 .sp
1575 .ne 2
1576 .na
1577 \fB\fB-o\fR \fIproperty\fR=\fIvalue\fR\fR
1578 .ad
1579 .sp .6
1580 .RS 4n
1581 Sets the specified property as if the command \fBzfs set\fR
1582 \fIproperty\fR=\fIvalue\fR was invoked at the same time the dataset was
1583 created. Any editable \fBZFS\fR property can also be set at creation time.
1584 Multiple \fB-o\fR options can be specified. An error results if the same
1585 property is specified in multiple \fB-o\fR options.
1586 .RE

1588 .RE

1590 .sp
1591 .ne 2
1592 .na
1593 \fB\fBzfs create\fR [\fB-ps\fR] [\fB-b\fR \fIblocksize\fR] [\fB-o\fR
1594 \fIproperty\fR=\fIvalue\fR] ... \fB-V\fR \fIsize\fR \fIvolume\fR\fR
1595 .ad
1596 .sp .6
1597 .RS 4n
1598 Creates a volume of the given size. The volume is exported as a block device in
1599 \fB/dev/zvol/{dsk,rdisk}/\fR\fIpath\fR, where \fIpath\fR is the name of the
1600 volume in the \fBZFS\fR namespace. The size represents the logical size as
1601 exported by the device. By default, a reservation of equal size is created.
1602 .sp
1603 \fIsize\fR is automatically rounded up to the nearest 128 Kbytes to ensure that
1604 the volume has an integral number of blocks regardless of \fIblocksize\fR.
1605 .sp
1606 .ne 2
1607 .na
1608 \fB\fB-p\fR\fR
1609 .ad
1610 .sp .6
1611 .RS 4n
1612 Creates all the non-existing parent datasets. Datasets created in this manner
1613 are automatically mounted according to the \fBmountpoint\fR property inherited
1614 from their parent. Any property specified on the command line using the
1615 \fB-o\fR option is ignored. If the target filesystem already exists, the
1616 operation completes successfully.
1617 .RE

1619 .sp
1620 .ne 2
1621 .na
1622 \fB\fB-s\fR\fR
1623 .ad
1624 .sp .6
1625 .RS 4n
1626 Creates a sparse volume with no reservation. See \fBvolsize\fR in the Native
1627 Properties section for more information about sparse volumes.
1628 .RE

1630 .sp
1631 .ne 2
1632 .na
1633 \fB\fB-o\fR \fIproperty\fR=\fIvalue\fR\fR
1634 .ad
1635 .sp .6
1636 .RS 4n
1637 Sets the specified property as if the \fBzfs set\fR \fIproperty\fR=\fIvalue\fR

```

```

1638 command was invoked at the same time the dataset was created. Any editable
1639 \fBZFS\fR property can also be set at creation time. Multiple \fB-o\fR options
1640 can be specified. An error results if the same property is specified in
1641 multiple \fB-o\fR options.
1642 .RE

1644 .sp
1645 .ne 2
1646 .na
1647 \fB\fB-b\fR \fIblocksize\fR\fR
1648 .ad
1649 .sp .6
1650 .RS 4n
1651 Equivalent to \fB-o\fR \fBvolblocksize\fR=\fIblocksize\fR. If this option is
1652 specified in conjunction with \fB-o\fR \fBvolblocksize\fR, the resulting
1653 behavior is undefined.
1654 .RE

1656 .RE

1658 .sp
1659 .ne 2
1660 .na
1661 \fBzfs destroy\fR [\fB-fnprrv\fR] \fIfilesystem\fR \fIvolume\fR
1662 .ad
1663 .sp .6
1664 .RS 4n
1665 Destroys the given dataset. By default, the command unshares any file systems
1666 that are currently shared, unmounts any file systems that are currently
1667 mounted, and refuses to destroy a dataset that has active dependents (children
1668 or clones).
1669 .sp
1670 .ne 2
1671 .na
1672 \fB\fB-r\fR\fR
1673 .ad
1674 .sp .6
1675 .RS 4n
1676 Recursively destroy all children.
1677 .RE

1679 .sp
1680 .ne 2
1681 .na
1682 \fB\fB-R\fR\fR
1683 .ad
1684 .sp .6
1685 .RS 4n
1686 Recursively destroy all dependents, including cloned file systems outside the
1687 target hierarchy.
1688 .RE

1690 .sp
1691 .ne 2
1692 .na
1693 \fB\fB-f\fR\fR
1694 .ad
1695 .sp .6
1696 .RS 4n
1697 Force an unmount of any file systems using the \fBunmount -f\fR command. This
1698 option has no effect on non-file systems or unmounted file systems.
1699 .RE

1701 .sp
1702 .ne 2
1703 .na

```

```

1704 \fB\fB-n\fR\fR
1705 .ad
1706 .sp .6
1707 .RS 4n
1708 Do a dry-run ("No-op") deletion. No data will be deleted. This is
1709 useful in conjunction with the \fB-v\fR or \fB-p\fR flags to determine what
1710 data would be deleted.
1711 .RE

1713 .sp
1714 .ne 2
1715 .na
1716 \fB\fB-p\fR\fR
1717 .ad
1718 .sp .6
1719 .RS 4n
1720 Print machine-parsable verbose information about the deleted data.
1721 .RE

1723 .sp
1724 .ne 2
1725 .na
1726 \fB\fB-v\fR\fR
1727 .ad
1728 .sp .6
1729 .RS 4n
1730 Print verbose information about the deleted data.
1731 .RE
1732 .sp
1733 Extreme care should be taken when applying either the \fB-r\fR or the \fB-R\fR
1734 options, as they can destroy large portions of a pool and cause unexpected
1735 behavior for mounted file systems in use.
1736 .RE

1738 .sp
1739 .ne 2
1740 .na
1741 \fBzfs destroy\fR [\fB-dnpRrv\fR] \fIfilesystem\fR|\fIvolume\fR@\fIsnap\fR[%\fIs
1742 .ad
1743 .sp .6
1744 .RS 4n
1745 The given snapshots are destroyed immediately if and only if the \fBzfs
1746 destroy\fR command without the \fB-d\fR option would have destroyed it. Such
1747 immediate destruction would occur, for example, if the snapshot had no clones
1748 and the user-initiated reference count were zero.
1749 .sp
1750 If a snapshot does not qualify for immediate destruction, it is marked for
1751 deferred deletion. In this state, it exists as a usable, visible snapshot until
1752 both of the preconditions listed above are met, at which point it is destroyed.
1753 .sp
1754 An inclusive range of snapshots may be specified by separating the
1755 first and last snapshots with a percent sign.
1756 The first and/or last snapshots may be left blank, in which case the
1757 filesystem's oldest or newest snapshot will be implied.
1758 .sp
1759 Multiple snapshots
1760 (or ranges of snapshots) of the same filesystem or volume may be specified
1761 in a comma-separated list of snapshots.
1762 Only the snapshot's short name (the
1763 part after the \fB@\fR) should be specified when using a range or
1764 comma-separated list to identify multiple snapshots.
1765 .sp
1766 .ne 2
1767 .na
1768 \fB\fB-d\fR\fR
1769 .ad

```

```

1770 .sp .6
1771 .RS 4n
1772 Defer snapshot deletion.
1773 .RE

1775 .sp
1776 .ne 2
1777 .na
1778 \fB\fB-r\fR\fR
1779 .ad
1780 .sp .6
1781 .RS 4n
1782 Destroy (or mark for deferred deletion) all snapshots with this name in
1783 descendent file systems.
1784 .RE

1786 .sp
1787 .ne 2
1788 .na
1789 \fB\fB-R\fR\fR
1790 .ad
1791 .sp .6
1792 .RS 4n
1793 Recursively destroy all dependents.
1794 .RE

1796 .sp
1797 .ne 2
1798 .na
1799 \fB\fB-n\fR\fR
1800 .ad
1801 .sp .6
1802 .RS 4n
1803 Do a dry-run ("No-op") deletion. No data will be deleted. This is
1804 useful in conjunction with the \fB-v\fR or \fB-p\fR flags to determine what
1805 data would be deleted.
1806 .RE

1808 .sp
1809 .ne 2
1810 .na
1811 \fB\fB-p\fR\fR
1812 .ad
1813 .sp .6
1814 .RS 4n
1815 Print machine-parsable verbose information about the deleted data.
1816 .RE

1818 .sp
1819 .ne 2
1820 .na
1821 \fB\fB-v\fR\fR
1822 .ad
1823 .sp .6
1824 .RS 4n
1825 Print verbose information about the deleted data.
1826 .RE

1828 .sp
1829 Extreme care should be taken when applying either the \fB-r\fR or the \fB-f\fR
1830 options, as they can destroy large portions of a pool and cause unexpected
1831 behavior for mounted file systems in use.
1832 .RE

1834 .RE

```

```

1836 .sp
1837 .ne 2
1838 .na
1839 \fB\fzfs snapshot\fR [\fB-r\fR] [\fB-o\fR \fIproperty\fR=\fIvalue\fR] ...
1840 \fIfilesystem@snapname\fR|\fIvolume@snapname\fR\fR...
1840 \fIfilesystem@snapname\fR|\fIvolume@snapname\fR\fR
1841 .ad
1842 .sp .6
1843 .RS 4n
1844 Creates snapshots with the given names. All previous modifications by
1845 successful system calls to the file system are part of the snapshots.
1846 Snapshots are taken atomically, so that all snapshots correspond to the same
1847 moment in time. See the "Snapshots" section for details.
1844 Creates a snapshot with the given name. All previous modifications by
1845 successful system calls to the file system are part of the snapshot. See the
1846 "Snapshots" section for details.
1848 .sp
1849 .ne 2
1850 .na
1851 \fB\fzfs r\fR
1852 .ad
1853 .sp .6
1854 .RS 4n
1855 Recursively create snapshots of all descendent datasets
1854 Recursively create snapshots of all descendent datasets. Snapshots are taken
1855 atomically, so that all recursive snapshots correspond to the same moment in
1856 time.
1856 .RE

1858 .sp
1859 .ne 2
1860 .na
1861 \fB\fzfs o\fR \fIproperty\fR=\fIvalue\fR\fR
1862 .ad
1863 .sp .6
1864 .RS 4n
1865 Sets the specified property; see \fzfs create\fR for details.
1866 .RE

1868 .RE

1870 .sp
1871 .ne 2
1872 .na
1873 \fB\fzfs rollback\fR [\fB-rf\fR] \fIsnapshot\fR\fR
1874 .ad
1875 .sp .6
1876 .RS 4n
1877 Roll back the given dataset to a previous snapshot. When a dataset is rolled
1878 back, all data that has changed since the snapshot is discarded, and the
1879 dataset reverts to the state at the time of the snapshot. By default, the
1880 command refuses to roll back to a snapshot other than the most recent one. In
1881 order to do so, all intermediate snapshots must be destroyed by specifying the
1882 \fB-r\fR option.
1883 .sp
1884 The \fB-r\fR options do not recursively destroy the child snapshots of a
1885 recursive snapshot. Only the top-level recursive snapshot is destroyed by
1886 either of these options. To completely roll back a recursive snapshot, you must
1887 rollback the individual child snapshots.
1888 .sp
1889 .ne 2
1890 .na
1891 \fB\fzfs r\fR
1892 .ad
1893 .sp .6
1894 .RS 4n

```

```

1895 Recursively destroy any snapshots more recent than the one specified.
1896 .RE

1898 .sp
1899 .ne 2
1900 .na
1901 \fB\fzfs R\fR
1902 .ad
1903 .sp .6
1904 .RS 4n
1905 Recursively destroy any more recent snapshots, as well as any clones of those
1906 snapshots.
1907 .RE

1909 .sp
1910 .ne 2
1911 .na
1912 \fB\fzfs f\fR
1913 .ad
1914 .sp .6
1915 .RS 4n
1916 Used with the \fB-R\fR option to force an unmount of any clone file systems
1917 that are to be destroyed.
1918 .RE

1920 .RE

1922 .sp
1923 .ne 2
1924 .na
1925 \fB\fzfs clone\fR [\fB-p\fR] [\fB-o\fR \fIproperty\fR=\fIvalue\fR] ...
1926 \fIsnapshot\fR \fIfilesystem\fR|\fIvolume\fR\fR
1927 .ad
1928 .sp .6
1929 .RS 4n
1930 Creates a clone of the given snapshot. See the "Clones" section for details.
1931 The target dataset can be located anywhere in the \fBZFS\fR hierarchy, and is
1932 created as the same type as the original.
1933 .sp
1934 .ne 2
1935 .na
1936 \fB\fzfs p\fR
1937 .ad
1938 .sp .6
1939 .RS 4n
1940 Creates all the non-existing parent datasets. Datasets created in this manner
1941 are automatically mounted according to the \fBmountpoint\fR property inherited
1942 from their parent. If the target filesystem or volume already exists, the
1943 operation completes successfully.
1944 .RE

1946 .sp
1947 .ne 2
1948 .na
1949 \fB\fzfs o\fR \fIproperty\fR=\fIvalue\fR\fR
1950 .ad
1951 .sp .6
1952 .RS 4n
1953 Sets the specified property; see \fzfs create\fR for details.
1954 .RE

1956 .RE

1958 .sp
1959 .ne 2
1960 .na

```

```

1961 \fBfbzfs promote\fR \fIclone-filesystem\fR\fR
1962 .ad
1963 .sp .6
1964 .RS 4n
1965 Promotes a clone file system to no longer be dependent on its "origin"
1966 snapshot. This makes it possible to destroy the file system that the clone was
1967 created from. The clone parent-child dependency relationship is reversed, so
1968 that the origin file system becomes a clone of the specified file system.
1969 .sp
1970 The snapshot that was cloned, and any snapshots previous to this snapshot, are
1971 now owned by the promoted clone. The space they use moves from the origin file
1972 system to the promoted clone, so enough space must be available to accommodate
1973 these snapshots. No new space is consumed by this operation, but the space
1974 accounting is adjusted. The promoted clone must not have any conflicting
1975 snapshot names of its own. The \fBrename\fR subcommand can be used to rename
1976 any conflicting snapshots.
1977 .RE

1979 .sp
1980 .ne 2
1981 .na
1982 \fBfbzfs rename\fR [\fB-f\fR] \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR\fR
1983 .ad
1984 .br
1985 .na
1986 \fBfbzfs rename\fR [\fB-fp\fR] \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR\fR
1987 .ad
1988 .br
1989 .na
1990 \fBfbzfs rename\fR [\fB-fp\fR] \fIfilesystem\fR|\fIvolume\fR
1991 \fIfilesystem\fR|\fIvolume\fR
1992 .ad
1993 .sp .6
1994 .RS 4n
1995 Renames the given dataset. The new target can be located anywhere in the
1996 \fBZFS\fR hierarchy, with the exception of snapshots. Snapshots can only be
1997 renamed within the parent file system or volume. When renaming a snapshot, the
1998 parent file system of the snapshot does not need to be specified as part of the
1999 second argument. Renamed file systems can inherit new mount points, in which
2000 case they are unmounted and remounted at the new mount point.
2001 .sp
2002 .ne 2
2003 .na
2004 \fBfbzfs promote\fR \fIclone-filesystem\fR\fR
2005 .ad
2006 .sp .6
2007 .RS 4n
2008 Creates all the nonexistent parent datasets. Datasets created in this manner
2009 are automatically mounted according to the \fBmountpoint\fR property inherited
2010 from their parent.
2011 .RE

2013 .sp
2014 .ne 2
2015 .na
2016 \fBfbzfs promote\fR \fIclone-filesystem\fR\fR
2017 .ad
2018 .sp .6
2019 .RS 4n
2020 Force unmount any filesystems that need to be unmounted in the process.
2021 .RE

2023 .RE

2025 .sp
2026 .ne 2

```

```

2027 .na
2028 \fBfbzfs rename\fR \fB-r\fR \fIsnapshot\fR \fIsnapshot\fR\fR
2029 .ad
2030 .sp .6
2031 .RS 4n
2032 Recursively rename the snapshots of all descendent datasets. Snapshots are the
2033 only dataset that can be renamed recursively.
2034 .RE

2036 .sp
2037 .ne 2
2038 .na
2039 \fBfbzfs list\fR [\fB-r\fR|\fB-d\fR \fIdepth\fR] [\fB-H\fR] [\fB-o\fR
2040 \fIproperty\fR[, \fI&...\fR]] [\fB-t\fR \fItype\fR[, \fI&...\fR]] [\fB-s\fR
2041 \fIproperty\fR ] ... [\fB-S\fR \fIproperty\fR ] ...
2042 [\fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR] ... \fR
2043 .ad
2044 .sp .6
2045 .RS 4n
2046 Lists the property information for the given datasets in tabular form. If
2047 specified, you can list property information by the absolute pathname or the
2048 relative pathname. By default, all file systems and volumes are displayed.
2049 Snapshots are displayed if the \fBlistsnaps\fR property is \fBon\fR (the
2050 default is \fBoff\fR). The following fields are displayed,
2051 \fBname,used,available,referenced,mountpoint\fR.
2052 .sp
2053 .ne 2
2054 .na
2055 \fBfbzfs list\fR [\fB-r\fR|\fB-d\fR \fIdepth\fR] [\fB-H\fR] [\fB-o\fR
2056 \fIproperty\fR[, \fI&...\fR]] [\fB-t\fR \fItype\fR[, \fI&...\fR]] [\fB-s\fR
2057 \fIproperty\fR ] ... [\fB-S\fR \fIproperty\fR ] ...
2058 .RS 4n
2059 Used for scripting mode. Do not print headers and separate fields by a single
2060 tab instead of arbitrary white space.
2061 .RE

2063 .sp
2064 .ne 2
2065 .na
2066 \fBfbzfs list\fR [\fB-r\fR|\fB-d\fR \fIdepth\fR] [\fB-H\fR] [\fB-o\fR
2067 \fIproperty\fR[, \fI&...\fR]] [\fB-t\fR \fItype\fR[, \fI&...\fR]] [\fB-s\fR
2068 \fIproperty\fR ] ... [\fB-S\fR \fIproperty\fR ] ...
2069 .RS 4n
2070 Recursively display any children of the dataset on the command line.
2071 .RE

2073 .sp
2074 .ne 2
2075 .na
2076 \fBfbzfs list\fR [\fB-r\fR|\fB-d\fR \fIdepth\fR] [\fB-H\fR] [\fB-o\fR
2077 \fIproperty\fR[, \fI&...\fR]] [\fB-t\fR \fItype\fR[, \fI&...\fR]] [\fB-s\fR
2078 \fIproperty\fR ] ... [\fB-S\fR \fIproperty\fR ] ...
2079 .RS 4n
2080 Recursively display any children of the dataset, limiting the recursion to
2081 \fBdepth\fR. A depth of \fB1\fR will display only the dataset and its direct
2082 children.
2083 .RE

2085 .sp
2086 .ne 2
2087 .na
2088 \fBfbzfs list\fR [\fB-r\fR|\fB-d\fR \fIdepth\fR] [\fB-H\fR] [\fB-o\fR
2089 \fIproperty\fR[, \fI&...\fR]] [\fB-t\fR \fItype\fR[, \fI&...\fR]] [\fB-s\fR
2090 \fIproperty\fR ] ... [\fB-S\fR \fIproperty\fR ] ...
2091 .RS 4n
2092 A comma-separated list of properties to display. The property must be:

```



```

2093 .RS +4
2094 .TP
2095 .ie t \(\bu
2096 .el o
2097 One of the properties described in the "Native Properties" section
2098 .RE
2099 .RS +4
2100 .TP
2101 .ie t \(\bu
2102 .el o
2103 A user property
2104 .RE
2105 .RS +4
2106 .TP
2107 .ie t \(\bu
2108 .el o
2109 The value \fBname\fR to display the dataset name
2110 .RE
2111 .RS +4
2112 .TP
2113 .ie t \(\bu
2114 .el o
2115 The value \fBspace\fR to display space usage properties on file systems and
2116 volumes. This is a shortcut for specifying \fB-o
2117 name,avail,used,usedsnap,usedds,usedrefreserv,usedchild\fR \fB-t
2118 filesystem,volume\fR syntax.
2119 .RE
2120 .RE

2122 .sp
2123 .ne 2
2124 .na
2125 \fB\fB-s\fR \fIproperty\fR\fR
2126 .ad
2127 .sp .6
2128 .RS 4n
2129 A property for sorting the output by column in ascending order based on the
2130 value of the property. The property must be one of the properties described in
2131 the "Properties" section, or the special value \fBname\fR to sort by the
2132 dataset name. Multiple properties can be specified at one time using multiple
2133 \fB-s\fR property options. Multiple \fB-s\fR options are evaluated from left to
2134 right in decreasing order of importance.
2135 .sp
2136 The following is a list of sorting criteria:
2137 .RS +4
2138 .TP
2139 .ie t \(\bu
2140 .el o
2141 Numeric types sort in numeric order.
2142 .RE
2143 .RS +4
2144 .TP
2145 .ie t \(\bu
2146 .el o
2147 String types sort in alphabetical order.
2148 .RE
2149 .RS +4
2150 .TP
2151 .ie t \(\bu
2152 .el o
2153 Types inappropriate for a row sort that row to the literal bottom, regardless
2154 of the specified ordering.
2155 .RE
2156 .RS +4
2157 .TP
2158 .ie t \(\bu

```

```

2159 .el o
2160 If no sorting options are specified the existing behavior of \fBzfs list\fR is
2161 preserved.
2162 .RE
2163 .RE

2165 .sp
2166 .ne 2
2167 .na
2168 \fB\fB-S\fR \fIproperty\fR\fR
2169 .ad
2170 .sp .6
2171 .RS 4n
2172 Same as the \fB-s\fR option, but sorts by property in descending order.
2173 .RE

2175 .sp
2176 .ne 2
2177 .na
2178 \fB\fB-t\fR \fItype\fR\fR
2179 .ad
2180 .sp .6
2181 .RS 4n
2182 A comma-separated list of types to display, where \fItype\fR is one of
2183 \fBfilesystem\fR, \fBsnapshot\fR, \fBvolume\fR, or \fBall\fR. For example,
2184 specifying \fB-t snapshot\fR displays only snapshots.
2185 .RE

2187 .RE

2189 .sp
2190 .ne 2
2191 .na
2192 \fB\fBzfs set\fR \fIproperty\fR=\fIvalue\fR
2193 \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR ... \fR
2194 .ad
2195 .sp .6
2196 .RS 4n
2197 Sets the property to the given value for each dataset. Only some properties can
2198 be edited. See the "Properties" section for more information on what properties
2199 can be set and acceptable values. Numeric values can be specified as exact
2200 values, or in a human-readable form with a suffix of \fBB\fR, \fBK\fR, \fBM\fR,
2201 \fBG\fR, \fBT\fR, \fBP\fR, \fBE\fR, \fBZ\fR (for bytes, kilobytes, megabytes,
2202 gigabytes, terabytes, petabytes, exabytes, or zettabytes, respectively). User
2203 properties can be set on snapshots. For more information, see the "User
2204 Properties" section.
2205 .RE

2207 .sp
2208 .ne 2
2209 .na
2210 \fB\fBzfs get\fR [\fB-r\fR|\fB-d\fR \fIdepth\fR] [\fB-Hp\fR] [\fB-o\fR
2211 \fIfield\fR[,...] [\fB-t\fR \fItype\fR[,...]] [\fB-s\fR \fIsource\fR[,...]] "\fIa
2212 \fIproperty\fR[,...] \fIfilesystem\fR|\fIvolume\fR|\fIsnapshot\fR ... \fR
2213 .ad
2214 .sp .6
2215 .RS 4n
2216 Displays properties for the given datasets. If no datasets are specified, then
2217 the command displays properties for all datasets on the system. For each
2218 property, the following columns are displayed:
2219 .sp
2220 .in +2
2221 .nf
2222     name      Dataset name
2223     property  Property name
2224     value     Property value

```

2225 source Property source. Can either be local, default,
 2226 temporary, inherited, or none (-).
 2227 .fi
 2228 .in -2
 2229 .sp

2231 All columns are displayed by default, though this can be controlled by using
 2232 the `\fb-o\fr` option. This command takes a comma-separated list of properties as
 2233 described in the "Native Properties" and "User Properties" sections.
 2234 .sp
 2235 The special value `\fball\fr` can be used to display all properties that apply to
 2236 the given dataset's type (filesystem, volume, or snapshot).
 2237 .sp
 2238 .ne 2
 2239 .na
 2240 `\fb\fb-r\fr\fr`
 2241 .ad
 2242 .sp .6
 2243 .RS 4n
 2244 Recursively display properties for any children.
 2245 .RE

2247 .sp
 2248 .ne 2
 2249 .na
 2250 `\fb\fb-d\fr \fidepth\fr\fr`
 2251 .ad
 2252 .sp .6
 2253 .RS 4n
 2254 Recursively display any children of the dataset, limiting the recursion to
 2255 `\fidepth\fr`. A depth of `\fbl\fr` will display only the dataset and its direct
 2256 children.
 2257 .RE

2259 .sp
 2260 .ne 2
 2261 .na
 2262 `\fb\fb-H\fr\fr`
 2263 .ad
 2264 .sp .6
 2265 .RS 4n
 2266 Display output in a form more easily parsed by scripts. Any headers are
 2267 omitted, and fields are explicitly separated by a single tab instead of an
 2268 arbitrary amount of space.
 2269 .RE

2271 .sp
 2272 .ne 2
 2273 .na
 2274 `\fb\fb-o\fr \fi\field\fr\fr`
 2275 .ad
 2276 .sp .6
 2277 .RS 4n
 2278 A comma-separated list of columns to display. `\fbname,property,value,source\fr`
 2279 is the default value.
 2280 .RE

2282 .sp
 2283 .ne 2
 2284 .na
 2285 `\fb\fb-s\fr \fisource\fr\fr`
 2286 .ad
 2287 .sp .6
 2288 .RS 4n
 2289 A comma-separated list of sources to display. Those properties coming from a
 2290 source other than those in this list are ignored. Each source must be one of

2291 the following: `\fblocal,default,inherited,temporary,none\fr`. The default value
 2292 is all sources.
 2293 .RE

2295 .sp
 2296 .ne 2
 2297 .na
 2298 `\fb\fb-p\fr\fr`
 2299 .ad
 2300 .sp .6
 2301 .RS 4n
 2302 Display numbers in parseable (exact) values.
 2303 .RE

2305 .RE

2307 .sp
 2308 .ne 2
 2309 .na
 2310 `\fb\fbzfs inherit\fr [\fb-r\fr] \fi\property\fr`
 2311 `\fi\filesystem\fr|\fi\volume\fr|\fi\snapshot\fr ... \fr`
 2312 .ad
 2313 .sp .6
 2314 .RS 4n
 2315 Clears the specified property, causing it to be inherited from an ancestor. If
 2316 no ancestor has the property set, then the default value is used. See the
 2317 "Properties" section for a listing of default values, and details on which
 2318 properties can be inherited.
 2319 .sp
 2320 .ne 2
 2321 .na
 2322 `\fb\fb-r\fr\fr`
 2323 .ad
 2324 .sp .6
 2325 .RS 4n
 2326 Recursively inherit the given property for all children.
 2327 .RE

2329 .RE

2331 .sp
 2332 .ne 2
 2333 .na
 2334 `\fb\fbzfs upgrade\fr [\fb-v\fr]\fr`
 2335 .ad
 2336 .sp .6
 2337 .RS 4n
 2338 Displays a list of file systems that are not the most recent version.
 2339 .RE

2341 .sp
 2342 .ne 2
 2343 .na
 2344 `\fb\fbzfs upgrade\fr [\fb-r\fr] [\fb-V\fr \fi\version\fr] [\fb-a\fr |`
 2345 `\fi\filesystem\fr]\fr`
 2346 .ad
 2347 .sp .6
 2348 .RS 4n
 2349 Upgrades file systems to a new on-disk version. Once this is done, the file
 2350 systems will no longer be accessible on systems running older versions of the
 2351 software. `\fbzfs send\fr` streams generated from new snapshots of these file
 2352 systems cannot be accessed on systems running older versions of the software.
 2353 .sp
 2354 In general, the file system version is independent of the pool version. See
 2355 `\fbzpool\fr(1M)` for information on the `\fbzpool upgrade\fr` command.
 2356 .sp

```

2357 In some cases, the file system version and the pool version are interrelated
2358 and the pool version must be upgraded before the file system version can be
2359 upgraded.
2360 .sp
2361 .ne 2
2362 .na
2363 \fB\fB-a\fR\fR
2364 .ad
2365 .sp .6
2366 .RS 4n
2367 Upgrade all file systems on all imported pools.
2368 .RE

2370 .sp
2371 .ne 2
2372 .na
2373 \fB\fIfilesystem\fR\fR
2374 .ad
2375 .sp .6
2376 .RS 4n
2377 Upgrade the specified file system.
2378 .RE

2380 .sp
2381 .ne 2
2382 .na
2383 \fB\fB-r\fR\fR
2384 .ad
2385 .sp .6
2386 .RS 4n
2387 Upgrade the specified file system and all descendent file systems
2388 .RE

2390 .sp
2391 .ne 2
2392 .na
2393 \fB\fB-V\fR \fIversion\fR\fR
2394 .ad
2395 .sp .6
2396 .RS 4n
2397 Upgrade to the specified \fIversion\fR. If the \fB-V\fR flag is not specified,
2398 this command upgrades to the most recent version. This option can only be used
2399 to increase the version number, and only up to the most recent version
2400 supported by this software.
2401 .RE

2403 .RE

2405 .sp
2406 .ne 2
2407 .na
2408 \fB\fBzfs userspace\fR [\fB-niHp\fR] [\fB-o\fR \fIifield\fR[,...]] [\fB-sS\fR
2409 \fIifield\fR]... [\fB-t\fR \fIitype\fR [,...]] \fIfilesystem\fR |
2410 \fIsnapshot\fR\fR
2411 .ad
2412 .sp .6
2413 .RS 4n
2414 Displays space consumed by, and quotas on, each user in the specified
2415 filesystem or snapshot. This corresponds to the \fBUserused@\fR\fIuser\fR and
2416 \fBUserquota@\fR\fIuser\fR properties.
2417 .sp
2418 .ne 2
2419 .na
2420 \fB\fB-n\fR\fR
2421 .ad
2422 .sp .6

```

```

2423 .RS 4n
2424 Print numeric ID instead of user/group name.
2425 .RE

2427 .sp
2428 .ne 2
2429 .na
2430 \fB\fB-H\fR\fR
2431 .ad
2432 .sp .6
2433 .RS 4n
2434 Do not print headers, use tab-delimited output.
2435 .RE

2437 .sp
2438 .ne 2
2439 .na
2440 \fB\fB-p\fR\fR
2441 .ad
2442 .sp .6
2443 .RS 4n
2444 Use exact (parseable) numeric output.
2445 .RE

2447 .sp
2448 .ne 2
2449 .na
2450 \fB\fB-o\fR \fIifield\fR[,...]\fR
2451 .ad
2452 .sp .6
2453 .RS 4n
2454 Display only the specified fields from the following set,
2455 \fBtype,name,used,quota\fR.The default is to display all fields.
2456 .RE

2458 .sp
2459 .ne 2
2460 .na
2461 \fB\fB-s\fR \fIifield\fR\fR
2462 .ad
2463 .sp .6
2464 .RS 4n
2465 Sort output by this field. The \fBIs\fR and \fBIS\fR flags may be specified
2466 multiple times to sort first by one field, then by another. The default is
2467 \fB-s type\fR \fB-s name\fR.
2468 .RE

2470 .sp
2471 .ne 2
2472 .na
2473 \fB\fB-S\fR \fIifield\fR\fR
2474 .ad
2475 .sp .6
2476 .RS 4n
2477 Sort by this field in reverse order. See \fB-s\fR.
2478 .RE

2480 .sp
2481 .ne 2
2482 .na
2483 \fB\fB-t\fR \fIitype\fR[,...]\fR
2484 .ad
2485 .sp .6
2486 .RS 4n
2487 Print only the specified types from the following set,
2488 \fBBall,posixuser,smbuser,posixgroup,smbgroup\fR.

```

```

2489 .sp
2490 The default is \fB-t posixuser, smbuser\fR
2491 .sp
2492 The default can be changed to include group types.
2493 .RE

2495 .sp
2496 .ne 2
2497 .na
2498 \fB\fb-i\fR\fR
2499 .ad
2500 .sp .6
2501 .RS 4n
2502 Translate SID to POSIX ID. The POSIX ID may be ephemeral if no mapping exists.
2503 Normal POSIX interfaces (for example, \fBstat\fR(2), \fBls\fR \fB-l\fR) perform
2504 this translation, so the \fB-i\fR option allows the output from \fBzfs
2505 userspace\fR to be compared directly with those utilities. However, \fB-i\fR
2506 may lead to confusion if some files were created by an SMB user before a
2507 SMB-to-POSIX name mapping was established. In such a case, some files are owned
2508 by the SMB entity and some by the POSIX entity. However, the \fB-i\fR option
2509 will report that the POSIX entity has the total usage and quota for both.
2510 .RE

2512 .RE

2514 .sp
2515 .ne 2
2516 .na
2517 \fB\fbzfs groupspace\fR [\fB-niHp\fR] [\fB-o\fR \fIfield\fR[,...]] [\fB-sS\fR
2518 \fIfield\fR]... [\fB-t\fR \fIttype\fR [,...]] \fIfilesystem\fR |
2519 \fIsnapshot\fR\fR
2520 .ad
2521 .sp .6
2522 .RS 4n
2523 Displays space consumed by, and quotas on, each group in the specified
2524 filesystem or snapshot. This subcommand is identical to \fBzfs userspace\fR,
2525 except that the default types to display are \fB-t posixgroup, smbgroup\fR.
2526 .sp
2527 .in +2
2528 .nf
2529 -
2530 .fi
2531 .in -2
2532 .sp

2534 .RE

2536 .sp
2537 .ne 2
2538 .na
2539 \fB\fbzfs mount\fR\fR
2540 .ad
2541 .sp .6
2542 .RS 4n
2543 Displays all \fBZFS\fR file systems currently mounted.
2544 .RE

2546 .sp
2547 .ne 2
2548 .na
2549 \fB\fbzfs mount\fR [\fB-vO\fR] [\fB-o\fR \fIooptions\fR] \fB-a\fR |
2550 \fIfilesystem\fR\fR
2551 .ad
2552 .sp .6
2553 .RS 4n
2554 Mounts \fBZFS\fR file systems. Invoked automatically as part of the boot

```

```

2555 process.
2556 .sp
2557 .ne 2
2558 .na
2559 \fB\fb-o\fR \fIooptions\fR\fR
2560 .ad
2561 .sp .6
2562 .RS 4n
2563 An optional, comma-separated list of mount options to use temporarily for the
2564 duration of the mount. See the "Temporary Mount Point Properties" section for
2565 details.
2566 .RE

2568 .sp
2569 .ne 2
2570 .na
2571 \fB\fb-O\fR\fR
2572 .ad
2573 .sp .6
2574 .RS 4n
2575 Perform an overlay mount. See \fBmount\fR(1M) for more information.
2576 .RE

2578 .sp
2579 .ne 2
2580 .na
2581 \fB\fb-v\fR\fR
2582 .ad
2583 .sp .6
2584 .RS 4n
2585 Report mount progress.
2586 .RE

2588 .sp
2589 .ne 2
2590 .na
2591 \fB\fb-a\fR\fR
2592 .ad
2593 .sp .6
2594 .RS 4n
2595 Mount all available \fBZFS\fR file systems. Invoked automatically as part of
2596 the boot process.
2597 .RE

2599 .sp
2600 .ne 2
2601 .na
2602 \fB\fbIfilesystem\fR\fR
2603 .ad
2604 .sp .6
2605 .RS 4n
2606 Mount the specified filesystem.
2607 .RE

2609 .RE

2611 .sp
2612 .ne 2
2613 .na
2614 \fB\fbzfs unmount\fR [\fB-f\fR] \fB-a\fR | \fIfilesystem\fR|\fImountpoint\fR\fR
2615 .ad
2616 .sp .6
2617 .RS 4n
2618 Unmounts currently mounted \fBZFS\fR file systems. Invoked automatically as
2619 part of the shutdown process.
2620 .sp

```

```

2621 .ne 2
2622 .na
2623 \fB\fB-f\fR\fR
2624 .ad
2625 .sp .6
2626 .RS 4n
2627 Forcefully unmount the file system, even if it is currently in use.
2628 .RE

2630 .sp
2631 .ne 2
2632 .na
2633 \fB\fB-a\fR\fR
2634 .ad
2635 .sp .6
2636 .RS 4n
2637 Unmount all available \fBZFS\fR file systems. Invoked automatically as part of
2638 the boot process.
2639 .RE

2641 .sp
2642 .ne 2
2643 .na
2644 \fB\fIfilesystem\fR|\fImountpoint\fR\fR
2645 .ad
2646 .sp .6
2647 .RS 4n
2648 Unmount the specified filesystem. The command can also be given a path to a
2649 \fBZFS\fR file system mount point on the system.
2650 .RE

2652 .RE

2654 .sp
2655 .ne 2
2656 .na
2657 \fB\fBzfs share\fR \fB-a\fR | \fIfilesystem\fR\fR
2658 .ad
2659 .sp .6
2660 .RS 4n
2661 Shares available \fBZFS\fR file systems.
2662 .sp
2663 .ne 2
2664 .na
2665 \fB\fB-a\fR\fR
2666 .ad
2667 .sp .6
2668 .RS 4n
2669 Share all available \fBZFS\fR file systems. Invoked automatically as part of
2670 the boot process.
2671 .RE

2673 .sp
2674 .ne 2
2675 .na
2676 \fB\fIfilesystem\fR\fR
2677 .ad
2678 .sp .6
2679 .RS 4n
2680 Share the specified filesystem according to the \fBsharefs\fR and
2681 \fBsharesmb\fR properties. File systems are shared when the \fBsharefs\fR or
2682 \fBsharesmb\fR property is set.
2683 .RE

2685 .RE

```

```

2687 .sp
2688 .ne 2
2689 .na
2690 \fB\fBzfs unshare\fR \fB-a\fR | \fIfilesystem\fR|\fImountpoint\fR\fR
2691 .ad
2692 .sp .6
2693 .RS 4n
2694 Unshares currently shared \fBZFS\fR file systems. This is invoked automatically
2695 as part of the shutdown process.
2696 .sp
2697 .ne 2
2698 .na
2699 \fB\fB-a\fR\fR
2700 .ad
2701 .sp .6
2702 .RS 4n
2703 Unshare all available \fBZFS\fR file systems. Invoked automatically as part of
2704 the boot process.
2705 .RE

2707 .sp
2708 .ne 2
2709 .na
2710 \fB\fIfilesystem\fR|\fImountpoint\fR\fR
2711 .ad
2712 .sp .6
2713 .RS 4n
2714 Unshare the specified filesystem. The command can also be given a path to a
2715 \fBZFS\fR file system shared on the system.
2716 .RE

2718 .RE

2720 .sp
2721 .ne 2
2722 .na
2723 \fBzfs send\fR [\fB-DnPrv\fR] [\fB-\fR[\fBiI\fR] \fIsnapshot\fR] \fIsnapshot\fR
2724 .ad
2725 .sp .6
2726 .RS 4n
2727 Creates a stream representation of the second \fIsnapshot\fR, which is written
2728 to standard output. The output can be redirected to a file or to a different
2729 system (for example, using \fBssh\fR(1)). By default, a full stream is
2730 generated.
2731 .sp
2732 .ne 2
2733 .na
2734 \fB\fB-i\fR \fIsnapshot\fR\fR
2735 .ad
2736 .sp .6
2737 .RS 4n
2738 Generate an incremental stream from the first \fIsnapshot\fR to the second
2739 \fIsnapshot\fR. The incremental source (the first \fIsnapshot\fR) can be
2740 specified as the last component of the snapshot name (for example, the part
2741 after the \fB@\fR), and it is assumed to be from the same file system as the
2742 second \fIsnapshot\fR.
2743 .sp
2744 If the destination is a clone, the source may be the origin snapshot, which
2745 must be fully specified (for example, \fBpool/fs@origin\fR, not just
2746 \fB@origin\fR).
2747 .RE

2749 .sp
2750 .ne 2
2751 .na
2752 \fB\fB-I\fR \fIsnapshot\fR\fR

```

```

2753 .ad
2754 .sp .6
2755 .RS 4n
2756 Generate a stream package that sends all intermediary snapshots from the first
2757 snapshot to the second snapshot. For example, \fB-I @a fs@d\fR is similar to
2758 \fB-i @a fs@b; -i @b fs@c; -i @c fs@d\fR. The incremental source snapshot may
2759 be specified as with the \fB-i\fR option.
2760 .RE

2762 .sp
2763 .ne 2
2764 .na
2765 \fB\fB-R\fR\fR
2766 .ad
2767 .sp .6
2768 .RS 4n
2769 Generate a replication stream package, which will replicate the specified
2770 filesystem, and all descendent file systems, up to the named snapshot. When
2771 received, all properties, snapshots, descendent file systems, and clones are
2772 preserved.
2773 .sp
2774 If the \fB-i\fR or \fB-I\fR flags are used in conjunction with the \fB-R\fR
2775 flag, an incremental replication stream is generated. The current values of
2776 properties, and current snapshot and file system names are set when the stream
2777 is received. If the \fB-F\fR flag is specified when this stream is received,
2778 snapshots and file systems that do not exist on the sending side are destroyed.
2779 .RE

2781 .sp
2782 .ne 2
2783 .na
2784 \fB\fB-D\fR\fR
2785 .ad
2786 .sp .6
2787 .RS 4n
2788 Generate a deduplicated stream. Blocks which would have been sent multiple
2789 times in the send stream will only be sent once. The receiving system must
2790 also support this feature to receive a deduplicated stream. This flag can
2791 be used regardless of the dataset's \fBduplicated\fR property, but performance
2792 will be much better if the filesystem uses a dedup-capable checksum (eg.
2793 \fBsha256\fR).
2794 .RE

2796 .sp
2797 .ne 2
2798 .na
2799 \fB\fB-r\fR\fR
2800 .ad
2801 .sp .6
2802 .RS 4n
2803 Recursively send all descendant snapshots. This is similar to the \fB-R\fR
2804 flag, but information about deleted and renamed datasets is not included, and
2805 property information is only included if the \fB-p\fR flag is specified.
2806 .RE

2808 .sp
2809 .ne 2
2810 .na
2811 \fB\fB-p\fR\fR
2812 .ad
2813 .sp .6
2814 .RS 4n
2815 Include the dataset's properties in the stream. This flag is implicit when
2816 \fB-R\fR is specified. The receiving system must also support this feature.
2817 .RE

```

```

2819 .sp
2820 .ne 2
2821 .na
2822 \fB\fB-n\fR\fR
2823 .ad
2824 .sp .6
2825 .RS 4n
2826 Do a dry-run ("No-op") send. Do not generate any actual send data. This is
2827 useful in conjunction with the \fB-v\fR or \fB-P\fR flags to determine what
2828 data will be sent.
2829 .RE

2831 .sp
2832 .ne 2
2833 .na
2834 \fB\fB-P\fR\fR
2835 .ad
2836 .sp .6
2837 .RS 4n
2838 Print machine-parsable verbose information about the stream package generated.
2839 .RE

2841 .sp
2842 .ne 2
2843 .na
2844 \fB\fB-v\fR\fR
2845 .ad
2846 .sp .6
2847 .RS 4n
2848 Print verbose information about the stream package generated. This information
2849 includes a per-second report of how much data has been sent.
2850 .RE

2852 The format of the stream is committed. You will be able to receive your streams
2853 on future versions of \fBZFS\fR.
2854 .RE

2856 .sp
2857 .ne 2
2858 .na
2859 \fB\fBzfs receive\fR [\fB-vnFu\fR]
2860 \fB\fBfilesystem\fR|\fBivol\fR|\fBisnapshot\fR\fR
2861 .ad
2862 .br
2863 .na
2864 \fB\fBzfs receive\fR [\fB-vnFu\fR] [\fB-d\fR|\fB-e\fR] \fBfilesystem\fR\fR
2865 .ad
2866 .sp .6
2867 .RS 4n
2868 Creates a snapshot whose contents are as specified in the stream provided on
2869 standard input. If a full stream is received, then a new file system is created
2870 as well. Streams are created using the \fBzfs send\fR subcommand, which by
2871 default creates a full stream. \fBzfs recv\fR can be used as an alias for
2872 \fBzfs receive\fR.
2873 .sp
2874 If an incremental stream is received, then the destination file system must
2875 already exist, and its most recent snapshot must match the incremental stream's
2876 source. For \fBzvol\fR, the destination device link is destroyed and
2877 recreated, which means the \fBzvol\fR cannot be accessed during the
2878 \fBreceive\fR operation.
2879 .sp
2880 When a snapshot replication package stream that is generated by using the
2881 \fBzfs send\fR \fB-R\fR command is received, any snapshots that do not exist
2882 on the sending location are destroyed by using the \fBzfs destroy\fR \fB-d\fR
2883 command.
2884 .sp

```

2885 The name of the snapshot (and file system, if a full stream is received) that
 2886 this subcommand creates depends on the argument type and the use of the
 2887 `\fB-d` or `\fB-e` options.
 2888 .sp
 2889 If the argument is a snapshot name, the specified `\fIsnapshot` is created. If
 2890 the argument is a file system or volume name, a snapshot with the same name as
 2891 the sent snapshot is created within the specified `\fIfilesystem` or
 2892 `\fIvolume`. If neither of the `\fB-d` or `\fB-e` options are specified,
 2893 the provided target snapshot name is used exactly as provided.
 2894 .sp
 2895 The `\fB-d` and `\fB-e` options cause the file system name of the target
 2896 snapshot to be determined by appending a portion of the sent snapshot's name to
 2897 the specified target `\fIfilesystem`. If the `\fB-d` option is specified, all
 2898 but the first element of the sent snapshot's file system path (usually the
 2899 pool name) is used and any required intermediate file systems within the
 2900 specified one are created. If the `\fB-e` option is specified, then only the
 2901 last element of the sent snapshot's file system name (i.e. the name of the
 2902 source file system itself) is used as the target file system name.
 2903 .sp
 2904 .ne 2
 2905 .na
 2906 `\fB\fB-d`
 2907 .ad
 2908 .sp .6
 2909 .RS 4n
 2910 Discard the first element of the sent snapshot's file system name, using
 2911 the remaining elements to determine the name of the target file system for
 2912 the new snapshot as described in the paragraph above.
 2913 .RE
 2915 .sp
 2916 .ne 2
 2917 .na
 2918 `\fB\fB-e`
 2919 .ad
 2920 .sp .6
 2921 .RS 4n
 2922 Discard all but the last element of the sent snapshot's file system name,
 2923 using that element to determine the name of the target file system for
 2924 the new snapshot as described in the paragraph above.
 2925 .RE
 2927 .sp
 2928 .ne 2
 2929 .na
 2930 `\fB\fB-u`
 2931 .ad
 2932 .sp .6
 2933 .RS 4n
 2934 File system that is associated with the received stream is not mounted.
 2935 .RE
 2937 .sp
 2938 .ne 2
 2939 .na
 2940 `\fB\fB-v`
 2941 .ad
 2942 .sp .6
 2943 .RS 4n
 2944 Print verbose information about the stream and the time required to perform the
 2945 receive operation.
 2946 .RE
 2948 .sp
 2949 .ne 2
 2950 .na

2951 `\fB\fB-n`
 2952 .ad
 2953 .sp .6
 2954 .RS 4n
 2955 Do not actually receive the stream. This can be useful in conjunction with the
 2956 `\fB-v` option to verify the name the receive operation would use.
 2957 .RE
 2959 .sp
 2960 .ne 2
 2961 .na
 2962 `\fB\fB-F`
 2963 .ad
 2964 .sp .6
 2965 .RS 4n
 2966 Force a rollback of the file system to the most recent snapshot before
 2967 performing the receive operation. If receiving an incremental replication
 2968 stream (for example, one generated by `\fBzfs send -R -[iI]`), destroy
 2969 snapshots and file systems that do not exist on the sending side.
 2970 .RE
 2972 .RE
 2974 .sp
 2975 .ne 2
 2976 .na
 2977 `\fB\fBzfs allow` `\fIfilesystem` `|` `\fIvolume`
 2978 .ad
 2979 .sp .6
 2980 .RS 4n
 2981 Displays permissions that have been delegated on the specified filesystem or
 2982 volume. See the other forms of `\fBzfs allow` for more information.
 2983 .RE
 2985 .sp
 2986 .ne 2
 2987 .na
 2988 `\fB\fBzfs allow` `[\fB-ldug]` `"\fIeveryone"` `|\fIuser` `|\fIgroup` `[\dots]`
 2989 `\fIperm` `|\fIsetname` `[\dots]` `\fIfilesystem` `|\fIvolume`
 2990 .ad
 2991 .br
 2992 .na
 2993 `\fB\fBzfs allow` `[\fB-ld]` `\fB-e` `\fIperm` `|\fIsetname` `[\dots]`
 2994 `\fIfilesystem` `|\fIvolume`
 2995 .ad
 2996 .sp .6
 2997 .RS 4n
 2998 Delegates `\fBZFS` administration permission for the file systems to
 2999 non-privileged users.
 3000 .sp
 3001 .ne 2
 3002 .na
 3003 `\fB[\fB-ug]` `"\fIeveryone"` `|\fIuser` `|\fIgroup` `[\dots]`
 3004 .ad
 3005 .sp .6
 3006 .RS 4n
 3007 Specifies to whom the permissions are delegated. Multiple entities can be
 3008 specified as a comma-separated list. If neither of the `\fB-ug` options are
 3009 specified, then the argument is interpreted preferentially as the keyword
 3010 "everyone", then as a user name, and lastly as a group name. To specify a user
 3011 or group named "everyone", use the `\fB-u` or `\fB-g` options. To specify a
 3012 group with the same name as a user, use the `\fB-g` options.
 3013 .RE
 3015 .sp
 3016 .ne 2

```

3017 .na
3018 \fB[\fB-e\fR] \fIperm\fR|@\fIsetname\fR[,...]\fR
3019 .ad
3020 .sp .6
3021 .RS 4n
3022 Specifies that the permissions be delegated to "everyone." Multiple permissions
3023 may be specified as a comma-separated list. Permission names are the same as
3024 \fBZFS\fR subcommand and property names. See the property list below. Property
3025 set names, which begin with an at sign (\fB@\fR) , may be specified. See the
3026 \fB-s\fR form below for details.
3027 .RE

3029 .sp
3030 .ne 2
3031 .na
3032 \fB[\fB-ld\fR] \fIfilesystem\fR|\fIvolume\fR\fR
3033 .ad
3034 .sp .6
3035 .RS 4n
3036 Specifies where the permissions are delegated. If neither of the \fB-ld\fR
3037 options are specified, or both are, then the permissions are allowed for the
3038 file system or volume, and all of its descendants. If only the \fB-l\fR option
3039 is used, then is allowed "locally" only for the specified file system. If only
3040 the \fB-d\fR option is used, then is allowed only for the descendent file
3041 systems.
3042 .RE

3044 .RE

3046 .sp
3047 .LP
3048 Permissions are generally the ability to use a \fBZFS\fR subcommand or change a
3049 \fBZFS\fR property. The following permissions are available:
3050 .sp
3051 .in +2
3052 .nf
3053 NAME          TYPE          NOTES
3054 allow         subcommand   Must also have the permission that is being
3055               subcommand   allowed
3056 clone        subcommand   Must also have the 'create' ability and 'mount'
3057               subcommand   ability in the origin file system
3058 create       subcommand   Must also have the 'mount' ability
3059 destroy      subcommand   Must also have the 'mount' ability
3060 mount        subcommand   Allows mount/umount of ZFS datasets
3061 promote      subcommand   Must also have the 'mount'
3062               subcommand   and 'promote' ability in the origin file system
3063 receive      subcommand   Must also have the 'mount' and 'create' ability
3064 rename       subcommand   Must also have the 'mount' and 'create'
3065               subcommand   ability in the new parent
3066 rollback    subcommand   Must also have the 'mount' ability
3067 send         subcommand
3068 share        subcommand   Allows sharing file systems over NFS or SMB
3069               subcommand   protocols
3070 snapshot     subcommand   Must also have the 'mount' ability
3071 groupquota  other        Allows accessing any groupquota@... property
3072 groupused   other        Allows reading any groupused@... property
3073 userprop    other        Allows changing any user property
3074 userquota  other        Allows accessing any userquota@... property
3075 userused   other        Allows reading any userused@... property

3077 aclinherit  property
3078 aclmode    property
3079 atime      property
3080 canmount   property
3081 casesensitivity property
3082 checksum   property

```

```

3083 compression property
3084 copies        property
3085 devices       property
3086 exec          property
3087 mountpoint    property
3088 nbmand        property
3089 normalization property
3090 primarycache  property
3091 quota         property
3092 readonly      property
3093 recordsize    property
3094 refquota      property
3095 refreservation property
3096 reservation   property
3097 secondarycache property
3098 setuid        property
3099 shareiscsi    property
3100 sharenfs      property
3101 sharesmb      property
3102 snapdir       property
3103 utf8only      property
3104 version       property
3105 volblocksize  property
3106 volsize       property
3107 vscan         property
3108 xattr         property
3109 zoned         property
3110 .fi
3111 .in -2
3112 .sp

3114 .sp
3115 .ne 2
3116 .na
3117 \fB\fBzfs allow\fR \fB-c\fR \fIperm\fR|@\fIsetname\fR[,...]\fR
3118 \fIfilesystem\fR|\fIvolume\fR\fR
3119 .ad
3120 .sp .6
3121 .RS 4n
3122 Sets "create time" permissions. These permissions are granted (locally) to the
3123 creator of any newly-created descendent file system.
3124 .RE

3126 .sp
3127 .ne 2
3128 .na
3129 \fB\fBzfs allow\fR \fB-s\fR @\fIsetname\fR \fIperm\fR|@\fIsetname\fR[,...]\fR
3130 \fIfilesystem\fR|\fIvolume\fR\fR
3131 .ad
3132 .sp .6
3133 .RS 4n
3134 Defines or adds permissions to a permission set. The set can be used by other
3135 \fBzfs allow\fR commands for the specified file system and its descendants.
3136 Sets are evaluated dynamically, so changes to a set are immediately reflected.
3137 Permission sets follow the same naming restrictions as ZFS file systems, but
3138 the name must begin with an "at sign" (\fB@\fR), and can be no more than 64
3139 characters long.
3140 .RE

3142 .sp
3143 .ne 2
3144 .na
3145 \fB\fBzfs unallow\fR [\fB-rldug\fR]
3146 "\fIeveryone\fR"|\fIuser\fR|\fIgroup\fR[,...]\fR
3147 [\fIperm\fR|@\fIsetname\fR[, ...]] \fIfilesystem\fR|\fIvolume\fR\fR
3148 .ad

```



```

3149 .br
3150 .na
3151 \fB\bzfs unallow\fR [\fB-rld\fR] \fB-e\fR [\fIperm\fR|@\fIsetname\fR [,...]]
3152 \fIfilesystem\fR|\fIvolume\fR\fR
3153 .ad
3154 .br
3155 .na
3156 \fB\bzfs unallow\fR [\fB-r\fR] \fB-c\fR [\fIperm\fR|@\fIsetname\fR [,...]]\fR
3157 .ad
3158 .br
3159 .na
3160 \fB\fIfilesystem\fR|\fIvolume\fR\fR
3161 .ad
3162 .sp .6
3163 .RS 4n
3164 Removes permissions that were granted with the \fBzfs allow\fR command. No
3165 permissions are explicitly denied, so other permissions granted are still in
3166 effect. For example, if the permission is granted by an ancestor. If no
3167 permissions are specified, then all permissions for the specified \fIuser\fR,
3168 \fIgroup\fR, or \fIeveryone\fR are removed. Specifying "everyone" (or using the
3169 \fB-e\fR option) only removes the permissions that were granted to "everyone",
3170 not all permissions for every user and group. See the \fBzfs allow\fR command
3171 for a description of the \fBldugec\fR options.
3172 .sp
3173 .ne 2
3174 .na
3175 \fB\bzfs r\fR\fR
3176 .ad
3177 .sp .6
3178 .RS 4n
3179 Recursively remove the permissions from this file system and all descendents.
3180 .RE

3182 .RE

3184 .sp
3185 .ne 2
3186 .na
3187 \fB\bzfs unallow\fR [\fB-r\fR] \fB-s\fR @\fIsetname\fR
3188 [\fIperm\fR|@\fIsetname\fR [,...]]\fR
3189 .ad
3190 .br
3191 .na
3192 \fB\fIfilesystem\fR|\fIvolume\fR\fR
3193 .ad
3194 .sp .6
3195 .RS 4n
3196 Removes permissions from a permission set. If no permissions are specified,
3197 then all permissions are removed, thus removing the set entirely.
3198 .RE

3200 .sp
3201 .ne 2
3202 .na
3203 \fB\bzfs hold\fR [\fB-r\fR] \fIitag\fR \fIsnapshot\fR...\fR
3204 .ad
3205 .sp .6
3206 .RS 4n
3207 Adds a single reference, named with the \fIitag\fR argument, to the specified
3208 snapshot or snapshots. Each snapshot has its own tag namespace, and tags must
3209 be unique within that space.
3210 .sp
3211 If a hold exists on a snapshot, attempts to destroy that snapshot by using the
3212 \fBzfs destroy\fR command return \fBBEBUSY\fR.
3213 .sp
3214 .ne 2

```

```

3215 .na
3216 \fB\bzfs r\fR\fR
3217 .ad
3218 .sp .6
3219 .RS 4n
3220 Specifies that a hold with the given tag is applied recursively to the
3221 snapshots of all descendent file systems.
3222 .RE

3224 .RE

3226 .sp
3227 .ne 2
3228 .na
3229 \fB\bzfs holds\fR [\fB-r\fR] \fIsnapshot\fR...\fR
3230 .ad
3231 .sp .6
3232 .RS 4n
3233 Lists all existing user references for the given snapshot or snapshots.
3234 .sp
3235 .ne 2
3236 .na
3237 \fB\bzfs r\fR\fR
3238 .ad
3239 .sp .6
3240 .RS 4n
3241 Lists the holds that are set on the named descendent snapshots, in addition to
3242 listing the holds on the named snapshot.
3243 .RE

3245 .RE

3247 .sp
3248 .ne 2
3249 .na
3250 \fB\bzfs release\fR [\fB-r\fR] \fIitag\fR \fIsnapshot\fR...\fR
3251 .ad
3252 .sp .6
3253 .RS 4n
3254 Removes a single reference, named with the \fIitag\fR argument, from the
3255 specified snapshot or snapshots. The tag must already exist for each snapshot.
3256 .sp
3257 If a hold exists on a snapshot, attempts to destroy that snapshot by using the
3258 \fBzfs destroy\fR command return \fBBEBUSY\fR.
3259 .sp
3260 .ne 2
3261 .na
3262 \fB\bzfs r\fR\fR
3263 .ad
3264 .sp .6
3265 .RS 4n
3266 Recursively releases a hold with the given tag on the snapshots of all
3267 descendent file systems.
3268 .RE

3270 .RE

3272 .SH EXAMPLES
3273 .LP
3274 \fBExample 1 \fRCreating a ZFS File System Hierarchy
3275 .sp
3276 .LP
3277 The following commands create a file system named \fBpool/home\fR and a file
3278 system named \fBpool/home/bob\fR. The mount point \fB/export/home\fR is set for
3279 the parent file system, and is automatically inherited by the child file
3280 system.

```

```

3282 .sp
3283 .in +2
3284 .nf
3285 # \fBzfs create pool/home\fR
3286 # \fBzfs set mountpoint=/export/home pool/home\fR
3287 # \fBzfs create pool/home/bob\fR
3288 .fi
3289 .in -2
3290 .sp

3292 .LP
3293 \fBExample 2 \fRCreating a ZFS Snapshot
3294 .sp
3295 .LP
3296 The following command creates a snapshot named \fByesterday\fR. This snapshot
3297 is mounted on demand in the \fB&.zfs/snapshot\fR directory at the root of the
3298 \fBpool/home/bob\fR file system.

3300 .sp
3301 .in +2
3302 .nf
3303 # \fBzfs snapshot pool/home/bob@yesterday\fR
3304 .fi
3305 .in -2
3306 .sp

3308 .LP
3309 \fBExample 3 \fRCreating and Destroying Multiple Snapshots
3310 .sp
3311 .LP
3312 The following command creates snapshots named \fByesterday\fR of
3313 \fBpool/home\fR and all of its descendent file systems. Each snapshot is
3314 mounted on demand in the \fB&.zfs/snapshot\fR directory at the root of its
3315 file system. The second command destroys the newly created snapshots.

3317 .sp
3318 .in +2
3319 .nf
3320 # \fBzfs snapshot -r pool/home@yesterday\fR
3321 # \fBzfs destroy -r pool/home@yesterday\fR
3322 .fi
3323 .in -2
3324 .sp

3326 .LP
3327 \fBExample 4 \fRDisabling and Enabling File System Compression
3328 .sp
3329 .LP
3330 The following command disables the \fBcompression\fR property for all file
3331 systems under \fBpool/home\fR. The next command explicitly enables
3332 \fBcompression\fR for \fBpool/home/anne\fR.

3334 .sp
3335 .in +2
3336 .nf
3337 # \fBzfs set compression=off pool/home\fR
3338 # \fBzfs set compression=on pool/home/anne\fR
3339 .fi
3340 .in -2
3341 .sp

3343 .LP
3344 \fBExample 5 \fRListing ZFS Datasets
3345 .sp
3346 .LP

```

```

3347 The following command lists all active file systems and volumes in the system.
3348 Snapshots are displayed if the \fBlistsnapshots\fR property is \fBon\fR. The
3349 default is \fBoff\fR. See \fBzpool\fR(1M) for more information on pool
3350 properties.

3352 .sp
3353 .in +2
3354 .nf
3355 # \fBzfs list\fR
3356 NAME USED AVAIL REFER MOUNTPOINT
3357 pool 450K 457G 18K /pool
3358 pool/home 315K 457G 21K /export/home
3359 pool/home/anne 18K 457G 18K /export/home/anne
3360 pool/home/bob 276K 457G 276K /export/home/bob
3361 .fi
3362 .in -2
3363 .sp

3365 .LP
3366 \fBExample 6 \fRSetting a Quota on a ZFS File System
3367 .sp
3368 .LP
3369 The following command sets a quota of 50 Gbytes for \fBpool/home/bob\fR.

3371 .sp
3372 .in +2
3373 .nf
3374 # \fBzfs set quota=50G pool/home/bob\fR
3375 .fi
3376 .in -2
3377 .sp

3379 .LP
3380 \fBExample 7 \fRListing ZFS Properties
3381 .sp
3382 .LP
3383 The following command lists all properties for \fBpool/home/bob\fR.

3385 .sp
3386 .in +2
3387 .nf
3388 # \fBzfs get all pool/home/bob\fR
3389 NAME PROPERTY VALUE SOURCE
3390 pool/home/bob type filesystem -
3391 pool/home/bob creation Tue Jul 21 15:53 2009 -
3392 pool/home/bob used 21K -
3393 pool/home/bob available 20.0G -
3394 pool/home/bob referenced 21K -
3395 pool/home/bob compression 1.00x -
3396 pool/home/bob mounted yes -
3397 pool/home/bob quota 20G local
3398 pool/home/bob reservation none default
3399 pool/home/bob recordsize 128K default
3400 pool/home/bob mountpoint /pool/home/bob default
3401 pool/home/bob sharenfs off default
3402 pool/home/bob checksum on default
3403 pool/home/bob compression on local
3404 pool/home/bob atime on default
3405 pool/home/bob devices on default
3406 pool/home/bob exec on default
3407 pool/home/bob setuid on default
3408 pool/home/bob readonly off default
3409 pool/home/bob zoned off default
3410 pool/home/bob snapdir hidden default
3411 pool/home/bob aclmode discard default
3412 pool/home/bob aclinherit restricted default

```

```

3413 pool/home/bob canmount on default
3414 pool/home/bob shareiscsi off default
3415 pool/home/bob xattr on default
3416 pool/home/bob copies 1 default
3417 pool/home/bob version 4 -
3418 pool/home/bob utf8only off -
3419 pool/home/bob normalization none -
3420 pool/home/bob casesensitivity sensitive -
3421 pool/home/bob vscan off default
3422 pool/home/bob nbmand off default
3423 pool/home/bob sharesmb off default
3424 pool/home/bob refquota none default
3425 pool/home/bob refreservation none default
3426 pool/home/bob primarycache all default
3427 pool/home/bob secondarycache all default
3428 pool/home/bob usedbysnapshots 0 -
3429 pool/home/bob usedbydataset 21K -
3430 pool/home/bob usedbychildren 0 -
3431 pool/home/bob usedbyrefreservation 0 -
3432 .fi
3433 .in -2
3434 .sp

```

```

3436 .sp
3437 .LP
3438 The following command gets a single property value.

```

```

3440 .sp
3441 .in +2
3442 .nf
3443 # \fBzfs get -H -o value compression pool/home/bob\fR
3444 on
3445 .fi
3446 .in -2
3447 .sp

```

```

3449 .sp
3450 .LP
3451 The following command lists all properties with local settings for
3452 \fBpool/home/bob\fR.

```

```

3454 .sp
3455 .in +2
3456 .nf
3457 # \fBzfs get -r -s local -o name,property,value all pool/home/bob\fR
3458 NAME PROPERTY VALUE
3459 pool/home/bob quota 20G
3460 pool/home/bob compression on
3461 .fi
3462 .in -2
3463 .sp

```

```

3465 .LP
3466 \fBExample 8 \fRRolling Back a ZFS File System
3467 .sp
3468 .LP
3469 The following command reverts the contents of \fBpool/home/anne\fR to the
3470 snapshot named \fByesterday\fR, deleting all intermediate snapshots.

```

```

3472 .sp
3473 .in +2
3474 .nf
3475 # \fBzfs rollback -r pool/home/anne@yesterday\fR
3476 .fi
3477 .in -2
3478 .sp

```

```

3480 .LP
3481 \fBExample 9 \fRCreating a ZFS Clone
3482 .sp
3483 .LP
3484 The following command creates a writable file system whose initial contents are
3485 the same as \fBpool/home/bob@yesterday\fR.

```

```

3487 .sp
3488 .in +2
3489 .nf
3490 # \fBzfs clone pool/home/bob@yesterday pool/clone\fR
3491 .fi
3492 .in -2
3493 .sp

```

```

3495 .LP
3496 \fBExample 10 \fRPromoting a ZFS Clone
3497 .sp
3498 .LP
3499 The following commands illustrate how to test out changes to a file system, and
3500 then replace the original file system with the changed one, using clones, clone
3501 promotion, and renaming:

```

```

3503 .sp
3504 .in +2
3505 .nf
3506 # \fBzfs create pool/project/production\fR
3507 populate /pool/project/production with data
3508 # \fBzfs snapshot pool/project/production@today\fR
3509 # \fBzfs clone pool/project/production@today pool/project/beta\fR
3510 make changes to /pool/project/beta and test them
3511 # \fBzfs promote pool/project/beta\fR
3512 # \fBzfs rename pool/project/production pool/project/legacy\fR
3513 # \fBzfs rename pool/project/beta pool/project/production\fR
3514 once the legacy version is no longer needed, it can be destroyed
3515 # \fBzfs destroy pool/project/legacy\fR
3516 .fi
3517 .in -2
3518 .sp

```

```

3520 .LP
3521 \fBExample 11 \fRInheriting ZFS Properties
3522 .sp
3523 .LP
3524 The following command causes \fBpool/home/bob\fR and \fBpool/home/anne\fR to
3525 inherit the \fBchecksum\fR property from their parent.

```

```

3527 .sp
3528 .in +2
3529 .nf
3530 # \fBzfs inherit checksum pool/home/bob pool/home/anne\fR
3531 .fi
3532 .in -2
3533 .sp

```

```

3535 .LP
3536 \fBExample 12 \fRRemotely Replicating ZFS Data
3537 .sp
3538 .LP
3539 The following commands send a full stream and then an incremental stream to a
3540 remote machine, restoring them into \fBpoolB/received/fs@a\fRand
3541 \fBpoolB/received/fs@b\fR, respectively. \fBpoolB\fR must contain the file
3542 system \fBpoolB/received\fR, and must not initially contain
3543 \fBpoolB/received/fs\fR.

```

```

3545 .sp
3546 .in +2
3547 .nf
3548 # \fBzfs send pool/fs@a | \e\fR
3549   \fBssh host zfs receive poolB/received/fs@a\fR
3550 # \fBzfs send -i a pool/fs@b | ssh host \e\fR
3551   \fBzfs receive poolB/received/fs\fR
3552 .fi
3553 .in -2
3554 .sp

3556 .LP
3557 \fBExample 13 \fRUsing the \fBzfs receive\fR \fB-d\fR Option
3558 .sp
3559 .LP
3560 The following command sends a full stream of \fBpoolA/fsA/fsB@snap\fR to a
3561 remote machine, receiving it into \fBpoolB/received/fsA/fsB@snap\fR. The
3562 \fBfsA/fsB@snap\fR portion of the received snapshot's name is determined from
3563 the name of the sent snapshot. \fBpoolB\fR must contain the file system
3564 \fBpoolB/received\fR. If \fBpoolB/received/fsA\fR does not exist, it is created
3565 as an empty file system.

3567 .sp
3568 .in +2
3569 .nf
3570 # \fBzfs send poolA/fsA/fsB@snap | \e
3571   ssh host zfs receive -d poolB/received\fR
3572 .fi
3573 .in -2
3574 .sp

3576 .LP
3577 \fBExample 14 \fRSetting User Properties
3578 .sp
3579 .LP
3580 The following example sets the user-defined \fBcom.example:department\fR
3581 property for a dataset.

3583 .sp
3584 .in +2
3585 .nf
3586 # \fBzfs set com.example:department=12345 tank/accounting\fR
3587 .fi
3588 .in -2
3589 .sp

3591 .LP
3592 \fBExample 15 \fRCreating a ZFS Volume as an iSCSI Target Device
3593 .sp
3594 .LP
3595 The following example shows how to create a \fBZFS\fR volume as an \fBiSCSI\fR
3596 target.

3598 .sp
3599 .in +2
3600 .nf
3601 # \fBzfs create -V 2g pool/volumes/voll\fR
3602 # \fBzfs set shareiscsi=on pool/volumes/voll\fR
3603 # \fBiscsitadm list target\fR
3604 Target: pool/volumes/voll
3605 iSCSI Name:
3606 iqn.1986-03.com.sun:02:7b4b02a6-3277-eb1b-e686-a24762c52a8c
3607 Connections: 0
3608 .fi
3609 .in -2
3610 .sp

```

```

3612 .sp
3613 .LP
3614 After the \fBiSCSI\fR target is created, set up the \fBiSCSI\fR initiator. For
3615 more information about the Solaris \fBiSCSI\fR initiator, see
3616 \fBiscsitadm\fR(1M).
3617 .LP
3618 \fBExample 16 \fRPerforming a Rolling Snapshot
3619 .sp
3620 .LP
3621 The following example shows how to maintain a history of snapshots with a
3622 consistent naming scheme. To keep a week's worth of snapshots, the user
3623 destroys the oldest snapshot, renames the remaining snapshots, and then creates
3624 a new snapshot, as follows:

3626 .sp
3627 .in +2
3628 .nf
3629 # \fBzfs destroy -r pool/users@7daysago\fR
3630 # \fBzfs rename -r pool/users@6daysago @7daysago\fR
3631 # \fBzfs rename -r pool/users@5daysago @6daysago\fR
3632 # \fBzfs rename -r pool/users@yesterday @5daysago\fR
3633 # \fBzfs rename -r pool/users@yesterday @4daysago\fR
3634 # \fBzfs rename -r pool/users@yesterday @3daysago\fR
3635 # \fBzfs rename -r pool/users@yesterday @2daysago\fR
3636 # \fBzfs rename -r pool/users@today @yesterday\fR
3637 # \fBzfs snapshot -r pool/users@today\fR
3638 .fi
3639 .in -2
3640 .sp

3642 .LP
3643 \fBExample 17 \fRSetting \fBsharens\fR Property Options on a ZFS File System
3644 .sp
3645 .LP
3646 The following commands show how to set \fBsharens\fR property options to
3647 enable \fBBrw\fR access for a set of \fBIP\fR addresses and to enable root
3648 access for system \fBneo\fR on the \fBtank/home\fR file system.

3650 .sp
3651 .in +2
3652 .nf
3653 # \fBzfs set sharens='rw=@123.123.0.0/16,root=neo' tank/home\fR
3654 .fi
3655 .in -2
3656 .sp

3658 .sp
3659 .LP
3660 If you are using \fBBDNS\fR for host name resolution, specify the fully
3661 qualified hostname.

3663 .LP
3664 \fBExample 18 \fRDelegating ZFS Administration Permissions on a ZFS Dataset
3665 .sp
3666 .LP
3667 The following example shows how to set permissions so that user \fBcindys\fR
3668 can create, destroy, mount, and take snapshots on \fBtank/cindys\fR. The
3669 permissions on \fBtank/cindys\fR are also displayed.

3671 .sp
3672 .in +2
3673 .nf
3674 # \fBzfs allow cindys create,destroy,mount,snapshot tank/cindys\fR
3675 # \fBzfs allow tank/cindys\fR
3676 -----

```

```

3677 Local+Descendent permissions on (tank/cindys)
3678     user cindys create,destroy,mount,snapshot
3679 -----
3680 .fi
3681 .in -2
3682 .sp

3684 .sp
3685 .LP
3686 Because the \fBtank/cindys\fR mount point permission is set to 755 by default,
3687 user \fBcindys\fR will be unable to mount file systems under \fBtank/cindys\fR.
3688 Set an \fBACL\fR similar to the following syntax to provide mount point access:
3689 .sp
3690 .in +2
3691 .nf
3692 # \fBchmod A+user:cindys:add_subdirectory:allow /tank/cindys\fR
3693 .fi
3694 .in -2
3695 .sp

3697 .LP
3698 \fBExample 19 \fRDelegating Create Time Permissions on a ZFS Dataset
3699 .sp
3700 .LP
3701 The following example shows how to grant anyone in the group \fBstaff\fR to
3702 create file systems in \fBtank/users\fR. This syntax also allows staff members
3703 to destroy their own file systems, but not destroy anyone else's file system.
3704 The permissions on \fBtank/users\fR are also displayed.

3706 .sp
3707 .in +2
3708 .nf
3709 # \fB# zfs allow staff create,mount tank/users\fR
3710 # \fBzfs allow -c destroy tank/users\fR
3711 # \fBzfs allow tank/users\fR
3712 -----
3713 Create time permissions on (tank/users)
3714     create,destroy
3715 Local+Descendent permissions on (tank/users)
3716     group staff create,mount
3717 -----
3718 .fi
3719 .in -2
3720 .sp

3722 .LP
3723 \fBExample 20 \fRDefining and Granting a Permission Set on a ZFS Dataset
3724 .sp
3725 .LP
3726 The following example shows how to define and grant a permission set on the
3727 \fBtank/users\fR file system. The permissions on \fBtank/users\fR are also
3728 displayed.

3730 .sp
3731 .in +2
3732 .nf
3733 # \fBzfs allow -s @pset create,destroy,snapshot,mount tank/users\fR
3734 # \fBzfs allow staff @pset tank/users\fR
3735 # \fBzfs allow tank/users\fR
3736 -----
3737 Permission sets on (tank/users)
3738     @pset create,destroy,mount,snapshot
3739 Create time permissions on (tank/users)
3740     create,destroy
3741 Local+Descendent permissions on (tank/users)
3742     group staff @pset,create,mount

```

```

3743 -----
3744 .fi
3745 .in -2
3746 .sp

3748 .LP
3749 \fBExample 21 \fRDelegating Property Permissions on a ZFS Dataset
3750 .sp
3751 .LP
3752 The following example shows to grant the ability to set quotas and reservations
3753 on the \fBusers/home\fR file system. The permissions on \fBusers/home\fR are
3754 also displayed.

3756 .sp
3757 .in +2
3758 .nf
3759 # \fBzfs allow cindys quota,reservation users/home\fR
3760 # \fBzfs allow users/home\fR
3761 -----
3762 Local+Descendent permissions on (users/home)
3763     user cindys quota,reservation
3764 -----
3765 cindys% \fBzfs set quota=10G users/home/marks\fR
3766 cindys% \fBzfs get quota users/home/marks\fR
3767 NAME                PROPERTY VALUE          SOURCE
3768 users/home/marks    quota      10G             local
3769 .fi
3770 .in -2
3771 .sp

3773 .LP
3774 \fBExample 22 \fRRemoving ZFS Delegated Permissions on a ZFS Dataset
3775 .sp
3776 .LP
3777 The following example shows how to remove the snapshot permission from the
3778 \fBstaff\fR group on the \fBtank/users\fR file system. The permissions on
3779 \fBtank/users\fR are also displayed.

3781 .sp
3782 .in +2
3783 .nf
3784 # \fBzfs unallow staff snapshot tank/users\fR
3785 # \fBzfs allow tank/users\fR
3786 -----
3787 Permission sets on (tank/users)
3788     @pset create,destroy,mount,snapshot
3789 Create time permissions on (tank/users)
3790     create,destroy
3791 Local+Descendent permissions on (tank/users)
3792     group staff @pset,create,mount
3793 -----
3794 .fi
3795 .in -2
3796 .sp

3798 .SH EXIT STATUS
3799 .sp
3800 .LP
3801 The following exit values are returned:
3802 .sp
3803 .ne 2
3804 .na
3805 \fB0\fR
3806 .ad
3807 .sp .6
3808 .RS 4n

```

```
3809 Successful completion.
3810 .RE

3812 .sp
3813 .ne 2
3814 .na
3815 \fB\fB1\fR\fR
3816 .ad
3817 .sp .6
3818 .RS 4n
3819 An error occurred.
3820 .RE

3822 .sp
3823 .ne 2
3824 .na
3825 \fB\fB2\fR\fR
3826 .ad
3827 .sp .6
3828 .RS 4n
3829 Invalid command line options were specified.
3830 .RE

3832 .SH ATTRIBUTES
3833 .sp
3834 .LP
3835 See \fBattributes\fR(5) for descriptions of the following attributes:
3836 .sp

3838 .sp
3839 .TS
3840 box;
3841 c | c
3842 l | l .
3843 ATTRIBUTE TYPE ATTRIBUTE VALUE
3844 _
3845 Interface Stability Committed
3846 .TE

3848 .SH SEE ALSO
3849 .sp
3850 .LP
3851 \fBbssh\fR(1), \fBbiscsitadm\fR(1M), \fBbmount\fR(1M), \fBbshare\fR(1M),
3852 \fBbsharemgr\fR(1M), \fBbunshare\fR(1M), \fBbzonecfg\fR(1M), \fBbzpool\fR(1M),
3853 \fBbchmod\fR(2), \fBbstat\fR(2), \fBbwrite\fR(2), \fBbfsync\fR(3C),
3854 \fBbdfstab\fR(4), \fBattributes\fR(5)
3855 .sp
3856 .LP
3857 See the \fBgzip\fR(1) man page, which is not part of the SunOS man page
3858 collection.
3859 .sp
3860 .LP
3861 For information about using the \fBZFS\fR web-based management tool and other
3862 \fBZFS\fR features, see the \fBSolaris ZFS Administration Guide\fR.
```

new/usr/src/pkg/manifests/system-file-system-zfs.mf

1

```
*****
8107 Thu Jun 28 15:57:14 2012
new/usr/src/pkg/manifests/system-file-system-zfs.mf
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 #
23 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright (c) 2012 by Delphix. All rights reserved.
25 #
26 #
27 set name=pkg.fmri value=pkg:/system/file-system/zfs@$(PKGVERS)
28 set name=pkg.description value="ZFS libraries and commands"
29 set name=pkg.summary value=ZFS
30 set name=info.classification \
31     value="org.opensolaris.category.2008:System/File System"
32 #
33 #
34 # Skip manifest generation until build 134 can be required on the
35 # build systems, due to a Python flavor identification bug in pkgdepend.
36 #
37 set name=org.opensolaris.nodepend value=true
38 set name=variant.arch value=$(ARCH)
39 dir path=etc group=sys
40 dir path=etc/fs group=sys
41 dir path=etc/fs/zfs group=sys
42 dir path=etc/sysevent group=sys
43 dir path=etc/sysevent/config group=sys
44 dir path=etc/zfs group=sys
45 dir path=kernel group=sys
46 dir path=kernel/drv group=sys
47 dir path=kernel/drv/$(ARCH64) group=sys
48 dir path=kernel/fs group=sys
49 dir path=kernel/fs/$(ARCH64) group=sys
50 dir path=kernel/kmdb group=sys
51 dir path=kernel/kmdb/$(ARCH64) group=sys
52 dir path=lib
53 dir path=lib/$(ARCH64)
54 dir path=sbin group=sys
```

new/usr/src/pkg/manifests/system-file-system-zfs.mf

2

```
55 dir path=usr group=sys
56 dir path=usr/lib
57 dir path=usr/lib/$(ARCH64)
58 dir path=usr/lib/devfsadm group=sys
59 dir path=usr/lib/devfsadm/linkmod group=sys
60 dir path=usr/lib/fs group=sys
61 dir path=usr/lib/fs/zfs group=sys
62 dir path=usr/lib/mdb group=sys
63 dir path=usr/lib/mdb/kvm group=sys
64 dir path=usr/lib/mdb/kvm/$(ARCH64) group=sys
65 dir path=usr/lib/mdb/proc group=sys
66 $(sparc_ONLY)dir path=usr/lib/mdb/proc/$(ARCH64) group=sys
67 $(i386_ONLY)dir path=usr/lib/mdb/proc/$(ARCH64)
68 dir path=usr/lib/python2.6
69 dir path=usr/lib/python2.6/vendor-packages
70 dir path=usr/lib/python2.6/vendor-packages/zfs
71 dir path=usr/lib/sysevent
72 dir path=usr/lib/sysevent/modules
73 dir path=usr/lib/zfs
74 dir path=usr/sbin
75 $(i386_ONLY)dir path=usr/sbin/$(ARCH32)
76 dir path=usr/sbin/$(ARCH64)
77 dir path=usr/share/man/man1m
78 driver name=zfs perms="* 0600 root sys" perms="zfs 0666 root sys"
79 file \
80     path=etc/sysevent/config/SUNW_EC_zfs,ESC_ZFS_bootfs_vdev_attach,sysevent.conf
81     group=sys
82 file path=kernel/drv/$(ARCH64)/zfs group=sys
83 $(i386_ONLY)file path=kernel/drv/zfs group=sys
84 file path=kernel/drv/zfs.conf group=sys
85 file path=kernel/kmdb/$(ARCH64)/zfs group=sys mode=0555
86 $(i386_ONLY)file path=kernel/kmdb/zfs group=sys mode=0555
87 file path=lib/$(ARCH64)/libzfs.so.1
88 file path=lib/$(ARCH64)/libzfs_core.so.1
89 #endif /* !codereview */
90 file path=lib/$(ARCH64)/llib-lzfs.ln
91 file path=lib/$(ARCH64)/llib-lzfs_core.ln
92 #endif /* !codereview */
93 file path=lib/libzfs.so.1
94 file path=lib/libzfs_core.so.1
95 #endif /* !codereview */
96 file path=lib/llib-lzfs
97 file path=lib/llib-lzfs.ln
98 file path=lib/llib-lzfs_core
99 file path=lib/llib-lzfs_core.ln
100 #endif /* !codereview */
101 file path=sbin/zfs mode=0555
102 file path=sbin/zpool mode=0555
103 file path=usr/lib/$(ARCH64)/libzfs_jni.so.1
104 file path=usr/lib/$(ARCH64)/libzpool.so.1
105 file path=usr/lib/devfsadm/linkmod/SUNW_zfs_link.so group=sys
106 file path=usr/lib/fs/zfs/bootinstall mode=0555
107 file path=usr/lib/fs/zfs/fstyp.so.1 mode=0555
108 file path=usr/lib/libzfs_jni.so.1
109 $(i386_ONLY)file path=usr/lib/libzpool.so.1
110 file path=usr/lib/mdb/kvm/$(ARCH64)/zfs.so group=sys mode=0555
111 $(i386_ONLY)file path=usr/lib/mdb/kvm/zfs.so group=sys mode=0555
112 file path=usr/lib/mdb/proc/$(ARCH64)/libzpool.so group=sys mode=0555
113 file path=usr/lib/mdb/proc/libzpool.so group=sys mode=0555
114 file path=usr/lib/python2.6/vendor-packages/zfs/__init__.py
115 file path=usr/lib/python2.6/vendor-packages/zfs/__init__.pyc
116 file path=usr/lib/python2.6/vendor-packages/zfs/allow.py
117 file path=usr/lib/python2.6/vendor-packages/zfs/allow.pyc
118 file path=usr/lib/python2.6/vendor-packages/zfs/dataset.py
119 file path=usr/lib/python2.6/vendor-packages/zfs/dataset.pyc
120 file path=usr/lib/python2.6/vendor-packages/zfs/groupspace.py
```

```

121 file path=usr/lib/python2.6/vendor-packages/zfs/groupspace.pyc
122 file path=usr/lib/python2.6/vendor-packages/zfs/holds.py
123 file path=usr/lib/python2.6/vendor-packages/zfs/holds.pyc
124 file path=usr/lib/python2.6/vendor-packages/zfs/ioctl.so
125 file path=usr/lib/python2.6/vendor-packages/zfs/table.py
126 file path=usr/lib/python2.6/vendor-packages/zfs/table.pyc
127 file path=usr/lib/python2.6/vendor-packages/zfs/unallow.py
128 file path=usr/lib/python2.6/vendor-packages/zfs/unallow.pyc
129 file path=usr/lib/python2.6/vendor-packages/zfs/userspace.py
130 file path=usr/lib/python2.6/vendor-packages/zfs/userspace.pyc
131 file path=usr/lib/python2.6/vendor-packages/zfs/util.py
132 file path=usr/lib/python2.6/vendor-packages/zfs/util.pyc
133 file path=usr/lib/sysevent/modules/zfs_mod.so group=sys
134 file path=usr/lib/zfs/availdevs mode=0555
135 file path=usr/lib/zfs/pyzfs.py mode=0555
136 file path=usr/lib/zfs/pyzfs.pyc mode=0555
137 $(i386_ONLY)file path=usr/sbin/$(ARCH32)/zdb mode=0555
138 file path=usr/sbin/$(ARCH64)/zdb mode=0555
139 file path=usr/sbin/zstreamdump mode=0555
140 file path=usr/share/man/man1m/zdb.1m
141 file path=usr/share/man/man1m/zfs.1m
142 file path=usr/share/man/man1m/zpool.1m
143 file path=usr/share/man/man1m/zstreamdump.1m
144 file path=usr/share/man/man5/zpool-features.5
145 hardlink path=kernel/fs/$(ARCH64)/zfs target=../../kernel/drv/$(ARCH64)/zfs
146 $(i386_ONLY)hardlink path=kernel/fs/zfs target=../../kernel/drv/zfs
147 hardlink path=usr/lib/fs/zfs/fstyp target=../../sbin/fstyp
148 hardlink path=usr/sbin/zdb target=../../usr/lib/isaexec
149 legacy pkg=SUNWzfskr desc="ZFS kernel root components" \
150     name="ZFS Kernel (Root)"
151 legacy pkg=SUNWzfsr desc="ZFS root components" name="ZFS (Root)"
152 legacy pkg=SUNWzfsu desc="ZFS libraries and commands" name="ZFS (Usr)"
153 license cr_Sun license=cr_Sun
154 license lic_CDDL license=lic_CDDL
155 link path=etc/fs/zfs/mount target=../../sbin/zfs
156 link path=etc/fs/zfs/umount target=../../sbin/zfs
157 link path=lib/$(ARCH64)/libzfs.so target=libzfs.so.1
158 link path=lib/$(ARCH64)/libzfs_core.so target=libzfs_core.so.1
159 #endif /* ! codereview */
160 link path=lib/libzfs.so target=libzfs.so.1
161 link path=lib/libzfs_core.so target=libzfs_core.so.1
162 #endif /* ! codereview */
163 link path=usr/lib/$(ARCH64)/libzfs.so \
164     target=../../lib/$(ARCH64)/libzfs.so.1
165 link path=usr/lib/$(ARCH64)/libzfs.so.1 \
166     target=../../lib/$(ARCH64)/libzfs.so.1
167 link path=usr/lib/$(ARCH64)/libzfs_core.so \
168     target=../../lib/$(ARCH64)/libzfs_core.so.1
169 link path=usr/lib/$(ARCH64)/libzfs_core.so.1 \
170     target=../../lib/$(ARCH64)/libzfs_core.so.1
171 #endif /* ! codereview */
172 link path=usr/lib/$(ARCH64)/libzfs_jni.so target=libzfs_jni.so.1
173 link path=usr/lib/$(ARCH64)/libzpool.so target=libzpool.so.1
174 link path=usr/lib/$(ARCH64)/llib-lzfs.ln \
175     target=../../lib/$(ARCH64)/llib-lzfs.ln
176 link path=usr/lib/$(ARCH64)/llib-lzfs_core.ln \
177     target=../../lib/$(ARCH64)/llib-lzfs_core.ln
178 #endif /* ! codereview */
179 link path=usr/lib/fs/zfs/mount target=../../sbin/zfs
180 link path=usr/lib/fs/zfs/umount target=../../sbin/zfs
181 link path=usr/lib/libzfs.so target=../../lib/libzfs.so.1
182 link path=usr/lib/libzfs_core.so target=../../lib/libzfs_core.so.1
183 link path=usr/lib/libzfs_core.so target=../../lib/libzfs_core.so.1
184 link path=usr/lib/libzfs_core.so.1 target=../../lib/libzfs_core.so.1
185 #endif /* ! codereview */
186 link path=usr/lib/libzfs_jni.so target=libzfs_jni.so.1

```

```

187 $(i386_ONLY)link path=usr/lib/libzpool.so target=libzpool.so.1
188 link path=usr/lib/llib-lzfs target=../../lib/llib-lzfs
189 link path=usr/lib/llib-lzfs.ln target=../../lib/llib-lzfs.ln
190 link path=usr/lib/llib-lzfs_core target=../../lib/llib-lzfs_core
191 link path=usr/lib/llib-lzfs_core.ln target=../../lib/llib-lzfs_core.ln
192 #endif /* ! codereview */
193 link path=usr/sbin/zfs target=../../sbin/zfs
194 link path=usr/sbin/zpool target=../../sbin/zpool
195 depend fmri=runtime/python-26 type=require

```


new/usr/src/pkg/manifests/system-header.mf

1

```
*****
88403 Thu Jun 28 15:57:14 2012
new/usr/src/pkg/manifests/system-header.mf
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 #
23 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
24 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
25 # Copyright (c) 2012 by Delphix. All rights reserved.
26 #endif /* ! codereview */
27 #
28 #
29 set name=pkg.fmri value=pkg:/system/header@$(PKGVERS)
30 set name=pkg.description \
31     value="SunOS C/C++ header files for general development of software"
32 set name=pkg.summary value="SunOS Header Files"
33 set name=info.classification value=org.opensolaris.category.2008:System/Core
34 set name=variant.arch value=$(ARCH)
35 dir path=usr group=sys
36 dir path=usr/include
37 $(i386_ONLY)dir path=usr/include/$(ARCH64)
38 $(i386_ONLY)dir path=usr/include/$(ARCH64)/sys
39 dir path=usr/include/arpa
40 dir path=usr/include/asm
41 dir path=usr/include/ast
42 dir path=usr/include/bsm
43 dir path=usr/include/dat
44 dir path=usr/include/des
45 dir path=usr/include/gssapi
46 dir path=usr/include/hal
47 $(i386_ONLY)dir path=usr/include/ia32
48 $(i386_ONLY)dir path=usr/include/ia32/sys
49 dir path=usr/include/inet
50 dir path=usr/include/inet/kssl
51 dir path=usr/include/ipp
52 dir path=usr/include/ipp/ipgpc
53 dir path=usr/include/iso
54 dir path=usr/include/kerberosv5
```

new/usr/src/pkg/manifests/system-header.mf

2

```
55 dir path=usr/include/libpolkit
56 dir path=usr/include/net
57 dir path=usr/include/netinet
58 dir path=usr/include/nfs
59 dir path=usr/include/protocols
60 dir path=usr/include/rpc
61 dir path=usr/include/rpcsvc
62 dir path=usr/include/sasl
63 dir path=usr/include/scsi
64 dir path=usr/include/scsi/plugins
65 dir path=usr/include/scsi/plugins/ses
66 dir path=usr/include/scsi/plugins/ses/framework
67 dir path=usr/include/scsi/plugins/ses/vendor
68 dir path=usr/include/scsi/plugins/smp
69 dir path=usr/include/scsi/plugins/smp/engine
70 dir path=usr/include/scsi/plugins/smp/framework
71 dir path=usr/include/security
72 dir path=usr/include/sharefs
73 dir path=usr/include/sys
74 dir path=usr/include/sys/av
75 dir path=usr/include/sys/contract
76 dir path=usr/include/sys/crypto
77 dir path=usr/include/sys/dktp
78 dir path=usr/include/sys/fc4
79 dir path=usr/include/sys/fm
80 dir path=usr/include/sys/fm/cpu
81 dir path=usr/include/sys/fm/fs
82 dir path=usr/include/sys/fm/io
83 $(sparc_ONLY)dir path=usr/include/sys/fpu
84 dir path=usr/include/sys/fs
85 dir path=usr/include/sys/hotplug
86 dir path=usr/include/sys/hotplug/pci
87 dir path=usr/include/sys/ib
88 dir path=usr/include/sys/ib/adapters
89 dir path=usr/include/sys/ib/adapters/hermon
90 dir path=usr/include/sys/ib/adapters/tavor
91 dir path=usr/include/sys/ib/clients
92 dir path=usr/include/sys/ib/clients/ibd
93 dir path=usr/include/sys/ib/clients/of
94 dir path=usr/include/sys/ib/clients/of/rdma
95 dir path=usr/include/sys/ib/clients/of/sol_ofs
96 dir path=usr/include/sys/ib/clients/of/sol_ucma
97 dir path=usr/include/sys/ib/clients/of/sol_umad
98 dir path=usr/include/sys/ib/clients/of/sol_uverbs
99 dir path=usr/include/sys/ib/ibnex
100 dir path=usr/include/sys/ib/ibt1
101 dir path=usr/include/sys/ib/ibt1/impl
102 dir path=usr/include/sys/ib/mgt
103 dir path=usr/include/sys/ib/mgt/ibmf
104 dir path=usr/include/sys/iso
105 dir path=usr/include/sys/lvm
106 dir path=usr/include/sys/pcmcia
107 dir path=usr/include/sys/proc
108 dir path=usr/include/sys/rsm
109 $(i386_ONLY)dir path=usr/include/sys/sata group=sys
110 dir path=usr/include/sys/scsi
111 dir path=usr/include/sys/scsi/adapters
112 dir path=usr/include/sys/scsi/conf
113 dir path=usr/include/sys/scsi/generic
114 dir path=usr/include/sys/scsi/impl
115 dir path=usr/include/sys/scsi/targets
116 dir path=usr/include/sys/sysevent
117 dir path=usr/include/sys/tsol
118 dir path=usr/include/tsol
119 dir path=usr/include/uuid
120 $(sparc_ONLY)dir path=usr/include/v7
```

```

121 $(sparc_ONLY)dir path=usr/include/v7/sys
122 $(sparc_ONLY)dir path=usr/include/v9
123 $(sparc_ONLY)dir path=usr/include/v9/sys
124 dir path=usr/include/vm
125 dir path=usr/platform group=sys
126 $(sparc_ONLY)dir path=usr/platform/SUNW,A70 group=sys
127 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300 group=sys
128 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300/include
129 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010 group=sys
130 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010/include
131 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T12 group=sys
132 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T4 group=sys
133 $(sparc_ONLY)dir path=usr/platform/SUNW,SPARC-Enterprise group=sys
134 $(sparc_ONLY)dir path=usr/platform/SUNW,Serverbladel group=sys
135 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-100 group=sys
136 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1000 group=sys
137 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1500 group=sys
138 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-2500 group=sys
139 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire group=sys
140 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-15000 group=sys
141 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-280R group=sys
142 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-480R group=sys
143 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-880 group=sys
144 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V215 group=sys
145 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V240 group=sys
146 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V250 group=sys
147 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V440 group=sys
148 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V445 group=sys
149 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V490 group=sys
150 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V890 group=sys
151 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-2 group=sys
152 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-250 group=sys
153 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-4 group=sys
154 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise group=sys
155 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise-10000 group=sys
156 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40 group=sys
157 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60 group=sys
158 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-III-NetraCT group=sys
159 $(i386_ONLY)dir path=usr/platform/i86pc group=sys
160 $(i386_ONLY)dir path=usr/platform/i86pc/include
161 $(i386_ONLY)dir path=usr/platform/i86pc/include/sys
162 $(i386_ONLY)dir path=usr/platform/i86pc/include/vm
163 $(i386_ONLY)dir path=usr/platform/i86xpv group=sys
164 $(i386_ONLY)dir path=usr/platform/i86xpv/include
165 $(i386_ONLY)dir path=usr/platform/i86xpv/include/sys
166 $(i386_ONLY)dir path=usr/platform/i86xpv/include/vm
167 $(sparc_ONLY)dir path=usr/platform/sun4u group=sys
168 $(sparc_ONLY)dir path=usr/platform/sun4u/include
169 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys
170 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c
171 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/clients
172 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/misc
173 $(sparc_ONLY)dir path=usr/platform/sun4u/include/vm
174 $(sparc_ONLY)dir path=usr/platform/sun4v group=sys
175 $(sparc_ONLY)dir path=usr/platform/sun4v/include
176 $(sparc_ONLY)dir path=usr/platform/sun4v/include/sys
177 $(sparc_ONLY)dir path=usr/platform/sun4v/include/vm
178 dir path=usr/share
179 dir path=usr/share/man
180 dir path=usr/share/man/man3head
181 dir path=usr/share/man/man4
182 dir path=usr/share/man/man5
183 dir path=usr/share/man/man7i
184 dir path=usr/share/src group=sys
185 dir path=usr/share/src/uts
186 $(i386_ONLY)dir path=usr/share/src/uts/i86pc

```

```

187 $(i386_ONLY)dir path=usr/share/src/uts/i86xpv
188 $(sparc_ONLY)dir path=usr/share/src/uts/sun4u
189 $(sparc_ONLY)dir path=usr/share/src/uts/sun4v
190 dir path=usr/xpg4
191 dir path=usr/xpg4/include
192 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/kdi_regs.h
193 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privmregs.h
194 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privregs.h
195 file path=usr/include/aio.h
196 file path=usr/include/alloca.h
197 file path=usr/include/aptrace.h
198 file path=usr/include/aptrace_impl.h
199 file path=usr/include/ar.h
200 file path=usr/include/archives.h
201 file path=usr/include/arpa/ftp.h
202 file path=usr/include/arpa/inet.h
203 file path=usr/include/arpa/nameser.h
204 file path=usr/include/arpa/nameser_compat.h
205 file path=usr/include/arpa/telnet.h
206 file path=usr/include/arpa/tftp.h
207 $(i386_ONLY)file path=usr/include/asm/atomic.h
208 $(i386_ONLY)file path=usr/include/asm/bitmap.h
209 $(i386_ONLY)file path=usr/include/asm/byteorder.h
210 $(i386_ONLY)file path=usr/include/asm/clock.h
211 $(i386_ONLY)file path=usr/include/asm/cpu.h
212 $(i386_ONLY)file path=usr/include/asm/cpuid.h
213 $(sparc_ONLY)file path=usr/include/asm/flush.h
214 $(i386_ONLY)file path=usr/include/asm/htable.h
215 $(i386_ONLY)file path=usr/include/asm/mmu.h
216 file path=usr/include/asm/sunddi.h
217 file path=usr/include/asm/thread.h
218 file path=usr/include/assert.h
219 file path=usr/include/ast/align.h
220 file path=usr/include/ast/ast.h
221 file path=usr/include/ast/ast_botch.h
222 file path=usr/include/ast/ast_ccode.h
223 file path=usr/include/ast/ast_common.h
224 file path=usr/include/ast/ast_dir.h
225 file path=usr/include/ast/ast_dirent.h
226 file path=usr/include/ast/ast_fcctl.h
227 file path=usr/include/ast/ast_float.h
228 file path=usr/include/ast/ast_fs.h
229 file path=usr/include/ast/ast_getopt.h
230 file path=usr/include/ast/ast_iconv.h
231 file path=usr/include/ast/ast_lib.h
232 file path=usr/include/ast/ast_limits.h
233 file path=usr/include/ast/ast_map.h
234 file path=usr/include/ast/ast_mmap.h
235 file path=usr/include/ast/ast_mode.h
236 file path=usr/include/ast/ast_namval.h
237 file path=usr/include/ast/ast_ndbm.h
238 file path=usr/include/ast/ast_nl_types.h
239 file path=usr/include/ast/ast_param.h
240 file path=usr/include/ast/ast_standards.h
241 file path=usr/include/ast/ast_std.h
242 file path=usr/include/ast/ast_stdio.h
243 file path=usr/include/ast/ast_sys.h
244 file path=usr/include/ast/ast_time.h
245 file path=usr/include/ast/ast_tty.h
246 file path=usr/include/ast/ast_version.h
247 file path=usr/include/ast/ast_vfork.h
248 file path=usr/include/ast/ast_wait.h
249 file path=usr/include/ast/ast_wchar.h
250 file path=usr/include/ast/ast_windows.h
251 file path=usr/include/ast/bytesex.h
252 file path=usr/include/ast/ccode.h

```

```
253 file path=usr/include/ast/cdt.h
254 file path=usr/include/ast/cmd.h
255 file path=usr/include/ast/cmdext.h
256 file path=usr/include/ast/debug.h
257 file path=usr/include/ast/dirent.h
258 file path=usr/include/ast/dlldefs.h
259 file path=usr/include/ast/dt.h
260 file path=usr/include/ast/endian.h
261 file path=usr/include/ast/error.h
262 file path=usr/include/ast/find.h
263 file path=usr/include/ast/fnmatch.h
264 file path=usr/include/ast/fnv.h
265 file path=usr/include/ast/fs3d.h
266 file path=usr/include/ast/fts.h
267 file path=usr/include/ast/ftw.h
268 file path=usr/include/ast/ftwalk.h
269 file path=usr/include/ast/getopt.h
270 file path=usr/include/ast/glob.h
271 file path=usr/include/ast/hash.h
272 file path=usr/include/ast/hashkey.h
273 file path=usr/include/ast/hashpart.h
274 file path=usr/include/ast/history.h
275 file path=usr/include/ast/iconv.h
276 file path=usr/include/ast/ip6.h
277 file path=usr/include/ast/lc.h
278 file path=usr/include/ast/ls.h
279 file path=usr/include/ast/magic.h
280 file path=usr/include/ast/magicid.h
281 file path=usr/include/ast/mc.h
282 file path=usr/include/ast/mime.h
283 file path=usr/include/ast/mnt.h
284 file path=usr/include/ast/modecanon.h
285 file path=usr/include/ast/modex.h
286 file path=usr/include/ast/namval.h
287 file path=usr/include/ast/nl_types.h
288 file path=usr/include/ast/nval.h
289 file path=usr/include/ast/option.h
290 file path=usr/include/ast/preroot.h
291 file path=usr/include/ast/proc.h
292 file path=usr/include/ast/prototyped.h
293 file path=usr/include/ast/re_comp.h
294 file path=usr/include/ast/recfmt.h
295 file path=usr/include/ast/regex.h
296 file path=usr/include/ast/regexp.h
297 file path=usr/include/ast/sfdisc.h
298 file path=usr/include/ast/sfio.h
299 file path=usr/include/ast/sfio_s.h
300 file path=usr/include/ast/sfio_t.h
301 file path=usr/include/ast/shcmd.h
302 file path=usr/include/ast/shell.h
303 file path=usr/include/ast/sig.h
304 file path=usr/include/ast/stack.h
305 file path=usr/include/ast/stak.h
306 file path=usr/include/ast/stdio.h
307 file path=usr/include/ast/stk.h
308 file path=usr/include/ast/sum.h
309 file path=usr/include/ast/swap.h
310 file path=usr/include/ast/tar.h
311 file path=usr/include/ast/times.h
312 file path=usr/include/ast/tm.h
313 file path=usr/include/ast/tmx.h
314 file path=usr/include/ast/tok.h
315 file path=usr/include/ast/tv.h
316 file path=usr/include/ast/usage.h
317 file path=usr/include/ast/vdb.h
318 file path=usr/include/ast/vecargs.h
```

```
319 file path=usr/include/ast/vmalloc.h
320 file path=usr/include/ast/wait.h
321 file path=usr/include/ast/wchar.h
322 file path=usr/include/ast/wordexp.h
323 file path=usr/include/atomic.h
324 file path=usr/include/attr.h
325 file path=usr/include/auth_attr.h
326 file path=usr/include/bsm/adt.h
327 file path=usr/include/bsm/adt_event.h
328 file path=usr/include/bsm/audit.h
329 file path=usr/include/bsm/audit_kernel.h
330 file path=usr/include/bsm/audit_kevents.h
331 file path=usr/include/bsm/audit_record.h
332 file path=usr/include/bsm/audit_uevents.h
333 file path=usr/include/bsm/devices.h
334 file path=usr/include/bsm/libbsm.h
335 file path=usr/include/config_admin.h
336 file path=usr/include/cpio.h
337 file path=usr/include/crypt.h
338 file path=usr/include/cryptoutil.h
339 file path=usr/include/ctype.h
340 file path=usr/include/curses.h
341 file path=usr/include/dat/dat.h
342 file path=usr/include/dat/dat_error.h
343 file path=usr/include/dat/dat_platform_specific.h
344 file path=usr/include/dat/dat_redirection.h
345 file path=usr/include/dat/dat_registry.h
346 file path=usr/include/dat/dat_vendor_specific.h
347 file path=usr/include/dat/udat.h
348 file path=usr/include/dat/udat_config.h
349 file path=usr/include/dat/udat_redirection.h
350 file path=usr/include/dat/udat_vendor_specific.h
351 file path=usr/include/default.h
352 file path=usr/include/des/des.h
353 file path=usr/include/des/desdata.h
354 file path=usr/include/des/softdes.h
355 file path=usr/include/device_info.h
356 file path=usr/include/devid.h
357 file path=usr/include/devmgmt.h
358 file path=usr/include/devpoll.h
359 file path=usr/include/dial.h
360 file path=usr/include/dirent.h
361 file path=usr/include/dlfcn.h
362 file path=usr/include/door.h
363 file path=usr/include/elf.h
364 file path=usr/include/err.h
365 file path=usr/include/errno.h
366 file path=usr/include/eti.h
367 file path=usr/include/euc.h
368 file path=usr/include/exacct.h
369 file path=usr/include/exacct_impl.h
370 file path=usr/include/exec_attr.h
371 file path=usr/include/execinfo.h
372 file path=usr/include/fatal.h
373 file path=usr/include/fcntl.h
374 file path=usr/include/float.h
375 file path=usr/include/fmtmsg.h
376 file path=usr/include/fnmatch.h
377 file path=usr/include/form.h
378 file path=usr/include/ftw.h
379 file path=usr/include/gelf.h
380 file path=usr/include/getopt.h
381 file path=usr/include/getwidth.h
382 file path=usr/include/glob.h
383 file path=usr/include/grp.h
384 file path=usr/include/gssapi/gssapi.h
```

```

385 file path=usr/include/gssapi/gssapi_ext.h
386 file path=usr/include/hal/libhal-storage.h
387 file path=usr/include/hal/libhal.h
388 $(i386_ONLY)file path=usr/include/ia32/sys/asm_linkage.h
389 $(i386_ONLY)file path=usr/include/ia32/sys/kdi_regs.h
390 $(i386_ONLY)file path=usr/include/ia32/sys/machtypes.h
391 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
392 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
393 $(i386_ONLY)file path=usr/include/ia32/sys/psw.h
394 $(i386_ONLY)file path=usr/include/ia32/sys/pte.h
395 $(i386_ONLY)file path=usr/include/ia32/sys/reg.h
396 $(i386_ONLY)file path=usr/include/ia32/sys/stack.h
397 $(i386_ONLY)file path=usr/include/ia32/sys/trap.h
398 $(i386_ONLY)file path=usr/include/ia32/sys/traptrace.h
399 file path=usr/include/iconv.h
400 file path=usr/include/idmap.h
401 file path=usr/include/ieeefp.h
402 file path=usr/include/ifaddrs.h
403 file path=usr/include/inet/arp.h
404 file path=usr/include/inet/common.h
405 file path=usr/include/inet/ip.h
406 file path=usr/include/inet/ip6.h
407 file path=usr/include/inet/ip6_asp.h
408 file path=usr/include/inet/ip_arp.h
409 file path=usr/include/inet/ip_ftable.h
410 file path=usr/include/inet/ip_if.h
411 file path=usr/include/inet/ip_ire.h
412 file path=usr/include/inet/ip_multi.h
413 file path=usr/include/inet/ip_netinfo.h
414 file path=usr/include/inet/ip_rts.h
415 file path=usr/include/inet/ip_stack.h
416 file path=usr/include/inet/ipclassifier.h
417 file path=usr/include/inet/ipdrop.h
418 file path=usr/include/inet/ipnet.h
419 file path=usr/include/inet/ipp_common.h
420 file path=usr/include/inet/kssl/ksslapl.h
421 file path=usr/include/inet/led.h
422 file path=usr/include/inet/mi.h
423 file path=usr/include/inet/mib2.h
424 file path=usr/include/inet/nd.h
425 file path=usr/include/inet/optcom.h
426 file path=usr/include/inet/sctp_itf.h
427 file path=usr/include/inet/snmpcom.h
428 file path=usr/include/inet/tcp.h
429 file path=usr/include/inet/tcp_sack.h
430 file path=usr/include/inet/tcp_stack.h
431 file path=usr/include/inet/tcp_stats.h
432 file path=usr/include/inet/tunables.h
433 file path=usr/include/inet/wifi_ioctl.h
434 file path=usr/include/inttypes.h
435 file path=usr/include/ipmp.h
436 file path=usr/include/ipmp_admin.h
437 file path=usr/include/ipmp_mpathd.h
438 file path=usr/include/ipmp_query.h
439 file path=usr/include/ipp/ipgpc/ipgpc.h
440 file path=usr/include/ipp/ipp.h
441 file path=usr/include/ipp/ipp_config.h
442 file path=usr/include/ipp/ipp_impl.h
443 file path=usr/include/ipp/ippctl.h
444 file path=usr/include/iso/ctype_c99.h
445 file path=usr/include/iso/ctype_iso.h
446 file path=usr/include/iso/limits_iso.h
447 file path=usr/include/iso/locale_iso.h
448 file path=usr/include/iso/setjmp_iso.h
449 file path=usr/include/iso/signal_iso.h
450 file path=usr/include/iso/stdarg_c99.h

```

```

451 file path=usr/include/iso/stdarg_iso.h
452 file path=usr/include/iso/stddef_iso.h
453 file path=usr/include/iso/stdio_c99.h
454 file path=usr/include/iso/stdio_iso.h
455 file path=usr/include/iso/stdlib_c99.h
456 file path=usr/include/iso/stdlib_iso.h
457 file path=usr/include/iso/string_iso.h
458 file path=usr/include/iso/time_iso.h
459 file path=usr/include/iso/wchar_c99.h
460 file path=usr/include/iso/wchar_iso.h
461 file path=usr/include/iso/wctype_c99.h
462 file path=usr/include/iso/wctype_iso.h
463 file path=usr/include/iso646.h
464 file path=usr/include/kerberos5/com_err.h
465 file path=usr/include/kerberos5/krb5.h
466 file path=usr/include/kerberos5/mit-sipb-copyright.h
467 file path=usr/include/kerberos5/mit_copyright.h
468 file path=usr/include/klpd.h
469 file path=usr/include/kmfapi.h
470 file path=usr/include/kmftypes.h
471 file path=usr/include/kstat.h
472 file path=usr/include/kvm.h
473 file path=usr/include/langinfo.h
474 file path=usr/include/lastlog.h
475 file path=usr/include/lber.h
476 file path=usr/include/ldap.h
477 file path=usr/include/libcontract.h
478 file path=usr/include/libctf.h
479 file path=usr/include/libdevice.h
480 file path=usr/include/libdevinfo.h
481 file path=usr/include/libdladm.h
482 file path=usr/include/libdlbridge.h
483 file path=usr/include/libdlib.h
484 file path=usr/include/libdlink.h
485 file path=usr/include/libdlpi.h
486 file path=usr/include/libdlvlan.h
487 file path=usr/include/libelf.h
488 $(i386_ONLY)file path=usr/include/libfdisk.h
489 file path=usr/include/libfstyp.h
490 file path=usr/include/libfstyp_module.h
491 file path=usr/include/libgen.h
492 file path=usr/include/libgrubmgmt.h
493 file path=usr/include/libintl.h
494 file path=usr/include/libipmi.h
495 file path=usr/include/libipp.h
496 file path=usr/include/libnvpair.h
497 file path=usr/include/libnwam.h
498 file path=usr/include/libpolkit/libpolkit.h
499 file path=usr/include/librcm.h
500 file path=usr/include/libscf.h
501 file path=usr/include/libscf_priv.h
502 file path=usr/include/libshare.h
503 file path=usr/include/libsvm.h
504 file path=usr/include/libsysevent.h
505 file path=usr/include/libsysevent_impl.h
506 file path=usr/include/libtsnet.h
507 $(sparc_ONLY)file path=usr/include/libv12n.h
508 file path=usr/include/libw.h
509 file path=usr/include/libzfs.h
510 file path=usr/include/libzfs_core.h
511 #endif /* ! codereview */
512 file path=usr/include/libzoneinfo.h
513 file path=usr/include/limits.h
514 file path=usr/include/linenum.h
515 file path=usr/include/link.h
516 file path=usr/include/listen.h

```

```
517 file path=usr/include/locale.h
518 file path=usr/include/macros.h
519 file path=usr/include/maillock.h
520 file path=usr/include/malloc.h
521 file path=usr/include/md4.h
522 file path=usr/include/md5.h
523 file path=usr/include/mdiox.h
524 file path=usr/include/mdm_changelog.h
525 file path=usr/include/memory.h
526 file path=usr/include/menu.h
527 file path=usr/include/meta.h
528 file path=usr/include/meta_basic.h
529 file path=usr/include/meta_runtime.h
530 file path=usr/include/metacl.h
531 file path=usr/include/metad.h
532 file path=usr/include/metadyn.h
533 file path=usr/include/metamed.h
534 file path=usr/include/metamhd.h
535 file path=usr/include/mhdx.h
536 file path=usr/include/mon.h
537 file path=usr/include/monetary.h
538 file path=usr/include/mp.h
539 file path=usr/include/mqueue.h
540 file path=usr/include/mtmalloc.h
541 file path=usr/include/nan.h
542 file path=usr/include/nbdbm.h
543 file path=usr/include/ndpd.h
544 file path=usr/include/net/af.h
545 file path=usr/include/net/bridge.h
546 file path=usr/include/net/if.h
547 file path=usr/include/net/if_arp.h
548 file path=usr/include/net/if_dl.h
549 file path=usr/include/net/if_types.h
550 file path=usr/include/net/pfkeyv2.h
551 file path=usr/include/net/pfpolicy.h
552 file path=usr/include/net/ppp-comp.h
553 file path=usr/include/net/ppp-defs.h
554 file path=usr/include/net/pppio.h
555 file path=usr/include/net/radix.h
556 file path=usr/include/net/route.h
557 file path=usr/include/net/trill.h
558 file path=usr/include/net/vjcompress.h
559 file path=usr/include/netconfig.h
560 file path=usr/include/netdb.h
561 file path=usr/include/netdir.h
562 file path=usr/include/netinet/arp.h
563 file path=usr/include/netinet/dhcp.h
564 file path=usr/include/netinet/dhcp6.h
565 file path=usr/include/netinet/icmp6.h
566 file path=usr/include/netinet/icmp_var.h
567 file path=usr/include/netinet/if_ether.h
568 file path=usr/include/netinet/igmp.h
569 file path=usr/include/netinet/igmp_var.h
570 file path=usr/include/netinet/in.h
571 file path=usr/include/netinet/in_pcb.h
572 file path=usr/include/netinet/in_system.h
573 file path=usr/include/netinet/in_var.h
574 file path=usr/include/netinet/ip.h
575 file path=usr/include/netinet/ip6.h
576 file path=usr/include/netinet/ip_icmp.h
577 file path=usr/include/netinet/ip_mroute.h
578 file path=usr/include/netinet/ip_var.h
579 file path=usr/include/netinet/pim.h
580 file path=usr/include/netinet/sctp.h
581 file path=usr/include/netinet/tcp.h
582 file path=usr/include/netinet/tcp_debug.h
```

```
583 file path=usr/include/netinet/tcp_fsm.h
584 file path=usr/include/netinet/tcp_seq.h
585 file path=usr/include/netinet/tcp_timer.h
586 file path=usr/include/netinet/tcp_var.h
587 file path=usr/include/netinet/tcpip.h
588 file path=usr/include/netinet/udp.h
589 file path=usr/include/netinet/udp_var.h
590 file path=usr/include/netinet/vrrp.h
591 file path=usr/include/nfs/auth.h
592 file path=usr/include/nfs/export.h
593 file path=usr/include/nfs/lm.h
594 file path=usr/include/nfs/mapid.h
595 file path=usr/include/nfs/mount.h
596 file path=usr/include/nfs/nfs.h
597 file path=usr/include/nfs/nfs4.h
598 file path=usr/include/nfs/nfs4_attr.h
599 file path=usr/include/nfs/nfs4_clnt.h
600 file path=usr/include/nfs/nfs4_db_impl.h
601 file path=usr/include/nfs/nfs4_idmap_impl.h
602 file path=usr/include/nfs/nfs4_kprot.h
603 file path=usr/include/nfs/nfs_acl.h
604 file path=usr/include/nfs/nfs_clnt.h
605 file path=usr/include/nfs/nfs_cmd.h
606 file path=usr/include/nfs/nfs_log.h
607 file path=usr/include/nfs/nfs_sec.h
608 file path=usr/include/nfs/nfsid_map.h
609 file path=usr/include/nfs/nfssys.h
610 file path=usr/include/nfs/rnode.h
611 file path=usr/include/nfs/rnode4.h
612 file path=usr/include/nl_types.h
613 file path=usr/include/nlist.h
614 file path=usr/include/note.h
615 file path=usr/include/nss_common.h
616 file path=usr/include/nss_dbdefs.h
617 file path=usr/include/nss_netdir.h
618 file path=usr/include/nsswitch.h
619 file path=usr/include/panel.h
620 file path=usr/include/paths.h
621 file path=usr/include/pcsample.h
622 file path=usr/include/pfmt.h
623 file path=usr/include/pkgdev.h
624 file path=usr/include/pkginfo.h
625 file path=usr/include/pkglocs.h
626 file path=usr/include/pkgstrct.h
627 file path=usr/include/pkgtrans.h
628 file path=usr/include/poll.h
629 file path=usr/include/port.h
630 file path=usr/include/priv.h
631 file path=usr/include/proc_service.h
632 file path=usr/include/procfs.h
633 file path=usr/include/prof.h
634 file path=usr/include/prof_attr.h
635 file path=usr/include/project.h
636 file path=usr/include/protocols/dumppstore.h
637 file path=usr/include/protocols/routed.h
638 file path=usr/include/protocols/rwhod.h
639 file path=usr/include/protocols/timed.h
640 file path=usr/include/pthread.h
641 file path=usr/include/pw.h
642 file path=usr/include/pwd.h
643 file path=usr/include/rcm_module.h
644 file path=usr/include/rctl.h
645 file path=usr/include/re_comp.h
646 file path=usr/include/regex.h
647 file path=usr/include/regexp.h
648 file path=usr/include/regexpr.h
```

```

649 file path=usr/include/resolv.h
650 file path=usr/include/rje.h
651 file path=usr/include/rp_plugin.h
652 file path=usr/include/rpc/auth.h
653 file path=usr/include/rpc/auth_des.h
654 file path=usr/include/rpc/auth_sys.h
655 file path=usr/include/rpc/auth_unix.h
656 file path=usr/include/rpc/bootparam.h
657 file path=usr/include/rpc/clnt.h
658 file path=usr/include/rpc/clnt_soc.h
659 file path=usr/include/rpc/clnt_stat.h
660 file path=usr/include/rpc/des_crypt.h
661 $(sparc_ONLY)file path=usr/include/rpc/ib.h
662 file path=usr/include/rpc/key_prot.h
663 file path=usr/include/rpc/nettype.h
664 file path=usr/include/rpc/pmap_clnt.h
665 file path=usr/include/rpc/pmap_prot.h
666 file path=usr/include/rpc/pmap_prot.x
667 file path=usr/include/rpc/pmap_rmt.h
668 file path=usr/include/rpc/raw.h
669 file path=usr/include/rpc/rpc.h
670 file path=usr/include/rpc/rpc_com.h
671 file path=usr/include/rpc/rpc_msg.h
672 file path=usr/include/rpc/rpc_rdma.h
673 file path=usr/include/rpc/rpc_sztypes.h
674 file path=usr/include/rpc/rpcb_clnt.h
675 file path=usr/include/rpc/rpcb_prot.h
676 file path=usr/include/rpc/rpcb_prot.x
677 file path=usr/include/rpc/rpcent.h
678 file path=usr/include/rpc/rpcsec_gss.h
679 file path=usr/include/rpc/rpcsys.h
680 file path=usr/include/rpc/svc.h
681 file path=usr/include/rpc/svc_auth.h
682 file path=usr/include/rpc/svc_mt.h
683 file path=usr/include/rpc/svc_soc.h
684 file path=usr/include/rpc/types.h
685 file path=usr/include/rpc/xdr.h
686 file path=usr/include/rpcsvc/autofs_prot.h
687 file path=usr/include/rpcsvc/autofs_prot.x
688 file path=usr/include/rpcsvc/bootparam.h
689 file path=usr/include/rpcsvc/bootparam_prot.h
690 file path=usr/include/rpcsvc/bootparam_prot.x
691 file path=usr/include/rpcsvc/dbm.h
692 file path=usr/include/rpcsvc/key_prot.x
693 file path=usr/include/rpcsvc/mount.h
694 file path=usr/include/rpcsvc/mount.x
695 file path=usr/include/rpcsvc/nfs4_prot.h
696 file path=usr/include/rpcsvc/nfs4_prot.x
697 file path=usr/include/rpcsvc/nfs_acl.h
698 file path=usr/include/rpcsvc/nfs_acl.x
699 file path=usr/include/rpcsvc/nfs_prot.h
700 file path=usr/include/rpcsvc/nfs_prot.x
701 file path=usr/include/rpcsvc/nis.h
702 file path=usr/include/rpcsvc/nis.x
703 file path=usr/include/rpcsvc/nis_db.h
704 file path=usr/include/rpcsvc/nis_object.x
705 file path=usr/include/rpcsvc/nislib.h
706 file path=usr/include/rpcsvc/nlm_prot.h
707 file path=usr/include/rpcsvc/nlm_prot.x
708 file path=usr/include/rpcsvc/nsm_addr.h
709 file path=usr/include/rpcsvc/nsm_addr.x
710 file path=usr/include/rpcsvc/rex.h
711 file path=usr/include/rpcsvc/rex.x
712 file path=usr/include/rpcsvc/rpc_sztypes.h
713 file path=usr/include/rpcsvc/rpc_sztypes.x
714 file path=usr/include/rpcsvc/rquota.h

```

```

715 file path=usr/include/rpcsvc/rquota.x
716 file path=usr/include/rpcsvc/rstat.h
717 file path=usr/include/rpcsvc/rstat.x
718 file path=usr/include/rpcsvc/rusers.h
719 file path=usr/include/rpcsvc/rusers.x
720 file path=usr/include/rpcsvc/rwall.h
721 file path=usr/include/rpcsvc/rwall.x
722 file path=usr/include/rpcsvc/sm_inter.h
723 file path=usr/include/rpcsvc/sm_inter.x
724 file path=usr/include/rpcsvc/spray.h
725 file path=usr/include/rpcsvc/spray.x
726 file path=usr/include/rpcsvc/ufs_prot.h
727 file path=usr/include/rpcsvc/ufs_prot.x
728 file path=usr/include/rpcsvc/yp.x
729 file path=usr/include/rpcsvc/yp_prot.h
730 file path=usr/include/rpcsvc/ypclnt.h
731 file path=usr/include/rpcsvc/yppasswd.h
732 file path=usr/include/rpcsvc/ypupd.h
733 file path=usr/include/rsmapi.h
734 file path=usr/include/rtld_db.h
735 file path=usr/include/sac.h
736 file path=usr/include/sasl/prop.h
737 file path=usr/include/sasl/sasl.h
738 file path=usr/include/sasl/saslplug.h
739 file path=usr/include/sasl/saslutil.h
740 file path=usr/include/sched.h
741 file path=usr/include/schedctl.h
742 file path=usr/include/scsi/libscsi.h
743 file path=usr/include/scsi/libses.h
744 file path=usr/include/scsi/libses_plugin.h
745 file path=usr/include/scsi/libsmpt.h
746 file path=usr/include/scsi/libsmpt_plugin.h
747 file path=usr/include/scsi/plugins/ses/framework/libses.h
748 file path=usr/include/scsi/plugins/ses/framework/ses2.h
749 file path=usr/include/scsi/plugins/ses/framework/ses2_impl.h
750 file path=usr/include/scsi/plugins/ses/vendor/sun.h
751 file path=usr/include/sdp.h
752 file path=usr/include/search.h
753 file path=usr/include/secdb.h
754 file path=usr/include/security/auditd.h
755 file path=usr/include/security/cryptoki.h
756 file path=usr/include/security/pam_appl.h
757 file path=usr/include/security/pam_modules.h
758 file path=usr/include/security/pkcs11.h
759 file path=usr/include/security/pkcs11f.h
760 file path=usr/include/security/pkcs11t.h
761 file path=usr/include/semaphore.h
762 file path=usr/include/setjmp.h
763 file path=usr/include/sgtty.h
764 file path=usr/include/sha1.h
765 file path=usr/include/sha2.h
766 file path=usr/include/shadow.h
767 file path=usr/include/sharefs/share.h
768 file path=usr/include/sharefs/sharefs.h
769 file path=usr/include/sharefs/sharetab.h
770 file path=usr/include/signinfo.h
771 file path=usr/include/signal.h
772 file path=usr/include/sip.h
773 file path=usr/include/smbios.h
774 file path=usr/include/spawn.h
775 $(i386_ONLY)file path=usr/include/stack_unwind.h
776 file path=usr/include/stdarg.h
777 file path=usr/include/stdbool.h
778 file path=usr/include/stddef.h
779 file path=usr/include/stdint.h
780 file path=usr/include/stdio.h

```

```

781 file path=usr/include/stdio_ext.h
782 file path=usr/include/stdio_impl.h
783 file path=usr/include/stdio_tag.h
784 file path=usr/include/stdlib.h
785 file path=usr/include/storclass.h
786 file path=usr/include/string.h
787 file path=usr/include/strings.h
788 file path=usr/include/stropts.h
789 file path=usr/include/syms.h
790 file path=usr/include/synch.h
791 file path=usr/include/sys/acct.h
792 file path=usr/include/sys/acctctl.h
793 file path=usr/include/sys/acl.h
794 file path=usr/include/sys/acl_impl.h
795 file path=usr/include/sys/acpi_drv.h
796 file path=usr/include/sys/aio.h
797 file path=usr/include/sys/aio_impl.h
798 file path=usr/include/sys/aio_req.h
799 file path=usr/include/sys/aioch.h
800 file path=usr/include/sys/archsystem.h
801 file path=usr/include/sys/ascii.h
802 file path=usr/include/sys/asm_linkage.h
803 file path=usr/include/sys/asynch.h
804 file path=usr/include/sys/atomic.h
805 file path=usr/include/sys/attr.h
806 file path=usr/include/sys/autocconf.h
807 file path=usr/include/sys/auxv.h
808 file path=usr/include/sys/auxv_386.h
809 file path=usr/include/sys/auxv_SPARC.h
810 file path=usr/include/sys/av/iec61883.h
811 file path=usr/include/sys/avintr.h
812 file path=usr/include/sys/avl.h
813 file path=usr/include/sys/avl_impl.h
814 file path=usr/include/sys/bitmap.h
815 file path=usr/include/sys/bitset.h
816 file path=usr/include/sys/bl.h
817 file path=usr/include/sys/blkdev.h
818 file path=usr/include/sys/bmc_intf.h
819 file path=usr/include/sys/bofi.h
820 file path=usr/include/sys/bofi_impl.h
821 file path=usr/include/sys/bootconf.h
822 $(i386_ONLY)file path=usr/include/sys/bootregs.h
823 file path=usr/include/sys/bootstat.h
824 $(i386_ONLY)file path=usr/include/sys/bootsvcs.h
825 file path=usr/include/sys/bpp_io.h
826 file path=usr/include/sys/brand.h
827 file path=usr/include/sys/buf.h
828 file path=usr/include/sys/bufmod.h
829 file path=usr/include/sys/bustypes.h
830 file path=usr/include/sys/byteorder.h
831 file path=usr/include/sys/callb.h
832 file path=usr/include/sys/callo.h
833 file path=usr/include/sys/cap_util.h
834 file path=usr/include/sys/ccompile.h
835 file path=usr/include/sys/cdio.h
836 file path=usr/include/sys/cis.h
837 file path=usr/include/sys/cis_handlers.h
838 file path=usr/include/sys/cis_protos.h
839 file path=usr/include/sys/cladm.h
840 file path=usr/include/sys/class.h
841 file path=usr/include/sys/clconf.h
842 file path=usr/include/sys/cmlb.h
843 file path=usr/include/sys/cmn_err.h
844 $(sparc_ONLY)file path=usr/include/sys/cmpregs.h
845 file path=usr/include/sys/compress.h
846 file path=usr/include/sys/condvar.h

```

```

847 file path=usr/include/sys/condvar_impl.h
848 file path=usr/include/sys/conf.h
849 file path=usr/include/sys/consdev.h
850 file path=usr/include/sys/console.h
851 file path=usr/include/sys/consplat.h
852 file path=usr/include/sys/contract.h
853 file path=usr/include/sys/contract/device.h
854 file path=usr/include/sys/contract/device_impl.h
855 file path=usr/include/sys/contract/process.h
856 file path=usr/include/sys/contract/process_impl.h
857 file path=usr/include/sys/contract_impl.h
858 $(i386_ONLY)file path=usr/include/sys/controlregs.h
859 file path=usr/include/sys/copyops.h
860 file path=usr/include/sys/core.h
861 file path=usr/include/sys/corectl.h
862 file path=usr/include/sys/cpc_impl.h
863 file path=usr/include/sys/cpc_pcbe.h
864 file path=usr/include/sys/cpr.h
865 file path=usr/include/sys/cpu.h
866 file path=usr/include/sys/cpucaps.h
867 file path=usr/include/sys/cpucaps_impl.h
868 file path=usr/include/sys/cpupart.h
869 file path=usr/include/sys/cpuvar.h
870 file path=usr/include/sys/crc32.h
871 file path=usr/include/sys/cred.h
872 file path=usr/include/sys/cred_impl.h
873 file path=usr/include/sys/crtctl.h
874 file path=usr/include/sys/crypto/api.h
875 file path=usr/include/sys/crypto/common.h
876 file path=usr/include/sys/crypto/ioctl.h
877 file path=usr/include/sys/crypto/ioctladmin.h
878 file path=usr/include/sys/crypto/spi.h
879 file path=usr/include/sys/cs.h
880 file path=usr/include/sys/cs_priv.h
881 file path=usr/include/sys/cs_strings.h
882 file path=usr/include/sys/cs_stubs.h
883 file path=usr/include/sys/cs_types.h
884 file path=usr/include/sys/csiiioctl.h
885 file path=usr/include/sys/ctf.h
886 file path=usr/include/sys/ctf_api.h
887 file path=usr/include/sys/ctfs.h
888 file path=usr/include/sys/ctfs_impl.h
889 file path=usr/include/sys/ctype.h
890 file path=usr/include/sys/cyclic.h
891 file path=usr/include/sys/cyclic_impl.h
892 file path=usr/include/sys/dacf.h
893 file path=usr/include/sys/dacf_impl.h
894 file path=usr/include/sys/damap.h
895 file path=usr/include/sys/damap_impl.h
896 file path=usr/include/sys/dc_ki.h
897 file path=usr/include/sys/ddi.h
898 file path=usr/include/sys/ddi_hp.h
899 file path=usr/include/sys/ddi_hp_impl.h
900 file path=usr/include/sys/ddi_impldefs.h
901 file path=usr/include/sys/ddi_implfuncs.h
902 file path=usr/include/sys/ddi_intr.h
903 file path=usr/include/sys/ddi_intr_impl.h
904 file path=usr/include/sys/ddi_isa.h
905 file path=usr/include/sys/ddi_obsolete.h
906 file path=usr/include/sys/ddi_timer.h
907 file path=usr/include/sys/ddidevmap.h
908 file path=usr/include/sys/ddidmareq.h
909 file path=usr/include/sys/ddifm.h
910 file path=usr/include/sys/ddifm_impl.h
911 file path=usr/include/sys/ddimapreq.h
912 file path=usr/include/sys/ddipropdefs.h

```

```
913 file path=usr/include/sys/dditypes.h
914 file path=usr/include/sys/debug.h
915 $(i386_ONLY)file path=usr/include/sys/debugreg.h
916 file path=usr/include/sys/des.h
917 file path=usr/include/sys/devcache.h
918 file path=usr/include/sys/devcache_impl.h
919 file path=usr/include/sys/devctl.h
920 file path=usr/include/sys/devfm.h
921 file path=usr/include/sys/devid_cache.h
922 file path=usr/include/sys/devinfo_impl.h
923 file path=usr/include/sys/devops.h
924 file path=usr/include/sys/devpolicy.h
925 file path=usr/include/sys/devpoll.h
926 file path=usr/include/sys/dirent.h
927 file path=usr/include/sys/disp.h
928 file path=usr/include/sys/dkbad.h
929 file path=usr/include/sys/dkio.h
930 file path=usr/include/sys/dklabel.h
931 $(sparc_ONLY)file path=usr/include/sys/dkmpio.h
932 $(i386_ONLY)file path=usr/include/sys/dktp/altctr.h
933 $(i386_ONLY)file path=usr/include/sys/dktp/cmpkt.h
934 file path=usr/include/sys/dktp/dadkio.h
935 file path=usr/include/sys/dktp/fdisk.h
936 file path=usr/include/sys/dl.h
937 file path=usr/include/sys/dld.h
938 file path=usr/include/sys/dlpi.h
939 file path=usr/include/sys/dls_mgmt.h
940 $(i386_ONLY)file path=usr/include/sys/dma_engine.h
941 file path=usr/include/sys/dma_i8237A.h
942 file path=usr/include/sys/dnlc.h
943 file path=usr/include/sys/door.h
944 file path=usr/include/sys/door_data.h
945 file path=usr/include/sys/door_impl.h
946 file path=usr/include/sys/dumphdr.h
947 file path=usr/include/sys/ecppio.h
948 file path=usr/include/sys/ecppreg.h
949 file path=usr/include/sys/ecppsys.h
950 file path=usr/include/sys/ecppvar.h
951 file path=usr/include/sys/efi_partition.h
952 file path=usr/include/sys/elf.h
953 file path=usr/include/sys/elf_386.h
954 file path=usr/include/sys/elf_SPARC.h
955 file path=usr/include/sys/elf_amd64.h
956 file path=usr/include/sys/elf_notes.h
957 file path=usr/include/sys/elftypes.h
958 file path=usr/include/sys/epm.h
959 file path=usr/include/sys/errno.h
960 file path=usr/include/sys/errorq.h
961 file path=usr/include/sys/errorq_impl.h
962 file path=usr/include/sys/esunddi.h
963 file path=usr/include/sys/ethernet.h
964 file path=usr/include/sys/euc.h
965 file path=usr/include/sys/eucioct1.h
966 file path=usr/include/sys/exacct.h
967 file path=usr/include/sys/exacct_catalog.h
968 file path=usr/include/sys/exacct_impl.h
969 file path=usr/include/sys/exec.h
970 file path=usr/include/sys/exechdr.h
971 file path=usr/include/sys/fault.h
972 file path=usr/include/sys/fbio.h
973 file path=usr/include/sys/fbuf.h
974 file path=usr/include/sys/fc4/fc.h
975 file path=usr/include/sys/fc4/fc_transport.h
976 file path=usr/include/sys/fc4/fcal.h
977 file path=usr/include/sys/fc4/fcal_linkapp.h
978 file path=usr/include/sys/fc4/fcal_transport.h
```

```
979 file path=usr/include/sys/fc4/fcio.h
980 file path=usr/include/sys/fc4/fcp.h
981 file path=usr/include/sys/fc4/linkapp.h
982 file path=usr/include/sys/fcntl.h
983 file path=usr/include/sys/fdbuffer.h
984 file path=usr/include/sys/fdio.h
985 $(sparc_ONLY)file path=usr/include/sys/fdreg.h
986 $(sparc_ONLY)file path=usr/include/sys/fdvar.h
987 file path=usr/include/sys/feature_tests.h
988 file path=usr/include/sys/fem.h
989 file path=usr/include/sys/file.h
990 file path=usr/include/sys/filio.h
991 file path=usr/include/sys/flock.h
992 file path=usr/include/sys/flock_impl.h
993 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/SPARC64-VI.h
994 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-II.h
995 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-III.h
996 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-T1.h
997 file path=usr/include/sys/fm/fs/zfs.h
998 file path=usr/include/sys/fm/io/ddi.h
999 file path=usr/include/sys/fm/io/disk.h
1000 file path=usr/include/sys/fm/io/opl_mc_fm.h
1001 file path=usr/include/sys/fm/io/pci.h
1002 file path=usr/include/sys/fm/io/scsi.h
1003 file path=usr/include/sys/fm/io/sun4upci.h
1004 file path=usr/include/sys/fm/protocol.h
1005 file path=usr/include/sys/fm/util.h
1006 file path=usr/include/sys/fork.h
1007 $(i386_ONLY)file path=usr/include/sys/fp.h
1008 $(sparc_ONLY)file path=usr/include/sys/fpu/fpu_simulator.h
1009 $(sparc_ONLY)file path=usr/include/sys/fpu/fpusystem.h
1010 $(sparc_ONLY)file path=usr/include/sys/fpu/globals.h
1011 $(sparc_ONLY)file path=usr/include/sys/fpu/ieee.h
1012 file path=usr/include/sys/frame.h
1013 file path=usr/include/sys/fs/autofs.h
1014 file path=usr/include/sys/fs/cacheofs_dir.h
1015 file path=usr/include/sys/fs/cacheofs_dlog.h
1016 file path=usr/include/sys/fs/cacheofs_filegrp.h
1017 file path=usr/include/sys/fs/cacheofs_fs.h
1018 file path=usr/include/sys/fs/cacheofs_fscache.h
1019 file path=usr/include/sys/fs/cacheofs_ioctl.h
1020 file path=usr/include/sys/fs/cacheofs_log.h
1021 file path=usr/include/sys/fs/decomp.h
1022 file path=usr/include/sys/fs/dv_node.h
1023 file path=usr/include/sys/fs/fifonode.h
1024 file path=usr/include/sys/fs/hsfs_isospec.h
1025 file path=usr/include/sys/fs/hsfs_node.h
1026 file path=usr/include/sys/fs/hsfs_rrip.h
1027 file path=usr/include/sys/fs/hsfs_spec.h
1028 file path=usr/include/sys/fs/hsfs_susp.h
1029 file path=usr/include/sys/fs/lofs_info.h
1030 file path=usr/include/sys/fs/lofs_node.h
1031 file path=usr/include/sys/fs/mntdata.h
1032 file path=usr/include/sys/fs/namenode.h
1033 file path=usr/include/sys/fs/pc_dir.h
1034 file path=usr/include/sys/fs/pc_fs.h
1035 file path=usr/include/sys/fs/pc_label.h
1036 file path=usr/include/sys/fs/pc_node.h
1037 file path=usr/include/sys/fs/pxfs_ki.h
1038 file path=usr/include/sys/fs/sdev_impl.h
1039 file path=usr/include/sys/fs/snode.h
1040 file path=usr/include/sys/fs/swapnode.h
1041 file path=usr/include/sys/fs/tmp.h
1042 file path=usr/include/sys/fs/tmpnode.h
1043 file path=usr/include/sys/fs/udf_inode.h
1044 file path=usr/include/sys/fs/udf_volume.h
```



```

1045 file path=usr/include/sys/fs/ufs_acl.h
1046 file path=usr/include/sys/fs/ufs_bio.h
1047 file path=usr/include/sys/fs/ufs_filio.h
1048 file path=usr/include/sys/fs/ufs_fs.h
1049 file path=usr/include/sys/fs/ufs_fsdirent.h
1050 file path=usr/include/sys/fs/ufs_inode.h
1051 file path=usr/include/sys/fs/ufs_lockfs.h
1052 file path=usr/include/sys/fs/ufs_log.h
1053 file path=usr/include/sys/fs/ufs_mount.h
1054 file path=usr/include/sys/fs/ufs_panic.h
1055 file path=usr/include/sys/fs/ufs_prot.h
1056 file path=usr/include/sys/fs/ufs_quota.h
1057 file path=usr/include/sys/fs/ufs_snap.h
1058 file path=usr/include/sys/fs/ufs_trans.h
1059 file path=usr/include/sys/fs/zfs.h
1060 file path=usr/include/sys/fs_reparse.h
1061 file path=usr/include/sys/fs_subr.h
1062 file path=usr/include/sys/fsid.h
1063 $(sparc_ONLY)file path=usr/include/sys/fsr.h
1064 file path=usr/include/sys/fss.h
1065 file path=usr/include/sys/fssnap.h
1066 file path=usr/include/sys/fssnap_if.h
1067 file path=usr/include/sys/fsspriocntl.h
1068 file path=usr/include/sys/fstyp.h
1069 file path=usr/include/sys/fttrace.h
1070 file path=usr/include/sys/fx.h
1071 file path=usr/include/sys/fxpriocntl.h
1072 file path=usr/include/sys/gfs.h
1073 file path=usr/include/sys/gld.h
1074 file path=usr/include/sys/gldpriv.h
1075 file path=usr/include/sys/group.h
1076 file path=usr/include/sys/hdio.h
1077 file path=usr/include/sys/hook.h
1078 file path=usr/include/sys/hook_event.h
1079 file path=usr/include/sys/hook_impl.h
1080 file path=usr/include/sys/hotplug/hpcsvc.h
1081 file path=usr/include/sys/hotplug/hpctrl.h
1082 file path=usr/include/sys/hotplug/pci/pcicfg.h
1083 file path=usr/include/sys/hotplug/pci/pcihp.h
1084 file path=usr/include/sys/hwconf.h
1085 $(i386_ONLY)file path=usr/include/sys/hypervisor.h
1086 $(i386_ONLY)file path=usr/include/sys/i8272A.h
1087 file path=usr/include/sys/ia.h
1088 file path=usr/include/sys/iapriocntl.h
1089 file path=usr/include/sys/ib/adapters/hermon/hermon_ioctl.h
1090 file path=usr/include/sys/ib/adapters/mlnx_umap.h
1091 file path=usr/include/sys/ib/adapters/tavor/tavor_ioctl.h
1092 file path=usr/include/sys/ib/clients/ibd/ibd.h
1093 file path=usr/include/sys/ib/clients/of/ofa_solaris.h
1094 file path=usr/include/sys/ib/clients/of/ofed_kernel.h
1095 file path=usr/include/sys/ib/clients/of/rdma/ib_addr.h
1096 file path=usr/include/sys/ib/clients/of/rdma/ib_user_mad.h
1097 file path=usr/include/sys/ib/clients/of/rdma/ib_user_sa.h
1098 file path=usr/include/sys/ib/clients/of/rdma/ib_user_verbs.h
1099 file path=usr/include/sys/ib/clients/of/rdma/ib_verbs.h
1100 file path=usr/include/sys/ib/clients/of/rdma/rdma_cm.h
1101 file path=usr/include/sys/ib/clients/of/rdma/rdma_user_cm.h
1102 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_cma.h
1103 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ib_cma.h
1104 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_kverb_impl.h
1105 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ofs_common.h
1106 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_rdma_user_cm.h
1107 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_ucma.h
1108 file path=usr/include/sys/ib/clients/of/sol_umad/sol_umad.h
1109 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs.h
1110 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs2ucma.h

```

```

1111 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_comp.h
1112 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_event.h
1113 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_hca.h
1114 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_qp.h
1115 file path=usr/include/sys/ib/ib_pkt_hdrs.h
1116 file path=usr/include/sys/ib/ib_types.h
1117 file path=usr/include/sys/ib/ibnexus/ibnexus_devctl.h
1118 file path=usr/include/sys/ib/ibtl/ibci.h
1119 file path=usr/include/sys/ib/ibtl/ibti.h
1120 file path=usr/include/sys/ib/ibtl/ibtl_cm.h
1121 file path=usr/include/sys/ib/ibtl/ibtl_common.h
1122 file path=usr/include/sys/ib/ibtl/ibtl_ci_types.h
1123 file path=usr/include/sys/ib/ibtl/ibtl_status.h
1124 file path=usr/include/sys/ib/ibtl/ibtl_types.h
1125 file path=usr/include/sys/ib/ibtl/ibvti.h
1126 file path=usr/include/sys/ib/ibtl/impl/ibtl_util.h
1127 file path=usr/include/sys/ib/mgt/ib_dm_attr.h
1128 file path=usr/include/sys/ib/mgt/ib_mad.h
1129 file path=usr/include/sys/ib/mgt/ibmf/ibmf.h
1130 file path=usr/include/sys/ib/mgt/ibmf/ibmf_msg.h
1131 file path=usr/include/sys/ib/mgt/ibmf/ibmf_saa.h
1132 file path=usr/include/sys/ib/mgt/ibmf/ibmf_utils.h
1133 file path=usr/include/sys/ib/mgt/sa_recs.h
1134 file path=usr/include/sys/ib/mgt/sm_attr.h
1135 file path=usr/include/sys/ibpart.h
1136 file path=usr/include/sys/id32.h
1137 file path=usr/include/sys/id_space.h
1138 file path=usr/include/sys/idmap.h
1139 file path=usr/include/sys/inline.h
1140 file path=usr/include/sys/instance.h
1141 file path=usr/include/sys/int_const.h
1142 file path=usr/include/sys/int_fmtio.h
1143 file path=usr/include/sys/int_limits.h
1144 file path=usr/include/sys/int_types.h
1145 file path=usr/include/sys/inttypes.h
1146 file path=usr/include/sys/ioccom.h
1147 file path=usr/include/sys/ioctl.h
1148 $(i386_ONLY)file path=usr/include/sys/iommu/lib.h
1149 file path=usr/include/sys/ipc.h
1150 file path=usr/include/sys/ipc_impl.h
1151 file path=usr/include/sys/ipc_rctl.h
1152 file path=usr/include/sys/isa_defs.h
1153 file path=usr/include/sys/iso/signal_iso.h
1154 file path=usr/include/sys/jioctl.h
1155 file path=usr/include/sys/kbd.h
1156 file path=usr/include/sys/kbdreg.h
1157 file path=usr/include/sys/kbio.h
1158 file path=usr/include/sys/kcpc.h
1159 file path=usr/include/sys/kd.h
1160 file path=usr/include/sys/kdi.h
1161 file path=usr/include/sys/kdi_impl.h
1162 file path=usr/include/sys/kdi_machimpl.h
1163 $(i386_ONLY)file path=usr/include/sys/kdi_regs.h
1164 file path=usr/include/sys/kiconv.h
1165 file path=usr/include/sys/kidmap.h
1166 file path=usr/include/sys/klpd.h
1167 file path=usr/include/sys/klwp.h
1168 file path=usr/include/sys/kmem.h
1169 file path=usr/include/sys/kmem_impl.h
1170 file path=usr/include/sys/kobj.h
1171 file path=usr/include/sys/kobj_impl.h
1172 file path=usr/include/sys/ksocket.h
1173 file path=usr/include/sys/kstat.h
1174 file path=usr/include/sys/kstr.h
1175 file path=usr/include/sys/ksyms.h
1176 file path=usr/include/sys/ksynch.h

```

1177 file path=usr/include/sys/lc_core.h
1178 file path=usr/include/sys/ldterm.h
1179 file path=usr/include/sys/lgrp.h
1180 file path=usr/include/sys/lgrp_user.h
1181 file path=usr/include/sys/link.h
1182 file path=usr/include/sys/list.h
1183 file path=usr/include/sys/list_impl.h
1184 file path=usr/include/sys/llcl.h
1185 file path=usr/include/sys/loadavg.h
1186 file path=usr/include/sys/localedef.h
1187 file path=usr/include/sys/lock.h
1188 file path=usr/include/sys/lockfs.h
1189 file path=usr/include/sys/lofi.h
1190 file path=usr/include/sys/log.h
1191 file path=usr/include/sys/logindmux.h
1192 file path=usr/include/sys/lvm/md_basic.h
1193 file path=usr/include/sys/lvm/md_convert.h
1194 file path=usr/include/sys/lvm/md_crc.h
1195 file path=usr/include/sys/lvm/md_hotspares.h
1196 file path=usr/include/sys/lvm/md_mddb.h
1197 file path=usr/include/sys/lvm/md_mdiox.h
1198 file path=usr/include/sys/lvm/md_mhdx.h
1199 file path=usr/include/sys/lvm/md_mirror.h
1200 file path=usr/include/sys/lvm/md_mirror_shared.h
1201 file path=usr/include/sys/lvm/md_names.h
1202 file path=usr/include/sys/lvm/md_notify.h
1203 file path=usr/include/sys/lvm/md_raid.h
1204 file path=usr/include/sys/lvm/md_rename.h
1205 file path=usr/include/sys/lvm/md_sp.h
1206 file path=usr/include/sys/lvm/md_stripe.h
1207 file path=usr/include/sys/lvm/md_trans.h
1208 file path=usr/include/sys/lvm/mdio.h
1209 file path=usr/include/sys/lvm/mdmed.h
1210 file path=usr/include/sys/lvm/mdmn_commd.h
1211 file path=usr/include/sys/lvm/mdvar.h
1212 file path=usr/include/sys/lwp.h
1213 file path=usr/include/sys/lwp_timer_impl.h
1214 file path=usr/include/sys/lwp_upimutex_impl.h
1215 file path=usr/include/sys/mac.h
1216 file path=usr/include/sys/mac_ether.h
1217 file path=usr/include/sys/mac_flow.h
1218 file path=usr/include/sys/mac_provider.h
1219 file path=usr/include/sys/machelf.h
1220 file path=usr/include/sys/machlock.h
1221 file path=usr/include/sys/machsig.h
1222 file path=usr/include/sys/machtypes.h
1223 file path=usr/include/sys/map.h
1224 \$(i386_ONLY)file path=usr/include/sys/mc.h
1225 \$(i386_ONLY)file path=usr/include/sys/mc_amd.h
1226 \$(i386_ONLY)file path=usr/include/sys/mc_intel.h
1227 \$(i386_ONLY)file path=usr/include/sys/mca_amd.h
1228 \$(i386_ONLY)file path=usr/include/sys/mca_x86.h
1229 file path=usr/include/sys/md4.h
1230 file path=usr/include/sys/md5.h
1231 file path=usr/include/sys/md5_consts.h
1232 file path=usr/include/sys/mdi_impldefs.h
1233 file path=usr/include/sys/mem.h
1234 file path=usr/include/sys/mem_config.h
1235 file path=usr/include/sys/memlist.h
1236 file path=usr/include/sys/mhd.h
1237 file path=usr/include/sys/mii.h
1238 file path=usr/include/sys/miiregs.h
1239 file path=usr/include/sys/mkdev.h
1240 file path=usr/include/sys/mman.h
1241 file path=usr/include/sys/mmapobj.h
1242 file path=usr/include/sys/mntent.h

1243 file path=usr/include/sys/mntio.h
1244 file path=usr/include/sys/mnttab.h
1245 file path=usr/include/sys/modctl.h
1246 file path=usr/include/sys/mode.h
1247 file path=usr/include/sys/model.h
1248 file path=usr/include/sys/modhash.h
1249 file path=usr/include/sys/modhash_impl.h
1250 file path=usr/include/sys/mount.h
1251 file path=usr/include/sys/mouse.h
1252 file path=usr/include/sys/msacct.h
1253 file path=usr/include/sys/msg.h
1254 file path=usr/include/sys/msg_impl.h
1255 file path=usr/include/sys/msio.h
1256 file path=usr/include/sys/msreg.h
1257 file path=usr/include/sys/mtio.h
1258 file path=usr/include/sys/multidata.h
1259 file path=usr/include/sys/mutex.h
1260 \$(i386_ONLY)file path=usr/include/sys/mutex_impl.h
1261 file path=usr/include/sys/nbmlck.h
1262 file path=usr/include/sys/ndi_impldefs.h
1263 file path=usr/include/sys/ndifm.h
1264 file path=usr/include/sys/netconfig.h
1265 file path=usr/include/sys/neti.h
1266 file path=usr/include/sys/netstack.h
1267 file path=usr/include/sys/nexusdefs.h
1268 file path=usr/include/sys/note.h
1269 file path=usr/include/sys/nvpair.h
1270 file path=usr/include/sys/nvpair_impl.h
1271 file path=usr/include/sys/objfs.h
1272 file path=usr/include/sys/objfs_impl.h
1273 file path=usr/include/sys/obpdefs.h
1274 file path=usr/include/sys/old_procfs.h
1275 file path=usr/include/sys/open.h
1276 file path=usr/include/sys/openpromio.h
1277 file path=usr/include/sys/panic.h
1278 file path=usr/include/sys/param.h
1279 file path=usr/include/sys/pathconf.h
1280 file path=usr/include/sys/pathname.h
1281 file path=usr/include/sys/pattr.h
1282 file path=usr/include/sys/pbio.h
1283 file path=usr/include/sys/pcb.h
1284 file path=usr/include/sys/pccard.h
1285 file path=usr/include/sys/pci.h
1286 \$(i386_ONLY)file path=usr/include/sys/pcic_reg.h
1287 \$(i386_ONLY)file path=usr/include/sys/pcic_var.h
1288 file path=usr/include/sys/pcie.h
1289 file path=usr/include/sys/pcmcia.h
1290 file path=usr/include/sys/pcmcia/pcata.h
1291 file path=usr/include/sys/pcmcia/pcser_conf.h
1292 file path=usr/include/sys/pcmcia/pcser_io.h
1293 file path=usr/include/sys/pcmcia/pcser_manuspec.h
1294 file path=usr/include/sys/pcmcia/pcser_reg.h
1295 file path=usr/include/sys/pcmcia/pcser_var.h
1296 file path=usr/include/sys/pctypes.h
1297 file path=usr/include/sys/pfmod.h
1298 file path=usr/include/sys/pg.h
1299 file path=usr/include/sys/pghw.h
1300 file path=usr/include/sys/phymem.h
1301 \$(i386_ONLY)file path=usr/include/sys/pic.h
1302 \$(i386_ONLY)file path=usr/include/sys/pit.h
1303 file path=usr/include/sys/psk_hash.h
1304 file path=usr/include/sys/pm.h
1305 \$(i386_ONLY)file path=usr/include/sys/pmem.h
1306 file path=usr/include/sys/policy.h
1307 file path=usr/include/sys/poll.h
1308 file path=usr/include/sys/poll_impl.h

```

1309 file path=usr/include/sys/pool.h
1310 file path=usr/include/sys/pool_impl.h
1311 file path=usr/include/sys/pool_pset.h
1312 file path=usr/include/sys/port.h
1313 file path=usr/include/sys/port_impl.h
1314 file path=usr/include/sys/port_kernel.h
1315 file path=usr/include/sys/ppmio.h
1316 file path=usr/include/sys/priocntl.h
1317 file path=usr/include/sys/priv.h
1318 file path=usr/include/sys/priv_const.h
1319 file path=usr/include/sys/priv_impl.h
1320 file path=usr/include/sys/priv_names.h
1321 $(i386_ONLY)file path=usr/include/sys/privregs.h
1322 $(i386_ONLY)file path=usr/include/sys/privregs.h
1323 file path=usr/include/sys/prnio.h
1324 file path=usr/include/sys/proc.h
1325 file path=usr/include/sys/proc/prdata.h
1326 file path=usr/include/sys/processor.h
1327 file path=usr/include/sys/procfs.h
1328 file path=usr/include/sys/procfs_isa.h
1329 file path=usr/include/sys/procset.h
1330 file path=usr/include/sys/project.h
1331 $(i386_ONLY)file path=usr/include/sys/prom_emul.h
1332 $(i386_ONLY)file path=usr/include/sys/prom_isa.h
1333 $(i386_ONLY)file path=usr/include/sys/prom_plat.h
1334 file path=usr/include/sys/promif.h
1335 file path=usr/include/sys/promimpl.h
1336 file path=usr/include/sys/protosw.h
1337 file path=usr/include/sys/prsysm.h
1338 file path=usr/include/sys/pset.h
1339 file path=usr/include/sys/psw.h
1340 $(i386_ONLY)file path=usr/include/sys/pte.h
1341 file path=usr/include/sys/ptem.h
1342 file path=usr/include/sys/ptms.h
1343 file path=usr/include/sys/ptyvar.h
1344 file path=usr/include/sys/queue.h
1345 file path=usr/include/sys/raidioctl.h
1346 file path=usr/include/sys/ramdisk.h
1347 file path=usr/include/sys/random.h
1348 file path=usr/include/sys/rctl.h
1349 file path=usr/include/sys/rctl_impl.h
1350 file path=usr/include/sys/rds.h
1351 file path=usr/include/sys/reboot.h
1352 file path=usr/include/sys/refstr.h
1353 file path=usr/include/sys/refstr_impl.h
1354 file path=usr/include/sys/reg.h
1355 file path=usr/include/sys/regset.h
1356 file path=usr/include/sys/resource.h
1357 file path=usr/include/sys/rliocntl.h
1358 file path=usr/include/sys/rsm/rsm.h
1359 file path=usr/include/sys/rsm/rsm_common.h
1360 file path=usr/include/sys/rsm/rsmapi_common.h
1361 file path=usr/include/sys/rsm/rsmka_path_int.h
1362 file path=usr/include/sys/rsm/rsmndi.h
1363 file path=usr/include/sys/rsm/rsmapi.h
1364 file path=usr/include/sys/rsm/rsmapi_driver.h
1365 file path=usr/include/sys/rt.h
1366 $(i386_ONLY)file path=usr/include/sys/rtc.h
1367 file path=usr/include/sys/rtpriocntl.h
1368 file path=usr/include/sys/rwlock.h
1369 file path=usr/include/sys/rwlock_impl.h
1370 file path=usr/include/sys/rwstlock.h
1371 file path=usr/include/sys/sad.h
1372 $(i386_ONLY)file path=usr/include/sys/sata/sata_defs.h
1373 $(i386_ONLY)file path=usr/include/sys/sata/sata_hba.h
1374 file path=usr/include/sys/schedctl.h

```

```

1375 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/ifpio.h
1376 file path=usr/include/sys/scsi/adapters/scsi_vhci.h
1377 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/sfvar.h
1378 file path=usr/include/sys/scsi/conf/autocnf.h
1379 file path=usr/include/sys/scsi/conf/device.h
1380 file path=usr/include/sys/scsi/generic/commands.h
1381 file path=usr/include/sys/scsi/generic/dad_mode.h
1382 file path=usr/include/sys/scsi/generic/inquiry.h
1383 file path=usr/include/sys/scsi/generic/message.h
1384 file path=usr/include/sys/scsi/generic/mode.h
1385 file path=usr/include/sys/scsi/generic/persist.h
1386 file path=usr/include/sys/scsi/generic/sense.h
1387 file path=usr/include/sys/scsi/generic/sff_frames.h
1388 file path=usr/include/sys/scsi/generic/smp_frames.h
1389 file path=usr/include/sys/scsi/generic/status.h
1390 file path=usr/include/sys/scsi/impl/commands.h
1391 file path=usr/include/sys/scsi/impl/inquiry.h
1392 file path=usr/include/sys/scsi/impl/mode.h
1393 file path=usr/include/sys/scsi/impl/scsi_reset_notify.h
1394 file path=usr/include/sys/scsi/impl/scsi_sas.h
1395 file path=usr/include/sys/scsi/impl/sense.h
1396 file path=usr/include/sys/scsi/impl/services.h
1397 file path=usr/include/sys/scsi/impl/smp_transport.h
1398 file path=usr/include/sys/scsi/impl/spc3_types.h
1399 file path=usr/include/sys/scsi/impl/status.h
1400 file path=usr/include/sys/scsi/impl/transport.h
1401 file path=usr/include/sys/scsi/impl/types.h
1402 file path=usr/include/sys/scsi/impl/uscsi.h
1403 file path=usr/include/sys/scsi/impl/usmp.h
1404 file path=usr/include/sys/scsi/scsi.h
1405 file path=usr/include/sys/scsi/scsi_address.h
1406 file path=usr/include/sys/scsi/scsi_ctl.h
1407 file path=usr/include/sys/scsi/scsi_fm.h
1408 file path=usr/include/sys/scsi/scsi_params.h
1409 file path=usr/include/sys/scsi/scsi_pkt.h
1410 file path=usr/include/sys/scsi/scsi_resource.h
1411 file path=usr/include/sys/scsi/scsi_types.h
1412 file path=usr/include/sys/scsi/scsi_watch.h
1413 file path=usr/include/sys/scsi/targets/sddef.h
1414 file path=usr/include/sys/scsi/targets/ses.h
1415 file path=usr/include/sys/scsi/targets/sesio.h
1416 file path=usr/include/sys/scsi/targets/sgendef.h
1417 file path=usr/include/sys/scsi/targets/smp.h
1418 $(sparc_ONLY)file path=usr/include/sys/scsi/targets/ssddef.h
1419 file path=usr/include/sys/scsi/targets/stdef.h
1420 $(i386_ONLY)file path=usr/include/sys/segment.h
1421 $(i386_ONLY)file path=usr/include/sys/segments.h
1422 file path=usr/include/sys/select.h
1423 file path=usr/include/sys/sem.h
1424 file path=usr/include/sys/sem_impl.h
1425 file path=usr/include/sys/semaphore.h
1426 file path=usr/include/sys/semaphore.h
1427 file path=usr/include/sys/sendfile.h
1428 $(sparc_ONLY)file path=usr/include/sys/ser_async.h
1429 file path=usr/include/sys/ser_sync.h
1430 $(sparc_ONLY)file path=usr/include/sys/ser_zscc.h
1431 file path=usr/include/sys/serializer.h
1432 file path=usr/include/sys/session.h
1433 file path=usr/include/sys/sha1.h
1434 file path=usr/include/sys/sha2.h
1435 file path=usr/include/sys/share.h
1436 file path=usr/include/sys/shm.h
1437 file path=usr/include/sys/shm_impl.h
1438 file path=usr/include/sys/sid.h
1439 file path=usr/include/sys/siginfo.h
1440 file path=usr/include/sys/signal.h

```

1441 file path=usr/include/sys/sleepq.h
1442 file path=usr/include/sys/smbios.h
1443 file path=usr/include/sys/smbios_impl.h
1444 file path=usr/include/sys/smedia.h
1445 file path=usr/include/sys/subject.h
1446 \$(sparc_ONLY)file path=usr/include/sys/socal_cq_defs.h
1447 \$(sparc_ONLY)file path=usr/include/sys/socalio.h
1448 \$(sparc_ONLY)file path=usr/include/sys/socalmap.h
1449 \$(sparc_ONLY)file path=usr/include/sys/socalreg.h
1450 \$(sparc_ONLY)file path=usr/include/sys/socalvar.h
1451 file path=usr/include/sys/socket.h
1452 file path=usr/include/sys/socket_impl.h
1453 file path=usr/include/sys/socket_proto.h
1454 file path=usr/include/sys/socketvar.h
1455 file path=usr/include/sys/sockio.h
1456 file path=usr/include/sys/spl.h
1457 file path=usr/include/sys/queue.h
1458 file path=usr/include/sys/queue_impl.h
1459 file path=usr/include/sys/sservice.h
1460 file path=usr/include/sys/stack.h
1461 file path=usr/include/sys/stat.h
1462 file path=usr/include/sys/stat_impl.h
1463 file path=usr/include/sys/statfs.h
1464 file path=usr/include/sys/statvfs.h
1465 file path=usr/include/sys/stdbool.h
1466 file path=usr/include/sys/stdint.h
1467 file path=usr/include/sys/stermio.h
1468 file path=usr/include/sys/stream.h
1469 file path=usr/include/sys/strft.h
1470 file path=usr/include/sys/strlog.h
1471 file path=usr/include/sys/strmdep.h
1472 file path=usr/include/sys/stropts.h
1473 file path=usr/include/sys/strredir.h
1474 file path=usr/include/sys/strstat.h
1475 file path=usr/include/sys/strsubr.h
1476 file path=usr/include/sys/strsun.h
1477 file path=usr/include/sys/strtty.h
1478 file path=usr/include/sys/sunddi.h
1479 file path=usr/include/sys/sunldi.h
1480 file path=usr/include/sys/sunldi_impl.h
1481 file path=usr/include/sys/sunmdi.h
1482 file path=usr/include/sys/sunndi.h
1483 file path=usr/include/sys/sunpm.h
1484 file path=usr/include/sys/suntpi.h
1485 file path=usr/include/sys/suntty.h
1486 file path=usr/include/sys/swap.h
1487 file path=usr/include/sys/synch.h
1488 file path=usr/include/sys/syscall.h
1489 file path=usr/include/sys/sysconf.h
1490 file path=usr/include/sys/sysconfig.h
1491 file path=usr/include/sys/sysconfig_impl.h
1492 file path=usr/include/sys/sysdc.h
1493 file path=usr/include/sys/sysdc_impl.h
1494 file path=usr/include/sys/sysevent.h
1495 file path=usr/include/sys/sysevent/ap_driver.h
1496 file path=usr/include/sys/sysevent/dev.h
1497 file path=usr/include/sys/sysevent/domain.h
1498 file path=usr/include/sys/sysevent/dr.h
1499 file path=usr/include/sys/sysevent/env.h
1500 file path=usr/include/sys/sysevent/eventdefs.h
1501 file path=usr/include/sys/sysevent/imp.h
1502 file path=usr/include/sys/sysevent/pwrctl.h
1503 file path=usr/include/sys/sysevent/svm.h
1504 file path=usr/include/sys/sysevent/vrrp.h
1505 file path=usr/include/sys/sysevent_impl.h
1506 \$(i386_ONLY)file path=usr/include/sys/sysi86.h

1507 file path=usr/include/sys/sysinfo.h
1508 file path=usr/include/sys/syslog.h
1509 file path=usr/include/sys/sysmacros.h
1510 file path=usr/include/sys/systeminfo.h
1511 file path=usr/include/sys/system.h
1512 file path=usr/include/sys/t_kuser.h
1513 file path=usr/include/sys/t_lock.h
1514 file path=usr/include/sys/task.h
1515 file path=usr/include/sys/taskq.h
1516 file path=usr/include/sys/taskq_impl.h
1517 file path=usr/include/sys/teliocntl.h
1518 file path=usr/include/sys/termio.h
1519 file path=usr/include/sys/termios.h
1520 file path=usr/include/sys/termiox.h
1521 file path=usr/include/sys/thread.h
1522 file path=usr/include/sys/ticlts.h
1523 file path=usr/include/sys/ticots.h
1524 file path=usr/include/sys/ticotsord.h
1525 file path=usr/include/sys/tihdr.h
1526 file path=usr/include/sys/time.h
1527 file path=usr/include/sys/time_impl.h
1528 file path=usr/include/sys/time_std_impl.h
1529 file path=usr/include/sys/timeb.h
1530 file path=usr/include/sys/timer.h
1531 file path=usr/include/sys/times.h
1532 file path=usr/include/sys/timex.h
1533 file path=usr/include/sys/timod.h
1534 file path=usr/include/sys/tirdwr.h
1535 file path=usr/include/sys/tiuser.h
1536 file path=usr/include/sys/tl.h
1537 file path=usr/include/sys/tnf.h
1538 file path=usr/include/sys/tnf_com.h
1539 file path=usr/include/sys/tnf_probe.h
1540 file path=usr/include/sys/tnf_writer.h
1541 file path=usr/include/sys/todio.h
1542 file path=usr/include/sys/tpicommon.h
1543 file path=usr/include/sys/trap.h
1544 \$(i386_ONLY)file path=usr/include/sys/traptrace.h
1545 file path=usr/include/sys/ts.h
1546 file path=usr/include/sys/tsol/label.h
1547 file path=usr/include/sys/tsol/label_macro.h
1548 file path=usr/include/sys/tsol/priv.h
1549 file path=usr/include/sys/tsol/tndb.h
1550 file path=usr/include/sys/tsol/tsyscall.h
1551 file path=usr/include/sys/tspriccntl.h
1552 \$(i386_ONLY)file path=usr/include/sys/tss.h
1553 file path=usr/include/sys/ttcompat.h
1554 file path=usr/include/sys/ttold.h
1555 file path=usr/include/sys/tty.h
1556 file path=usr/include/sys/ttychars.h
1557 file path=usr/include/sys/ttydev.h
1558 \$(sparc_ONLY)file path=usr/include/sys/ttymux.h
1559 \$(sparc_ONLY)file path=usr/include/sys/ttymuxuser.h
1560 file path=usr/include/sys/tuneable.h
1561 file path=usr/include/sys/turnstile.h
1562 file path=usr/include/sys/types.h
1563 file path=usr/include/sys/types32.h
1564 file path=usr/include/sys/tzfile.h
1565 file path=usr/include/sys/u8_textprep.h
1566 file path=usr/include/sys/uadmin.h
1567 \$(i386_ONLY)file path=usr/include/sys/ucode.h
1568 file path=usr/include/sys/ucontext.h
1569 file path=usr/include/sys/uio.h
1570 file path=usr/include/sys/ulimit.h
1571 file path=usr/include/sys/un.h
1572 file path=usr/include/sys/unistd.h

```

1573 file path=usr/include/sys/user.h
1574 file path=usr/include/sys/ustat.h
1575 file path=usr/include/sys/utime.h
1576 file path=usr/include/sys/utrap.h
1577 file path=usr/include/sys/utsname.h
1578 file path=usr/include/sys/utssys.h
1579 file path=usr/include/sys/uuid.h
1580 file path=usr/include/sys/va_impl.h
1581 file path=usr/include/sys/va_list.h
1582 file path=usr/include/sys/var.h
1583 file path=usr/include/sys/varargs.h
1584 file path=usr/include/sys/vfs.h
1585 file path=usr/include/sys/vfs_opreg.h
1586 file path=usr/include/sys/vfstab.h
1587 file path=usr/include/sys/videodev2.h
1588 file path=usr/include/sys/visual_io.h
1589 file path=usr/include/sys/vm.h
1590 file path=usr/include/sys/vm_usage.h
1591 file path=usr/include/sys/vmem.h
1592 file path=usr/include/sys/vmem_impl.h
1593 file path=usr/include/sys/vmem_impl_user.h
1594 file path=usr/include/sys/vmparam.h
1595 file path=usr/include/sys/vmsystem.h
1596 file path=usr/include/sys/vnode.h
1597 file path=usr/include/sys/vt.h
1598 file path=usr/include/sys/vtdaemon.h
1599 file path=usr/include/sys/vtoc.h
1600 file path=usr/include/sys/vtrace.h
1601 file path=usr/include/sys/vuid_event.h
1602 file path=usr/include/sys/vuid_queue.h
1603 file path=usr/include/sys/vuid_state.h
1604 file path=usr/include/sys/vuid_store.h
1605 file path=usr/include/sys/vuid_wheel.h
1606 file path=usr/include/sys/wait.h
1607 file path=usr/include/sys/waitq.h
1608 file path=usr/include/sys/watchpoint.h
1609 $(i386_ONLY)file path=usr/include/sys/x86_archext.h
1610 $(i386_ONLY)file path=usr/include/sys/xen_errno.h
1611 file path=usr/include/sys/xti_inet.h
1612 file path=usr/include/sys/xti_osi.h
1613 file path=usr/include/sys/xti_xtiopt.h
1614 file path=usr/include/sys/zcons.h
1615 file path=usr/include/sys/zmod.h
1616 file path=usr/include/sys/zone.h
1617 $(sparc_ONLY)file path=usr/include/sys/zsdev.h
1618 file path=usr/include/syssexits.h
1619 file path=usr/include/syslog.h
1620 file path=usr/include/tar.h
1621 file path=usr/include/tcpd.h
1622 file path=usr/include/term.h
1623 file path=usr/include/termcap.h
1624 file path=usr/include/termio.h
1625 file path=usr/include/termios.h
1626 file path=usr/include/thread.h
1627 file path=usr/include/thread_db.h
1628 file path=usr/include/time.h
1629 file path=usr/include/tiuser.h
1630 file path=usr/include/tsol/label.h
1631 file path=usr/include/tzfile.h
1632 file path=usr/include/ucontext.h
1633 file path=usr/include/ucred.h
1634 file path=usr/include/uid_stp.h
1635 file path=usr/include/ulimit.h
1636 file path=usr/include/umem.h
1637 file path=usr/include/umem_impl.h
1638 file path=usr/include/unctrl.h

```

```

1639 file path=usr/include/unistd.h
1640 file path=usr/include/user_attr.h
1641 file path=usr/include/userdefs.h
1642 file path=usr/include/ustat.h
1643 file path=usr/include/utility.h
1644 file path=usr/include/utime.h
1645 file path=usr/include/utmp.h
1646 file path=usr/include/utmpx.h
1647 file path=usr/include/uuid/uuid.h
1648 $(sparc_ONLY)file path=usr/include/v7/sys/machpcb.h
1649 $(sparc_ONLY)file path=usr/include/v7/sys/machtrap.h
1650 $(sparc_ONLY)file path=usr/include/v7/sys/mutex_impl.h
1651 $(sparc_ONLY)file path=usr/include/v7/sys/privregs.h
1652 $(sparc_ONLY)file path=usr/include/v7/sys/prom_isa.h
1653 $(sparc_ONLY)file path=usr/include/v7/sys/psr.h
1654 $(sparc_ONLY)file path=usr/include/v7/sys/traptrace.h
1655 $(sparc_ONLY)file path=usr/include/v9/sys/asi.h
1656 $(sparc_ONLY)file path=usr/include/v9/sys/machpcb.h
1657 $(sparc_ONLY)file path=usr/include/v9/sys/machtrap.h
1658 $(sparc_ONLY)file path=usr/include/v9/sys/membar.h
1659 $(sparc_ONLY)file path=usr/include/v9/sys/mutex_impl.h
1660 $(sparc_ONLY)file path=usr/include/v9/sys/privregs.h
1661 $(sparc_ONLY)file path=usr/include/v9/sys/prom_isa.h
1662 $(sparc_ONLY)file path=usr/include/v9/sys/psr_compat.h
1663 $(sparc_ONLY)file path=usr/include/v9/sys/vis_simulator.h
1664 file path=usr/include/valtools.h
1665 file path=usr/include/values.h
1666 file path=usr/include/varargs.h
1667 file path=usr/include/vm/anon.h
1668 file path=usr/include/vm/as.h
1669 file path=usr/include/vm/faultcode.h
1670 file path=usr/include/vm/hat.h
1671 file path=usr/include/vm/kpm.h
1672 file path=usr/include/vm/page.h
1673 file path=usr/include/vm/pvn.h
1674 file path=usr/include/vm/rm.h
1675 file path=usr/include/vm/seg.h
1676 file path=usr/include/vm/seg_dev.h
1677 file path=usr/include/vm/seg_enum.h
1678 file path=usr/include/vm/seg_kmem.h
1679 file path=usr/include/vm/seg_kp.h
1680 file path=usr/include/vm/seg_kpm.h
1681 file path=usr/include/vm/seg_map.h
1682 file path=usr/include/vm/seg_spt.h
1683 file path=usr/include/vm/seg_vn.h
1684 file path=usr/include/vm/vpage.h
1685 file path=usr/include/vm/vpm.h
1686 file path=usr/include/volmgt.h
1687 file path=usr/include/wait.h
1688 file path=usr/include/wchar.h
1689 file path=usr/include/wchar_impl.h
1690 file path=usr/include/wctype.h
1691 file path=usr/include/widex.h
1692 file path=usr/include/wordexp.h
1693 file path=usr/include/xti.h
1694 file path=usr/include/xti_inet.h
1695 file path=usr/include/zone.h
1696 file path=usr/include/zonestat.h
1697 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/acpidev.h
1698 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/amd_iommu.h
1699 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/asm_misc.h
1700 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/clock.h
1701 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/cram.h
1702 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/ddi_subrdefs.h
1703 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/debug_info.h
1704 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/fastboot.h

```

```

1705 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/mach_mmu.h
1706 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machclock.h
1707 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machcpuvar.h
1708 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machparam.h
1709 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machprivregs.h
1710 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machsystem.h
1711 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machthread.h
1712 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/memnode.h
1713 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/pc_mmu.h
1714 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm.h
1715 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_defs.h
1716 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_modctl.h
1717 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_types.h
1718 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/rm_platter.h
1719 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/sbd_ioctl.h
1720 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/smp_impldefs.h
1721 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/vm_machparam.h
1722 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/x_call.h
1723 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/xc_levels.h
1724 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/xsvc.h
1725 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_186.h
1726 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_pte.h
1727 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hment.h
1728 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/htable.h
1729 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/kboot_mmu.h
1730 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/balloon.h
1731 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/machprivregs.h
1732 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xen_mmu.h
1733 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xpv_impl.h
1734 $(i386_ONLY)file path=usr/platform/i86xpv/include/vm/seg_mf.h
1735 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ac.h
1736 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/asynch.h
1737 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cheetahregs.h
1738 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cherrystone.h
1739 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/clock.h
1740 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cmp.h
1741 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpc_ultra.h
1742 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpr_impl.h
1743 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_impl.h
1744 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_sgnblk_defs.h
1745 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cvc.h
1746 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/daktari.h
1747 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ddi_subrdefs.h
1748 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/dvma.h
1749 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ecc_kstat.h
1750 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/EEPROM.h
1751 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl.h
1752 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_gen.h
1753 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue250.h
1754 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue450.h
1755 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envrion.h
1756 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/errclassif.h
1757 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/fhc.h
1758 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/gpio_87317.h
1759 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/hpc3130_events.h
1760 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/hpc3130.h
1761 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/i2c_client.h
1762 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/lm75.h
1763 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/max1617.h
1764 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/pcf8591.h
1765 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/ssc050.h
1766 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_misc/i2c_svc.h
1767 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/idprom.h
1768 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intr.h
1769 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intreg.h
1770 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iocache.h

```

```

1771 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iommu.h
1772 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ivintr.h
1773 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/lom_io.h
1774 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machasi.h
1775 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machclock.h
1776 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machcpuvar.h
1777 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machparam.h
1778 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machsystem.h
1779 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machthread.h
1780 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mem_cache.h
1781 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memlist_plat.h
1782 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memnode.h
1783 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mmu.h
1784 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/nexusdebug.h
1785 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_hwdesc.h
1786 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_module.h
1787 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_debug.h
1788 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_plat.h
1789 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/pte.h
1790 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sbd_ioctl.h
1791 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scb.h
1792 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scsb_led.h
1793 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/simstat.h
1794 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/spitregs.h
1795 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sram.h
1796 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/starfire.h
1797 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sun4asi.h
1798 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysctrl.h
1799 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysioerr.h
1800 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysoisbus.h
1801 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/tod.h
1802 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/todmostek.h
1803 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/trapstat.h
1804 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/traptrace.h
1805 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/tris.h
1806 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vm_machparam.h
1807 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/x_call.h
1808 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/xc_impl.h
1809 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/zsmach.h
1810 $(sparc_ONLY)file path=usr/platform/sun4u/include/vm/hat_sfmmu.h
1811 $(sparc_ONLY)file path=usr/platform/sun4u/include/vm/mach_sfmmu.h
1812 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/clock.h
1813 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cmp.h
1814 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpc_ultra.h
1815 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpu_sgnblk_defs.h
1816 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ddi_subrdefs.h
1817 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_pri.h
1818 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_smp.h
1819 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/dvma.h
1820 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/EEPROM.h
1821 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/fcdev.h
1822 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hsvc.h
1823 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hypervisor_api.h
1824 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/idprom.h
1825 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intr.h
1826 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intreg.h
1827 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ivintr.h
1828 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machasi.h
1829 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machclock.h
1830 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machcpuvar.h
1831 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machintreg.h
1832 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machparam.h
1833 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machsystem.h
1834 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machthread.h
1835 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memlist_plat.h
1836 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memnode.h

```

```

1837 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/mmu.h
1838 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/nexusdebug.h
1839 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagaraasi.h
1840 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagararegs.h
1841 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ntwdt.h
1842 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pri.h
1843 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_debug.h
1844 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_plat.h
1845 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pte.h
1846 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/qcn.h
1847 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/scb.h
1848 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/soft_state.h
1849 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/sun4asi.h
1850 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/tod.h
1851 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/trapstat.h
1852 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/traptrace.h
1853 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vis.h
1854 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vm_machparam.h
1855 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/x_call.h
1856 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/xc_impl.h
1857 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/zsmach.h
1858 $(sparc_ONLY)file path=usr/platform/sun4v/include/vm/hat_sfmmu.h
1859 $(sparc_ONLY)file path=usr/platform/sun4v/include/vm/mach_sfmmu.h
1860 file path=usr/share/man/man3head/acct.3head
1861 file path=usr/share/man/man3head/acct.h.3head
1862 file path=usr/share/man/man3head/aio.3head
1863 file path=usr/share/man/man3head/aio.h.3head
1864 file path=usr/share/man/man3head/ar.3head
1865 file path=usr/share/man/man3head/ar.h.3head
1866 file path=usr/share/man/man3head/archives.3head
1867 file path=usr/share/man/man3head/archives.h.3head
1868 file path=usr/share/man/man3head/assert.3head
1869 file path=usr/share/man/man3head/assert.h.3head
1870 file path=usr/share/man/man3head/complex.3head
1871 file path=usr/share/man/man3head/complex.h.3head
1872 file path=usr/share/man/man3head/cpio.3head
1873 file path=usr/share/man/man3head/cpio.h.3head
1874 file path=usr/share/man/man3head/dirent.3head
1875 file path=usr/share/man/man3head/dirent.h.3head
1876 file path=usr/share/man/man3head/errno.3head
1877 file path=usr/share/man/man3head/errno.h.3head
1878 file path=usr/share/man/man3head/fcntl.3head
1879 file path=usr/share/man/man3head/fcntl.h.3head
1880 file path=usr/share/man/man3head/fenv.3head
1881 file path=usr/share/man/man3head/fenv.h.3head
1882 file path=usr/share/man/man3head/float.3head
1883 file path=usr/share/man/man3head/float.h.3head
1884 file path=usr/share/man/man3head/floatingpoint.3head
1885 file path=usr/share/man/man3head/floatingpoint.h.3head
1886 file path=usr/share/man/man3head/fmtmsg.3head
1887 file path=usr/share/man/man3head/fmtmsg.h.3head
1888 file path=usr/share/man/man3head/fnmatch.3head
1889 file path=usr/share/man/man3head/fnmatch.h.3head
1890 file path=usr/share/man/man3head/ftw.3head
1891 file path=usr/share/man/man3head/ftw.h.3head
1892 file path=usr/share/man/man3head/glob.3head
1893 file path=usr/share/man/man3head/glob.h.3head
1894 file path=usr/share/man/man3head/grp.3head
1895 file path=usr/share/man/man3head/grp.h.3head
1896 file path=usr/share/man/man3head/iconv.3head
1897 file path=usr/share/man/man3head/iconv.h.3head
1898 file path=usr/share/man/man3head/if.3head
1899 file path=usr/share/man/man3head/if.h.3head
1900 file path=usr/share/man/man3head/in.3head
1901 file path=usr/share/man/man3head/in.h.3head
1902 file path=usr/share/man/man3head/inet.3head

```

```

1903 file path=usr/share/man/man3head/inet.h.3head
1904 file path=usr/share/man/man3head/inttypes.3head
1905 file path=usr/share/man/man3head/inttypes.h.3head
1906 file path=usr/share/man/man3head/ipc.3head
1907 file path=usr/share/man/man3head/ipc.h.3head
1908 file path=usr/share/man/man3head/iso646.3head
1909 file path=usr/share/man/man3head/iso646.h.3head
1910 file path=usr/share/man/man3head/langinfo.3head
1911 file path=usr/share/man/man3head/langinfo.h.3head
1912 file path=usr/share/man/man3head/libgen.3head
1913 file path=usr/share/man/man3head/libgen.h.3head
1914 file path=usr/share/man/man3head/libintl.3head
1915 file path=usr/share/man/man3head/libintl.h.3head
1916 file path=usr/share/man/man3head/limits.3head
1917 file path=usr/share/man/man3head/limits.h.3head
1918 file path=usr/share/man/man3head/locale.3head
1919 file path=usr/share/man/man3head/locale.h.3head
1920 file path=usr/share/man/man3head/math.3head
1921 file path=usr/share/man/man3head/math.h.3head
1922 file path=usr/share/man/man3head/mman.3head
1923 file path=usr/share/man/man3head/mman.h.3head
1924 file path=usr/share/man/man3head/monetary.3head
1925 file path=usr/share/man/man3head/monetary.h.3head
1926 file path=usr/share/man/man3head/mqueue.3head
1927 file path=usr/share/man/man3head/mqueue.h.3head
1928 file path=usr/share/man/man3head/msg.3head
1929 file path=usr/share/man/man3head/msg.h.3head
1930 file path=usr/share/man/man3head/ndbm.3head
1931 file path=usr/share/man/man3head/ndbm.h.3head
1932 file path=usr/share/man/man3head/netdb.3head
1933 file path=usr/share/man/man3head/netdb.h.3head
1934 file path=usr/share/man/man3head/nl_types.3head
1935 file path=usr/share/man/man3head/nl_types.h.3head
1936 file path=usr/share/man/man3head/poll.3head
1937 file path=usr/share/man/man3head/poll.h.3head
1938 file path=usr/share/man/man3head/pthread.3head
1939 file path=usr/share/man/man3head/pthread.h.3head
1940 file path=usr/share/man/man3head/pwd.3head
1941 file path=usr/share/man/man3head/pwd.h.3head
1942 file path=usr/share/man/man3head/regex.3head
1943 file path=usr/share/man/man3head/regex.h.3head
1944 file path=usr/share/man/man3head/resource.3head
1945 file path=usr/share/man/man3head/resource.h.3head
1946 file path=usr/share/man/man3head/sched.3head
1947 file path=usr/share/man/man3head/sched.h.3head
1948 file path=usr/share/man/man3head/search.3head
1949 file path=usr/share/man/man3head/search.h.3head
1950 file path=usr/share/man/man3head/select.3head
1951 file path=usr/share/man/man3head/select.h.3head
1952 file path=usr/share/man/man3head/sem.3head
1953 file path=usr/share/man/man3head/sem.h.3head
1954 file path=usr/share/man/man3head/semaphore.3head
1955 file path=usr/share/man/man3head/semaphore.h.3head
1956 file path=usr/share/man/man3head/setjmp.3head
1957 file path=usr/share/man/man3head/setjmp.h.3head
1958 file path=usr/share/man/man3head/shm.3head
1959 file path=usr/share/man/man3head/shm.h.3head
1960 file path=usr/share/man/man3head/signinfo.3head
1961 file path=usr/share/man/man3head/signinfo.h.3head
1962 file path=usr/share/man/man3head/signal.3head
1963 file path=usr/share/man/man3head/signal.h.3head
1964 file path=usr/share/man/man3head/socket.3head
1965 file path=usr/share/man/man3head/socket.h.3head
1966 file path=usr/share/man/man3head/spawn.3head
1967 file path=usr/share/man/man3head/spawn.h.3head
1968 file path=usr/share/man/man3head/stat.3head

```

```

1969 file path=usr/share/man/man3head/stat.h.3head
1970 file path=usr/share/man/man3head/statvfs.3head
1971 file path=usr/share/man/man3head/statvfs.h.3head
1972 file path=usr/share/man/man3head/stdbool.3head
1973 file path=usr/share/man/man3head/stdbool.h.3head
1974 file path=usr/share/man/man3head/stddef.3head
1975 file path=usr/share/man/man3head/stddef.h.3head
1976 file path=usr/share/man/man3head/stdint.3head
1977 file path=usr/share/man/man3head/stdint.h.3head
1978 file path=usr/share/man/man3head/stdio.3head
1979 file path=usr/share/man/man3head/stdio.h.3head
1980 file path=usr/share/man/man3head/stdlib.3head
1981 file path=usr/share/man/man3head/stdlib.h.3head
1982 file path=usr/share/man/man3head/string.3head
1983 file path=usr/share/man/man3head/string.h.3head
1984 file path=usr/share/man/man3head/strings.3head
1985 file path=usr/share/man/man3head/strings.h.3head
1986 file path=usr/share/man/man3head/stropts.3head
1987 file path=usr/share/man/man3head/stropts.h.3head
1988 file path=usr/share/man/man3head/syslog.3head
1989 file path=usr/share/man/man3head/syslog.h.3head
1990 file path=usr/share/man/man3head/tar.3head
1991 file path=usr/share/man/man3head/tar.h.3head
1992 file path=usr/share/man/man3head/tcp.3head
1993 file path=usr/share/man/man3head/tcp.h.3head
1994 file path=usr/share/man/man3head/termios.3head
1995 file path=usr/share/man/man3head/termios.h.3head
1996 file path=usr/share/man/man3head/tgmath.3head
1997 file path=usr/share/man/man3head/tgmath.h.3head
1998 file path=usr/share/man/man3head/time.3head
1999 file path=usr/share/man/man3head/time.h.3head
2000 file path=usr/share/man/man3head/timeb.3head
2001 file path=usr/share/man/man3head/timeb.h.3head
2002 file path=usr/share/man/man3head/times.3head
2003 file path=usr/share/man/man3head/times.h.3head
2004 file path=usr/share/man/man3head/types.3head
2005 file path=usr/share/man/man3head/types.h.3head
2006 file path=usr/share/man/man3head/types32.3head
2007 file path=usr/share/man/man3head/types32.h.3head
2008 file path=usr/share/man/man3head/ucontext.3head
2009 file path=usr/share/man/man3head/ucontext.h.3head
2010 file path=usr/share/man/man3head/uioc.3head
2011 file path=usr/share/man/man3head/uioc.h.3head
2012 file path=usr/share/man/man3head/ulimit.3head
2013 file path=usr/share/man/man3head/ulimit.h.3head
2014 file path=usr/share/man/man3head/un.3head
2015 file path=usr/share/man/man3head/un.h.3head
2016 file path=usr/share/man/man3head/unistd.3head
2017 file path=usr/share/man/man3head/unistd.h.3head
2018 file path=usr/share/man/man3head/utime.3head
2019 file path=usr/share/man/man3head/utime.h.3head
2020 file path=usr/share/man/man3head/utmpx.3head
2021 file path=usr/share/man/man3head/utmpx.h.3head
2022 file path=usr/share/man/man3head/utsname.3head
2023 file path=usr/share/man/man3head/utsname.h.3head
2024 file path=usr/share/man/man3head/values.3head
2025 file path=usr/share/man/man3head/values.h.3head
2026 file path=usr/share/man/man3head/wait.3head
2027 file path=usr/share/man/man3head/wait.h.3head
2028 file path=usr/share/man/man3head/wchar.3head
2029 file path=usr/share/man/man3head/wchar.h.3head
2030 file path=usr/share/man/man3head/wctype.3head
2031 file path=usr/share/man/man3head/wctype.h.3head
2032 file path=usr/share/man/man3head/wordexp.3head
2033 file path=usr/share/man/man3head/wordexp.h.3head
2034 file path=usr/share/man/man4/note.4

```

```

2035 file path=usr/share/man/man5/prof.5
2036 file path=usr/share/man/man7i/cdio.7i
2037 file path=usr/share/man/man7i/dkio.7i
2038 file path=usr/share/man/man7i/fbio.7i
2039 file path=usr/share/man/man7i/fdio.7i
2040 file path=usr/share/man/man7i/hdio.7i
2041 file path=usr/share/man/man7i/iec61883.7i
2042 file path=usr/share/man/man7i/mhd.7i
2043 file path=usr/share/man/man7i/mtio.7i
2044 file path=usr/share/man/man7i/prnio.7i
2045 file path=usr/share/man/man7i/quotactl.7i
2046 file path=usr/share/man/man7i/sesio.7i
2047 file path=usr/share/man/man7i/sockio.7i
2048 file path=usr/share/man/man7i/streamio.7i
2049 file path=usr/share/man/man7i/termio.7i
2050 file path=usr/share/man/man7i/termiox.7i
2051 file path=usr/share/man/man7i/uscsi.7i
2052 file path=usr/share/man/man7i/visual_io.7i
2053 file path=usr/share/man/man7i/vt.7i
2054 file path=usr/xpg4/include/curses.h
2055 file path=usr/xpg4/include/term.h
2056 file path=usr/xpg4/include/unctrl.h
2057 legacy pkg=SUNWhea \
2058 desc="SunOS C/C++ header files for general development of software" \
2059 name="SunOS Header Files"
2060 license cr_Sun license=cr_Sun
2061 license lic_CDDL license=lic_CDDL
2062 license license_in_headers license=license_in_headers
2063 license usr/src/lib/pkcs11/include/THIRDPARTYLICENSE \
2064 license=usr/src/lib/pkcs11/include/THIRDPARTYLICENSE
2065 link path=usr/include/iso/assert_iso.h target=./assert.h
2066 link path=usr/include/iso/errno_iso.h target=./errno.h
2067 link path=usr/include/iso/float_iso.h target=./float.h
2068 link path=usr/include/iso/iso646_iso.h target=./iso646.h
2069 $(sparc_ONLY)link path=usr/platform/SUNW,A70/include target=./sun4u/include
2070 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T12/include \
2071 target=./sun4u/include
2072 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T4/include \
2073 target=./sun4u/include
2074 $(sparc_ONLY)link path=usr/platform/SUNW,SPARC-Enterprise/include \
2075 target=./sun4u/include
2076 $(sparc_ONLY)link path=usr/platform/SUNW,Serverblade1/include \
2077 target=./sun4u/include
2078 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-100/include \
2079 target=./sun4u/include
2080 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1000/include \
2081 target=./sun4u/include
2082 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1500/include \
2083 target=./sun4u/include
2084 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-2500/include \
2085 target=./sun4u/include
2086 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-15000/include \
2087 target=./sun4u/include
2088 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-280R/include \
2089 target=./sun4u/include
2090 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-480R/include \
2091 target=./sun4u/include
2092 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-880/include \
2093 target=./sun4u/include
2094 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V215/include \
2095 target=./sun4u/include
2096 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V24t/include \
2097 target=./sun4u/include
2098 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V250/include \
2099 target=./sun4u/include
2100 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V440/include \

```



```
2101     target=../sun4u/include
2102 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V445/include \
2103     target=../sun4u/include
2104 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V490/include \
2105     target=../sun4u/include
2106 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V890/include \
2107     target=../sun4u/include
2108 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire/include \
2109     target=../sun4u/include
2110 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-2/include \
2111     target=../sun4u/include
2112 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-250/include \
2113     target=../sun4u/include
2114 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-4/include \
2115     target=../sun4u/include
2116 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise-10000/include \
2117     target=../sun4u/include
2118 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise/include \
2119     target=../sun4u/include
2120 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-Netract-40/include \
2121     target=../sun4u/include
2122 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-Netract-60/include \
2123     target=../sun4u/include
2124 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-III-Netract/include \
2125     target=../sun4u/include
2126 $(i386_ONLY)link path=usr/share/src/uts/i86pc/sys \
2127     target=../../platform/i86pc/include/sys
2128 $(i386_ONLY)link path=usr/share/src/uts/i86pc/vm \
2129     target=../../platform/i86pc/include/vm
2130 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/sys \
2131     target=../../platform/i86xpv/include/sys
2132 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/vm \
2133     target=../../platform/i86xpv/include/vm
2134 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/sys \
2135     target=../../platform/sun4u/include/sys
2136 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/vm \
2137     target=../../platform/sun4u/include/vm
2138 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/sys \
2139     target=../../platform/sun4v/include/sys
2140 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/vm \
2141     target=../../platform/sun4v/include/vm
```

new/usr/src/uts/common/fs/zfs/dmu_objset.c

1

```
*****
45215 Thu Jun 28 15:57:15 2012
new/usr/src/uts/common/fs/zfs/dmu_objset.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* !codereview */
25 */
27 /* Portions Copyright 2010 Robert Milkowski */
29 #include <sys/cred.h>
30 #include <sys/zfs_context.h>
31 #include <sys/dmu_objset.h>
32 #include <sys/dsl_dir.h>
33 #include <sys/dsl_dataset.h>
34 #include <sys/dsl_prop.h>
35 #include <sys/dsl_pool.h>
36 #include <sys/dsl_synctask.h>
37 #include <sys/dsl_deleg.h>
38 #include <sys/dnode.h>
39 #include <sys/dbuf.h>
40 #include <sys/zvol.h>
41 #include <sys/dmu_tx.h>
42 #include <sys/zap.h>
43 #include <sys/zil.h>
44 #include <sys/dmu_impl.h>
45 #include <sys/zfs_ioctl.h>
46 #include <sys/sa.h>
47 #include <sys/zfs_onexit.h>
49 /*
50 * Needed to close a window in dnode_move() that allows the objset to be freed
51 * before it can be safely accessed.
52 */
53 krwlock_t os_lock;
```

new/usr/src/uts/common/fs/zfs/dmu_objset.c

2

```
55 void
56 dmu_objset_init(void)
57 {
58     rw_init(&os_lock, NULL, RW_DEFAULT, NULL);
59 }
61 void
62 dmu_objset_fini(void)
63 {
64     rw_destroy(&os_lock);
65 }
67 spa_t *
68 dmu_objset_spa(objset_t *os)
69 {
70     return (os->os_spa);
71 }
73 zillog_t *
74 dmu_objset_zil(objset_t *os)
75 {
76     return (os->os_zil);
77 }
79 dsl_pool_t *
80 dmu_objset_pool(objset_t *os)
81 {
82     dsl_dataset_t *ds;
84     if ((ds = os->os_dsl_dataset) != NULL && ds->ds_dir)
85         return (ds->ds_dir->dd_pool);
86     else
87         return (spa_get_dsl(os->os_spa));
88 }
90 dsl_dataset_t *
91 dmu_objset_ds(objset_t *os)
92 {
93     return (os->os_dsl_dataset);
94 }
96 dmu_objset_type_t
97 dmu_objset_type(objset_t *os)
98 {
99     return (os->os_phys->os_type);
100 }
102 void
103 dmu_objset_name(objset_t *os, char *buf)
104 {
105     dsl_dataset_name(os->os_dsl_dataset, buf);
106 }
108 uint64_t
109 dmu_objset_id(objset_t *os)
110 {
111     dsl_dataset_t *ds = os->os_dsl_dataset;
113     return (ds ? ds->ds_object : 0);
114 }
116 uint64_t
117 dmu_objset_syncprop(objset_t *os)
118 {
119     return (os->os_sync);
120 }
```

```

122 uint64_t
123 dmu_objset_logbias(objset_t *os)
124 {
125     return (os->os_logbias);
126 }

128 static void
129 checksum_changed_cb(void *arg, uint64_t newval)
130 {
131     objset_t *os = arg;

133     /*
134      * Inheritance should have been done by now.
135      */
136     ASSERT(newval != ZIO_CHECKSUM_INHERIT);

138     os->os_checksum = zio_checksum_select(newval, ZIO_CHECKSUM_ON_VALUE);
139 }

141 static void
142 compression_changed_cb(void *arg, uint64_t newval)
143 {
144     objset_t *os = arg;

146     /*
147      * Inheritance and range checking should have been done by now.
148      */
149     ASSERT(newval != ZIO_COMPRESS_INHERIT);

151     os->os_compress = zio_compress_select(newval, ZIO_COMPRESS_ON_VALUE);
152 }

154 static void
155 copies_changed_cb(void *arg, uint64_t newval)
156 {
157     objset_t *os = arg;

159     /*
160      * Inheritance and range checking should have been done by now.
161      */
162     ASSERT(newval > 0);
163     ASSERT(newval <= spa_max_replication(os->os_spa));

165     os->os_copies = newval;
166 }

168 static void
169 dedup_changed_cb(void *arg, uint64_t newval)
170 {
171     objset_t *os = arg;
172     spa_t *spa = os->os_spa;
173     enum zio_checksum checksum;

175     /*
176      * Inheritance should have been done by now.
177      */
178     ASSERT(newval != ZIO_CHECKSUM_INHERIT);

180     checksum = zio_checksum_dedup_select(spa, newval, ZIO_CHECKSUM_OFF);

182     os->os_dedup_checksum = checksum & ZIO_CHECKSUM_MASK;
183     os->os_dedup_verify = !!(checksum & ZIO_CHECKSUM_VERIFY);
184 }

186 static void

```

```

187 primary_cache_changed_cb(void *arg, uint64_t newval)
188 {
189     objset_t *os = arg;

191     /*
192      * Inheritance and range checking should have been done by now.
193      */
194     ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
195            newval == ZFS_CACHE_METADATA);

197     os->os_primary_cache = newval;
198 }

200 static void
201 secondary_cache_changed_cb(void *arg, uint64_t newval)
202 {
203     objset_t *os = arg;

205     /*
206      * Inheritance and range checking should have been done by now.
207      */
208     ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
209            newval == ZFS_CACHE_METADATA);

211     os->os_secondary_cache = newval;
212 }

214 static void
215 sync_changed_cb(void *arg, uint64_t newval)
216 {
217     objset_t *os = arg;

219     /*
220      * Inheritance and range checking should have been done by now.
221      */
222     ASSERT(newval == ZFS_SYNC_STANDARD || newval == ZFS_SYNC_ALWAYS ||
223            newval == ZFS_SYNC_DISABLED);

225     os->os_sync = newval;
226     if (os->os_zil)
227         zil_set_sync(os->os_zil, newval);
228 }

230 static void
231 logbias_changed_cb(void *arg, uint64_t newval)
232 {
233     objset_t *os = arg;

235     ASSERT(newval == ZFS_LOGBIAS_LATENCY ||
236            newval == ZFS_LOGBIAS_THROUGHPUT);
237     os->os_logbias = newval;
238     if (os->os_zil)
239         zil_set_logbias(os->os_zil, newval);
240 }

242 void
243 dmu_objset_byteswap(void *buf, size_t size)
244 {
245     objset_phys_t *osp = buf;

247     ASSERT(size == OBJSET_OLD_PHYS_SIZE || size == sizeof (objset_phys_t));
248     dnode_byteswap(&osp->os_meta_dnode);
249     byteswap_uint64_array(&osp->os_zil_header, sizeof (zil_header_t));
250     osp->os_type = BSWAP_64(osp->os_type);
251     osp->os_flags = BSWAP_64(osp->os_flags);
252     if (size == sizeof (objset_phys_t)) {

```

```

253         dnode_byteswap(&osp->os_userused_dnode);
254         dnode_byteswap(&osp->os_groupused_dnode);
255     }
256 }

258 int
259 dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
260     objset_t **osp)
261 {
262     objset_t *os;
263     int i, err;

265     ASSERT(ds == NULL || MUTEX_HELD(&ds->ds_opening_lock));

267     os = kmem_zalloc(sizeof (objset_t), KM_SLEEP);
268     os->os_dsl_dataset = ds;
269     os->os_spa = spa;
270     os->os_rootbp = bp;
271     if (!BP_IS_HOLE(os->os_rootbp)) {
272         uint32_t aflags = ARC_WAIT;
273         zbookmark_t zb;
274         SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
275             ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);

277         if (DMU_OS_IS_L2CACHEABLE(os))
278             aflags |= ARC_L2CACHE;

280         dprintf_bp(os->os_rootbp, "reading %s", "");
281         /*
282          * XXX when bprewrite scrub can change the bp,
283          * and this is called from dmu_objset_open_ds_os, the bp
284          * could change, and we'll need a lock.
285          */
286         err = dsl_read_nolock(NULL, spa, os->os_rootbp,
287             arc_getbuf_func, &os->os_phys_buf,
288             ZIO_PRIORITY_SYNC_READ, ZIO_FLAG_CANFAIL, &aflags, &zb);
289         if (err) {
290             kmem_free(os, sizeof (objset_t));
291             /* convert checksum errors into IO errors */
292             if (err == ECKSUM)
293                 err = EIO;
294             return (err);
295         }

297         /* Increase the blocksize if we are permitted. */
298         if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
299             arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
300             arc_buf_t *buf = arc_buf_alloc(spa,
301                 sizeof (objset_phys_t), &os->os_phys_buf,
302                 ARC_BUFC_METADATA);
303             bzero(buf->b_data, sizeof (objset_phys_t));
304             bcopy(os->os_phys_buf->b_data, buf->b_data,
305                 arc_buf_size(os->os_phys_buf));
306             (void) arc_buf_remove_ref(os->os_phys_buf,
307                 &os->os_phys_buf);
308             os->os_phys_buf = buf;
309         }

311         os->os_phys = os->os_phys_buf->b_data;
312         os->os_flags = os->os_phys->os_flags;
313     } else {
314         int size = spa_version(spa) >= SPA_VERSION_USERSPACE ?
315             sizeof (objset_phys_t) : OBJSET_OLD_PHYS_SIZE;
316         os->os_phys_buf = arc_buf_alloc(spa, size,
317             &os->os_phys_buf, ARC_BUFC_METADATA);
318         os->os_phys = os->os_phys_buf->b_data;

```

```

319         bzero(os->os_phys, size);
320     }

322     /*
323     * Note: the changed_cb will be called once before the register
324     * func returns, thus changing the checksum/compression from the
325     * default (fletcher2/off). Snapshots don't need to know about
326     * checksum/compression/copies.
327     */
328     if (ds) {
329         err = dsl_prop_register(ds, "primarycache",
330             primary_cache_changed_cb, os);
331         if (err == 0)
332             err = dsl_prop_register(ds, "secondarycache",
333                 secondary_cache_changed_cb, os);
334         if (!dsl_dataset_is_snapshot(ds)) {
335             if (err == 0)
336                 err = dsl_prop_register(ds, "checksum",
337                     checksum_changed_cb, os);
338             if (err == 0)
339                 err = dsl_prop_register(ds, "compression",
340                     compression_changed_cb, os);
341             if (err == 0)
342                 err = dsl_prop_register(ds, "copies",
343                     copies_changed_cb, os);
344             if (err == 0)
345                 err = dsl_prop_register(ds, "dedup",
346                     dedup_changed_cb, os);
347             if (err == 0)
348                 err = dsl_prop_register(ds, "logbias",
349                     logbias_changed_cb, os);
350             if (err == 0)
351                 err = dsl_prop_register(ds, "sync",
352                     sync_changed_cb, os);
353         }
354         if (err) {
355             VERIFY(arc_buf_remove_ref(os->os_phys_buf,
356                 &os->os_phys_buf) == 1);
357             kmem_free(os, sizeof (objset_t));
358             return (err);
359         }
360     } else if (ds == NULL) {
361         /* It's the meta-objset. */
362         os->os_checksum = ZIO_CHECKSUM_FLETCHER_4;
363         os->os_compress = ZIO_COMPRESS_LZJB;
364         os->os_copies = spa_max_replication(spa);
365         os->os_dedup_checksum = ZIO_CHECKSUM_OFF;
366         os->os_dedup_verify = 0;
367         os->os_logbias = 0;
368         os->os_sync = 0;
369         os->os_primary_cache = ZFS_CACHE_ALL;
370         os->os_secondary_cache = ZFS_CACHE_ALL;
371     }

373     if (ds == NULL || !dsl_dataset_is_snapshot(ds))
374         os->os_zil_header = os->os_phys->os_zil_header;
375     os->os_zil = zil_alloc(os, &os->os_zil_header);

377     for (i = 0; i < TXG_SIZE; i++) {
378         list_create(&os->os_dirty_dnodes[i], sizeof (dnode_t),
379             offsetof(dnode_t, dn_dirty_link[i]));
380         list_create(&os->os_free_dnodes[i], sizeof (dnode_t),
381             offsetof(dnode_t, dn_dirty_link[i]));
382     }
383     list_create(&os->os_dnodes, sizeof (dnode_t),
384         offsetof(dnode_t, dn_link));

```

```

385 list_create(&os->os_downgraded_dbufs, sizeof (dmu_buf_impl_t),
386             offsetof(dmu_buf_impl_t, db_link));
388 mutex_init(&os->os_lock, NULL, MUTEX_DEFAULT, NULL);
389 mutex_init(&os->os_obj_lock, NULL, MUTEX_DEFAULT, NULL);
390 mutex_init(&os->os_user_ptr_lock, NULL, MUTEX_DEFAULT, NULL);
392 DMU_META_DNODE(os) = dnode_special_open(os,
393     &os->os_phys->os_meta_dnode, DMU_META_DNODE_OBJECT,
394     &os->os_meta_dnode);
395 if (arc_buf_size(os->os_phys_buf) >= sizeof (objset_phys_t)) {
396     DMU_USERUSED_DNODE(os) = dnode_special_open(os,
397         &os->os_phys->os_userused_dnode, DMU_USERUSED_OBJECT,
398         &os->os_userused_dnode);
399     DMU_GROUPUSED_DNODE(os) = dnode_special_open(os,
400         &os->os_phys->os_groupused_dnode, DMU_GROUPUSED_OBJECT,
401         &os->os_groupused_dnode);
402 }
404 /*
405  * We should be the only thread trying to do this because we
406  * have ds_opening_lock
407  */
408 if (ds) {
409     mutex_enter(&ds->ds_lock);
410     ASSERT(ds->ds_objset == NULL);
411     ds->ds_objset = os;
412     mutex_exit(&ds->ds_lock);
413 }
415 *osp = os;
416 return (0);
417 }
419 int
420 dmu_objset_from_ds(dsl_dataset_t *ds, objset_t **osp)
421 {
422     int err = 0;
424     mutex_enter(&ds->ds_opening_lock);
425     *osp = ds->ds_objset;
426     if (*osp == NULL) {
427         err = dmu_objset_open_impl(dsl_dataset_get_spa(ds),
428             ds, dsl_dataset_get_blkptr(ds), osp);
429     }
430     mutex_exit(&ds->ds_opening_lock);
431     return (err);
432 }
434 /* called from zpl */
435 int
436 dmu_objset_hold(const char *name, void *tag, objset_t **osp)
437 {
438     dsl_dataset_t *ds;
439     int err;
441     err = dsl_dataset_hold(name, tag, &ds);
442     if (err)
443         return (err);
445     err = dmu_objset_from_ds(ds, osp);
446     if (err)
447         dsl_dataset_rele(ds, tag);
449     return (err);
450 }

```

```

452 /* called from zpl */
453 int
454 dmu_objset_own(const char *name, dmu_objset_type_t type,
455               boolean_t readonly, void *tag, objset_t **osp)
456 {
457     dsl_dataset_t *ds;
458     int err;
460     err = dsl_dataset_own(name, B_FALSE, tag, &ds);
461     if (err)
462         return (err);
464     err = dmu_objset_from_ds(ds, osp);
465     if (err) {
466         dsl_dataset_disown(ds, tag);
467     } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
468         dmu_objset_disown(*osp, tag);
469         return (EINVAL);
470     } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
471         dmu_objset_disown(*osp, tag);
472         return (EROFS);
473     }
474     return (err);
475 }
477 void
478 dmu_objset_rele(objset_t *os, void *tag)
479 {
480     dsl_dataset_rele(os->os_dsl_dataset, tag);
481 }
483 void
484 dmu_objset_disown(objset_t *os, void *tag)
485 {
486     dsl_dataset_disown(os->os_dsl_dataset, tag);
487 }
489 int
490 dmu_objset_evict_dbufs(objset_t *os)
491 {
492     dnode_t *dn;
494     mutex_enter(&os->os_lock);
496     /* process the mdn last, since the other dnodes have holds on it */
497     list_remove(&os->os_dnodes, DMU_META_DNODE(os));
498     list_insert_tail(&os->os_dnodes, DMU_META_DNODE(os));
500     /*
501      * Find the first dnode with holds. We have to do this dance
502      * because dnode_add_ref() only works if you already have a
503      * hold. If there are no holds then it has no dbufs so OK to
504      * skip.
505      */
506     for (dn = list_head(&os->os_dnodes);
507          dn && !dnode_add_ref(dn, FTAG);
508          dn = list_next(&os->os_dnodes, dn))
509         continue;
511     while (dn) {
512         dnode_t *next_dn = dn;
514         do {
515             next_dn = list_next(&os->os_dnodes, next_dn);
516         } while (next_dn && !dnode_add_ref(next_dn, FTAG));

```

```

518         mutex_exit(&os->os_lock);
519         dnode_evict_dbufs(dn);
520         dnode_rele(dn, FTAG);
521         mutex_enter(&os->os_lock);
522         dn = next_dn;
523     }
524     dn = list_head(&os->os_dnodes);
525     mutex_exit(&os->os_lock);
526     return (dn != DMU_META_DNODE(os));
527 }

529 void
530 dmu_objset_evict(objset_t *os)
531 {
532     dsl_dataset_t *ds = os->os_dsl_dataset;

534     for (int t = 0; t < TXG_SIZE; t++)
535         ASSERT(!dmu_objset_is_dirty(os, t));

537     if (ds) {
538         if (!dsl_dataset_is_snapshot(ds)) {
539             VERIFY(0 == dsl_prop_unregister(ds, "checksum",
540                 checksum_changed_cb, os));
541             VERIFY(0 == dsl_prop_unregister(ds, "compression",
542                 compression_changed_cb, os));
543             VERIFY(0 == dsl_prop_unregister(ds, "copies",
544                 copies_changed_cb, os));
545             VERIFY(0 == dsl_prop_unregister(ds, "dedup",
546                 dedup_changed_cb, os));
547             VERIFY(0 == dsl_prop_unregister(ds, "logbias",
548                 logbias_changed_cb, os));
549             VERIFY(0 == dsl_prop_unregister(ds, "sync",
550                 sync_changed_cb, os));
551         }
552         VERIFY(0 == dsl_prop_unregister(ds, "primarycache",
553             primary_cache_changed_cb, os));
554         VERIFY(0 == dsl_prop_unregister(ds, "secondarycache",
555             secondary_cache_changed_cb, os));
556     }

558     if (os->os_sa)
559         sa_tear_down(os);

561     /*
562      * We should need only a single pass over the dnode list, since
563      * nothing can be added to the list at this point.
564      */
565     (void) dmu_objset_evict_dbufs(os);

567     dnode_special_close(&os->os_meta_dnode);
568     if (DMU_USERUSED_DNODE(os)) {
569         dnode_special_close(&os->os_userused_dnode);
570         dnode_special_close(&os->os_groupused_dnode);
571     }
572     zil_free(os->os_zil);

574     ASSERT3P(list_head(&os->os_dnodes), ==, NULL);

576     VERIFY(arc_buf_remove_ref(os->os_phys_buf, &os->os_phys_buf) == 1);

578     /*
579      * This is a barrier to prevent the objset from going away in
580      * dnode_move() until we can safely ensure that the objset is still in
581      * use. We consider the objset valid before the barrier and invalid
582      * after the barrier.

```

```

583     /*
584     rw_enter(&os_lock, RW_READER);
585     rw_exit(&os_lock);

587     mutex_destroy(&os->os_lock);
588     mutex_destroy(&os->os_obj_lock);
589     mutex_destroy(&os->os_user_ptr_lock);
590     kmem_free(os, sizeof (objset_t));
591     }

593 timestruc_t
594 dmu_objset_snap_cmtime(objset_t *os)
595 {
596     return (dsl_dir_snap_cmtime(os->os_dsl_dataset->ds_dir));
597 }

599 /* called from dsl for meta-objset */
600 objset_t *
601 dmu_objset_create_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
602     dmu_objset_type_t type, dmu_tx_t *tx)
603 {
604     objset_t *os;
605     dnode_t *mdn;

607     ASSERT(dmu_tx_is_syncing(tx));
608     if (ds != NULL)
609         VERIFY(0 == dmu_objset_from_ds(ds, &os));
610     else
611         VERIFY(0 == dmu_objset_open_impl(spa, NULL, bp, &os));

613     mdn = DMU_META_DNODE(os);

615     dnode_allocate(mdn, DMU_OT_DNODE, 1 << DNODE_BLOCK_SHIFT,
616         DN_MAX_INDBLKSHIFT, DMU_OT_NONE, 0, tx);

618     /*
619      * We don't want to have to increase the meta-dnode's nlevels
620      * later, because then we could do it in quiescing context while
621      * we are also accessing it in open context.
622      *
623      * This precaution is not necessary for the MOS (ds == NULL),
624      * because the MOS is only updated in syncing context.
625      * This is most fortunate: the MOS is the only objset that
626      * needs to be synced multiple times as spa_sync() iterates
627      * to convergence, so minimizing its dn_nlevels matters.
628      */
629     if (ds != NULL) {
630         int levels = 1;

632         /*
633          * Determine the number of levels necessary for the meta-dnode
634          * to contain DN_MAX_OBJECT dnodes.
635          */
636         while ((uint64_t)mdn->dn_nblkptr << (mdn->dn_datablkshift +
637             (levels - 1) * (mdn->dn_indblkshift - SPA_BLKPTRSHIFT)) <
638             DN_MAX_OBJECT * sizeof (dnode_phys_t))
639             levels++;

641         mdn->dn_next_nlevels[tx->tx_txg & TXG_MASK] =
642             mdn->dn_nlevels = levels;
643     }

645     ASSERT(type != DMU_OST_NONE);
646     ASSERT(type != DMU_OST_ANY);
647     ASSERT(type < DMU_OST_NUMTYPES);
648     os->os_phys->os_type = type;

```

```

649     if (dmu_objset_userused_enabled(os)) {
650         os->os_phys->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
651         os->os_flags = os->os_phys->os_flags;
652     }

654     dsl_dataset_dirty(ds, tx);

656     return (os);
657 }

659 struct oscarg {
660     void (*userfunc)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx);
661     void *userarg;
662     dsl_dataset_t *clone_origin;
663     const char *lastname;
664     dmu_objset_type_t type;
665     uint64_t flags;
666     cred_t *cr;
667 };

669 /*ARGSUSED*/
670 static int
671 dmu_objset_create_check(void *arg1, void *arg2, dmu_tx_t *tx)
672 {
673     dsl_dir_t *dd = arg1;
674     struct oscarg *oa = arg2;
675     objset_t *mos = dd->dd_pool->dp_meta_objset;
676     int err;
677     uint64_t ddoobj;

679     err = zap_lookup(mos, dd->dd_phys->dd_child_dir_zapobj,
680         oa->lastname, sizeof(uint64_t), 1, &ddobj);
681     if (err != ENOENT)
682         return (err ? err : EEXIST);

684     if (oa->clone_origin != NULL) {
685         /* You can't clone across pools. */
686         if (oa->clone_origin->ds_dir->dd_pool != dd->dd_pool)
687             return (EXDEV);

689         /* You can only clone snapshots, not the head datasets. */
690         if (!dsl_dataset_is_snapshot(oa->clone_origin))
691             return (EINVAL);
692     }

694     return (0);
695 }

697 static void
698 dmu_objset_create_sync(void *arg1, void *arg2, dmu_tx_t *tx)
699 {
700     dsl_dir_t *dd = arg1;
701     spa_t *spa = dd->dd_pool->dp_spa;
702     struct oscarg *oa = arg2;
703     uint64_t obj;
704     dsl_dataset_t *ds;
705     blkptr_t *bp;
706 #endif /* !codereview */

708     ASSERT(dmu_tx_is_syncing(tx));

710     obj = dsl_dataset_create_sync(dd, oa->lastname,
711         oa->clone_origin, oa->flags, oa->cr, tx);

713     VERIFY3U(0, ==, dsl_dataset_hold_obj(dd->dd_pool, obj, FTAG, &ds));
23     if (oa->clone_origin == NULL) {

```

```

24         dsl_pool_t *dp = dd->dd_pool;
25         dsl_dataset_t *ds;
26         blkptr_t *bp;
27         objset_t *os;

29         VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, obj, FTAG, &ds));
714     bp = dsl_dataset_get_blkptr(ds);
715     if (BP_IS_HOLE(bp)) {
716         objset_t *os =
717             dmu_objset_create_impl(spa, ds, bp, oa->type, tx);
31         ASSERT(BP_IS_HOLE(bp));

33         os = dmu_objset_create_impl(spa, ds, bp, oa->type, tx);

719         if (oa->userfunc)
720             oa->userfunc(os, oa->userarg, oa->cr, tx);
37         dsl_dataset_rele(ds, FTAG);
721     }

723     if (oa->clone_origin == NULL) {
724         spa_history_log_internal_ds(ds, "create", tx, "");
725     } else {
726         char namebuf[MAXNAMELEN];
727         dsl_dataset_name(oa->clone_origin, namebuf);
728         spa_history_log_internal_ds(ds, "clone", tx,
729             "origin=%s (%llu)", namebuf, oa->clone_origin->ds_object);
730     }
731     dsl_dataset_rele(ds, FTAG);
40     spa_history_log_internal(LOG_DS_CREATE, spa, tx, "dataset = %llu", obj);
732 }

unchanged_portion_omitted

808 typedef struct snapallarg {
809     dsl_sync_task_group_t *saa_dstg;
810     boolean_t saa_needsuspend;
811     nvlist_t *saa_props;

813     /* the following are used only if 'temporary' is set: */
814     boolean_t saa_temporary;
815     const char *saa_htag;
816     struct dsl_ds_holdarg *saa_ha;
817     dsl_dataset_t *saa_newds;
818 } snapallarg_t;

820 typedef struct snaponearg {
821     const char *soa_longname; /* long snap name */
822     const char *soa_snapname; /* short snap name */
823     snapallarg_t *soa_saa;
824 } snaponearg_t;
117 struct snaparg {
118     dsl_sync_task_group_t *dstg;
119     char *snapname;
120     char *htag;
121     char failed[MAXPATHLEN];
122     boolean_t recursive;
123     boolean_t needsuspend;
124     boolean_t temporary;
125     nvlist_t *props;
126     struct dsl_ds_holdarg *ha; /* only needed in the temporary case */
127     dsl_dataset_t *newds;
128 };

826 static int
827 snapshot_check(void *arg1, void *arg2, dmu_tx_t *tx)
828 {
829     objset_t *os = arg1;

```

```

830     snaponearg_t *soa = arg2;
831     snapallarg_t *saa = soa->soa_saa;
134     struct snaparg *sn = arg2;
832     int error;

834     /* The props have already been checked by zfs_check_userprops(). */

836     error = dsl_dataset_snapshot_check(os->os_dsl_dataset,
837     soa->soa_snapname, tx);
140     sn->snapname, tx);
838     if (error)
839         return (error);

841     if (saa->saa_temporary) {
144     if (sn->temporary) {
842         /*
843          * Ideally we would just call
844          * dsl_dataset_user_hold_check() and
845          * dsl_dataset_destroy_check() here. However the
846          * dataset we want to hold and destroy is the snapshot
847          * that we just confirmed we can create, but it won't
848          * exist until after these checks are run. Do any
849          * checks we can here and if more checks are added to
850          * those routines in the future, similar checks may be
851          * necessary here.
852          */
853         if (spa_version(os->os_spa) < SPA_VERSION_USERREFS)
854             return (ENOTSUP);
855         /*
856          * Not checking number of tags because the tag will be
857          * unique, as it will be the only tag.
858          */
859         if (strlen(saa->saa_htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
162         if (strlen(sn->htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
860             return (E2BIG);

862         saa->saa_ha = kmem_alloc(sizeof (struct dsl_ds_holdarg),
863         KM_SLEEP);
864         saa->saa_ha->temphold = B_TRUE;
865         saa->saa_ha->htag = saa->saa_htag;
165         sn->ha = kmem_alloc(sizeof (struct dsl_ds_holdarg), KM_SLEEP);
166         sn->ha->temphold = B_TRUE;
167         sn->ha->htag = sn->htag;
866     }
867     return (error);
868 }

870 static void
871 snapshot_sync(void *arg1, void *arg2, dmu_tx_t *tx)
872 {
873     objset_t *os = arg1;
874     dsl_dataset_t *ds = os->os_dsl_dataset;
875     snaponearg_t *soa = arg2;
876     snapallarg_t *saa = soa->soa_saa;
177     struct snaparg *sn = arg2;

878     dsl_dataset_snapshot_sync(ds, soa->soa_snapname, tx);
179     dsl_dataset_snapshot_sync(ds, sn->snapname, tx);

880     if (saa->saa_props != NULL) {
181         if (sn->props) {
881             dsl_props_arg_t pa;
882             pa.pa_props = saa->saa_props;
183             pa.pa_props = sn->props;
883             pa.pa_source = ZPROP_SRC_LOCAL;
884             dsl_props_set_sync(ds->ds_prev, &pa, tx);

```

```

885     }

887     if (saa->saa_temporary) {
188     if (sn->temporary) {
888         struct dsl_ds_destroyarg da;

890         dsl_dataset_user_hold_sync(ds->ds_prev, saa->saa_ha, tx);
891         kmem_free(saa->saa_ha, sizeof (struct dsl_ds_holdarg));
892         saa->saa_ha = NULL;
893         saa->saa_newds = ds->ds_prev;
191         dsl_dataset_user_hold_sync(ds->ds_prev, sn->ha, tx);
192         kmem_free(sn->ha, sizeof (struct dsl_ds_holdarg));
193         sn->ha = NULL;
194         sn->newds = ds->ds_prev;

895         da.ds = ds->ds_prev;
896         da.defer = B_TRUE;
897         dsl_dataset_destroy_sync(&da, FTAG, tx);
898     }
899 }

901 static int
902 snapshot_one_impl(const char *snapname, void *arg)
203 dmu_objset_snapshot_one(const char *name, void *arg)
903 {
904     char fsname[MAXPATHLEN];
905     snapallarg_t *saa = arg;
906     snaponearg_t *soa;
205     struct snaparg *sn = arg;
907     objset_t *os;
908     int err;
208     char *cp;

210     /*
211      * If the objset starts with a '%', then ignore it unless it was
212      * explicitly named (ie, not recursive). These hidden datasets
213      * are always inconsistent, and by not opening them here, we can
214      * avoid a race with dsl_dir_destroy_check().
215      */
216     cp = strchr(name, '/');
217     if (cp && cp[1] == '%' && sn->recursive)
218         return (0);

910     (void) strncpy(fsname, snapname, sizeof (fsname));
911     strchr(fsname, '@')[0] = '\0';
220     (void) strcpy(sn->failed, name);

913     err = dmu_objset_hold(fsname, saa, &os);
222     /*
223      * Check permissions if we are doing a recursive snapshot. The
224      * permission checks for the starting dataset have already been
225      * performed in zfs_secpolicy_snapshot()
226      */
227     if (sn->recursive && (err = zfs_secpolicy_snapshot_perms(name, CRED())))
228         return (err);

230     err = dmu_objset_hold(name, sn, &os);
914     if (err != 0)
915         return (err);

917     /*
918      * If the objset is in an inconsistent state (eg, in the process
919      * of being destroyed), don't snapshot it.
236     * of being destroyed), don't snapshot it. As with %hidden
237     * datasets, we return EBUSY if this name was explicitly
238     * requested (ie, not recursive), and otherwise ignore it.

```



```

920     */
921     if (os->os_dsl_dataset->ds_phys->ds_flags & DS_FLAG_INCONSISTENT) {
922         dmu_objset_rele(os, saa);
923         return (EBUSY);
924     }
925
926     if (saa->saa_needsuspend) {
927         if (sn->needsuspend) {
928             err = zil_suspend(dmu_objset_zil(os));
929             if (err) {
930                 dmu_objset_rele(os, saa);
931                 dmu_objset_rele(os, sn);
932                 return (err);
933             }
934         }
935         dsl_sync_task_create(sn->dstg, snapshot_check, snapshot_sync,
936             os, sn, 3);
937
938         soa = kmem_zalloc(sizeof (*soa), KM_SLEEP);
939         soa->soa_saa = saa;
940         soa->soa_longname = snapname;
941         soa->soa_snapname = strchr(snapname, '@') + 1;
942
943         dsl_sync_task_create(saa->saa_dstg, snapshot_check, snapshot_sync,
944             os, soa, 3);
945
946         return (0);
947     }
948
949     /*
950     * The snapshots must all be in the same pool.
951     */
952     int
953     dmu_objset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors)
954     {
955         dsl_sync_task_t *dst;
956         snapallarg_t saa = { 0 };
957         spa_t *spa;
958         int rv = 0;
959         int err;
960         nvpair_t *pair;
961
962         pair = nvlist_next_nvpair(snaps, NULL);
963         if (pair == NULL)
964             #endif /* ! codereview */
965             return (0);
966
967         err = spa_open(nvpair_name(pair), &spa, FTAG);
968         if (err)
969             return (err);
970         saa.saa_dstg = dsl_sync_task_group_create(spa_get_dsl(spa));
971         saa.saa_props = props;
972         saa.saa_needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
973
974         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
975             pair = nvlist_next_nvpair(snaps, pair)) {
976             err = snapshot_one_impl(nvpair_name(pair), &saa);
977             if (err != 0) {
978                 if (errors != NULL) {
979                     fnvlist_add_int32(errors,
980                         nvpair_name(pair), err);
981                 }
982                 rv = err;
983             }
984         }
985     }

```

```

986     }
987     /*
988     * If any call to snapshot_one_impl() failed, don't execute the
989     * sync task. The error handling code below will clean up the
990     * snaponearg_t from any successful calls to
991     * snapshot_one_impl().
992     */
993     if (rv == 0)
994         err = dsl_sync_task_group_wait(saa.saa_dstg);
995     if (err != 0)
996         rv = err;
997
998     for (dst = list_head(&saa.saa_dstg->dstg_tasks); dst;
999         dst = list_next(&saa.saa_dstg->dstg_tasks, dst)) {
1000         objset_t *os = dst->dst_arg1;
1001         snaponearg_t *soa = dst->dst_arg2;
1002         if (dst->dst_err != 0) {
1003             if (errors != NULL) {
1004                 fnvlist_add_int32(errors,
1005                     soa->soa_longname, dst->dst_err);
1006             }
1007             rv = dst->dst_err;
1008         }
1009
1010         if (saa.saa_needsuspend)
1011             zil_resume(dmu_objset_zil(os));
1012         dmu_objset_rele(os, &saa);
1013         kmem_free(soa, sizeof (*soa));
1014     }
1015
1016     dsl_sync_task_group_destroy(saa.saa_dstg);
1017     spa_close(spa, FTAG);
1018     return (rv);
1019 }
1020
1021 int
1022 dmu_objset_snapshot_one(const char *fsname, const char *snapname)
1023 {
1024     int err;
1025     char *longsnap = kmem_asprintf("%s@%s", fsname, snapname);
1026     nvlist_t *snaps = fnvlist_alloc();
1027
1028     fnvlist_add_boolean(snaps, longsnap);
1029     err = dmu_objset_snapshot(snaps, NULL, NULL);
1030     fnvlist_free(snaps);
1031     strfree(longsnap);
1032     return (err);
1033 }
1034
1035 #endif /* ! codereview */
1036
1037 int
1038 dmu_objset_snapshot_tmp(const char *snapname, const char *tag, int cleanup_fd)
1039 {
1040     dmu_objset_snapshot(char *fsname, char *snapname, char *tag,
1041         nvlist_t *props, boolean_t recursive, boolean_t temporary, int cleanup_fd)
1042     {
1043         dsl_sync_task_t *dst;
1044         snapallarg_t saa = { 0 };
1045         struct snaparg sn;
1046         spa_t *spa;
1047         minor_t minor;
1048         int err;
1049
1050         err = spa_open(snapname, &spa, FTAG);
1051         (void) strcpy(sn.failed, fsname);

```

```

266     err = spa_open(fsname, &spa, FTAG);
1041     if (err)
1042         return (err);
1043     saa.saa_dstg = dsl_sync_task_group_create(spa_get_dsl(spa));
1044     saa.saa_htag = tag;
1045     saa.saa_needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
1046     saa.saa_temporary = B_TRUE;
1047 #endif /* ! codereview */

269     if (temporary) {
1049     if (cleanup_fd < 0) {
1050         spa_close(spa, FTAG);
1051         return (EINVAL);
1052     }
1053     if ((err = zfs_onexit_fd_hold(cleanup_fd, &minor)) != 0) {
1054         spa_close(spa, FTAG);
1055         return (err);
1056     }
278 }

1058     err = snapshot_one_impl(snapname, &saa);
280     sn.dstg = dsl_sync_task_group_create(spa_get_dsl(spa));
281     sn.snapname = snapname;
282     sn.htag = tag;
283     sn.props = props;
284     sn.recursive = recursive;
285     sn.needsuspend = (spa_version(spa) < SPA_VERSION_FAST_SNAP);
286     sn.temporary = temporary;
287     sn.ha = NULL;
288     sn.newds = NULL;

290     if (recursive) {
291         err = dmu_objset_find(fsname,
292             dmu_objset_snapshot_one, &sn, DS_FIND_CHILDREN);
293     } else {
294         err = dmu_objset_snapshot_one(fsname, &sn);
295     }

1060     if (err == 0)
1061         err = dsl_sync_task_group_wait(saa.saa_dstg);
298     err = dsl_sync_task_group_wait(sn.dstg);

1063     for (dst = list_head(&saa.saa_dstg->dstg_tasks); dst;
1064         dst = list_next(&saa.saa_dstg->dstg_tasks, dst)) {
300     for (dst = list_head(&sn.dstg->dstg_tasks); dst;
301         dst = list_next(&sn.dstg->dstg_tasks, dst)) {
1065         objset_t *os = dst->dst_arg1;
1066         dsl_register_onexit_hold_cleanup(saa.saa_newds, tag, minor);
1067         if (saa.saa_needsuspend)
1068             dsl_dataset_t *ds = os->os_dsl_dataset;
303         if (dst->dst_err) {
304             dsl_dataset_name(ds, sn.failed);
305         } else if (temporary) {
306             dsl_register_onexit_hold_cleanup(sn.newds, tag, minor);
307         }
308     }
309     if (sn.needsuspend)
1068         zil_resume(dmu_objset_zil(os));
1069     dmu_objset_rele(os, &saa);
311     dmu_objset_rele(os, &sn);
1070 }

314     if (err)
315         (void) strcpy(fsname, sn.failed);
316     if (temporary)
1072     zfs_onexit_fd_rele(cleanup_fd);
1073     dsl_sync_task_group_destroy(saa.saa_dstg);

```

```

318     dsl_sync_task_group_destroy(sn.dstg);
1074     spa_close(spa, FTAG);
1075     return (err);
1076 }

1079 #endif /* ! codereview */
1080 static void
1081 dmu_objset_sync_dnodes(list_t *list, list_t *newlist, dmu_tx_t *tx)
1082 {
1083     dnode_t *dn;

1085     while (dn = list_head(list)) {
1086         ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
1087         ASSERT(dn->dn_dbuf->db_data_pending);
1088         /*
1089          * Initialize dn_zio outside dnode_sync() because the
1090          * meta-dnode needs to set it outside dnode_sync().
1091          */
1092         dn->dn_zio = dn->dn_dbuf->db_data_pending->dr_zio;
1093         ASSERT(dn->dn_zio);

1095         ASSERT3U(dn->dn_nlevels, <=, DN_MAX_LEVELS);
1096         list_remove(list, dn);

1098         if (newlist) {
1099             (void) dnode_add_ref(dn, newlist);
1100             list_insert_tail(newlist, dn);
1101         }

1103         dnode_sync(dn, tx);
1104     }
1105 }

1107 /* ARGSUSED */
1108 static void
1109 dmu_objset_write_ready(zio_t *zio, arc_buf_t *abuf, void *arg)
1110 {
1111     blkptr_t *bp = zio->io_bp;
1112     objset_t *os = arg;
1113     dnode_phys_t *dnp = &os->os_phys->os_meta_dnode;

1115     ASSERT(bp == os->os_rootbp);
1116     ASSERT(BP_GET_TYPE(bp) == DMU_OT_OBJSET);
1117     ASSERT(BP_GET_LEVEL(bp) == 0);

1119     /*
1120      * Update rootbp fill count: it should be the number of objects
1121      * allocated in the object set (not counting the "special"
1122      * objects that are stored in the objset_phys_t -- the meta
1123      * dnode and user/group accounting objects).
1124      */
1125     bp->blk_fill = 0;
1126     for (int i = 0; i < dnp->dn_nblkptr; i++)
1127         bp->blk_fill += dnp->dn_blkptr[i].blk_fill;
1128 }

1130 /* ARGSUSED */
1131 static void
1132 dmu_objset_write_done(zio_t *zio, arc_buf_t *abuf, void *arg)
1133 {
1134     blkptr_t *bp = zio->io_bp;
1135     blkptr_t *bp_orig = &zio->io_bp_orig;
1136     objset_t *os = arg;

1138     if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {

```

```

1139     ASSERT(BP_EQUAL(bp, bp_orig));
1140 } else {
1141     dsl_dataset_t *ds = os->os_dsl_dataset;
1142     dmu_tx_t *tx = os->os_synctx;
1143
1144     (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
1145     dsl_dataset_block_born(ds, bp, tx);
1146 }
1147 }
1148
1149 /* called from dsl */
1150 void
1151 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
1152 {
1153     int txgoff;
1154     zbookmark_t zb;
1155     zio_prop_t zp;
1156     zio_t *zio;
1157     list_t *list;
1158     list_t *newlist = NULL;
1159     dbuf_dirty_record_t *dr;
1160
1161     dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);
1162
1163     ASSERT(dmu_tx_is_syncing(tx));
1164     /* XXX the write_done callback should really give us the tx... */
1165     os->os_synctx = tx;
1166
1167     if (os->os_dsl_dataset == NULL) {
1168         /*
1169          * This is the MOS. If we have upgraded,
1170          * spa_max_replication() could change, so reset
1171          * os_copies here.
1172          */
1173         os->os_copies = spa_max_replication(os->os_spa);
1174     }
1175
1176     /*
1177      * Create the root block IO
1178      */
1179     SET_BOOKMARK(&zb, os->os_dsl_dataset ?
1180         os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
1181         ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1182     VERIFY3U(0, ==, arc_release_bp(os->os_phys_buf, &os->os_phys_buf,
1183         os->os_rootbp, os->os_spa, &zb));
1184
1185     dmu_write_policy(os, NULL, 0, 0, &zp);
1186
1187     zio = arc_write(pio, os->os_spa, tx->tx_txg,
1188         os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os), &zp,
1189         dmu_objset_write_ready, dmu_objset_write_done, os,
1190         ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
1191
1192     /*
1193      * Sync special dnodes - the parent IO for the sync is the root block
1194      */
1195     DMU_META_DNODE(os->dn_zio = zio;
1196     dnode_sync(DMU_META_DNODE(os), tx);
1197
1198     os->os_phys->os_flags = os->os_flags;
1199
1200     if (DMU_USERUSED_DNODE(os) &&
1201         DMU_USERUSED_DNODE(os->dn_type != DMU_OT_NONE) {
1202         DMU_USERUSED_DNODE(os->dn_zio = zio;
1203         dnode_sync(DMU_USERUSED_DNODE(os), tx);
1204         DMU_GROUPUSED_DNODE(os->dn_zio = zio;

```

```

1205     dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
1206 }
1207
1208     txgoff = tx->tx_txg & TXG_MASK;
1209
1210     if (dmu_objset_userused_enabled(os)) {
1211         newlist = &os->os_synced_dnodes;
1212         /*
1213          * We must create the list here because it uses the
1214          * dn_dirty_link[] of this txg.
1215          */
1216         list_create(newlist, sizeof (dnode_t),
1217             offsetof(dnode_t, dn_dirty_link[txgoff]));
1218     }
1219
1220     dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
1221     dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);
1222
1223     list = &DMU_META_DNODE(os->dn_dirty_records[txgoff]);
1224     while (dr = list_head(list)) {
1225         ASSERT(dr->dr_dbuf->db_level == 0);
1226         list_remove(list, dr);
1227         if (dr->dr_zio)
1228             zio_nowait(dr->dr_zio);
1229     }
1230     /*
1231      * Free intent log blocks up to this tx.
1232      */
1233     zil_sync(os->os_zil, tx);
1234     os->os_phys->os_zil_header = os->os_zil_header;
1235     zio_nowait(zio);
1236 }
1237
1238 boolean_t
1239 dmu_objset_is_dirty(objset_t *os, uint64_t txg)
1240 {
1241     return (!list_is_empty(&os->os_dirty_dnodes[txg & TXG_MASK]) ||
1242         !list_is_empty(&os->os_free_dnodes[txg & TXG_MASK]));
1243 }
1244
1245 boolean_t
1246 dmu_objset_is_dirty_anywhere(objset_t *os)
1247 {
1248     for (int t = 0; t < TXG_SIZE; t++)
1249         if (dmu_objset_is_dirty(os, t))
1250             return (B_TRUE);
1251     return (B_FALSE);
1252 }
1253
1254 static objset_used_cb_t *used_cbs[DMU_OST_NUMTYPES];
1255
1256 void
1257 dmu_objset_register_type(dmu_objset_type_t ost, objset_used_cb_t *cb)
1258 {
1259     used_cbs[ost] = cb;
1260 }
1261
1262 boolean_t
1263 dmu_objset_userused_enabled(objset_t *os)
1264 {
1265     return (spa_version(os->os_spa) >= SPA_VERSION_USERSPACE &&
1266         used_cbs[os->os_phys->os_type] != NULL &&
1267         DMU_USERUSED_DNODE(os) != NULL);
1268 }
1269
1270 static void

```

```

1271 do_userquota_update(objset_t *os, uint64_t used, uint64_t flags,
1272     uint64_t user, uint64_t group, boolean_t subtract, dmu_tx_t *tx)
1273 {
1274     if ((flags & DNODE_FLAG_USERUSED_ACCOUNTED)) {
1275         int64_t delta = DNODE_SIZE + used;
1276         if (subtract)
1277             delta = -delta;
1278         VERIFY3U(0, ==, zap_increment_int(os, DMU_USERUSED_OBJECT,
1279             user, delta, tx));
1280         VERIFY3U(0, ==, zap_increment_int(os, DMU_GROUPUSED_OBJECT,
1281             group, delta, tx));
1282     }
1283 }

1285 void
1286 dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx)
1287 {
1288     dnode_t *dn;
1289     list_t *list = &os->os_synced_dnodes;

1291     ASSERT(list_head(list) == NULL || dmu_objset_userused_enabled(os));

1293     while (dn = list_head(list)) {
1294         int flags;
1295         ASSERT(!DMU_OBJECT_IS_SPECIAL(dn->dn_object));
1296         ASSERT(dn->dn_phys->dn_type == DMU_OT_NONE ||
1297             dn->dn_phys->dn_flags &
1298             DNODE_FLAG_USERUSED_ACCOUNTED);

1300         /* Allocate the user/groupused objects if necessary. */
1301         if (DMU_USERUSED_DNODE(os)->dn_type == DMU_OT_NONE) {
1302             VERIFY(0 == zap_create_claim(os,
1303                 DMU_USERUSED_OBJECT,
1304                 DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1305             VERIFY(0 == zap_create_claim(os,
1306                 DMU_GROUPUSED_OBJECT,
1307                 DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1308         }

1310         /*
1311          * We intentionally modify the zap object even if the
1312          * net delta is zero. Otherwise
1313          * the block of the zap obj could be shared between
1314          * datasets but need to be different between them after
1315          * a bprewrite.
1316          */

1318         flags = dn->dn_id_flags;
1319         ASSERT(flags);
1320         if (flags & DN_ID_OLD_EXIST) {
1321             do_userquota_update(os, dn->dn_oldused, dn->dn_oldflags,
1322                 dn->dn_olduid, dn->dn_oldgid, B_TRUE, tx);
1323         }
1324         if (flags & DN_ID_NEW_EXIST) {
1325             do_userquota_update(os, DN_USED_BYTES(dn->dn_phys),
1326                 dn->dn_phys->dn_flags, dn->dn_newuid,
1327                 dn->dn_newgid, B_FALSE, tx);
1328         }

1330         mutex_enter(&dn->dn_mtx);
1331         dn->dn_oldused = 0;
1332         dn->dn_oldflags = 0;
1333         if (dn->dn_id_flags & DN_ID_NEW_EXIST) {
1334             dn->dn_olduid = dn->dn_newuid;
1335             dn->dn_oldgid = dn->dn_newgid;
1336             dn->dn_id_flags |= DN_ID_OLD_EXIST;

```

```

1337         if (dn->dn_bonuslen == 0)
1338             dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1339         else
1340             dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1341     }
1342     dn->dn_id_flags &= ~(DN_ID_NEW_EXIST);
1343     mutex_exit(&dn->dn_mtx);

1345     list_remove(list, dn);
1346     dnode_rele(dn, list);
1347 }
1348 }

1350 /*
1351  * Returns a pointer to data to find uid/gid from
1352  *
1353  * If a dirty record for transaction group that is syncing can't
1354  * be found then NULL is returned. In the NULL case it is assumed
1355  * the uid/gid aren't changing.
1356  */
1357 static void *
1358 dmu_objset_userquota_find_data(dmu_buf_impl_t *db, dmu_tx_t *tx)
1359 {
1360     dbuf_dirty_record_t *dr, **drp;
1361     void *data;

1363     if (db->db_dirtycnt == 0)
1364         return (db->db_data); /* Nothing is changing */

1366     for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1367         if (dr->dr_txg == tx->tx_txg)
1368             break;

1370     if (dr == NULL) {
1371         data = NULL;
1372     } else {
1373         dnode_t *dn;

1375         DB_DNODE_ENTER(dr->dr_dbuf);
1376         dn = DB_DNODE(dr->dr_dbuf);

1378         if (dn->dn_bonuslen == 0 &&
1379             dr->dr_dbuf->db_blkid == DMU_SPILL_BLKID)
1380             data = dr->dt.dl.dr_data->b_data;
1381         else
1382             data = dr->dt.dl.dr_data;

1384         DB_DNODE_EXIT(dr->dr_dbuf);
1385     }

1387     return (data);
1388 }

1390 void
1391 dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx)
1392 {
1393     objset_t *os = dn->dn_objset;
1394     void *data = NULL;
1395     dmu_buf_impl_t *db = NULL;
1396     uint64_t *user, *group;
1397     int flags = dn->dn_id_flags;
1398     int error;
1399     boolean_t have_spill = B_FALSE;

1401     if (!dmu_objset_userused_enabled(dn->dn_objset))
1402         return;

```

```

1404     if (before && (flags & (DN_ID_CHKED_BONUS|DN_ID_OLD_EXIST|
1405         DN_ID_CHKED_SPILL)))
1406         return;

1408     if (before && dn->dn_bonuslen != 0)
1409         data = DN_BONUS(dn->dn_phys);
1410     else if (!before && dn->dn_bonuslen != 0) {
1411         if (dn->dn_bonus) {
1412             db = dn->dn_bonus;
1413             mutex_enter(&db->db_mtx);
1414             data = dmu_objset_userquota_find_data(db, tx);
1415         } else {
1416             data = DN_BONUS(dn->dn_phys);
1417         }
1418     } else if (dn->dn_bonuslen == 0 && dn->dn_bonustype == DMU_OT_SA) {
1419         int rf = 0;

1421         if (RW_WRITE_HELD(&dn->dn_struct_rwlock))
1422             rf |= DB_RF_HAVESTRUCT;
1423         error = dmu_spill_hold_by_dnode(dn,
1424             rf | DB_RF_MUST_SUCCEED,
1425             FTAG, (dmu_buf_t **)&db);
1426         ASSERT(error == 0);
1427         mutex_enter(&db->db_mtx);
1428         data = (before) ? db->db_data :
1429             dmu_objset_userquota_find_data(db, tx);
1430         have_spill = B_TRUE;
1431     } else {
1432         mutex_enter(&dn->dn_mtx);
1433         dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1434         mutex_exit(&dn->dn_mtx);
1435         return;
1436     }

1438     if (before) {
1439         ASSERT(data);
1440         user = &dn->dn_olduid;
1441         group = &dn->dn_oldgid;
1442     } else if (data) {
1443         user = &dn->dn_newuid;
1444         group = &dn->dn_newgid;
1445     }

1447     /*
1448     * Must always call the callback in case the object
1449     * type has changed and that type isn't an object type to track
1450     */
1451     error = used_cbs[os->os_phys->os_type](dn->dn_bonustype, data,
1452         user, group);

1454     /*
1455     * Preserve existing uid/gid when the callback can't determine
1456     * what the new uid/gid are and the callback returned EEXIST.
1457     * The EEXIST error tells us to just use the existing uid/gid.
1458     * If we don't know what the old values are then just assign
1459     * them to 0, since that is a new file being created.
1460     */
1461     if (!before && data == NULL && error == EEXIST) {
1462         if (flags & DN_ID_OLD_EXIST) {
1463             dn->dn_newuid = dn->dn_olduid;
1464             dn->dn_newgid = dn->dn_oldgid;
1465         } else {
1466             dn->dn_newuid = 0;
1467             dn->dn_newgid = 0;
1468         }

```

```

1469         error = 0;
1470     }

1472     if (db)
1473         mutex_exit(&db->db_mtx);

1475     mutex_enter(&dn->dn_mtx);
1476     if (error == 0 && before)
1477         dn->dn_id_flags |= DN_ID_OLD_EXIST;
1478     if (error == 0 && !before)
1479         dn->dn_id_flags |= DN_ID_NEW_EXIST;

1481     if (have_spill) {
1482         dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1483     } else {
1484         dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1485     }
1486     mutex_exit(&dn->dn_mtx);
1487     if (have_spill)
1488         dmu_buf_rele((dmu_buf_t *)db, FTAG);
1489 }

1491 boolean_t
1492 dmu_objset_userspace_present(objset_t *os)
1493 {
1494     return (os->os_phys->os_flags &
1495         OBJSET_FLAG_USERACCOUNTING_COMPLETE);
1496 }

1498 int
1499 dmu_objset_userspace_upgrade(objset_t *os)
1500 {
1501     uint64_t obj;
1502     int err = 0;

1504     if (dmu_objset_userspace_present(os))
1505         return (0);
1506     if (!dmu_objset_userused_enabled(os))
1507         return (ENOTSUP);
1508     if (dmu_objset_is_snapshot(os))
1509         return (EINVAL);

1511     /*
1512     * We simply need to mark every object dirty, so that it will be
1513     * synced out and now accounted. If this is called
1514     * concurrently, or if we already did some work before crashing,
1515     * that's fine, since we track each object's accounted state
1516     * independently.
1517     */

1519     for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE, 0)) {
1520         dmu_tx_t *tx;
1521         dmu_buf_t *db;
1522         int objerr;

1524         if (issig(JUSTLOOKING) && issig(FORREAL))
1525             return (EINTR);

1527         objerr = dmu_bonus_hold(os, obj, FTAG, &db);
1528         if (objerr)
1529             continue;
1530         tx = dmu_tx_create(os);
1531         dmu_tx_hold_bonus(tx, obj);
1532         objerr = dmu_tx_assign(tx, TXG_WAIT);
1533         if (objerr) {
1534             dmu_tx_abort(tx);

```

```

1535         continue;
1536     }
1537     dmu_buf_will_dirty(db, tx);
1538     dmu_buf_rele(db, FTAG);
1539     dmu_tx_commit(tx);
1540 }
1541
1542 os->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
1543 txg_wait_synced(dmu_objset_pool(os), 0);
1544 return (0);
1545 }
1546
1547 void
1548 dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
1549                 uint64_t *usedobjsp, uint64_t *availobjsp)
1550 {
1551     dsl_dataset_space(os->os_dsl_dataset, refdbytesp, availbytesp,
1552                      usedobjsp, availobjsp);
1553 }
1554
1555 uint64_t
1556 dmu_objset_fsid_guid(objset_t *os)
1557 {
1558     return (dsl_dataset_fsid_guid(os->os_dsl_dataset));
1559 }
1560
1561 void
1562 dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat)
1563 {
1564     stat->dds_type = os->os_phys->os_type;
1565     if (os->os_dsl_dataset)
1566         dsl_dataset_fast_stat(os->os_dsl_dataset, stat);
1567 }
1568
1569 void
1570 dmu_objset_stats(objset_t *os, nvlist_t *nv)
1571 {
1572     ASSERT(os->os_dsl_dataset ||
1573           os->os_phys->os_type == DMU_OST_META);
1574
1575     if (os->os_dsl_dataset != NULL)
1576         dsl_dataset_stats(os->os_dsl_dataset, nv);
1577
1578     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_TYPE,
1579                               os->os_phys->os_type);
1580     dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERACCOUNTING,
1581                               dmu_objset_userspace_present(os));
1582 }
1583
1584 int
1585 dmu_objset_is_snapshot(objset_t *os)
1586 {
1587     if (os->os_dsl_dataset != NULL)
1588         return (dsl_dataset_is_snapshot(os->os_dsl_dataset));
1589     else
1590         return (B_FALSE);
1591 }
1592
1593 int
1594 dmu_snapshot_realname(objset_t *os, char *name, char *real, int maxlen,
1595                       boolean_t *conflict)
1596 {
1597     dsl_dataset_t *ds = os->os_dsl_dataset;
1598     uint64_t ignored;
1599
1600     if (ds->ds_phys->ds_snapnames_zapobj == 0)

```

```

1601         return (ENOENT);
1602
1603     return (zap_lookup_norm(ds->ds_dir->dd_pool->dp_meta_objset,
1604                             ds->ds_phys->ds_snapnames_zapobj, name, 8, 1, &ignored, MT_FIRST,
1605                             real, maxlen, conflict));
1606 }
1607
1608 int
1609 dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
1610                       uint64_t *idp, uint64_t *offp, boolean_t *case_conflict)
1611 {
1612     dsl_dataset_t *ds = os->os_dsl_dataset;
1613     zap_cursor_t cursor;
1614     zap_attribute_t attr;
1615
1616     if (ds->ds_phys->ds_snapnames_zapobj == 0)
1617         return (ENOENT);
1618
1619     zap_cursor_init_serialized(&cursor,
1620                                ds->ds_dir->dd_pool->dp_meta_objset,
1621                                ds->ds_phys->ds_snapnames_zapobj, *offp);
1622
1623     if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1624         zap_cursor_fini(&cursor);
1625         return (ENOENT);
1626     }
1627
1628     if (strlen(attr.za_name) + 1 > namelen) {
1629         zap_cursor_fini(&cursor);
1630         return (ENAMETOOLONG);
1631     }
1632
1633     (void) strcpy(name, attr.za_name);
1634     if (idp)
1635         *idp = attr.za_first_integer;
1636     if (case_conflict)
1637         *case_conflict = attr.za_normalization_conflict;
1638     zap_cursor_advance(&cursor);
1639     *offp = zap_cursor_serialize(&cursor);
1640     zap_cursor_fini(&cursor);
1641
1642     return (0);
1643 }
1644
1645 int
1646 dmu_dir_list_next(objset_t *os, int namelen, char *name,
1647                  uint64_t *idp, uint64_t *offp)
1648 {
1649     dsl_dir_t *dd = os->os_dsl_dataset->ds_dir;
1650     zap_cursor_t cursor;
1651     zap_attribute_t attr;
1652
1653     /* there is no next dir on a snapshot! */
1654     if (os->os_dsl_dataset->ds_object !=
1655         dd->dd_phys->dd_head_dataset_obj)
1656         return (ENOENT);
1657
1658     zap_cursor_init_serialized(&cursor,
1659                                dd->dd_pool->dp_meta_objset,
1660                                dd->dd_phys->dd_child_dir_zapobj, *offp);
1661
1662     if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1663         zap_cursor_fini(&cursor);
1664         return (ENOENT);
1665     }

```

```

1667     if (strlen(attr.za_name) + 1 > namelen) {
1668         zap_cursor_fini(&cursor);
1669         return (ENAMETOOLONG);
1670     }

1672     (void) strcpy(name, attr.za_name);
1673     if (idp)
1674         *idp = attr.za_first_integer;
1675     zap_cursor_advance(&cursor);
1676     *offp = zap_cursor_serialize(&cursor);
1677     zap_cursor_fini(&cursor);

1679     return (0);
1680 }

1682 struct findarg {
1683     int (*func)(const char *, void *);
1684     void *arg;
1685 };

1687 /* ARGSUSED */
1688 static int
1689 findfunc(spa_t *spa, uint64_t dsobj, const char *dsname, void *arg)
1690 {
1691     struct findarg *fa = arg;
1692     return (fa->func(dsname, fa->arg));
1693 }

1695 /*
1696  * Find all objsets under name, and for each, call 'func(child_name, arg)'.
1697  * Perhaps change all callers to use dmu_objset_find_spa()?
1698  */
1699 int
1700 dmu_objset_find(char *name, int func(const char *, void *), void *arg,
1701                int flags)
1702 {
1703     struct findarg fa;
1704     fa.func = func;
1705     fa.arg = arg;
1706     return (dmu_objset_find_spa(NULL, name, findfunc, &fa, flags));
1707 }

1709 /*
1710  * Find all objsets under name, call func on each
1711  */
1712 int
1713 dmu_objset_find_spa(spa_t *spa, const char *name,
1714                    int func(spa_t *, uint64_t, const char *, void *), void *arg, int flags)
1715 {
1716     dsl_dir_t *dd;
1717     dsl_pool_t *dp;
1718     dsl_dataset_t *ds;
1719     zap_cursor_t zc;
1720     zap_attribute_t *attr;
1721     char *child;
1722     uint64_t thisobj;
1723     int err;

1725     if (name == NULL)
1726         name = spa_name(spa);
1727     err = dsl_dir_open_spa(spa, name, FTAG, &dd, NULL);
1728     if (err)
1729         return (err);

1731     /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
1732     if (dd->dd_myname[0] == '$') {

```

```

1733         dsl_dir_close(dd, FTAG);
1734         return (0);
1735     }

1737     thisobj = dd->dd_phys->dd_head_dataset_obj;
1738     attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
1739     dp = dd->dd_pool;

1741     /*
1742      * Iterate over all children.
1743      */
1744     if (flags & DS_FIND_CHILDREN) {
1745         for (zap_cursor_init(&zc, dp->dp_meta_objset,
1746                          dd->dd_phys->dd_child_dir_zapobj);
1747              zap_cursor_retrieve(&zc, attr) == 0;
1748              (void) zap_cursor_advance(&zc)) {
1749             ASSERT(attr->za_integer_length == sizeof (uint64_t));
1750             ASSERT(attr->za_num_integers == 1);

1752             child = kmem_asprintf("%s/%s", name, attr->za_name);
1753             err = dmu_objset_find_spa(spa, child, func, arg, flags);
1754             strfree(child);
1755             if (err)
1756                 break;
1757         }
1758         zap_cursor_fini(&zc);

1760     if (err) {
1761         dsl_dir_close(dd, FTAG);
1762         kmem_free(attr, sizeof (zap_attribute_t));
1763         return (err);
1764     }
1765 }

1767 /*
1768  * Iterate over all snapshots.
1769  */
1770 if (flags & DS_FIND_SNAPSHOTS) {
1771     if (!dsl_pool_sync_context(dp))
1772         rw_enter(&dp->dp_config_rwlock, RW_READER);
1773     err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);
1774     if (!dsl_pool_sync_context(dp))
1775         rw_exit(&dp->dp_config_rwlock);

1777     if (err == 0) {
1778         uint64_t snapobj = ds->ds_phys->ds_snapnames_zapobj;
1779         dsl_dataset_rele(ds, FTAG);

1781         for (zap_cursor_init(&zc, dp->dp_meta_objset, snapobj);
1782              zap_cursor_retrieve(&zc, attr) == 0;
1783              (void) zap_cursor_advance(&zc)) {
1784             ASSERT(attr->za_integer_length ==
1785                    sizeof (uint64_t));
1786             ASSERT(attr->za_num_integers == 1);

1788             child = kmem_asprintf("%s@%s",
1789                                   name, attr->za_name);
1790             err = func(spa, attr->za_first_integer,
1791                      child, arg);
1792             strfree(child);
1793             if (err)
1794                 break;
1795         }
1796         zap_cursor_fini(&zc);
1797     }
1798 }

```

```
1800     dsl_dir_close(dd, FTAG);
1801     kmem_free(attr, sizeof (zap_attribute_t));
1803     if (err)
1804         return (err);
1806     /*
1807      * Apply to self if appropriate.
1808      */
1809     err = func(spa, thisobj, name, arg);
1810     return (err);
1811 }
1813 /* ARGSUSED */
1814 int
1815 dmu_objset_prefetch(const char *name, void *arg)
1816 {
1817     dsl_dataset_t *ds;
1819     if (dsl_dataset_hold(name, FTAG, &ds))
1820         return (0);
1822     if (!BP_IS_HOLE(&ds->ds_phys->ds_bp)) {
1823         mutex_enter(&ds->ds_opening_lock);
1824         if (ds->ds_objset == NULL) {
1825             uint32_t aflags = ARC_NOWAIT | ARC_PREFETCH;
1826             zbookmark_t zb;
1828             SET_BOOKMARK(&zb, ds->ds_object, ZB_ROOT_OBJECT,
1829                 ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1831             (void) dsl_read_nolock(NULL, dsl_dataset_get_spa(ds),
1832                 &ds->ds_phys->ds_bp, NULL, NULL,
1833                 ZIO_PRIORITY_ASYNC_READ,
1834                 ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
1835                 &aflags, &zb);
1836         }
1837         mutex_exit(&ds->ds_opening_lock);
1838     }
1840     dsl_dataset_rele(ds, FTAG);
1841     return (0);
1842 }
1844 void
1845 dmu_objset_set_user(objset_t *os, void *user_ptr)
1846 {
1847     ASSERT(MUTEX_HELD(&os->os_user_ptr_lock));
1848     os->os_user_ptr = user_ptr;
1849 }
1851 void *
1852 dmu_objset_get_user(objset_t *os)
1853 {
1854     ASSERT(MUTEX_HELD(&os->os_user_ptr_lock));
1855     return (os->os_user_ptr);
1856 }
```



```

*****
45463 Thu Jun 28 15:57:15 2012
new/usr/src/uts/common/fs/zfs/dmu_send.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
unchanged_portion_omitted

390 /*
391  * Return TRUE if 'earlier' is an earlier snapshot in 'later's timeline.
392  * For example, they could both be snapshots of the same filesystem, and
393  * 'earlier' is before 'later'. Or 'earlier' could be the origin of
394  * 'later's filesystem. Or 'earlier' could be an older snapshot in the origin's
395  * filesystem. Or 'earlier' could be the origin's origin.
396  */
397 static boolean_t
398 is_before(dsl_dataset_t *later, dsl_dataset_t *earlier)
399 {
400     dsl_pool_t *dp = later->ds_dir->dd_pool;
401     int error;
402     boolean_t ret;
403     dsl_dataset_t *origin;
404
405     if (earlier->ds_phys->ds_creation_txg >=
406         later->ds_phys->ds_creation_txg)
407         return (B_FALSE);
408
409     if (later->ds_dir == earlier->ds_dir)
410         return (B_TRUE);
411     if (!dsl_dir_is_clone(later->ds_dir))
412         return (B_FALSE);
413
414     rw_enter(&dp->dp_config_rwlock, RW_READER);
415     if (later->ds_dir->dd_phys->dd_origin_obj == earlier->ds_object) {
416         rw_exit(&dp->dp_config_rwlock);
417         return (B_TRUE);
418     }
419     error = dsl_dataset_hold_obj(dp,
420         later->ds_dir->dd_phys->dd_origin_obj, FTAG, &origin);
421     rw_exit(&dp->dp_config_rwlock);
422     if (error != 0)
423         return (B_FALSE);
424     ret = is_before(origin, earlier);
425     dsl_dataset_rele(origin, FTAG);
426     return (ret);
427 }
428
429 #endif /* ! codereview */
430 int
431 dmu_send(objset_t *tosnap, objset_t *fromsnap, int outfd, vnode_t *vp,
432     offset_t *off)
433 dmu_send(objset_t *tosnap, objset_t *fromsnap, boolean_t fromorigin,
434     int outfd, vnode_t *vp, offset_t *off)
435 {
436     dsl_dataset_t *ds = tosnap->os_dsl_dataset;
437     dsl_dataset_t *fromds = fromsnap ? fromsnap->os_dsl_dataset : NULL;
438     dmu_replay_record_t *drr;
439     dmu_sendarg_t *dsp;
440     int err;
441     uint64_t fromtxg = 0;

```

```

441     /* tosnap must be a snapshot */
442     if (ds->ds_phys->ds_next_snap_obj == 0)
443         return (EINVAL);
444
445     /*
446     * fromsnap must be an earlier snapshot from the same fs as tosnap,
447     * or the origin's fs.
448     */
449     if (fromds != NULL && !is_before(ds, fromds))
450         /* fromsnap must be an earlier snapshot from the same fs as tosnap */
451         if (fromds && (ds->ds_dir != fromds->ds_dir ||
452             fromds->ds_phys->ds_creation_txg >= ds->ds_phys->ds_creation_txg))
453             return (EXDEV);
454
455     if (fromorigin) {
456         dsl_pool_t *dp = ds->ds_dir->dd_pool;
457
458         if (fromsnap)
459             return (EINVAL);
460
461         if (dsl_dir_is_clone(ds->ds_dir)) {
462             rw_enter(&dp->dp_config_rwlock, RW_READER);
463             err = dsl_dataset_hold_obj(dp,
464                 ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &fromds);
465             rw_exit(&dp->dp_config_rwlock);
466             if (err)
467                 return (err);
468         } else {
469             fromorigin = B_FALSE;
470         }
471     }
472
473     drr = kmem_zalloc(sizeof (dmu_replay_record_t), KM_SLEEP);
474     drr->drr_type = DRR_BEGIN;
475     DMU_SET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo,
476         DMU_SUBSTREAM);
477
478     #ifdef _KERNEL
479     if (dmu_objset_type(tosnap) == DMU_OST_ZFS) {
480         uint64_t version;
481         if (zfs_get_zplprop(tosnap, ZFS_PROP_VERSION, &version) != 0) {
482             kmem_free(drr, sizeof (dmu_replay_record_t));
483             return (EINVAL);
484         }
485         if (version == ZPL_VERSION_SA) {
486             DMU_SET_FEATUREFLAGS(
487                 drr->drr_u.drr_begin.drr_versioninfo,
488                 DMU_BACKUP_FEATURE_SA_SPILL);
489         }
490     }
491     #endif
492
493     drr->drr_u.drr_begin.drr_creation_time =
494         ds->ds_phys->ds_creation_time;
495     drr->drr_u.drr_begin.drr_type = tosnap->os_phys->os_type;
496     if (fromds != NULL && ds->ds_dir != fromds->ds_dir)
497         if (fromorigin)
498             drr->drr_u.drr_begin.drr_flags |= DRR_FLAG_CLONE;
499     drr->drr_u.drr_begin.drr_toguid = ds->ds_phys->ds_guid;
500     if (ds->ds_phys->ds_flags & DS_FLAG_CI_DATASET)
501         drr->drr_u.drr_begin.drr_flags |= DRR_FLAG_CI_DATA;
502
503     if (fromds)

```

```

483         drr->drr_u.drr_begin.drr_fromguid = fromds->ds_phys->ds_guid;
484         dsl_dataset_name(ds, drr->drr_u.drr_begin.drr_toname);

486     if (fromds)
487         fromtxg = fromds->ds_phys->ds_creation_txg;
488     if (fromorigin)
489         dsl_dataset_rele(fromds, FTAG);

489     dsp = kmem_zalloc(sizeof (dmu_sendarg_t), KM_SLEEP);

491     dsp->dsa_drr = drr;
492     dsp->dsa_vp = vp;
493     dsp->dsa_outfd = outfd;
494     dsp->dsa_proc = curproc;
495     dsp->dsa_os = tosnap;
496     dsp->dsa_off = off;
497     dsp->dsa_toguid = ds->ds_phys->ds_guid;
498     ZIO_SET_CHECKSUM(&dsp->dsa_zc, 0, 0, 0);
499     dsp->dsa_pending_op = PENDING_NONE;

501     mutex_enter(&ds->ds_sendstream_lock);
502     list_insert_head(&ds->ds_sendstreams, dsp);
503     mutex_exit(&ds->ds_sendstream_lock);

505     if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0) {
506         err = dsp->dsa_err;
507         goto out;
508     }

510     err = traverse_dataset(ds, fromtxg, TRAVERSE_PRE | TRAVERSE_PREFETCH,
511         backup_cb, dsp);

513     if (dsp->dsa_pending_op != PENDING_NONE)
514         if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0)
515             err = EINTR;

517     if (err) {
518         if (err == EINTR && dsp->dsa_err)
519             err = dsp->dsa_err;
520         goto out;
521     }

523     bzero(drr, sizeof (dmu_replay_record_t));
524     drr->drr_type = DRR_END;
525     drr->drr_u.drr_end.drr_checksum = dsp->dsa_zc;
526     drr->drr_u.drr_end.drr_toguid = dsp->dsa_toguid;

528     if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0) {
529         err = dsp->dsa_err;
530         goto out;
531     }

533 out:
534     mutex_enter(&ds->ds_sendstream_lock);
535     list_remove(&ds->ds_sendstreams, dsp);
536     mutex_exit(&ds->ds_sendstream_lock);

538     kmem_free(drr, sizeof (dmu_replay_record_t));
539     kmem_free(dsp, sizeof (dmu_sendarg_t));

541     return (err);
542 }

544 int
545 dmu_send_estimate(objset_t *tosnap, objset_t *fromsnap, uint64_t *sizep)
523 dmu_send_estimate(objset_t *tosnap, objset_t *fromsnap, boolean_t fromorigin,

```

```

524     uint64_t *sizep)
546 {
547     dsl_dataset_t *ds = tosnap->os_dsl_dataset;
548     dsl_dataset_t *fromds = fromsnap ? fromsnap->os_dsl_dataset : NULL;
549     dsl_pool_t *dp = ds->ds_dir->dd_pool;
550     int err;
551     uint64_t size;

553     /* tosnap must be a snapshot */
554     if (ds->ds_phys->ds_next_snap_obj == 0)
555         return (EINVAL);

557     /*
558      * fromsnap must be an earlier snapshot from the same fs as tosnap,
559      * or the origin's fs.
560      */
561     if (fromds != NULL && !is_before(ds, fromds))
562         /* fromsnap must be an earlier snapshot from the same fs as tosnap */
563         if (fromds && (ds->ds_dir != fromds->ds_dir ||
564             fromds->ds_phys->ds_creation_txg >= ds->ds_phys->ds_creation_txg))
565             return (EXDEV);

541     if (fromorigin) {
542         if (fromsnap)
543             return (EINVAL);

545         if (dsl_dir_is_clone(ds->ds_dir)) {
546             rw_enter(&dp->dp_config_rwlock, RW_READER);
547             err = dsl_dataset_hold_obj(dp,
548                 ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &fromds);
549             rw_exit(&dp->dp_config_rwlock);
550             if (err)
551                 return (err);
552         } else {
553             fromorigin = B_FALSE;
554         }
555     }

564     /* Get uncompressed size estimate of changed data. */
565     if (fromds == NULL) {
566         size = ds->ds_phys->ds_uncompressed_bytes;
567     } else {
568         uint64_t used, comp;
569         err = dsl_dataset_space_written(fromds, ds,
570             &used, &comp, &size);
564         if (fromorigin)
565             dsl_dataset_rele(fromds, FTAG);
571         if (err)
572             return (err);
573     }

575     /*
576      * Assume that space (both on-disk and in-stream) is dominated by
577      * data. We will adjust for indirect blocks and the copies property,
578      * but ignore per-object space used (eg, dnodes and DRR_OBJECT records).
579      */

581     /*
582      * Subtract out approximate space used by indirect blocks.
583      * Assume most space is used by data blocks (non-indirect, non-dnode).
584      * Assume all blocks are recordsize. Assume ditto blocks and
585      * internal fragmentation counter out compression.
586      *
587      * Therefore, space used by indirect blocks is sizeof(blkptr_t) per
588      * block, which we observe in practice.
589      */

```

```

590     uint64_t recordsize;
591     rw_enter(&dp->dp_config_rwlock, RW_READER);
592     err = dsl_prop_get_ds(ds, "recordsize",
593         sizeof(recordsize), 1, &recordsize, NULL);
594     rw_exit(&dp->dp_config_rwlock);
595     if (err)
596         return (err);
597     size -= size / recordsize * sizeof(blkptr_t);

599     /* Add in the space for the record associated with each block. */
600     size += size / recordsize * sizeof(dmu_replay_record_t);

602     *sizep = size;

604     return (0);
605 }
    unchanged portion omitted

650 static void
651 recv_new_sync(void *arg1, void *arg2, dmu_tx_t *tx)
652 {
653     dsl_dir_t *dd = arg1;
654     struct recvbeginsyncarg *rbsa = arg2;
655     uint64_t flags = DS_FLAG_INCONSISTENT | rbsa->dsflags;
656     uint64_t dsobj;

658     /* Create and open new dataset. */
659     dsobj = dsl_dataset_create_sync(dd, strchr(rbsa->tofs, '/') + 1,
660         rbsa->origin, flags, rbsa->cr, tx);
661     VERIFY(0 == dsl_dataset_own_obj(dd->dd_pool, dsobj,
662         B_TRUE, dmu_recv_tag, &rbsa->ds));

664     if (rbsa->origin == NULL) {
665         (void) dmu_objset_create_impl(dd->dd_pool->dp_spa,
666             rbsa->ds, &rbsa->ds->ds_phys->ds_bp, rbsa->type, tx);
667     }

669     spa_history_log_internal_ds(rbsa->ds, "receive new", tx, "");
670     spa_history_log_internal(LOG_DS_REPLAY_FULL_SYNC,
671         dd->dd_pool->dp_spa, tx, "dataset = %lld", dsobj);
    unchanged portion omitted

743 /* ARGSUSED */
744 static void
745 recv_existing_sync(void *arg1, void *arg2, dmu_tx_t *tx)
746 {
747     dsl_dataset_t *ohds = arg1;
748     struct recvbeginsyncarg *rbsa = arg2;
749     dsl_pool_t *dp = ohds->ds_dir->dd_pool;
750     dsl_dataset_t *cnds;
751     uint64_t flags = DS_FLAG_INCONSISTENT | rbsa->dsflags;
752     uint64_t dsobj;

754     /* create and open the temporary clone */
755     dsobj = dsl_dataset_create_sync(ohds->ds_dir, rbsa->clonelastname,
756         ohds->ds_prev, flags, rbsa->cr, tx);
757     VERIFY(0 == dsl_dataset_own_obj(dp, dsobj, B_TRUE, dmu_recv_tag, &cnds));

759     /*
760      * If we actually created a non-clone, we need to create the
761      * objset in our new dataset.
762      */
763     if (BP_IS_HOLE(dsl_dataset_get_blkptr(cnds))) {
764         (void) dmu_objset_create_impl(dp->dp_spa,
765             cnds, dsl_dataset_get_blkptr(cnds), rbsa->type, tx);

```

```

766     }

768     rbsa->ds = cnds;

770     spa_history_log_internal_ds(cnds, "receive over existing", tx, "");
771     spa_history_log_internal(LOG_DS_REPLAY_INC_SYNC,
772         dp->dp_spa, tx, "dataset = %lld", dsobj);
    unchanged portion omitted

1562 static void
1563 recv_end_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1564 {
1565     dsl_dataset_t *ds = arg1;
1566     struct recvendsyncarg *resa = arg2;

1568     dsl_dataset_snapshot_sync(ds, resa->tosnap, tx);

1570     /* set snapshot's creation time and guid */
1571     dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
1572     ds->ds_prev->ds_phys->ds_creation_time = resa->creation_time;
1573     ds->ds_prev->ds_phys->ds_guid = resa->toguid;
1574     ds->ds_prev->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;

1576     dmu_buf_will_dirty(ds->ds_dbuf, tx);
1577     ds->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;
1578     spa_history_log_internal_ds(ds, "finished receiving", tx, "");
1579 #endif /* !codereview */
1580 }

1582 static int
1583 add_ds_to_guidmap(avl_tree_t *guid_map, dsl_dataset_t *ds)
1584 {
1585     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1586     uint64_t snapobj = ds->ds_phys->ds_prev_snap_obj;
1587     dsl_dataset_t *snapds;
1588     guid_map_entry_t *gmep;
1589     int err;

1591     ASSERT(guid_map != NULL);

1593     rw_enter(&dp->dp_config_rwlock, RW_READER);
1594     err = dsl_dataset_hold_obj(dp, snapobj, guid_map, &snapds);
1595     if (err == 0) {
1596         gmep = kmem_alloc(sizeof(guid_map_entry_t), KM_SLEEP);
1597         gmep->guid = snapds->ds_phys->ds_guid;
1598         gmep->gme_ds = snapds;
1599         avl_add(guid_map, gmep);
1600     }

1602     rw_exit(&dp->dp_config_rwlock);
1603     return (err);
1604 }

1606 static int
1607 dmu_recv_existing_end(dmu_recv_cookie_t *drc)
1608 {
1609     struct recvendsyncarg resa;
1610     dsl_dataset_t *ds = drc->drc_logical_ds;
1611     int err, myerr;

1613     /*
1614      * XXX hack; seems the ds is still dirty and dsl_pool_zil_clean()
1615      * expects it to have a ds_user_ptr (and zil), but clone_swap()
1616      * can close it.
1617      */

```

```

1618     txg_wait_synced(ds->ds_dir->dd_pool, 0);

1620     if (dsl_dataset_tryown(ds, FALSE, dmu_recv_tag)) {
1621         err = dsl_dataset_clone_swap(drc->drc_real_ds, ds,
1622             drc->drc_force);
1623         if (err)
1624             goto out;
1625     } else {
1626         mutex_exit(&ds->ds_recvlock);
1627         dsl_dataset_rele(ds, dmu_recv_tag);
1628         (void) dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag,
1629             B_FALSE);
1630         return (EBUSY);
1631     }

1633     resa.creation_time = drc->drc_drrb->drr_creation_time;
1634     resa.toguid = drc->drc_drrb->drr_toguid;
1635     resa.tosnap = drc->drc_tosnap;

1637     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1638         recv_end_check, recv_end_sync, ds, &resa, 3);
1639     if (err) {
1640         /* swap back */
1641         (void) dsl_dataset_clone_swap(drc->drc_real_ds, ds, B_TRUE);
1642     }

1644 out:
1645     mutex_exit(&ds->ds_recvlock);
1646     if (err == 0 && drc->drc_guid_to_ds_map != NULL)
1647         (void) add_ds_to_guidmap(drc->drc_guid_to_ds_map, ds);
1648     dsl_dataset_disown(ds, dmu_recv_tag);
1649     myerr = dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag, B_FALSE);
1650     ASSERT3U(myerr, ==, 0);
1651     return (err);
1652 }

1654 static int
1655 dmu_recv_new_end(dmu_recv_cookie_t *drc)
1656 {
1657     struct recvendsyncarg resa;
1658     dsl_dataset_t *ds = drc->drc_logical_ds;
1659     int err;

1661     /*
1662     * XXX hack; seems the ds is still dirty and dsl_pool_zil_clean()
1663     * expects it to have a ds_user_ptr (and zil), but clone_swap()
1664     * can close it.
1665     */
1666     txg_wait_synced(ds->ds_dir->dd_pool, 0);

1668     resa.creation_time = drc->drc_drrb->drr_creation_time;
1669     resa.toguid = drc->drc_drrb->drr_toguid;
1670     resa.tosnap = drc->drc_tosnap;

1672     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1673         recv_end_check, recv_end_sync, ds, &resa, 3);
1674     if (err) {
1675         /* clean up the fs we just recv'd into */
1676         (void) dsl_dataset_destroy(ds, dmu_recv_tag, B_FALSE);
1677     } else {
1678         if (drc->drc_guid_to_ds_map != NULL)
1679             (void) add_ds_to_guidmap(drc->drc_guid_to_ds_map, ds);
1680         /* release the hold from dmu_recv_begin */
1681         dsl_dataset_disown(ds, dmu_recv_tag);
1682     }
1683     return (err);

```

```

1684 }

1686 int
1687 dmu_recv_end(dmu_recv_cookie_t *drc)
1688 {
1689     if (drc->drc_logical_ds != drc->drc_real_ds)
1690         return (dmu_recv_existing_end(drc));
1691     else
1692         return (dmu_recv_new_end(drc));
1693 }

```

new/usr/src/uts/common/fs/zfs/dmu_tx.c

1

```
*****
35017 Thu Jun 28 15:57:15 2012
new/usr/src/uts/common/fs/zfs/dmu_tx.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
24 * Copyright (c) 2012 by Delphix. All rights reserved.
25 */

27 #include <sys/dmu.h>
28 #include <sys/dmu_impl.h>
29 #include <sys/dbuf.h>
30 #include <sys/dmu_tx.h>
31 #include <sys/dmu_objset.h>
32 #include <sys/dsl_dataset.h> /* for dsl_dataset_block_freeable() */
33 #include <sys/dsl_dir.h> /* for dsl_dir_tempreserve_*() */
34 #include <sys/dsl_pool.h>
35 #include <sys/zap_impl.h> /* for fzap_default_block_shift */
36 #include <sys/spa.h>
37 #include <sys/sa.h>
38 #include <sys/sa_impl.h>
39 #include <sys/zfs_context.h>
40 #include <sys/varargs.h>

42 typedef void (*dmu_tx_hold_func_t)(dmu_tx_t *tx, struct dnode *dn,
43     uint64_t arg1, uint64_t arg2);

46 dmu_tx_t *
47 dmu_tx_create_dd(dsl_dir_t *dd)
48 {
49     dmu_tx_t *tx = kmem_zalloc(sizeof (dmu_tx_t), KM_SLEEP);
50     tx->tx_dir = dd;
51     if (dd != NULL)
52         if (dd)
53             tx->tx_pool = dd->dd_pool;
54     list_create(&tx->tx_holds, sizeof (dmu_tx_hold_t),
```

new/usr/src/uts/common/fs/zfs/dmu_tx.c

2

```
54         offsetof(dmu_tx_hold_t, txh_node));
55     list_create(&tx->tx_callbacks, sizeof (dmu_tx_callback_t),
56         offsetof(dmu_tx_callback_t, dcb_node));
57 #ifdef ZFS_DEBUG
58     refcount_create(&tx->tx_space_written);
59     refcount_create(&tx->tx_space_freed);
60 #endif
61     return (tx);
62 }
_____unchanged_portion_omitted_
```

```

*****
118083 Thu Jun 28 15:57:16 2012
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
unchanged_portion_omitted_

913 /*
914 * The snapshots must all be in the same pool.
915 */
916 int
917 dmu_snapshots_destroy_nvlist(nvlist_t *snaps, boolean_t defer,
918                             nvlist_t *errlist)
919 dmu_snapshots_destroy_nvlist(nvlist_t *snaps, boolean_t defer, char *failed)
920 {
921     int err;
922     dsl_sync_task_t *dst;
923     spa_t *spa;
924     nvpair_t *pair;
925     dsl_sync_task_group_t *dstg;

926     pair = nvlist_next_nvpair(snaps, NULL);
927     if (pair == NULL)
928         return (0);

930     err = spa_open(nvpair_name(pair), &spa, FTAG);
931     if (err)
932         return (err);
933     dstg = dsl_sync_task_group_create(spa_get_dsl(spa));

935     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
936          pair = nvlist_next_nvpair(snaps, pair)) {
937         dsl_dataset_t *ds;

939         err = dsl_dataset_own(nvpair_name(pair), B_TRUE, dstg, &ds);
940         if (err == 0) {
941             struct dsl_ds_destroyarg *dsda;

943             dsl_dataset_make_exclusive(ds, dstg);
944             dsda = kmem_zalloc(sizeof(struct dsl_ds_destroyarg),
945                               KM_SLEEP);
946             dsda->ds = ds;
947             dsda->defer = defer;
948             dsl_sync_task_create(dstg, dsl_dataset_destroy_check,
949                                 dsl_dataset_destroy_sync, dsda, dstg, 0);
950         } else if (err == ENOENT) {
951             err = 0;
952         } else {
953             fnvlist_add_int32(errlist, nvpair_name(pair), err);
954             (void) strcpy(failed, nvpair_name(pair));
955             break;
956         }
957     }

958     if (err == 0)
959         err = dsl_sync_task_group_wait(dstg);

961     for (dst = list_head(&dstg->dstg_tasks); dst;
962          dst = list_next(&dstg->dstg_tasks, dst)) {

```

```

963     struct dsl_ds_destroyarg *dsda = dst->dst_arg1;
964     dsl_dataset_t *ds = dsda->ds;

966     /*
967     * Return the snapshots that triggered the error.
968     * Return the file system name that triggered the error
969     */
970     if (dst->dst_err != 0) {
971         char name[ZFS_MAXNAMELEN];
972         dsl_dataset_name(ds, name);
973         fnvlist_add_int32(errlist, name, dst->dst_err);
974     }
975     if (dst->dst_err) {
976         dsl_dataset_name(ds, failed);
977     }
978     ASSERT3P(dsda->rm_origin, ==, NULL);
979     dsl_dataset_disown(ds, dstg);
980     kmem_free(dsda, sizeof(struct dsl_ds_destroyarg));
981 }

979     dsl_sync_task_group_destroy(dstg);
980     spa_close(spa, FTAG);
981     return (err);

983 }
unchanged_portion_omitted_

1038 /*
1039 * ds must be opened as OWNER. On return (whether successful or not),
1040 * ds will be closed and caller can no longer dereference it.
1041 */
1042 int
1043 dsl_dataset_destroy(dsl_dataset_t *ds, void *tag, boolean_t defer)
1044 {
1045     int err;
1046     dsl_sync_task_group_t *dstg;
1047     objset_t *os;
1048     dsl_dir_t *dd;
1049     uint64_t obj;
1050     struct dsl_ds_destroyarg dsda = { 0 };
1051     dsl_dataset_t dummy_ds = { 0 };

1052     dsda.ds = ds;

1054     if (dsl_dataset_is_snapshot(ds)) {
1055         /* Destroying a snapshot is simpler */
1056         dsl_dataset_make_exclusive(ds, tag);

1058         dsda.defer = defer;
1059         err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1060                               dsl_dataset_destroy_check, dsl_dataset_destroy_sync,
1061                               &dsda, tag, 0);
1062         ASSERT3P(dsda.rm_origin, ==, NULL);
1063         goto out;
1064     } else if (defer) {
1065         err = EINVAL;
1066         goto out;
1067     }

1069     dd = ds->ds_dir;
1070     dummy_ds.ds_dir = dd;
1071     dummy_ds.ds_object = ds->ds_object;

1071     /*
1072     * Check for errors and mark this ds as inconsistent, in
1073     * case we crash while freeing the objects.
1074     */

```

```

1075     err = dsl_sync_task_do(dd->dd_pool, dsl_dataset_destroy_begin_check,
1076         dsl_dataset_destroy_begin_sync, ds, NULL, 0);
1077     if (err)
1078         goto out;

1080     err = dmu_objset_from_ds(ds, &os);
1081     if (err)
1082         goto out;

1084     /*
1085      * If async destruction is not enabled try to remove all objects
1086      * while in the open context so that there is less work to do in
1087      * the syncing context.
1088      */
1089     if (!spa_feature_is_enabled(dsl_dataset_get_spa(ds),
1090         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
1091         for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE,
1092             ds->ds_phys->ds_prev_snap_txg)) {
1093             /*
1094              * Ignore errors, if there is not enough disk space
1095              * we will deal with it in dsl_dataset_destroy_sync().
1096              */
1097             (void) dmu_free_object(os, obj);
1098         }
1099         if (err != ESRCH)
1100             goto out;
1101     }

1103     /*
1104      * Only the ZIL knows how to free log blocks.
1105      */
1106     zil_destroy(dmu_objset_zil(os), B_FALSE);

1108     /*
1109      * Sync out all in-flight IO.
1110      */
1111     txg_wait_synced(dd->dd_pool, 0);

1113     /*
1114      * If we managed to free all the objects in open
1115      * context, the user space accounting should be zero.
1116      */
1117     if (ds->ds_phys->ds_bp.blk_fill == 0 &&
1118         dmu_objset_userused_enabled(os)) {
1119         uint64_t count;

1121         ASSERT(zap_count(os, DMU_USERUSED_OBJECT, &count) != 0 ||
1122             count == 0);
1123         ASSERT(zap_count(os, DMU_GROUPUSED_OBJECT, &count) != 0 ||
1124             count == 0);
1125     }

1127     rw_enter(&dd->dd_pool->dp_config_rwlock, RW_READER);
1128     err = dsl_dir_open_obj(dd->dd_pool, dd->dd_object, NULL, FTAG, &dd);
1129     rw_exit(&dd->dd_pool->dp_config_rwlock);

1131     if (err)
1132         goto out;

1134     /*
1135      * Blow away the dsl_dir + head dataset.
1136      */
1137     dsl_dataset_make_exclusive(ds, tag);
1138     /*
1139      * If we're removing a clone, we might also need to remove its
1140      * origin.

```

```

1141     /*
1142     do {
1143         dsda.need_prep = B_FALSE;
1144         if (dsl_dir_is_clone(dd)) {
1145             err = dsl_dataset_origin_rm_prep(&dsda, tag);
1146             if (err) {
1147                 dsl_dir_close(dd, FTAG);
1148                 goto out;
1149             }
1150         }

1152         dstg = dsl_sync_task_group_create(ds->ds_dir->dd_pool);
1153         dsl_sync_task_create(dstg, dsl_dataset_destroy_check,
1154             dsl_dataset_destroy_sync, &dsda, tag, 0);
1155         dsl_sync_task_create(dstg, dsl_dir_destroy_check,
1156             dsl_dir_destroy_sync, dd, FTAG, 0;
1157             dsl_dir_destroy_sync, &dummy_ds, FTAG, 0;
1158         err = dsl_sync_task_group_wait(dstg);
1159         dsl_sync_task_group_destroy(dstg);

1160     /*
1161      * We could be racing against 'zfs release' or 'zfs destroy -d'
1162      * on the origin snap, in which case we can get EBUSY if we
1163      * needed to destroy the origin snap but were not ready to
1164      * do so.
1165      */
1166     if (dsda.need_prep) {
1167         ASSERT(err == EBUSY);
1168         ASSERT(dsl_dir_is_clone(dd));
1169         ASSERT(dsda.rm_origin == NULL);
1170     }
1171     } while (dsda.need_prep);

1173     if (dsda.rm_origin != NULL)
1174         dsl_dataset_disown(dsda.rm_origin, tag);

1176     /* if it is successful, dsl_dir_destroy_sync will close the dd */
1177     if (err)
1178         dsl_dir_close(dd, FTAG);
1179 out:
1180     dsl_dataset_disown(ds, tag);
1181     return (err);
1182 }
unchanged portion omitted

1326 /* ARGSUSED */
1327 static void
1328 dsl_dataset_destroy_begin_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1329 {
1330     dsl_dataset_t *ds = arg1;
1331     dsl_pool_t *dp = ds->ds_dir->dd_pool;

1332     /* Mark it as inconsistent on-disk, in case we crash */
1333     dmu_buf_will_dirty(ds->ds_dbuf, tx);
1334     ds->ds_phys->ds_flags |= DS_FLAG_INCONSISTENT;

1336     spa_history_log_internal_ds(ds, "destroy begin", tx, "");
1337     spa_history_log_internal(LOG_DS_DESTROY_BEGIN, dp->dp_spa, tx,
1338         "dataset = %llu", ds->ds_object);
1337 }
unchanged portion omitted

1635 void
1636 dsl_dataset_destroy_sync(void *arg1, void *tag, dmu_tx_t *tx)
1637 {
1638     struct dsl_ds_destroyarg *dsda = arg1;

```



```

1771         ds_prev->ds_phys->ds_unique_bytes += used;
1772     }
1773
1774     /* Adjust snapused. */
1775     dsl_deadlist_space_range(&ds_next->ds_deadlist,
1776         ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
1777         &used, &comp, &uncomp);
1778     dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
1779         -used, -comp, -uncomp, tx);
1780
1781     /* Move blocks to be freed to pool's free list. */
1782     dsl_deadlist_move_bproj(&ds_next->ds_deadlist,
1783         &dp->dp_free_bproj, ds->ds_phys->ds_prev_snap_txg,
1784         tx);
1785     dsl_dir_diduse_space(tx->tx_pool->dp_free_dir,
1786         DD_USED_HEAD, used, comp, uncomp, tx);
1787
1788     /* Merge our deadlist into next's and free it. */
1789     dsl_deadlist_merge(&ds_next->ds_deadlist,
1790         ds->ds_phys->ds_deadlist_obj, tx);
1791 }
1792 dsl_deadlist_close(&ds->ds_deadlist);
1793 dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
1794
1795 /* Collapse range in clone heads */
1796 dsl_dataset_remove_clones_key(ds,
1797     ds->ds_phys->ds_creation_txg, tx);
1798
1799 if (dsl_dataset_is_snapshot(ds_next)) {
1800     dsl_dataset_t *ds_nextnext;
1801
1802     /*
1803      * Update next's unique to include blocks which
1804      * were previously shared by only this snapshot
1805      * and it. Those blocks will be born after the
1806      * prev snap and before this snap, and will have
1807      * died after the next snap and before the one
1808      * after that (ie. be on the snap after next's
1809      * deadlist).
1810      */
1811     VERIFY(0 == dsl_dataset_hold_obj(dp,
1812         ds_next->ds_phys->ds_next_snap_obj,
1813         FTAG, &ds_nextnext));
1814     dsl_deadlist_space_range(&ds_nextnext->ds_deadlist,
1815         ds->ds_phys->ds_prev_snap_txg,
1816         ds->ds_phys->ds_creation_txg,
1817         &used, &comp, &uncomp);
1818     ds_next->ds_phys->ds_unique_bytes += used;
1819     dsl_dataset_rele(ds_nextnext, FTAG);
1820     ASSERT3P(ds_next->ds_prev, ==, NULL);
1821
1822     /* Collapse range in this head. */
1823     dsl_dataset_t *hds;
1824     VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
1825         ds->ds_dir->dd_phys->dd_head_dataset_obj,
1826         FTAG, &hds));
1827     dsl_deadlist_remove_key(&hds->ds_deadlist,
1828         ds->ds_phys->ds_creation_txg, tx);
1829     dsl_dataset_rele(hds, FTAG);
1830
1831 } else {
1832     ASSERT3P(ds_next->ds_prev, ==, ds);
1833     dsl_dataset_drop_ref(ds_next->ds_prev, ds_next);
1834     ds_next->ds_prev = NULL;
1835     if (ds_prev) {
1836         VERIFY(0 == dsl_dataset_get_ref(dp,

```

```

1837         ds->ds_phys->ds_prev_snap_obj,
1838         ds_next, &ds_next->ds_prev));
1839     }
1840
1841     dsl_dataset_recalc_head_uniq(ds_next);
1842
1843     /*
1844      * Reduce the amount of our unconsmed refreservation
1845      * being charged to our parent by the amount of
1846      * new unique data we have gained.
1847      */
1848     if (old_unique < ds_next->ds_reserved) {
1849         int64_t mrsdelta;
1850         uint64_t new_unique =
1851             ds_next->ds_phys->ds_unique_bytes;
1852
1853         ASSERT(old_unique <= new_unique);
1854         mrsdelta = MIN(new_unique - old_unique,
1855             ds_next->ds_reserved - old_unique);
1856         dsl_dir_diduse_space(ds->ds_dir,
1857             DD_USED_REFRESRV, -mrsdelta, 0, 0, tx);
1858     }
1859     }
1860     dsl_dataset_rele(ds_next, FTAG);
1861 } else {
1862     zfeature_info_t *async_destroy =
1863         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY];
1864
1865     /*
1866      * There's no next snapshot, so this is a head dataset.
1867      * Destroy the deadlist. Unless it's a clone, the
1868      * deadlist should be empty. (If it's a clone, it's
1869      * safe to ignore the deadlist contents.)
1870      */
1871     dsl_deadlist_close(&ds->ds_deadlist);
1872     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
1873     ds->ds_phys->ds_deadlist_obj = 0;
1874
1875     if (!spa_feature_is_enabled(dp->dp_spa, async_destroy)) {
1876         err = old_synchronous_dataset_destroy(ds, tx);
1877     } else {
1878         /*
1879          * Move the bptree into the pool's list of trees to
1880          * clean up and update space accounting information.
1881          */
1882         uint64_t used, comp, uncomp;
1883
1884         ASSERT(err == 0 || err == EBUSY);
1885         if (!spa_feature_is_active(dp->dp_spa, async_destroy)) {
1886             spa_feature_incr(dp->dp_spa, async_destroy, tx);
1887             dp->dp_bptree_obj = bptree_alloc(
1888                 dp->dp_meta_objset, tx);
1889             VERIFY(zap_add(dp->dp_meta_objset,
1890                 DMU_POOL_DIRECTORY_OBJECT,
1891                 DMU_POOL_BPTREE_OBJ, sizeof(uint64_t), 1,
1892                 &dp->dp_bptree_obj, tx) == 0);
1893         }
1894
1895         used = ds->ds_dir->dd_phys->dd_used_bytes;
1896         comp = ds->ds_dir->dd_phys->dd_compressed_bytes;
1897         uncomp = ds->ds_dir->dd_phys->dd_uncompressed_bytes;
1898
1899         ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) ||
1900             ds->ds_phys->ds_unique_bytes == used);
1901
1902         bptree_add(dp->dp_meta_objset, dp->dp_bptree_obj,

```

```

1903         &ds->ds_phys->ds_bp, ds->ds_phys->ds_prev_snap_txg,
1904         used, comp, uncomp, tx);
1905     dsl_dir_diduse_space(ds->ds_dir, DD_USED_HEAD,
1906         -used, -comp, -uncomp, tx);
1907     dsl_dir_diduse_space(dp->dp_free_dir, DD_USED_HEAD,
1908         used, comp, uncomp, tx);
1909 }
1910
1911     if (ds->ds_prev != NULL) {
1912         if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
1913             VERIFY3U(0, ==, zap_remove_int(mos,
1914                 ds->ds_prev->ds_dir->dd_phys->dd_clones,
1915                 ds->ds_object, tx));
1916         }
1917         dsl_dataset_rele(ds->ds_prev, ds);
1918         ds->ds_prev = ds_prev = NULL;
1919     }
1920 }
1921
1922 /*
1923  * This must be done after the dsl_traverse(), because it will
1924  * re-open the objset.
1925  */
1926 if (ds->ds_objset) {
1927     dmu_objset_evict(ds->ds_objset);
1928     ds->ds_objset = NULL;
1929 }
1930
1931 if (ds->ds_dir->dd_phys->dd_head_dataset_obj == ds->ds_object) {
1932     /* Erase the link in the dir */
1933     dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
1934     ds->ds_dir->dd_phys->dd_head_dataset_obj = 0;
1935     ASSERT(ds->ds_phys->ds_snapnames_zapobj != 0);
1936     err = zap_destroy(mos, ds->ds_phys->ds_snapnames_zapobj, tx);
1937     ASSERT(err == 0);
1938 } else {
1939     /* remove from snapshot namespace */
1940     dsl_dataset_t *ds_head;
1941     ASSERT(ds->ds_phys->ds_snapnames_zapobj == 0);
1942     VERIFY(0 == dsl_dataset_hold_obj(dp,
1943         ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ds_head));
1944     VERIFY(0 == dsl_dataset_get_snapname(ds));
1945 #ifdef ZFS_DEBUG
1946     {
1947         uint64_t val;
1948
1949         err = dsl_dataset_snap_lookup(ds_head,
1950             ds->ds_snapname, &val);
1951         ASSERT3U(err, ==, 0);
1952         ASSERT3U(val, ==, objj);
1953     }
1954 #endif
1955     err = dsl_dataset_snap_remove(ds_head, ds->ds_snapname, tx);
1956     ASSERT(err == 0);
1957     dsl_dataset_rele(ds_head, FTAG);
1958 }
1959
1960 if (ds_prev && ds->ds_prev != ds_prev)
1961     dsl_dataset_rele(ds_prev, FTAG);
1962
1963 spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
1964 spa_history_log_internal(LOG_DS_DESTROY, dp->dp_spa, tx,
1965     "dataset = %llu", ds->ds_object);
1966
1967 if (ds->ds_phys->ds_next_clones_obj != 0) {
1968     uint64_t count;

```

```

1967     ASSERT(0 == zap_count(mos,
1968         ds->ds_phys->ds_next_clones_obj, &count) && count == 0);
1969     VERIFY(0 == dmu_object_free(mos,
1970         ds->ds_phys->ds_next_clones_obj, tx));
1971 }
1972 if (ds->ds_phys->ds_props_obj != 0)
1973     VERIFY(0 == zap_destroy(mos, ds->ds_phys->ds_props_obj, tx));
1974 if (ds->ds_phys->ds_userrefs_obj != 0)
1975     VERIFY(0 == zap_destroy(mos, ds->ds_phys->ds_userrefs_obj, tx));
1976 dsl_dir_close(ds->ds_dir, ds);
1977 ds->ds_dir = NULL;
1978 dsl_dataset_drain_refs(ds, tag);
1979 VERIFY(0 == dmu_object_free(mos, obj, tx));
1980
1981 if (dsda->rm_origin) {
1982     /*
1983      * Remove the origin of the clone we just destroyed.
1984      */
1985     struct dsl_ds_destroyarg ndsda = {0};
1986
1987     ndsda.ds = dsda->rm_origin;
1988     dsl_dataset_destroy_sync(&ndsda, tag, tx);
1989 }
1990 }
1991
1992 static int
1993 dsl_dataset_snapshot_reserve_space(dsl_dataset_t *ds, dmu_tx_t *tx)
1994 {
1995     uint64_t asize;
1996
1997     if (!dmu_tx_is_syncing(tx))
1998         return (0);
1999
2000     /*
2001      * If there's an fs-only reservation, any blocks that might become
2002      * owned by the snapshot dataset must be accommodated by space
2003      * outside of the reservation.
2004      */
2005     ASSERT(ds->ds_reserved == 0 || DS_UNIQUE_IS_ACCURATE(ds));
2006     asize = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
2007     if (asize > dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE))
2008         return (ENOSPC);
2009
2010     /*
2011      * Propagate any reserved space for this snapshot to other
2012      * Propogate any reserved space for this snapshot to other
2013      * snapshot checks in this sync group.
2014      */
2015     if (asize > 0)
2016         dsl_dir_willuse_space(ds->ds_dir, asize, tx);
2017
2018     return (0);
2019 }
2020
2021 int
2022 dsl_dataset_snapshot_check(dsl_dataset_t *ds, const char *snapname,
2023     dmu_tx_t *tx)
2024 dsl_dataset_snapshot_check(void *arg1, void *arg2, dmu_tx_t *tx)
2025 {
2026     dsl_dataset_t *ds = arg1;
2027     const char *snapname = arg2;
2028     int err;
2029     uint64_t value;
2030
2031     /*
2032      * We don't allow multiple snapshots of the same txg. If there

```

```

2029     * is already one, try again.
2030     */
2031     if (ds->ds_phys->ds_prev_snap_txg >= tx->tx_txg)
2032         return (EAGAIN);

2034     /*
2035     * Check for conflicting snapshot name.
2036     * Check for conflicting name snapshot name.
2037     */
2038     err = dsl_dataset_snap_lookup(ds, snapname, &value);
2039     if (err == 0)
2040         return (EEXIST);
2041     if (err != ENOENT)
2042         return (err);

2043     /*
2044     * Check that the dataset's name is not too long. Name consists
2045     * of the dataset's length + 1 for the @-sign + snapshot name's length
2046     */
2047     if (dsl_dataset_namelen(ds) + 1 + strlen(snapname) >= MAXNAMELEN)
2048         return (ENAMETOOLONG);

2050     err = dsl_dataset_snapshot_reserve_space(ds, tx);
2051     if (err)
2052         return (err);

2054     ds->ds_trysnap_txg = tx->tx_txg;
2055     return (0);
2056 }

2058 void
2059 dsl_dataset_snapshot_sync(dsl_dataset_t *ds, const char *snapname,
2060     dmu_tx_t *tx)
2061 {
2062     dsl_dataset_t *ds = arg1;
2063     const char *snapname = arg2;
2064     dsl_pool_t *dp = ds->ds_dir->dd_pool;
2065     dmu_buf_t *dbuf;
2066     dsl_dataset_phys_t *dsphys;
2067     uint64_t dsobj, crtngx;
2068     objset_t *mos = dp->dp_meta_objset;
2069     int err;

2069     ASSERT(RW_WRITE_HELD(&dp->dp_config_rwlock));

2071     /*
2072     * The origin's ds_creation_txg has to be < TXG_INITIAL
2073     */
2074     if (strcmp(snapname, ORIGIN_DIR_NAME) == 0)
2075         crtngx = 1;
2076     else
2077         crtngx = tx->tx_txg;

2079     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
2080         DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
2081     VERIFY(0 == dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
2082     dmu_buf_will_dirty(dbuf, tx);
2083     dsphys = dbuf->db_data;
2084     bzero(dsphys, sizeof (dsl_dataset_phys_t));
2085     dsphys->ds_dir_obj = ds->ds_dir->dd_object;
2086     dsphys->ds_fsid_guid = unique_create();
2087     (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
2088         sizeof (dsphys->ds_guid));
2089     dsphys->ds_prev_snap_obj = ds->ds_phys->ds_prev_snap_obj;
2090     dsphys->ds_prev_snap_txg = ds->ds_phys->ds_prev_snap_txg;

```

```

2091     dsphys->ds_next_snap_obj = ds->ds_object;
2092     dsphys->ds_num_children = 1;
2093     dsphys->ds_creation_time = gethrstime_sec();
2094     dsphys->ds_creation_txg = crtngx;
2095     dsphys->ds_deadlist_obj = ds->ds_phys->ds_deadlist_obj;
2096     dsphys->ds_referenced_bytes = ds->ds_phys->ds_referenced_bytes;
2097     dsphys->ds_compressed_bytes = ds->ds_phys->ds_compressed_bytes;
2098     dsphys->ds_uncompressed_bytes = ds->ds_phys->ds_uncompressed_bytes;
2099     dsphys->ds_flags = ds->ds_phys->ds_flags;
2100     dsphys->ds_bp = ds->ds_phys->ds_bp;
2101     dmu_buf_rele(dbuf, FTAG);

2103     ASSERT3U(ds->ds_prev != 0, ==, ds->ds_phys->ds_prev_snap_obj != 0);
2104     if (ds->ds_prev) {
2105         uint64_t next_clones_obj =
2106             ds->ds_prev->ds_phys->ds_next_clones_obj;
2107         ASSERT(ds->ds_prev->ds_phys->ds_next_snap_obj ==
2108             ds->ds_object ||
2109             ds->ds_prev->ds_phys->ds_num_children > 1);
2110         if (ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object) {
2111             dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
2112             ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
2113                 ds->ds_prev->ds_phys->ds_creation_txg);
2114             ds->ds_prev->ds_phys->ds_next_snap_obj = dsobj;
2115         } else if (next_clones_obj != 0) {
2116             remove_from_next_clones(ds->ds_prev,
2117                 dsphys->ds_next_snap_obj, tx);
2118             VERIFY3U(0, ==, zap_add_int(mos,
2119                 next_clones_obj, dsobj, tx));
2120         }
2121     }

2123     /*
2124     * If we have a reference-reservation on this dataset, we will
2125     * need to increase the amount of reservation being charged
2126     * since our unique space is going to zero.
2127     */
2128     if (ds->ds_reserved) {
2129         int64_t delta;
2130         ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
2131         delta = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
2132         dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV,
2133             delta, 0, 0, tx);
2134     }

2136     dmu_buf_will_dirty(ds->ds_dbuf, tx);
2137     zfs_dbgmsg("taking snapshot %s@%s%llu",
2138         ds->ds_dir->dd_myname, snapname, dsobj,
2139         ds->ds_phys->ds_prev_snap_txg);
2140     ds->ds_phys->ds_deadlist_obj = dsl_deadlist_clone(&ds->ds_deadlist,
2141         UINT64_MAX, ds->ds_phys->ds_prev_snap_obj, tx);
2142     dsl_deadlist_close(&ds->ds_deadlist);
2143     dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
2144     dsl_deadlist_add_key(&ds->ds_deadlist,
2145         ds->ds_phys->ds_prev_snap_txg, tx);

2147     ASSERT3U(ds->ds_phys->ds_prev_snap_txg, <, tx->tx_txg);
2148     ds->ds_phys->ds_prev_snap_obj = dsobj;
2149     ds->ds_phys->ds_prev_snap_txg = crtngx;
2150     ds->ds_phys->ds_unique_bytes = 0;
2151     if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
2152         ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

2154     err = zap_add(mos, ds->ds_phys->ds_snapnames_zapobj,
2155         snapname, 8, 1, &dsobj, tx);
2156     ASSERT(err == 0);

```



```

2344 uint64_t
2345 dsl_dataset_fsid_guid(dsl_dataset_t *ds)
2346 {
2347     return (ds->ds_fsid_guid);
2348 }

2350 void
2351 dsl_dataset_space(dsl_dataset_t *ds,
2352     uint64_t *refdbytesp, uint64_t *availbytesp,
2353     uint64_t *usedobjsp, uint64_t *availobjsp)
2354 {
2355     *refdbytesp = ds->ds_phys->ds_referenced_bytes;
2356     *availbytesp = dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE);
2357     if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes)
2358         *availbytesp += ds->ds_reserved - ds->ds_phys->ds_unique_bytes;
2359     if (ds->ds_quota != 0) {
2360         /*
2361          * Adjust available bytes according to refquota
2362          */
2363         if (*refdbytesp < ds->ds_quota)
2364             *availbytesp = MIN(*availbytesp,
2365                 ds->ds_quota - *refdbytesp);
2366         else
2367             *availbytesp = 0;
2368     }
2369     *usedobjsp = ds->ds_phys->ds_bp.blk_fill;
2370     *availobjsp = DN_MAX_OBJECT - *usedobjsp;
2371 }

2373 boolean_t
2374 dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds)
2375 {
2376     dsl_pool_t *dp = ds->ds_dir->dd_pool;

2378     ASSERT(RW_LOCK_HELD(&dp->dp_config_rwlock) ||
2379         dsl_pool_sync_context(dp));
2380     if (ds->ds_prev == NULL)
2381         return (B_FALSE);
2382     if (ds->ds_phys->ds_bp.blk_birth >
2383         ds->ds_prev->ds_phys->ds_creation_txg) {
2384         objset_t *os, *os_prev;
2385         /*
2386          * It may be that only the ZIL differs, because it was
2387          * reset in the head. Don't count that as being
2388          * modified.
2389          */
2390         if (dmu_objset_from_ds(ds, &os) != 0)
2391             return (B_TRUE);
2392         if (dmu_objset_from_ds(ds->ds_prev, &os_prev) != 0)
2393             return (B_TRUE);
2394         return (bcmp(&os->os_phys->os_meta_dnode,
2395             &os_prev->os_phys->os_meta_dnode,
2396             sizeof (os->os_phys->os_meta_dnode)) != 0);
2397     }
2398     return (B_FALSE);
2399 }

2401 /* ARGSUSED */
2402 static int
2403 dsl_dataset_snapshot_rename_check(void *arg1, void *arg2, dmu_tx_t *tx)
2404 {
2405     dsl_dataset_t *ds = arg1;
2406     char *newsnapname = arg2;
2407     dsl_dir_t *dd = ds->ds_dir;
2408     dsl_dataset_t *hds;
2409     uint64_t val;

```

```

2410     int err;

2412     err = dsl_dataset_hold_obj(dd->dd_pool,
2413         dd->dd_phys->dd_head_dataset_obj, FTAG, &hds);
2414     if (err)
2415         return (err);

2417     /* new name better not be in use */
2418     err = dsl_dataset_snap_lookup(hds, newsnapname, &val);
2419     dsl_dataset_rele(hds, FTAG);

2421     if (err == 0)
2422         err = EEXIST;
2423     else if (err == ENOENT)
2424         err = 0;

2426     /* dataset name + 1 for the "@" + the new snapshot name must fit */
2427     if (dsl_dir_namelen(ds->ds_dir) + 1 + strlen(newsnapname) >= MAXNAMELEN)
2428         err = ENAMETOOLONG;

2430     return (err);
2431 }

2433 static void
2434 dsl_dataset_snapshot_rename_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2435 {
2436     dsl_dataset_t *ds = arg1;
2437     const char *newsnapname = arg2;
2438     dsl_dir_t *dd = ds->ds_dir;
2439     objset_t *mos = dd->dd_pool->dp_meta_objset;
2440     dsl_dataset_t *hds;
2441     int err;

2443     ASSERT(ds->ds_phys->ds_next_snap_obj != 0);

2445     VERIFY(0 == dsl_dataset_hold_obj(dd->dd_pool,
2446         dd->dd_phys->dd_head_dataset_obj, FTAG, &hds));

2448     VERIFY(0 == dsl_dataset_get_snapname(ds));
2449     err = dsl_dataset_snap_remove(hds, ds->ds_snapname, tx);
2450     ASSERT3U(err, ==, 0);
2451     mutex_enter(&ds->ds_lock);
2452     (void) strcpy(ds->ds_snapname, newsnapname);
2453     mutex_exit(&ds->ds_lock);
2454     err = zap_add(mos, hds->ds_phys->ds_snapnames_zapobj,
2455         ds->ds_snapname, 8, 1, &ds->ds_object, tx);
2456     ASSERT3U(err, ==, 0);

2458     spa_history_log_internal_ds(ds, "rename", tx,
2459         "-> %s", newsnapname);
2003     spa_history_log_internal(LOG_DS_RENAME, dd->dd_pool->dp_spa, tx,
2004         "dataset = %llu", ds->ds_object);
2460     dsl_dataset_rele(hds, FTAG);
2461 }

unchanged_portion_omitted

2778 static void
2779 dsl_dataset_promote_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2780 {
2781     dsl_dataset_t *hds = arg1;
2782     struct promotearg *pa = arg2;
2783     struct promotenode *snap = list_head(&pa->shared_snaps);
2784     dsl_dataset_t *origin_ds = snap->ds;
2785     dsl_dataset_t *origin_head;
2786     dsl_dir_t *dd = hds->ds_dir;
2787     dsl_pool_t *dp = hds->ds_dir->dd_pool;

```

```

2788     dsl_dir_t *odd = NULL;
2789     uint64_t oldnext_obj;
2790     int64_t delta;

2792     ASSERT(0 == (hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE));

2794     snap = list_head(&pa->origin_snaps);
2795     origin_head = snap->ds;

2797     /*
2798      * We need to explicitly open odd, since origin_ds's dd will be
2799      * changing.
2800      */
2801     VERIFY(0 == dsl_dir_open_obj(dp, origin_ds->ds_dir->dd_object,
2802     NULL, FTAG, &odd));

2804     /* change origin's next snap */
2805     dmu_buf_will_dirty(origin_ds->ds_dbuf, tx);
2806     oldnext_obj = origin_ds->ds_phys->ds_next_snap_obj;
2807     snap = list_tail(&pa->clone_snaps);
2808     ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
2809     origin_ds->ds_phys->ds_next_snap_obj = snap->ds->ds_object;

2811     /* change the origin's next clone */
2812     if (origin_ds->ds_phys->ds_next_clones_obj) {
2813         remove_from_next_clones(origin_ds, snap->ds->ds_object, tx);
2814         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2815         origin_ds->ds_phys->ds_next_clones_obj,
2816         oldnext_obj, tx));
2817     }

2819     /* change origin */
2820     dmu_buf_will_dirty(dd->dd_dbuf, tx);
2821     ASSERT3U(dd->dd_phys->dd_origin_obj, ==, origin_ds->ds_object);
2822     dd->dd_phys->dd_origin_obj = odd->dd_phys->dd_origin_obj;
2823     dd->dd_origin_txg = origin_head->ds_dir->dd_origin_txg;
2824     dmu_buf_will_dirty(odd->dd_dbuf, tx);
2825     odd->dd_phys->dd_origin_obj = origin_ds->ds_object;
2826     origin_head->ds_dir->dd_origin_txg =
2827     origin_ds->ds_phys->ds_creation_txg;

2829     /* change dd clone entries */
2830     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2831         VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
2832         odd->dd_phys->dd_clones, hds->ds_object, tx));
2833         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2834         pa->origin_origin->ds_dir->dd_phys->dd_clones,
2835         hds->ds_object, tx));

2837         VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
2838         pa->origin_origin->ds_dir->dd_phys->dd_clones,
2839         origin_head->ds_object, tx));
2840         if (dd->dd_phys->dd_clones == 0) {
2841             dd->dd_phys->dd_clones = zap_create(dp->dp_meta_objset,
2842             DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
2843         }
2844         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2845         dd->dd_phys->dd_clones, origin_head->ds_object, tx));
2847     }

2849     /* move snapshots to this dir */
2850     for (snap = list_head(&pa->shared_snaps); snap;
2851          snap = list_next(&pa->shared_snaps, snap)) {
2852         dsl_dataset_t *ds = snap->ds;

```

```

2854         /* unregister props as dsl_dir is changing */
2855         if (ds->ds_objset) {
2856             dmu_objset_evict(ds->ds_objset);
2857             ds->ds_objset = NULL;
2858         }
2859         /* move snap name entry */
2860         VERIFY(0 == dsl_dataset_get_snapname(ds));
2861         VERIFY(0 == dsl_dataset_snap_remove(origin_head,
2862         ds->ds_snapname, tx));
2863         VERIFY(0 == zap_add(dp->dp_meta_objset,
2864         hds->ds_phys->ds_snapnames_zapobj, ds->ds_snapname,
2865         8, 1, &ds->ds_object, tx));

2867         /* change containing dsl_dir */
2868         dmu_buf_will_dirty(ds->ds_dbuf, tx);
2869         ASSERT3U(ds->ds_phys->ds_dir_obj, ==, odd->dd_object);
2870         ds->ds_phys->ds_dir_obj = dd->dd_object;
2871         ASSERT3P(ds->ds_dir, ==, odd);
2872         dsl_dir_close(ds->ds_dir, ds);
2873         VERIFY(0 == dsl_dir_open_obj(dp, dd->dd_object,
2874         NULL, ds, &ds->ds_dir));

2876         /* move any clone references */
2877         if (ds->ds_phys->ds_next_clones_obj &&
2878             spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2879             zap_cursor_t zc;
2880             zap_attribute_t za;

2882             for (zap_cursor_init(&zc, dp->dp_meta_objset,
2883             ds->ds_phys->ds_next_clones_obj);
2884                 zap_cursor_retrieve(&zc, &za) == 0;
2885                 zap_cursor_advance(&zc)) {
2886                 dsl_dataset_t *cnds;
2887                 uint64_t o;

2889                 if (za.za_first_integer == oldnext_obj) {
2890                     /*
2891                      * We've already moved the
2892                      * origin's reference.
2893                      */
2894                     continue;
2895                 }

2897                 VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
2898                 za.za_first_integer, FTAG, &cnds));
2899                 o = cnds->ds_dir->dd_phys->dd_head_dataset_obj;

2901                 VERIFY3U(zap_remove_int(dp->dp_meta_objset,
2902                 odd->dd_phys->dd_clones, o, tx), ==, 0);
2903                 VERIFY3U(zap_add_int(dp->dp_meta_objset,
2904                 dd->dd_phys->dd_clones, o, tx), ==, 0);
2905                 dsl_dataset_rele(cnds, FTAG);
2906             }
2907             zap_cursor_fini(&zc);
2908         }

2910         ASSERT3U(dsl_prop_numcb(ds), ==, 0);
2911     }

2913     /*
2914     * Change space accounting.
2915     * Note, pa->*usedsnap and dd_used_breakdown[SNAP] will either
2916     * both be valid, or both be 0 (resulting in delta == 0). This
2917     * is true for each of {clone,origin} independently.
2918     */

```

```

2920     delta = pa->cloneusedsnap -
2921         dd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2922     ASSERT3S(delta, >=, 0);
2923     ASSERT3U(pa->used, >=, delta);
2924     dsl_dir_diduse_space(dd, DD_USED_SNAP, delta, 0, 0, tx);
2925     dsl_dir_diduse_space(dd, DD_USED_HEAD,
2926         pa->used - delta, pa->comp, pa->uncomp, tx);

2928     delta = pa->originusedsnap -
2929         odd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2930     ASSERT3S(delta, <=, 0);
2931     ASSERT3U(pa->used, >=, -delta);
2932     dsl_dir_diduse_space(odd, DD_USED_SNAP, delta, 0, 0, tx);
2933     dsl_dir_diduse_space(odd, DD_USED_HEAD,
2934         -pa->used - delta, -pa->comp, -pa->uncomp, tx);

2936     origin_ds->ds_phys->ds_unique_bytes = pa->unique;

2938     /* log history record */
2939     spa_history_log_internal_ds(hds, "promote", tx, "");
2484     spa_history_log_internal(LOG_DS_PROMOTE, dd->dd_pool->dp_spa, tx,
2485         "dataset = %llu", hds->ds_object);

2941     dsl_dir_close(odd, FTAG);
2942 }
    unchanged portion omitted

3182 /* ARGSUSED */
3183 static void
3184 dsl_dataset_clone_swap_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3185 {
3186     struct cloneswaparg *csa = arg1;
3187     dsl_pool_t *dp = csa->cds->ds_dir->dd_pool;

3189     ASSERT(csa->cds->ds_reserved == 0);
3190     ASSERT(csa->ohds->ds_quota == 0 ||
3191         csa->cds->ds_phys->ds_unique_bytes <= csa->ohds->ds_quota);

3193     dmu_buf_will_dirty(csa->cds->ds_dbuf, tx);
3194     dmu_buf_will_dirty(csa->ohds->ds_dbuf, tx);

3196     if (csa->cds->ds_objset != NULL) {
3197         dmu_objset_evict(csa->cds->ds_objset);
3198         csa->cds->ds_objset = NULL;
3199     }

3201     if (csa->ohds->ds_objset != NULL) {
3202         dmu_objset_evict(csa->ohds->ds_objset);
3203         csa->ohds->ds_objset = NULL;
3204     }

3206     /*
3207      * Reset origin's unique bytes, if it exists.
3208      */
3209     if (csa->cds->ds_prev) {
3210         dsl_dataset_t *origin = csa->cds->ds_prev;
3211         uint64_t comp, uncomp;

3213         dmu_buf_will_dirty(origin->ds_dbuf, tx);
3214         dsl_deadlist_space_range(&csa->cds->ds_deadlist,
3215             origin->ds_phys->ds_prev_snap_txg, UINT64_MAX,
3216             &origin->ds_phys->ds_unique_bytes, &comp, &uncomp);
3217     }

3219     /* swap blkptrs */
3220     {

```

```

3221         blkptr_t tmp;
3222         tmp = csa->ohds->ds_phys->ds_bp;
3223         csa->ohds->ds_phys->ds_bp = csa->cds->ds_phys->ds_bp;
3224         csa->cds->ds_phys->ds_bp = tmp;
3225     }

3227     /* set dd*_bytes */
3228     {
3229         int64_t dused, dcomp, duncomp;
3230         uint64_t cdl_used, cdl_comp, cdl_uncomp;
3231         uint64_t odl_used, odl_comp, odl_uncomp;

3233         ASSERT3U(csa->cds->ds_dir->dd_phys->
3234             dd_used_breakdown[DD_USED_SNAP], ==, 0);

3236         dsl_deadlist_space(&csa->cds->ds_deadlist,
3237             &cdl_used, &cdl_comp, &cdl_uncomp);
3238         dsl_deadlist_space(&csa->ohds->ds_deadlist,
3239             &odl_used, &odl_comp, &odl_uncomp);

3241         dused = csa->cds->ds_phys->ds_referenced_bytes + cdl_used -
3242             (csa->ohds->ds_phys->ds_referenced_bytes + odl_used);
3243         dcomp = csa->cds->ds_phys->ds_compressed_bytes + cdl_comp -
3244             (csa->ohds->ds_phys->ds_compressed_bytes + odl_comp);
3245         duncomp = csa->cds->ds_phys->ds_uncompressed_bytes +
3246             cdl_uncomp -
3247             (csa->ohds->ds_phys->ds_uncompressed_bytes + odl_uncomp);

3249         dsl_dir_diduse_space(csa->ohds->ds_dir, DD_USED_HEAD,
3250             dused, dcomp, duncomp, tx);
3251         dsl_dir_diduse_space(csa->cds->ds_dir, DD_USED_HEAD,
3252             -dused, -dcomp, -duncomp, tx);

3254         /*
3255          * The difference in the space used by snapshots is the
3256          * difference in snapshot space due to the head's
3257          * deadlist (since that's the only thing that's
3258          * changing that affects the snapused).
3259          */
3260         dsl_deadlist_space_range(&csa->cds->ds_deadlist,
3261             csa->ohds->ds_dir->dd_origin_txg, UINT64_MAX,
3262             &cdl_used, &cdl_comp, &cdl_uncomp);
3263         dsl_deadlist_space_range(&csa->ohds->ds_deadlist,
3264             csa->ohds->ds_dir->dd_origin_txg, UINT64_MAX,
3265             &odl_used, &odl_comp, &odl_uncomp);
3266         dsl_dir_transfer_space(csa->ohds->ds_dir, cdl_used - odl_used,
3267             DD_USED_HEAD, DD_USED_SNAP, tx);
3268     }

3270     /* swap ds*_bytes */
3271     SWITCH64(csa->ohds->ds_phys->ds_referenced_bytes,
3272         csa->cds->ds_phys->ds_referenced_bytes);
3273     SWITCH64(csa->ohds->ds_phys->ds_compressed_bytes,
3274         csa->cds->ds_phys->ds_compressed_bytes);
3275     SWITCH64(csa->ohds->ds_phys->ds_uncompressed_bytes,
3276         csa->cds->ds_phys->ds_uncompressed_bytes);
3277     SWITCH64(csa->ohds->ds_phys->ds_unique_bytes,
3278         csa->cds->ds_phys->ds_unique_bytes);

3280     /* apply any parent delta for change in unconsumed reservation */
3281     dsl_dir_diduse_space(csa->ohds->ds_dir, DD_USED_REFRSRV,
3282         csa->unused_refres_delta, 0, 0, tx);

3284     /*
3285      * Swap deadlists.
3286      */

```

```

3287     dsl_deadlist_close(&csa->cds->ds_deadlist);
3288     dsl_deadlist_close(&csa->ohds->ds_deadlist);
3289     SWITCH64(csa->ohds->ds_phys->ds_deadlist_obj,
3290             csa->cds->ds_phys->ds_deadlist_obj);
3291     dsl_deadlist_open(&csa->cds->ds_deadlist, dp->dp_meta_objset,
3292                     csa->cds->ds_phys->ds_deadlist_obj);
3293     dsl_deadlist_open(&csa->ohds->ds_deadlist, dp->dp_meta_objset,
3294                     csa->ohds->ds_phys->ds_deadlist_obj);
3296     dsl_scan_ds_clone_swapped(csa->ohds, csa->cds, tx);
3298     spa_history_log_internal_ds(csa->cds, "clone swap", tx,
3299                               "parent=%s", csa->ohds->ds_dir->dd_myname);
3300 #endif /* ! codereview */
3301 }
3303 /*
3304  * Swap 'clone' with its origin head datasets. Used at the end of "zfs
3305  * recv" into an existing fs to swizzle the file system to the new
3306  * version, and by "zfs rollback". Can also be used to swap two
3307  * independent head datasets if neither has any snapshots.
3308  */
3309 int
3310 dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
3311                       boolean_t force)
3312 {
3313     struct cloneswaparg csa;
3314     int error;
3316     ASSERT(clone->ds_owner);
3317     ASSERT(origin_head->ds_owner);
3318     retry:
3319     /*
3320      * Need exclusive access for the swap. If we're swapping these
3321      * datasets back after an error, we already hold the locks.
3322      */
3323     if (!RW_WRITE_HELD(&clone->ds_rwlock))
3324         rw_enter(&clone->ds_rwlock, RW_WRITER);
3325     if (!RW_WRITE_HELD(&origin_head->ds_rwlock) &&
3326         !rw_tryenter(&origin_head->ds_rwlock, RW_WRITER)) {
3327         rw_exit(&clone->ds_rwlock);
3328         rw_enter(&origin_head->ds_rwlock, RW_WRITER);
3329         if (!rw_tryenter(&clone->ds_rwlock, RW_WRITER)) {
3330             rw_exit(&origin_head->ds_rwlock);
3331             goto retry;
3332         }
3333     }
3334     csa.cds = clone;
3335     csa.ohds = origin_head;
3336     csa.force = force;
3337     error = dsl_sync_task_do(clone->ds_dir->dd_pool,
3338                             dsl_dataset_clone_swap_check,
3339                             dsl_dataset_clone_swap_sync, &csa, NULL, 9);
3340     return (error);
3341 }
3343 /*
3344  * Given a pool name and a dataset object number in that pool,
3345  * return the name of that dataset.
3346  */
3347 int
3348 dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf)
3349 {
3350     spa_t *spa;
3351     dsl_pool_t *dp;
3352     dsl_dataset_t *ds;

```

```

3353     int error;
3355     if ((error = spa_open(pname, &spa, FTAG)) != 0)
3356         return (error);
3357     dp = spa_get_dsl(spa);
3358     rw_enter(&dp->dp_config_rwlock, RW_READER);
3359     if ((error = dsl_dataset_hold_obj(dp, obj, FTAG, &ds)) == 0) {
3360         dsl_dataset_name(ds, buf);
3361         dsl_dataset_rele(ds, FTAG);
3362     }
3363     rw_exit(&dp->dp_config_rwlock);
3364     spa_close(spa, FTAG);
3366     return (error);
3367 }
3369 int
3370 dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
3371                        uint64_t asize, uint64_t inflight, uint64_t *used, uint64_t *ref_rsrv)
3372 {
3373     int error = 0;
3375     ASSERT3S(asize, >, 0);
3377     /*
3378      * *ref_rsrv is the portion of asize that will come from any
3379      * unconsumed reservation space.
3380      */
3381     *ref_rsrv = 0;
3383     mutex_enter(&ds->ds_lock);
3384     /*
3385      * Make a space adjustment for reserved bytes.
3386      */
3387     if (ds->ds_reserved > ds->ds_phys->ds_unique_bytes) {
3388         ASSERT3U(*used, >=,
3389                 ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
3390         *used -= (ds->ds_reserved - ds->ds_phys->ds_unique_bytes);
3391         *ref_rsrv =
3392             asize - MIN(asize, parent_delta(ds, asize + inflight));
3393     }
3395     if (!check_quota || ds->ds_quota == 0) {
3396         mutex_exit(&ds->ds_lock);
3397         return (0);
3398     }
3399     /*
3400      * If they are requesting more space, and our current estimate
3401      * is over quota, they get to try again unless the actual
3402      * on-disk is over quota and there are no pending changes (which
3403      * may free up space for us).
3404      */
3405     if (ds->ds_phys->ds_referenced_bytes + inflight >= ds->ds_quota) {
3406         if (inflight > 0 ||
3407             ds->ds_phys->ds_referenced_bytes < ds->ds_quota)
3408             error = ERESTART;
3409         else
3410             error = EDQUOT;
3411     }
3412     mutex_exit(&ds->ds_lock);
3414     return (error);
3415 }
3417 /* ARGSUSED */
3418 static int

```



```

3419 dsl_dataset_set_quota_check(void *arg1, void *arg2, dmu_tx_t *tx)
3420 {
3421     dsl_dataset_t *ds = arg1;
3422     dsl_prop_setarg_t *psa = arg2;
3423     int err;
3424
3425     if (spa_version(ds->ds_dir->dd_pool->dp_spa) < SPA_VERSION_REFQUOTA)
3426         return (ENOTSUP);
3427
3428     if ((err = dsl_prop_predict_sync(ds->ds_dir, psa)) != 0)
3429         return (err);
3430
3431     if (psa->psa_effective_value == 0)
3432         return (0);
3433
3434     if (psa->psa_effective_value < ds->ds_phys->ds_referenced_bytes ||
3435         psa->psa_effective_value < ds->ds_reserved)
3436         return (ENOSPC);
3437
3438     return (0);
3439 }
3440
3441 extern void dsl_prop_set_sync(void *, void *, dmu_tx_t *);
3442
3443 void
3444 dsl_dataset_set_quota_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3445 {
3446     dsl_dataset_t *ds = arg1;
3447     dsl_prop_setarg_t *psa = arg2;
3448     uint64_t effective_value = psa->psa_effective_value;
3449
3450     dsl_prop_set_sync(ds, psa, tx);
3451     DSL_PROP_CHECK_PREDICTION(ds->ds_dir, psa);
3452
3453     if (ds->ds_quota != effective_value) {
3454         dmu_buf_will_dirty(ds->ds_dbuf, tx);
3455         ds->ds_quota = effective_value;
3456
3457         spa_history_log_internal_ds(ds, "set refquota", tx,
3458             "refquota=%lld", (longlong_t)ds->ds_quota);
3459         spa_history_log_internal(LOG_DS_REFQUOTA,
3460             ds->ds_dir->dd_pool->dp_spa, tx, "%lld dataset = %llu ",
3461             (longlong_t)ds->ds_quota, ds->ds_object);
3462     }
3463 }
3464
3465 unchanged portion omitted
3466
3467 static void
3468 dsl_dataset_set_reservation_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3469 {
3470     dsl_dataset_t *ds = arg1;
3471     dsl_prop_setarg_t *psa = arg2;
3472     uint64_t effective_value = psa->psa_effective_value;
3473     uint64_t unique;
3474     int64_t delta;
3475
3476     dsl_prop_set_sync(ds, psa, tx);
3477     DSL_PROP_CHECK_PREDICTION(ds->ds_dir, psa);
3478
3479     dmu_buf_will_dirty(ds->ds_dbuf, tx);
3480
3481     mutex_enter(&ds->ds_dir->dd_lock);
3482     mutex_enter(&ds->ds_lock);
3483     ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
3484     unique = ds->ds_phys->ds_unique_bytes;
3485     delta = MAX(0, (int64_t)(effective_value - unique)) -

```

```

3556         MAX(0, (int64_t)(ds->ds_reserved - unique));
3557     ds->ds_reserved = effective_value;
3558     mutex_exit(&ds->ds_lock);
3559
3560     dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV, delta, 0, 0, tx);
3561     mutex_exit(&ds->ds_dir->dd_lock);
3562
3563     spa_history_log_internal_ds(ds, "set reservation", tx,
3564         "reservation=%lld", (longlong_t)effective_value);
3565     spa_history_log_internal(LOG_DS_REFRESERV,
3566         ds->ds_dir->dd_pool->dp_spa, tx, "%lld dataset = %llu",
3567         (longlong_t)effective_value, ds->ds_object);
3568 }
3569
3570 unchanged portion omitted
3571
3572 /*
3573  * If you add new checks here, you may need to add
3574  * additional checks to the "temporary" case in
3575  * snapshot_check() in dmu_objset.c.
3576  */
3577 static int
3578 dsl_dataset_user_hold_check(void *arg1, void *arg2, dmu_tx_t *tx)
3579 {
3580     dsl_dataset_t *ds = arg1;
3581     struct dsl_ds_holdarg *ha = arg2;
3582     const char *htag = ha->htag;
3583     char *htag = ha->htag;
3584     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
3585     int error = 0;
3586
3587     if (spa_version(ds->ds_dir->dd_pool->dp_spa) < SPA_VERSION_USERREFS)
3588         return (ENOTSUP);
3589
3590     if (!dsl_dataset_is_snapshot(ds))
3591         return (EINVAL);
3592
3593     /* tags must be unique */
3594     mutex_enter(&ds->ds_lock);
3595     if (ds->ds_phys->ds_userrefs_obj) {
3596         error = zap_lookup(mos, ds->ds_phys->ds_userrefs_obj, htag,
3597             8, 1, tx);
3598         if (error == 0)
3599             error = EEXIST;
3600         else if (error == ENOENT)
3601             error = 0;
3602     }
3603     mutex_exit(&ds->ds_lock);
3604
3605     if (error == 0 && ha->temphold &&
3606         strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
3607         error = E2BIG;
3608
3609     return (error);
3610 }
3611
3612 void
3613 dsl_dataset_user_hold_sync(void *arg1, void *arg2, dmu_tx_t *tx)
3614 {
3615     dsl_dataset_t *ds = arg1;
3616     struct dsl_ds_holdarg *ha = arg2;
3617     const char *htag = ha->htag;
3618     char *htag = ha->htag;
3619     dsl_pool_t *dp = ds->ds_dir->dd_pool;
3620     objset_t *mos = dp->dp_meta_objset;
3621     uint64_t now = gethrstime_sec();
3622     uint64_t zapobj;

```

```

3670 mutex_enter(&ds->ds_lock);
3671 if (ds->ds_phys->ds_userrefs_obj == 0) {
3672     /*
3673      * This is the first user hold for this dataset. Create
3674      * the userrefs zap object.
3675      */
3676     dmu_buf_will_dirty(ds->ds_dbuf, tx);
3677     zapobj = ds->ds_phys->ds_userrefs_obj =
3678         zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
3679 } else {
3680     zapobj = ds->ds_phys->ds_userrefs_obj;
3681 }
3682 ds->ds_userrefs++;
3683 mutex_exit(&ds->ds_lock);

3685 VERIFY(0 == zap_add(mos, zapobj, htag, 8, 1, &now, tx));

3687 if (ha->temphold) {
3688     VERIFY(0 == dsl_pool_user_hold(dp, ds->ds_object,
3689         htag, &now, tx));
3690 }

3692 spa_history_log_internal(ds, "hold", tx,
3693     "tag = %s temp = %d holds now = %llu",
3694     htag, (int)ha->temphold, ds->ds_userrefs);
3695 spa_history_log_internal(LOG_DS_USER_HOLD,
3696     dp->dp_spa, tx, "<%s> temp = %d dataset = %llu", htag,
3697     (int)ha->temphold, ds->ds_object);
3698 }
3699 unchanged_portion_omitted

```

```

3893 static void
3894 dsl_dataset_user_release_sync(void *arg1, void *tag, dmu_tx_t *tx)
3895 {
3896     struct dsl_ds_releasearg *ra = arg1;
3897     dsl_dataset_t *ds = ra->ds;
3898     dsl_pool_t *dp = ds->ds_dir->dd_pool;
3899     objset_t *mos = dp->dp_meta_objset;
3900     uint64_t zapobj;
3901     uint64_t dsobj = ds->ds_object;
3902     uint64_t refs;
3903     int error;

3904     mutex_enter(&ds->ds_lock);
3905     ds->ds_userrefs--;
3906     refs = ds->ds_userrefs;
3907     mutex_exit(&ds->ds_lock);
3908     error = dsl_pool_user_release(dp, ds->ds_object, ra->htag, tx);
3909     VERIFY(error == 0 || error == ENOENT);
3910     zapobj = ds->ds_phys->ds_userrefs_obj;
3911     VERIFY(0 == zap_remove(mos, zapobj, ra->htag, tx));
3912     if (ds->ds_userrefs == 0 && ds->ds_phys->ds_num_children == 1 &&
3913         DS_IS_DEFER_DESTROY(ds)) {
3914         struct dsl_ds_destroyarg dsda = {0};

3916         ASSERT(ra->own);
3917         dsda.ds = ds;
3918         dsda.releasing = B_TRUE;
3919         /* We already did the destroy_check */
3920         dsl_dataset_destroy_sync(&dsda, tag, tx);
3921     }

3923 spa_history_log_internal(ds, "release", tx,
3924     "tag = %s refs now = %lld", ra->htag, (longlong_t)refs);
3925 spa_history_log_internal(LOG_DS_USER_RELEASE,

```

```

3313     dp->dp_spa, tx, "<%s> %lld dataset = %llu",
3314     ra->htag, (longlong_t)refs, dsobj);
3315 }
3316 unchanged_portion_omitted

```

```

*****
19263 Thu Jun 28 15:57:16 2012
new/usr/src/uts/common/fs/zfs/dsl_deleg.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

150 static void
151 dsl_deleg_set_sync(void *arg1, void *arg2, dmu_tx_t *tx)
152 {
153     dsl_dir_t *dd = arg1;
154     nvlist_t *nvp = arg2;
155     objset_t *mos = dd->dd_pool->dp_meta_objset;
156     nvpair_t *whopair = NULL;
157     uint64_t zapobj = dd->dd_phys->dd_deleg_zapobj;

159     if (zapobj == 0) {
160         dmu_buf_will_dirty(dd->dd_dbuf, tx);
161         zapobj = dd->dd_phys->dd_deleg_zapobj = zap_create(mos,
162             DMU_OT_DSL_PERMS, DMU_OT_NONE, 0, tx);
163     }

165     while (whopair = nvlist_next_nvpair(nvp, whopair)) {
166         const char *whokey = nvpair_name(whopair);
167         nvlist_t *perms;
168         nvpair_t *permpair = NULL;
169         uint64_t jumpobj;

171         VERIFY(nvpair_value_nvlist(whopair, &perms) == 0);

173         if (zap_lookup(mos, zapobj, whokey, 8, 1, &jumpobj) != 0) {
174             jumpobj = zap_create_link(mos, DMU_OT_DSL_PERMS,
175                 zapobj, whokey, tx);
176         }

178         while (permpair = nvlist_next_nvpair(perms, permpair)) {
179             const char *perm = nvpair_name(permpair);
180             uint64_t n = 0;

182             VERIFY(zap_update(mos, jumpobj,
183                 perm, 8, 1, &n, tx) == 0);
184             spa_history_log_internal_dd(dd, "permission update", tx,
185                 "%s %s", whokey, perm);
186             spa_history_log_internal(LOG_DS_PERM_UPDATE,
187                 dd->dd_pool->dp_spa, tx,
188                 "%s %s dataset = %llu", whokey, perm,
189                 dd->dd_phys->dd_head_dataset_obj);
186         }
187     }
188 }

190 static void
191 dsl_deleg_unset_sync(void *arg1, void *arg2, dmu_tx_t *tx)
192 {
193     dsl_dir_t *dd = arg1;
194     nvlist_t *nvp = arg2;
195     objset_t *mos = dd->dd_pool->dp_meta_objset;
196     nvpair_t *whopair = NULL;
197     uint64_t zapobj = dd->dd_phys->dd_deleg_zapobj;

```

```

199     if (zapobj == 0)
200         return;

202     while (whopair = nvlist_next_nvpair(nvp, whopair)) {
203         const char *whokey = nvpair_name(whopair);
204         nvlist_t *perms;
205         nvpair_t *permpair = NULL;
206         uint64_t jumpobj;

208         if (nvpair_value_nvlist(whopair, &perms) != 0) {
209             if (zap_lookup(mos, zapobj, whokey, 8,
210                 1, &jumpobj) == 0) {
211                 (void) zap_remove(mos, zapobj, whokey, tx);
212                 VERIFY(0 == zap_destroy(mos, jumpobj, tx));
213             }
214             spa_history_log_internal_dd(dd, "permission who remove",
215                 tx, "%s", whokey);
216             spa_history_log_internal(LOG_DS_PERM_WHO_REMOVE,
217                 dd->dd_pool->dp_spa, tx,
218                 "%s dataset = %llu", whokey,
219                 dd->dd_phys->dd_head_dataset_obj);
216         }
217         continue;

219         if (zap_lookup(mos, zapobj, whokey, 8, 1, &jumpobj) != 0)
220             continue;

222         while (permpair = nvlist_next_nvpair(perms, permpair)) {
223             const char *perm = nvpair_name(permpair);
224             uint64_t n = 0;

226             (void) zap_remove(mos, jumpobj, perm, tx);
227             if (zap_count(mos, jumpobj, &n) == 0 && n == 0) {
228                 (void) zap_remove(mos, zapobj,
229                     whokey, tx);
230                 VERIFY(0 == zap_destroy(mos,
231                     jumpobj, tx));
232             }
233             spa_history_log_internal_dd(dd, "permission remove", tx,
234                 "%s %s", whokey, perm);
237             spa_history_log_internal(LOG_DS_PERM_REMOVE,
238                 dd->dd_pool->dp_spa, tx,
239                 "%s %s dataset = %llu", whokey, perm,
240                 dd->dd_phys->dd_head_dataset_obj);
235         }
236     }
237 }
_____unchanged_portion_omitted_____

520 /*
521 * Check if user has requested permission.
522 * Check if user has requested permission. If descendent is set, must have
523 * descendent perms.
524 */
525 int
526 dsl_deleg_access_impl(dsl_dataset_t *ds, const char *perm, cred_t *cr)
527 dsl_deleg_access_impl(dsl_dataset_t *ds, boolean_t descendent, const char *perm,
528     cred_t *cr)
529 {
530     dsl_dir_t *dd;
531     dsl_pool_t *dp;
532     void *cookie;
533     int error;
534     char checkflag;
535     objset_t *mos;

```

```

532     avl_tree_t permsets;
533     perm_set_t *setnode;

535     dp = ds->ds_dir->dd_pool;
536     mos = dp->dp_meta_objset;

538     if (dsl_delegation_on(mos) == B_FALSE)
539         return (ECANCELED);

541     if (spa_version(dmu_objset_spa(dp->dp_meta_objset)) <
542         SPA_VERSION_DELEGATED_PERMS)
543         return (EPERM);

545     if (dsl_dataset_is_snapshot(ds)) {
553     if (dsl_dataset_is_snapshot(ds) || descendent) {
546         /*
547          * Snapshots are treated as descendents only,
548          * local permissions do not apply.
549          */
550         checkflag = ZFS_DELEG_DESCENDENT;
551     } else {
552         checkflag = ZFS_DELEG_LOCAL;
553     }

555     avl_create(&permsets, perm_set_compare, sizeof (perm_set_t),
556         offsetof(perm_set_t, p_node));

558     rw_enter(&dp->dp_config_rwlock, RW_READER);
559     for (dd = ds->ds_dir; dd != NULL; dd = dd->dd_parent,
560         checkflag = ZFS_DELEG_DESCENDENT) {
561         uint64_t zapobj;
562         boolean_t expanded;

564         /*
565          * If not in global zone then make sure
566          * the zoned property is set
567          */
568         if (!INGLOBALZONE(curproc)) {
569             uint64_t zoned;

571             if (dsl_prop_get_dd(dd,
572                 zfs_prop_to_name(ZFS_PROP_ZONED),
573                 8, 1, &zoned, NULL, B_FALSE) != 0)
574                 break;
575             if (!zoned)
576                 break;
577         }
578         zapobj = dd->dd_phys->dd_deleg_zapobj;

580         if (zapobj == 0)
581             continue;

583         dsl_load_user_sets(mos, zapobj, &permsets, checkflag, cr);
584     again:
585         expanded = B_FALSE;
586         for (setnode = avl_first(&permsets); setnode;
587             setnode = AVL_NEXT(&permsets, setnode)) {
588             if (setnode->p_matched == B_TRUE)
589                 continue;

591             /* See if this set directly grants this permission */
592             error = dsl_check_access(mos, zapobj,
593                 ZFS_DELEG_NAMED_SET, 0, setnode->p_setname, perm);
594             if (error == 0)
595                 goto success;
596             if (error == EPERM)

```

```

597         setnode->p_matched = B_TRUE;

599         /* See if this set includes other sets */
600         error = dsl_load_sets(mos, zapobj,
601             ZFS_DELEG_NAMED_SET_SETS, 0,
602             setnode->p_setname, &permsets);
603         if (error == 0)
604             setnode->p_matched = expanded = B_TRUE;
605     }
606     /*
607      * If we expanded any sets, that will define more sets,
608      * which we need to check.
609      */
610     if (expanded)
611         goto again;

613     error = dsl_check_user_access(mos, zapobj, perm, checkflag, cr);
614     if (error == 0)
615         goto success;
616     }
617     error = EPERM;
618 success:
619     rw_exit(&dp->dp_config_rwlock);

621     cookie = NULL;
622     while ((setnode = avl_destroy_nodes(&permsets, &cookie)) != NULL)
623         kmem_free(setnode, sizeof (perm_set_t));

625     return (error);
626 }

628 int
629 dsl_deleg_access(const char *dsname, const char *perm, cred_t *cr)
630 {
631     dsl_dataset_t *ds;
632     int error;

634     error = dsl_dataset_hold(dsname, FTAG, &ds);
635     if (error)
636         return (error);

638     error = dsl_deleg_access_impl(ds, perm, cr);
646     error = dsl_deleg_access_impl(ds, B_FALSE, perm, cr);
639     dsl_dataset_rele(ds, FTAG);

641     return (error);
642 }

```

unchanged_portion_omitted

new/usr/src/uts/common/fs/zfs/dsl_dir.c

1

```
*****
36590 Thu Jun 28 15:57:16 2012
new/usr/src/uts/common/fs/zfs/dsl_dir.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */

27 #include <sys/dmu.h>
28 #include <sys/dmu_objset.h>
29 #include <sys/dmu_tx.h>
30 #include <sys/dsl_dataset.h>
31 #include <sys/dsl_dir.h>
32 #include <sys/dsl_prop.h>
33 #include <sys/dsl_synctask.h>
34 #include <sys/dsl_deleg.h>
35 #include <sys/spa.h>
36 #include <sys/metastab.h>
37 #include <sys/zap.h>
38 #include <sys/zio.h>
39 #include <sys/arc.h>
40 #include <sys/sunddi.h>
41 #include "zfs_namecheck.h"

43 static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);
44 static void dsl_dir_set_reservation_sync_impl(dsl_dir_t *dd,
45     uint64_t value, dmu_tx_t *tx);
23 static void dsl_dir_set_reservation_sync(void *arg1, void *arg2, dmu_tx_t *tx);

47 /* ARGSUSED */
48 static void
49 dsl_dir_evict(dmu_buf_t *db, void *arg)
50 {
51     dsl_dir_t *dd = arg;
52     dsl_pool_t *dp = dd->dd_pool;
```

new/usr/src/uts/common/fs/zfs/dsl_dir.c

2

```
53     int t;
54
55     for (t = 0; t < TXG_SIZE; t++) {
56         ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
57         ASSERT(dd->dd_tempreserved[t] == 0);
58         ASSERT(dd->dd_space_towrite[t] == 0);
59     }
60
61     if (dd->dd_parent)
62         dsl_dir_close(dd->dd_parent, dd);
63
64     spa_close(dd->dd_pool->dp_spa, dd);
65
66     /*
67      * The props callback list should have been cleaned up by
68      * objset_evict().
69      */
70     list_destroy(&dd->dd_prop_cbs);
71     mutex_destroy(&dd->dd_lock);
72     kmem_free(dd, sizeof (dsl_dir_t));
73 }
unchanged portion omitted

448 /* ARGSUSED */
449 int
450 dsl_dir_destroy_check(void *arg1, void *arg2, dmu_tx_t *tx)
451 {
452     dsl_dir_t *dd = arg1;
453     dsl_dataset_t *ds = arg1;
454     dsl_dir_t *dd = ds->ds_dir;
455     dsl_pool_t *dp = dd->dd_pool;
456     objset_t *mos = dp->dp_meta_objset;
457     int err;
458     uint64_t count;
459
460     /*
461      * There should be exactly two holds, both from
462      * dsl_dataset_destroy: one on the dd directory, and one on its
463      * head ds. Otherwise, someone is trying to lookup something
464      * inside this dir while we want to destroy it. The
465      * config_rwlock ensures that nobody else opens it after we
466      * check.
467      */
468     if (dmu_buf_refcount(dd->dd_dbuf) > 2)
469         return (EBUSY);
470
471     err = zap_count(mos, dd->dd_phys->dd_child_dir_zapobj, &count);
472     if (err)
473         return (err);
474     if (count != 0)
475         return (EEXIST);
476
477     return (0);
478 }
479 void
480 dsl_dir_destroy_sync(void *arg1, void *tag, dmu_tx_t *tx)
481 {
482     dsl_dir_t *dd = arg1;
483     dsl_dataset_t *ds = arg1;
484     dsl_dir_t *dd = ds->ds_dir;
485     objset_t *mos = dd->dd_pool->dp_meta_objset;
486     dsl_prop_setarg_t psa;
487     uint64_t value = 0;
488     uint64_t obj;
489     dd_used_t t;
```

```

486     ASSERT(RW_WRITE_HELD(&dd->dd_pool->dp_config_rwlock));
487     ASSERT(dd->dd_phys->dd_head_dataset_obj == 0);

489     /*
490      * Remove our reservation. The impl() routine avoids setting the
491      * actual property, which would require the (already destroyed) ds.
492      */
493     dsl_dir_set_reservation_sync_impl(dd, 0, tx);
494     /* Remove our reservation. */
495     dsl_prop_setarg_init_uint64(&psa, "reservation",
496         (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED),
497         &value);
498     psa_psa_effective_value = 0; /* predict default value */

499     dsl_dir_set_reservation_sync(ds, &psa, tx);

500     ASSERT3U(dd->dd_phys->dd_used_bytes, ==, 0);
501     ASSERT3U(dd->dd_phys->dd_reserved, ==, 0);
502     for (t = 0; t < DD_USED_NUM; t++)
503         ASSERT3U(dd->dd_phys->dd_used_breakdown[t], ==, 0);

504     VERIFY(0 == zap_destroy(mos, dd->dd_phys->dd_child_dir_zapobj, tx));
505     VERIFY(0 == zap_destroy(mos, dd->dd_phys->dd_props_zapobj, tx));
506     VERIFY(0 == dsl_deleg_destroy(mos, dd->dd_phys->dd_deleg_zapobj, tx));
507     VERIFY(0 == zap_remove(mos,
508         dd->dd_parent->dd_phys->dd_child_dir_zapobj, dd->dd_myname, tx));

509     obj = dd->dd_object;
510     dsl_dir_close(dd, tag);
511     VERIFY(0 == dmu_object_free(mos, obj, tx));
512 }
513 _____unchanged_portion_omitted_____

1040 extern dsl_syncfunc_t dsl_prop_set_sync;

1042 static void
1043 dsl_dir_set_quota_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1044 {
1045     dsl_dataset_t *ds = arg1;
1046     dsl_dir_t *dd = ds->ds_dir;
1047     dsl_prop_setarg_t *psa = arg2;
1048     uint64_t effective_value = psa->psa_effective_value;

1049     dsl_prop_set_sync(ds, psa, tx);
1050     DSL_PROP_CHECK_PREDICTION(dd, psa);

1051     dmu_buf_will_dirty(dd->dd_dbuf, tx);

1052     mutex_enter(&dd->dd_lock);
1053     dd->dd_phys->dd_quota = effective_value;
1054     mutex_exit(&dd->dd_lock);

1055     spa_history_log_internal_dd(dd, "set quota", tx,
1056         "quota=%lld", (longlong_t)effective_value);
1057     spa_history_log_internal(LOG_DS_QUOTA, dd->dd_pool->dp_spa,
1058         tx, "%lld dataset = %llu ",
1059         (longlong_t)effective_value, dd->dd_phys->dd_head_dataset_obj);
1060 }
1061 _____unchanged_portion_omitted_____

1146 static void
1147 dsl_dir_set_reservation_sync_impl(dsl_dir_t *dd, uint64_t value, dmu_tx_t *tx)
1148 {
1149     dsl_dataset_t *ds = arg1;

```

```

1136     dsl_dir_t *dd = ds->ds_dir;
1137     dsl_prop_setarg_t *psa = arg2;
1138     uint64_t effective_value = psa->psa_effective_value;
1139     uint64_t used;
1140     int64_t delta;

1141     dsl_prop_set_sync(ds, psa, tx);
1142     DSL_PROP_CHECK_PREDICTION(dd, psa);

1143     dmu_buf_will_dirty(dd->dd_dbuf, tx);

1144     mutex_enter(&dd->dd_lock);
1145     used = dd->dd_phys->dd_used_bytes;
1146     delta = MAX(used, value) - MAX(used, dd->dd_phys->dd_reserved);
1147     dd->dd_phys->dd_reserved = value;
1148     delta = MAX(used, effective_value) -
1149         MAX(used, dd->dd_phys->dd_reserved);
1150     dd->dd_phys->dd_reserved = effective_value;

1151     if (dd->dd_parent != NULL) {
1152         /* Roll up this additional usage into our ancestors */
1153         dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
1154             delta, 0, 0, tx);
1155     }
1156     mutex_exit(&dd->dd_lock);
1157 }

1158 #endif /* !codereview */

1159 static void
1160 dsl_dir_set_reservation_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1161 {
1162     dsl_dataset_t *ds = arg1;
1163     dsl_dir_t *dd = ds->ds_dir;
1164     dsl_prop_setarg_t *psa = arg2;
1165     uint64_t value = psa->psa_effective_value;

1166     dsl_prop_set_sync(ds, psa, tx);
1167     DSL_PROP_CHECK_PREDICTION(dd, psa);

1168     dsl_dir_set_reservation_sync_impl(dd, value, tx);

1169     spa_history_log_internal_dd(dd, "set reservation", tx,
1170         "reservation=%lld", (longlong_t)value);
1171     spa_history_log_internal(LOG_DS_RESERVATION, dd->dd_pool->dp_spa,
1172         tx, "%lld dataset = %llu",
1173         (longlong_t)effective_value, dd->dd_phys->dd_head_dataset_obj);
1174 }
1175 _____unchanged_portion_omitted_____

1296 static void
1297 dsl_dir_rename_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1298 {
1299     dsl_dir_t *dd = arg1;
1300     struct renamearg *ra = arg2;
1301     dsl_pool_t *dp = dd->dd_pool;
1302     objset_t *mos = dp->dp_meta_objset;
1303     int err;
1304     char namebuf[MAXNAMELEN];
1305 #endif /* !codereview */

1306     ASSERT(dmu_buf_refcount(dd->dd_dbuf) <= 2);

1307     /* Log this before we change the name. */
1308     dsl_dir_name(ra->newparent, namebuf);
1309     spa_history_log_internal_dd(dd, "rename", tx,

```

```

1312         "-> %s/%s", namebuf, ra->mynewname);
1314 #endif /* ! codereview */
1315     if (ra->newparent != dd->dd_parent) {
1316         dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
1317             -dd->dd_phys->dd_used_bytes,
1318             -dd->dd_phys->dd_compressed_bytes,
1319             -dd->dd_phys->dd_uncompressed_bytes, tx);
1320         dsl_dir_diduse_space(ra->newparent, DD_USED_CHILD,
1321             dd->dd_phys->dd_used_bytes,
1322             dd->dd_phys->dd_compressed_bytes,
1323             dd->dd_phys->dd_uncompressed_bytes, tx);
1325         if (dd->dd_phys->dd_reserved > dd->dd_phys->dd_used_bytes) {
1326             uint64_t unused_rsrv = dd->dd_phys->dd_reserved -
1327                 dd->dd_phys->dd_used_bytes;
1329             dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
1330                 -unused_rsrv, 0, 0, tx);
1331             dsl_dir_diduse_space(ra->newparent, DD_USED_CHILD_RSRV,
1332                 unused_rsrv, 0, 0, tx);
1333         }
1334     }
1336     dmu_buf_will_dirty(dd->dd_dbuf, tx);
1338     /* remove from old parent zapobj */
1339     err = zap_remove(mos, dd->dd_parent->dd_phys->dd_child_dir_zapobj,
1340         dd->dd_myname, tx);
1341     ASSERT3U(err, ==, 0);
1343     (void) strcpy(dd->dd_myname, ra->mynewname);
1344     dsl_dir_close(dd->dd_parent, dd);
1345     dd->dd_phys->dd_parent_obj = ra->newparent->dd_object;
1346     VERIFY(0 == dsl_dir_open_obj(dd->dd_pool,
1347         ra->newparent->dd_object, NULL, dd, &dd->dd_parent));
1349     /* add to new parent zapobj */
1350     err = zap_add(mos, ra->newparent->dd_phys->dd_child_dir_zapobj,
1351         dd->dd_myname, 8, 1, &dd->dd_object, tx);
1352     ASSERT3U(err, ==, 0);
1282     spa_history_log_internal(LOG_DS_RENAME, dd->dd_pool->dp_spa,
1283         tx, "dataset = %llu", dd->dd_phys->dd_head_dataset_obj);
1354 }

```

unchanged portion omitted

```

*****
29576 Thu Jun 28 15:57:17 2012
new/usr/src/uts/common/fs/zfs/dsl_prop.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */
27 #include <sys/zfs_context.h>
28 #include <sys/dmu.h>
29 #include <sys/dmu_objset.h>
30 #include <sys/dmu_tx.h>
31 #include <sys/dsl_dataset.h>
32 #include <sys/dsl_dir.h>
33 #include <sys/dsl_prop.h>
34 #include <sys/dsl_synctask.h>
35 #include <sys/spa.h>
36 #include <sys/zap.h>
37 #include <sys/fs/zfs.h>
39 #include "zfs_prop.h"
41 #define ZPROP_INHERIT_SUFFIX "$inherit"
42 #define ZPROP_RECVD_SUFFIX "$recvd"
44 static int
45 dodefault(const char *propname, int intsz, int numints, void *buf)
46 {
47     zfs_prop_t prop;
49     /*
50      * The setonce properties are read-only, BUT they still
51      * have a default value that can be used as the initial
52      * value.
53      */
54     if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVALID ||

```

```

55     (zfs_prop_readonly(prop) && !zfs_prop_setonce(prop)))
56     return (ENOENT);
58     if (zfs_prop_get_type(prop) == PROP_TYPE_STRING) {
59         if (intsz != 1)
60             return (EOVERFLOW);
61         (void) strncpy(buf, zfs_prop_default_string(prop),
62             numints);
63     } else {
64         if (intsz != 8 || numints < 1)
65             return (EOVERFLOW);
67         *(uint64_t *)buf = zfs_prop_default_numeric(prop);
68     }
70     return (0);
71 }
73 int
74 dsl_prop_get_dd(dsl_dir_t *dd, const char *propname,
75     int intsz, int numints, void *buf, char *setpoint, boolean_t snapshot)
76 {
77     int err = ENOENT;
78     dsl_dir_t *target = dd;
79     objset_t *mos = dd->dd_pool->dp_meta_objset;
80     zfs_prop_t prop;
81     boolean_t inheritable;
82     boolean_t inheriting = B_FALSE;
83     char *inheritstr;
84     char *recvdstr;
86     ASSERT(RW_LOCK_HELD(&dd->dd_pool->dp_config_rwlock));
88     if (setpoint)
89         setpoint[0] = '\0';
91     prop = zfs_name_to_prop(propname);
92     inheritable = (prop == ZPROP_INVALID || zfs_prop_inheritable(prop));
93     inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
94     recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);
96     /*
97      * Note: dd may become NULL, therefore we shouldn't dereference it
98      * after this loop.
99      */
100    for (; dd != NULL; dd = dd->dd_parent) {
101        ASSERT(RW_LOCK_HELD(&dd->dd_pool->dp_config_rwlock));
103        if (dd != target || snapshot) {
104            if (!inheritable)
105                break;
106            inheriting = B_TRUE;
107        }
109        /* Check for a local value. */
110        err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj, propname,
111            intsz, numints, buf);
112        if (err != ENOENT) {
113            if (setpoint != NULL && err == 0)
114                dsl_dir_name(dd, setpoint);
115            break;
116        }
118        /*
119         * Skip the check for a received value if there is an explicit
120         * inheritance entry.

```



```

121     */
122     err = zap_contains(mos, dd->dd_phys->dd_props_zapobj,
123         inheritstr);
124     if (err != 0 && err != ENOENT)
125         break;
126
127     if (err == ENOENT) {
128         /* Check for a received value. */
129         err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj,
130             recvdstr, intsz, numints, buf);
131         if (err != ENOENT) {
132             if (setpoint != NULL && err == 0) {
133                 if (inheriting) {
134                     dsl_dir_name(dd, setpoint);
135                 } else {
136                     (void) strcpy(setpoint,
137                         ZPROP_SOURCE_VAL_RECVD);
138                 }
139             }
140             break;
141         }
142     }
143
144     /*
145     * If we found an explicit inheritance entry, err is zero even
146     * though we haven't yet found the value, so reinitializing err
147     * at the end of the loop (instead of at the beginning) ensures
148     * that err has a valid post-loop value.
149     */
150     err = ENOENT;
151 }
152
153 if (err == ENOENT)
154     err = dodefault(propname, intsz, numints, buf);
155
156 strfree(inheritstr);
157 strfree(recvdstr);
158
159 return (err);
160 }
161
162 int
163 dsl_prop_get_ds(dsl_dataset_t *ds, const char *propname,
164     int intsz, int numints, void *buf, char *setpoint)
165 {
166     zfs_prop_t prop = zfs_name_to_prop(propname);
167     boolean_t inheritable;
168     boolean_t snapshot;
169     uint64_t zapobj;
170
171     ASSERT(RW_LOCK_HELD(&ds->ds_dir->dd_pool->dp_config_rwlock));
172     inheritable = (prop == ZPROP_INVALID || zfs_prop_inheritable(prop));
173     snapshot = (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds));
174     zapobj = (ds->ds_phys == NULL ? 0 : ds->ds_phys->ds_props_obj);
175
176     if (zapobj != 0) {
177         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
178         int err;
179
180         ASSERT(snapshot);
181
182         /* Check for a local value. */
183         err = zap_lookup(mos, zapobj, propname, intsz, numints, buf);
184         if (err != ENOENT) {
185             if (setpoint != NULL && err == 0)
186                 dsl_dataset_name(ds, setpoint);

```

```

187         return (err);
188     }
189
190     /*
191     * Skip the check for a received value if there is an explicit
192     * inheritance entry.
193     */
194     if (inheritable) {
195         char *inheritstr = kmem_asprintf("%s%s", propname,
196             ZPROP_INHERIT_SUFFIX);
197         err = zap_contains(mos, zapobj, inheritstr);
198         strfree(inheritstr);
199         if (err != 0 && err != ENOENT)
200             return (err);
201     }
202
203     if (err == ENOENT) {
204         /* Check for a received value. */
205         char *recvdstr = kmem_asprintf("%s%s", propname,
206             ZPROP_RECVD_SUFFIX);
207         err = zap_lookup(mos, zapobj, recvdstr,
208             intsz, numints, buf);
209         strfree(recvdstr);
210         if (err != ENOENT) {
211             if (setpoint != NULL && err == 0)
212                 (void) strcpy(setpoint,
213                     ZPROP_SOURCE_VAL_RECVD);
214             return (err);
215         }
216     }
217 }
218
219 return (dsl_prop_get_dd(ds->ds_dir, propname,
220     intsz, numints, buf, setpoint, snapshot));
221 }
222
223 /*
224 * Register interest in the named property. We'll call the callback
225 * once to notify it of the current property value, and again each time
226 * the property changes, until this callback is unregistered.
227 *
228 * Return 0 on success, errno if the prop is not an integer value.
229 */
230 int
231 dsl_prop_register(dsl_dataset_t *ds, const char *propname,
232     dsl_prop_changed_cb_t *callback, void *cbarg)
233 {
234     dsl_dir_t *dd = ds->ds_dir;
235     dsl_pool_t *dp = dd->dd_pool;
236     uint64_t value;
237     dsl_prop_cb_record_t *cbr;
238     int err;
239     int need_rwlock;
240
241     need_rwlock = !RW_WRITE_HELD(&dp->dp_config_rwlock);
242     if (need_rwlock)
243         rw_enter(&dp->dp_config_rwlock, RW_READER);
244
245     err = dsl_prop_get_ds(ds, propname, 8, 1, &value, NULL);
246     if (err != 0) {
247         if (need_rwlock)
248             rw_exit(&dp->dp_config_rwlock);
249         return (err);
250     }
251
252     cbr = kmem_alloc(sizeof (dsl_prop_cb_record_t), KM_SLEEP);

```

```

253     cbr->cbr_ds = ds;
254     cbr->cbr_propname = kmem_alloc(strlen(propname)+1, KM_SLEEP);
255     (void) strcpy((char *)cbr->cbr_propname, propname);
256     cbr->cbr_func = callback;
257     cbr->cbr_arg = cbar;
258     mutex_enter(&dd->dd_lock);
259     list_insert_head(&dd->dd_prop_cbs, cbr);
260     mutex_exit(&dd->dd_lock);

262     cbr->cbr_func(cbr->cbr_arg, value);

264     if (need_rwlock)
265         rw_exit(&dp->dp_config_rwlock);
266     return (0);
267 }

269 int
270 dsl_prop_get(const char *dsname, const char *propname,
271             int intsz, int numints, void *buf, char *setpoint)
272 {
273     dsl_dataset_t *ds;
274     int err;

276     err = dsl_dataset_hold(dsname, FTAG, &ds);
277     if (err)
278         return (err);

280     rw_enter(&ds->ds_dir->dd_pool->dp_config_rwlock, RW_READER);
281     err = dsl_prop_get_ds(ds, propname, intsz, numints, buf, setpoint);
282     rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);

284     dsl_dataset_rele(ds, FTAG);
285     return (err);
286 }

288 /*
289  * Get the current property value. It may have changed by the time this
290  * function returns, so it is NOT safe to follow up with
291  * dsl_prop_register() and assume that the value has not changed in
292  * between.
293  *
294  * Return 0 on success, ENOENT if dsname is invalid.
295  */
296 int
297 dsl_prop_get_integer(const char *ddname, const char *propname,
298                    uint64_t *valuep, char *setpoint)
299 {
300     return (dsl_prop_get(ddname, propname, 8, 1, valuep, setpoint));
301 }

303 void
304 dsl_prop_setarg_init_uint64(dsl_prop_setarg_t *psa, const char *propname,
305                            zprop_source_t source, uint64_t *value)
306 {
307     psa->psa_name = propname;
308     psa->psa_source = source;
309     psa->psa_intsz = 8;
310     psa->psa_numints = 1;
311     psa->psa_value = value;

313     psa->psa_effective_value = -1ULL;
314 }

316 /*
317  * Predict the effective value of the given special property if it were set with
318  * the given value and source. This is not a general purpose function. It exists

```

```

319  * only to handle the special requirements of the quota and reservation
320  * properties. The fact that these properties are non-inheritable greatly
321  * simplifies the prediction logic.
322  *
323  * Returns 0 on success, a positive error code on failure, or -1 if called with
324  * a property not handled by this function.
325  */
326 int
327 dsl_prop_predict_sync(dsl_dir_t *dd, dsl_prop_setarg_t *psa)
328 {
329     const char *propname = psa->psa_name;
330     zfs_prop_t prop = zfs_name_to_prop(propname);
331     zprop_source_t source = psa->psa_source;
332     objset_t *mos;
333     uint64_t zapobj;
334     uint64_t version;
335     char *recvdstr;
336     int err = 0;

338     switch (prop) {
339     case ZFS_PROP_QUOTA:
340     case ZFS_PROP_RESERVATION:
341     case ZFS_PROP_REFQUOTA:
342     case ZFS_PROP_REFRESERVATION:
343         break;
344     default:
345         return (-1);
346     }

348     mos = dd->dd_pool->dp_meta_objset;
349     zapobj = dd->dd_phys->dd_props_zapobj;
350     recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

352     version = spa_version(dd->dd_pool->dp_spa);
353     if (version < SPA_VERSION_RECVD_PROPS) {
354         if (source & ZPROP_SRC_NONE)
355             source = ZPROP_SRC_NONE;
356         else if (source & ZPROP_SRC_RECEIVED)
357             source = ZPROP_SRC_LOCAL;
358     }

360     switch (source) {
361     case ZPROP_SRC_NONE:
362         /* Revert to the received value, if any. */
363         err = zap_lookup(mos, zapobj, recvdstr, 8, 1,
364                        &psa->psa_effective_value);
365         if (err == ENOENT)
366             psa->psa_effective_value = 0;
367         break;
368     case ZPROP_SRC_LOCAL:
369         psa->psa_effective_value = *(uint64_t *)psa->psa_value;
370         break;
371     case ZPROP_SRC_RECEIVED:
372         /*
373          * If there's no local setting, then the new received value will
374          * be the effective value.
375          */
376         err = zap_lookup(mos, zapobj, propname, 8, 1,
377                        &psa->psa_effective_value);
378         if (err == ENOENT)
379             psa->psa_effective_value = *(uint64_t *)psa->psa_value;
380         break;
381     case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
382         /*
383          * We're clearing the received value, so the local setting (if
384          * it exists) remains the effective value.

```

```

385     */
386     err = zap_lookup(mos, zapobj, propname, 8, 1,
387                    &psa->psa_effective_value);
388     if (err == ENOENT)
389         psa->psa_effective_value = 0;
390     break;
391 default:
392     cmn_err(CE_PANIC, "unexpected property source: %d", source);
393 }
394
395     strfree(recvdstr);
396
397     if (err == ENOENT)
398         return (0);
399
400     return (err);
401 }
402
403 #ifdef ZFS_DEBUG
404 void
405 dsl_prop_check_prediction(dsl_dir_t *dd, dsl_prop_setarg_t *psa)
406 {
407     zfs_prop_t prop = zfs_name_to_prop(psa->psa_name);
408     uint64_t intval;
409     char setpoint[MAXNAMELEN];
410     uint64_t version = spa_version(dd->dd_pool->dp_spa);
411     int err;
412
413     if (version < SPA_VERSION_RECVD_PROPS) {
414         switch (prop) {
415             case ZFS_PROP_QUOTA:
416             case ZFS_PROP_RESERVATION:
417                 return;
418         }
419     }
420
421     err = dsl_prop_get_dd(dd, psa->psa_name, 8, 1, &intval,
422                          setpoint, B_FALSE);
423     if (err == 0 && intval != psa->psa_effective_value) {
424         cmn_err(CE_PANIC, "%s property, source: %x, "
425              "predicted effective value: %llu, "
426              "actual effective value: %llu (setpoint: %s)",
427              psa->psa_name, psa->psa_source,
428              (unsigned long long)psa->psa_effective_value,
429              (unsigned long long)intval, setpoint);
430     }
431 }
432 #endif
433
434 /*
435  * Unregister this callback. Return 0 on success, ENOENT if ddname is
436  * invalid, ENOMSG if no matching callback registered.
437  */
438 int
439 dsl_prop_unregister(dsl_dataset_t *ds, const char *propname,
440                   dsl_prop_changed_cb_t *callback, void *cbarg)
441 {
442     dsl_dir_t *dd = ds->ds_dir;
443     dsl_prop_cb_record_t *cbr;
444
445     mutex_enter(&dd->dd_lock);
446     for (cbr = list_head(&dd->dd_prop_cbs);
447          cbr; cbr = list_next(&dd->dd_prop_cbs, cbr)) {
448         if (cbr->cbr_ds == ds &&
449             cbr->cbr_func == callback &&
450             cbr->cbr_arg == cbarg &&

```

```

451         strcmp(cbr->cbr_propname, propname) == 0)
452             break;
453     }
454
455     if (cbr == NULL) {
456         mutex_exit(&dd->dd_lock);
457         return (ENOMSG);
458     }
459
460     list_remove(&dd->dd_prop_cbs, cbr);
461     mutex_exit(&dd->dd_lock);
462     kmem_free((void*)cbr->cbr_propname, strlen(cbr->cbr_propname)+1);
463     kmem_free(cbr, sizeof(dsl_prop_cb_record_t));
464
465     return (0);
466 }
467
468 /*
469  * Return the number of callbacks that are registered for this dataset.
470  */
471 int
472 dsl_prop_numcb(dsl_dataset_t *ds)
473 {
474     dsl_dir_t *dd = ds->ds_dir;
475     dsl_prop_cb_record_t *cbr;
476     int num = 0;
477
478     mutex_enter(&dd->dd_lock);
479     for (cbr = list_head(&dd->dd_prop_cbs);
480          cbr; cbr = list_next(&dd->dd_prop_cbs, cbr)) {
481         if (cbr->cbr_ds == ds)
482             num++;
483     }
484     mutex_exit(&dd->dd_lock);
485
486     return (num);
487 }
488
489 static void
490 dsl_prop_changed_notify(dsl_pool_t *dp, uint64_t ddojb,
491                       const char *propname, uint64_t value, int first)
492 {
493     dsl_dir_t *dd;
494     dsl_prop_cb_record_t *cbr;
495     objset_t *mos = dp->dp_meta_objset;
496     zap_cursor_t zc;
497     zap_attribute_t *za;
498     int err;
499
500     ASSERT(RW_WRITE_HELD(&dp->dp_config_rwlock));
501     err = dsl_dir_open_obj(dp, ddojb, NULL, FTAG, &dd);
502     if (err)
503         return;
504
505     if (!first) {
506         /*
507          * If the prop is set here, then this change is not
508          * being inherited here or below; stop the recursion.
509          */
510         err = zap_contains(mos, dd->dd_phys->dd_props_zapobj, propname);
511         if (err == 0) {
512             dsl_dir_close(dd, FTAG);
513             return;
514         }
515         ASSERT3U(err, ==, ENOENT);
516     }

```

```

518     mutex_enter(&dd->dd_lock);
519     for (cbr = list_head(&dd->dd_prop_cbs); cbr;
520          cbr = list_next(&dd->dd_prop_cbs, cbr)) {
521         uint64_t propobj = cbr->cbr_ds->ds_phys->ds_props_obj;

523         if (strcmp(cbr->cbr_propname, propname) != 0)
524             continue;

526         /*
527          * If the property is set on this ds, then it is not
528          * inherited here; don't call the callback.
529          */
530         if (propobj && 0 == zap_contains(mos, propobj, propname))
531             continue;

533         cbr->cbr_func(cbr->cbr_arg, value);
534     }
535     mutex_exit(&dd->dd_lock);

537     za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
538     for (zap_cursor_init(&z, mos,
539                        dd->dd_phys->dd_child_dir_zapobj);
540          zap_cursor_retrieve(&z, za) == 0;
541          zap_cursor_advance(&z)) {
542         dsl_prop_changed_notify(dp, za->za_first_integer,
543                                propname, value, FALSE);
544     }
545     kmem_free(za, sizeof (zap_attribute_t));
546     zap_cursor_fini(&z);
547     dsl_dir_close(dd, FTAG);
548 }

550 void
551 dsl_prop_set_sync(void *arg1, void *arg2, dmu_tx_t *tx)
552 {
553     dsl_dataset_t *ds = arg1;
554     dsl_prop_setarg_t *psa = arg2;
555     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
556     uint64_t zapobj, intval, dummy;
557     int isint;
558     char valbuf[32];
559     char *valstr = NULL;
560     char *inheritstr;
561     char *recvdstr;
562     char *tbuf = NULL;
563     int err;
564     uint64_t version = spa_version(ds->ds_dir->dd_pool->dp_spa);
565     const char *propname = psa->psa_name;
566     zprop_source_t source = psa->psa_source;

568     isint = (dodefult(propname, 8, 1, &intval) == 0);

570     if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
571         ASSERT(version >= SPA_VERSION_SNAP_PROPS);
572         if (ds->ds_phys->ds_props_obj == 0) {
573             dmu_buf_will_dirty(ds->ds_dbuf, tx);
574             ds->ds_phys->ds_props_obj =
575                 zap_create(mos,
576                           DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
577         }
578         zapobj = ds->ds_phys->ds_props_obj;
579     } else {
580         zapobj = ds->ds_dir->dd_phys->dd_props_zapobj;
581     }

```

```

583     if (version < SPA_VERSION_RECVD_PROPS) {
584         zfs_prop_t prop = zfs_name_to_prop(propname);
585         if (prop == ZFS_PROP_QUOTA || prop == ZFS_PROP_RESERVATION)
586             return;

588         if (source & ZPROP_SRC_NONE)
589             source = ZPROP_SRC_NONE;
590         else if (source & ZPROP_SRC_RECEIVED)
591             source = ZPROP_SRC_LOCAL;
592     }

594     inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
595     recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

597     switch (source) {
598     case ZPROP_SRC_NONE:
599         /*
600          * revert to received value, if any (inherit -S)
601          * - remove propname
602          * - remove propname$inherit
603          */
604         err = zap_remove(mos, zapobj, propname, tx);
605         ASSERT(err == 0 || err == ENOENT);
606         err = zap_remove(mos, zapobj, inheritstr, tx);
607         ASSERT(err == 0 || err == ENOENT);
608         break;
609     case ZPROP_SRC_LOCAL:
610         /*
611          * remove propname$inherit
612          * set propname -> value
613          */
614         err = zap_remove(mos, zapobj, inheritstr, tx);
615         ASSERT(err == 0 || err == ENOENT);
616         VERIFY(0 == zap_update(mos, zapobj, propname,
617                                psa->psa_intsz, psa->psa_numints, psa->psa_value, tx));
618         break;
619     case ZPROP_SRC_INHERITED:
620         /*
621          * explicitly inherit
622          * - remove propname
623          * - set propname$inherit
624          */
625         err = zap_remove(mos, zapobj, propname, tx);
626         ASSERT(err == 0 || err == ENOENT);
627         if (version >= SPA_VERSION_RECVD_PROPS &&
628             dsl_prop_get_ds(ds, ZPROP_HAS_RECVD, 8, 1, &dummy,
629                             NULL) == 0) {
630             dummy = 0;
631             err = zap_update(mos, zapobj, inheritstr,
632                             8, 1, &dummy, tx);
633             ASSERT(err == 0);
634         }
635         break;
636     case ZPROP_SRC_RECEIVED:
637         /*
638          * set propname$recvd -> value
639          */
640         err = zap_update(mos, zapobj, recvdstr,
641                          psa->psa_intsz, psa->psa_numints, psa->psa_value, tx);
642         ASSERT(err == 0);
643         break;
644     case (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED):
645         /*
646          * clear local and received settings
647          * - remove propname
648          * - remove propname$inherit

```

```

649     * - remove propname$recvd
650     */
651     err = zap_remove(mos, zapobj, propname, tx);
652     ASSERT(err == 0 || err == ENOENT);
653     err = zap_remove(mos, zapobj, inheritstr, tx);
654     ASSERT(err == 0 || err == ENOENT);
655     /* FALLTHRU */
656     case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
657     /*
658     * remove propname$recvd
659     */
660     err = zap_remove(mos, zapobj, recvdstr, tx);
661     ASSERT(err == 0 || err == ENOENT);
662     break;
663     default:
664     cmn_err(CE_PANIC, "unexpected property source: %d", source);
665     }
666
667     strfree(inheritstr);
668     strfree(recvdstr);
669
670     if (isint) {
671     VERIFY(0 == dsl_prop_get_ds(ds, propname, 8, 1, &intval, NULL));
672
673     if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
674     dsl_prop_cb_record_t *cbr;
675     /*
676     * It's a snapshot; nothing can inherit this
677     * property, so just look for callbacks on this
678     * ds here.
679     */
680     mutex_enter(&ds->ds_dir->dd_lock);
681     for (cbr = list_head(&ds->ds_dir->dd_prop_cbs); cbr;
682     cbr = list_next(&ds->ds_dir->dd_prop_cbs, cbr)) {
683     if (cbr->cbr_ds == ds &&
684     strcmp(cbr->cbr_propname, propname) == 0)
685     cbr->cbr_func(cbr->cbr_arg, intval);
686     }
687     mutex_exit(&ds->ds_dir->dd_lock);
688     } else {
689     dsl_prop_changed_notify(ds->ds_dir->dd_pool,
690     ds->ds_dir->dd_object, propname, intval, TRUE);
691     }
692
693     (void) snprintf(valbuf, sizeof(valbuf),
694     "%lld", (longlong_t)intval);
695     valstr = valbuf;
696     } else {
697     if (source == ZPROP_SRC_LOCAL) {
698     valstr = (char *)psa->psa_value;
699     } else {
700     tbuf = kmem_alloc(ZAP_MAXVALUELEN, KM_SLEEP);
701     if (dsl_prop_get_ds(ds, propname, 1,
702     ZAP_MAXVALUELEN, tbuf, NULL) == 0)
703     valstr = tbuf;
704     }
705     }
706
707     spa_history_log_internal_ds(ds, (source == ZPROP_SRC_NONE ||
708     source == ZPROP_SRC_INHERITED) ? "inherit" : "set", tx,
709     "%s=%s", propname, (valstr == NULL ? "" : valstr));
710     spa_history_log_internal((source == ZPROP_SRC_NONE ||
711     source == ZPROP_SRC_INHERITED) ? LOG_DS_INHERIT :
712     LOG_DS_PROPSSET, ds->ds_dir->dd_pool->dp_spa, tx,
713     "%s=%s dataset = %llu", propname,
714     (valstr == NULL ? "" : valstr), ds->ds_object);

```

```

711     if (tbuf != NULL)
712     kmem_free(tbuf, ZAP_MAXVALUELEN);
713 }
714
715     unchanged_portion_omitted
716
717     void
718     dsl_dir_prop_set_uint64_sync(dsl_dir_t *dd, const char *name, uint64_t val,
719     dmu_tx_t *tx)
720     {
721     objset_t *mos = dd->dd_pool->dp_meta_objset;
722     uint64_t zapobj = dd->dd_phys->dd_props_zapobj;
723
724     ASSERT(dmu_tx_is_syncing(tx));
725
726     VERIFY(0 == zap_update(mos, zapobj, name, sizeof(val), 1, &val, tx));
727
728     dsl_prop_changed_notify(dd->dd_pool, dd->dd_object, name, val, TRUE);
729
730     spa_history_log_internal(LOG_DS_PROPSSET, dd->dd_pool->dp_spa, tx,
731     "%s=%llu dataset = %llu", name, (u_longlong_t)val,
732     dd->dd_phys->dd_head_dataset_obj);
733 }
734
735     int
736     dsl_prop_set(const char *dsname, const char *propname, zprop_source_t source,
737     int intsz, int numints, const void *buf)
738     {
739     dsl_dataset_t *ds;
740     uint64_t version;
741     int err;
742     dsl_prop_setarg_t psa;
743
744     /*
745     * We must do these checks before we get to the syncfunc, since
746     * it can't fail.
747     */
748     if (strlen(propname) >= ZAP_MAXNAMELEN)
749     return (ENAMETOOLONG);
750
751     err = dsl_dataset_hold(dsname, FTAG, &ds);
752     if (err)
753     return (err);
754
755     version = spa_version(ds->ds_dir->dd_pool->dp_spa);
756     if (intsz * numints >= (version < SPA_VERSION_STMF_PROP ?
757     ZAP_OLDMAXVALUELEN : ZAP_MAXVALUELEN)) {
758     dsl_dataset_rele(ds, FTAG);
759     return (E2BIG);
760     }
761     if (dsl_dataset_is_snapshot(ds) &&
762     version < SPA_VERSION_SNAP_PROPS) {
763     dsl_dataset_rele(ds, FTAG);
764     return (ENOTSUP);
765     }
766
767     psa.psa_name = propname;
768     psa.psa_source = source;
769     psa.psa_intsz = intsz;
770     psa.psa_numints = numints;
771     psa.psa_value = buf;
772     psa.psa_effective_value = -1ULL;
773
774     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
775     NULL, dsl_prop_set_sync, ds, &psa, 2);

```

`new/usr/src/uts/common/fs/zfs/dsl_prop.c`

13

```
800         dsl_dataset_rele(ds, FTAG);  
801         return (err);  
802     }  
_____unchanged_portion_omitted_
```

```

*****
50647 Thu Jun 28 15:57:17 2012
new/usr/src/uts/common/fs/zfs/dsl_scan.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

167 /* ARGSUSED */
168 static void
169 dsl_scan_setup_sync(void *arg1, void *arg2, dmu_tx_t *tx)
170 {
171     dsl_scan_t *scn = arg1;
172     pool_scan_func_t *funcp = arg2;
173     dmu_object_type_t ot = 0;
174     dsl_pool_t *dp = scn->scn_dp;
175     spa_t *spa = dp->dp_spa;

177     ASSERT(scn->scn_phys.scn_state != DSS_SCANNING);
178     ASSERT(*funcp > POOL_SCAN_NONE && *funcp < POOL_SCAN_FUNCS);
179     bzero(&scn->scn_phys, sizeof (scn->scn_phys));
180     scn->scn_phys.scn_func = *funcp;
181     scn->scn_phys.scn_state = DSS_SCANNING;
182     scn->scn_phys.scn_min_txcg = 0;
183     scn->scn_phys.scn_max_txcg = tx->tx_txcg;
184     scn->scn_phys.scn_ddt_class_max = DDT_CLASSES - 1; /* the entire DDT */
185     scn->scn_phys.scn_start_time = gethrtime_sec();
186     scn->scn_phys.scn_errors = 0;
187     scn->scn_phys.scn_to_examine = spa->spa_root_vdev->vdev_stat.vs_alloc;
188     scn->scn_restart_txcg = 0;
189     spa_scan_stat_init(spa);

191     if (DSL_SCAN_IS_SCRUB_RESILVER(scn)) {
192         scn->scn_phys.scn_ddt_class_max = zfs_scrub_ddt_class_max;

194         /* rewrite all disk labels */
195         vdev_config_dirty(spa->spa_root_vdev);

197         if (vdev_resilver_needed(spa->spa_root_vdev,
198             &scn->scn_phys.scn_min_txcg, &scn->scn_phys.scn_max_txcg)) {
199             spa_event_notify(spa, NULL, ESC_ZFS_RESILVER_START);
200         } else {
201             spa_event_notify(spa, NULL, ESC_ZFS_SCRUB_START);
202         }

204         spa->spa_scrub_started = B_TRUE;
205         /*
206          * If this is an incremental scrub, limit the DDT scrub phase
207          * to just the auto-ditto class (for correctness); the rest
208          * of the scrub should go faster using top-down pruning.
209          */
210         if (scn->scn_phys.scn_min_txcg > TXG_INITIAL)
211             scn->scn_phys.scn_ddt_class_max = DDT_CLASS_DITTO;

213     }

215     /* back to the generic stuff */

217     if (dp->dp_blkstats == NULL) {
218         dp->dp_blkstats =

```

```

219         kmem_alloc(sizeof (zfs_all_blkstats_t), KM_SLEEP);
220     }
221     bzero(dp->dp_blkstats, sizeof (zfs_all_blkstats_t));

223     if (spa_version(spa) < SPA_VERSION_DSL_SCRUB)
224         ot = DMU_OT_ZAP_OTHER;

226     scn->scn_phys.scn_queue_obj = zap_create(dp->dp_meta_objset,
227         ot ? ot : DMU_OT_SCAN_QUEUE, DMU_OT_NONE, 0, tx);

229     dsl_scan_sync_state(scn, tx);

231     spa_history_log_internal(spa, "scan setup", tx,
231     spa_history_log_internal(LOG_POOL_SCAN, spa, tx,
232     "func=%u mintxcg=%llu maxtxcg=%llu",
233     *funcp, scn->scn_phys.scn_min_txcg, scn->scn_phys.scn_max_txcg);
234 }

236 /* ARGSUSED */
237 static void
238 dsl_scan_done(dsl_scan_t *scn, boolean_t complete, dmu_tx_t *tx)
239 {
240     static const char *old_names[] = {
241         "scrub_bookmark",
242         "scrub_ddt_bookmark",
243         "scrub_ddt_class_max",
244         "scrub_queue",
245         "scrub_min_txcg",
246         "scrub_max_txcg",
247         "scrub_func",
248         "scrub_errors",
249         NULL
250     };

252     dsl_pool_t *dp = scn->scn_dp;
253     spa_t *spa = dp->dp_spa;
254     int i;

256     /* Remove any remnants of an old-style scrub. */
257     for (i = 0; old_names[i]; i++) {
258         (void) zap_remove(dp->dp_meta_objset,
259             DMU_POOL_DIRECTORY_OBJECT, old_names[i], tx);
260     }

262     if (scn->scn_phys.scn_queue_obj != 0) {
263         VERIFY(0 == dmu_object_free(dp->dp_meta_objset,
264             scn->scn_phys.scn_queue_obj, tx));
265         scn->scn_phys.scn_queue_obj = 0;
266     }

268     /*
269      * If we were "restarted" from a stopped state, don't bother
270      * with anything else.
271      */
272     if (scn->scn_phys.scn_state != DSS_SCANNING)
273         return;

275     if (complete)
276         scn->scn_phys.scn_state = DSS_FINISHED;
277     else
278         scn->scn_phys.scn_state = DSS_CANCELED;

280     spa_history_log_internal(spa, "scan done", tx,
280     spa_history_log_internal(LOG_POOL_SCAN_DONE, spa, tx,
281     "complete=%u", complete);

```

```
283     if (DSL_SCAN_IS_SCRUB_RESILVER(scn)) {
284         mutex_enter(&spa->spa_scrub_lock);
285         while (spa->spa_scrub_inflight > 0) {
286             cv_wait(&spa->spa_scrub_io_cv,
287                 &spa->spa_scrub_lock);
288         }
289         mutex_exit(&spa->spa_scrub_lock);
290         spa->spa_scrub_started = B_FALSE;
291         spa->spa_scrub_active = B_FALSE;
292
293         /*
294          * If the scrub/resilver completed, update all DTLs to
295          * reflect this. Whether it succeeded or not, vacate
296          * all temporary scrub DTLs.
297          */
298         vdev_dtl_reassess(spa->spa_root_vdev, tx->tx_txc,
299             complete ? scn->scn_phys.scn_max_txc : 0, B_TRUE);
300         if (complete) {
301             spa_event_notify(spa, NULL, scn->scn_phys.scn_min_txc ?
302                 ESC_ZFS_RESILVER_FINISH : ESC_ZFS_SCRUB_FINISH);
303         }
304         spa_errlog_rotate(spa);
305
306         /*
307          * We may have finished replacing a device.
308          * Let the async thread assess this and handle the detach.
309          */
310         spa_async_request(spa, SPA_ASYNC_RESILVER_DONE);
311     }
312
313     scn->scn_phys.scn_end_time = gethrestime_sec();
314 }
_____unchanged_portion_omitted_____
```



```

*****
6288 Thu Jun 28 15:57:17 2012
new/usr/src/uts/common/fs/zfs/dsl_synctask.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */

27 #include <sys/dmu.h>
28 #include <sys/dmu_tx.h>
29 #include <sys/dsl_pool.h>
30 #include <sys/dsl_dir.h>
31 #include <sys/dsl_synctask.h>
32 #include <sys/metaslab.h>

34 #define DST_AVG_BLKSHIFT 14

36 /* ARGSUSED */
37 static int
38 dsl_null_checkfunc(void *arg1, void *arg2, dmu_tx_t *tx)
39 {
40     return (0);
41 }

43 dsl_sync_task_group_t *
44 dsl_sync_task_group_create(dsl_pool_t *dp)
45 {
46     dsl_sync_task_group_t *dstg;

48     dstg = kmem_zalloc(sizeof (dsl_sync_task_group_t), KM_SLEEP);
49     list_create(&dstg->dstg_tasks, sizeof (dsl_sync_task_t),
50             offsetof(dsl_sync_task_t, dst_node));
51     dstg->dstg_pool = dp;

53     return (dstg);
54 }

```

```

56 void
57 dsl_sync_task_create(dsl_sync_task_group_t *dstg,
58     dsl_checkfunc_t *checkfunc, dsl_syncfunc_t *syncfunc,
59     void *arg1, void *arg2, int blocks_modified)
60 {
61     dsl_sync_task_t *dst;

63     if (checkfunc == NULL)
64         checkfunc = dsl_null_checkfunc;
65     dst = kmem_zalloc(sizeof (dsl_sync_task_t), KM_SLEEP);
66     dst->dst_checkfunc = checkfunc;
67     dst->dst_syncfunc = syncfunc;
68     dst->dst_arg1 = arg1;
69     dst->dst_arg2 = arg2;
70     list_insert_tail(&dstg->dstg_tasks, dst);

72     dstg->dstg_space += blocks_modified << DST_AVG_BLKSHIFT;
73 }

75 int
76 dsl_sync_task_group_wait(dsl_sync_task_group_t *dstg)
77 {
78     dmu_tx_t *tx;
79     uint64_t txg;
80     dsl_sync_task_t *dst;

82 top:
83     tx = dmu_tx_create_dd(dstg->dstg_pool->dp_mos_dir);
84     VERIFY(0 == dmu_tx_assign(tx, TXG_WAIT));

86     txg = dmu_tx_get_txg(tx);

88     /* Do a preliminary error check. */
89     dstg->dstg_err = 0;
90     rw_enter(&dstg->dstg_pool->dp_config_rwlock, RW_READER);
91     for (dst = list_head(&dstg->dstg_tasks); dst;
92         dst = list_next(&dstg->dstg_tasks, dst)) {
93         /*
94          * Only check half the time, otherwise, the sync-context
95          * check will almost never fail.
96          */
97         if (spa_get_random(2) == 0)
98             goto skip;
99         continue;
100     }
101 #endif
102     /* ! codereview */
103     dst->dst_err =
104     dst->dst_checkfunc(dst->dst_arg1, dst->dst_arg2, tx);
105     if (dst->dst_err)
106         dstg->dstg_err = dst->dst_err;
107     rw_exit(&dstg->dstg_pool->dp_config_rwlock);

109     if (dstg->dstg_err) {
110         dmu_tx_commit(tx);
111         return (dstg->dstg_err);
112     }
113 skip:
114 #endif /* ! codereview */

116     /*

```

```

117     * We don't generally have many sync tasks, so pay the price of
118     * add_tail to get the tasks executed in the right order.
119     */
120     VERIFY(0 == txg_list_add_tail(&dstg->dstg_pool->dp_sync_tasks,
121     dstg, txg));
122
123     dmu_tx_commit(tx);
124
125     txg_wait_synced(dstg->dstg_pool, txg);
126
127     if (dstg->dstg_err == EAGAIN) {
128         txg_wait_synced(dstg->dstg_pool, txg + TXG_DEFER_SIZE);
129         goto top;
130     }
131
132     return (dstg->dstg_err);
133 }
134
135 void
136 dsl_sync_task_group_nowait(dsl_sync_task_group_t *dstg, dmu_tx_t *tx)
137 {
138     uint64_t txg;
139
140     dstg->dstg_nowaiter = B_TRUE;
141     txg = dmu_tx_get_txg(tx);
142     /*
143     * We don't generally have many sync tasks, so pay the price of
144     * add_tail to get the tasks executed in the right order.
145     */
146     VERIFY(0 == txg_list_add_tail(&dstg->dstg_pool->dp_sync_tasks,
147     dstg, txg));
148 }
149
150 void
151 dsl_sync_task_group_destroy(dsl_sync_task_group_t *dstg)
152 {
153     dsl_sync_task_t *dst;
154
155     while (dst = list_head(&dstg->dstg_tasks)) {
156         list_remove(&dstg->dstg_tasks, dst);
157         kmem_free(dst, sizeof (dsl_sync_task_t));
158     }
159     kmem_free(dstg, sizeof (dsl_sync_task_group_t));
160 }
161
162 void
163 dsl_sync_task_group_sync(dsl_sync_task_group_t *dstg, dmu_tx_t *tx)
164 {
165     dsl_sync_task_t *dst;
166     dsl_pool_t *dp = dstg->dstg_pool;
167     uint64_t quota, used;
168
169     ASSERT3U(dstg->dstg_err, ==, 0);
170
171     /*
172     * Check for sufficient space. We just check against what's
173     * on-disk; we don't want any in-flight accounting to get in our
174     * way, because open context may have already used up various
175     * in-core limits (arc_tempreserve, dsl_pool_tempreserve).
176     */
177     quota = dsl_pool_adjustedsize(dp, B_FALSE) -
178         metaslab_class_get_deferred(spa_normal_class(dp->dp_spa));
179     used = dp->dp_root_dir->dd_phys->dd_used_bytes;
180     /* MOS space is triple-dittoed, so we multiply by 3. */
181     if (dstg->dstg_space > 0 && used + dstg->dstg_space * 3 > quota) {
182         dstg->dstg_err = ENOSPC;

```

```

183         return;
184     }
185
186     /*
187     * Check for errors by calling checkfuncs.
188     */
189     rw_enter(&dp->dp_config_rwlock, RW_WRITER);
190     for (dst = list_head(&dstg->dstg_tasks); dst;
191         dst = list_next(&dstg->dstg_tasks, dst)) {
192         dst->dst_err =
193             dst->dst_checkfunc(dst->dst_arg1, dst->dst_arg2, tx);
194         if (dst->dst_err)
195             dstg->dstg_err = dst->dst_err;
196     }
197
198     if (dstg->dstg_err == 0) {
199         /*
200         * Execute sync tasks.
201         */
202         for (dst = list_head(&dstg->dstg_tasks); dst;
203             dst = list_next(&dstg->dstg_tasks, dst)) {
204             dst->dst_syncfunc(dst->dst_arg1, dst->dst_arg2, tx);
205         }
206     }
207     rw_exit(&dp->dp_config_rwlock);
208
209     if (dstg->dstg_nowaiter)
210         dsl_sync_task_group_destroy(dstg);
211 }
212
213 int
214 dsl_sync_task_do(dsl_pool_t *dp,
215     dsl_checkfunc_t *checkfunc, dsl_syncfunc_t *syncfunc,
216     void *arg1, void *arg2, int blocks_modified)
217 {
218     dsl_sync_task_group_t *dstg;
219     int err;
220
221     ASSERT(spa_writeable(dp->dp_spa));
222
223     dstg = dsl_sync_task_group_create(dp);
224     dsl_sync_task_create(dstg, checkfunc, syncfunc,
225         arg1, arg2, blocks_modified);
226     err = dsl_sync_task_group_wait(dstg);
227     dsl_sync_task_group_destroy(dstg);
228     return (err);
229 }
230
231 void
232 dsl_sync_task_do_nowait(dsl_pool_t *dp,
233     dsl_checkfunc_t *checkfunc, dsl_syncfunc_t *syncfunc,
234     void *arg1, void *arg2, int blocks_modified, dmu_tx_t *tx)
235 {
236     dsl_sync_task_group_t *dstg;
237
238     if (!spa_writeable(dp->dp_spa))
239         return;
240
241     dstg = dsl_sync_task_group_create(dp);
242     dsl_sync_task_create(dstg, checkfunc, syncfunc,
243         arg1, arg2, blocks_modified);
244     dsl_sync_task_group_nowait(dstg, tx);
245 }

```

```

*****
7879 Thu Jun 28 15:57:17 2012
new/usr/src/uts/common/fs/zfs/rrwlock.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 /*
26 * Copyright (c) 2012 by Delphix. All rights reserved.
27 */
28 #endif /* ! codereview */
29
30 #include <sys/refcount.h>
31 #include <sys/rrwlock.h>
32
33 /*
34 * This file contains the implementation of a re-entrant read
35 * reader/writer lock (aka "rrwlock").
36 *
37 * This is a normal reader/writer lock with the additional feature
38 * of allowing threads who have already obtained a read lock to
39 * re-enter another read lock (re-entrant read) - even if there are
40 * waiting writers.
41 *
42 * Callers who have not obtained a read lock give waiting writers priority.
43 *
44 * The rrwlock_t lock does not allow re-entrant writers, nor does it
45 * allow a re-entrant mix of reads and writes (that is, it does not
46 * allow a caller who has already obtained a read lock to be able to
47 * then grab a write lock without first dropping all read locks, and
48 * vice versa).
49 *
50 * The rrwlock_t uses tsd (thread specific data) to keep a list of
51 * nodes (rrw_node_t), where each node keeps track of which specific
52 * lock (rrw_node_t:rrn_rrl) the thread has grabbed. Since re-entering
53 * should be rare, a thread that grabs multiple reads on the same rrwlock_t
54 * will store multiple rrw_node_ts of the same 'rrn_rrl'. Nodes on the

```

```

55 * tsd list can represent a different rrwlock_t. This allows a thread
56 * to enter multiple and unique rrwlock_ts for read locks at the same time.
57 *
58 * Since using tsd exposes some overhead, the rrwlock_t only needs to
59 * keep tsd data when writers are waiting. If no writers are waiting, then
60 * a reader just bumps the anonymous read count (rr_anon_rcount) - no tsd
61 * is needed. Once a writer attempts to grab the lock, readers then
62 * keep tsd data and bump the linked readers count (rr_linked_rcount).
63 *
64 * If there are waiting writers and there are anonymous readers, then a
65 * reader doesn't know if it is a re-entrant lock. But since it may be one,
66 * we allow the read to proceed (otherwise it could deadlock). Since once
67 * waiting writers are active, readers no longer bump the anonymous count,
68 * the anonymous readers will eventually flush themselves out. At this point,
69 * readers will be able to tell if they are a re-entrant lock (have a
70 * rrw_node_t entry for the lock) or not. If they are a re-entrant lock, then
71 * we must let the proceed. If they are not, then the reader blocks for the
72 * waiting writers. Hence, we do not starve writers.
73 */
74
75 /* global key for TSD */
76 uint_t rrw_tsd_key;
77
78 typedef struct rrw_node {
79     struct rrw_node *rn_next;
80     rrwlock_t *rn_rrl;
81 } rrw_node_t;
82
83 static rrw_node_t *
84 rrn_find(rrwlock_t *r1)
85 {
86     rrw_node_t *rn;
87
88     if (refcount_count(&r1->rr_linked_rcount) == 0)
89         return (NULL);
90
91     for (rn = tsd_get(rrw_tsd_key); rn != NULL; rn = rn->rn_next) {
92         if (rn->rn_rrl == r1)
93             return (rn);
94     }
95     return (NULL);
96 }
97
98 /*
99 * Add a node to the head of the singly linked list.
100 */
101 static void
102 rrn_add(rrwlock_t *r1)
103 {
104     rrw_node_t *rn;
105
106     rn = kmem_alloc(sizeof (*rn), KM_SLEEP);
107     rn->rn_rrl = r1;
108     rn->rn_next = tsd_get(rrw_tsd_key);
109     VERIFY(tsd_set(rrw_tsd_key, rn) == 0);
110 }
111
112 /*
113 * If a node is found for 'r1', then remove the node from this
114 * thread's list and return TRUE; otherwise return FALSE.
115 */
116 static boolean_t
117 rrn_find_and_remove(rrwlock_t *r1)
118 {
119     rrw_node_t *rn;
120     rrw_node_t *prev = NULL;

```

```

122     if (refcount_count(&rrl->rr_linked_rcount) == 0)
123         return (B_FALSE);

125     for (rn = tsd_get(rrw_tsd_key); rn != NULL; rn = rn->rn_next) {
126         if (rn->rn_rrl == rrl) {
127             if (prev)
128                 prev->rn_next = rn->rn_next;
129             else
130                 VERIFY(tsd_set(rrw_tsd_key, rn->rn_next) == 0);
131             kmem_free(rn, sizeof (*rn));
132             return (B_TRUE);
133         }
134         prev = rn;
135     }
136     return (B_FALSE);
137 }

139 void
140 rrw_init(rrwlock_t *rrl)
141 {
142     mutex_init(&rrl->rr_lock, NULL, MUTEX_DEFAULT, NULL);
143     cv_init(&rrl->rr_cv, NULL, CV_DEFAULT, NULL);
144     rrl->rr_writer = NULL;
145     refcount_create(&rrl->rr_anon_rcount);
146     refcount_create(&rrl->rr_linked_rcount);
147     rrl->rr_writer_wanted = B_FALSE;
148 }

150 void
151 rrw_destroy(rrwlock_t *rrl)
152 {
153     mutex_destroy(&rrl->rr_lock);
154     cv_destroy(&rrl->rr_cv);
155     ASSERT(rrl->rr_writer == NULL);
156     refcount_destroy(&rrl->rr_anon_rcount);
157     refcount_destroy(&rrl->rr_linked_rcount);
158 }

160 static void
161 rrw_enter_read(rrwlock_t *rrl, void *tag)
162 {
163     mutex_enter(&rrl->rr_lock);
164     #if !defined(DEBUG) && defined(KERNEL)
165         if (!rrl->rr_writer && !rrl->rr_writer_wanted) {
166             rrl->rr_anon_rcount.rc_count++;
167             mutex_exit(&rrl->rr_lock);
168             return;
169         }
170     DTRACE_PROBE(zfs_rrwfastpath_rdmis);
171 #endif
172     ASSERT(rrl->rr_writer != curthread);
173     ASSERT(refcount_count(&rrl->rr_anon_rcount) >= 0);

175     while (rrl->rr_writer || (rrl->rr_writer_wanted &&
176         refcount_is_zero(&rrl->rr_anon_rcount) &&
177         rrn_find(rrl) == NULL))
178         cv_wait(&rrl->rr_cv, &rrl->rr_lock);

180     if (rrl->rr_writer_wanted) {
181         /* may or may not be a re-entrant enter */
182         rrn_add(rrl);
183         (void) refcount_add(&rrl->rr_linked_rcount, tag);
184     } else {
185         (void) refcount_add(&rrl->rr_anon_rcount, tag);
186     }

```

```

187     ASSERT(rrl->rr_writer == NULL);
188     mutex_exit(&rrl->rr_lock);
189 }

191 static void
192 rrw_enter_write(rrwlock_t *rrl)
193 {
194     mutex_enter(&rrl->rr_lock);
195     ASSERT(rrl->rr_writer != curthread);

197     while (refcount_count(&rrl->rr_anon_rcount) > 0 ||
198         refcount_count(&rrl->rr_linked_rcount) > 0 ||
199         rrl->rr_writer != NULL) {
200         rrl->rr_writer_wanted = B_TRUE;
201         cv_wait(&rrl->rr_cv, &rrl->rr_lock);
202     }
203     rrl->rr_writer_wanted = B_FALSE;
204     rrl->rr_writer = curthread;
205     mutex_exit(&rrl->rr_lock);
206 }

208 void
209 rrw_enter(rrwlock_t *rrl, krw_t rw, void *tag)
210 {
211     if (rw == RW_READER)
212         rrw_enter_read(rrl, tag);
213     else
214         rrw_enter_write(rrl);
215 }

217 void
218 rrw_exit(rrwlock_t *rrl, void *tag)
219 {
220     mutex_enter(&rrl->rr_lock);
221     #if !defined(DEBUG) && defined(KERNEL)
222         if (!rrl->rr_writer && rrl->rr_linked_rcount.rc_count == 0) {
223             rrl->rr_anon_rcount.rc_count--;
224             if (rrl->rr_anon_rcount.rc_count == 0)
225                 cv_broadcast(&rrl->rr_cv);
226             mutex_exit(&rrl->rr_lock);
227             return;
228         }
229     DTRACE_PROBE(zfs_rrwfastpath_exitmiss);
230 #endif
231     ASSERT(!refcount_is_zero(&rrl->rr_anon_rcount) ||
232         !refcount_is_zero(&rrl->rr_linked_rcount) ||
233         rrl->rr_writer != NULL);

235     if (rrl->rr_writer == NULL) {
236         int64_t count;
237         if (rrn_find_and_remove(rrl))
238             count = refcount_remove(&rrl->rr_linked_rcount, tag);
239         else
240             count = refcount_remove(&rrl->rr_anon_rcount, tag);
241         if (count == 0)
242             cv_broadcast(&rrl->rr_cv);
243     } else {
244         ASSERT(rrl->rr_writer == curthread);
245         ASSERT(refcount_is_zero(&rrl->rr_anon_rcount) &&
246             refcount_is_zero(&rrl->rr_linked_rcount));
247         rrl->rr_writer = NULL;
248         cv_broadcast(&rrl->rr_cv);
249     }
250     mutex_exit(&rrl->rr_lock);
251 }

```

```
253 boolean_t
254 rrw_held(rrwlock_t *rrl, krw_t rw)
255 {
256     boolean_t held;
257
258     mutex_enter(&rrl->rr_lock);
259     if (rw == RW_WRITER) {
260         held = (rrl->rr_writer == curthread);
261     } else {
262         held = (!refcount_is_zero(&rrl->rr_anon_rcount) ||
263             !refcount_is_zero(&rrl->rr_linked_rcount));
264     }
265     mutex_exit(&rrl->rr_lock);
266
267     return (held);
268 }
269
270 void
271 rrw_tsd_destroy(void *arg)
272 {
273     rrw_node_t *rn = arg;
274     if (rn != NULL) {
275         panic("thread %p terminating with rrw lock %p held",
276             (void *)curthread, (void *)rn->rn_rrl);
277     }
278 }
279 #endif /* ! codereview */
```

new/usr/src/uts/common/fs/zfs/spa.c

1

```
*****
169869 Thu Jun 28 15:57:18 2012
new/usr/src/uts/common/fs/zfs/spa.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

1939 /*
1940  * Load an existing storage pool, using the pool's builtin spa_config as a
1941  * source of configuration information.
1942  */
1943 static int
1944 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
1945              spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
1946              char **ereport)
1947 {
1948     int error = 0;
1949     nvlist_t *nvroot = NULL;
1950     nvlist_t *label;
1951     vdev_t *rvd;
1952     uberblock_t *ub = &spa->spa_uberblock;
1953     uint64_t children, config_cache_txg = spa->spa_config_txg;
1954     int orig_mode = spa->spa_mode;
1955     int parse;
1956     uint64_t obj;
1957     boolean_t missing_feat_write = B_FALSE;

1959     /*
1960      * If this is an untrusted config, access the pool in read-only mode.
1961      * This prevents things like resilvering recently removed devices.
1962      */
1963     if (!mosconfig)
1964         spa->spa_mode = FREAD;

1966     ASSERT(MUTEX_HELD(&spa_namespace_lock));

1968     spa->spa_load_state = state;

1970     if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
1971         return (EINVAL);

1973     parse = (type == SPA_IMPORT_EXISTING ?
1974             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

1976     /*
1977      * Create "The Godfather" zio to hold all async IOs
1978      */
1979     spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
1980                                     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

1982     /*
1983      * Parse the configuration into a vdev tree. We explicitly set the
1984      * value that will be returned by spa_version() since parsing the
1985      * configuration requires knowing the version number.
1986      */
1987     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
1988     error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
1989     spa_config_exit(spa, SCL_ALL, FTAG);
```

new/usr/src/uts/common/fs/zfs/spa.c

2

```
1991     if (error != 0)
1992         return (error);

1994     ASSERT(spa->spa_root_vdev == rvd);

1996     if (type != SPA_IMPORT_ASSEMBLE) {
1997         ASSERT(spa_guid(spa) == pool_guid);
1998     }

2000     /*
2001      * Try to open all vdevs, loading each label in the process.
2002      */
2003     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2004     error = vdev_open(rvd);
2005     spa_config_exit(spa, SCL_ALL, FTAG);
2006     if (error != 0)
2007         return (error);

2009     /*
2010      * We need to validate the vdev labels against the configuration that
2011      * we have in hand, which is dependent on the setting of mosconfig. If
2012      * mosconfig is true then we're validating the vdev labels based on
2013      * that config. Otherwise, we're validating against the cached config
2014      * (zpool.cache) that was read when we loaded the zfs module, and then
2015      * later we will recursively call spa_load() and validate against
2016      * the vdev config.
2017      *
2018      * If we're assembling a new pool that's been split off from an
2019      * existing pool, the labels haven't yet been updated so we skip
2020      * validation for now.
2021      */
2022     if (type != SPA_IMPORT_ASSEMBLE) {
2023         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2024         error = vdev_validate(rvd, mosconfig);
2025         spa_config_exit(spa, SCL_ALL, FTAG);

2027         if (error != 0)
2028             return (error);

2030         if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2031             return (ENXIO);
2032     }

2034     /*
2035      * Find the best uberblock.
2036      */
2037     vdev_uberblock_load(rvd, ub, &label);

2039     /*
2040      * If we weren't able to find a single valid uberblock, return failure.
2041      */
2042     if (ub->ub_txg == 0) {
2043         nvlist_free(label);
2044         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2045     }

2047     /*
2048      * If the pool has an unsupported version we can't open it.
2049      */
2050     if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2051         nvlist_free(label);
2052         return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2053     }

2055     if (ub->ub_version >= SPA_VERSION_FEATURES) {
2056         nvlist_t *features;
```

```

2058     /*
2059     * If we weren't able to find what's necessary for reading the
2060     * MOS in the label, return failure.
2061     */
2062     if (label == NULL || nvlist_lookup_nvlist(label,
2063         ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2064         nvlist_free(label);
2065         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2066             ENXIO));
2067     }
2069     /*
2070     * Update our in-core representation with the definitive values
2071     * from the label.
2072     */
2073     nvlist_free(spa->spa_label_features);
2074     VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2075 }
2077 nvlist_free(label);
2079 /*
2080 * Look through entries in the label nvlist's features_for_read. If
2081 * there is a feature listed there which we don't understand then we
2082 * cannot open a pool.
2083 */
2084 if (ub->ub_version >= SPA_VERSION_FEATURES) {
2085     nvlist_t *unsup_feat;
2087     VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2088         0);
2090     for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2091         NULL); nvp != NULL;
2092         nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2093         if (!zfeature_is_supported(nvpair_name(nvp))) {
2094             VERIFY(nvlist_add_string(unsup_feat,
2095                 nvpair_name(nvp), "") == 0);
2096         }
2097     }
2099     if (!nvlist_empty(unsup_feat)) {
2100         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2101             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2102         nvlist_free(unsup_feat);
2103         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2104             ENOTSUP));
2105     }
2107     nvlist_free(unsup_feat);
2108 }
2110 /*
2111 * If the vdev guid sum doesn't match the uberblock, we have an
2112 * incomplete configuration. We first check to see if the pool
2113 * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2114 * If it is, defer the vdev_guid_sum check till later so we
2115 * can handle missing vdevs.
2116 */
2117 if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2118     &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2119     rvd->vdev_guid_sum != ub->ub_guid_sum)
2120     return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));
2122 if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {

```

```

2123     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2124     spa_try_repair(spa, config);
2125     spa_config_exit(spa, SCL_ALL, FTAG);
2126     nvlist_free(spa->spa_config_splitting);
2127     spa->spa_config_splitting = NULL;
2128 }
2130 /*
2131 * Initialize internal SPA structures.
2132 */
2133 spa->spa_state = POOL_STATE_ACTIVE;
2134 spa->spa_ubsync = spa->spa_uberblock;
2135 spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2136     TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2137 spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2138     spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2139 spa->spa_claim_max_txg = spa->spa_first_txg;
2140 spa->spa_prev_software_version = ub->ub_software_version;
2142 error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2143 if (error)
2144     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2145 spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;
2147 if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2148     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2150 if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2151     boolean_t missing_feat_read = B_FALSE;
2152     nvlist_t *unsup_feat;
2154     if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2155         &spa->spa_feat_for_read_obj) != 0) {
2156         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2157     }
2159     if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2160         &spa->spa_feat_for_write_obj) != 0) {
2161         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2162     }
2164     if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2165         &spa->spa_feat_desc_obj) != 0) {
2166         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2167     }
2169     VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2170         0);
2172     if (!feature_is_supported(spa->spa_meta_objset,
2173         spa->spa_feat_for_read_obj, spa->spa_feat_desc_obj,
2174         unsup_feat))
2175         missing_feat_read = B_TRUE;
2177     if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2178         if (!feature_is_supported(spa->spa_meta_objset,
2179             spa->spa_feat_for_write_obj, spa->spa_feat_desc_obj,
2180             unsup_feat))
2181             missing_feat_write = B_TRUE;
2182     }
2184     if (!nvlist_empty(unsup_feat)) {
2185         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2186             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2187     }

```

```

2189     nvlist_free(unsup_feat);
2191     if (!missing_feat_read) {
2192         fnvlist_add_boolean(spa->spa_load_info,
2193             ZPOOL_CONFIG_CAN_RDONLY);
2194     }
2196     /*
2197     * If the state is SPA_LOAD_TRYIMPORT, our objective is
2198     * twofold: to determine whether the pool is available for
2199     * import in read-write mode and (if it is not) whether the
2200     * pool is available for import in read-only mode. If the pool
2201     * is available for import in read-write mode, it is displayed
2202     * as available in userland; if it is not available for import
2203     * in read-only mode, it is displayed as unavailable in
2204     * userland. If the pool is available for import in read-only
2205     * mode but not read-write mode, it is displayed as unavailable
2206     * in userland with a special note that the pool is actually
2207     * available for open in read-only mode.
2208     *
2209     * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2210     * missing a feature for write, we must first determine whether
2211     * the pool can be opened read-only before returning to
2212     * userland in order to know whether to display the
2213     * abovementioned note.
2214     */
2215     if (missing_feat_read || (missing_feat_write &&
2216         spa_writeable(spa))) {
2217         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2218             ENOTSUP));
2219     }
2220 }
2222 spa->spa_is_initializing = B_TRUE;
2223 error = dsl_pool_open(spa->spa_dsl_pool);
2224 spa->spa_is_initializing = B_FALSE;
2225 if (error != 0)
2226     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2228 if (!mosconfig) {
2229     uint64_t hostid;
2230     nvlist_t *policy = NULL, *nvconfig;
2232     if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2233         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2235     if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2236         ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2237         char *hostname;
2238         unsigned long myhostid = 0;
2240         VERIFY(nvlist_lookup_string(nvconfig,
2241             ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);
2243 #ifdef _KERNEL
2244         myhostid = zone_get_hostid(NULL);
2245 #else /* _KERNEL */
2246         /*
2247         * We're emulating the system's hostid in userland, so
2248         * we can't use zone_get_hostid().
2249         */
2250         (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
2251 #endif /* _KERNEL */
2252         if (hostid != 0 && myhostid != 0 &&
2253             hostid != myhostid) {
2254             nvlist_free(nvconfig);

```

```

2255         cmn_err(CE_WARN, "pool '%s' could not be "
2256             "loaded as it was last accessed by "
2257             "another system (host: %s hostid: 0x%lx). "
2258             "See: http://illumos.org/msg/ZFS-8000-EY",
2259             spa_name(spa), hostname,
2260             (unsigned long)hostid);
2261         return (EBADF);
2262     }
2263 }
2264 if (nvlist_lookup_nvlist(spa->spa_config,
2265     ZPOOL_REWIND_POLICY, &policy) == 0)
2266     VERIFY(nvlist_add_nvlist(nvconfig,
2267         ZPOOL_REWIND_POLICY, policy) == 0);
2269 spa_config_set(spa, nvconfig);
2270 spa_unload(spa);
2271 spa_deactivate(spa);
2272 spa_activate(spa, orig_mode);
2274     return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
2275 }
2277 if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2278     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2279 error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2280 if (error != 0)
2281     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2283 /*
2284 * Load the bit that tells us to use the new accounting function
2285 * (raid-z deflation). If we have an older pool, this will not
2286 * be present.
2287 */
2288 error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2289 if (error != 0 && error != ENOENT)
2290     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2292 error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2293     &spa->spa_creation_version);
2294 if (error != 0 && error != ENOENT)
2295     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2297 /*
2298 * Load the persistent error log. If we have an older pool, this will
2299 * not be present.
2300 */
2301 error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2302 if (error != 0 && error != ENOENT)
2303     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2305 error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
2306     &spa->spa_errlog_scrub);
2307 if (error != 0 && error != ENOENT)
2308     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2310 /*
2311 * Load the history object. If we have an older pool, this
2312 * will not be present.
2313 */
2314 error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2315 if (error != 0 && error != ENOENT)
2316     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2318 /*
2319 * If we're assembling the pool from the split-off vdevs of
2320 * an existing pool, we don't want to attach the spares & cache

```



```

2321     * devices.
2322     */
2324 /*
2325  * Load any hot spares for this pool.
2326  */
2327 error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spare_sav_object);
2328 if (error != 0 && error != ENOENT)
2329     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2330 if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2331     ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2332     if (load_nvlist(spa, spa->spa_spare_sav_object,
2333         &spa->spa_spare_sav_config) != 0)
2334         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2336     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2337     spa_load_spare(spa);
2338     spa_config_exit(spa, SCL_ALL, FTAG);
2339 } else if (error == 0) {
2340     spa->spa_spare_sav_sync = B_TRUE;
2341 }
2343 /*
2344  * Load any level 2 ARC devices for this pool.
2345  */
2346 error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2347     &spa->spa_l2cache_sav_object);
2348 if (error != 0 && error != ENOENT)
2349     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2350 if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2351     ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2352     if (load_nvlist(spa, spa->spa_l2cache_sav_object,
2353         &spa->spa_l2cache_sav_config) != 0)
2354         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2356     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2357     spa_load_l2cache(spa);
2358     spa_config_exit(spa, SCL_ALL, FTAG);
2359 } else if (error == 0) {
2360     spa->spa_l2cache_sav_sync = B_TRUE;
2361 }
2363 spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
2365 error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2366 if (error && error != ENOENT)
2367     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2369 if (error == 0) {
2370     uint64_t autoreplace;
2372     spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2373     spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2374     spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2375     spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2376     spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2377     spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2378         &spa->spa_dedup_ditto);
2380     spa->spa_autoreplace = (autoreplace != 0);
2381 }
2383 /*
2384  * If the 'autoreplace' property is set, then post a resource notifying
2385  * the ZFS DE that it should not issue any faults for unopenable
2386  * devices. We also iterate over the vdevs, and post a sysevent for any

```

```

2387     * unopenable vdevs so that the normal autoreplace handler can take
2388     * over.
2389     */
2390 if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2391     spa_check_removed(spa->spa_root_vdev);
2392     /*
2393      * For the import case, this is done in spa_import(), because
2394      * at this point we're using the spare definitions from
2395      * the MOS config, not necessarily from the userland config.
2396      */
2397     if (state != SPA_LOAD_IMPORT) {
2398         spa_aux_check_removed(&spa->spa_spare);
2399         spa_aux_check_removed(&spa->spa_l2cache);
2400     }
2401 }
2403 /*
2404  * Load the vdev state for all toplevel vdevs.
2405  */
2406 vdev_load(rvd);
2408 /*
2409  * Propagate the leaf DTLs we just loaded all the way up the tree.
2410  */
2411 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2412 vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2413 spa_config_exit(spa, SCL_ALL, FTAG);
2415 /*
2416  * Load the DDTs (dedup tables).
2417  */
2418 error = ddt_load(spa);
2419 if (error != 0)
2420     return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2422 spa_update_dspace(spa);
2424 /*
2425  * Validate the config, using the MOS config to fill in any
2426  * information which might be missing. If we fail to validate
2427  * the config then declare the pool unfit for use. If we're
2428  * assembling a pool from a split, the log is not transferred
2429  * over.
2430  */
2431 if (type != SPA_IMPORT_ASSEMBLE) {
2432     nvlist_t *nvconfig;
2434     if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2435         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2437     if (!spa_config_valid(spa, nvconfig)) {
2438         nvlist_free(nvconfig);
2439         return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2440             ENXIO));
2441     }
2442     nvlist_free(nvconfig);
2444     /*
2445      * Now that we've validated the config, check the state of the
2446      * root vdev. If it can't be opened, it indicates one or
2447      * more toplevel vdevs are faulted.
2448      */
2449     if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2450         return (ENXIO);
2452     if (spa_check_logs(spa)) {

```

```

2453         *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2454         return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2455     }
2456 }

2458 if (missing_feat_write) {
2459     ASSERT(state == SPA_LOAD_TRYIMPORT);

2461     /*
2462      * At this point, we know that we can open the pool in
2463      * read-only mode but not read-write mode. We now have enough
2464      * information and can return to userland.
2465      */
2466     return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2467 }

2469 /*
2470  * We've successfully opened the pool, verify that we're ready
2471  * to start pushing transactions.
2472  */
2473 if (state != SPA_LOAD_TRYIMPORT) {
2474     if (error = spa_load_verify(spa))
2475         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2476                             error));
2477 }

2479 if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2480 spa->spa_load_max_tngx == UINT64_MAX)) {
2481     dmu_tx_t *tx;
2482     int need_update = B_FALSE;

2484     ASSERT(state != SPA_LOAD_TRYIMPORT);

2486     /*
2487      * Claim log blocks that haven't been committed yet.
2488      * This must all happen in a single txg.
2489      * Note: spa_claim_max_tngx is updated by spa_claim_notify(),
2490      * invoked from zil_claim_log_block()'s i/o done callback.
2491      * Price of rollback is that we abandon the log.
2492      */
2493     spa->spa_claiming = B_TRUE;

2495     tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2496 spa_first_tngx(spa));
2497     (void) dmu_objset_find(spa_name(spa),
2498 zil_claim, tx, DS_FIND_CHILDREN);
2499     dmu_tx_commit(tx);

2501     spa->spa_claiming = B_FALSE;

2503     spa_set_log_state(spa, SPA_LOG_GOOD);
2504     spa->spa_sync_on = B_TRUE;
2505     txg_sync_start(spa->spa_dsl_pool);

2507     /*
2508      * Wait for all claims to sync. We sync up to the highest
2509      * claimed log block birth time so that claimed log blocks
2510      * don't appear to be from the future. spa_claim_max_tngx
2511      * will have been set for us by either zil_check_log_chain()
2512      * (invoked from spa_check_logs()) or zil_claim() above.
2513      */
2514     txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_tngx);

2516     /*
2517      * If the config cache is stale, or we have uninitialized
2518      * metaslabs (see spa_vdev_add()), then update the config.

```

```

2519     *
2520     * If this is a verbatim import, trust the current
2521     * in-core spa_config and update the disk labels.
2522     */
2523     if (config_cache_tngx != spa->spa_config_tngx ||
2524         state == SPA_LOAD_IMPORT ||
2525         state == SPA_LOAD_RECOVER ||
2526         (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2527         need_update = B_TRUE;

2529     for (int c = 0; c < rvd->vdev_children; c++)
2530         if (rvd->vdev_child[c]->vdev_ms_array == 0)
2531             need_update = B_TRUE;

2533     /*
2534      * Update the config cache asynchronously in case we're the
2535      * root pool, in which case the config cache isn't writable yet.
2536      */
2537     if (need_update)
2538         spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2540     /*
2541      * Check all DTLs to see if anything needs resilvering.
2542      */
2543     if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2544         vdev_resilver_needed(rvd, NULL, NULL))
2545         spa_async_request(spa, SPA_ASYNC_RESILVER);

2547     /*
2548      * Log the fact that we booted up (so that we can detect if
2549      * we rebooted in the middle of an operation).
2550      */
2551     spa_history_log_version(spa, "open");

2553     /*
2554     #endif /* ! codereview */
2555     * Delete any inconsistent datasets.
2556     */
2557     (void) dmu_objset_find(spa_name(spa),
2558 dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2560     /*
2561      * Clean up any stale temporary dataset userrefs.
2562      */
2563     dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2564 }

2566     return (0);
2567 }

2569 static int
2570 spa_load_retry(spa_t *spa, spa_load_state_t state, int mosconfig)
2571 {
2572     int mode = spa->spa_mode;

2574     spa_unload(spa);
2575     spa_deactivate(spa);

2577     spa->spa_load_max_tngx--;

2579     spa_activate(spa, mode);
2580     spa_async_suspend(spa);

2582     return (spa_load(spa, state, SPA_IMPORT_EXISTING, mosconfig));
2583 }

```

```

2585 /*
2586  * If spa_load() fails this function will try loading prior txg's. If
2587  * 'state' is SPA_LOAD_RECOVER and one of these loads succeeds the pool
2588  * will be rewound to that txg. If 'state' is not SPA_LOAD_RECOVER this
2589  * function will not rewind the pool and will return the same error as
2590  * spa_load().
2591  */
2592 static int
2593 spa_load_best(spa_t *spa, spa_load_state_t state, int mosconfig,
2594              uint64_t max_request, int rewind_flags)
2595 {
2596     nvlist_t *loadinfo = NULL;
2597     nvlist_t *config = NULL;
2598     int load_error, rewind_error;
2599     uint64_t safe_rewind_txg;
2600     uint64_t min_txg;
2601
2602     if (spa->spa_load_txg && state == SPA_LOAD_RECOVER) {
2603         spa->spa_load_max_txg = spa->spa_load_txg;
2604         spa_set_log_state(spa, SPA_LOG_CLEAR);
2605     } else {
2606         spa->spa_load_max_txg = max_request;
2607     }
2608
2609     load_error = rewind_error = spa_load(spa, state, SPA_IMPORT_EXISTING,
2610                                         mosconfig);
2611     if (load_error == 0)
2612         return (0);
2613
2614     if (spa->spa_root_vdev != NULL)
2615         config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
2616
2617     spa->spa_last_ubsync_txg = spa->spa_uberblock.ub_txg;
2618     spa->spa_last_ubsync_txg_ts = spa->spa_uberblock.ub_timestamp;
2619
2620     if (rewind_flags & ZPOOL_NEVER_REWIND) {
2621         nvlist_free(config);
2622         return (load_error);
2623     }
2624
2625     if (state == SPA_LOAD_RECOVER) {
2626         /* Price of rolling back is discarding txgs, including log */
2627         spa_set_log_state(spa, SPA_LOG_CLEAR);
2628     } else {
2629         /*
2630          * If we aren't rolling back save the load info from our first
2631          * import attempt so that we can restore it after attempting
2632          * to rewind.
2633          */
2634         loadinfo = spa->spa_load_info;
2635         spa->spa_load_info = fnvlist_alloc();
2636     }
2637
2638     spa->spa_load_max_txg = spa->spa_last_ubsync_txg;
2639     safe_rewind_txg = spa->spa_last_ubsync_txg - TXG_DEFER_SIZE;
2640     min_txg = (rewind_flags & ZPOOL_EXTREME_REWIND) ?
2641              TXG_INITIAL : safe_rewind_txg;
2642
2643     /*
2644      * Continue as long as we're finding errors, we're still within
2645      * the acceptable rewind range, and we're still finding uberblocks
2646      */
2647     while (rewind_error && spa->spa_uberblock.ub_txg >= min_txg &&
2648           spa->spa_uberblock.ub_txg <= spa->spa_load_max_txg) {
2649         if (spa->spa_load_max_txg < safe_rewind_txg)
2650             spa->spa_extreme_rewind = B_TRUE;

```

```

2651         rewind_error = spa_load_retry(spa, state, mosconfig);
2652     }
2653
2654     spa->spa_extreme_rewind = B_FALSE;
2655     spa->spa_load_max_txg = UINT64_MAX;
2656
2657     if (config && (rewind_error || state != SPA_LOAD_RECOVER))
2658         spa_config_set(spa, config);
2659
2660     if (state == SPA_LOAD_RECOVER) {
2661         ASSERT3P(loadinfo, ==, NULL);
2662         return (rewind_error);
2663     } else {
2664         /* Store the rewind info as part of the initial load info */
2665         fnvlist_add_nvlist(loadinfo, ZPOOL_CONFIG_REWIND_INFO,
2666                           spa->spa_load_info);
2667
2668         /* Restore the initial load info */
2669         fnvlist_free(spa->spa_load_info);
2670         spa->spa_load_info = loadinfo;
2671
2672         return (load_error);
2673     }
2674 }
2675
2676 /*
2677  * Pool Open/Import
2678  *
2679  * The import case is identical to an open except that the configuration is sent
2680  * down from userland, instead of grabbed from the configuration cache. For the
2681  * case of an open, the pool configuration will exist in the
2682  * POOL_STATE_UNINITIALIZED state.
2683  *
2684  * The stats information (gen/count/ustats) is used to gather vdev statistics at
2685  * the same time open the pool, without having to keep around the spa_t in some
2686  * ambiguous state.
2687  */
2688 static int
2689 spa_open_common(const char *pool, spa_t **spapp, void *tag, nvlist_t *nvpolicy,
2690                nvlist_t **config)
2691 {
2692     spa_t *spa;
2693     spa_load_state_t state = SPA_LOAD_OPEN;
2694     int error;
2695     int locked = B_FALSE;
2696
2697     *spapp = NULL;
2698
2699     /*
2700      * As disgusting as this is, we need to support recursive calls to this
2701      * function because dsl_dir_open() is called during spa_load(), and ends
2702      * up calling spa_open() again. The real fix is to figure out how to
2703      * avoid dsl_dir_open() calling this in the first place.
2704      */
2705     if (mutex_owner(&spa_namespace_lock) != curthread) {
2706         mutex_enter(&spa_namespace_lock);
2707         locked = B_TRUE;
2708     }
2709
2710     if ((spa = spa_lookup(pool)) == NULL) {
2711         if (locked)
2712             mutex_exit(&spa_namespace_lock);
2713         return (ENOENT);
2714     }
2715
2716     if (spa->spa_state == POOL_STATE_UNINITIALIZED) {

```

```

2717     zpool_rewind_policy_t policy;
2719     zpool_get_rewind_policy(nvpolicy ? nvpolicy : spa->spa_config,
2720     &policy);
2721     if (policy.zrp_request & ZPOOL_DO_REWIND)
2722         state = SPA_LOAD_RECOVER;
2724     spa_activate(spa, spa_mode_global);
2726     if (state != SPA_LOAD_RECOVER)
2727         spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;
2729     error = spa_load_best(spa, state, B_FALSE, policy.zrp_txg,
2730     policy.zrp_request);
2732     if (error == EBADF) {
2733         /*
2734          * If vdev_validate() returns failure (indicated by
2735          * EBADF), it indicates that one of the vdevs indicates
2736          * that the pool has been exported or destroyed. If
2737          * this is the case, the config cache is out of sync and
2738          * we should remove the pool from the namespace.
2739          */
2740         spa_unload(spa);
2741         spa_deactivate(spa);
2742         spa_config_sync(spa, B_TRUE, B_TRUE);
2743         spa_remove(spa);
2744         if (locked)
2745             mutex_exit(&spa_namespace_lock);
2746         return (ENOENT);
2747     }
2749     if (error) {
2750         /*
2751          * We can't open the pool, but we still have useful
2752          * information: the state of each vdev after the
2753          * attempted vdev_open(). Return this to the user.
2754          */
2755         if (config != NULL && spa->spa_config) {
2756             VERIFY(nvlist_dup(spa->spa_config, config,
2757             KM_SLEEP) == 0);
2758             VERIFY(nvlist_add_nvlist(*config,
2759             ZPOOL_CONFIG_LOAD_INFO,
2760             spa->spa_load_info) == 0);
2761         }
2762         spa_unload(spa);
2763         spa_deactivate(spa);
2764         spa->spa_last_open_failed = error;
2765         if (locked)
2766             mutex_exit(&spa_namespace_lock);
2767         *spapp = NULL;
2768         return (error);
2769     }
2770 }
2772 spa_open_ref(spa, tag);
2774 if (config != NULL)
2775     *config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
2777 /*
2778  * If we've recovered the pool, pass back any information we
2779  * gathered while doing the load.
2780  */
2781 if (state == SPA_LOAD_RECOVER) {
2782     VERIFY(nvlist_add_nvlist(*config, ZPOOL_CONFIG_LOAD_INFO,

```

```

2783         spa->spa_load_info) == 0);
2784     }
2786     if (locked) {
2787         spa->spa_last_open_failed = 0;
2788         spa->spa_last_ubsync_txg = 0;
2789         spa->spa_load_txg = 0;
2790         mutex_exit(&spa_namespace_lock);
2791     }
2793     *spapp = spa;
2795     return (0);
2796 }
2798 int
2799 spa_open_rewind(const char *name, spa_t **spapp, void *tag, nvlist_t *policy,
2800     nvlist_t **config)
2801 {
2802     return (spa_open_common(name, spapp, tag, policy, config));
2803 }
2805 int
2806 spa_open(const char *name, spa_t **spapp, void *tag)
2807 {
2808     return (spa_open_common(name, spapp, tag, NULL, NULL));
2809 }
2811 /*
2812  * Lookup the given spa_t, incrementing the inject count in the process,
2813  * preventing it from being exported or destroyed.
2814  */
2815 spa_t *
2816 spa_inject_addrf(char *name)
2817 {
2818     spa_t *spa;
2820     mutex_enter(&spa_namespace_lock);
2821     if ((spa = spa_lookup(name)) == NULL) {
2822         mutex_exit(&spa_namespace_lock);
2823         return (NULL);
2824     }
2825     spa->spa_inject_ref++;
2826     mutex_exit(&spa_namespace_lock);
2828     return (spa);
2829 }
2831 void
2832 spa_inject_delref(spa_t *spa)
2833 {
2834     mutex_enter(&spa_namespace_lock);
2835     spa->spa_inject_ref--;
2836     mutex_exit(&spa_namespace_lock);
2837 }
2839 /*
2840  * Add spares device information to the nvlist.
2841  */
2842 static void
2843 spa_add_spares(spa_t *spa, nvlist_t *config)
2844 {
2845     nvlist_t **spares;
2846     uint_t i, nspares;
2847     nvlist_t *nvroot;
2848     uint64_t guid;

```

```

2849     vdev_stat_t *vs;
2850     uint_t vsc;
2851     uint64_t pool;

2853     ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));

2855     if (spa->spa_spare.sav_count == 0)
2856         return;

2858     VERIFY(nvlist_lookup_nvlist(config,
2859         ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
2860     VERIFY(nvlist_lookup_nvlist_array(spa->spa_spare.sav_config,
2861         ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
2862     if (nspares != 0) {
2863         VERIFY(nvlist_add_nvlist_array(nvroot,
2864             ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
2865         VERIFY(nvlist_lookup_nvlist_array(nvroot,
2866             ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);

2868         /*
2869          * Go through and find any spares which have since been
2870          * repurposed as an active spare.  If this is the case, update
2871          * their status appropriately.
2872          */
2873         for (i = 0; i < nspares; i++) {
2874             VERIFY(nvlist_lookup_uint64(spares[i],
2875                 ZPOOL_CONFIG_GUID, &guid) == 0);
2876             if (spa_spare_exists(guid, &pool, NULL) &&
2877                 pool != 0ULL) {
2878                 VERIFY(nvlist_lookup_uint64_array(
2879                     spares[i], ZPOOL_CONFIG_VDEV_STATS,
2880                     (uint64_t **)&vs, &vsc) == 0);
2881                 vs->vs_state = VDEV_STATE_CANT_OPEN;
2882                 vs->vs_aux = VDEV_AUX_SPARED;
2883             }
2884         }
2885     }
2886 }

2888 /*
2889  * Add l2cache device information to the nvlist, including vdev stats.
2890  */
2891 static void
2892 spa_add_l2cache(spa_t *spa, nvlist_t *config)
2893 {
2894     nvlist_t **l2cache;
2895     uint_t i, j, nl2cache;
2896     nvlist_t *nvroot;
2897     uint64_t guid;
2898     vdev_t *vd;
2899     vdev_stat_t *vs;
2900     uint_t vsc;

2902     ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));

2904     if (spa->spa_l2cache.sav_count == 0)
2905         return;

2907     VERIFY(nvlist_lookup_nvlist(config,
2908         ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
2909     VERIFY(nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config,
2910         ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
2911     if (nl2cache != 0) {
2912         VERIFY(nvlist_add_nvlist_array(nvroot,
2913             ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
2914         VERIFY(nvlist_lookup_nvlist_array(nvroot,

```

```

2915         ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);

2917         /*
2918          * Update level 2 cache device stats.
2919          */

2921         for (i = 0; i < nl2cache; i++) {
2922             VERIFY(nvlist_lookup_uint64(l2cache[i],
2923                 ZPOOL_CONFIG_GUID, &guid) == 0);

2925             vd = NULL;
2926             for (j = 0; j < spa->spa_l2cache.sav_count; j++) {
2927                 if (guid ==
2928                     spa->spa_l2cache.sav_vdevs[j]->vdev_guid) {
2929                     vd = spa->spa_l2cache.sav_vdevs[j];
2930                     break;
2931                 }
2932             }
2933             ASSERT(vd != NULL);

2935             VERIFY(nvlist_lookup_uint64_array(l2cache[i],
2936                 ZPOOL_CONFIG_VDEV_STATS, (uint64_t **)&vs, &vsc)
2937                 == 0);
2938             vdev_get_stats(vd, vs);
2939         }
2940     }
2941 }

2943 static void
2944 spa_add_feature_stats(spa_t *spa, nvlist_t *config)
2945 {
2946     nvlist_t *features;
2947     zap_cursor_t zc;
2948     zap_attribute_t za;

2950     ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
2951     VERIFY(nvlist_alloc(&features, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2953     if (spa->spa_feat_for_read_obj != 0) {
2954         for (zap_cursor_init(&zc, spa->spa_meta_objset,
2955             spa->spa_feat_for_read_obj);
2956             zap_cursor_retrieve(&zc, &za) == 0;
2957             zap_cursor_advance(&zc)) {
2958             ASSERT(za.za_integer_length == sizeof(uint64_t) &&
2959                 za.za_num_integers == 1);
2960             VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
2961                 za.za_first_integer));
2962         }
2963         zap_cursor_fini(&zc);
2964     }

2966     if (spa->spa_feat_for_write_obj != 0) {
2967         for (zap_cursor_init(&zc, spa->spa_meta_objset,
2968             spa->spa_feat_for_write_obj);
2969             zap_cursor_retrieve(&zc, &za) == 0;
2970             zap_cursor_advance(&zc)) {
2971             ASSERT(za.za_integer_length == sizeof(uint64_t) &&
2972                 za.za_num_integers == 1);
2973             VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
2974                 za.za_first_integer));
2975         }
2976         zap_cursor_fini(&zc);
2977     }

2979     VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
2980         features) == 0);

```

```

2981     nvlist_free(features);
2982 }

2984 int
2985 spa_get_stats(const char *name, nvlist_t **config,
2986              char *altroot, size_t buflen)
2987 {
2988     int error;
2989     spa_t *spa;

2991     *config = NULL;
2992     error = spa_open_common(name, &spa, FTAG, NULL, config);

2994     if (spa != NULL) {
2995         /*
2996          * This still leaves a window of inconsistency where the spares
2997          * or l2cache devices could change and the config would be
2998          * self-inconsistent.
2999          */
3000         spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);

3002         if (*config != NULL) {
3003             uint64_t loadtimes[2];

3005             loadtimes[0] = spa->spa_loaded_ts.tv_sec;
3006             loadtimes[1] = spa->spa_loaded_ts.tv_nsec;
3007             VERIFY(nvlist_add_uint64_array(*config,
3008                ZPOOL_CONFIG_LOADED_TIME, loadtimes, 2) == 0);

3010             VERIFY(nvlist_add_uint64(*config,
3011                ZPOOL_CONFIG_ERRRCOUNT,
3012                spa_get_errlog_size(spa)) == 0);

3014             if (spa_suspended(spa))
3015                 VERIFY(nvlist_add_uint64(*config,
3016                    ZPOOL_CONFIG_SUSPENDED,
3017                    spa->spa_failmode) == 0);

3019             spa_add_spares(spa, *config);
3020             spa_add_l2cache(spa, *config);
3021             spa_add_feature_stats(spa, *config);
3022         }
3023     }

3025     /*
3026      * We want to get the alternate root even for faulted pools, so we cheat
3027      * and call spa_lookup() directly.
3028      */
3029     if (altroot) {
3030         if (spa == NULL) {
3031             mutex_enter(&spa_namespace_lock);
3032             spa = spa_lookup(name);
3033             if (spa)
3034                 spa_altroot(spa, altroot, buflen);
3035             else
3036                 altroot[0] = '\0';
3037             spa = NULL;
3038             mutex_exit(&spa_namespace_lock);
3039         } else {
3040             spa_altroot(spa, altroot, buflen);
3041         }
3042     }

3044     if (spa != NULL) {
3045         spa_config_exit(spa, SCL_CONFIG, FTAG);
3046         spa_close(spa, FTAG);

```

```

3047     }

3049     return (error);
3050 }

3052 /*
3053  * Validate that the auxiliary device array is well formed. We must have an
3054  * array of nvlists, each which describes a valid leaf vdev. If this is an
3055  * import (mode is VDEV_ALLOC_SPARE), then we allow corrupted spares to be
3056  * specified, as long as they are well-formed.
3057  */
3058 static int
3059 spa_validate_aux_devs(spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode,
3060                      spa_aux_vdev_t *sav, const char *config, uint64_t version,
3061                      vdev_labeltype_t label)
3062 {
3063     nvlist_t **dev;
3064     uint_t i, ndev;
3065     vdev_t *vd;
3066     int error;

3068     ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3070     /*
3071      * It's acceptable to have no devs specified.
3072      */
3073     if (nvlist_lookup_nvlist_array(nvroot, config, &dev, &ndev) != 0)
3074         return (0);

3076     if (ndev == 0)
3077         return (EINVAL);

3079     /*
3080      * Make sure the pool is formatted with a version that supports this
3081      * device type.
3082      */
3083     if (spa_version(spa) < version)
3084         return (ENOTSUP);

3086     /*
3087      * Set the pending device list so we correctly handle device in-use
3088      * checking.
3089      */
3090     sav->sav_pending = dev;
3091     sav->sav_npending = ndev;

3093     for (i = 0; i < ndev; i++) {
3094         if ((error = spa_config_parse(spa, &vd, dev[i], NULL, 0,
3095             mode)) != 0)
3096             goto out;

3098         if (!vd->vdev_ops->vdev_op_leaf) {
3099             vdev_free(vd);
3100             error = EINVAL;
3101             goto out;
3102         }

3104         /*
3105          * The L2ARC currently only supports disk devices in
3106          * kernel context. For user-level testing, we allow it.
3107          */
3108         #ifdef _KERNEL
3109             if ((strcmp(config, ZPOOL_CONFIG_L2CACHE) == 0) &&
3110                 strcmp(vd->vdev_ops->vdev_op_type, VDEV_TYPE_DISK) != 0) {
3111                 error = ENOTBLK;
3112                 vdev_free(vd);

```

```

3113         goto out;
3114     }
3115 #endif
3116     vd->vdev_top = vd;

3118     if ((error = vdev_open(vd)) == 0 &&
3119         (error = vdev_label_init(vd, crtngx, label)) == 0) {
3120         VERIFY(nvlist_add_uint64(dev[i], ZPOOL_CONFIG_GUID,
3121             vd->vdev_guid) == 0);
3122     }

3124     vdev_free(vd);

3126     if (error &&
3127         (mode != VDEV_ALLOC_SPARE && mode != VDEV_ALLOC_L2CACHE))
3128         goto out;
3129     else
3130         error = 0;
3131 }

3133 out:
3134     sav->sav_pending = NULL;
3135     sav->sav_npending = 0;
3136     return (error);
3137 }

3139 static int
3140 spa_validate_aux(spa_t *spa, nvlist_t *nvroot, uint64_t crtngx, int mode)
3141 {
3142     int error;

3144     ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3146     if ((error = spa_validate_aux_devs(spa, nvroot, crtngx, mode,
3147         &spa->spa_spares, ZPOOL_CONFIG_SPARES, SPA_VERSION_SPARES,
3148         VDEV_LABEL_SPARE)) != 0) {
3149         return (error);
3150     }

3152     return (spa_validate_aux_devs(spa, nvroot, crtngx, mode,
3153         &spa->spa_l2cache, ZPOOL_CONFIG_L2CACHE, SPA_VERSION_L2CACHE,
3154         VDEV_LABEL_L2CACHE));
3155 }

3157 static void
3158 spa_set_aux_vdevs(spa_aux_vdev_t *sav, nvlist_t **devs, int ndevs,
3159     const char *config)
3160 {
3161     int i;

3163     if (sav->sav_config != NULL) {
3164         nvlist_t **olddevs;
3165         uint_t oldndevs;
3166         nvlist_t **newdevs;

3168         /*
3169          * Generate new dev list by concatenating with the
3170          * current dev list.
3171          */
3172         VERIFY(nvlist_lookup_nvlist_array(sav->sav_config, config,
3173             &olddevs, &oldndevs) == 0);

3175         newdevs = kmem_alloc(sizeof (void *) *
3176             (ndevs + oldndevs), KM_SLEEP);
3177         for (i = 0; i < oldndevs; i++)
3178             VERIFY(nvlist_dup(olddevs[i], &newdevs[i],

```

```

3179             KM_SLEEP) == 0);
3180         for (i = 0; i < ndevs; i++)
3181             VERIFY(nvlist_dup(devs[i], &newdevs[i + oldndevs],
3182                 KM_SLEEP) == 0);

3184         VERIFY(nvlist_remove(sav->sav_config, config,
3185             DATA_TYPE_NVLIST_ARRAY) == 0);

3187         VERIFY(nvlist_add_nvlist_array(sav->sav_config,
3188             config, newdevs, ndevs + oldndevs) == 0);
3189         for (i = 0; i < oldndevs + ndevs; i++)
3190             nvlist_free(newdevs[i]);
3191         kmem_free(newdevs, (oldndevs + ndevs) * sizeof (void *));
3192     } else {
3193         /*
3194          * Generate a new dev list.
3195          */
3196         VERIFY(nvlist_alloc(&sav->sav_config, NV_UNIQUE_NAME,
3197             KM_SLEEP) == 0);
3198         VERIFY(nvlist_add_nvlist_array(sav->sav_config, config,
3199             devs, ndevs) == 0);
3200     }
3201 }

3203 /*
3204  * Stop and drop level 2 ARC devices
3205  */
3206 void
3207 spa_l2cache_drop(spa_t *spa)
3208 {
3209     vdev_t *vd;
3210     int i;
3211     spa_aux_vdev_t *sav = &spa->spa_l2cache;

3213     for (i = 0; i < sav->sav_count; i++) {
3214         uint64_t pool;

3216         vd = sav->sav_vdevs[i];
3217         ASSERT(vd != NULL);

3219         if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
3220             pool != 0ULL && l2arc_vdev_present(vd))
3221             l2arc_remove_vdev(vd);
3222     }
3223 }

3225 /*
3226  * Pool Creation
3227  */
3228 int
3229 spa_create(const char *pool, nvlist_t *nvroot, nvlist_t *props,
3230     nvlist_t *zplprops)
3231 {
3232     spa_t *spa;
3233     char *altroot = NULL;
3234     vdev_t *rvd;
3235     dsl_pool_t *dp;
3236     dm_u_tx_t *tx;
3237     int error = 0;
3238     uint64_t txg = TXG_INITIAL;
3239     nvlist_t **spares, **l2cache;
3240     uint_t nspares, nl2cache;
3241     uint64_t version, obj;
3242     boolean_t has_features;

```

```

3244 /*
3245  * If this pool already exists, return failure.
3246  */
3247 mutex_enter(&spa_namespace_lock);
3248 if (spa_lookup(pool) != NULL) {
3249     mutex_exit(&spa_namespace_lock);
3250     return (EEXIST);
3251 }
3252
3253 /*
3254  * Allocate a new spa_t structure.
3255  */
3256 (void) nvlist_lookup_string(props,
3257     zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3258 spa = spa_add(pool, NULL, altroot);
3259 spa_activate(spa, spa_mode_global);
3260
3261 if (props && (error = spa_prop_validate(spa, props))) {
3262     spa_deactivate(spa);
3263     spa_remove(spa);
3264     mutex_exit(&spa_namespace_lock);
3265     return (error);
3266 }
3267
3268 has_features = B_FALSE;
3269 for (nvpair_t *elem = nvlist_next_nvpair(props, NULL);
3270     elem != NULL; elem = nvlist_next_nvpair(props, elem)) {
3271     if (zpool_prop_feature(nvpair_name(elem)))
3272         has_features = B_TRUE;
3273 }
3274
3275 if (has_features || nvlist_lookup_uint64(props,
3276     zpool_prop_to_name(ZPOOL_PROP_VERSION), &version) != 0) {
3277     version = SPA_VERSION;
3278 }
3279 ASSERT(SPA_VERSION_IS_SUPPORTED(version));
3280
3281 spa->spa_first_txg = txg;
3282 spa->spa_uberblock.ub_txg = txg - 1;
3283 spa->spa_uberblock.ub_version = version;
3284 spa->spa_ubsync = spa->spa_uberblock;
3285
3286 /*
3287  * Create "The Godfather" zio to hold all async IOs
3288  */
3289 spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
3290     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);
3291
3292 /*
3293  * Create the root vdev.
3294  */
3295 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3296
3297 error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, VDEV_ALLOC_ADD);
3298
3299 ASSERT(error != 0 || rvd != NULL);
3300 ASSERT(error != 0 || spa->spa_root_vdev == rvd);
3301
3302 if (error == 0 && !zfs_allocatable_devs(nvroot))
3303     error = EINVAL;
3304
3305 if (error == 0 &&
3306     (error = vdev_create(rvd, txg, B_FALSE)) == 0 &&
3307     (error = spa_validate_aux(spa, nvroot, txg,
3308         VDEV_ALLOC_ADD)) == 0) {
3309     for (int c = 0; c < rvd->vdev_children; c++) {

```

```

3310         vdev metaslab_set_size(rvd->vdev_child[c]);
3311         vdev_expand(rvd->vdev_child[c], txg);
3312     }
3313 }
3314
3315 spa_config_exit(spa, SCL_ALL, FTAG);
3316
3317 if (error != 0) {
3318     spa_unload(spa);
3319     spa_deactivate(spa);
3320     spa_remove(spa);
3321     mutex_exit(&spa_namespace_lock);
3322     return (error);
3323 }
3324
3325 /*
3326  * Get the list of spares, if specified.
3327  */
3328 if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3329     &spares, &nspares) == 0) {
3330     VERIFY(nvlist_alloc(&spa->spa_spares.sav_config, NV_UNIQUE_NAME,
3331         KM_SLEEP) == 0);
3332     VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3333         ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3334     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3335     spa_load_spares(spa);
3336     spa_config_exit(spa, SCL_ALL, FTAG);
3337     spa->spa_spares.sav_sync = B_TRUE;
3338 }
3339
3340 /*
3341  * Get the list of level 2 cache devices, if specified.
3342  */
3343 if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3344     &l2cache, &nl2cache) == 0) {
3345     VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3346         NV_UNIQUE_NAME, KM_SLEEP) == 0);
3347     VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3348         ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3349     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3350     spa_load_l2cache(spa);
3351     spa_config_exit(spa, SCL_ALL, FTAG);
3352     spa->spa_l2cache.sav_sync = B_TRUE;
3353 }
3354
3355 spa->spa_is_initializing = B_TRUE;
3356 spa->spa_dsl_pool = dp = dsl_pool_create(spa, zplprops, txg);
3357 spa->spa_meta_objset = dp->dp_meta_objset;
3358 spa->spa_is_initializing = B_FALSE;
3359
3360 /*
3361  * Create DDTs (dedup tables).
3362  */
3363 ddt_create(spa);
3364
3365 spa_update_dspace(spa);
3366
3367 tx = dmu_tx_create_assigned(dp, txg);
3368
3369 /*
3370  * Create the pool config object.
3371  */
3372 spa->spa_config_object = dmu_object_alloc(spa->spa_meta_objset,
3373     DMU_OT_PACKED_NVLIST, SPA_CONFIG_BLOCKSIZE,
3374     DMU_OT_PACKED_NVLIST_SIZE, sizeof (uint64_t), tx);

```



```

3376     if (zap_add(spa->spa_meta_objset,
3377         DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CONFIG,
3378         sizeof(uint64_t), 1, &spa->spa_config_object, tx) != 0) {
3379         cmn_err(CE_PANIC, "failed to add pool config");
3380     }

3382     if (spa_version(spa) >= SPA_VERSION_FEATURES)
3383         spa_feature_create_zap_objects(spa, tx);

3385     if (zap_add(spa->spa_meta_objset,
3386         DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CREATION_VERSION,
3387         sizeof(uint64_t), 1, &version, tx) != 0) {
3388         cmn_err(CE_PANIC, "failed to add pool version");
3389     }

3391     /* Newly created pools with the right version are always deflated. */
3392     if (version >= SPA_VERSION_RAIDZ_DEFLATE) {
3393         spa->spa_deflate = TRUE;
3394         if (zap_add(spa->spa_meta_objset,
3395             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
3396             sizeof(uint64_t), 1, &spa->spa_deflate, tx) != 0) {
3397             cmn_err(CE_PANIC, "failed to add deflate");
3398         }
3399     }

3401     /*
3402     * Create the deferred-free bpobj. Turn off compression
3403     * because sync-to-convergence takes longer if the blocksize
3404     * keeps changing.
3405     */
3406     obj = bpobj_alloc(spa->spa_meta_objset, 1 << 14, tx);
3407     dmu_object_set_compress(spa->spa_meta_objset, obj,
3408         ZIO_COMPRESS_OFF, tx);
3409     if (zap_add(spa->spa_meta_objset,
3410         DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_SYNC_BPOBJ,
3411         sizeof(uint64_t), 1, &obj, tx) != 0) {
3412         cmn_err(CE_PANIC, "failed to add bpobj");
3413     }
3414     VERIFY3U(0, ==, bpobj_open(&spa->spa_deferred_bpobj,
3415         spa->spa_meta_objset, obj));

3417     /*
3418     * Create the pool's history object.
3419     */
3420     if (version >= SPA_VERSION_ZPOOL_HISTORY)
3421         spa_history_create_obj(spa, tx);

3423     /*
3424     * Set pool properties.
3425     */
3426     spa->spa_bootfs = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
3427     spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
3428     spa->spa_failmode = zpool_prop_default_numeric(ZPOOL_PROP_FAILUREMODE);
3429     spa->spa_autoexpand = zpool_prop_default_numeric(ZPOOL_PROP_AUTOEXPAND);

3431     if (props != NULL) {
3432         spa_configfile_set(spa, props, B_FALSE);
3433         spa_sync_props(spa, props, tx);
3434     }

3436     dmu_tx_commit(tx);

3438     spa->spa_sync_on = B_TRUE;
3439     txg_sync_start(spa->spa_dsl_pool);

3441     /*

```

```

3442     * We explicitly wait for the first transaction to complete so that our
3443     * bean counters are appropriately updated.
3444     */
3445     txg_wait_synced(spa->spa_dsl_pool, txg);

3447     spa_config_sync(spa, B_FALSE, B_TRUE);

3449     spa_history_log_version(spa, "create");
3450     if (version >= SPA_VERSION_ZPOOL_HISTORY && history_str != NULL)
3451         (void) spa_history_log(spa, history_str, LOG_CMD_POOL_CREATE);
3452     spa_history_log_version(spa, LOG_POOL_CREATE);

3451     spa->spa_minref = refcount_count(&spa->spa_refcount);

3453     mutex_exit(&spa_namespace_lock);

3455     return (0);
3456 }

unchanged_portion_omitted

3537 /*
3538 * Import a root pool.
3539 *
3540 * For x86, devpath_list will consist of devid and/or physpath name of
3541 * the vdev (e.g. "idl,sd@SSEAGATE..." or "/pci@1f,0/ide@disk@0,0:a").
3542 * The GRUB "findroot" command will return the vdev we should boot.
3543 *
3544 * For Sparc, devpath_list consists the physpath name of the booting device
3545 * no matter the rootpool is a single device pool or a mirrored pool.
3546 * e.g.
3547 *     "/pci@1f,0/ide@disk@0,0:a"
3548 */
3549 int
3550 spa_import_rootpool(char *devpath, char *devid)
3551 {
3552     spa_t *spa;
3553     vdev_t *rvd, *bvd, *avd = NULL;
3554     nvlist_t *config, *nvtop;
3555     uint64_t guid, txg;
3556     char *pname;
3557     int error;

3559     /*
3560     * Read the label from the boot device and generate a configuration.
3561     */
3562     config = spa_generate_rootconf(devpath, devid, &guid);
3563     #if defined(_OBP) && defined(_KERNEL)
3564     if (config == NULL) {
3565         if (strstr(devpath, "/iscsi/ssd") != NULL) {
3566             /* iscsi boot */
3567             get_iscsi_bootpath_phy(devpath);
3568             config = spa_generate_rootconf(devpath, devid, &guid);
3569         }
3570     }
3571     #endif
3572     if (config == NULL) {
3573         cmn_err(CE_NOTE, "Cannot read the pool label from '%s'",
3574             devpath);
3575         return (EIO);
3576     }

3578     VERIFY(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME,
3579         &pname) == 0);
3580     VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG, &txg) == 0);

3582     mutex_enter(&spa_namespace_lock);

```

```

3583     if ((spa = spa_lookup(pname)) != NULL) {
3584         /*
3585          * Remove the existing root pool from the namespace so that we
3586          * can replace it with the correct config we just read in.
3587          */
3588         spa_remove(spa);
3589     }

3591     spa = spa_add(pname, config, NULL);
3592     spa->spa_is_root = B_TRUE;
3593     spa->spa_import_flags = ZFS_IMPORT_VERBATIM;

3595     /*
3596      * Build up a vdev tree based on the boot device's label config.
3597      */
3598     VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3599         &nvtop) == 0);
3600     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3601     error = spa_config_parse(spa, &rvd, nvtop, NULL, 0,
3602         VDEV_ALLOC_ROOTPOOL);
3603     spa_config_exit(spa, SCL_ALL, FTAG);
3604     if (error) {
3605         mutex_exit(&spa_namespace_lock);
3606         nvlist_free(config);
3607         cmn_err(CE_NOTE, "Can not parse the config for pool '%s'",
3608             pname);
3609         return (error);
3610     }

3612     /*
3613      * Get the boot vdev.
3614      */
3615     if ((bvd = vdev_lookup_by_guid(rvd, guid)) == NULL) {
3616         cmn_err(CE_NOTE, "Can not find the boot vdev for guid %llu",
3617             (u_longlong_t)guid);
3618         error = ENOENT;
3619         goto out;
3620     }

3622     /*
3623      * Determine if there is a better boot device.
3624      */
3625     avd = bvd;
3626     spa_alt_rootvdev(rvd, &avd, &txg);
3627     if (avd != bvd) {
3628         cmn_err(CE_NOTE, "The boot device is 'degraded'. Please "
3629             "try booting from '%s'", avd->vdev_path);
3630         error = EINVAL;
3631         goto out;
3632     }

3634     /*
3635      * If the boot device is part of a spare vdev then ensure that
3636      * we're booting off the active spare.
3637      */
3638     if (bvd->vdev_parent->vdev_ops == &vdev_spare_ops &&
3639         !bvd->vdev_isspare) {
3640         cmn_err(CE_NOTE, "The boot device is currently spared. Please "
3641             "try booting from '%s'",
3642             bvd->vdev_parent->
3643             vdev_child[bvd->vdev_parent->vdev_children - 1]->vdev_path);
3644         error = EINVAL;
3645         goto out;
3646     }

3648     error = 0;

```

```

2969     spa_history_log_version(spa, LOG_POOL_IMPORT);
3649 out:
3650     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3651     vdev_free(rvd);
3652     spa_config_exit(spa, SCL_ALL, FTAG);
3653     mutex_exit(&spa_namespace_lock);

3655     nvlist_free(config);
3656     return (error);
3657 }

3659 #endif

3661 /*
3662  * Import a non-root pool into the system.
3663  */
3664 int
3665 spa_import(const char *pool, nvlist_t *config, nvlist_t *props, uint64_t flags)
3666 {
3667     spa_t *spa;
3668     char *altroot = NULL;
3669     spa_load_state_t state = SPA_LOAD_IMPORT;
3670     zpool_rewind_policy_t policy;
3671     uint64_t mode = spa_mode_global;
3672     uint64_t readonly = B_FALSE;
3673     int error;
3674     nvlist_t *nvroot;
3675     nvlist_t **spares, **l2cache;
3676     uint_t nspares, nl2cache;

3678     /*
3679      * If a pool with this name exists, return failure.
3680      */
3681     mutex_enter(&spa_namespace_lock);
3682     if (spa_lookup(pool) != NULL) {
3683         mutex_exit(&spa_namespace_lock);
3684         return (EEXIST);
3685     }

3687     /*
3688      * Create and initialize the spa structure.
3689      */
3690     (void) nvlist_lookup_string(props,
3691         zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3692     (void) nvlist_lookup_uint64(props,
3693         zpool_prop_to_name(ZPOOL_PROP_READONLY), &readonly);
3694     if (readonly)
3695         mode = FREAD;
3696     spa = spa_add(pool, config, altroot);
3697     spa->spa_import_flags = flags;

3699     /*
3700      * Verbatim import - Take a pool and insert it into the namespace
3701      * as if it had been loaded at boot.
3702      */
3703     if (spa->spa_import_flags & ZFS_IMPORT_VERBATIM) {
3704         if (props != NULL)
3705             spa_configfile_set(spa, props, B_FALSE);

3707         spa_config_sync(spa, B_FALSE, B_TRUE);

3709         mutex_exit(&spa_namespace_lock);
3710         spa_history_log_version(spa, "import");
3711         spa_history_log_version(spa, LOG_POOL_IMPORT);

3712         return (0);

```

```

3713     }
3715     spa_activate(spa, mode);
3717     /*
3718     * Don't start async tasks until we know everything is healthy.
3719     */
3720     spa_async_suspend(spa);
3722     zpool_get_rewind_policy(config, &policy);
3723     if (policy.zrp_request & ZPOOL_DO_REWIND)
3724         state = SPA_LOAD_RECOVER;
3726     /*
3727     * Pass off the heavy lifting to spa_load(). Pass TRUE for mosconfig
3728     * because the user-supplied config is actually the one to trust when
3729     * doing an import.
3730     */
3731     if (state != SPA_LOAD_RECOVER)
3732         spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;
3734     error = spa_load_best(spa, state, B_TRUE, policy.zrp_txg,
3735                          policy.zrp_request);
3737     /*
3738     * Propagate anything learned while loading the pool and pass it
3739     * back to caller (i.e. rewind info, missing devices, etc).
3740     */
3741     VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
3742                             spa->spa_load_info) == 0);
3744     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3745     /*
3746     * Toss any existing sparelist, as it doesn't have any validity
3747     * anymore, and conflicts with spa_has_spare().
3748     */
3749     if (spa->spa_spare.sav_config) {
3750         nvlist_free(spa->spa_spare.sav_config);
3751         spa->spa_spare.sav_config = NULL;
3752     }
3753     if (spa->spa_l2cache.sav_config) {
3754         nvlist_free(spa->spa_l2cache.sav_config);
3755         spa->spa_l2cache.sav_config = NULL;
3756     }
3757     spa_load_l2cache(spa);
3758
3760     VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3761                                &nvroot) == 0);
3762     if (error == 0)
3763         error = spa_validate_aux(spa, nvroot, -1ULL,
3764                                VDEV_ALLOC_SPARE);
3765     if (error == 0)
3766         error = spa_validate_aux(spa, nvroot, -1ULL,
3767                                VDEV_ALLOC_L2CACHE);
3768     spa_config_exit(spa, SCL_ALL, FTAG);
3770     if (props != NULL)
3771         spa_configfile_set(spa, props, B_FALSE);
3773     if (error != 0 || (props && spa_writeable(spa) &&
3774                      (error = spa_prop_set(spa, props)))) {
3775         spa_unload(spa);
3776         spa_deactivate(spa);
3777         spa_remove(spa);
3778         mutex_exit(&spa_namespace_lock);

```

```

3779         return (error);
3780     }
3782     spa_async_resume(spa);
3784     /*
3785     * Override any spares and level 2 cache devices as specified by
3786     * the user, as these may have correct device names/devids, etc.
3787     */
3788     if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3789                                    &spares, &nspares) == 0) {
3790         if (spa->spa_spare.sav_config)
3791             VERIFY(nvlist_remove(spa->spa_spare.sav_config,
3792                                 ZPOOL_CONFIG_SPARES, DATA_TYPE_NVLIST_ARRAY) == 0);
3793         else
3794             VERIFY(nvlist_alloc(&spa->spa_spare.sav_config,
3795                                NV_UNIQUE_NAME, KM_SLEEP) == 0);
3796         VERIFY(nvlist_add_nvlist_array(spa->spa_spare.sav_config,
3797                                       ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3798         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3799         spa_load_spare(spa);
3800         spa_config_exit(spa, SCL_ALL, FTAG);
3801         spa->spa_spare.sav_sync = B_TRUE;
3802     }
3803     if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3804                                    &l2cache, &nl2cache) == 0) {
3805         if (spa->spa_l2cache.sav_config)
3806             VERIFY(nvlist_remove(spa->spa_l2cache.sav_config,
3807                                 ZPOOL_CONFIG_L2CACHE, DATA_TYPE_NVLIST_ARRAY) == 0);
3808         else
3809             VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3810                                NV_UNIQUE_NAME, KM_SLEEP) == 0);
3811         VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3812                                       ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3813         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3814         spa_load_l2cache(spa);
3815         spa_config_exit(spa, SCL_ALL, FTAG);
3816         spa->spa_l2cache.sav_sync = B_TRUE;
3817     }
3819     /*
3820     * Check for any removed devices.
3821     */
3822     if (spa->spa_autoreplace) {
3823         spa_aux_check_removed(&spa->spa_spare);
3824         spa_aux_check_removed(&spa->spa_l2cache);
3825     }
3827     if (spa_writeable(spa)) {
3828         /*
3829         * Update the config cache to include the newly-imported pool.
3830         */
3831         spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
3832     }
3834     /*
3835     * It's possible that the pool was expanded while it was exported.
3836     * We kick off an async task to handle this for us.
3837     */
3838     spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);
3840     mutex_exit(&spa_namespace_lock);
3841     spa_history_log_version(spa, "import");
3842     spa_history_log_version(spa, LOG_POOL_IMPORT);
3844     return (0);

```

```

3844 }
      unchanged_portion_omitted
4187 /*
4188 * Attach a device to a mirror. The arguments are the path to any device
4189 * in the mirror, and the nvroot for the new device. If the path specifies
4190 * a device that is not mirrored, we automatically insert the mirror vdev.
4191 *
4192 * If 'replacing' is specified, the new device is intended to replace the
4193 * existing device; in this case the two devices are made into their own
4194 * mirror using the 'replacing' vdev, which is functionally identical to
4195 * the mirror vdev (it actually reuses all the same ops) but has a few
4196 * extra rules: you can't attach to it after it's been created, and upon
4197 * completion of resilvering, the first disk (the one being replaced)
4198 * is automatically detached.
4199 */
4200 int
4201 spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot, int replacing)
4202 {
4203     uint64_t txg, dtl_max_txg;
4204     vdev_t *rvd = spa->spa_root_vdev;
4205     vdev_t *oldvd, *newvd, *newrootvd, *pvd, *tvd;
4206     vdev_ops_t *pvops;
4207     char *oldvdpath, *newvdpath;
4208     int newvd_isspare;
4209     int error;
4210
4211     ASSERT(spa_writeable(spa));
4212
4213     txg = spa_vdev_enter(spa);
4214
4215     oldvd = spa_lookup_by_guid(spa, guid, B_FALSE);
4216
4217     if (oldvd == NULL)
4218         return (spa_vdev_exit(spa, NULL, txg, ENODEV));
4219
4220     if (!oldvd->vdev_ops->vdev_op_leaf)
4221         return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));
4222
4223     pvd = oldvd->vdev_parent;
4224
4225     if ((error = spa_config_parse(spa, &newrootvd, nvroot, NULL, 0,
4226     VDEV_ALLOC_ATTACH)) != 0)
4227         return (spa_vdev_exit(spa, NULL, txg, EINVAL));
4228
4229     if (newrootvd->vdev_children != 1)
4230         return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));
4231
4232     newvd = newrootvd->vdev_child[0];
4233
4234     if (!newvd->vdev_ops->vdev_op_leaf)
4235         return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));
4236
4237     if ((error = vdev_create(newrootvd, txg, replacing)) != 0)
4238         return (spa_vdev_exit(spa, newrootvd, txg, error));
4239
4240     /*
4241     * Spares can't replace logs
4242     */
4243     if (oldvd->vdev_top->vdev_islog && newvd->vdev_isspare)
4244         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4245
4246     if (!replacing) {
4247         /*
4248         * For attach, the only allowable parent is a mirror or the root
4249         * vdev.

```

```

4250     */
4251     if (pvd->vdev_ops != &vdev_mirror_ops &&
4252     pvd->vdev_ops != &vdev_root_ops)
4253         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4254
4255     pvops = &vdev_mirror_ops;
4256 } else {
4257     /*
4258     * Active hot spares can only be replaced by inactive hot
4259     * spares.
4260     */
4261     if (pvd->vdev_ops == &vdev_spare_ops &&
4262     oldvd->vdev_isspare &&
4263     !spa_has_spare(spa, newvd->vdev_guid))
4264         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4265
4266     /*
4267     * If the source is a hot spare, and the parent isn't already a
4268     * spare, then we want to create a new hot spare. Otherwise, we
4269     * want to create a replacing vdev. The user is not allowed to
4270     * attach to a spared vdev child unless the 'isspare' state is
4271     * the same (spare replaces spare, non-spare replaces
4272     * non-spare).
4273     */
4274     if (pvd->vdev_ops == &vdev_replacing_ops &&
4275     spa_version(spa) < SPA_VERSION_MULTI_REPLACE) {
4276         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4277     } else if (pvd->vdev_ops == &vdev_spare_ops &&
4278     newvd->vdev_isspare != oldvd->vdev_isspare) {
4279         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4280     }
4281
4282     if (newvd->vdev_isspare)
4283         pvops = &vdev_spare_ops;
4284     else
4285         pvops = &vdev_replacing_ops;
4286 }
4287
4288     /*
4289     * Make sure the new device is big enough.
4290     */
4291     if (newvd->vdev_asize < vdev_get_min_asize(oldvd))
4292         return (spa_vdev_exit(spa, newrootvd, txg, EOVERFLOW));
4293
4294     /*
4295     * The new device cannot have a higher alignment requirement
4296     * than the top-level vdev.
4297     */
4298     if (newvd->vdev_ashift > oldvd->vdev_top->vdev_ashift)
4299         return (spa_vdev_exit(spa, newrootvd, txg, EDOM));
4300
4301     /*
4302     * If this is an in-place replacement, update oldvd's path and devid
4303     * to make it distinguishable from newvd, and unopenable from now on.
4304     */
4305     if (strcmp(oldvd->vdev_path, newvd->vdev_path) == 0) {
4306         spa_strfree(oldvd->vdev_path);
4307         oldvd->vdev_path = kmem_alloc(strlen(newvd->vdev_path) + 5,
4308         KM_SLEEP);
4309         (void) sprintf(oldvd->vdev_path, "%s%s",
4310         newvd->vdev_path, "old");
4311         if (oldvd->vdev_devid != NULL) {
4312             spa_strfree(oldvd->vdev_devid);
4313             oldvd->vdev_devid = NULL;
4314         }
4315     }

```

```

4317      /* mark the device being resilvered */
4318      newvd->vdev_resilvering = B_TRUE;

4320      /*
4321       * If the parent is not a mirror, or if we're replacing, insert the new
4322       * mirror/replacing/spare vdev above oldvd.
4323       */
4324      if (pvd->vdev_ops != pvops)
4325          pvd = vdev_add_parent(oldvd, pvops);

4327      ASSERT(pvd->vdev_top->vdev_parent == rvd);
4328      ASSERT(pvd->vdev_ops == pvops);
4329      ASSERT(oldvd->vdev_parent == pvd);

4331      /*
4332       * Extract the new device from its root and add it to pvd.
4333       */
4334      vdev_remove_child(newrootvd, newvd);
4335      newvd->vdev_id = pvd->vdev_children;
4336      newvd->vdev_crtxg = oldvd->vdev_crtxg;
4337      vdev_add_child(pvd, newvd);

4339      tvd = newvd->vdev_top;
4340      ASSERT(pvd->vdev_top == tvd);
4341      ASSERT(tvd->vdev_parent == rvd);

4343      vdev_config_dirty(tvd);

4345      /*
4346       * Set newvd's DTL to [TXG_INITIAL, dtl_max_txg] so that we account
4347       * for any dmu_sync-ed blocks. It will propagate upward when
4348       * spa_vdev_exit() calls vdev_dtl_reassess().
4349       */
4350      dtl_max_txg = txg + TXG_CONCURRENT_STATES;

4352      vdev_dtl_dirty(newvd, DTL_MISSING, TXG_INITIAL,
4353                    dtl_max_txg - TXG_INITIAL);

4355      if (newvd->vdev_isspare) {
4356          spa_spare_activate(newvd);
4357          spa_event_notify(spa, newvd, ESC_ZFS_VDEV_SPARE);
4358      }

4360      oldvdpath = spa_strdup(oldvd->vdev_path);
4361      newvdpath = spa_strdup(newvd->vdev_path);
4362      newvd_isspare = newvd->vdev_isspare;

4364      /*
4365       * Mark newvd's DTL dirty in this txg.
4366       */
4367      vdev_dirty(tvd, VDD_DTL, newvd, txg);

4369      /*
4370       * Restart the resilver
4371       */
4372      dsl_resilver_restart(spa->spa_dsl_pool, dtl_max_txg);

4374      /*
4375       * Commit the config
4376       */
4377      (void) spa_vdev_exit(spa, newrootvd, dtl_max_txg, 0);

4379      spa_history_log_internal(spa, "vdev attach", NULL,
3700      spa_history_log_internal(LOG_POOL_VDEV_ATTACH, spa, NULL,
4380      "%s vdev=%s %s vdev=%s",

```

```

4381          replacing && newvd_isspare ? "spare in" :
4382          replacing ? "replace" : "attach", newvdpath,
4383          replacing ? "for" : "to", oldvdpath);

4385          spa_strfree(oldvdpath);
4386          spa_strfree(newvdpath);

4388          if (spa->spa_bootfs)
4389              spa_event_notify(spa, newvd, ESC_ZFS_BOOTFS_VDEV_ATTACH);

4391          return (0);
4392      }

4394      /*
4395       * Detach a device from a mirror or replacing vdev.
4396       * If 'replace_done' is specified, only detach if the parent
4397       * is a replacing vdev.
4398       */
4399      int
4400      spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid, int replace_done)
4401      {
4402          uint64_t txg;
4403          int error;
4404          vdev_t *rvd = spa->spa_root_vdev;
4405          vdev_t *vd, *pvd, *cvd, *tvd;
4406          boolean_t unspare = B_FALSE;
4407          uint64_t unspare_guid;
4408          char *vdpath;

4410          ASSERT(spa_writable(spa));

4412          txg = spa_vdev_enter(spa);

4414          vd = spa_lookup_by_guid(spa, guid, B_FALSE);

4416          if (vd == NULL)
4417              return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4419          if (!vd->vdev_ops->vdev_op_leaf)
4420              return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4422          pvd = vd->vdev_parent;

4424          /*
4425           * If the parent/child relationship is not as expected, don't do it.
4426           * Consider M(A,R(B,C)) -- that is, a mirror of A with a replacing
4427           * vdev that's replacing B with C. The user's intent in replacing
4428           * is to go from M(A,B) to M(A,C). If the user decides to cancel
4429           * the replace by detaching C, the expected behavior is to end up
4430           * M(A,B). But suppose that right after deciding to detach C,
4431           * the replacement of B completes. We would have M(A,C), and then
4432           * ask to detach C, which would leave us with just A -- not what
4433           * the user wanted. To prevent this, we make sure that the
4434           * parent/child relationship hasn't changed -- in this example,
4435           * that C's parent is still the replacing vdev R.
4436           */
4437          if (pvd->vdev_guid != pguid && pguid != 0)
4438              return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4440          /*
4441           * Only 'replacing' or 'spare' vdevs can be replaced.
4442           */
4443          if (replace_done && pvd->vdev_ops != &vdev_replacing_ops &&
4444              pvd->vdev_ops != &vdev_spare_ops)
4445              return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

```

```

4447     ASSERT(pvd->vdev_ops != &vdev_spare_ops ||
4448            spa_version(spa) >= SPA_VERSION_SPARES);

4450     /*
4451      * Only mirror, replacing, and spare vdevs support detach.
4452      */
4453     if (pvd->vdev_ops != &vdev_replacing_ops &&
4454         pvd->vdev_ops != &vdev_mirror_ops &&
4455         pvd->vdev_ops != &vdev_spare_ops)
4456         return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4458     /*
4459      * If this device has the only valid copy of some data,
4460      * we cannot safely detach it.
4461      */
4462     if (vdev_dtl_required(vd))
4463         return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4465     ASSERT(pvd->vdev_children >= 2);

4467     /*
4468      * If we are detaching the second disk from a replacing vdev, then
4469      * check to see if we changed the original vdev's path to have "/old"
4470      * at the end in spa_vdev_attach(). If so, undo that change now.
4471      */
4472     if (pvd->vdev_ops == &vdev_replacing_ops && vd->vdev_id > 0 &&
4473         vd->vdev_path != NULL) {
4474         size_t len = strlen(vd->vdev_path);

4476         for (int c = 0; c < pvd->vdev_children; c++) {
4477             cvd = pvd->vdev_child[c];

4479             if (cvd == vd || cvd->vdev_path == NULL)
4480                 continue;

4482             if (strncmp(cvd->vdev_path, vd->vdev_path, len) == 0 &&
4483                 strcmp(cvd->vdev_path + len, "/old") == 0) {
4484                 spa_strfree(cvd->vdev_path);
4485                 cvd->vdev_path = spa_strdup(vd->vdev_path);
4486                 break;
4487             }
4488         }
4489     }

4491     /*
4492      * If we are detaching the original disk from a spare, then it implies
4493      * that the spare should become a real disk, and be removed from the
4494      * active spare list for the pool.
4495      */
4496     if (pvd->vdev_ops == &vdev_spare_ops &&
4497         vd->vdev_id == 0 &&
4498         pvd->vdev_child[pvd->vdev_children - 1]->vdev_isspare)
4499         unspare = B_TRUE;

4501     /*
4502      * Erase the disk labels so the disk can be used for other things.
4503      * This must be done after all other error cases are handled,
4504      * but before we disemowel vd (so we can still do I/O to it).
4505      * But if we can't do it, don't treat the error as fatal --
4506      * it may be that the unwritability of the disk is the reason
4507      * it's being detached!
4508      */
4509     error = vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);

4511     /*
4512      * Remove vd from its parent and compact the parent's children.

```

```

4513     /*
4514     vdev_remove_child(pvd, vd);
4515     vdev_compact_children(pvd);

4517     /*
4518     * Remember one of the remaining children so we can get tvd below.
4519     */
4520     cvd = pvd->vdev_child[pvd->vdev_children - 1];

4522     /*
4523     * If we need to remove the remaining child from the list of hot spares,
4524     * do it now, marking the vdev as no longer a spare in the process.
4525     * We must do this before vdev_remove_parent(), because that can
4526     * change the GUID if it creates a new toplevel GUID. For a similar
4527     * reason, we must remove the spare now, in the same txg as the detach;
4528     * otherwise someone could attach a new sibling, change the GUID, and
4529     * the subsequent attempt to spa_vdev_remove(unspare_guid) would fail.
4530     */
4531     if (unspare) {
4532         ASSERT(cvd->vdev_isspare);
4533         spa_spare_remove(cvd);
4534         unspare_guid = cvd->vdev_guid;
4535         (void) spa_vdev_remove(spa, unspare_guid, B_TRUE);
4536         cvd->vdev_unspare = B_TRUE;
4537     }

4539     /*
4540     * If the parent mirror/replacing vdev only has one child,
4541     * the parent is no longer needed. Remove it from the tree.
4542     */
4543     if (pvd->vdev_children == 1) {
4544         if (pvd->vdev_ops == &vdev_spare_ops)
4545             cvd->vdev_unspare = B_FALSE;
4546         vdev_remove_parent(cvd);
4547         cvd->vdev_resilvering = B_FALSE;
4548     }

4551     /*
4552     * We don't set tvd until now because the parent we just removed
4553     * may have been the previous top-level vdev.
4554     */
4555     tvd = cvd->vdev_top;
4556     ASSERT(tvd->vdev_parent == rvd);

4558     /*
4559     * Reevaluate the parent vdev state.
4560     */
4561     vdev_propagate_state(cvd);

4563     /*
4564     * If the 'autoexpand' property is set on the pool then automatically
4565     * try to expand the size of the pool. For example if the device we
4566     * just detached was smaller than the others, it may be possible to
4567     * add metaslabs (i.e. grow the pool). We need to reopen the vdev
4568     * first so that we can obtain the updated sizes of the leaf vdevs.
4569     */
4570     if (spa->spa_autoexpand) {
4571         vdev_reopen(tvd);
4572         vdev_expand(tvd, txg);
4573     }

4575     vdev_config_dirty(tvd);

4577     /*
4578     * Mark vd's DTL as dirty in this txg. vdev_dtl_sync() will see that

```

```

4579     * vd->vdev_detached is set and free vd's DTL object in syncing context.
4580     * But first make sure we're not on any *other* txg's DTL list, to
4581     * prevent vd from being accessed after it's freed.
4582     */
4583     vdp_path = spa_strdup(vd->vdev_path);
4584     for (int t = 0; t < TXG_SIZE; t++)
4585         (void) txg_list_remove_this(&tvd->vdev_dtl_list, vd, t);
4586     vd->vdev_detached = B_TRUE;
4587     vdev_dirty(tvd, VDD_DTL, vd, txg);

4589     spa_event_notify(spa, vd, ESC_ZFS_VDEV_REMOVE);

4591     /* hang on to the spa before we release the lock */
4592     spa_open_ref(spa, FTAG);

4594     error = spa_vdev_exit(spa, vd, txg, 0);

4596     spa_history_log_internal(spa, "detach", NULL,
3917     spa_history_log_internal(LOG_POOL_VDEV_DETACH, spa, NULL,
4597     "vdev=%s", vdp_path);
4598     spa_strfree(vdp_path);

4600     /*
4601     * If this was the removal of the original device in a hot spare vdev,
4602     * then we want to go through and remove the device from the hot spare
4603     * list of every other pool.
4604     */
4605     if (unspare) {
4606         spa_t *altspa = NULL;

4608         mutex_enter(&spa_namespace_lock);
4609         while ((altspa = spa_next(altspa)) != NULL) {
4610             if (altspa->spa_state != POOL_STATE_ACTIVE ||
4611                 altspa == spa)
4612                 continue;

4614             spa_open_ref(altspa, FTAG);
4615             mutex_exit(&spa_namespace_lock);
4616             (void) spa_vdev_remove(altspa, unspare_guid, B_TRUE);
4617             mutex_enter(&spa_namespace_lock);
4618             spa_close(altspa, FTAG);
4619         }
4620         mutex_exit(&spa_namespace_lock);

4622         /* search the rest of the vdevs for spares to remove */
4623         spa_vdev_resilver_done(spa);
4624     }

4626     /* all done with the spa; OK to release */
4627     mutex_enter(&spa_namespace_lock);
4628     spa_close(spa, FTAG);
4629     mutex_exit(&spa_namespace_lock);

4631     return (error);
4632 }

4634 /*
4635 * Split a set of devices from their mirrors, and create a new pool from them.
4636 */
4637 int
4638 spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
4639     nvlist_t *props, boolean_t exp)
4640 {
4641     int error = 0;
4642     uint64_t txg, *glist;
4643     spa_t *newspa;

```

```

4644     uint_t c, children, lastlog;
4645     nvlist_t **child, *nvl, *tmp;
4646     dmu_tx_t *tx;
4647     char *altroot = NULL;
4648     vdev_t *rvd, **vml = NULL;
4649     boolean_t activate_slog;

4651     ASSERT(spa_writeable(spa));

4653     txg = spa_vdev_enter(spa);

4655     /* clear the log and flush everything up to now */
4656     activate_slog = spa_passivate_log(spa);
4657     (void) spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4658     error = spa_offline_log(spa);
4659     txg = spa_vdev_config_enter(spa);

4661     if (activate_slog)
4662         spa_activate_log(spa);

4664     if (error != 0)
4665         return (spa_vdev_exit(spa, NULL, txg, error));

4667     /* check new spa name before going any further */
4668     if (spa_lookup(newname) != NULL)
4669         return (spa_vdev_exit(spa, NULL, txg, EEXIST));

4671     /*
4672     * scan through all the children to ensure they're all mirrors
4673     */
4674     if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvl) != 0 ||
4675         nvlist_lookup_nvlist_array(nvl, ZPOOL_CONFIG_CHILDREN, &child,
4676             &children) != 0)
4677         return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4679     /* first, check to ensure we've got the right child count */
4680     rvd = spa->spa_root_vdev;
4681     lastlog = 0;
4682     for (c = 0; c < rvd->vdev_children; c++) {
4683         vdev_t *vd = rvd->vdev_child[c];

4685         /* don't count the holes & logs as children */
4686         if (vd->vdev_islog || vd->vdev_ishole) {
4687             if (lastlog == 0)
4688                 lastlog = c;
4689             continue;
4690         }

4692         lastlog = 0;
4693     }
4694     if (children != (lastlog != 0 ? lastlog : rvd->vdev_children))
4695         return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4697     /* next, ensure no spare or cache devices are part of the split */
4698     if (nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_SPARES, &tmp) == 0 ||
4699         nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_L2CACHE, &tmp) == 0)
4700         return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4702     vml = kmem_zalloc(children * sizeof(vdev_t *), KM_SLEEP);
4703     glist = kmem_zalloc(children * sizeof(uint64_t), KM_SLEEP);

4705     /* then, loop over each vdev and validate it */
4706     for (c = 0; c < children; c++) {
4707         uint64_t is_hole = 0;

4709         (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_HOLE,

```

```

4710         &is_hole);
4711
4712     if (is_hole != 0) {
4713         if (spa->spa_root_vdev->vdev_child[c]->vdev_ishole ||
4714             spa->spa_root_vdev->vdev_child[c]->vdev_islog) {
4715             continue;
4716         } else {
4717             error = EINVAL;
4718             break;
4719         }
4720     }
4721
4722     /* which disk is going to be split? */
4723     if (nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_GUID,
4724         &glist[c]) != 0) {
4725         error = EINVAL;
4726         break;
4727     }
4728
4729     /* look it up in the spa */
4730     vml[c] = spa_lookup_by_guid(spa, glist[c], B_FALSE);
4731     if (vml[c] == NULL) {
4732         error = ENODEV;
4733         break;
4734     }
4735
4736     /* make sure there's nothing stopping the split */
4737     if (vml[c]->vdev_parent->vdev_ops != &vdev_mirror_ops ||
4738         vml[c]->vdev_islog ||
4739         vml[c]->vdev_ishole ||
4740         vml[c]->vdev_isspare ||
4741         vml[c]->vdev_isl2cache ||
4742         !vdev_writeable(vml[c]) ||
4743         vml[c]->vdev_children != 0 ||
4744         vml[c]->vdev_state != VDEV_STATE_HEALTHY ||
4745         c != spa->spa_root_vdev->vdev_child[c]->vdev_id) {
4746         error = EINVAL;
4747         break;
4748     }
4749
4750     if (vdev_dtl_required(vml[c])) {
4751         error = EBUSY;
4752         break;
4753     }
4754
4755     /* we need certain info from the top level */
4756     VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_ARRAY,
4757         vml[c]->vdev_top->vdev_ms_array) == 0);
4758     VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_SHIFT,
4759         vml[c]->vdev_top->vdev_ms_shift) == 0);
4760     VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASIZE,
4761         vml[c]->vdev_top->vdev_asize) == 0);
4762     VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASHIFT,
4763         vml[c]->vdev_top->vdev_ashift) == 0);
4764 }
4765
4766 if (error != 0) {
4767     kmem_free(vml, children * sizeof (vdev_t *));
4768     kmem_free(glist, children * sizeof (uint64_t));
4769     return (spa_vdev_exit(spa, NULL, txg, error));
4770 }
4771
4772 /* stop writers from using the disks */
4773 for (c = 0; c < children; c++) {
4774     if (vml[c] != NULL)
4775         vml[c]->vdev_offline = B_TRUE;

```

```

4776     }
4777     vdev_reopen(spa->spa_root_vdev);
4778
4779     /*
4780     * Temporarily record the splitting vdevs in the spa config. This
4781     * will disappear once the config is regenerated.
4782     */
4783     VERIFY(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);
4784     VERIFY(nvlist_add_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST,
4785         glist, children) == 0);
4786     kmem_free(glist, children * sizeof (uint64_t));
4787
4788     mutex_enter(&spa->spa_props_lock);
4789     VERIFY(nvlist_add_nvlist(spa->spa_config, ZPOOL_CONFIG_SPLIT,
4790         nvl) == 0);
4791     mutex_exit(&spa->spa_props_lock);
4792     spa->spa_config_splitting = nvl;
4793     vdev_config_dirty(spa->spa_root_vdev);
4794
4795     /* configure and create the new pool */
4796     VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME, newname) == 0);
4797     VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
4798         exp ? POOL_STATE_EXPORTED : POOL_STATE_ACTIVE) == 0);
4799     VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
4800         spa_version(spa)) == 0);
4801     VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG,
4802         spa->spa_config_txg) == 0);
4803     VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID,
4804         spa_generate_guid(NULL)) == 0);
4805     (void) nvlist_lookup_string(props,
4806         zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
4807
4808     /* add the new pool to the namespace */
4809     newspa = spa_add(newname, config, altroot);
4810     newspa->spa_config_txg = spa->spa_config_txg;
4811     spa_set_log_state(newspa, SPA_LOG_CLEAR);
4812
4813     /* release the spa config lock, retaining the namespace lock */
4814     spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4815
4816     if (zio_injection_enabled)
4817         zio_handle_panic_injection(spa, FTAG, 1);
4818
4819     spa_activate(newspa, spa_mode_global);
4820     spa_async_suspend(newspa);
4821
4822     /* create the new pool from the disks of the original pool */
4823     error = spa_load(newspa, SPA_LOAD_IMPORT, SPA_IMPORT_ASSEMBLE, B_TRUE);
4824     if (error)
4825         goto out;
4826
4827     /* if that worked, generate a real config for the new pool */
4828     if (newspa->spa_root_vdev != NULL) {
4829         VERIFY(nvlist_alloc(&newspa->spa_config_splitting,
4830             NV_UNIQUE_NAME, KM_SLEEP) == 0);
4831         VERIFY(nvlist_add_uint64(newspa->spa_config_splitting,
4832             ZPOOL_CONFIG_SPLIT_GUID, spa_guid(spa)) == 0);
4833         spa_config_set(newspa, spa_config_generate(newspa, NULL, -1ULL,
4834             B_TRUE));
4835     }
4836
4837     /* set the props */
4838     if (props != NULL) {
4839         spa_configfile_set(newspa, props, B_FALSE);
4840         error = spa_prop_set(newspa, props);
4841         if (error)

```



```

4842         goto out;
4843     }
4844
4845     /* flush everything */
4846     txg = spa_vdev_config_enter(newspa);
4847     vdev_config_dirty(newspa->spa_root_vdev);
4848     (void) spa_vdev_config_exit(newspa, NULL, txg, 0, FTAG);
4849
4850     if (zio_injection_enabled)
4851         zio_handle_panic_injection(spa, FTAG, 2);
4852
4853     spa_async_resume(newspa);
4854
4855     /* finally, update the original pool's config */
4856     txg = spa_vdev_config_enter(spa);
4857     tx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
4858     error = dmu_tx_assign(tx, TXG_WAIT);
4859     if (error != 0)
4860         dmu_tx_abort(tx);
4861     for (c = 0; c < children; c++) {
4862         if (vml[c] != NULL) {
4863             vdev_split(vml[c]);
4864             if (error == 0)
4865                 spa_history_log_internal(spa, "detach", tx,
4866                     "vdev=%s", vml[c]->vdev_path);
4867             spa_history_log_internal(LOG_POOL_VDEV_DETACH,
4868                 spa, tx, "vdev=%s",
4869                 vml[c]->vdev_path);
4870             vdev_free(vml[c]);
4871         }
4872     }
4873     vdev_config_dirty(spa->spa_root_vdev);
4874     spa->spa_config_splitting = NULL;
4875     nvlist_free(nvl);
4876     if (error == 0)
4877         dmu_tx_commit(tx);
4878     (void) spa_vdev_exit(spa, NULL, txg, 0);
4879
4880     if (zio_injection_enabled)
4881         zio_handle_panic_injection(spa, FTAG, 3);
4882
4883     /* split is complete; log a history record */
4884     spa_history_log_internal(newspa, "split", NULL,
4885         "from pool %s", spa_name(spa));
4886     spa_history_log_internal(LOG_POOL_SPLIT, newspa, NULL,
4887         "split new pool %s from pool %s", newname, spa_name(spa));
4888
4889     kmem_free(vml, children * sizeof (vdev_t *));
4890
4891     /* if we're not going to mount the filesystems in userland, export */
4892     if (exp)
4893         error = spa_export_common(newname, POOL_STATE_EXPORTED, NULL,
4894             B_FALSE, B_FALSE);
4895
4896     return (error);
4897
4898     out:
4899     spa_unload(newspa);
4900     spa_deactivate(newspa);
4901     spa_remove(newspa);
4902
4903     txg = spa_vdev_config_enter(spa);
4904
4905     /* re-online all offlined disks */
4906     for (c = 0; c < children; c++) {
4907         if (vml[c] != NULL)

```

```

4903         vml[c]->vdev_offline = B_FALSE;
4904     }
4905     vdev_reopen(spa->spa_root_vdev);
4906
4907     nvlist_free(spa->spa_config_splitting);
4908     spa->spa_config_splitting = NULL;
4909     (void) spa_vdev_exit(spa, NULL, txg, error);
4910
4911     kmem_free(vml, children * sizeof (vdev_t *));
4912     return (error);
4913 }
4914
4915 unchanged portion omitted
4916
4917 static void
4918 spa_async_thread(spa_t *spa)
4919 {
4920     int tasks;
4921
4922     ASSERT(spa->spa_sync_on);
4923
4924     mutex_enter(&spa->spa_async_lock);
4925     tasks = spa->spa_async_tasks;
4926     spa->spa_async_tasks = 0;
4927     mutex_exit(&spa->spa_async_lock);
4928
4929     /*
4930      * See if the config needs to be updated.
4931      */
4932     if (tasks & SPA_ASYNC_CONFIG_UPDATE) {
4933         uint64_t old_space, new_space;
4934
4935         mutex_enter(&spa_namespace_lock);
4936         old_space = metaslab_class_get_space(spa_normal_class(spa));
4937         spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
4938         new_space = metaslab_class_get_space(spa_normal_class(spa));
4939         mutex_exit(&spa_namespace_lock);
4940
4941         /*
4942          * If the pool grew as a result of the config update,
4943          * then log an internal history event.
4944          */
4945         if (new_space != old_space) {
4946             spa_history_log_internal(spa, "vdev online", NULL,
4947                 spa_history_log_internal(LOG_POOL_VDEV_ONLINE,
4948                     spa, NULL,
4949                     "pool '%s' size: %llu(+%llu)",
4950                     spa_name(spa), new_space, new_space - old_space);
4951         }
4952     }
4953
4954     /*
4955      * See if any devices need to be marked REMOVED.
4956      */
4957     if (tasks & SPA_ASYNC_REMOVE) {
4958         spa_vdev_state_enter(spa, SCL_NONE);
4959         spa_async_remove(spa, spa->spa_root_vdev);
4960         for (int i = 0; i < spa->spa_l2cache.sav_count; i++)
4961             spa_async_remove(spa, spa->spa_l2cache.sav_vdevs[i]);
4962         for (int i = 0; i < spa->spa_spare.sav_count; i++)
4963             spa_async_remove(spa, spa->spa_spare.sav_vdevs[i]);
4964         (void) spa_vdev_state_exit(spa, NULL, 0);
4965     }
4966
4967     if ((tasks & SPA_ASYNC_AUTOEXPAND) && !spa_suspended(spa)) {
4968         spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
4969         spa_async_autoexpand(spa, spa->spa_root_vdev);
4970     }

```

```

5490         spa_config_exit(spa, SCL_CONFIG, FTAG);
5491     }

5493     /*
5494     * See if any devices need to be probed.
5495     */
5496     if (tasks & SPA_ASYNC_PROBE) {
5497         spa_vdev_state_enter(spa, SCL_NONE);
5498         spa_async_probe(spa, spa->spa_root_vdev);
5499         (void) spa_vdev_state_exit(spa, NULL, 0);
5500     }

5502     /*
5503     * If any devices are done replacing, detach them.
5504     */
5505     if (tasks & SPA_ASYNC_RESILVER_DONE)
5506         spa_vdev_resilver_done(spa);

5508     /*
5509     * Kick off a resilver.
5510     */
5511     if (tasks & SPA_ASYNC_RESILVER)
5512         dsl_resilver_restart(spa->spa_dsl_pool, 0);

5514     /*
5515     * Let the world know that we're done.
5516     */
5517     mutex_enter(&spa->spa_async_lock);
5518     spa->spa_async_thread = NULL;
5519     cv_broadcast(&spa->spa_async_cv);
5520     mutex_exit(&spa->spa_async_lock);
5521     thread_exit();
5522 }

_____ unchanged portion omitted _____

5688 static void
5689 spa_sync_version(void *arg1, void *arg2, dmu_tx_t *tx)
5690 {
5691     spa_t *spa = arg1;
5692     uint64_t version = *(uint64_t *)arg2;

5694     /*
5695     * Setting the version is special cased when first creating the pool.
5696     */
5697     ASSERT(tx->tx_txx != TXG_INITIAL);

5699     ASSERT(version <= SPA_VERSION);
5700     ASSERT(version >= spa_version(spa));

5702     spa->spa_uberblock.ub_version = version;
5703     vdev_config_dirty(spa->spa_root_vdev);
5704     spa_history_log_internal(spa, "set", tx, "version=%lld", version);
5705 #endif /* ! codereview */
5706 }

5708 /*
5709 * Set zpool properties.
5710 */
5711 static void
5712 spa_sync_props(void *arg1, void *arg2, dmu_tx_t *tx)
5713 {
5714     spa_t *spa = arg1;
5715     objset_t *mos = spa->spa_meta_objset;
5716     nvlist_t *nvp = arg2;
5717     nvpair_t *elem = NULL;

```

```

5719     mutex_enter(&spa->spa_props_lock);

5721     while ((elem = nvlist_next_nvpair(nvp, elem)) {
5722         uint64_t intval;
5723         char *strval, *fname;
5724         zpool_prop_t prop;
5725         const char *propname;
5726         zprop_type_t proptype;
5727         zfeature_info_t *feature;

5729         switch (prop = zpool_name_to_prop(nvpair_name(elem))) {
5730         case ZPROP_INVALID:
5731             /*
5732              * We checked this earlier in spa_prop_validate().
5733              */
5734             ASSERT(zpool_prop_feature(nvpair_name(elem)));

5736             fname = strchr(nvpair_name(elem), '@') + 1;
5737             VERIFY3U(0, ==, zfeature_lookup_name(fname, &feature));

5739             spa_feature_enable(spa, feature, tx);
5740             spa_history_log_internal(spa, "set", tx,
5741                                     "%s=enabled", nvpair_name(elem));
5742 #endif /* ! codereview */
5743             break;

5745         case ZPOOL_PROP_VERSION:
5746             VERIFY(nvpair_value_uint64(elem, &intval) == 0);
5747             /*
5748              * The version is synced separately before other
5749              * properties and should be correct by now.
5750              */
5751             ASSERT3U(spa_version(spa), >=, intval);
5752             break;

5754         case ZPOOL_PROP_ALTROOT:
5755             /*
5756              * 'altroot' is a non-persistent property. It should
5757              * have been set temporarily at creation or import time.
5758              */
5759             ASSERT(spa->spa_root != NULL);
5760             break;

5762         case ZPOOL_PROP_READONLY:
5763         case ZPOOL_PROP_CACHEFILE:
5764             /*
5765              * 'readonly' and 'cachefile' are also non-persistent
5766              * properties.
5767              */
5768             break;
5769         case ZPOOL_PROP_COMMENT:
5770             VERIFY(nvpair_value_string(elem, &strval) == 0);
5771             if (spa->spa_comment != NULL)
5772                 spa_strfree(spa->spa_comment);
5773             spa->spa_comment = spa_strdup(strval);
5774             /*
5775              * We need to dirty the configuration on all the vdevs
5776              * so that their labels get updated. It's unnecessary
5777              * to do this for pool creation since the vdev's
5778              * configuratoion has already been dirtied.
5779              */
5780             if (tx->tx_txx != TXG_INITIAL)
5781                 vdev_config_dirty(spa->spa_root_vdev);
5782             spa_history_log_internal(spa, "set", tx,
5783                                     "%s=%s", nvpair_name(elem), strval);
5784 #endif /* ! codereview */

```

```

5785         break;
5786     default:
5787         /*
5788          * Set pool property values in the poolprops mos object.
5789          */
5790         if (spa->spa_pool_props_object == 0) {
5791             spa->spa_pool_props_object =
5792                 zap_create_link(mos, DMU_OT_POOL_PROPS,
5793                     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_PROPS,
5794                     tx);
5795         }
5797         /* normalize the property name */
5798         propname = zpool_prop_to_name(prop);
5799         proptype = zpool_prop_get_type(prop);
5801         if (nvpair_type(elem) == DATA_TYPE_STRING) {
5802             ASSERT(proptype == PROP_TYPE_STRING);
5803             VERIFY(nvpair_value_string(elem, &strval) == 0);
5804             VERIFY(zap_update(mos,
5805                 spa->spa_pool_props_object, propname,
5806                 1, strlen(strval) + 1, strval, tx) == 0);
5807             spa_history_log_internal(spa, "set", tx,
5808                 "%s=%s", nvpair_name(elem), strval);
5809         } else if (nvpair_type(elem) == DATA_TYPE_UINT64) {
5810             VERIFY(nvpair_value_uint64(elem, &intval) == 0);
5812             if (proptype == PROP_TYPE_INDEX) {
5813                 const char *unused;
5814                 VERIFY(zpool_prop_index_to_string(
5815                     prop, intval, &unused) == 0);
5816             }
5817             VERIFY(zap_update(mos,
5818                 spa->spa_pool_props_object, propname,
5819                 8, 1, &intval, tx) == 0);
5820             spa_history_log_internal(spa, "set", tx,
5821                 "%s=%lld", nvpair_name(elem), intval);
5822 #endif /* ! codereview */
5823         } else {
5824             ASSERT(0); /* not allowed */
5825         }
5827         switch (prop) {
5828         case ZPOOL_PROP_DELEGATION:
5829             spa->spa_delegation = intval;
5830             break;
5831         case ZPOOL_PROP_BOOTFS:
5832             spa->spa_bootfs = intval;
5833             break;
5834         case ZPOOL_PROP_FAILUREMODE:
5835             spa->spa_failmode = intval;
5836             break;
5837         case ZPOOL_PROP_AUTOEXPAND:
5838             spa->spa_autoexpand = intval;
5839             if (tx->tx_txg != TXG_INITIAL)
5840                 spa_async_request(spa,
5841                     SPA_ASYNC_AUTOEXPAND);
5842             break;
5843         case ZPOOL_PROP_DEDUPDITTO:
5844             spa->spa_dedup_ditto = intval;
5845             break;
5846         default:
5847             break;
5848         }
5849     }

```

```

5039         /* log internal history if this is not a zpool create */
5040         if (spa_version(spa) >= SPA_VERSION_ZPOOL_HISTORY &&
5041             tx->tx_txg != TXG_INITIAL) {
5042             spa_history_log_internal(LOG_POOL_PROPSSET,
5043                 spa, tx, "%s %lld %s",
5044                 nvpair_name(elem), intval, spa_name(spa));
5045         }
5051     }
5853     mutex_exit(&spa->spa_props_lock);
5854 }

```

unchanged_portion_omitted

new/usr/src/uts/common/fs/zfs/spa_history.c

1

```
*****
14964 Thu Jun 28 15:57:18 2012
new/usr/src/uts/common/fs/zfs/spa_history.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2012 by Delphix. All rights reserved.
25  * Copyright (c) 2011 by Delphix. All rights reserved.
26 */
27 #include <sys/spa.h>
28 #include <sys/spa_impl.h>
29 #include <sys/zap.h>
30 #include <sys/dsl_synctask.h>
31 #include <sys/dmu_tx.h>
32 #include <sys/dmu_objset.h>
33 #include <sys/dsl_dataset.h>
34 #include <sys/dsl_dir.h>
35 #endif /* ! codereview */
36 #include <sys/utsname.h>
37 #include <sys/cmn_err.h>
38 #include <sys/sunddi.h>
39 #include <sys/cred.h>
40 #endif /* ! codereview */
41 #include "zfs_comutil.h"
42 #ifdef _KERNEL
43 #include <sys/zone.h>
44 #endif
45
46 /*
47  * Routines to manage the on-disk history log.
48  *
49  * The history log is stored as a dmu object containing
50  * <packed record length, record nvlist> tuples.
51  *
52  * Where "record nvlist" is a nvlist containing uint64_ts and strings, and
53  * "packed record length" is the packed length of the "record nvlist" stored
```

new/usr/src/uts/common/fs/zfs/spa_history.c

2

```
54 * as a little endian uint64_t.
55 *
56 * The log is implemented as a ring buffer, though the original creation
57 * of the pool ('zpool create') is never overwritten.
58 *
59 * The history log is tracked as object 'spa_t::spa_history'. The bonus buffer
60 * of 'spa_history' stores the offsets for logging/retrieving history as
61 * 'spa_history_phys_t'. 'sh_pool_create_len' is the ending offset in bytes of
62 * where the 'zpool create' record is stored. This allows us to never
63 * overwrite the original creation of the pool. 'sh_phys_max_off' is the
64 * physical ending offset in bytes of the log. This tells you the length of
65 * the buffer. 'sh_eof' is the logical EOF (in bytes). Whenever a record
66 * is added, 'sh_eof' is incremented by the size of the record.
67 * 'sh_eof' is never decremented. 'sh_bof' is the logical BOF (in bytes).
68 * This is where the consumer should start reading from after reading in
69 * the 'zpool create' portion of the log.
70 *
71 * 'sh_records_lost' keeps track of how many records have been overwritten
72 * and permanently lost.
73 */
74
75 /* convert a logical offset to physical */
76 static uint64_t
77 spa_history_log_to_phys(uint64_t log_off, spa_history_phys_t *shpp)
78 {
79     uint64_t phys_len;
80
81     phys_len = shpp->sh_phys_max_off - shpp->sh_pool_create_len;
82     return ((log_off - shpp->sh_pool_create_len) % phys_len
83         + shpp->sh_pool_create_len);
84 }
85
86 void
87 spa_history_create_obj(spa_t *spa, dmu_tx_t *tx)
88 {
89     dmu_buf_t *dbp;
90     spa_history_phys_t *shpp;
91     objset_t *mos = spa->spa_meta_objset;
92
93     ASSERT(spa->spa_history == 0);
94     spa->spa_history = dmu_object_alloc(mos, DMU_OT_SPA_HISTORY,
95         SPA_MAXBLOCKSIZE, DMU_OT_SPA_HISTORY_OFFSETS,
96         sizeof (spa_history_phys_t), tx);
97
98     VERIFY(zap_add(mos, DMU_POOL_DIRECTORY_OBJECT,
99         DMU_POOL_HISTORY, sizeof (uint64_t), 1,
100         &spa->spa_history, tx) == 0);
101
102     VERIFY(0 == dmu_bonus_hold(mos, spa->spa_history, FTAG, &dbp));
103     ASSERT(dbp->db_size >= sizeof (spa_history_phys_t));
104
105     shpp = dbp->db_data;
106     dmu_buf_will_dirty(dbp, tx);
107
108     /*
109      * Figure out maximum size of history log. We set it at
110      * 0.1% of pool size, with a max of 1G and min of 128KB.
111      */
112     shpp->sh_phys_max_off =
113         metaslab_class_get_dspace(spa_normal_class(spa)) / 1000;
114     shpp->sh_phys_max_off = MIN(shpp->sh_phys_max_off, 1<<30);
115     shpp->sh_phys_max_off = MAX(shpp->sh_phys_max_off, 128<<10);
116
117     dmu_buf_rele(dbp, FTAG);
118 }
```

```

120 /*
121  * Change 'sh_bof' to the beginning of the next record.
122  */
123 static int
124 spa_history_advance_bof(spa_t *spa, spa_history_phys_t *shpp)
125 {
126     objset_t *mos = spa->spa_meta_objset;
127     uint64_t firstread, reclen, phys_bof;
128     char buf[sizeof (reclen)];
129     int err;

131     phys_bof = spa_history_log_to_phys(shpp->sh_bof, shpp);
132     firstread = MIN(sizeof (reclen), shpp->sh_phys_max_off - phys_bof);

134     if ((err = dmu_read(mos, spa->spa_history, phys_bof, firstread,
135         buf, DMU_READ_PREFETCH)) != 0)
136         return (err);
137     if (firstread != sizeof (reclen)) {
138         if ((err = dmu_read(mos, spa->spa_history,
139             shpp->sh_pool_create_len, sizeof (reclen) - firstread,
140             buf + firstread, DMU_READ_PREFETCH)) != 0)
141             return (err);
142     }

144     reclen = LE_64*((uint64_t *)buf);
145     shpp->sh_bof += reclen + sizeof (reclen);
146     shpp->sh_records_lost++;
147     return (0);
148 }

150 static int
151 spa_history_write(spa_t *spa, void *buf, uint64_t len, spa_history_phys_t *shpp,
152     dmu_tx_t *tx)
153 {
154     uint64_t firstwrite, phys_eof;
155     objset_t *mos = spa->spa_meta_objset;
156     int err;

158     ASSERT(MUTEX_HELD(&spa->spa_history_lock));

160     /* see if we need to reset logical BOF */
161     while (shpp->sh_phys_max_off - shpp->sh_pool_create_len -
162         (shpp->sh_eof - shpp->sh_bof) <= len) {
163         if ((err = spa_history_advance_bof(spa, shpp)) != 0) {
164             return (err);
165         }
166     }

168     phys_eof = spa_history_log_to_phys(shpp->sh_eof, shpp);
169     firstwrite = MIN(len, shpp->sh_phys_max_off - phys_eof);
170     shpp->sh_eof += len;
171     dmu_write(mos, spa->spa_history, phys_eof, firstwrite, buf, tx);

173     len -= firstwrite;
174     if (len > 0) {
175         /* write out the rest at the beginning of physical file */
176         dmu_write(mos, spa->spa_history, shpp->sh_pool_create_len,
177             len, (char *)buf + firstwrite, tx);
178     }

180     return (0);
181 }

183 static char *
184 spa_history_zone(void)
185 {
186     spa_history_zone();

```

```

185 {
186 #ifdef _KERNEL
187     if (INGLOBALZONE(curproc))
188         return (NULL);
189 #endif /* !codereview */
190     return (curproc->p_zone->zone_name);
191 #else
192     return (NULL);
193     return ("global");
194 #endif
195 }

196 /*
197  * Write out a history event.
198  */
199 /*ARGSUSED*/
200 static void
201 spa_history_log_sync(void *arg1, void *arg2, dmu_tx_t *tx)
202 {
203     spa_t *spa = arg1;
204     nvlist_t *nvl = arg2;
205     history_arg_t *hap = arg2;
206     const char *history_str = hap->ha_history_str;
207     objset_t *mos = spa->spa_meta_objset;
208     dmu_buf_t *dbp;
209     spa_history_phys_t *shpp;
210     size_t reclen;
211     uint64_t le_len;
212     nvlist_t *nvrecord;
213     char *record_packed = NULL;
214     int ret;

216     /*
217      * If we have an older pool that doesn't have a command
218      * history object, create it now.
219      */
220     mutex_enter(&spa->spa_history_lock);
221     if (!spa->spa_history)
222         spa_history_create_obj(spa, tx);
223     mutex_exit(&spa->spa_history_lock);

225     /*
226      * Get the offset of where we need to write via the bonus buffer.
227      * Update the offset when the write completes.
228      */
229     VERIFY(0 == dmu_bonus_hold(mos, spa->spa_history, FTAG, &dbp));
230     shpp = dbp->db_data;

232     dmu_buf_will_dirty(dbp, tx);

234 #ifdef ZFS_DEBUG
235 {
236     dmu_object_info_t doi;
237     dmu_object_info_from_db(dbp, &doi);
238     ASSERT3U(doi.doi_bonus_type, ==, DMU_OT_SPA_HISTORY_OFFSETS);
239 }
240 #endif

242     fnvlist_add_uint64(nvl, ZPOOL_HIST_TIME, gethrestime_sec());
243     VERIFY(nvlist_alloc(&nvrecord, NV_UNIQUE_NAME, KM_SLEEP) == 0);
244     VERIFY(nvlist_add_uint64(nvrecord, ZPOOL_HIST_TIME,
245         gethrestime_sec()) == 0);
246     VERIFY(nvlist_add_uint64(nvrecord, ZPOOL_HIST_WHO, hap->ha_uid) == 0);
247     if (hap->ha_zone != NULL)
248         VERIFY(nvlist_add_string(nvrecord, ZPOOL_HIST_ZONE,
249             hap->ha_zone) == 0);

```

```

240 #ifndef _KERNEL
241     fnvlist_add_string(nvl, ZPOOL_HIST_HOST, utsname.nodename);
93     VERIFY(nvlist_add_string(nvrecord, ZPOOL_HIST_HOST,
94         utsname.nodename) == 0);
242 #endif
243     if (nvlist_exists(nvl, ZPOOL_HIST_CMD)) {
244         zfs_dbgmsg("command: %s",
245             fnvlist_lookup_string(nvl, ZPOOL_HIST_CMD));
246     } else if (nvlist_exists(nvl, ZPOOL_HIST_INT_NAME)) {
247         if (nvlist_exists(nvl, ZPOOL_HIST_DSNAME)) {
248             zfs_dbgmsg("txg %lld %s %s (id %llu) %s",
249                 fnvlist_lookup_uint64(nvl, ZPOOL_HIST_TXG),
250                 fnvlist_lookup_string(nvl, ZPOOL_HIST_INT_NAME),
251                 fnvlist_lookup_string(nvl, ZPOOL_HIST_DSNAME),
252                 fnvlist_lookup_uint64(nvl, ZPOOL_HIST_DSID),
253                 fnvlist_lookup_string(nvl, ZPOOL_HIST_INT_STR));
96         if (hap->ha_log_type == LOG_CMD_POOL_CREATE ||
97             hap->ha_log_type == LOG_CMD_NORMAL) {
98             VERIFY(nvlist_add_string(nvrecord, ZPOOL_HIST_CMD,
99                 history_str) == 0);
101
254             zfs_dbgmsg("command: %s", history_str);
255         } else {
256             zfs_dbgmsg("txg %lld %s %s",
257                 fnvlist_lookup_uint64(nvl, ZPOOL_HIST_TXG),
258                 fnvlist_lookup_string(nvl, ZPOOL_HIST_INT_NAME),
259                 fnvlist_lookup_string(nvl, ZPOOL_HIST_INT_STR));
260         } else if (nvlist_exists(nvl, ZPOOL_HIST_IOCTL)) {
261             zfs_dbgmsg("ioctl %s",
262                 fnvlist_lookup_string(nvl, ZPOOL_HIST_IOCTL));
103             VERIFY(nvlist_add_uint64(nvrecord, ZPOOL_HIST_INT_EVENT,
104                 hap->ha_event) == 0);
105             VERIFY(nvlist_add_uint64(nvrecord, ZPOOL_HIST_TXG,
106                 tx->tx_txg) == 0);
107             VERIFY(nvlist_add_string(nvrecord, ZPOOL_HIST_INT_STR,
108                 history_str) == 0);
110
111             zfs_dbgmsg("internal %s pool:%s txg:%llu %s",
112                 zfs_history_event_names[hap->ha_event], spa_name(spa),
113                 (longlong_t)tx->tx_txg, history_str);
263     }
265     record_packed = fnvlist_pack(nvl, &reclen);
116     VERIFY(nvlist_size(nvrecord, &reclen, NV_ENCODE_XDR) == 0);
117     record_packed = kmem_alloc(reclen, KM_SLEEP);
119     VERIFY(nvlist_pack(nvrecord, &record_packed, &reclen,
120         NV_ENCODE_XDR, KM_SLEEP) == 0);
267     mutex_enter(&spa->spa_history_lock);
123     if (hap->ha_log_type == LOG_CMD_POOL_CREATE)
124         VERIFY(shpp->sh_eof == shpp->sh_pool_create_len);
269     /* write out the packed length as little endian */
270     le_len = LE_64((uint64_t)reclen);
271     ret = spa_history_write(spa, &le_len, sizeof(le_len), shpp, tx);
272     if (!ret)
273         ret = spa_history_write(spa, record_packed, reclen, shpp, tx);
275     /* The first command is the create, which we keep forever */
276     if (ret == 0 && shpp->sh_pool_create_len == 0 &&
277         nvlist_exists(nvl, ZPOOL_HIST_CMD)) {
278         shpp->sh_pool_create_len = shpp->sh_bof = shpp->sh_eof;
132     if (!ret && hap->ha_log_type == LOG_CMD_POOL_CREATE) {

```

```

133         shpp->sh_pool_create_len += sizeof(le_len) + reclen;
134         shpp->sh_bof = shpp->sh_pool_create_len;
279     }
281     mutex_exit(&spa->spa_history_lock);
282     fnvlist_pack_free(record_packed, reclen);
138     nvlist_free(nvrecord);
139     kmem_free(record_packed, reclen);
283     dmu_buf_rele(dbp, FTAG);
284     fnvlist_free(nvl);
142     strfree(hap->ha_history_str);
143     if (hap->ha_zone != NULL)
144         strfree(hap->ha_zone);
145     kmem_free(hap, sizeof(history_arg_t));
285 }
287 /*
288  * Write out a history event.
289  */
290 int
291 spa_history_log(spa_t *spa, const char *msg)
292 {
293     int err;
294     nvlist_t *nvl = fnvlist_alloc();
296     fnvlist_add_string(nvl, ZPOOL_HIST_CMD, msg);
297     err = spa_history_log_nvl(spa, nvl);
298     fnvlist_free(nvl);
299     return (err);
300 }
302 int
303 spa_history_log_nvl(spa_t *spa, nvlist_t *nvl)
152 spa_history_log(spa_t *spa, const char *history_str, history_log_type_t what)
304 {
154     history_arg_t *ha;
305     int err = 0;
306     dmu_tx_t *tx;
307     nvlist_t *nvarg;
308 #endif /* ! codereview */
310     if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY)
311         return (EINVAL);
157     ASSERT(what != LOG_INTERNAL);
313     tx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
314     err = dmu_tx_assign(tx, TXG_WAIT);
315     if (err) {
316         dmu_tx_abort(tx);
317         return (err);
318     }
320     nvarg = fnvlist_dup(nvl);
321     if (spa_history_zone() != NULL) {
322         fnvlist_add_string(nvarg, ZPOOL_HIST_ZONE,
323             spa_history_zone());
324     }
325     fnvlist_add_uint64(nvarg, ZPOOL_HIST_WHO, crgetruid(CRED()));
166     ha = kmem_alloc(sizeof(history_arg_t), KM_SLEEP);
167     ha->ha_history_str = strdup(history_str);
168     ha->ha_zone = strdup(spa_history_zone());
169     ha->ha_log_type = what;
170     ha->ha_uid = crgetruid(CRED());
327     /* Kick this off asynchronously; errors are ignored. */

```

```

328     dsl_sync_task_do_nowait(spa_get_dsl(spa), NULL,
329     spa_history_log_sync, spa, nvarg, 0, tx);
174     spa_history_log_sync, spa, ha, 0, tx);
330     dmu_tx_commit(tx);

332     /* spa_history_log_sync will free nvl */
177     /* spa_history_log_sync will free ha and strings */
333     return (err);

335 #endif /* ! codereview */
336 }

338 /*
339  * Read out the command history.
340  */
341 int
342 spa_history_get(spa_t *spa, uint64_t *offp, uint64_t *len, char *buf)
343 {
344     objset_t *mos = spa->spa_meta_objset;
345     dmu_buf_t *dbp;
346     uint64_t read_len, phys_read_off, phys_eof;
347     uint64_t leftover = 0;
348     spa_history_phys_t *shpp;
349     int err;

351     /*
352      * If the command history doesn't exist (older pool),
353      * that's ok, just return ENOENT.
354      */
355     if (!spa->spa_history)
356         return (ENOENT);

358     /*
359      * The history is logged asynchronously, so when they request
360      * the first chunk of history, make sure everything has been
361      * synced to disk so that we get it.
362      */
363     if (*offp == 0 && spa_writeable(spa))
364         txg_wait_synced(spa_get_dsl(spa), 0);

366     if ((err = dmu_bonus_hold(mos, spa->spa_history, FTAG, &dbp)) != 0)
367         return (err);
368     shpp = dbp->db_data;

370 #ifdef ZFS_DEBUG
371     {
372         dmu_object_info_t doi;
373         dmu_object_info_from_db(dbp, &doi);
374         ASSERT3U(doi.doi_bonus_type, ==, DMU_OT_SPA_HISTORY_OFFSETS);
375     }
376 #endif

378     mutex_enter(&spa->spa_history_lock);
379     phys_eof = spa_history_log_to_phys(shpp->sh_eof, shpp);

381     if (*offp < shpp->sh_pool_create_len) {
382         /* read in just the zpool create history */
383         phys_read_off = *offp;
384         read_len = MIN(*len, shpp->sh_pool_create_len -
385             phys_read_off);
386     } else {
387         /*
388          * Need to reset passed in offset to BOF if the passed in
389          * offset has since been overwritten.
390          */
391         *offp = MAX(*offp, shpp->sh_bof);

```

```

392     phys_read_off = spa_history_log_to_phys(*offp, shpp);

394     /*
395      * Read up to the minimum of what the user passed down or
396      * the EOF (physical or logical). If we hit physical EOF,
397      * use 'leftover' to read from the physical BOF.
398      */
399     if (phys_read_off <= phys_eof) {
400         read_len = MIN(*len, phys_eof - phys_read_off);
401     } else {
402         read_len = MIN(*len,
403             shpp->sh_phys_max_off - phys_read_off);
404         if (phys_read_off + *len > shpp->sh_phys_max_off) {
405             leftover = MIN(*len - read_len,
406                 phys_eof - shpp->sh_pool_create_len);
407         }
408     }
409 }

411     /* offset for consumer to use next */
412     *offp += read_len + leftover;

414     /* tell the consumer how much you actually read */
415     *len = read_len + leftover;

417     if (read_len == 0) {
418         mutex_exit(&spa->spa_history_lock);
419         dmu_buf_rele(dbp, FTAG);
420         return (0);
421     }

423     err = dmu_read(mos, spa->spa_history, phys_read_off, read_len, buf,
424         DMU_READ_PREFETCH);
425     if (leftover && err == 0) {
426         err = dmu_read(mos, spa->spa_history, shpp->sh_pool_create_len,
427             leftover, buf + read_len, DMU_READ_PREFETCH);
428     }
429     mutex_exit(&spa->spa_history_lock);

431     dmu_buf_rele(dbp, FTAG);
432     return (err);
433 }

435 /*
436  * The nvlist will be consumed by this call.
437  */
438 #endif /* ! codereview */
439 static void
440 log_internal(nvlist_t *nvl, const char *operation, spa_t *spa,
441     log_internal(history_internal_events_t event, spa_t *spa,
442     dmu_tx_t *tx, const char *fmt, va_list adx)
443 {
444     char *msg;
182     history_arg_t *ha;

445     /*
446      * If this is part of creating a pool, not everything is
447      * initialized yet, so don't bother logging the internal events.
448      */
449     if (tx->tx_txg == TXG_INITIAL)
450         return;

452     msg = kmem_alloc(vsnprintf(NULL, 0, fmt, adx) + 1, KM_SLEEP);
453     (void) vsprintf(msg, fmt, adx);
454     fnvlist_add_string(nvl, ZPOOL_HIST_INT_STR, msg);
455     strfree(msg);

```

```

457     fnvlist_add_string(nvl, ZPOOL_HIST_INT_NAME, operation);
458     fnvlist_add_uint64(nvl, ZPOOL_HIST_TXG, tx->tx_txg);
459     ha = kmem_alloc(sizeof (history_arg_t), KM_SLEEP);
460     ha->ha_history_str = kmem_alloc(vsnprintf(NULL, 0, fmt, adx) + 1,
461     KM_SLEEP);
462
463     (void) vsprintf(ha->ha_history_str, fmt, adx);
464
465     ha->ha_log_type = LOG_INTERNAL;
466     ha->ha_event = event;
467     ha->ha_zone = NULL;
468     ha->ha_uid = 0;
469
470     if (dmu_tx_is_syncing(tx)) {
471         spa_history_log_sync(spa, nvl, tx);
472     } else {
473         dsl_sync_task_do_nowait(spa_get_dsl(spa), NULL,
474         spa_history_log_sync, spa, nvl, 0, tx);
475     }
476     /* spa_history_log_sync() will free nvl */
477     /* spa_history_log_sync() will free ha and strings */
478 }
479
480 void
481 spa_history_log_internal(spa_t *spa, const char *operation,
482 spa_history_log_internal(history_internal_events_t event, spa_t *spa,
483     dmu_tx_t *tx, const char *fmt, ...)
484 {
485     dmu_tx_t *htx = tx;
486     va_list adx;
487
488     /* create a tx if we didn't get one */
489     if (tx == NULL) {
490         htx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
491         if (dmu_tx_assign(htx, TXG_WAIT) != 0) {
492             dmu_tx_abort(htx);
493             return;
494         }
495     }
496
497     va_start(adx, fmt);
498     log_internal(fnvlist_alloc(), operation, spa, htx, fmt, adx);
499     log_internal(event, spa, htx, fmt, adx);
500     va_end(adx);
501
502     /* if we didn't get a tx from the caller, commit the one we made */
503     if (tx == NULL)
504         dmu_tx_commit(htx);
505 }
506
507 void
508 spa_history_log_internal_ds(dsl_dataset_t *ds, const char *operation,
509     dmu_tx_t *tx, const char *fmt, ...)
510 {
511     va_list adx;
512     char namebuf[MAXNAMELEN];
513     nvlist_t *nvl = fnvlist_alloc();
514
515     ASSERT(tx != NULL);
516
517     dsl_dataset_name(ds, namebuf);
518     fnvlist_add_string(nvl, ZPOOL_HIST_DSNAME, namebuf);
519     fnvlist_add_uint64(nvl, ZPOOL_HIST_DSID, ds->ds_object);

```

```

520     va_start(adx, fmt);
521     log_internal(nvl, operation, dsl_dataset_get_spa(ds), tx, fmt, adx);
522     va_end(adx);
523 }
524
525 void
526 spa_history_log_internal_dd(dsl_dir_t *dd, const char *operation,
527     dmu_tx_t *tx, const char *fmt, ...)
528 {
529     va_list adx;
530     char namebuf[MAXNAMELEN];
531     nvlist_t *nvl = fnvlist_alloc();
532
533     ASSERT(tx != NULL);
534
535     dsl_dir_name(dd, namebuf);
536     fnvlist_add_string(nvl, ZPOOL_HIST_DSNAME, namebuf);
537     fnvlist_add_uint64(nvl, ZPOOL_HIST_DSID,
538     dd->dd_phys->dd_head_dataset_obj);
539
540     va_start(adx, fmt);
541     log_internal(nvl, operation, dd->dd_pool->dp_spa, tx, fmt, adx);
542     va_end(adx);
543 }
544
545 void
546 spa_history_log_version(spa_t *spa, const char *operation)
547 spa_history_log_version(spa_t *spa, history_internal_events_t event)
548 {
549     #ifdef _KERNEL
550         uint64_t current_vers = spa_version(spa);
551
552         spa_history_log_internal(spa, operation, NULL,
553         "pool version %llu; software version %llu/%d; uts %s %s %s %s",
554         if (current_vers >= SPA_VERSION_ZPOOL_HISTORY) {
555             spa_history_log_internal(event, spa, NULL,
556             "pool spa %llu; zfs spa %llu; zpl %d; uts %s %s %s %s",
557             (u_longlong_t)current_vers, SPA_VERSION, ZPL_VERSION,
558             utsname.nodename, utsname.release, utsname.version,
559             utsname.machine);
560         }
561         cmn_err(CE_CONT, "!%s version %llu pool %s using %llu", operation,
562         }
563         cmn_err(CE_CONT, "!%s version %llu pool %s using %llu",
564         event == LOG_POOL_IMPORT ? "imported" :
565         event == LOG_POOL_CREATE ? "created" : "accessed",
566         (u_longlong_t)current_vers, spa_name(spa), SPA_VERSION);
567     #endif
568 }
569
570     _____unchanged_portion_omitted_____

```



```

122 #define DMU_OT_BYTESWAP(ot) (((ot) & DMU_OT_NEWTYPED) ? \
123 ((ot) & DMU_OT_BYTESWAP_MASK) : \
124 dmu_ot[(ot)].ot_byteswap)

126 typedef enum dmu_object_type {
127 DMU_OT_NONE,
128 /* general: */
129 DMU_OT_OBJECT_DIRECTORY, /* ZAP */
130 DMU_OT_OBJECT_ARRAY, /* UINT64 */
131 DMU_OT_PACKED_NVLIST, /* UINT8 (XDR by nvlist_pack/unpack) */
132 DMU_OT_PACKED_NVLIST_SIZE, /* UINT64 */
133 DMU_OT_BPOBJ, /* UINT64 */
134 DMU_OT_BPOBJ_HDR, /* UINT64 */
135 /* spa: */
136 DMU_OT_SPACE_MAP_HEADER, /* UINT64 */
137 DMU_OT_SPACE_MAP, /* UINT64 */
138 /* zil: */
139 DMU_OT_INTENT_LOG, /* UINT64 */
140 /* dmu: */
141 DMU_OT_DNODE, /* DNODE */
142 DMU_OT_OBJSET, /* OBJSET */
143 /* dsl: */
144 DMU_OT_DSL_DIR, /* UINT64 */
145 DMU_OT_DSL_DIR_CHILD_MAP, /* ZAP */
146 DMU_OT_DSL_DS_SNAP_MAP, /* ZAP */
147 DMU_OT_DSL_PROPS, /* ZAP */
148 DMU_OT_DSL_DATASET, /* UINT64 */
149 /* zpl: */
150 DMU_OT_ZNODE, /* ZNODE */
151 DMU_OT_OLDACL, /* Old ACL */
152 DMU_OT_PLAIN_FILE_CONTENTS, /* UINT8 */
153 DMU_OT_DIRECTORY_CONTENTS, /* ZAP */
154 DMU_OT_MASTER_NODE, /* ZAP */
155 DMU_OT_UNLINKED_SET, /* ZAP */
156 /* zvol: */
157 DMU_OT_ZVOL, /* UINT8 */
158 DMU_OT_ZVOL_PROP, /* ZAP */
159 /* other; for testing only! */
160 DMU_OT_PLAIN_OTHER, /* UINT8 */
161 DMU_OT_UINT64_OTHER, /* UINT64 */
162 DMU_OT_ZAP_OTHER, /* ZAP */
163 /* new object types: */
164 DMU_OT_ERROR_LOG, /* ZAP */
165 DMU_OT_SPA_HISTORY, /* UINT8 */
166 DMU_OT_SPA_HISTORY_OFFSETS, /* spa_his_phys_t */
167 DMU_OT_POOL_PROPS, /* ZAP */
168 DMU_OT_DSL_PERMS, /* ZAP */
169 DMU_OT_ACL, /* ACL */
170 DMU_OT_SYSACL, /* SYSACL */
171 DMU_OT_FUID, /* FUID table (Packed NVLIST UINT8) */
172 DMU_OT_FUID_SIZE, /* FUID table size UINT64 */
173 DMU_OT_NEXT_CLONES, /* ZAP */
174 DMU_OT_SCAN_QUEUE, /* ZAP */
175 DMU_OT_USERGROUP_USED, /* ZAP */
176 DMU_OT_USERGROUP_QUOTA, /* ZAP */
177 DMU_OT_USERREFS, /* ZAP */
178 DMU_OT_DDT_ZAP, /* ZAP */
179 DMU_OT_DDT_STATS, /* ZAP */
180 DMU_OT_SA, /* System attr */
181 DMU_OT_SA_MASTER_NODE, /* ZAP */
182 DMU_OT_SA_ATTR_REGISTRATION, /* ZAP */
183 DMU_OT_SA_ATTR_LAYOUTS, /* ZAP */
184 DMU_OT_SCAN_XLATE, /* ZAP */
185 DMU_OT_DEDUP, /* fake dedup BP from ddt_bp_create() */
186 DMU_OT_DEADLIST, /* ZAP */

```

```

187 DMU_OT_DEADLIST_HDR, /* UINT64 */
188 DMU_OT_DSL_CLONES, /* ZAP */
189 DMU_OT_BPOBJ_SUBOBJ, /* UINT64 */
190 /*
191 * Do not allocate new object types here. Doing so makes the on-disk
192 * format incompatible with any other format that uses the same object
193 * type number.
194 *
195 * When creating an object which does not have one of the above types
196 * use the DMU_OTN_* type with the correct byteswap and metadata
197 * values.
198 *
199 * The DMU_OTN_* types do not have entries in the dmu_ot table,
200 * use the DMU_OT_IS_METADATA() and DMU_OT_BYTESWAP() macros instead
201 * of indexing into dmu_ot directly (this works for both DMU_OT_* types
202 * and DMU_OTN_* types).
203 */
204 DMU_OT_NUMTYPES,

206 /*
207 * Names for valid types declared with DMU_OT().
208 */
209 DMU_OTN_UINT8_DATA = DMU_OT(DMU_BSWAP_UINT8, B_FALSE),
210 DMU_OTN_UINT8_METADATA = DMU_OT(DMU_BSWAP_UINT8, B_TRUE),
211 DMU_OTN_UINT16_DATA = DMU_OT(DMU_BSWAP_UINT16, B_FALSE),
212 DMU_OTN_UINT16_METADATA = DMU_OT(DMU_BSWAP_UINT16, B_TRUE),
213 DMU_OTN_UINT32_DATA = DMU_OT(DMU_BSWAP_UINT32, B_FALSE),
214 DMU_OTN_UINT32_METADATA = DMU_OT(DMU_BSWAP_UINT32, B_TRUE),
215 DMU_OTN_UINT64_DATA = DMU_OT(DMU_BSWAP_UINT64, B_FALSE),
216 DMU_OTN_UINT64_METADATA = DMU_OT(DMU_BSWAP_UINT64, B_TRUE),
217 DMU_OTN_ZAP_DATA = DMU_OT(DMU_BSWAP_ZAP, B_FALSE),
218 DMU_OTN_ZAP_METADATA = DMU_OT(DMU_BSWAP_ZAP, B_TRUE),
219 } dmu_object_type_t;

47 typedef enum dmu_objset_type {
48 DMU_OST_NONE,
49 DMU_OST_META,
50 DMU_OST_ZFS,
51 DMU_OST_ZVOL,
52 DMU_OST_OTHER, /* For testing only! */
53 DMU_OST_ANY, /* Be careful! */
54 DMU_OST_NUMTYPES
55 } dmu_objset_type_t;

221 void byteswap_uint64_array(void *buf, size_t size);
222 void byteswap_uint32_array(void *buf, size_t size);
223 void byteswap_uint16_array(void *buf, size_t size);
224 void byteswap_uint8_array(void *buf, size_t size);
225 void zap_byteswap(void *buf, size_t size);
226 void zfs_oldacl_byteswap(void *buf, size_t size);
227 void zfs_acl_byteswap(void *buf, size_t size);
228 void zfs_znode_byteswap(void *buf, size_t size);

230 #define DS_FIND_SNAPSHOTS (1<<0)
231 #define DS_FIND_CHILDREN (1<<1)

233 /*
234 * The maximum number of bytes that can be accessed as part of one
235 * operation, including metadata.
236 */
237 #define DMU_MAX_ACCESS (10<<20) /* 10MB */
238 #define DMU_MAX_DELETEBLKCNT (20480) /* ~5MB of indirect blocks */

240 #define DMU_USERUSED_OBJECT (-1ULL)
241 #define DMU_GROUPUSED_OBJECT (-2ULL)
242 #define DMU_DEADLIST_OBJECT (-3ULL)

```

```

244 /*
245  * artificial blkids for bonus buffer and spill blocks
246  */
247 #define DMU_BONUS_BLKID      (-1ULL)
248 #define DMU_SPILL_BLKID     (-2ULL)
249 /*
250  * Public routines to create, destroy, open, and close objsets.
251  */
252 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
253 int dmu_objset_own(const char *name, dmu_objset_type_t type,
254     boolean_t readonly, void *tag, objset_t **osp);
255 void dmu_objset_rele(objset_t *os, void *tag);
256 void dmu_objset_disown(objset_t *os, void *tag);
257 int dmu_objset_open_ds(struct dsl_dataset *ds, objset_t **osp);

259 int dmu_objset_evict_dbufs(objset_t *os);
260 int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
261     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
262 int dmu_objset_clone(const char *name, struct dsl_dataset *clone_origin,
263     uint64_t flags);
264 int dmu_objset_destroy(const char *name, boolean_t defer);
265 int dmu_snapshots_destroy_nvlist(struct nvlist *snaps, boolean_t defer,
266     struct nvlist *errlist);
267 int dmu_objset_snapshot(struct nvlist *snaps, struct nvlist *, struct nvlist *);
268 int dmu_objset_snapshot_one(const char *fsname, const char *snapname);
269 int dmu_objset_snapshot_tmp(const char *, const char *, int);
270 int dmu_snapshots_destroy_nvlist(struct nvlist *snaps, boolean_t defer, char *);
271 int dmu_objset_snapshot(char *fsname, char *snapname, char *tag,
272     struct nvlist *props, boolean_t recursive, boolean_t temporary, int fd);
273 int dmu_objset_rename(const char *name, const char *newname,
274     boolean_t recursive);
275 int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
276     int flags);
277 void dmu_objset_byteswap(void *buf, size_t size);

278 typedef struct dmu_buf {
279     uint64_t db_object;           /* object that this buffer is part of */
280     uint64_t db_offset;         /* byte offset in this object */
281     uint64_t db_size;           /* size of buffer in bytes */
282     void *db_data;              /* data in buffer */
283 } dmu_buf_t;
284 unchanged_portion_omitted

765 typedef void dmu_sync_cb_t(zgd_t *arg, int error);
766 int dmu_sync(struct zio *zio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd);

768 /*
769  * Find the next hole or data block in file starting at *off
770  * Return found offset in *off. Return ESRCH for end of file.
771  */
772 int dmu_offset_next(objset_t *os, uint64_t object, boolean_t hole,
773     uint64_t *off);

775 /*
776  * Initial setup and final teardown.
777  */
778 extern void dmu_init(void);
779 extern void dmu_fini(void);

781 typedef void (*dmu_traverse_cb_t)(objset_t *os, void *arg, struct blkptr *bp,
782     uint64_t object, uint64_t offset, int len);
783 void dmu_traverse_objset(objset_t *os, uint64_t txg_start,
784     dmu_traverse_cb_t cb, void *arg);

786 int dmu_send(objset_t *tosnap, objset_t *fromsnap,

```

```

620 int dmu_send(objset_t *tosnap, objset_t *fromsnap, boolean_t fromorigin,
621     int outfd, struct vnode *vp, offset_t *off);
622 int dmu_send_estimate(objset_t *tosnap, objset_t *fromsnap, uint64_t *sizep);
623     uint64_t *sizep);

790 typedef struct dmu_recv_cookie {
791     /*
792      * This structure is opaque!
793      */
794     * If logical and real are different, we are recving the stream
795     * into the "real" temporary clone, and then switching it with
796     * the "logical" target.
797     */
798     struct dsl_dataset *drc_logical_ds;
799     struct dsl_dataset *drc_real_ds;
800     struct drr_begin *drc_drrb;
801     char *drc_tosnap;
802     char *drc_top_ds;
803     boolean_t drc_newfs;
804     boolean_t drc_force;
805     struct avl_tree *drc_guid_to_ds_map;
806 } dmu_recv_cookie_t;
unchanged_portion_omitted

```

```

*****
5438 Thu Jun 28 15:57:19 2012
new/usr/src/uts/common/fs/zfs/sys/dmu_objset.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */

27 /* Portions Copyright 2010 Robert Milkowski */

29 #ifndef _SYS_DMU_OBJSET_H
30 #define _SYS_DMU_OBJSET_H

32 #include <sys/spa.h>
33 #include <sys/arc.h>
34 #include <sys/txg.h>
35 #include <sys/zfs_context.h>
36 #include <sys/dnode.h>
37 #include <sys/zio.h>
38 #include <sys/zil.h>
39 #include <sys/sa.h>

41 #ifdef __cplusplus
42 extern "C" {
43 #endif

45 extern krwlock_t os_lock;

47 struct dsl_dataset;
48 struct dmu_tx;

50 #define OBJSET_PHYS_SIZE 2048
51 #define OBJSET_OLD_PHYS_SIZE 1024

53 #define OBJSET_BUF_HAS_USERUSED(buf) \
54     (arc_buf_size(buf) > OBJSET_OLD_PHYS_SIZE)

```

```

56 #define OBJSET_FLAG_USERACCOUNTING_COMPLETE    (1ULL<<0)

58 typedef struct objset_phys {
59     dnode_phys_t os_meta_dnode;
60     zil_header_t os_zil_header;
61     uint64_t os_type;
62     uint64_t os_flags;
63     char os_pad[OBJSET_PHYS_SIZE - sizeof (dnode_phys_t)*3 -
64         sizeof (zil_header_t) - sizeof (uint64_t)*2];
65     dnode_phys_t os_userused_dnode;
66     dnode_phys_t os_groupused_dnode;
67 } objset_phys_t;

69 struct objset {
70     /* Immutable: */
71     struct dsl_dataset *os_dsl_dataset;
72     spa_t *os_spa;
73     arc_buf_t *os_phys_buf;
74     objset_phys_t *os_phys;
75     /*
76      * The following "special" dnodes have no parent and are exempt from
77      * dnode_move(), but they root their descendants in this objset using
78      * handles anyway, so that all access to dnodes from dbufs consistently
79      * uses handles.
80      */
81     dnode_handle_t os_meta_dnode;
82     dnode_handle_t os_userused_dnode;
83     dnode_handle_t os_groupused_dnode;
84     zillog_t *os_zil;

86     /* can change, under dsl_dir's locks: */
87     uint8_t os_checksum;
88     uint8_t os_compress;
89     uint8_t os_copies;
90     uint8_t os_dedup_checksum;
91     uint8_t os_dedup_verify;
92     uint8_t os_logbias;
93     uint8_t os_primary_cache;
94     uint8_t os_secondary_cache;
95     uint8_t os_sync;

97     /* no lock needed: */
98     struct dmu_tx *os_synctx; /* XXX sketchy */
99     blkptr_t *os_rootbp;
100     zil_header_t os_zil_header;
101     list_t os_synced_dnodes;
102     uint64_t os_flags;

104     /* Protected by os_obj_lock */
105     kmutex_t os_obj_lock;
106     uint64_t os_obj_next;

108     /* Protected by os_lock */
109     kmutex_t os_lock;
110     list_t os_dirty_dnodes[TXG_SIZE];
111     list_t os_free_dnodes[TXG_SIZE];
112     list_t os_dnodes;
113     list_t os_downgraded_dbufs;

115     /* stuff we store for the user */
116     kmutex_t os_user_ptr_lock;
117     void *os_user_ptr;

119     /* SA layout/attribute registration */
120     sa_os_t *os_sa;

```

```

121 };

123 #define DMU_META_OBJSET 0
124 #define DMU_META_DNODE_OBJECT 0
125 #define DMU_OBJECT_IS_SPECIAL(obj) ((int64_t)(obj) <= 0)
126 #define DMU_META_DNODE(os) ((os)->os_meta_dnode.dnh_dnode)
127 #define DMU_USERUSED_DNODE(os) ((os)->os_userused_dnode.dnh_dnode)
128 #define DMU_GROUPUSED_DNODE(os) ((os)->os_groupused_dnode.dnh_dnode)

130 #define DMU_OS_IS_L2CACHEABLE(os) \
131 ((os)->os_secondary_cache == ZFS_CACHE_ALL || \
132 (os)->os_secondary_cache == ZFS_CACHE_METADATA)

134 /* called from zpl */
135 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
136 int dmu_objset_own(const char *name, dmu_objset_type_t type,
137     boolean_t readonly, void *tag, objset_t **osp);
138 void dmu_objset_rele(objset_t *os, void *tag);
139 void dmu_objset_disown(objset_t *os, void *tag);
140 int dmu_objset_from_ds(struct dsl_dataset *ds, objset_t **osp);

23 int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
24     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
25 int dmu_objset_clone(const char *name, struct dsl_dataset *clone_origin,
26     uint64_t flags);
27 int dmu_objset_destroy(const char *name, boolean_t defer);
28 int dmu_objset_snapshot(char *fsname, char *snapname, char *tag,
29     struct nvlist *props, boolean_t recursive, boolean_t temporary, int fd);
142 void dmu_objset_stats(objset_t *os, nvlist_t *nv);
143 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);
144 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
145     uint64_t *usedobjsp, uint64_t *availobjsp);
146 uint64_t dmu_objset_fsid_guid(objset_t *os);
35 int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
36     int flags);
147 int dmu_objset_find_spa(spa_t *spa, const char *name,
148     int func(spa_t *, uint64_t, const char *, void *), void *arg, int flags);
149 int dmu_objset_prefetch(const char *name, void *arg);
40 void dmu_objset_byteswap(void *buf, size_t size);
150 int dmu_objset_evict_dbufs(objset_t *os);
151 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

153 /* called from dsl */
154 void dmu_objset_sync(objset_t *os, zio_t *zio, dmu_tx_t *tx);
155 boolean_t dmu_objset_is_dirty(objset_t *os, uint64_t txg);
156 boolean_t dmu_objset_is_dirty_anywhere(objset_t *os);
157 objset_t *dmu_objset_create_impl(spa_t *spa, struct dsl_dataset *ds,
158     blkptr_t *bp, dmu_objset_type_t type, dmu_tx_t *tx);
159 int dmu_objset_open_impl(spa_t *spa, struct dsl_dataset *ds, blkptr_t *bp,
160     objset_t **osp);
161 void dmu_objset_evict(objset_t *os);
162 void dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx);
163 void dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx);
164 boolean_t dmu_objset_userused_enabled(objset_t *os);
165 int dmu_objset_userspace_upgrade(objset_t *os);
166 boolean_t dmu_objset_userspace_present(objset_t *os);

168 void dmu_objset_init(void);
169 void dmu_objset_fini(void);

171 #ifdef __cplusplus
172 }
_____unchanged_portion_omitted_____

```

```

*****
10374 Thu Jun 28 15:57:19 2012
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
_____unchanged_portion_omitted_____

175 /*
176 * The max length of a temporary tag prefix is the number of hex digits
177 * required to express UINT64_MAX plus one for the hyphen.
178 */
179 #define MAX_TAG_PREFIX_LEN      17

181 struct dsl_ds_holdarg {
182     dsl_sync_task_group_t *dstg;
183     const char *htag;
184     char *snapname;
185     boolean_t recursive;
186     boolean_t gotone;
187     boolean_t temphold;
188     char failed[MAXPATHLEN];
189 };

191 #define dsl_dataset_is_snapshot(ds) \
192     ((ds)->ds_phys->ds_num_children != 0)

194 #define DS_UNIQUE_IS_ACCURATE(ds) \
195     (((ds)->ds_phys->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)

197 int dsl_dataset_hold(const char *name, void *tag, dsl_dataset_t **dsp);
198 int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj,
199     void *tag, dsl_dataset_t **);
200 int dsl_dataset_own(const char *name, boolean_t inconsistentok,
201     void *tag, dsl_dataset_t **dsp);
202 int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
203     boolean_t inconsistentok, void *tag, dsl_dataset_t **dsp);
204 void dsl_dataset_name(dsl_dataset_t *ds, char *name);
205 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
206 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
207 void dsl_dataset_drop_ref(dsl_dataset_t *ds, void *tag);
208 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, boolean_t inconsistentok,
209     void *tag);
210 void dsl_dataset_make_exclusive(dsl_dataset_t *ds, void *tag);
211 void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
212     minor_t minor);
213 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
214     dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
215 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
216     uint64_t flags, dmu_tx_t *tx);
217 int dsl_dataset_destroy(dsl_dataset_t *ds, void *tag, boolean_t defer);
218 int dsl_snapshots_destroy(char *fsname, char *snapname, boolean_t defer);
219 dsl_checkfunc_t dsl_dataset_destroy_check;
220 dsl_syncfunc_t dsl_dataset_destroy_sync;
221 dsl_checkfunc_t dsl_dataset_snapshot_check;
222 dsl_syncfunc_t dsl_dataset_snapshot_sync;
223 dsl_syncfunc_t dsl_dataset_user_hold_sync;
224 int dsl_dataset_snapshot_check(dsl_dataset_t *ds, const char *, dmu_tx_t *tx);
225 void dsl_dataset_snapshot_sync(dsl_dataset_t *ds, const char *, dmu_tx_t *tx);

```

```

223 #endif /* ! codereview */
224 int dsl_dataset_rename(char *name, const char *newname, boolean_t recursive);
225 int dsl_dataset_promote(const char *name, char *conflsnap);
226 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
227     boolean_t force);
228 int dsl_dataset_user_hold(char *dsname, char *snapname, char *htag,
229     boolean_t recursive, boolean_t temphold, int cleanup_fd);
230 int dsl_dataset_user_hold_for_send(dsl_dataset_t *ds, char *htag,
231     boolean_t temphold);
232 int dsl_dataset_user_release(char *dsname, char *snapname, char *htag,
233     boolean_t recursive);
234 int dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
235     char *htag, boolean_t retry);
236 int dsl_dataset_get_holds(const char *dsname, nvlist_t **nvp);

238 blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
239 void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

241 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

243 boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

245 void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);

247 void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
248     dmu_tx_t *tx);
249 int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
250     dmu_tx_t *tx, boolean_t async);
251 boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
252     uint64_t blk_birth);
253 uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

255 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
256 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
257 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
258 void dsl_dataset_space(dsl_dataset_t *ds,
259     uint64_t *refdbbytesp, uint64_t *availbytesp,
260     uint64_t *usedobjsp, uint64_t *availobjsp);
261 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
262 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
263     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
264 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
265     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);

267 int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

269 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
270     uint64_t asize, uint64_t inflight, uint64_t *used,
271     uint64_t *ref_rsrv);
272 int dsl_dataset_set_quota(const char *dsname, zprop_source_t source,
273     uint64_t quota);
274 dsl_syncfunc_t dsl_dataset_set_quota_sync;
275 int dsl_dataset_set_reservation(const char *dsname, zprop_source_t source,
276     uint64_t reservation);

278 int dsl_destroy_inconsistent(const char *dsname, void *arg);

280 #ifdef ZFS_DEBUG
281 #define dprintf_ds(ds, fmt, ...) do { \
282     if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
283         char * __ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
284         dsl_dataset_name(ds, __ds_name); \
285         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
286         kmem_free(__ds_name, MAXNAMELEN); \
287     } \
288     _NOTE(CONSTCOND) } while (0)

```

```
289 #else
290 #define dprintf_ds(dd, fmt, ...)
291 #endif
293 #ifdef __cplusplus
294 }
295 #endif
297 #endif /* _SYS_DSL_DATASET_H */
```

new/usr/src/uts/common/fs/zfs/sys/dsl_deleg.h

1

```
*****
2790 Thu Jun 28 15:57:19 2012
new/usr/src/uts/common/fs/zfs/sys/dsl_deleg.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
23 * Copyright (c) 2011 by Delphix. All rights reserved.
24 */

26 #ifndef _SYS_DSL_DELEG_H
27 #define _SYS_DSL_DELEG_H

29 #include <sys/dmu.h>
30 #include <sys/dsl_pool.h>
31 #include <sys/zfs_context.h>

33 #ifdef __cplusplus
34 extern "C" {
35 #endif

37 #define ZFS_DELEG_PERM_NONE          ""
38 #define ZFS_DELEG_PERM_CREATE       "create"
39 #define ZFS_DELEG_PERM_DESTROY      "destroy"
40 #define ZFS_DELEG_PERM_SNAPSHOT     "snapshot"
41 #define ZFS_DELEG_PERM_ROLLBACK     "rollback"
42 #define ZFS_DELEG_PERM_CLONE        "clone"
43 #define ZFS_DELEG_PERM_PROMOTE      "promote"
44 #define ZFS_DELEG_PERM_RENAME       "rename"
45 #define ZFS_DELEG_PERM_MOUNT        "mount"
46 #define ZFS_DELEG_PERM_SHARE        "share"
47 #define ZFS_DELEG_PERM_SEND         "send"
48 #define ZFS_DELEG_PERM_RECEIVE      "receive"
49 #define ZFS_DELEG_PERM_ALLOW        "allow"
50 #define ZFS_DELEG_PERM_USERPROP     "userprop"
51 #define ZFS_DELEG_PERM_VSCAN        "vscan"
52 #define ZFS_DELEG_PERM_USERQUOTA    "userquota"
53 #define ZFS_DELEG_PERM_GROUPQUOTA   "groupquota"
```

new/usr/src/uts/common/fs/zfs/sys/dsl_deleg.h

2

```
54 #define ZFS_DELEG_PERM_USERUSED     "userused"
55 #define ZFS_DELEG_PERM_GROUPUSED    "groupused"
56 #define ZFS_DELEG_PERM_HOLD         "hold"
57 #define ZFS_DELEG_PERM_RELEASE      "release"
58 #define ZFS_DELEG_PERM_DIFF         "diff"

60 /*
61  * Note: the names of properties that are marked delegatable are also
62  * valid delegated permissions
63  */

65 int dsl_deleg_get(const char *ddname, nvlist_t **nvp);
66 int dsl_deleg_set(const char *ddname, nvlist_t *nvp, boolean_t unset);
67 int dsl_deleg_access(const char *ddname, const char *perm, cred_t *cr);
68 int dsl_deleg_access_impl(struct dsl_dataset *ds, const char *perm, cred_t *cr);
68 int dsl_deleg_access_impl(struct dsl_dataset *ds, boolean_t descendent,
69     const char *perm, cred_t *cr);
69 void dsl_deleg_set_create_perms(dsl_dir_t *dd, dmu_tx_t *tx, cred_t *cr);
70 int dsl_deleg_can_allow(char *ddname, nvlist_t *nvp, cred_t *cr);
71 int dsl_deleg_can_unallow(char *ddname, nvlist_t *nvp, cred_t *cr);
72 int dsl_deleg_destroy(objset_t *os, uint64_t zapobj, dmu_tx_t *tx);
73 boolean_t dsl_delegation_on(objset_t *os);

75 #ifdef __cplusplus
76 }
_____unchanged_portion_omitted_____
```



```

*****
3907 Thu Jun 28 15:57:19 2012
new/usr/src/uts/common/fs/zfs/sys/dsl_prop.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */

27 #ifndef _SYS_DSL_PROP_H
28 #define _SYS_DSL_PROP_H

30 #include <sys/dmu.h>
31 #include <sys/dsl_pool.h>
32 #include <sys/zfs_context.h>
33 #include <sys/dsl_synctask.h>

35 #ifdef __cplusplus
36 extern "C" {
37 #endif

39 struct dsl_dataset;
40 struct dsl_dir;

42 /* The callback func may not call into the DMU or DSL! */
43 typedef void (dsl_prop_changed_cb_t)(void *arg, uint64_t newval);

45 typedef struct dsl_prop_cb_record {
46     list_node_t cbr_node; /* link on dd_prop_cbs */
47     struct dsl_dataset *cbr_ds;
48     const char *cbr_propname;
49     dsl_prop_changed_cb_t *cbr_func;
50     void *cbr_arg;
51 } dsl_prop_cb_record_t;

53 typedef struct dsl_props_arg {
54     nvlist_t *pa_props;

```

```

55     zprop_source_t pa_source;
56 } dsl_props_arg_t;

58 typedef struct dsl_prop_set_arg {
59     const char *psa_name;
60     zprop_source_t psa_source;
61     int psa_intsz;
62     int psa_numints;
63     const void *psa_value;

65     /*
66      * Used to handle the special requirements of the quota and reservation
67      * properties.
68      */
69     uint64_t psa_effective_value;
70 } dsl_prop_setarg_t;

72 int dsl_prop_register(struct dsl_dataset *ds, const char *propname,
73     dsl_prop_changed_cb_t *callback, void *cbarg);
74 int dsl_prop_unregister(struct dsl_dataset *ds, const char *propname,
75     dsl_prop_changed_cb_t *callback, void *cbarg);
76 int dsl_prop_numcb(struct dsl_dataset *ds);

78 int dsl_prop_get(const char *ddname, const char *propname,
79     int intsz, int numints, void *buf, char *setpoint);
80 int dsl_prop_get_integer(const char *ddname, const char *propname,
81     uint64_t *valuep, char *setpoint);
82 int dsl_prop_get_all(objset_t *os, nvlist_t **nvp);
83 int dsl_prop_get_received(objset_t *os, nvlist_t **nvp);
84 int dsl_prop_get_ds(struct dsl_dataset *ds, const char *propname,
85     int intsz, int numints, void *buf, char *setpoint);
86 int dsl_prop_get_dd(struct dsl_dir *dd, const char *propname,
87     int intsz, int numints, void *buf, char *setpoint,
88     boolean_t snapshot);

90 dsl_syncfunc_t dsl_props_set_sync;
91 int dsl_prop_set(const char *ddname, const char *propname,
92     zprop_source_t source, int intsz, int numints, const void *buf);
93 int dsl_props_set(const char *dsname, zprop_source_t source, nvlist_t *nvl);
94 void dsl_dir_prop_set_uint64_sync(dsl_dir_t *dd, const char *name, uint64_t val,
95     dmu_tx_t *tx);

97 void dsl_prop_setarg_init_uint64(dsl_prop_setarg_t *psa, const char *propname,
98     zprop_source_t source, uint64_t *value);
99 int dsl_prop_predict_sync(dsl_dir_t *dd, dsl_prop_setarg_t *psa);
100 #ifdef ZFS_DEBUG
101 #define DSL_PROP_CHECK_PREDICTION(dd, psa) \
102     dsl_prop_check_prediction(dd, (psa))
103 #else
104 #define DSL_PROP_CHECK_PREDICTION(dd, psa) /* nothing */
105 #endif

107 /* flag first receive on or after SPA_VERSION_RECVD_PROPS */
108 boolean_t dsl_prop_get_hasrecvd(objset_t *os);
109 void dsl_prop_set_hasrecvd(objset_t *os);
110 void dsl_prop_unset_hasrecvd(objset_t *os);

111 void dsl_prop_nvlist_add_uint64(nvlist_t *nv, zfs_prop_t prop, uint64_t value);
112 void dsl_prop_nvlist_add_string(nvlist_t *nv,
113     zfs_prop_t prop, const char *value);

115 #ifdef __cplusplus
116 }

```

unchanged_portion_omitted_

new/usr/src/uts/common/fs/zfs/sys/rrwlock.h

1

```
*****
2466 Thu Jun 28 15:57:20 2012
new/usr/src/uts/common/fs/zfs/sys/rrwlock.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2007 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 /*
26 * Copyright (c) 2012 by Delphix. All rights reserved.
27 */
28 #endif /* ! codereview */
29
30 #ifndef _SYS_RR_RW_LOCK_H
31 #define _SYS_RR_RW_LOCK_H
32
33 #pragma ident "%Z%%M% %I% %E% SMI"
34
35 #ifdef __cplusplus
36 extern "C" {
37 #endif
38 #include <sys/inttypes.h>
39 #include <sys/zfs_context.h>
40 #include <sys/refcount.h>
41
42 /*
43 * A reader-writer lock implementation that allows re-entrant reads, but
44 * still gives writers priority on "new" reads.
45 * See rrwlock.c for more details about the implementation.
46 *
47 * Fields of the rrwlock_t structure:
48 * - rr_lock: protects modification and reading of rrwlock_t fields
49 * - rr_cv: cv for waking up readers or waiting writers
50 * - rr_writer: thread id of the current writer
51 * - rr_anon_rcount: number of active anonymous readers
52 * - rr_linked_rcount: total number of non-anonymous active readers

```

new/usr/src/uts/common/fs/zfs/sys/rrwlock.h

2

```
53 * - rr_writer_wanted: a writer wants the lock
54 */
55 typedef struct rrwlock {
56     kmutex_t      rr_lock;
57     kcondvar_t    rr_cv;
58     kthread_t     *rr_writer;
59     refcount_t    rr_anon_rcount;
60     refcount_t    rr_linked_rcount;
61     boolean_t     rr_writer_wanted;
62 } rrwlock_t;
63
64 /*
65 * 'tag' is used in reference counting tracking. The
66 * 'tag' must be the same in a rrw_enter() as in its
67 * corresponding rrw_exit().
68 */
69 void rrw_init(rrwlock_t *rrl);
70 void rrw_destroy(rrwlock_t *rrl);
71 void rrw_enter(rrwlock_t *rrl, krw_t rw, void *tag);
72 void rrw_exit(rrwlock_t *rrl, void *tag);
73 boolean_t rrw_held(rrwlock_t *rrl, krw_t rw);
74 void rrw_tsd_destroy(void *arg);
75 #endif /* ! codereview */
76
77 #define RRW_READ_HELD(x)      rrw_held(x, RW_READER)
78 #define RRW_WRITE_HELD(x)    rrw_held(x, RW_WRITER)
79
80 #ifdef __cplusplus
81 }
82 #endif
83
84 #endif /* _SYS_RR_RW_LOCK_H */

```

```

*****
25879 Thu Jun 28 15:57:20 2012
new/usr/src/uts/common/fs/zfs/sys/spa.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25 */
27 #ifndef _SYS_SPA_H
28 #define _SYS_SPA_H
29
30 #include <sys/avl.h>
31 #include <sys/zfs_context.h>
32 #include <sys/nvpair.h>
33 #include <sys/sysmacros.h>
34 #include <sys/types.h>
35 #include <sys/fs/zfs.h>
36
37 #ifdef __cplusplus
38 extern "C" {
39 #endif
40
41 /*
42  * Forward references that lots of things need.
43  */
44 typedef struct spa spa_t;
45 typedef struct vdev vdev_t;
46 typedef struct metaslab metaslab_t;
47 typedef struct metaslab_group metaslab_group_t;
48 typedef struct metaslab_class metaslab_class_t;
49 typedef struct zio zio_t;
50 typedef struct zillog zillog_t;
51 typedef struct spa_aux_vdev spa_aux_vdev_t;
52 typedef struct ddt ddt_t;
53 typedef struct ddt_entry ddt_entry_t;
54 struct dsl_pool;

```

```

55 struct dsl_dataset;
56 #endif /* !codereview */
57
58 /*
59  * General-purpose 32-bit and 64-bit bitfield encodings.
60  */
61 #define BF32_DECODE(x, low, len)      P2PHASE((x) >> (low), 1U << (len))
62 #define BF64_DECODE(x, low, len)      P2PHASE((x) >> (low), 1ULL << (len))
63 #define BF32_ENCODE(x, low, len)      (P2PHASE((x), 1U << (len)) << (low))
64 #define BF64_ENCODE(x, low, len)      (P2PHASE((x), 1ULL << (len)) << (low))
65
66 #define BF32_GET(x, low, len)          BF32_DECODE(x, low, len)
67 #define BF64_GET(x, low, len)          BF64_DECODE(x, low, len)
68
69 #define BF32_SET(x, low, len, val)     \
70     ((x) ^= BF32_ENCODE((x) >> low) ^ (val), low, len)
71 #define BF64_SET(x, low, len, val)     \
72     ((x) ^= BF64_ENCODE((x) >> low) ^ (val), low, len)
73
74 #define BF32_GET_SB(x, low, len, shift, bias) \
75     ((BF32_GET(x, low, len) + (bias)) << (shift))
76 #define BF64_GET_SB(x, low, len, shift, bias) \
77     ((BF64_GET(x, low, len) + (bias)) << (shift))
78
79 #define BF32_SET_SB(x, low, len, shift, bias, val) \
80     BF32_SET(x, low, len, ((val) >> (shift)) - (bias))
81 #define BF64_SET_SB(x, low, len, shift, bias, val) \
82     BF64_SET(x, low, len, ((val) >> (shift)) - (bias))
83
84 /*
85  * We currently support nine block sizes, from 512 bytes to 128K.
86  * We could go higher, but the benefits are near-zero and the cost
87  * of COWing a giant block to modify one byte would become excessive.
88  */
89 #define SPA_MINBLOCKSHIFT      9
90 #define SPA_MAXBLOCKSHIFT      17
91 #define SPA_MINBLOCKSIZE      (1ULL << SPA_MINBLOCKSHIFT)
92 #define SPA_MAXBLOCKSIZE      (1ULL << SPA_MAXBLOCKSHIFT)
93
94 #define SPA_BLOCKSIZE          (SPA_MAXBLOCKSHIFT - SPA_MINBLOCKSHIFT + 1)
95
96 /*
97  * Size of block to hold the configuration data (a packed nvlist)
98  */
99 #define SPA_CONFIG_BLOCKSIZE  (1ULL << 14)
100
101 /*
102  * The DVA size encodings for LSIZE and PSIZE support blocks up to 32MB.
103  * The ASIZE encoding should be at least 64 times larger (6 more bits)
104  * to support up to 4-way RAID-Z mirror mode with worst-case gang block
105  * overhead, three DVAs per bp, plus one more bit in case we do anything
106  * else that expands the ASIZE.
107  */
108 #define SPA_LSIZEBITS          16      /* LSIZE up to 32M (2^16 * 512) */
109 #define SPA_PSIZEBITS          16      /* PSIZE up to 32M (2^16 * 512) */
110 #define SPA_ASIZEBITS          24      /* ASIZE up to 64 times larger */
111
112 /*
113  * All SPA data is represented by 128-bit data virtual addresses (DVAs).
114  * The members of the dva_t should be considered opaque outside the SPA.
115  */
116 typedef struct dva {
117     uint64_t          dva_word[2];
118 } dva_t;
119
120 /*

```

```

121 * Each block has a 256-bit checksum -- strong enough for cryptographic hashes.
122 */
123 typedef struct zio_cksum {
124     uint64_t          zc_word[4];
125 } zio_cksum_t;

127 /*
128 * Each block is described by its DVAs, time of birth, checksum, etc.
129 * The word-by-word, bit-by-bit layout of the blkptr is as follows:
130 *
131 *      64      56      48      40      32      24      16      8      0
132 *      +-----+-----+-----+-----+-----+-----+-----+-----+
133 * 0      |          vdev1          | GRID |          ASIZE          |
134 *      +-----+-----+-----+-----+-----+-----+-----+
135 * 1      | G |          offset1          |
136 *      +-----+-----+-----+-----+-----+-----+-----+
137 * 2      |          vdev2          | GRID |          ASIZE          |
138 *      +-----+-----+-----+-----+-----+-----+-----+
139 * 3      | G |          offset2          |
140 *      +-----+-----+-----+-----+-----+-----+-----+
141 * 4      |          vdev3          | GRID |          ASIZE          |
142 *      +-----+-----+-----+-----+-----+-----+-----+
143 * 5      | G |          offset3          |
144 *      +-----+-----+-----+-----+-----+-----+-----+
145 * 6      | BDx|lvl| type | cksum | comp |          PSIZE          |          LSIZE          |
146 *      +-----+-----+-----+-----+-----+-----+-----+-----+
147 * 7      |          padding          |
148 *      +-----+-----+-----+-----+-----+-----+-----+
149 * 8      |          padding          |
150 *      +-----+-----+-----+-----+-----+-----+-----+
151 * 9      |          physical birth txg          |
152 *      +-----+-----+-----+-----+-----+-----+-----+
153 * a      |          logical birth txg          |
154 *      +-----+-----+-----+-----+-----+-----+-----+
155 * b      |          fill count          |
156 *      +-----+-----+-----+-----+-----+-----+-----+
157 * c      |          checksum[0]          |
158 *      +-----+-----+-----+-----+-----+-----+-----+
159 * d      |          checksum[1]          |
160 *      +-----+-----+-----+-----+-----+-----+-----+
161 * e      |          checksum[2]          |
162 *      +-----+-----+-----+-----+-----+-----+-----+
163 * f      |          checksum[3]          |
164 *      +-----+-----+-----+-----+-----+-----+-----+
165 *
166 * Legend:
167 *
168 * vdev      virtual device ID
169 * offset    offset into virtual device
170 * LSIZE     logical size
171 * PSIZE     physical size (after compression)
172 * ASIZE     allocated size (including RAID-Z parity and gang block headers)
173 * GRID      RAID-Z layout information (reserved for future use)
174 * cksum     checksum function
175 * comp      compression function
176 * G         gang block indicator
177 * B         byteorder (endianness)
178 * D         dedup
179 * X         unused
180 * lvl       level of indirection
181 * type      DMU object type
182 * phys birth txg of block allocation; zero if same as logical birth txg
183 * log. birth transaction group in which the block was logically born
184 * fill count number of non-zero blocks under this bp
185 * checksum[4] 256-bit checksum of the data this bp describes
186 */

```

```

187 #define SPA_BLKPTRSHIFT 7 /* blkptr_t is 128 bytes */
188 #define SPA_DVAS_PER_BP 3 /* Number of DVAs in a bp */

190 typedef struct blkptr {
191     dva_t          blk_dva[SPA_DVAS_PER_BP]; /* Data Virtual Addresses */
192     uint64_t       blk_prop; /* size, compression, type, etc */
193     blk_pad[2]; /* Extra space for the future */
194     uint64_t       blk_phys_birth; /* txg when block was allocated */
195     uint64_t       blk_birth; /* transaction group at birth */
196     uint64_t       blk_fill; /* fill count */
197     zio_cksum_t    blk_cksum; /* 256-bit checksum */
198 } blkptr_t;

200 /*
201 * Macros to get and set fields in a bp or DVA.
202 */
203 #define DVA_GET_ASIZE(dva) \
204     BF64_GET_SB((dva)->dva_word[0], 0, 24, SPA_MINBLOCKSHIFT, 0)
205 #define DVA_SET_ASIZE(dva, x) \
206     BF64_SET_SB((dva)->dva_word[0], 0, 24, SPA_MINBLOCKSHIFT, 0, x)

208 #define DVA_GET_GRID(dva) BF64_GET((dva)->dva_word[0], 24, 8)
209 #define DVA_SET_GRID(dva, x) BF64_SET((dva)->dva_word[0], 24, 8, x)

211 #define DVA_GET_VDEV(dva) BF64_GET((dva)->dva_word[0], 32, 32)
212 #define DVA_SET_VDEV(dva, x) BF64_SET((dva)->dva_word[0], 32, 32, x)

214 #define DVA_GET_OFFSET(dva) \
215     BF64_GET_SB((dva)->dva_word[1], 0, 63, SPA_MINBLOCKSHIFT, 0)
216 #define DVA_SET_OFFSET(dva, x) \
217     BF64_SET_SB((dva)->dva_word[1], 0, 63, SPA_MINBLOCKSHIFT, 0, x)

219 #define DVA_GET_GANG(dva) BF64_GET((dva)->dva_word[1], 63, 1)
220 #define DVA_SET_GANG(dva, x) BF64_SET((dva)->dva_word[1], 63, 1, x)

222 #define BP_GET_LSIZE(bp) \
223     BF64_GET_SB((bp)->blk_prop, 0, 16, SPA_MINBLOCKSHIFT, 1)
224 #define BP_SET_LSIZE(bp, x) \
225     BF64_SET_SB((bp)->blk_prop, 0, 16, SPA_MINBLOCKSHIFT, 1, x)

227 #define BP_GET_PSIZE(bp) \
228     BF64_GET_SB((bp)->blk_prop, 16, 16, SPA_MINBLOCKSHIFT, 1)
229 #define BP_SET_PSIZE(bp, x) \
230     BF64_SET_SB((bp)->blk_prop, 16, 16, SPA_MINBLOCKSHIFT, 1, x)

232 #define BP_GET_COMPRESS(bp) BF64_GET((bp)->blk_prop, 32, 8)
233 #define BP_SET_COMPRESS(bp, x) BF64_SET((bp)->blk_prop, 32, 8, x)

235 #define BP_GET_CHECKSUM(bp) BF64_GET((bp)->blk_prop, 40, 8)
236 #define BP_SET_CHECKSUM(bp, x) BF64_SET((bp)->blk_prop, 40, 8, x)

238 #define BP_GET_TYPE(bp) BF64_GET((bp)->blk_prop, 48, 8)
239 #define BP_SET_TYPE(bp, x) BF64_SET((bp)->blk_prop, 48, 8, x)

241 #define BP_GET_LEVEL(bp) BF64_GET((bp)->blk_prop, 56, 5)
242 #define BP_SET_LEVEL(bp, x) BF64_SET((bp)->blk_prop, 56, 5, x)

244 #define BP_GET_PROP_BIT_61(bp) BF64_GET((bp)->blk_prop, 61, 1)
245 #define BP_SET_PROP_BIT_61(bp, x) BF64_SET((bp)->blk_prop, 61, 1, x)

247 #define BP_GET_DEDUP(bp) BF64_GET((bp)->blk_prop, 62, 1)
248 #define BP_SET_DEDUP(bp, x) BF64_SET((bp)->blk_prop, 62, 1, x)

250 #define BP_GET_BYTEORDER(bp) (0 - BF64_GET((bp)->blk_prop, 63, 1))
251 #define BP_SET_BYTEORDER(bp, x) BF64_SET((bp)->blk_prop, 63, 1, x)

```

```

253 #define BP_PHYSICAL_BIRTH(bp)      \
254     ((bp)->blk_phys_birth ? (bp)->blk_phys_birth : (bp)->blk_birth)

256 #define BP_SET_BIRTH(bp, logical, physical)    \
257 {                                               \
258     (bp)->blk_birth = (logical);               \
259     (bp)->blk_phys_birth = ((logical) == (physical) ? 0 : (physical)); \
260 }

262 #define BP_GET_ASIZE(bp) \
263     (DVA_GET_ASIZE(&(amp)bp->blk_dva[0]) + DVA_GET_ASIZE(&(amp)bp->blk_dva[1]) + \
264     DVA_GET_ASIZE(&(amp)bp->blk_dva[2]))

266 #define BP_GET_UCSIZE(bp) \
267     ((BP_GET_LEVEL(bp) > 0 || DMU_OT_IS_METADATA(BP_GET_TYPE(bp))) ? \
268     BP_GET_PSIZE(bp) : BP_GET_LSIZE(bp))

270 #define BP_GET_NDVAS(bp) \
271     (!DVA_GET_ASIZE(&(amp)bp->blk_dva[0]) + \
272     !!DVA_GET_ASIZE(&(amp)bp->blk_dva[1]) + \
273     !!DVA_GET_ASIZE(&(amp)bp->blk_dva[2]))

275 #define BP_COUNT_GANG(bp) \
276     (DVA_GET_GANG(&(amp)bp->blk_dva[0]) + \
277     DVA_GET_GANG(&(amp)bp->blk_dva[1]) + \
278     DVA_GET_GANG(&(amp)bp->blk_dva[2]))

280 #define DVA_EQUAL(dva1, dva2) \
281     ((dva1)->dva_word[1] == (dva2)->dva_word[1] && \
282     (dva1)->dva_word[0] == (dva2)->dva_word[0])

284 #define BP_EQUAL(bp1, bp2) \
285     (BP_PHYSICAL_BIRTH(bp1) == BP_PHYSICAL_BIRTH(bp2) && \
286     DVA_EQUAL(&(amp)bp1->blk_dva[0], &(amp)bp2->blk_dva[0]) && \
287     DVA_EQUAL(&(amp)bp1->blk_dva[1], &(amp)bp2->blk_dva[1]) && \
288     DVA_EQUAL(&(amp)bp1->blk_dva[2], &(amp)bp2->blk_dva[2]))

290 #define ZIO_CHECKSUM_EQUAL(zc1, zc2) \
291     (0 == (((zc1).zc_word[0] - (zc2).zc_word[0]) | \
292     ((zc1).zc_word[1] - (zc2).zc_word[1]) | \
293     ((zc1).zc_word[2] - (zc2).zc_word[2]) | \
294     ((zc1).zc_word[3] - (zc2).zc_word[3])))

296 #define DVA_IS_VALID(dva)      (DVA_GET_ASIZE(dva) != 0)

298 #define ZIO_SET_CHECKSUM(zcp, w0, w1, w2, w3) \
299 { \
300     (zcp)->zc_word[0] = w0; \
301     (zcp)->zc_word[1] = w1; \
302     (zcp)->zc_word[2] = w2; \
303     (zcp)->zc_word[3] = w3; \
304 }

306 #define BP_IDENTITY(bp)      (&(bp)->blk_dva[0])
307 #define BP_IS_GANG(bp)      DVA_GET_GANG(BP_IDENTITY(bp))
308 #define BP_IS_HOLE(bp)      ((bp)->blk_birth == 0)

310 /* BP_IS_RAIDZ(bp) assumes no block compression */
311 #define BP_IS_RAIDZ(bp)      (DVA_GET_ASIZE(&(amp)bp->blk_dva[0]) > \
312     BP_GET_PSIZE(bp))

314 #define BP_ZERO(bp) \
315 { \
316     (bp)->blk_dva[0].dva_word[0] = 0; \
317     (bp)->blk_dva[0].dva_word[1] = 0; \
318     (bp)->blk_dva[1].dva_word[0] = 0; \

```

```

319     (bp)->blk_dva[1].dva_word[1] = 0; \
320     (bp)->blk_dva[2].dva_word[0] = 0; \
321     (bp)->blk_dva[2].dva_word[1] = 0; \
322     (bp)->blk_prop = 0; \
323     (bp)->blk_pad[0] = 0; \
324     (bp)->blk_pad[1] = 0; \
325     (bp)->blk_phys_birth = 0; \
326     (bp)->blk_birth = 0; \
327     (bp)->blk_fill = 0; \
328     ZIO_SET_CHECKSUM(&(amp)bp->blk_cksum, 0, 0, 0, 0); \
329 }

331 /*
332  * Note: the byteorder is either 0 or -1, both of which are palindromes.
333  * This simplifies the endianness handling a bit.
334  */
335 #ifdef _BIG_ENDIAN
336 #define ZFS_HOST_BYTEORDER      (0ULL)
337 #else
338 #define ZFS_HOST_BYTEORDER      (-1ULL)
339 #endif

341 #define BP_SHOULD_BYTESWAP(bp)  (BP_GET_BYTEORDER(bp) != ZFS_HOST_BYTEORDER)

343 #define BP_SPRINTF_LEN      320

345 /*
346  * This macro allows code sharing between zfs, libzpool, and mdb.
347  * 'func' is either snprintf() or mdb_snprintf().
348  * 'ws' (whitespace) can be ' ' for single-line format, '\n' for multi-line.
349  */
350 #define SPRINTF_BLKPTR(func, ws, buf, bp, type, checksum, compress) \
351 { \
352     static const char *copyname[] = \
353     { "zero", "single", "double", "triple" }; \
354     int size = BP_SPRINTF_LEN; \
355     int len = 0; \
356     int copies = 0; \
357 \
358     if (bp == NULL) { \
359         len = func(buf + len, size - len, "<NULL>"); \
360     } else if (BP_IS_HOLE(bp)) { \
361         len = func(buf + len, size - len, "<hole>"); \
362     } else { \
363         for (int d = 0; d < BP_GET_NDVAS(bp); d++) { \
364             const dva_t *dva = &bp->blk_dva[d]; \
365             if (DVA_IS_VALID(dva)) \
366                 copies++; \
367             len += func(buf + len, size - len, \
368                 "DVA[%d]=<llu:%llx:%llx>%c", d, \
369                 (u_longlong_t)DVA_GET_VDEF(dva), \
370                 (u_longlong_t)DVA_GET_OFFSET(dva), \
371                 (u_longlong_t)DVA_GET_ASIZE(dva), \
372                 ws); \
373         } \
374         if (BP_IS_GANG(bp) && \
375             DVA_GET_ASIZE(&bp->blk_dva[2]) <= \
376             DVA_GET_ASIZE(&bp->blk_dva[1]) / 2) \
377             copies--; \
378         len += func(buf + len, size - len, \
379             "[L%llu %s] %s %s %s %s %s %s%c", \
380             "size=%llxL/%llxP birth=%lluL/%lluP fill=%llu%c", \
381             "cksum=%llx:%llx:%llx:%llx", \
382             (u_longlong_t)BP_GET_LEVEL(bp), \
383             type, \
384             checksum, \

```

```

385     compress, \
386     BP_GET_BYTEORDER(bp) == 0 ? "BE" : "LE", \
387     BP_IS_GANG(bp) ? "gang" : "contiguous", \
388     BP_GET_DEDUP(bp) ? "dedup" : "unique", \
389     copyname[copies], \
390     ws, \
391     (u_longlong_t)BP_GET_LSIZE(bp), \
392     (u_longlong_t)BP_GET_PSIZE(bp), \
393     (u_longlong_t)bp->blk_birth, \
394     (u_longlong_t)BP_PHYSICAL_BIRTH(bp), \
395     (u_longlong_t)bp->blk_fill, \
396     ws, \
397     (u_longlong_t)bp->blk_cksum.zc_word[0], \
398     (u_longlong_t)bp->blk_cksum.zc_word[1], \
399     (u_longlong_t)bp->blk_cksum.zc_word[2], \
400     (u_longlong_t)bp->blk_cksum.zc_word[3]); \
401     } \
402     ASSERT(len < size); \
403 }

405 #include <sys/dmu.h>

407 #define BP_GET_BUFC_TYPE(bp) \
408     (((BP_GET_LEVEL(bp) > 0) || (DMU_OT_IS_METADATA(BP_GET_TYPE(bp)))) ? \
409     ARC_BUFC_METADATA : ARC_BUFC_DATA)

411 typedef enum spa_import_type {
412     SPA_IMPORT_EXISTING,
413     SPA_IMPORT_ASSEMBLE
414 } spa_import_type_t;

416 /* state manipulation functions */
417 extern int spa_open(const char *pool, spa_t **, void *tag);
418 extern int spa_open_rewind(const char *pool, spa_t **, void *tag,
419     nvlist_t *policy, nvlist_t **config);
420 extern int spa_get_stats(const char *pool, nvlist_t **config, char *altroot,
421     size_t buflen);
422 extern int spa_create(const char *pool, nvlist_t *config, nvlist_t *props,
423     nvlist_t *zplprops);
424     const char *history_str, nvlist_t *zplprops);
424 extern int spa_import_rootpool(char *devpath, char *devid);
425 extern int spa_import(const char *pool, nvlist_t *config, nvlist_t *props,
426     uint64_t flags);
427 extern nvlist_t *spa_tryimport(nvlist_t *tryconfig);
428 extern int spa_destroy(char *pool);
429 extern int spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
430     boolean_t hardforce);
431 extern int spa_reset(char *pool);
432 extern void spa_async_request(spa_t *spa, int flag);
433 extern void spa_async_unrequest(spa_t *spa, int flag);
434 extern void spa_async_suspend(spa_t *spa);
435 extern void spa_async_resume(spa_t *spa);
436 extern spa_t *spa_inject_addrf(char *pool);
437 extern void spa_inject_delref(spa_t *spa);
438 extern void spa_scan_stat_init(spa_t *spa);
439 extern int spa_scan_get_stats(spa_t *spa, pool_scan_stat_t *ps);

441 #define SPA_ASYNC_CONFIG_UPDATE 0x01
442 #define SPA_ASYNC_REMOVE 0x02
443 #define SPA_ASYNC_PROBE 0x04
444 #define SPA_ASYNC_RESILVER_DONE 0x08
445 #define SPA_ASYNC_RESILVER 0x10
446 #define SPA_ASYNC_AUTOEXPAND 0x20
447 #define SPA_ASYNC_REMOVE_DONE 0x40
448 #define SPA_ASYNC_REMOVE_STOP 0x80

```

```

450 /*
451  * Controls the behavior of spa_vdev_remove().
452  */
453 #define SPA_REMOVE_UNSPARE 0x01
454 #define SPA_REMOVE_DONE 0x02

456 /* device manipulation */
457 extern int spa_vdev_add(spa_t *spa, nvlist_t *nvroot);
458 extern int spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot,
459     int replacing);
460 extern int spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid,
461     int replace_done);
462 extern int spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare);
463 extern boolean_t spa_vdev_remove_active(spa_t *spa);
464 extern int spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath);
465 extern int spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru);
466 extern int spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
467     nvlist_t *props, boolean_t exp);

469 /* spare state (which is global across all pools) */
470 extern void spa_spare_add(vdev_t *vd);
471 extern void spa_spare_remove(vdev_t *vd);
472 extern boolean_t spa_spare_exists(uint64_t guid, uint64_t *pool, int *refcnt);
473 extern void spa_spare_activate(vdev_t *vd);

475 /* L2ARC state (which is global across all pools) */
476 extern void spa_l2cache_add(vdev_t *vd);
477 extern void spa_l2cache_remove(vdev_t *vd);
478 extern boolean_t spa_l2cache_exists(uint64_t guid, uint64_t *pool);
479 extern void spa_l2cache_activate(vdev_t *vd);
480 extern void spa_l2cache_drop(spa_t *spa);

482 /* scanning */
483 extern int spa_scan(spa_t *spa, pool_scan_func_t func);
484 extern int spa_scan_stop(spa_t *spa);

486 /* spa syncing */
487 extern void spa_sync(spa_t *spa, uint64_t txg); /* only for DMU use */
488 extern void spa_sync_allpools(void);

490 /*
491  * DEFERRED_FREE must be large enough that regular blocks are not
492  * deferred. XXX so can't we change it back to 1?
493  */
494 #define SYNC_PASS_DEFERRED_FREE 2 /* defer frees after this pass */
495 #define SYNC_PASS_DONT_COMPRESS 4 /* don't compress after this pass */
496 #define SYNC_PASS_REWRITE 1 /* rewrite new bps after this pass */

498 /* spa namespace global mutex */
499 extern kmutex_t spa_namespace_lock;

501 /*
502  * SPA configuration functions in spa_config.c
503  */

505 #define SPA_CONFIG_UPDATE_POOL 0
506 #define SPA_CONFIG_UPDATE_VDEVS 1

508 extern void spa_config_sync(spa_t *, boolean_t, boolean_t);
509 extern void spa_config_load(void);
510 extern nvlist_t *spa_all_configs(uint64_t *);
511 extern void spa_config_set(spa_t *spa, nvlist_t *config);
512 extern nvlist_t *spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg,
513     int getstats);
514 extern void spa_config_update(spa_t *spa, int what);

```

```

516 /*
517  * Miscellaneous SPA routines in spa_misc.c
518  */

520 /* Namespace manipulation */
521 extern spa_t *spa_lookup(const char *name);
522 extern spa_t *spa_add(const char *name, nvlist_t *config, const char *altroot);
523 extern void spa_remove(spa_t *spa);
524 extern spa_t *spa_next(spa_t *prev);

526 /* Refcount functions */
527 extern void spa_open_ref(spa_t *spa, void *tag);
528 extern void spa_close(spa_t *spa, void *tag);
529 extern boolean_t spa_refcount_zero(spa_t *spa);

531 #define SCL_NONE        0x00
532 #define SCL_CONFIG     0x01
533 #define SCL_STATE      0x02
534 #define SCL_L2ARC     0x04          /* hack until L2ARC 2.0 */
535 #define SCL_ALLOC     0x08
536 #define SCL_ZIO       0x10
537 #define SCL_FREE      0x20
538 #define SCL_VDEV     0x40
539 #define SCL_LOCKS    7
540 #define SCL_ALL      ((1 << SCL_LOCKS) - 1)
541 #define SCL_STATE_ALL (SCL_STATE | SCL_L2ARC | SCL_ZIO)

543 /* Pool configuration locks */
544 extern int spa_config_tryenter(spa_t *spa, int locks, void *tag, krw_t rw);
545 extern void spa_config_enter(spa_t *spa, int locks, void *tag, krw_t rw);
546 extern void spa_config_exit(spa_t *spa, int locks, void *tag);
547 extern int spa_config_held(spa_t *spa, int locks, krw_t rw);

549 /* Pool vdev add/remove lock */
550 extern uint64_t spa_vdev_enter(spa_t *spa);
551 extern uint64_t spa_vdev_config_enter(spa_t *spa);
552 extern void spa_vdev_config_exit(spa_t *spa, vdev_t *vd, uint64_t txg,
553     int error, char *tag);
554 extern int spa_vdev_exit(spa_t *spa, vdev_t *vd, uint64_t txg, int error);

556 /* Pool vdev state change lock */
557 extern void spa_vdev_state_enter(spa_t *spa, int oplock);
558 extern int spa_vdev_state_exit(spa_t *spa, vdev_t *vd, int error);

560 /* Log state */
561 typedef enum spa_log_state {
562     SPA_LOG_UNKNOWN = 0,      /* unknown log state */
563     SPA_LOG_MISSING,         /* missing log(s) */
564     SPA_LOG_CLEAR,           /* clear the log(s) */
565     SPA_LOG_GOOD,            /* log(s) are good */
566 } spa_log_state_t;

568 extern spa_log_state_t spa_get_log_state(spa_t *spa);
569 extern void spa_set_log_state(spa_t *spa, spa_log_state_t state);
570 extern int spa_offline_log(spa_t *spa);

572 /* Log claim callback */
573 extern void spa_claim_notify(zio_t *zio);

575 /* Accessor functions */
576 extern boolean_t spa_shutting_down(spa_t *spa);
577 extern struct dsl_pool *spa_get_dsl(spa_t *spa);
578 extern boolean_t spa_is_initializing(spa_t *spa);
579 extern blkptr_t *spa_get_rootblkptr(spa_t *spa);
580 extern void spa_set_rootblkptr(spa_t *spa, const blkptr_t *bp);
581 extern void spa_altroot(spa_t *, char *, size_t);

```

```

582 extern int spa_sync_pass(spa_t *spa);
583 extern char *spa_name(spa_t *spa);
584 extern uint64_t spa_guid(spa_t *spa);
585 extern uint64_t spa_load_guid(spa_t *spa);
586 extern uint64_t spa_last_synced_txg(spa_t *spa);
587 extern uint64_t spa_first_txg(spa_t *spa);
588 extern uint64_t spa_syncing_txg(spa_t *spa);
589 extern uint64_t spa_version(spa_t *spa);
590 extern pool_state_t spa_state(spa_t *spa);
591 extern spa_load_state_t spa_load_state(spa_t *spa);
592 extern uint64_t spa_freeze_txg(spa_t *spa);
593 extern uint64_t spa_get_asize(spa_t *spa, uint64_t lsize);
594 extern uint64_t spa_get_dspace(spa_t *spa);
595 extern void spa_update_dspace(spa_t *spa);
596 extern uint64_t spa_version(spa_t *spa);
597 extern boolean_t spa_deflate(spa_t *spa);
598 extern metaslab_class_t *spa_normal_class(spa_t *spa);
599 extern metaslab_class_t *spa_log_class(spa_t *spa);
600 extern int spa_max_replication(spa_t *spa);
601 extern int spa_prev_software_version(spa_t *spa);
602 extern int spa_busy(void);
603 extern uint8_t spa_get_failmode(spa_t *spa);
604 extern boolean_t spa_suspended(spa_t *spa);
605 extern uint64_t spa_bootfs(spa_t *spa);
606 extern uint64_t spa_delegation(spa_t *spa);
607 extern objset_t *spa_meta_objset(spa_t *spa);

609 /* Miscellaneous support routines */
610 extern void spa_activate_mos_feature(spa_t *spa, const char *feature);
611 extern void spa_deactivate_mos_feature(spa_t *spa, const char *feature);
612 extern int spa_rename(const char *oldname, const char *newname);
613 extern spa_t *spa_by_guid(uint64_t pool_guid, uint64_t device_guid);
614 extern boolean_t spa_guid_exists(uint64_t pool_guid, uint64_t device_guid);
615 extern char *spa_strdup(const char *);
616 extern void spa_strfree(char *);
617 extern uint64_t spa_get_random(uint64_t range);
618 extern uint64_t spa_generate_guid(spa_t *spa);
619 extern void sprintf_blkptr(char *buf, const blkptr_t *bp);
620 extern void spa_freeze(spa_t *spa);
621 extern int spa_change_guid(spa_t *spa);
622 extern void spa_upgrade(spa_t *spa, uint64_t version);
623 extern void spa_evict_all(void);
624 extern vdev_t *spa_lookup_by_guid(spa_t *spa, uint64_t guid,
625     boolean_t l2cache);
626 extern boolean_t spa_has_spare(spa_t *, uint64_t guid);
627 extern uint64_t dva_get_dsize_sync(spa_t *spa, const dva_t *dva);
628 extern uint64_t bp_get_dsize_sync(spa_t *spa, const blkptr_t *bp);
629 extern uint64_t bp_get_dsize(spa_t *spa, const blkptr_t *bp);
630 extern boolean_t spa_has_slogs(spa_t *spa);
631 extern boolean_t spa_is_root(spa_t *spa);
632 extern boolean_t spa_writeable(spa_t *spa);

634 extern int spa_mode(spa_t *spa);
635 extern uint64_t strtonum(const char *str, char **nptr);

269 /* history logging */
270 typedef enum history_log_type {
271     LOG_CMD_POOL_CREATE,
272     LOG_CMD_NORMAL,
273     LOG_INTERNAL
274 } history_log_type_t;

276 typedef struct history_arg {
277     char *ha_history_str;
278     history_log_type_t ha_log_type;
279     history_internal_events_t ha_event;

```

```

280     char *ha_zone;
281     uid_t ha_uid;
282 } history_arg_t;

637 extern char *spa_his_ievent_table[];

639 extern void spa_history_create_obj(spa_t *spa, dmu_tx_t *tx);
640 extern int spa_history_get(spa_t *spa, uint64_t *offset, uint64_t *len_read,
641     char *his_buf);
642 extern int spa_history_log(spa_t *spa, const char *his_buf);
643 extern int spa_history_log_nvl(spa_t *spa, nvlist_t *nvl);
644 extern void spa_history_log_version(spa_t *spa, const char *operation);
645 extern void spa_history_log_internal(spa_t *spa, const char *operation,
646     dmu_tx_t *tx, const char *fmt, ...);
647 extern void spa_history_log_internal_ds(struct dsl_dataset *ds, const char *op,
648     dmu_tx_t *tx, const char *fmt, ...);
649 extern void spa_history_log_internal_dd(dsl_dir_t *dd, const char *operation,
650     dmu_tx_t *tx, const char *fmt, ...);
289 extern int spa_history_log(spa_t *spa, const char *his_buf,
290     history_log_type_t what);
291 extern void spa_history_log_internal(history_internal_events_t event,
292     spa_t *spa, dmu_tx_t *tx, const char *fmt, ...);
293 extern void spa_history_log_version(spa_t *spa, history_internal_events_t evt);

652 /* error handling */
653 struct zbookmark;
654 extern void spa_log_error(spa_t *spa, zio_t *zio);
655 extern void zfs_ereport_post(const char *class, spa_t *spa, vdev_t *vd,
656     zio_t *zio, uint64_t stateoroffset, uint64_t length);
657 extern void zfs_post_remove(spa_t *spa, vdev_t *vd);
658 extern void zfs_post_state_change(spa_t *spa, vdev_t *vd);
659 extern void zfs_post_autoreplace(spa_t *spa, vdev_t *vd);
660 extern uint64_t spa_get_errlog_size(spa_t *spa);
661 extern int spa_get_errlog(spa_t *spa, void *uaddr, size_t *count);
662 extern void spa_errlog_rotate(spa_t *spa);
663 extern void spa_errlog_drain(spa_t *spa);
664 extern void spa_errlog_sync(spa_t *spa, uint64_t txg);
665 extern void spa_get_errlists(spa_t *spa, avl_tree_t *last, avl_tree_t *scrub);

667 /* vdev cache */
668 extern void vdev_cache_stat_init(void);
669 extern void vdev_cache_stat_fini(void);

671 /* Initialization and termination */
672 extern void spa_init(int flags);
673 extern void spa_fini(void);
674 extern void spa_boot_init();

676 /* properties */
677 extern int spa_prop_set(spa_t *spa, nvlist_t *nvp);
678 extern int spa_prop_get(spa_t *spa, nvlist_t **nvp);
679 extern void spa_prop_clear_bootfs(spa_t *spa, uint64_t obj, dmu_tx_t *tx);
680 extern void spa_configfile_set(spa_t *spa, nvlist_t *nvp, boolean_t);

682 /* asynchronous event notification */
683 extern void spa_event_notify(spa_t *spa, vdev_t *vdev, const char *name);

685 #ifdef ZFS_DEBUG
686 #define dprintf_bp(bp, fmt, ...) do {
687     if (zfs_flags & ZFS_DEBUG_DPRINTF) {
688         char *blkbuf = kmem_alloc(BP_SPRINTF_LEN, KM_SLEEP);
689         sprintf_blkptr(blkbuf, (bp));
690         dprintf(fmt " %s\n", __VA_ARGS__, blkbuf);
691         kmem_free(blkbuf, BP_SPRINTF_LEN);
692     } \
693     _NOTE(CONSTCOND) } while (0)

```

```

694 #else
695 #define dprintf_bp(bp, fmt, ...)
696 #endif

698 extern boolean_t spa_debug_enabled(spa_t *spa);
699 #define spa_dbgmsg(spa, ...) \
700 { \
701     if (spa_debug_enabled(spa)) \
702         zfs_dbgmsg(__VA_ARGS__); \
703 }

```

_____ unchanged_portion_omitted _____

new/usr/src/uts/common/fs/zfs/sys/zfs_ioctl.h

1

```
*****
9827 Thu Jun 28 15:57:20 2012
new/usr/src/uts/common/fs/zfs/sys/zfs_ioctl.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */
27 #ifndef _SYS_ZFS_IOCTL_H
28 #define _SYS_ZFS_IOCTL_H
29
30 #include <sys/cred.h>
31 #include <sys/dmu.h>
32 #include <sys/zio.h>
33 #include <sys/dsl_deleg.h>
34 #include <sys/spa.h>
35 #include <sys/zfs_stat.h>
36
37 #ifdef _KERNEL
38 #include <sys/nvpair.h>
39 #endif /* _KERNEL */
40
41 #ifdef __cplusplus
42 extern "C" {
43 #endif
44
45 /*
46 * The structures in this file are passed between userland and the
47 * kernel. Userland may be running a 32-bit process, while the kernel
48 * is 64-bit. Therefore, these structures need to compile the same in
49 * 32-bit and 64-bit. This means not using type "long", and adding
50 * explicit padding so that the 32-bit structure will not be packed more
51 * tightly than the 64-bit structure (which requires 64-bit alignment).
52 */
54 /*
```

new/usr/src/uts/common/fs/zfs/sys/zfs_ioctl.h

2

```
55 #endif /* ! codereview */
56 * Property values for snapdir
57 */
58 #define ZFS_SNAPDIR_HIDDEN 0
59 #define ZFS_SNAPDIR_VISIBLE 1
60
61 /*
62 * Field manipulation macros for the drr_versioninfo field of the
63 * send stream header.
64 */
65
66 /*
67 * Header types for zfs send streams.
68 */
69 typedef enum drr_headertype {
70     DMU_SUBSTREAM = 0x1,
71     DMU_COMPOUNDSTREAM = 0x2
72 } drr_headertype_t;
73
74 #define DMU_GET_STREAM_HDRTYPE(vi) BF64_GET((vi), 0, 2)
75 #define DMU_SET_STREAM_HDRTYPE(vi, x) BF64_SET((vi), 0, 2, x)
76
77 #define DMU_GET_FEATUREFLAGS(vi) BF64_GET((vi), 2, 30)
78 #define DMU_SET_FEATUREFLAGS(vi, x) BF64_SET((vi), 2, 30, x)
79
80 /*
81 * Feature flags for zfs send streams (flags in drr_versioninfo)
82 */
83
84 #define DMU_BACKUP_FEATURE_DEDUP (0x1)
85 #define DMU_BACKUP_FEATURE_DEDUPPROPS (0x2)
86 #define DMU_BACKUP_FEATURE_SA_SPILL (0x4)
87
88 /*
89 * Mask of all supported backup features
90 */
91 #define DMU_BACKUP_FEATURE_MASK (DMU_BACKUP_FEATURE_DEDUP | \
92     DMU_BACKUP_FEATURE_DEDUPPROPS | DMU_BACKUP_FEATURE_SA_SPILL)
93
94 /* Are all features in the given flag word currently supported? */
95 #define DMU_STREAM_SUPPORTED(x) (!(x) & ~DMU_BACKUP_FEATURE_MASK)
96
97 /*
98 * The drr_versioninfo field of the dmu_replay_record has the
99 * following layout:
100 *
101 *      64      56      48      40      32      24      16      8      0
102 *      +-----+-----+-----+-----+-----+-----+-----+
103 *      |           reserved           |           feature-flags           |C|S|
104 *      +-----+-----+-----+-----+-----+-----+-----+
105 *
106 * The low order two bits indicate the header type: SUBSTREAM (0x1)
107 * or COMPOUNDSTREAM (0x2). Using two bits for this is historical:
108 * this field used to be a version number, where the two version types
109 * were 1 and 2. Using two bits for this allows earlier versions of
110 * the code to be able to recognize send streams that don't use any
111 * of the features indicated by feature flags.
112 */
113
114 #define DMU_BACKUP_MAGIC 0x2F5bacbacULL
115
116 #define DRR_FLAG_CLONE (1<<0)
117 #define DRR_FLAG_CI_DATA (1<<1)
118
119 /*
120 * flags in the drr_checksumflags field in the DRR_WRITE and
```

```

121 * DRR_WRITE_BYREF blocks
122 */
123 #define DRR_CHECKSUM_DEDUP      (1<<0)

125 #define DRR_IS_DEDUP_CAPABLE(flags)    ((flags) & DRR_CHECKSUM_DEDUP)

127 /*
128 * zfs ioctl command structure
129 */
130 typedef struct dmuf_replay_record {
131     enum {
132         DRR_BEGIN, DRR_OBJECT, DRR_FREEOBJECTS,
133         DRR_WRITE, DRR_FREE, DRR_END, DRR_WRITE_BYREF,
134         DRR_SPILL, DRR_NUMTYPES
135     } drr_type;
136     uint32_t drr_payloadlen;
137     union {
138         struct drr_begin {
139             uint64_t drr_magic;
140             uint64_t drr_versioninfo; /* was drr_version */
141             uint64_t drr_creation_time;
142             dmuf_objset_type_t drr_type;
143             uint32_t drr_flags;
144             uint64_t drr_toguid;
145             uint64_t drr_fromguid;
146             char drr_toname[MAXNAMELEN];
147         } drr_begin;
148         struct drr_end {
149             zio_cksum_t drr_checksum;
150             uint64_t drr_toguid;
151         } drr_end;
152         struct drr_object {
153             uint64_t drr_object;
154             dmuf_object_type_t drr_type;
155             dmuf_object_type_t drr_bonustype;
156             uint32_t drr_blkisz;
157             uint32_t drr_bonuslen;
158             uint8_t drr_checksumtype;
159             uint8_t drr_compress;
160             uint8_t drr_pad[6];
161             uint64_t drr_toguid;
162             /* bonus content follows */
163         } drr_object;
164         struct drr_freeobjects {
165             uint64_t drr_firstobj;
166             uint64_t drr_numobjs;
167             uint64_t drr_toguid;
168         } drr_freeobjects;
169         struct drr_write {
170             uint64_t drr_object;
171             dmuf_object_type_t drr_type;
172             uint32_t drr_pad;
173             uint64_t drr_offset;
174             uint64_t drr_length;
175             uint64_t drr_toguid;
176             uint8_t drr_checksumtype;
177             uint8_t drr_checksumflags;
178             uint8_t drr_pad2[6];
179             ddt_key_t drr_key; /* deduplication key */
180             /* content follows */
181         } drr_write;
182         struct drr_free {
183             uint64_t drr_object;
184             uint64_t drr_offset;
185             uint64_t drr_length;
186             uint64_t drr_toguid;

```

```

187     } drr_free;
188     struct drr_write_byref {
189         /* where to put the data */
190         uint64_t drr_object;
191         uint64_t drr_offset;
192         uint64_t drr_length;
193         uint64_t drr_toguid;
194         /* where to find the prior copy of the data */
195         uint64_t drr_refguid;
196         uint64_t drr_refobject;
197         uint64_t drr_refoffset;
198         /* properties of the data */
199         uint8_t drr_checksumtype;
200         uint8_t drr_checksumflags;
201         uint8_t drr_pad2[6];
202         ddt_key_t drr_key; /* deduplication key */
203     } drr_write_byref;
204     struct drr_spill {
205         uint64_t drr_object;
206         uint64_t drr_length;
207         uint64_t drr_toguid;
208         uint64_t drr_pad[4]; /* needed for crypto */
209         /* spill data follows */
210     } drr_spill;
211     } drr_u;
212 } dmuf_replay_record_t;

214 /* diff record range types */
215 typedef enum diff_type {
216     DDR_NONE = 0x1,
217     DDR_INUSE = 0x2,
218     DDR_FREE = 0x4
219 } diff_type_t;

221 /*
222 * The diff reports back ranges of free or in-use objects.
223 */
224 typedef struct dmuf_diff_record {
225     uint64_t ddr_type;
226     uint64_t ddr_first;
227     uint64_t ddr_last;
228 } dmuf_diff_record_t;

230 typedef struct zinject_record {
231     uint64_t zi_objset;
232     uint64_t zi_object;
233     uint64_t zi_start;
234     uint64_t zi_end;
235     uint64_t zi_guid;
236     uint32_t zi_level;
237     uint32_t zi_error;
238     uint64_t zi_type;
239     uint32_t zi_freq;
240     uint32_t zi_failfast;
241     char zi_func[MAXNAMELEN];
242     uint32_t zi_iotype;
243     int32_t zi_duration;
244     uint64_t zi_timer;
245 } zinject_record_t;

247 #define ZINJECT_NULL      0x1
248 #define ZINJECT_FLUSH_ARC 0x2
249 #define ZINJECT_UNLOAD_SPA 0x4

251 typedef struct zfs_share {
252     uint64_t z_exportdata;

```

```

253     uint64_t      z_sharedata;
254     uint64_t      z_sharetype; /* 0 = share, 1 = unshare */
255     uint64_t      z_sharemax; /* max length of share string */
256 } zfs_share_t;

258 /*
259  * ZFS file systems may behave the usual, POSIX-compliant way, where
260  * name lookups are case-sensitive. They may also be set up so that
261  * all the name lookups are case-insensitive, or so that only some
262  * lookups, the ones that set an FIGNORECASE flag, are case-insensitive.
263  */
264 typedef enum zfs_case {
265     ZFS_CASE_SENSITIVE,
266     ZFS_CASE_INSENSITIVE,
267     ZFS_CASE_MIXED
268 } zfs_case_t;

270 typedef struct zfs_cmd {
271     char          zc_name[MAXPATHLEN]; /* name of pool or dataset */
272     uint64_t      zc_nvlist_src; /* really (char *) */
273     uint64_t      zc_nvlist_src_size;
274     uint64_t      zc_nvlist_dst; /* really (char *) */
275     uint64_t      zc_nvlist_dst_size;
276     boolean_t     zc_nvlist_dst_filled; /* put an nvlist in dst? */
277     int           zc_pad2;

279     /*
280      * The following members are for legacy ioctls which haven't been
281      * converted to the new method.
282      */
283     uint64_t      zc_history; /* really (char *) */
284     char          zc_name[MAXPATHLEN];
285     char          zc_value[MAXPATHLEN * 2];
286     char          zc_string[MAXNAMELEN];
287     char          zc_top_ds[MAXPATHLEN];
288     uint64_t      zc_guid;
289     uint64_t      zc_nvlist_conf; /* really (char *) */
290     uint64_t      zc_nvlist_conf_size;
291     uint64_t      zc_nvlist_src; /* really (char *) */
292     uint64_t      zc_nvlist_src_size;
293     uint64_t      zc_nvlist_dst; /* really (char *) */
294     uint64_t      zc_nvlist_dst_size;
295     uint64_t      zc_cookie;
296     uint64_t      zc_objset_type;
297     uint64_t      zc_perm_action;
298     uint64_t      zc_history; /* really (char *) */
299     uint64_t      zc_history_len;
300     uint64_t      zc_history_offset;
301     uint64_t      zc_obj;
302     uint64_t      zc_iflags; /* internal to zfs(7fs) */
303     zfs_share_t   zc_share;
304     dmub_objset_stats_t zc_objset_stats;
305     struct drr_begin zc_begin_record;
306     zinject_record_t zc_inject_record;
307     boolean_t     zc_defer_destroy;
308     boolean_t     zc_temphold;
309     uint64_t      zc_action_handle;
310     int           zc_cleanup_fd;
311     uint8_t       zc_pad[4]; /* alignment */
312     uint64_t      zc_sendobj;
313     uint64_t      zc_fromobj;
314     uint64_t      zc_createtxg;
315     zfs_stat_t    zc_stat;
316 } zfs_cmd_t;

```

unchanged portion omitted

new/usr/src/uts/common/fs/zfs/zfs_ctldir.c

1

```
*****
33871 Thu Jun 28 15:57:20 2012
new/usr/src/uts/common/fs/zfs/zfs_ctldir.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */
27 /*
28 * ZFS control directory (a.k.a. ".zfs")
29 *
30 * This directory provides a common location for all ZFS meta-objects.
31 * Currently, this is only the 'snapshot' directory, but this may expand in the
32 * future. The elements are built using the GFS primitives, as the hierarchy
33 * does not actually exist on disk.
34 *
35 * For 'snapshot', we don't want to have all snapshots always mounted, because
36 * this would take up a huge amount of space in /etc/mnttab. We have three
37 * types of objects:
38 *
39 *      ctldir -----> snapshotdir -----> snapshot
40 *
41 *
42 *
43 *
44 *
45 * The 'snapshot' node contains just enough information to lookup '..' and act
46 * as a mountpoint for the snapshot. Whenever we lookup a specific snapshot, we
47 * perform an automount of the underlying filesystem and return the
48 * corresponding vnode.
49 *
50 * All mounts are handled automatically by the kernel, but unmounts are
51 * (currently) handled from user land. The main reason is that there is no
52 * reliable way to auto-unmount the filesystem when it's "no longer in use".
53 * When the user unmounts a filesystem, we call zfsctl_unmount(), which
54 * unmounts any snapshots within the snapshot directory.
```

new/usr/src/uts/common/fs/zfs/zfs_ctldir.c

2

```
55 *
56 * The '.zfs', '.zfs/snapshot', and all directories created under
57 * '.zfs/snapshot' (ie: '.zfs/snapshot/<snapname>') are all GFS nodes and
58 * share the same vfs_t as the head filesystem (what '.zfs' lives under).
59 *
60 * File systems mounted atop of the GFS nodes '.zfs/snapshot/<snapname>'
61 * (ie: snapshots) are ZFS nodes and have their own unique vfs_t.
62 * However, vnodes within these mounted on file systems have their v_vfso
63 * fields set to the head filesystem to make NFS happy (see
64 * zfsctl_snapdir_lookup()). We VFS_HOLD the head filesystem's vfs_t
65 * so that it cannot be freed until all snapshots have been unmounted.
66 */
68 #include <fs/fs_subr.h>
69 #include <sys/zfs_ctldir.h>
70 #include <sys/zfs_ioctl.h>
71 #include <sys/zfs_vfsops.h>
72 #include <sys/vfs_opreg.h>
73 #include <sys/gfs.h>
74 #include <sys/stat.h>
75 #include <sys/dmu.h>
76 #include <sys/dsl_deleg.h>
77 #include <sys/mount.h>
78 #include <sys/sunddi.h>
80 #include "zfs_namecheck.h"
82 typedef struct zfsctl_node {
83     gfs_dir_t         zc_gfs_private;
84     uint64_t         zc_id;
85     timestruct_t     zc_cmtime;      /* ctime and mtime, always the same */
86 } zfsctl_node_t;
88 typedef struct zfsctl_snapdir {
89     zfsctl_node_t     sd_node;
90     kmutex_t         sd_lock;
91     avl_tree_t       sd_snaps;
92 } zfsctl_snapdir_t;
94 typedef struct {
95     char             *se_name;
96     vnode_t         *se_root;
97     avl_node_t       se_node;
98 } zfs_snapentry_t;
100 static int
101 snapentry_compare(const void *a, const void *b)
102 {
103     const zfs_snapentry_t *sa = a;
104     const zfs_snapentry_t *sb = b;
105     int ret = strcmp(sa->se_name, sb->se_name);
107     if (ret < 0)
108         return (-1);
109     else if (ret > 0)
110         return (1);
111     else
112         return (0);
113 }
115 vnodeops_t *zfsctl_ops_root;
116 vnodeops_t *zfsctl_ops_snapdir;
117 vnodeops_t *zfsctl_ops_snapshot;
118 vnodeops_t *zfsctl_ops_shares;
119 vnodeops_t *zfsctl_ops_shares_dir;
```

```

121 static const fs_operation_def_t zfsctl_tops_root[];
122 static const fs_operation_def_t zfsctl_tops_snapdir[];
123 static const fs_operation_def_t zfsctl_tops_snapshot[];
124 static const fs_operation_def_t zfsctl_tops_shares[];

126 static vnode_t *zfsctl_mknnode_snapdir(vnode_t *);
127 static vnode_t *zfsctl_mknnode_shares(vnode_t *);
128 static vnode_t *zfsctl_snapshot_mknnode(vnode_t *, uint64_t objset);
129 static int zfsctl_unmount_snap(zfs_snapentry_t *, int, cred_t *);

131 static gfs_opsvec_t zfsctl_opsvec[] = {
132     { ".zfs", zfsctl_tops_root, &zfsctl_ops_root },
133     { ".zfs/snapshot", zfsctl_tops_snapdir, &zfsctl_ops_snapdir },
134     { ".zfs/snapshot/vnode", zfsctl_tops_snapshot, &zfsctl_ops_snapshot },
135     { ".zfs/shares", zfsctl_tops_shares, &zfsctl_ops_shares_dir },
136     { ".zfs/shares/vnode", zfsctl_tops_shares, &zfsctl_ops_shares },
137     { NULL }
138 };

140 /*
141  * Root directory elements. We only have two entries
142  * snapshot and shares.
143  */
144 static gfs_dirent_t zfsctl_root_entries[] = {
145     { "snapshot", zfsctl_mknnode_snapdir, GFS_CACHE_VNODE },
146     { "shares", zfsctl_mknnode_shares, GFS_CACHE_VNODE },
147     { NULL }
148 };

150 /* include . and .. in the calculation */
151 #define NROOT_ENTRIES ((sizeof(zfsctl_root_entries) / \
152     sizeof(gfs_dirent_t)) + 1)

155 /*
156  * Initialize the various GFS pieces we'll need to create and manipulate .zfs
157  * directories. This is called from the ZFS init routine, and initializes the
158  * vnode ops vectors that we'll be using.
159  */
160 void
161 zfsctl_init(void)
162 {
163     VERIFY(gfs_make_opsvec(zfsctl_opsvec) == 0);
164 }

166 void
167 zfsctl_fini(void)
168 {
169     /*
170      * Remove vfstl vnode ops
171      */
172     if (zfsctl_ops_root)
173         vn_freenvnodeops(zfsctl_ops_root);
174     if (zfsctl_ops_snapdir)
175         vn_freenvnodeops(zfsctl_ops_snapdir);
176     if (zfsctl_ops_snapshot)
177         vn_freenvnodeops(zfsctl_ops_snapshot);
178     if (zfsctl_ops_shares)
179         vn_freenvnodeops(zfsctl_ops_shares);
180     if (zfsctl_ops_shares_dir)
181         vn_freenvnodeops(zfsctl_ops_shares_dir);

183     zfsctl_ops_root = NULL;
184     zfsctl_ops_snapdir = NULL;
185     zfsctl_ops_snapshot = NULL;
186     zfsctl_ops_shares = NULL;

```

```

187     zfsctl_ops_shares_dir = NULL;
188 }

190 boolean_t
191 zfsctl_is_node(vnode_t *vp)
192 {
193     return (vn_matchchops(vp, zfsctl_ops_root) ||
194         vn_matchchops(vp, zfsctl_ops_snapdir) ||
195         vn_matchchops(vp, zfsctl_ops_snapshot) ||
196         vn_matchchops(vp, zfsctl_ops_shares) ||
197         vn_matchchops(vp, zfsctl_ops_shares_dir));
199 }

201 /*
202  * Return the inode number associated with the 'snapshot' or
203  * 'shares' directory.
204  */
205 /* ARGSUSED */
206 static ino64_t
207 zfsctl_root_inode_cb(vnode_t *vp, int index)
208 {
209     zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
211     ASSERT(index <= 2);

213     if (index == 0)
214         return (ZFSCTL_INO_SNAPDIR);

216     return (zfsvfs->z_shares_dir);
217 }

219 /*
220  * Create the '.zfs' directory. This directory is cached as part of the VFS
221  * structure. This results in a hold on the vfs_t. The code in zfs_umount()
222  * therefore checks against a vfs_count of 2 instead of 1. This reference
223  * is removed when the ctldir is destroyed in the unmount.
224  */
225 void
226 zfsctl_create(zfsvfs_t *zfsvfs)
227 {
228     vnode_t *vp, *rvp;
229     zfsctl_node_t *zcp;
230     uint64_t crtime[2];

232     ASSERT(zfsvfs->z_ctldir == NULL);

234     vp = gfs_root_create(sizeof(zfsctl_node_t), zfsvfs->z_vfs,
235         zfsctl_ops_root, ZFSCTL_INO_ROOT, zfsctl_root_entries,
236         zfsctl_root_inode_cb, MAXNAMELEN, NULL, NULL);
237     zcp = vp->v_data;
238     zcp->zc_id = ZFSCTL_INO_ROOT;

240     VERIFY(VFS_ROOT(zfsvfs->z_vfs, &rvp) == 0);
241     VERIFY(0 == sa_lookup(VTOZ(rvp)->z_sa_hdl, SA_ZPL_CRTIME(zfsvfs),
242         &crtime, sizeof(crtime)));
243     ZFS_TIME_DECODE(&zcp->zc_mtime, crtime);
244     VN_RELE(rvp);

246     /*
247      * We're only faking the fact that we have a root of a filesystem for
248      * the sake of the GFS interfaces. Undo the flag manipulation it did
249      * for us.
250      */
251     vp->v_flag &= ~(VROOT | VNOCACHE | VNOMAP | VNOSWAP | VNOMOUNT);

```

```

253     zfsvfs->z_ctldir = vp;
254 }

256 /*
257 * Destroy the '.zfs' directory. Only called when the filesystem is unmounted.
258 * There might still be more references if we were force unmounted, but only
259 * new zfs_inactive() calls can occur and they don't reference .zfs
260 */
261 void
262 zfsctl_destroy(zfsvfs_t *zfsvfs)
263 {
264     VN_RELE(zfsvfs->z_ctldir);
265     zfsvfs->z_ctldir = NULL;
266 }

268 /*
269 * Given a root znode, retrieve the associated .zfs directory.
270 * Add a hold to the vnode and return it.
271 */
272 vnode_t *
273 zfsctl_root(znode_t *zp)
274 {
275     ASSERT(zfs_has_ctldir(zp));
276     VN_HOLD(zp->z_zfsvfs->z_ctldir);
277     return (zp->z_zfsvfs->z_ctldir);
278 }

280 /*
281 * Common open routine. Disallow any write access.
282 */
283 /* ARGSUSED */
284 static int
285 zfsctl_common_open(vnode_t **vpp, int flags, cred_t *cr, caller_context_t *ct)
286 {
287     if (flags & FWRITE)
288         return (EACCES);

290     return (0);
291 }

293 /*
294 * Common close routine. Nothing to do here.
295 */
296 /* ARGSUSED */
297 static int
298 zfsctl_common_close(vnode_t *vpp, int flags, int count, offset_t off,
299     cred_t *cr, caller_context_t *ct)
300 {
301     return (0);
302 }

304 /*
305 * Common access routine. Disallow writes.
306 */
307 /* ARGSUSED */
308 static int
309 zfsctl_common_access(vnode_t *vp, int mode, int flags, cred_t *cr,
310     caller_context_t *ct)
311 {
312     if (flags & V_ACE_MASK) {
313         if (mode & ACE_ALL_WRITE_PERMS)
314             return (EACCES);
315     } else {
316         if (mode & VWRITE)
317             return (EACCES);
318     }

```

```

320     return (0);
321 }

323 /*
324 * Common getattr function. Fill in basic information.
325 */
326 static void
327 zfsctl_common_getattr(vnode_t *vp, vattr_t *vap)
328 {
329     timestruc_t    now;

331     vap->va_uid = 0;
332     vap->va_gid = 0;
333     vap->va_rdev = 0;
334     /*
335      * We are a purely virtual object, so we have no
336      * blocksize or allocated blocks.
337      */
338     vap->va_blksize = 0;
339     vap->va_nblocks = 0;
340     vap->va_seq = 0;
341     vap->va_fsid = vp->v_vfsp->vfs_dev;
342     vap->va_mode = S_IRUSR | S_IXUSR | S_IRGRP | S_IXGRP |
343         S_IROTH | S_IXOTH;
344     vap->va_type = VDIR;
345     /*
346      * We live in the now (for atime).
347      */
348     getthretime(&now);
349     vap->va_atime = now;
350 }

352 /*ARGSUSED*/
353 static int
354 zfsctl_common_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
355 {
356     zfsvfs_t        *zfsvfs = vp->v_vfsp->vfs_data;
357     zfsctl_node_t    *zcp = vp->v_data;
358     uint64_t         object = zcp->z_c_id;
359     zfid_short_t     *zfid;
360     int               i;

362     ZFS_ENTER(zfsvfs);

364     if (fidp->fid_len < SHORT_FID_LEN) {
365         fidp->fid_len = SHORT_FID_LEN;
366         ZFS_EXIT(zfsvfs);
367         return (ENOSPC);
368     }

370     zfid = (zfid_short_t *)fidp;
372     zfid->zf_len = SHORT_FID_LEN;

374     for (i = 0; i < sizeof (zfid->zf_object); i++)
375         zfid->zf_object[i] = (uint8_t)(object >> (8 * i));

377     /* .zfs znodes always have a generation number of 0 */
378     for (i = 0; i < sizeof (zfid->zf_gen); i++)
379         zfid->zf_gen[i] = 0;

381     ZFS_EXIT(zfsvfs);
382     return (0);
383 }

```

```

386 /*ARGSUSED*/
387 static int
388 zfsctl_shares_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
389 {
390     zfsvfs_t      *zfsvfs = vp->v_vfsp->vfs_data;
391     znode_t        *dzp;
392     int            error;
393
394     ZFS_ENTER(zfsvfs);
395
396     if (zfsvfs->z_shares_dir == 0) {
397         ZFS_EXIT(zfsvfs);
398         return (ENOTSUP);
399     }
400
401     if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &dzp)) == 0) {
402         error = VOP_FID(ZTOV(dzp), fidp, ct);
403         VN_RELE(ZTOV(dzp));
404     }
405
406     ZFS_EXIT(zfsvfs);
407     return (error);
408 }
409 /*
410  * .zfs inode namespace
411  *
412  * We need to generate unique inode numbers for all files and directories
413  * within the .zfs pseudo-file-system. We use the following scheme:
414  *
415  *     ENTRY                ZFSCTL_INODE
416  *     .zfs                  1
417  *     .zfs/snapshot        2
418  *     .zfs/snapshot/<snap> objectid(snap)
419  */
420 #define ZFSCTL_INO_SNAP(id)    (id)
421
422 /*
423  * Get root directory attributes.
424  */
425 /* ARGSUSED */
426 static int
427 zfsctl_root_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
428 caller_context_t *ct)
429 {
430     zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
431     zfsctl_node_t *zcp = vp->v_data;
432
433     ZFS_ENTER(zfsvfs);
434     vap->va_nodeid = ZFSCTL_INO_ROOT;
435     vap->va_nlink = vap->va_size = NROOT_ENTRIES;
436     vap->va_mtime = vap->va_ctime = zcp->z_cmtime;
437
438     zfsctl_common_getattr(vp, vap);
439     ZFS_EXIT(zfsvfs);
440
441     return (0);
442 }
443
444 /*
445  * Special case the handling of "...".
446  */
447 /* ARGSUSED */
448 int
449 zfsctl_root_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, pathname_t *pnp,

```

```

451 int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
452 int *direntflags, pathname_t *realpnp)
453 {
454     zfsvfs_t *zfsvfs = dvp->v_vfsp->vfs_data;
455     int err;
456
457     /*
458      * No extended attributes allowed under .zfs
459      */
460     if (flags & LOOKUP_XATTR)
461         return (EINVAL);
462
463     ZFS_ENTER(zfsvfs);
464
465     if (strcmp(nm, ".") == 0) {
466         err = VFS_ROOT(dvp->v_vfsp, vpp);
467     } else {
468         err = gfs_vop_lookup(dvp, nm, vpp, pnp, flags, rdir,
469 cr, ct, direntflags, realpnp);
470     }
471
472     ZFS_EXIT(zfsvfs);
473
474     return (err);
475 }
476
477 static int
478 zfsctl_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
479 caller_context_t *ct)
480 {
481     /*
482      * We only care about ACL_ENABLED so that libsec can
483      * display ACL correctly and not default to POSIX draft.
484      */
485     if (cmd == PC_ACL_ENABLED) {
486         *valp = _ACL_ACE_ENABLED;
487         return (0);
488     }
489
490     return (fs_pathconf(vp, cmd, valp, cr, ct));
491 }
492
493 static const fs_operation_def_t zfsctl_tops_root[] = {
494     { VOPNAME_OPEN, { .vop_open = zfsctl_common_open } },
495     { VOPNAME_CLOSE, { .vop_close = zfsctl_common_close } },
496     { VOPNAME_IOCTL, { .error = fs_inval } },
497     { VOPNAME_GETATTR, { .vop_getattr = zfsctl_root_getattr } },
498     { VOPNAME_ACCESS, { .vop_access = zfsctl_common_access } },
499     { VOPNAME_READDIR, { .vop_readdir = gfs_vop_readdir } },
500     { VOPNAME_LOOKUP, { .vop_lookup = zfsctl_root_lookup } },
501     { VOPNAME_SEEK, { .vop_seek = fs_seek } },
502     { VOPNAME_INACTIVE, { .vop_inactive = gfs_vop_inactive } },
503     { VOPNAME_PATHCONF, { .vop_pathconf = zfsctl_pathconf } },
504     { VOPNAME_FID, { .vop_fid = zfsctl_common_fid } },
505     { NULL }
506 };
507
508 static int
509 zfsctl_snapshot_zname(vnode_t *vp, const char *name, int len, char *zname)
510 {
511     objset_t *os = ((zfsvfs_t *) (vp->v_vfsp->vfs_data))->z_os;
512
513     if (snapshot_namecheck(name, NULL, NULL) != 0)
514         return (EILSEQ);
515     dmu_objset_name(os, zname);
516     if (strlen(zname) + 1 + strlen(name) >= len)

```

```

517         return (ENAMETOOLONG);
518     (void) strcat(zname, "@");
519     (void) strcat(zname, name);
520     return (0);
521 }

523 static int
524 zfsctl_unmount_snap(zfs_snapentry_t *sep, int fflags, cred_t *cr)
525 {
526     vnode_t *svp = sep->se_root;
527     int error;

529     ASSERT(vn_ismntpt(svp));

531     /* this will be dropped by dounmount() */
532     if ((error = vn_vfswlock(svp)) != 0)
533         return (error);

535     VN_HOLD(svp);
536     error = dounmount(vn_mountedvfs(svp), fflags, cr);
537     if (error) {
538         VN_RELE(svp);
539         return (error);
540     }

542     /*
543      * We can't use VN_RELE(), as that will try to invoke
544      * zfsctl_snapdir_inactive(), which would cause us to destroy
545      * the sd_lock mutex held by our caller.
546      */
547     ASSERT(svp->v_count == 1);
548     gfs_vop_inactive(svp, cr, NULL);

550     kmem_free(sep->se_name, strlen(sep->se_name) + 1);
551     kmem_free(sep, sizeof (zfs_snapentry_t));

553     return (0);
554 }

556 static void
557 zfsctl_rename_snap(zfsctl_snapdir_t *sdp, zfs_snapentry_t *sep, const char *nm)
558 {
559     avl_index_t where;
560     vfs_t *vfsp;
561     refstr_t *pathref;
562     char newpath[MAXNAMELEN];
563     char *tail;

565     ASSERT(MUTEX_HELD(&sdp->sd_lock));
566     ASSERT(sep != NULL);

568     vfsp = vn_mountedvfs(sep->se_root);
569     ASSERT(vfsp != NULL);

571     vfs_lock_wait(vfsp);

573     /*
574      * Change the name in the AVL tree.
575      */
576     avl_remove(&sdp->sd_snaps, sep);
577     kmem_free(sep->se_name, strlen(sep->se_name) + 1);
578     sep->se_name = kmem_alloc(strlen(nm) + 1, KM_SLEEP);
579     (void) strcpy(sep->se_name, nm);
580     VERIFY(avl_find(&sdp->sd_snaps, sep, &where) == NULL);
581     avl_insert(&sdp->sd_snaps, sep, where);

```

```

583     /*
584      * Change the current mountpoint info:
585      *   - update the tail of the mountpoint path
586      *   - update the tail of the resource path
587      */
588     pathref = vfs_getmntpoint(vfsp);
589     (void) strncpy(newpath, refstr_value(pathref), sizeof (newpath));
590     VERIFY((tail = strrchr(newpath, '/')) != NULL);
591     *(tail+1) = '\0';
592     ASSERT3U(strlen(newpath) + strlen(nm), <, sizeof (newpath));
593     (void) strcat(newpath, nm);
594     refstr_rele(pathref);
595     vfs_setmntpoint(vfsp, newpath, 0);

597     pathref = vfs_getresource(vfsp);
598     (void) strncpy(newpath, refstr_value(pathref), sizeof (newpath));
599     VERIFY((tail = strrchr(newpath, '@')) != NULL);
600     *(tail+1) = '\0';
601     ASSERT3U(strlen(newpath) + strlen(nm), <, sizeof (newpath));
602     (void) strcat(newpath, nm);
603     refstr_rele(pathref);
604     vfs_setresource(vfsp, newpath, 0);

606     vfs_unlock(vfsp);
607 }

609 /*ARGSUSED*/
610 static int
611 zfsctl_snapdir_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm,
612     cred_t *cr, caller_context_t *ct, int flags)
613 {
614     zfsctl_snapdir_t *sdp = sdvp->v_data;
615     zfs_snapentry_t search, *sep;
616     zfsvfs_t *zfsvfs;
617     avl_index_t where;
618     char from[MAXNAMELEN], to[MAXNAMELEN];
619     char real[MAXNAMELEN];
620     int err;

622     zfsvfs = sdvp->v_vfsp->vfs_data;
623     ZFS_ENTER(zfsvfs);

625     if ((flags & FIGIGNORECASE) || zfsvfs->z_case == ZFS_CASE_INSENSITIVE) {
626         err = dmu_snapshot_realname(zfsvfs->z_os, snm, real,
627             MAXNAMELEN, NULL);
628         if (err == 0) {
629             snm = real;
630         } else if (err != ENOTSUP) {
631             ZFS_EXIT(zfsvfs);
632             return (err);
633         }
634     }

636     ZFS_EXIT(zfsvfs);

638     err = zfsctl_snapshot_zname(sdvp, snm, MAXNAMELEN, from);
639     if (!err)
640         err = zfsctl_snapshot_zname(tdvp, tnm, MAXNAMELEN, to);
641     if (!err)
642         err = zfs_secpolicy_rename_perms(from, to, cr);
643     if (err)
644         return (err);

646     /*
647      * Cannot move snapshots out of the snapdir.
648      */

```



```

649     if (sdvp != tdvp)
650         return (EINVAL);

652     if (strcmp(snm, tnm) == 0)
653         return (0);

655     mutex_enter(&sdp->sd_lock);

657     search.se_name = (char *)snm;
658     if ((sep = avl_find(&sdp->sd_snaps, &search, &where)) == NULL) {
659         mutex_exit(&sdp->sd_lock);
660         return (ENOENT);
661     }

663     err = dmub_objset_rename(from, to, B_FALSE);
664     if (err == 0)
665         zfsctl_rename_snap(sdp, sep, tnm);

667     mutex_exit(&sdp->sd_lock);

669     return (err);
670 }

672 /* ARGSUSED */
673 static int
674 zfsctl_snapdir_remove(vnode_t *dvp, char *name, vnode_t *cwd, cred_t *cr,
675     caller_context_t *ct, int flags)
676 {
677     zfsctl_snapdir_t *sdp = dvp->v_data;
678     zfs_snapentry_t *sep;
679     zfs_snapentry_t search;
680     zfsvfs_t *zfsvfs;
681     char snapname[MAXNAMELEN];
682     char real[MAXNAMELEN];
683     int err;

685     zfsvfs = dvp->v_vfsp->vfs_data;
686     ZFS_ENTER(zfsvfs);

688     if ((flags & FIGNORECASE) || zfsvfs->z_case == ZFS_CASE_INSENSITIVE) {
690         err = dmub_snapshot_realname(zfsvfs->z_os, name, real,
691             MAXNAMELEN, NULL);
692         if (err == 0) {
693             name = real;
694         } else if (err != ENOTSUP) {
695             ZFS_EXIT(zfsvfs);
696             return (err);
697         }
698     }

700     ZFS_EXIT(zfsvfs);

702     err = zfsctl_snapshot_zname(dvp, name, MAXNAMELEN, snapname);
703     if (!err)
704         err = zfs_secpolicy_destroy_perms(snapname, cr);
705     if (err)
706         return (err);

708     mutex_enter(&sdp->sd_lock);

710     search.se_name = name;
711     sep = avl_find(&sdp->sd_snaps, &search, NULL);
712     if (sep) {
713         avl_remove(&sdp->sd_snaps, sep);
714         err = zfsctl_unmount_snap(sep, MS_FORCE, cr);

```

```

715         if (err)
716             avl_add(&sdp->sd_snaps, sep);
717         else
718             err = dmub_objset_destroy(snapname, B_FALSE);
719     } else {
720         err = ENOENT;
721     }

723     mutex_exit(&sdp->sd_lock);

725     return (err);
726 }

728 /*
729  * This creates a snapshot under '.zfs/snapshot'.
730  */
731 /* ARGSUSED */
732 static int
733 zfsctl_snapdir_mkdir(vnode_t *dvp, char *dirname, vattr_t *vap, vnode_t **vpp,
734     cred_t *cr, caller_context_t *cc, int flags, vsecattr_t *vsecp)
735 {
736     zfsvfs_t *zfsvfs = dvp->v_vfsp->vfs_data;
737     char name[MAXNAMELEN];
738     int err;
739     static enum symfollow follow = NO_FOLLOW;
740     static enum uio_seg seg = UIO_SYSSPACE;

742     if (snapshot_namecheck(dirname, NULL, NULL) != 0)
743         return (EILSEQ);

745     dmub_objset_name(zfsvfs->z_os, name);

747     *vpp = NULL;

749     err = zfs_secpolicy_snapshot_perms(name, cr);
750     if (err)
751         return (err);

753     if (err == 0) {
754         err = dmub_objset_snapshot_one(name, dirname);
755         err = dmub_objset_snapshot(name, dirname, NULL, NULL,
756             B_FALSE, B_FALSE, -1);
757         if (err)
758             return (err);
759         err = lookupnameat(dirname, seg, follow, NULL, vpp, dvp);
760     }
761     return (err);
}

```

unchanged_portion_omitted

new/usr/src/uts/common/fs/zfs/zfs_ioctl.c

1

```
*****
142372 Thu Jun 28 15:57:21 2012
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Portions Copyright 2011 Martin Matuska
25  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
26  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
27 #endif /* ! codereview */
28 * Copyright (c) 2012 by Delphix. All rights reserved.
29 */
30
31 /*
32  * ZFS ioctls.
33  *
34  * This file handles the ioctls to /dev/zfs, used for configuring ZFS storage
35  * pools and filesystems, e.g. with /sbin/zfs and /sbin/zpool.
36  *
37  * There are two ways that we handle ioctls: the legacy way where almost
38  * all of the logic is in the ioctl callback, and the new way where most
39  * of the marshalling is handled in the common entry point, zfsdev_ioctl().
40  *
41  * Non-legacy ioctls should be registered by calling
42  * zfs_ioctl_register() from zfs_ioctl_init(). The ioctl is invoked
43  * from userland by lzc_ioctl().
44  *
45  * The registration arguments are as follows:
46  *
47  * const char *name
48  *   The name of the ioctl. This is used for history logging. If the
49  *   ioctl returns successfully (the callback returns 0), and allow_log
50  *   is true, then a history log entry will be recorded with the input &
51  *   output nvlists. The log entry can be printed with "zpool history -i".
52  *
53  * zfs_ioc_t ioc
54  *   The ioctl request number, which userland will pass to ioctl(2).
```

new/usr/src/uts/common/fs/zfs/zfs_ioctl.c

2

```
55 * The ioctl numbers can change from release to release, because
56 * the caller (libzfs) must be matched to the kernel.
57 *
58 * zfs_secpolicy_func_t *secpolicy
59 * This function will be called before the zfs_ioc_func_t, to
60 * determine if this operation is permitted. It should return EPERM
61 * on failure, and 0 on success. Checks include determining if the
62 * dataset is visible in this zone, and if the user has either all
63 * zfs privileges in the zone (SYS_MOUNT), or has been granted permission
64 * to do this operation on this dataset with "zfs allow".
65 *
66 * zfs_ioc_namecheck_t namecheck
67 * This specifies what to expect in the zfs_cmd_t:zc_name -- a pool
68 * name, a dataset name, or nothing. If the name is not well-formed,
69 * the ioctl will fail and the callback will not be called.
70 * Therefore, the callback can assume that the name is well-formed
71 * (e.g. is null-terminated, doesn't have more than one '@' character,
72 * doesn't have invalid characters).
73 *
74 * zfs_ioc_poolcheck_t pool_check
75 * This specifies requirements on the pool state. If the pool does
76 * not meet them (is suspended or is readonly), the ioctl will fail
77 * and the callback will not be called. If any checks are specified
78 * (i.e. it is not POOL_CHECK_NONE), namecheck must not be NO_NAME.
79 * Multiple checks can be or-ed together (e.g. POOL_CHECK_SUSPENDED |
80 * POOL_CHECK_READONLY).
81 *
82 * boolean_t smush_outnvlst
83 * If smush_outnvlst is true, then the output is presumed to be a
84 * list of errors, and it will be "smushed" down to fit into the
85 * caller's buffer, by removing some entries and replacing them with a
86 * single "N_MORE_ERRORS" entry indicating how many were removed. See
87 * nvlst_smush() for details. If smush_outnvlst is false, and the
88 * outnvlst does not fit into the userland-provided buffer, then the
89 * ioctl will fail with ENOMEM.
90 *
91 * zfs_ioc_func_t *func
92 * The callback function that will perform the operation.
93 *
94 * The callback should return 0 on success, or an error number on
95 * failure. If the function fails, the userland ioctl will return -1,
96 * and errno will be set to the callback's return value. The callback
97 * will be called with the following arguments:
98 *
99 * const char *name
100 *   The name of the pool or dataset to operate on, from
101 *   zfs_cmd_t:zc_name. The 'namecheck' argument specifies the
102 *   expected type (pool, dataset, or none).
103 *
104 * nvlst_t *innvl
105 *   The input nvlst, deserialized from zfs_cmd_t:zc_nvlst_src. Or
106 *   NULL if no input nvlst was provided. Changes to this nvlst are
107 *   ignored. If the input nvlst could not be deserialized, the
108 *   ioctl will fail and the callback will not be called.
109 *
110 * nvlst_t *outnvl
111 *   The output nvlst, initially empty. The callback can fill it in,
112 *   and it will be returned to userland by serializing it into
113 *   zfs_cmd_t:zc_nvlst_dst. If it is non-empty, and serialization
114 *   fails (e.g. because the caller didn't supply a large enough
115 *   buffer), then the overall ioctl will fail. See the
116 *   'smush_nvlst' argument above for additional behaviors.
117 *
118 * There are two typical uses of the output nvlst:
119 * - To return state, e.g. property values. In this case,
120 *   smush_outnvlst should be false. If the buffer was not large
```

```

121 *      enough, the caller will reallocate a larger buffer and try
122 *      the ioctl again.
123 *
124 *      - To return multiple errors from an ioctl which makes on-disk
125 *      changes. In this case, smush_outnvlst should be true.
126 *      Iocltls which make on-disk modifications should generally not
127 *      use the outnvl if they succeed, because the caller can not
128 *      distinguish between the operation failing, and
129 *      deserialization failing.
130 *
131 *      Copyright (c) 2012, Joyent, Inc. All rights reserved.
132 */
133
132 #include <sys/types.h>
133 #include <sys/param.h>
134 #include <sys/errno.h>
135 #include <sys/uio.h>
136 #include <sys/buf.h>
137 #include <sys/modctl.h>
138 #include <sys/open.h>
139 #include <sys/file.h>
140 #include <sys/kmem.h>
141 #include <sys/conf.h>
142 #include <sys/cmn_err.h>
143 #include <sys/stat.h>
144 #include <sys/zfs_ioctl.h>
145 #include <sys/zfs_vfsops.h>
146 #include <sys/zfs_znode.h>
147 #include <sys/zap.h>
148 #include <sys/spa.h>
149 #include <sys/spa_impl.h>
150 #include <sys/vdev.h>
151 #include <sys/priv_impl.h>
152 #include <sys/dmu.h>
153 #include <sys/dsl_dir.h>
154 #include <sys/dsl_dataset.h>
155 #include <sys/dsl_prop.h>
156 #include <sys/dsl_deleg.h>
157 #include <sys/dmu_objset.h>
158 #include <sys/dmu_impl.h>
159 #include <sys/ddi.h>
160 #include <sys/sunndi.h>
161 #include <sys/sunlndi.h>
162 #include <sys/policy.h>
163 #include <sys/zone.h>
164 #include <sys/nvpair.h>
165 #include <sys/pathname.h>
166 #include <sys/mount.h>
167 #include <sys/sdt.h>
168 #include <sys/fs/zfs.h>
169 #include <sys/zfs_ctldir.h>
170 #include <sys/zfs_dir.h>
171 #include <sys/zfs_onexit.h>
172 #include <sys/zvol.h>
173 #include <sys/dsl_scan.h>
174 #include <sharefs/share.h>
175 #include <sys/dmu_objset.h>
176
177 #include "zfs_namecheck.h"
178 #include "zfs_prop.h"
179 #include "zfs_deleg.h"
180 #include "zfs_comutil.h"
181
182 extern struct modlfs zfs_modlfs;
183
184 extern void zfs_init(void);
185 extern void zfs_fini(void);

```

```

187 ldi_ident_t zfs_li = NULL;
188 dev_info_t *zfs_dip;
189
190 uint_t zfs_fsyncer_key;
191 extern uint_t rrw_tsd_key;
192 static uint_t zfs_allow_log_key;
193
194 typedef int zfs_ioc_legacy_func_t(zfs_cmd_t *);
195 typedef int zfs_ioc_func_t(const char *, nvlist_t *, nvlist_t *);
196 typedef int zfs_secpolicy_func_t(zfs_cmd_t *, nvlist_t *, cred_t *);
197
198 #define ZFS_IOCTL_LEGACY_FUNC_T(zfs_cmd_t *) \
199     int zfs_ioc_legacy_func_t(zfs_cmd_t *);
200 #define ZFS_IOCTL_FUNC_T(const char *, nvlist_t *, nvlist_t *) \
201     int zfs_ioc_func_t(const char *, nvlist_t *, nvlist_t *);
202 #define ZFS_IOCTL_SECPOLICY_FUNC_T(zfs_cmd_t *, nvlist_t *, cred_t *) \
203     int zfs_secpolicy_func_t(zfs_cmd_t *, nvlist_t *, cred_t *);
204
205 typedef enum {
206     NO_NAME,
207     POOL_NAME,
208     DATASET_NAME
209 } zfs_ioc_namecheck_t;
210
211 typedef enum {
212     POOL_CHECK_NONE = 1 << 0,
213     POOL_CHECK_SUSPENDED = 1 << 1,
214     POOL_CHECK_READONLY = 1 << 2,
215     POOL_CHECK_READONLY = 1 << 2
216 } zfs_ioc_poolcheck_t;
217
218 typedef struct zfs_ioc_vec {
219     zfs_ioc_legacy_func_t *zvec_legacy_func;
220 #ifndef CODEREVIEW
221     zfs_ioc_func_t *zvec_func;
222     zfs_secpolicy_func_t *zvec_secpolicy;
223     zfs_ioc_namecheck_t zvec_namecheck;
224     boolean_t zvec_allow_log;
225     boolean_t zvec_his_log;
226     zfs_ioc_poolcheck_t zvec_pool_check;
227     boolean_t zvec_smush_outnvlst;
228     const char *zvec_name;
229 #endif
230 } zfs_ioc_vec_t;
231
232 /* This array is indexed by zfs_userquota_prop_t */
233 static const char *userquota_perms[] = {
234     ZFS_DELEG_PERM_USERUSED,
235     ZFS_DELEG_PERM_USERQUOTA,
236     ZFS_DELEG_PERM_GROUPUSED,
237     ZFS_DELEG_PERM_GROUPQUOTA,
238 };
239
240 static int zfs_ioc_userspace_upgrade(zfs_cmd_t *zc);
241 static int zfs_check_settable(const char *name, nvpair_t *property,
242     cred_t *cr);
243 static int zfs_check_clearable(char *dataset, nvlist_t *props,
244     nvlist_t **errors);
245 static int zfs_fill_zplprops_root(uint64_t, nvlist_t *, nvlist_t *,
246     boolean_t *);
247 int zfs_set_prop_nvlist(const char *, zprop_source_t, nvlist_t *, nvlist_t *);
248 static int get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp);
249 int zfs_set_prop_nvlist(const char *, zprop_source_t, nvlist_t *, nvlist_t *);
250
251 /* _NOTE(PRINTFLIKE(4)) - this is printf-like, but lint is too whiney */
252 void
253 _dprintf(const char *file, const char *func, int line, const char *fmt, ...)
254 {
255     const char *newfile;
256     char buf[512];

```

```

247     va_list adx;
248
249     /*
250      * Get rid of annoying "../common/" prefix to filename.
251      */
252     newfile = strrchr(file, '/');
253     if (newfile != NULL) {
254         newfile = newfile + 1; /* Get rid of leading / */
255     } else {
256         newfile = file;
257     }
258
259     va_start(adx, fmt);
260     (void) vsnprintf(buf, sizeof (buf), fmt, adx);
261     va_end(adx);
262
263     /*
264      * To get this data, use the zfs-dprintf probe as so:
265      * dtrace -q -n 'zfs-dprintf \
266      *     /stringof(arg0) == "dbuf.c"/ \
267      *     {printf("%s: %s", stringof(arg1), stringof(arg3))}'
268      * arg0 = file name
269      * arg1 = function name
270      * arg2 = line number
271      * arg3 = message
272      */
273     DTRACE_PROBE4(zfs_dprintf,
274                 char *, newfile, char *, func, int, line, char *, buf);
275 }

```

unchanged portion omitted

```

366 static void
367 zfs_log_history(zfs_cmd_t *zc)
368 {
369     spa_t *spa;
370     char *buf;
371
372     if ((buf = history_str_get(zc)) == NULL)
373         return;
374
375     if (spa_open(zc->zc_name, &spa, FTAG) == 0) {
376         if (spa_version(spa) >= SPA_VERSION_ZPOOL_HISTORY)
377             (void) spa_history_log(spa, buf);
378         (void) spa_history_log(spa, buf, LOG_CMD_NORMAL);
379     }
380     spa_close(spa, FTAG);
381     history_str_free(buf);
382 }
383
384 /*
385  * Policy for top-level read operations (list pools). Requires no privileges,
386  * and can be used in the local zone, as there is no associated dataset.
387  */
388 /* ARGSUSED */
389 static int
390 zfs_secpolicy_none(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
391 {
392     (void) zfs_secpolicy_none(zfs_cmd_t *zc, cred_t *cr);
393     return (0);
394 }
395
396 /*
397  * Policy for dataset read operations (list children, get statistics). Requires
398  * no privileges, but must be visible in the local zone.
399  */
400 /* ARGSUSED */

```

```

399 static int
400 zfs_secpolicy_read(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
401 {
402     zfs_secpolicy_read(zfs_cmd_t *zc, cred_t *cr);
403     if (INGLOBALZONE(curproc) ||
404         zone_dataset_visible(zc->zc_name, NULL))
405         return (0);
406
407     return (ENOENT);
408 }

```

unchanged portion omitted

```

469 static int
470 zfs_secpolicy_write_perms(const char *name, const char *perm, cred_t *cr)
471 {
472     int error;
473     boolean_t descendent = B_FALSE;
474     dsl_dataset_t *ds;
475     char *at;
476
477     at = strchr(name, '@');
478     if (at != NULL && at[1] == '\0') {
479         *at = '\0';
480         descendent = B_TRUE;
481     }
482
483     error = dsl_dataset_hold(name, FTAG, &ds);
484     if (at != NULL)
485         *at = '@';
486     if (error != 0)
487         return (error);
488
489     error = zfs_dozonecheck_ds(name, ds, cr);
490     if (error == 0) {
491         error = secpolicy_zfs(cr);
492         if (error)
493             error = dsl_deleg_access_impl(ds, perm, cr);
494         error = dsl_deleg_access_impl(ds, descendent, perm, cr);
495     }
496
497     dsl_dataset_rele(ds, FTAG);
498     return (error);
499 }
500
501 static int
502 zfs_secpolicy_write_perms_ds(const char *name, dsl_dataset_t *ds,
503                             const char *perm, cred_t *cr)
504 {
505     int error;
506
507     error = zfs_dozonecheck_ds(name, ds, cr);
508     if (error == 0) {
509         error = secpolicy_zfs(cr);
510         if (error)
511             error = dsl_deleg_access_impl(ds, perm, cr);
512         error = dsl_deleg_access_impl(ds, B_FALSE, perm, cr);
513     }
514     return (error);
515 }

```

unchanged portion omitted

```

654 /* ARGSUSED */
655 static int
656 zfs_secpolicy_set_fsacl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
657 {
658     int error;
659
660     error = zfs_dozonecheck(zc->zc_name, cr);
661     if (error)
662         return (error);
663
664     /*
665      * permission to set permissions will be evaluated later in
666      * dsl_deleg_can_allow()
667      */
668     return (0);
669 }
670
671 /* ARGSUSED */
672 static int
673 zfs_secpolicy_rollback(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
674 {
675     return (zfs_secpolicy_write_perms(zc->zc_name,
676         ZFS_DELEG_PERM_ROLLBACK, cr));
677 }
678
679 /* ARGSUSED */
680 static int
681 zfs_secpolicy_send(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
682 {
683     spa_t *spa;
684     dsl_pool_t *dp;
685     dsl_dataset_t *ds;
686     char *cp;
687     int error;
688
689     /*
690      * Generate the current snapshot name from the given objsetid, then
691      * use that name for the secpolicy/zone checks.
692      */
693     cp = strchr(zc->zc_name, '@');
694     if (cp == NULL)
695         return (EINVAL);
696     error = spa_open(zc->zc_name, &spa, FTAG);
697     if (error)
698         return (error);
699
700     dp = spa_get_dsl(spa);
701     rw_enter(&dp->dp_config_rwlock, RW_READER);
702     error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
703     rw_exit(&dp->dp_config_rwlock);
704     spa_close(spa, FTAG);
705     if (error)
706         return (error);
707
708     dsl_dataset_name(ds, zc->zc_name);
709
710     error = zfs_secpolicy_write_perms_ds(zc->zc_name, ds,
711         ZFS_DELEG_PERM_SEND, cr);
712     dsl_dataset_rele(ds, FTAG);

```

```

714         return (error);
715     }
716
717 /* ARGSUSED */
718 static int
719 zfs_secpolicy_send_new(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
720 {
721     return (zfs_secpolicy_write_perms(zc->zc_name,
722         ZFS_DELEG_PERM_SEND, cr));
723 }
724
725 /* ARGSUSED */
726 #endif /* ! codereview */
727 static int
728 zfs_secpolicy_deleg_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
729 {
730     vnode_t *vp;
731     int error;
732
733     if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
734         NO_FOLLOW, NULL, &vp)) != 0)
735         return (error);
736
737     /* Now make sure mntpnt and dataset are ZFS */
738
739     if (vp->v_vfsp->vfs_fstype != zfsfstype ||
740         (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
741             zc->zc_name) != 0)) {
742         VN_RELE(vp);
743         return (EPERM);
744     }
745
746     VN_RELE(vp);
747     return (dsl_deleg_access(zc->zc_name,
748         ZFS_DELEG_PERM_SHARE, cr));
749 }
750
751 int
752 zfs_secpolicy_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
753 {
754     if (!INGLOBALZONE(curproc))
755         return (EPERM);
756
757     if (secpolicy_nfs(cr) == 0) {
758         return (0);
759     } else {
760         return (zfs_secpolicy_deleg_share(zc, innvl, cr));
761     }
762 }
763
764 int
765 zfs_secpolicy_smb_acl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
766 {
767     if (!INGLOBALZONE(curproc))
768         return (EPERM);
769
770     if (secpolicy_smb(cr) == 0) {
771         return (0);
772     } else {
773         return (zfs_secpolicy_deleg_share(zc, innvl, cr));
774     }

```

```

775 }
    unchanged_portion_omitted_

811 /* ARGSUSED */
812 #endif /* ! codereview */
813 static int
814 zfs_secpolicy_destroy(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
815 {
816     return (zfs_secpolicy_destroy_perms(zc->zc_name, cr));
817 }

819 /*
820  * Destroying snapshots with delegated permissions requires
821  * descendant mount and destroy permissions.
822  * descendant mount and destroy permissions.
823  */
824 /* ARGSUSED */
825 #endif /* ! codereview */
826 static int
827 zfs_secpolicy_destroy_snaps(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
828 {
829     nvlist_t *snaps;
830     nvpair_t *pair, *nextpair;
831     int error = 0;
832     int error;
833     char *dsname;

834     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
835         return (EINVAL);
836     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
837          pair = nextpair) {
838         dsl_dataset_t *ds;
839         dsname = kmem_asprintf("%s@", zc->zc_name);

840         nextpair = nvlist_next_nvpair(snaps, pair);
841         error = dsl_dataset_hold(nvpair_name(pair), FTAG, &ds);
842         if (error == 0) {
843             dsl_dataset_rele(ds, FTAG);
844         } else if (error == ENOENT) {
845             /*
846              * Ignore any snapshots that don't exist (we consider
847              * them "already destroyed"). Remove the name from the
848              * nvl here in case the snapshot is created between
849              * now and when we try to destroy it (in which case
850              * we don't want to destroy it since we haven't
851              * checked for permission).
852              */
853             fnvlist_remove_nvpair(snaps, pair);
854             error = 0;
855             continue;
856         } else {
857             break;
858         }
859     }
860     error = zfs_secpolicy_destroy_perms(nvpair_name(pair), cr);
861     if (error != 0)
862         break;
863 }
    unchanged_portion_omitted_

```

```

894 /* ARGSUSED */
895 #endif /* ! codereview */
896 static int
897 zfs_secpolicy_rename(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
898 {
899     return (zfs_secpolicy_rename_perms(zc->zc_name, zc->zc_value, cr));
900 }

902 /* ARGSUSED */
903 #endif /* ! codereview */
904 static int
905 zfs_secpolicy_promote(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
906 {
907     char parentname[MAXNAMELEN];
908     objset_t *clone;
909     int error;

910     error = zfs_secpolicy_write_perms(zc->zc_name,
911                                       ZFS_DELEG_PERM_PROMOTE, cr);
912     if (error)
913         return (error);

914     error = dmu_objset_hold(zc->zc_name, FTAG, &clone);

915     if (error == 0) {
916         dsl_dataset_t *pclone = NULL;
917         dsl_dir_t *dd;
918         dd = clone->os_dsl_dataset->ds_dir;

919         rw_enter(&dd->dd_pool->dp_config_rwlock, RW_READER);
920         error = dsl_dataset_hold_obj(dd->dd_pool,
921                                     dd->dd_phys->dd_origin_obj, FTAG, &pclone);
922         rw_exit(&dd->dd_pool->dp_config_rwlock);
923         if (error) {
924             dmu_objset_rele(clone, FTAG);
925             return (error);
926         }

927         error = zfs_secpolicy_write_perms(zc->zc_name,
928                                           ZFS_DELEG_PERM_MOUNT, cr);

929         dsl_dataset_name(pclone, parentname);
930         dmu_objset_rele(clone, FTAG);
931         dsl_dataset_rele(pclone, FTAG);
932         if (error == 0)
933             error = zfs_secpolicy_write_perms(parentname,
934                                               ZFS_DELEG_PERM_PROMOTE, cr);
935     }
936     return (error);
937 }

939 /* ARGSUSED */
940 #endif /* ! codereview */
941 static int
942 zfs_secpolicy_rcv(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
943 {
944     zfs_secpolicy_receive(zfs_cmd_t *zc, cred_t *cr)
945     {
946         int error;

947         if ((error = zfs_secpolicy_write_perms(zc->zc_name,
948                                               ZFS_DELEG_PERM_RECEIVE, cr)) != 0)
949             return (error);

950         if ((error = zfs_secpolicy_write_perms(zc->zc_name,

```

```

957     ZFS_DELEG_PERM_MOUNT, cr)) != 0)
958     return (error);

960     return (zfs_secpolicy_write_perms(zc->zc_name,
961     ZFS_DELEG_PERM_CREATE, cr));
962 }
    unchanged_portion_omitted

971 /*
972  * Check for permission to create each snapshot in the nvlist.
973  */
974 /* ARGSUSED */
975 #endif /* ! codereview */
976 static int
977 zfs_secpolicy_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
978 zfs_secpolicy_snapshot(zfs_cmd_t *zc, cred_t *cr)
979 {
980     nvlist_t *snaps;
981     int error;
982     nvpair_t *pair;
983 #endif /* ! codereview */

984     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
985         return (EINVAL);
986     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
987         pair = nvlist_next_nvpair(snaps, pair)) {
988         char *name = nvpair_name(pair);
989         char *atp = strchr(name, '@');

991         if (atp == NULL) {
992             error = EINVAL;
993             break;
994         }
995         *atp = '\0';
996         error = zfs_secpolicy_snapshot_perms(name, cr);
997         *atp = '@';
998         if (error != 0)
999             break;
1000     }
1001     return (error);
1002 }

1004 /* ARGSUSED */
1005 static int
1006 zfs_secpolicy_log_history(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1007 {
1008     /*
1009     * Even root must have a proper TSD so that we know what pool
1010     * to log to.
1011     */
1012     if (tsd_get(zfs_allow_log_key) == NULL)
1013         return (EPERM);
1014     return (0);
1015     return (zfs_secpolicy_snapshot_perms(zc->zc_name, cr));
1016 }

1017 static int
1018 zfs_secpolicy_create_clone(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1019 zfs_secpolicy_create(zfs_cmd_t *zc, cred_t *cr)
1020 {
1021     char    parentname[MAXNAMELEN];
1022     int    error;
1023     char    *origin;
1024 #endif /* ! codereview */

1025     if ((error = zfs_get_parent(zc->zc_name, parentname,
```

```

1026     sizeof (parentname))) != 0)
1027     return (error);

1029     if (nvlist_lookup_string(innvl, "origin", &origin) == 0 &&
1030         (error = zfs_secpolicy_write_perms(origin,
1031         if (zc->zc_value[0] != '\0') {
1032             if ((error = zfs_secpolicy_write_perms(zc->zc_value,
1033             ZFS_DELEG_PERM_CLONE, cr)) != 0)
1034                 return (error);
1035         }
1036     }

1037     if ((error = zfs_secpolicy_write_perms(parentname,
1038     ZFS_DELEG_PERM_CREATE, cr)) != 0)
1039         return (error);

1040     return (zfs_secpolicy_write_perms(parentname,
1041     ZFS_DELEG_PERM_MOUNT, cr));
1042     error = zfs_secpolicy_write_perms(parentname,
1043     ZFS_DELEG_PERM_MOUNT, cr);

1044     return (error);
1045 }

1046 static int
1047 zfs_secpolicy_umount(zfs_cmd_t *zc, cred_t *cr)
1048 {
1049     int error;

1050     error = secpolicy_fs_unmount(cr, NULL);
1051     if (error) {
1052         error = dsl_deleg_access(zc->zc_name, ZFS_DELEG_PERM_MOUNT, cr);
1053     }
1054     return (error);
1055 }

1056 /*
1057  * Policy for pool operations - create/destroy pools, add vdevs, etc. Requires
1058  * SYS_CONFIG privilege, which is not available in a local zone.
1059  */
1060 /* ARGSUSED */
1061 static int
1062 zfs_secpolicy_config(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1063 zfs_secpolicy_config(zfs_cmd_t *zc, cred_t *cr)
1064 {
1065     if (secpolicy_sys_config(cr, B_FALSE) != 0)
1066         return (EPERM);

1067     return (0);
1068 }

1069 /*
1070  * Policy for object to name lookups.
1071  */
1072 /* ARGSUSED */
1073 static int
1074 zfs_secpolicy_diff(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1075 zfs_secpolicy_diff(zfs_cmd_t *zc, cred_t *cr)
1076 {
1077     int error;

1078     if ((error = secpolicy_sys_config(cr, B_FALSE)) == 0)
1079         return (0);

1080     error = zfs_secpolicy_write_perms(zc->zc_name, ZFS_DELEG_PERM_DIFF, cr);
1081     return (error);
1082 }

```

```

1072 /*
1073  * Policy for fault injection.  Requires all privileges.
1074  */
1075 /* ARGSUSED */
1076 static int
1077 zfs_secpolicy_inject(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1078 {
1079     return (secpolicy_zinject(cr));
1080 }

1082 /* ARGSUSED */
1083 #endif /* ! codereview */
1084 static int
1085 zfs_secpolicy_inherit_prop(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1086 {
1087     zfs_prop_t prop = zfs_name_to_prop(zc->zc_value);

1089     if (prop == ZPROP_INVALID) {
1090         if (!zfs_prop_user(zc->zc_value))
1091             return (EINVAL);
1092         return (zfs_secpolicy_write_perms(zc->zc_name,
1093             ZFS_DELEG_PERM_USERPROP, cr));
1094     } else {
1095         return (zfs_secpolicy_setprop(zc->zc_name, prop,
1096             NULL, cr));
1097     }
1098 }

1100 static int
1101 zfs_secpolicy_userspace_one(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1102 {
1103     int err = zfs_secpolicy_read(zc, innvl, cr);
1104     if (err)
1105         return (err);

1107     if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
1108         return (EINVAL);

1110     if (zc->zc_value[0] == 0) {
1111         /*
1112          * They are asking about a posix uid/gid.  If it's
1113          * themself, allow it.
1114          */
1115         if (zc->zc_objset_type == ZFS_PROP_USERUSED ||
1116             zc->zc_objset_type == ZFS_PROP_USERQUOTA) {
1117             if (zc->zc_guid == crgetuid(cr))
1118                 return (0);
1119         } else {
1120             if (groupmember(zc->zc_guid, cr))
1121                 return (0);
1122         }
1123     }

1125     return (zfs_secpolicy_write_perms(zc->zc_name,
1126         userquota_perms[zc->zc_objset_type], cr));
1127 }

1129 static int
1130 zfs_secpolicy_userspace_many(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1131 {
1132     zfs_secpolicy_userspace_many(zfs_cmd_t *zc, cred_t *cr)

```

```

1132     int err = zfs_secpolicy_read(zc, innvl, cr);
1133     if (err)
1134         return (err);

1136     if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
1137         return (EINVAL);

1139     return (zfs_secpolicy_write_perms(zc->zc_name,
1140         userquota_perms[zc->zc_objset_type], cr));
1141 }

1143 /* ARGSUSED */
1144 #endif /* ! codereview */
1145 static int
1146 zfs_secpolicy_userspace_upgrade(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1147 {
1148     return (zfs_secpolicy_setprop(zc->zc_name, ZFS_PROP_VERSION,
1149         NULL, cr));
1150 }

1152 /* ARGSUSED */
1153 #endif /* ! codereview */
1154 static int
1155 zfs_secpolicy_hold(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1156 {
1157     return (zfs_secpolicy_write_perms(zc->zc_name,
1158         ZFS_DELEG_PERM_HOLD, cr));
1159 }

1161 /* ARGSUSED */
1162 #endif /* ! codereview */
1163 static int
1164 zfs_secpolicy_release(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1165 {
1166     return (zfs_secpolicy_write_perms(zc->zc_name,
1167         ZFS_DELEG_PERM_RELEASE, cr));
1168 }

1170 /*
1171  * Policy for allowing temporary snapshots to be taken or released
1172  */
1173 static int
1174 zfs_secpolicy_tmp_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1175 {
1176     /*
1177      * A temporary snapshot is the same as a snapshot,
1178      * hold, destroy and release all rolled into one.
1179      * Delegated diff alone is sufficient that we allow this.
1180      */
1181     int error;

1183     if ((error = zfs_secpolicy_write_perms(zc->zc_name,
1184         ZFS_DELEG_PERM_DIFF, cr)) == 0)
1185         return (0);

1187     error = zfs_secpolicy_snapshot_perms(zc->zc_name, cr);
1188     error = zfs_secpolicy_snapshot(zc, cr);
1189     if (!error)
1190         error = zfs_secpolicy_hold(zc, innvl, cr);
1191     if (!error)
1192         error = zfs_secpolicy_hold(zc, cr);
1193     if (!error)
1194         return (0);

```



```

1191         error = zfs_secpolicy_release(zc, innvl, cr);
1190         error = zfs_secpolicy_release(zc, cr);
1192         if (!error)
1193             error = zfs_secpolicy_destroy(zc, innvl, cr);
1194             error = zfs_secpolicy_destroy(zc, cr);
1195         return (error);
1196     }
1197     unchanged_portion_omitted
1198
1199     /*
1200     * Reduce the size of this nvlist until it can be serialized in 'max' bytes.
1201     * Entries will be removed from the end of the nvlist, and one int32 entry
1202     * named "N_MORE_ERRORS" will be added indicating how many entries were
1203     * removed.
1204     */
1205     #endif /* ! codereview */
1206     static int
1207     nvlist_smush(nvlist_t *errors, size_t max)
1208     fit_error_list(zfs_cmd_t *zc, nvlist_t **errors)
1209     {
1210         size_t size;
1211
1212         size = fnvlist_size(errors);
1213         VERIFY(nvlist_size(*errors, &size, NV_ENCODE_NATIVE) == 0);
1214
1215         if (size > max) {
1216             if (size > zc->zc_nvlist_dst_size) {
1217                 nvpair_t *more_errors;
1218                 int n = 0;
1219
1220                 if (max < 1024)
1221                     if (zc->zc_nvlist_dst_size < 1024)
1222                         return (ENOMEM);
1223
1224                 fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, 0);
1225                 more_errors = nvlist_prev_nvpair(errors, NULL);
1226                 VERIFY(nvlist_add_int32(*errors, ZPROP_N_MORE_ERRORS, 0) == 0);
1227                 more_errors = nvlist_prev_nvpair(*errors, NULL);
1228
1229                 do {
1230                     nvpair_t *pair = nvlist_prev_nvpair(errors,
1231                                                         nvpair_t *pair = nvlist_prev_nvpair(*errors,
1232                                                         more_errors);
1233                     fnvlist_remove_nvpair(errors, pair);
1234                     VERIFY(nvlist_remove_nvpair(*errors, pair) == 0);
1235                     n++;
1236                     size = fnvlist_size(errors);
1237                 } while (size > max);
1238
1239                 fnvlist_remove_nvpair(errors, more_errors);
1240                 fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, n);
1241                 ASSERT3U(fnvlist_size(errors), <=, max);
1242                 VERIFY(nvlist_size(*errors, &size,
1243                                 NV_ENCODE_NATIVE) == 0);
1244             } while (size > zc->zc_nvlist_dst_size);
1245
1246             VERIFY(nvlist_remove_nvpair(*errors, more_errors) == 0);
1247             VERIFY(nvlist_add_int32(*errors, ZPROP_N_MORE_ERRORS, n) == 0);
1248             ASSERT(nvlist_size(*errors, &size, NV_ENCODE_NATIVE) == 0);
1249             ASSERT(size <= zc->zc_nvlist_dst_size);
1250         }
1251
1252         return (0);
1253     }
1254
1255     static int

```

```

1273 put_nvlist(zfs_cmd_t *zc, nvlist_t *nvl)
1274 {
1275     char *packed = NULL;
1276     int error = 0;
1277     size_t size;
1278
1279     size = fnvlist_size(nvl);
1280     VERIFY(nvlist_size(nvl, &size, NV_ENCODE_NATIVE) == 0);
1281
1282     if (size > zc->zc_nvlist_dst_size) {
1283         error = ENOMEM;
1284     } else {
1285         packed = fnvlist_pack(nvl, &size);
1286         packed = kmem_alloc(size, KM_SLEEP);
1287         VERIFY(nvlist_pack(nvl, &packed, &size, NV_ENCODE_NATIVE,
1288                           KM_SLEEP) == 0);
1289         if (ddi_copyout(packed, (void *) (uintptr_t) zc->zc_nvlist_dst,
1290                       size, zc->zc_iflags) != 0)
1291             error = EFAULT;
1292         fnvlist_pack_free(packed, size);
1293         kmem_free(packed, size);
1294     }
1295
1296     zc->zc_nvlist_dst_size = size;
1297     zc->zc_nvlist_dst_filled = B_TRUE;
1298     #endif /* ! codereview */
1299     return (error);
1300 }
1301
1302 static int
1303 getzfsvfs(const char *dsname, zfsvfs_t **zfvp)
1304 {
1305     objset_t *os;
1306     int error;
1307
1308     error = dmub_objset_hold(dsname, FTAG, &os);
1309     if (error)
1310         return (error);
1311     if (dmub_objset_type(os) != DMU_OST_ZFS) {
1312         dmub_objset_rele(os, FTAG);
1313         return (EINVAL);
1314     }
1315
1316     mutex_enter(&os->os_user_ptr_lock);
1317     *zfvp = dmub_objset_get_user(os);
1318     if (*zfvp) {
1319         VFS_HOLD((*zfvp)->z_vfs);
1320     } else {
1321         error = ESRCH;
1322     }
1323     mutex_exit(&os->os_user_ptr_lock);
1324     dmub_objset_rele(os, FTAG);
1325     return (error);
1326 }
1327
1328 /*
1329 * Find a zfsvfs_t for a mounted filesystem, or create our own, in which
1330 * case its z_vfs will be NULL, and it will be opened as the owner.
1331 * If 'writer' is set, the z_teardown_lock will be held for RW_WRITER,
1332 * which prevents all vnode ops from running.
1333 */
1334 static int
1335 zfsvfs_hold(const char *name, void *tag, zfsvfs_t **zfvp, boolean_t writer)
1336 {
1337     int error = 0;

```

```

1334     if (getzfsvfs(name, zfvp) != 0)
1335         error = zfsvfs_create(name, zfvp);
1336     if (error == 0) {
1337         rrw_enter(&(*zfvp)->z_teardown_lock, (writer) ? RW_WRITER :
1338             RW_READER, tag);
1339         if ((*zfvp)->z_unmounted) {
1340             /*
1341              * XXX we could probably try again, since the unmounting
1342              * thread should be just about to disassociate the
1343              * objset from the zfsvfs.
1344              */
1345             rrw_exit(&(*zfvp)->z_teardown_lock, tag);
1346             return (EBUSY);
1347         }
1348     }
1349     return (error);
1350 }

1352 static void
1353 zfsvfs_rele(zfsvfs_t *zfsvfs, void *tag)
1354 {
1355     rrw_exit(&zfsvfs->z_teardown_lock, tag);

1357     if (zfsvfs->z_vfs) {
1358         VFS_RELE(zfsvfs->z_vfs);
1359     } else {
1360         dmu_objset_disown(zfsvfs->z_os, zfsvfs);
1361         zfsvfs_free(zfsvfs);
1362     }
1363 }

1365 static int
1366 zfs_ioc_pool_create(zfs_cmd_t *zc)
1367 {
1368     int error;
1369     nvlist_t *config, *props = NULL;
1370     nvlist_t *rootprops = NULL;
1371     nvlist_t *zplprops = NULL;
1372     char *buf;

1373     if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1374         zc->zc_iflags, &config))
1375         return (error);

1377     if (zc->zc_nvlist_src_size != 0 && (error =
1378         get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1379         zc->zc_iflags, &props))) {
1380         nvlist_free(config);
1381         return (error);
1382     }

1384     if (props) {
1385         nvlist_t *nvl = NULL;
1386         uint64_t version = SPA_VERSION;

1388         (void) nvlist_lookup_uint64(props,
1389             zpool_prop_to_name(ZPOOL_PROP_VERSION), &version);
1390         if (!SPA_VERSION_IS_SUPPORTED(version)) {
1391             error = EINVAL;
1392             goto pool_props_bad;
1393         }
1394         (void) nvlist_lookup_nvlist(props, ZPOOL_ROOTFS_PROPS, &nvl);
1395         if (nvl) {
1396             error = nvlist_dup(nvl, &rootprops, KM_SLEEP);
1397             if (error != 0) {
1398                 nvlist_free(config);

```

```

1399         nvlist_free(props);
1400         return (error);
1401     }
1402     (void) nvlist_remove_all(props, ZPOOL_ROOTFS_PROPS);
1403 }
1404     VERIFY(nvlist_alloc(&zplprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);
1405     error = zfs_fill_zplprops_root(version, rootprops,
1406         zplprops, NULL);
1407     if (error)
1408         goto pool_props_bad;
1409 }

1411     error = spa_create(zc->zc_name, config, props, zplprops);
1412     buf = history_str_get(zc);

1414     error = spa_create(zc->zc_name, config, props, buf, zplprops);

1416     /*
1417      * Set the remaining root properties
1418      */
1419     if (!error && (error = zfs_set_prop_nvlist(zc->zc_name,
1420         ZPROP_SRC_LOCAL, rootprops, NULL)) != 0)
1421         (void) spa_destroy(zc->zc_name);

1422     if (buf != NULL)
1423         history_str_free(buf);

1424 pool_props_bad:
1425     nvlist_free(rootprops);
1426     nvlist_free(zplprops);
1427     nvlist_free(config);
1428     nvlist_free(props);

1429     return (error);
1430 }

1431     unchanged_portion_omitted

1432 /*
1433  * This function is best effort. If it fails to set any of the given properties,
1434  * it continues to set as many as it can and returns the last error
1435  * encountered. If the caller provides a non-NULL errlist, it will be filled in
1436  * with the list of names of all the properties that failed along with the
1437  * corresponding error numbers.
1438  * it continues to set as many as it can and returns the first error
1439  * encountered. If the caller provides a non-NULL errlist, it also gives the
1440  * complete list of names of all the properties it failed to set along with the
1441  * corresponding error numbers. The caller is responsible for freeing the
1442  * returned errlist.
1443  *
1444  * If every property is set successfully, zero is returned and errlist is not
1445  * modified.
1446  * If every property is set successfully, zero is returned and the list pointed
1447  * at by errlist is NULL.
1448  */
1449 int
1450 zfs_set_prop_nvlist(const char *dsname, zpool_source_t source, nvlist_t *nvl,
1451     nvlist_t *errlist)
1452 {
1453     nvlist_t **errlist;
1454     {
1455         nvpair_t *pair;
1456         nvpair_t *propval;
1457         int rv = 0;
1458         uint64_t intval;
1459         char *strval;
1460         nvlist_t *genericnvl = fnvlist_alloc();
1461         nvlist_t *retrynvl = fnvlist_alloc();

```

```

2133     nvlist_t *genericnvl;
2134     nvlist_t *errors;
2135     nvlist_t *retrynvl;

2137     VERIFY(nvlist_alloc(&genericnvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2138     VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2139     VERIFY(nvlist_alloc(&retrynvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2434 retry:
2435     pair = NULL;
2436     while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2437         const char *propname = nvpair_name(pair);
2438         zfs_prop_t prop = zfs_name_to_prop(propname);
2439         int err = 0;

2441         /* decode the property value */
2442         propval = pair;
2443         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2444             nvlist_t *attrs;
2445             attrs = fnvpair_value_nvlist(pair);
2446             VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2447             if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2448                 &propval) != 0)
2449                 err = EINVAL;
2449         }

2451         /* Validate value type */
2452         if (err == 0 && prop == ZPROP_INVAL) {
2453             if (zfs_prop_user(propname)) {
2454                 if (nvpair_type(propval) != DATA_TYPE_STRING)
2455                     err = EINVAL;
2456             } else if (zfs_prop_userquota(propname)) {
2457                 if (nvpair_type(propval) !=
2458                     DATA_TYPE_UINT64_ARRAY)
2459                     err = EINVAL;
2460             } else {
2461                 err = EINVAL;
2462             }
2463         } else if (err == 0) {
2464             if (nvpair_type(propval) == DATA_TYPE_STRING) {
2465                 if (zfs_prop_get_type(prop) != PROP_TYPE_STRING)
2466                     err = EINVAL;
2467             } else if (nvpair_type(propval) == DATA_TYPE_UINT64) {
2468                 const char *unused;

2470                 intval = fnvpair_value_uint64(propval);
2471                 VERIFY(nvpair_value_uint64(propval,
2472                     &intval) == 0);

2472                 switch (zfs_prop_get_type(prop)) {
2473                     case PROP_TYPE_NUMBER:
2474                         break;
2475                     case PROP_TYPE_STRING:
2476                         err = EINVAL;
2477                         break;
2478                     case PROP_TYPE_INDEX:
2479                         if (zfs_prop_index_to_string(prop,
2480                             intval, &unused) != 0)
2481                             err = EINVAL;
2482                         break;
2483                     default:
2484                         cmn_err(CE_PANIC,
2485                             "unknown property type");
2486                 }
2487             } else {
2488                 err = EINVAL;

```

```

2489     }
2490 }

2492 /* Validate permissions */
2493 if (err == 0)
2494     err = zfs_check_settable(dsname, pair, CRED());

2496 if (err == 0) {
2497     err = zfs_prop_set_special(dsname, source, pair);
2498     if (err == -1) {
2499         /*
2500          * For better performance we build up a list of
2501          * properties to set in a single transaction.
2502          */
2503         err = nvlist_add_nvpair(genericnvl, pair);
2504     } else if (err != 0 && nvl != retrynvl) {
2505         /*
2506          * This may be a spurious error caused by
2507          * receiving quota and reservation out of order.
2508          * Try again in a second pass.
2509          */
2510         err = nvlist_add_nvpair(retrynvl, pair);
2511     }
2512 }

2514 if (err != 0) {
2515     if (errlist != NULL)
2516         fnvlist_add_int32(errlist, propname, err);
2517     rv = err;
2518 }
2519 if (err != 0)
2520     VERIFY(nvlist_add_int32(errors, propname, err) == 0);

2521 if (nvl != retrynvl && !nvlist_empty(retrynvl)) {
2522     nvl = retrynvl;
2523     goto retry;
2524 }

2526 if (!nvlist_empty(genericnvl) &&
2527     dsl_props_set(dsname, source, genericnvl) != 0) {
2528     /*
2529      * If this fails, we still want to set as many properties as we
2530      * can, so try setting them individually.
2531      */
2532     pair = NULL;
2533     while ((pair = nvlist_next_nvpair(genericnvl, pair)) != NULL) {
2534         const char *propname = nvpair_name(pair);
2535         int err = 0;

2537         propval = pair;
2538         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2539             nvlist_t *attrs;
2540             attrs = fnvpair_value_nvlist(pair);
2541             propval = fnvlist_lookup_nvpair(attrs,
2542                 ZPROP_VALUE);
2543             VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2544             VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2545                 &propval) == 0);
2546         }

2547         if (nvpair_type(propval) == DATA_TYPE_STRING) {
2548             strval = fnvpair_value_string(propval);
2549             VERIFY(nvpair_value_string(propval,
2550                 &strval) == 0);
2551             err = dsl_prop_set(dsname, propname, source, 1,

```

```

2548         strlen(strval) + 1, strval);
2549     } else {
2550         intval = fnvpair_value_uint64(propval);
2551         VERIFY(nvpair_value_uint64(propval,
2552             &intval) == 0);
2553         err = dsl_prop_set(dsname, propname, source, 8,
2554             1, &intval);
2555     }
2556     if (err != 0) {
2557         if (errlist != NULL) {
2558             fnvlist_add_int32(errlist, propname,
2559                 err);
2560         }
2561         rv = err;
2562         VERIFY(nvlist_add_int32(errors, propname,
2563             err) == 0);
2564     }
2565     nvlist_free(genericnvl);
2566     nvlist_free(retrynvl);
2567
2568     if ((pair = nvlist_next_nvpair(errors, NULL)) == NULL) {
2569         nvlist_free(errors);
2570         errors = NULL;
2571     } else {
2572         VERIFY(nvpair_value_int32(pair, &rv) == 0);
2573     }
2574
2575     if (errlist == NULL)
2576         nvlist_free(errors);
2577     else
2578         *errlist = errors;
2579
2580     return (rv);
2581 }
2582
2583 /*
2584  * Check that all the properties are valid user properties.
2585  */
2586 static int
2587 zfs_check_userprops(const char *fsname, nvlist_t *nvl)
2588 zfs_check_userprops(char *fsname, nvlist_t *nvl)
2589 {
2590     nvpair_t *pair = NULL;
2591     int error = 0;
2592
2593     while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2594         const char *propname = nvpair_name(pair);
2595         char *valstr;
2596
2597         if (!zfs_prop_user(propname) ||
2598             nvpair_type(pair) != DATA_TYPE_STRING)
2599             return (EINVAL);
2600
2601         if (error = zfs_secpolicy_write_perms(fsname,
2602             ZFS_DELEG_PERM_USERPROP, CRED()))
2603             return (error);
2604
2605         if (strlen(propname) >= ZAP_MAXNAMELEN)
2606             return (ENAMETOOLONG);
2607
2608         VERIFY(nvpair_value_string(pair, &valstr) == 0);
2609         if (strlen(valstr) >= ZAP_MAXVALUELEN)
2610             return (E2BIG);

```

```

2597     }
2598     return (0);
2599 }
2600 _____unchanged_portion_omitted_____
2601
2602 /*
2603  * inputs:
2604  * zc_name          name of filesystem
2605  * zc_value         name of property to set
2606  * zc_nvlist_src[_size] nvlist of properties to apply
2607  * zc_cookie        received properties flag
2608  * outputs:
2609  * zc_nvlist_dst[_size] error for each unapplied received property
2610  */
2611 static int
2612 zfs_ioc_set_prop(zfs_cmd_t *zc)
2613 {
2614     nvlist_t *nvl;
2615     boolean_t received = zc->zc_cookie;
2616     zprop_source_t source = (received ? ZPROP_SRC_RECEIVED :
2617         ZPROP_SRC_LOCAL);
2618     nvlist_t *errors;
2619     int error;
2620
2621     if ((error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2622         zc->zc_iflags, &nvl)) != 0)
2623         return (error);
2624
2625     if (received) {
2626         nvlist_t *origprops;
2627         objset_t *os;
2628
2629         if (dmu_objset_hold(zc->zc_name, FTAG, &os) == 0) {
2630             if (dsl_prop_get_received(os, &origprops) == 0) {
2631                 (void) clear_received_props(os,
2632                     zc->zc_name, origprops, nvl);
2633                 nvlist_free(origprops);
2634             }
2635
2636             dsl_prop_set_hasrecvd(os);
2637             dmu_objset_rele(os, FTAG);
2638         }
2639
2640         errors = fnvlist_alloc();
2641         error = zfs_set_prop_nvlist(zc->zc_name, source, nvl, errors);
2642         error = zfs_set_prop_nvlist(zc->zc_name, source, nvl, &errors);
2643
2644         if (zc->zc_nvlist_dst != NULL && errors != NULL) {
2645             (void) put_nvlist(zc, errors);
2646         }
2647
2648         nvlist_free(errors);
2649         nvlist_free(nvl);
2650         return (error);
2651     }
2652
2653     /*
2654      * inputs:
2655      * zc_name          name of filesystem
2656      * zc_value         name of property to inherit
2657      * zc_cookie        revert to received value if TRUE
2658      * outputs:         none

```

```

2696 */
2697 static int
2698 zfs_ioc_inherit_prop(zfs_cmd_t *zc)
2699 {
2700     const char *propname = zc->zc_value;
2701     zfs_prop_t prop = zfs_name_to_prop(propname);
2702     boolean_t received = zc->zc_cookie;
2703     zprop_source_t source = (received
2704         ? ZPROP_SRC_NONE
2705         : ZPROP_SRC_INHERITED); /* revert to received value, if any */
2706         /* explicitly inherit */
2707
2708     if (received) {
2709         nvlist_t *dummy;
2710         nvpair_t *pair;
2711         zprop_type_t type;
2712         int err;
2713
2714         /*
2715          * zfs_prop_set_special() expects properties in the form of an
2716          * nvpair with type info.
2717          */
2718         if (prop == ZPROP_INVALID) {
2719             if (!zfs_prop_user(propname))
2720                 return (EINVAL);
2721
2722             type = PROP_TYPE_STRING;
2723         } else if (prop == ZFS_PROP_VOLSIZE ||
2724             prop == ZFS_PROP_VERSION) {
2725             return (EINVAL);
2726         } else {
2727             type = zfs_prop_get_type(prop);
2728         }
2729
2730         VERIFY(nvlist_alloc(&dummy, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2731
2732         switch (type) {
2733             case PROP_TYPE_STRING:
2734                 VERIFY(0 == nvlist_add_string(dummy, propname, ""));
2735                 break;
2736             case PROP_TYPE_NUMBER:
2737             case PROP_TYPE_INDEX:
2738                 VERIFY(0 == nvlist_add_uint64(dummy, propname, 0));
2739                 break;
2740             default:
2741                 nvlist_free(dummy);
2742                 return (EINVAL);
2743         }
2744
2745         pair = nvlist_next_nvpair(dummy, NULL);
2746         err = zfs_prop_set_special(zc->zc_name, source, pair);
2747         nvlist_free(dummy);
2748         if (err != -1)
2749             return (err); /* special property already handled */
2750     } else {
2751         /*
2752          * Only check this in the non-received case. We want to allow
2753          * 'inherit -S' to revert non-inheritable properties like quota
2754          * and reservation to the received or default values even though
2755          * they are not considered inheritable.
2756          */
2757         if (prop != ZPROP_INVALID && !zfs_prop_inheritable(prop))
2758             return (EINVAL);
2759     }
2760
2761     /* property name has been validated by zfs_secpolicy_inherit_prop() */
2762     /* the property name has been validated by zfs_secpolicy_inherit() */

```

```

2761         return (dsl_prop_set(zc->zc_name, zc->zc_value, source, 0, 0, NULL));
2762     }
2763     _____
2764     unchanged_portion_omitted_
2765
2766     3102 /*
2767     3103 * innvl: {
2768     3104 *     "type" -> dmub_objset_type_t (int32)
2769     3105 *     (optional) "props" -> { prop -> value }
2770     3106 * }
2771     3107 *
2772     3108 * inputs:
2773     3109 *     zc_objset_type      type of objset to create (fs vs zvol)
2774     3110 *     zc_name            name of new objset
2775     3111 *     zc_value          name of snapshot to clone from (may be empty)
2776     3112 *     zc_nvlist_src_size nvlist of properties to apply
2777     3113 *
2778     3114 * outnvl: propname -> error code (int32)
2779     3115 * outputs: none
2780     3116 */
2781     3117 static int
2782     3118 zfs_ioc_create(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
2783     3119 zfs_ioc_create(zfs_cmd_t *zc)
2784     3120 {
2785     3121     objset_t *clone;
2786     3122     int error = 0;
2787     3123     zfs_creat_t zct = { 0 };
2788     3124     zfs_creat_t zct;
2789     3125     nvlist_t *nvprops = NULL;
2790     3126     void (*cbfunc)(objset_t *os, void *arg, cred_t *cr, dmub_tx_t *tx);
2791     3127     int32_t type32;
2792     3128     dmub_objset_type_t type;
2793     3129     boolean_t is_insensitive = B_FALSE;
2794
2795     3130     if (nvlist_lookup_int32(innvl, "type", &type32) != 0)
2796         return (EINVAL);
2797     3131     type = type32;
2798     3132     (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);
2799     3133     dmub_objset_type_t type = zc->zc_objset_type;
2800
2801     3134     switch (type) {
2802     3135     case DMU_OST_ZFS:
2803         3136         cbfunc = zfs_create_cb;
2804         3137         break;
2805     case DMU_OST_ZVOL:
2806         3138         cbfunc = zvol_create_cb;
2807         3139         break;
2808     default:
2809         3140         cbfunc = NULL;
2810         3141         break;
2811     }
2812     3142     if (strchr(fsname, '@') ||
2813         3143         strchr(fsname, '%'))
2814         3144         if (strchr(zc->zc_name, '@') ||
2815             3145             strchr(zc->zc_name, '%'))
2816             3146             return (EINVAL);
2817
2818     3147     if (zc->zc_nvlist_src != NULL &&
2819         3148         (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2820             3149             zc->zc_iflags, &nvprops)) != 0)
2821         3150             return (error);
2822
2823     3151     zct.zct_zplprops = NULL;
2824     3152     zct.zct_props = nvprops;

```

```

3145     if (cbfunc == NULL)
3146     if (zc->zc_value[0] != '\0') {
3147         /*
3148          * We're creating a clone of an existing snapshot.
3149          */
3150         zc->zc_value[sizeof (zc->zc_value) - 1] = '\0';
3151         if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0) {
3152             nvlist_free(nvprops);
3153             return (EINVAL);
3154         }
3155
3156         error = dmu_objset_hold(zc->zc_value, FTAG, &clone);
3157         if (error) {
3158             nvlist_free(nvprops);
3159             return (error);
3160         }
3161
3162         error = dmu_objset_clone(zc->zc_name, dmu_objset_ds(clone), 0);
3163         dmu_objset_rele(clone, FTAG);
3164         if (error) {
3165             nvlist_free(nvprops);
3166             return (error);
3167         }
3168     } else {
3169         boolean_t is_insensitive = B_FALSE;
3170
3171         if (cbfunc == NULL) {
3172             nvlist_free(nvprops);
3173             return (EINVAL);
3174         }
3175
3176         if (type == DMU_OST_ZVOL) {
3177             uint64_t volsize, volblocksize;
3178
3179             if (nvprops == NULL)
3180                 return (EINVAL);
3181             if (nvlist_lookup_uint64(nvprops,
3182                 zfs_prop_to_name(ZFS_PROP_VOLSIZE), &volsize) != 0)
3183                 if (nvprops == NULL ||
3184                     nvlist_lookup_uint64(nvprops,
3185                         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
3186                             &volblocksize) != 0) {
3187                     nvlist_free(nvprops);
3188                     return (EINVAL);
3189                 }
3190
3191             if ((error = nvlist_lookup_uint64(nvprops,
3192                 zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
3193                     &volblocksize)) != 0 && error != ENOENT) {
3194                 &volblocksize) != 0 && error != ENOENT) {
3195                     nvlist_free(nvprops);
3196                     return (EINVAL);
3197                 }
3198
3199             if (error != 0)
3200                 volblocksize = zfs_prop_default_numeric(
3201                     ZFS_PROP_VOLBLOCKSIZE);
3202
3203             if ((error = zvol_check_volblocksize(
3204                 volblocksize)) != 0 ||
3205                 (error = zvol_check_volsize(volsize,
3206                     volblocksize)) != 0) {
3207                 volblocksize) != 0) {
3208                     nvlist_free(nvprops);
3209                     return (error);
3210                 }
3211
3212             if (error != 0)
3213                 volblocksize = zfs_prop_default_numeric(
3214                     ZFS_PROP_VOLBLOCKSIZE);
3215
3216             if ((error = zvol_check_volblocksize(
3217                 volblocksize)) != 0 ||
3218                 (error = zvol_check_volsize(volsize,
3219                     volblocksize)) != 0) {
3220                 volblocksize) != 0) {
3221                     nvlist_free(nvprops);
3222                     return (error);
3223                 }

```

```

3171     } else if (type == DMU_OST_ZFS) {
3172         int error;
3173
3174         /*
3175          * We have to have normalization and
3176          * case-folding flags correct when we do the
3177          * file system creation, so go figure them out
3178          * now.
3179          */
3180         VERIFY(nvlist_alloc(&zct.zct_zplprops,
3181             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3182         error = zfs_fill_zplprops(fsname, nvprops,
3183             error = zfs_fill_zplprops(zc->zc_name, nvprops,
3184                 zct.zct_zplprops, &is_insensitive);
3185             if (error != 0) {
3186                 nvlist_free(nvprops);
3187                 nvlist_free(zct.zct_zplprops);
3188                 return (error);
3189             }
3190
3191         error = dmu_objset_create(fsname, type,
3192             error = dmu_objset_create(zc->zc_name, type,
3193                 is_insensitive ? DS_FLAG_CI_DATASET : 0, cbfunc, &zct);
3194             nvlist_free(zct.zct_zplprops);
3195             }
3196
3197         /*
3198          * It would be nice to do this atomically.
3199          */
3200         if (error == 0) {
3201             error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3202                 nvprops, outnvl);
3203             error = zfs_set_prop_nvlist(zc->zc_name, ZPROP_SRC_LOCAL,
3204                 nvprops, NULL);
3205             if (error != 0)
3206                 (void) dmu_objset_destroy(fsname, B_FALSE);
3207             (void) dmu_objset_destroy(zc->zc_name, B_FALSE);
3208         }
3209         nvlist_free(nvprops);
3210         return (error);
3211     }
3212 }
3213
3214 /*
3215  * innvl: {
3216  *     "origin" -> name of origin snapshot
3217  *     (optional) "props" -> { prop -> value }
3218  * }
3219  * inputs:
3220  *   zc_name      name of filesystem
3221  *   zc_value     short name of snapshot
3222  *   zc_cookie    recursive flag
3223  *   zc_nvlist_src[_size] property list
3224  *
3225  * outnvl: propname -> error code (int32)
3226  * outputs:
3227  *   zc_value     short snapname (i.e. part after the '@')
3228  */
3229 static int
3230 zfs_ioc_clone(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3231 {
3232     zfs_ioc_snapshot(zfs_cmd_t *zc)
3233     {
3234         int error = 0;
3235         #endif /* !codereview */
3236         nvlist_t *nvprops = NULL;
3237         char *origin_name;

```

```

3221     dsl_dataset_t *origin;
3222
3223     if (nvlist_lookup_string(innvl, "origin", &origin_name) != 0)
3224         return (EINVAL);
3225     (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);
3226
3227     if (strchr(fsname, '@') ||
3228         strchr(fsname, '%'))
3229         return (EINVAL);
3230     int error;
3231     boolean_t recursive = zc->zc_cookie;
3232
3233     if (dataset_namecheck(origin_name, NULL, NULL) != 0)
3234         if (snapshot_namecheck(zc->zc_value, NULL, NULL) != 0)
3235             return (EINVAL);
3236
3237     error = dsl_dataset_hold(origin_name, FTAG, &origin);
3238     if (error)
3239         if (zc->zc_nvlist_src != NULL &&
3240             (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
3241                 zc->zc_iflags, &nvprops)) != 0)
3242             return (error);
3243
3244     error = dmub_objset_clone(fsname, origin, 0);
3245     dsl_dataset_rele(origin, FTAG);
3246     error = zfs_check_userprops(zc->zc_name, nvprops);
3247     if (error)
3248         return (error);
3249
3250     /*
3251      * It would be nice to do this atomically.
3252      */
3253     if (error == 0) {
3254         error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3255             nvprops, outnvl);
3256         if (error != 0)
3257             (void) dmub_objset_destroy(fsname, B_FALSE);
3258     }
3259     return (error);
3260 }
3261
3262 /*
3263 * innvl: {
3264 *     "snaps" -> { snapshot1, snapshot2 }
3265 *     (optional) "props" -> { prop -> value (string) }
3266 * }
3267 * outnvl: snapshot -> error code (int32)
3268 */
3269 static int
3270 zfs_ioc_snapshot(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3271 {
3272     nvlist_t *snaps;
3273     nvlist_t *props = NULL;
3274     int error, poollen;
3275     nvpair_t *pair;
3276
3277     (void) nvlist_lookup_nvlist(innvl, "props", &props);
3278     if ((error = zfs_check_userprops(poolname, props)) != 0)
3279         return (error);
3280
3281     if (!nvlist_empty(props) &&
3282         zfs_earlier_version(poolname, SPA_VERSION_SNAP_PROPS))
3283         return (ENOTSUP);

```

```

3284     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3285         return (EINVAL);
3286     poollen = strlen(poolname);
3287     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3288         pair = nvlist_next_nvpair(snaps, pair)) {
3289         const char *name = nvpair_name(pair);
3290         const char *cp = strchr(name, '@');
3291
3292         /*
3293          * The snap name must contain an @, and the part after it must
3294          * contain only valid characters.
3295          */
3296         if (cp == NULL || snapshot_namecheck(cp + 1, NULL, NULL) != 0)
3297             return (EINVAL);
3298
3299         /*
3300          * The snap must be in the specified pool.
3301          */
3302         if (strncmp(name, poolname, poollen) != 0 ||
3303             (name[poollen] != '/' && name[poollen] != '@'))
3304             return (EXDEV);
3305
3306         /* This must be the only snap of this fs. */
3307         for (nvpair_t *pair2 = nvlist_next_nvpair(snaps, pair);
3308             pair2 != NULL; pair2 = nvlist_next_nvpair(snaps, pair2)) {
3309             if (strcmp(name, nvpair_name(pair2), cp - name + 1)
3310                 == 0) {
3311                 return (EXDEV);
3312             }
3313         }
3314     }
3315
3316     error = dmub_objset_snapshot(snaps, props, outnvl);
3317     return (error);
3318 }
3319
3320 /*
3321 * innvl: "message" -> string
3322 */
3323 /* ARGSUSED */
3324 static int
3325 zfs_ioc_log_history(const char *unused, nvlist_t *innvl, nvlist_t *outnvl)
3326 {
3327     char *message;
3328     spa_t *spa;
3329     int error;
3330     char *poolname;
3331
3332     /*
3333      * The poolname in the ioctl is not set, we get it from the TSD,
3334      * which was set at the end of the last successful ioctl that allows
3335      * logging. The secpolicy func already checked that it is set.
3336      * Only one log ioctl is allowed after each successful ioctl, so
3337      * we clear the TSD here.
3338      */
3339     poolname = tsd_get(zfs_allow_log_key);
3340     (void) tsd_set(zfs_allow_log_key, NULL);
3341     error = spa_open(poolname, &spa, FTAG);
3342     strfree(poolname);
3343     if (error != 0)
3344         return (error);
3345     goto out;
3346
3347     if (nvlist_lookup_string(innvl, "message", &message) != 0) {
3348         spa_close(spa, FTAG);
3349         return (EINVAL);

```

```

2986     if (!nvlist_empty(nvprops) &&
2987         zfs_earlier_version(zc->zc_name, SPA_VERSION_SNAP_PROPS)) {
2988         error = ENOTSUP;
2989         goto out;
3345     }

3347     if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
3348         spa_close(spa, FTAG);
3349         return (ENOTSUP);
3350     }
2992     error = dmu_objset_snapshot(zc->zc_name, zc->zc_value, NULL,
2993         nvprops, recursive, B_FALSE, -1);

3352     error = spa_history_log(spa, message);
3353     spa_close(spa, FTAG);
2995 out:
2996     nvlist_free(nvprops);
3354     return (error);
3355 }

3357 /* ARGSUSED */
3358 #endif /* ! codereview */
3359 int
3360 zfs_unmount_snap(const char *name, void *arg)
3361 {
3362     vfs_t *vfsp;
3363     int err;

3365     if (strchr(name, '@') == NULL)
3366         return (0);
3000     vfs_t *vfsp = NULL;

3002     if (arg) {
3003         char *snapname = arg;
3004         char *fullname = kmem_asprintf("%s%s", name, snapname);
3005         vfsp = zfs_get_vfs(fullname);
3006         strfree(fullname);
3007     } else if (strchr(name, '@')) {
3368         vfsp = zfs_get_vfs(name);
3369         if (vfsp == NULL)
3370             return (0);
3009     }

3011     if (vfsp) {
3012         /*
3013          * Always force the unmount for snapshots.
3014          */
3015         int flag = MS_FORCE;
3016         int err;

3372         if ((err = vn_vfswlock(vfsp->vfs_vnodecovered)) != 0) {
3373             VFS_RELE(vfsp);
3374             return (err);
3375         }
3376         VFS_RELE(vfsp);

3378         /*
3379          * Always force the unmount for snapshots.
3380          */
3381         return (dounmount(vfsp, MS_FORCE, kcred));
3023         if ((err = dounmount(vfsp, flag, kcred)) != 0)
3024             return (err);
3025     }
3026     return (0);
3382 }

```

```

3384 /*
3385  * innvl: {
3386  *     "snaps" -> { snapshot1, snapshot2 }
3387  *     (optional boolean) "defer"
3388  * }
3389  *
3390  * outnvl: snapshot -> error code (int32)
3391  *
3392  * inputs:
3393  *   zc_name           name of filesystem, snaps must be under it
3394  *   zc_nvlist_src[_size] full names of snapshots to destroy
3395  *   zc_defer_destroy mark for deferred destroy
3396  *
3397  * outputs:
3398  *   zc_name           on failure, name of failed snapshot
3399  */
3393 static int
3394 zfs_ioc_destroy_snaps(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3395 {
3396     int poollen;
3397     nvlist_t *snaps;
3401     int err, len;
3402     nvlist_t *nvl;
3398     nvpair_t *pair;
3399     boolean_t defer;
3400 #endif /* ! codereview */

3402     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3403         return (EINVAL);
3404     defer = nvlist_exists(innvl, "defer");
3404     if ((err = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
3405         zc->zc_iflags, &nvl)) != 0)
3406         return (err);

3406     poollen = strlen(poolname);
3407     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3408         pair = nvlist_next_nvpair(snaps, pair)) {
3409         len = strlen(zc->zc_name);
3409         for (pair = nvlist_next_nvpair(nvl, NULL); pair != NULL;
3405             pair = nvlist_next_nvpair(nvl, pair)) {
3409             const char *name = nvpair_name(pair);

3411 #endif /* ! codereview */
3412         /*
3413          * The snap must be in the specified pool.
3414          * The snap name must be underneath the zc_name. This ensures
3415          * that our permission checks were legitimate.
3416          */
3417         if (strncmp(name, poolname, poollen) != 0 ||
3416             (name[poollen] != '/' && name[poollen] != '@'))
3417             return (EXDEV);
3417         if (strncmp(zc->zc_name, name, len) != 0 ||
3405             (name[len] != '@' && name[len] != '/')) {
3405             nvlist_free(nvl);
3405             return (EINVAL);
3409         }

3419         /*
3420          * Ignore failures to unmount; dmu_snapshots_destroy_nvlist()
3421          * will deal with this gracefully (by filling in outnvl).
3422          */
3423 #endif /* ! codereview */
3424         (void) zfs_unmount_snap(name, NULL);
3425     }

3427     return (dmu_snapshots_destroy_nvlist(snaps, defer, outnvl));

```



```

3061     err = dmu_snapshots_destroy_nvlist(nvlist, zc->zc_defer_destroy,
3062         zc->zc_name);
3063     nvlist_free(nvlist);
3064     return (err);
3428 }
    unchanged_portion_omitted

3696 /*
3697  * Removes properties from the given props list that fail permission checks
3698  * needed to clear them and to restore them in case of a receive error. For each
3699  * property, make sure we have both set and inherit permissions.
3700  *
3701  * Returns the first error encountered if any permission checks fail. If the
3702  * caller provides a non-NULL errlist, it also gives the complete list of names
3703  * of all the properties that failed a permission check along with the
3704  * corresponding error numbers. The caller is responsible for freeing the
3705  * returned errlist.
3706  *
3707  * If every property checks out successfully, zero is returned and the list
3708  * pointed at by errlist is NULL.
3709  */
3710 static int
3711 zfs_check_clearable(char *dataset, nvlist_t *props, nvlist_t **errlist)
3712 {
3713     zfs_cmd_t *zc;
3714     nvpair_t *pair, *next_pair;
3715     nvlist_t *errors;
3716     int err, rv = 0;

3718     if (props == NULL)
3719         return (0);

3721     VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

3723     zc = kmem_alloc(sizeof (zfs_cmd_t), KM_SLEEP);
3724     (void) strcpy(zc->zc_name, dataset);
3725     pair = nvlist_next_nvpair(props, NULL);
3726     while (pair != NULL) {
3727         next_pair = nvlist_next_nvpair(props, pair);

3729         (void) strcpy(zc->zc_value, nvpair_name(pair));
3730         if ((err = zfs_check_settable(dataset, pair, CRED())) != 0 ||
3731             (err = zfs_secpolicy_inherit_prop(zc, NULL, CRED())) != 0) {
3732             (err = zfs_secpolicy_inherit(zc, CRED())) != 0) {
3732                 VERIFY(nvlist_remove_nvpair(props, pair) == 0);
3733                 VERIFY(nvlist_add_int32(errors,
3734                     zc->zc_value, err) == 0);
3735             }
3736             pair = next_pair;
3737         }
3738         kmem_free(zc, sizeof (zfs_cmd_t));

3740     if ((pair = nvlist_next_nvpair(errors, NULL)) == NULL) {
3741         nvlist_free(errors);
3742         errors = NULL;
3743     } else {
3744         VERIFY(nvpair_value_int32(pair, &rv) == 0);
3745     }

3747     if (errlist == NULL)
3748         nvlist_free(errors);
3749     else
3750         *errlist = errors;

3752     return (rv);
3753 }
    unchanged_portion_omitted

```

```

3824 #ifdef  DEBUG
3825 static boolean_t zfs_ioc_recv_inject_err;
3826 #endif

3828 /*
3829  * inputs:
3830  * zc_name             name of containing filesystem
3831  * zc_nvlist_src[_size] nvlist of properties to apply
3832  * zc_value            name of snapshot to create
3833  * zc_string           name of clone origin (if DRR_FLAG_CLONE)
3834  * zc_cookie           file descriptor to recv from
3835  * zc_begin_record    the BEGIN record of the stream (not byteswapped)
3836  * zc_guid             force flag
3837  * zc_cleanup_fd       cleanup-on-exit file descriptor
3838  * zc_action_handle    handle for this guid/ds mapping (or zero on first call)
3839  *
3840  * outputs:
3841  * zc_cookie           number of bytes read
3842  * zc_nvlist_dst[_size] error for each unapplied received property
3843  * zc_obj              zprop_errflags_t
3844  * zc_action_handle    handle for this guid/ds mapping
3845  */
3846 static int
3847 zfs_ioc_recv(zfs_cmd_t *zc)
3848 {
3849     file_t *fp;
3850     objset_t *os;
3851     dmu_recv_cookie_t drc;
3852     boolean_t force = (boolean_t)zc->zc_guid;
3853     int fd;
3854     int error = 0;
3855     int props_error = 0;
3856     nvlist_t *errors;
3857     offset_t off;
3858     nvlist_t *props = NULL; /* sent properties */
3859     nvlist_t *origprops = NULL; /* existing properties */
3860     objset_t *origin = NULL;
3861     char *tosnap;
3862     char tofs[ZFS_MAXNAMELEN];
3863     boolean_t first_recvd_props = B_FALSE;

3865     if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
3866         strchr(zc->zc_value, '@') == NULL ||
3867         strchr(zc->zc_value, '%'))
3868         return (EINVAL);

3870     (void) strcpy(tofs, zc->zc_value);
3871     tosnap = strchr(tofs, '@');
3872     *tosnap++ = '\0';

3874     if (zc->zc_nvlist_src != NULL &&
3875         (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
3876             zc->zc_iflags, &props)) != 0)
3877         return (error);

3879     fd = zc->zc_cookie;
3880     fp = getf(fd);
3881     if (fp == NULL) {
3882         nvlist_free(props);
3883         return (EBADF);
3884     }

3886     VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

3888     if (props && dmu_objset_hold(tofs, FTAG, &os) == 0) {

```

```

3889     if ((spa_version(os->os_spa) >= SPA_VERSION_RECVD_PROPS) &&
3890         !dsl_prop_get_hasrecvd(os)) {
3891         first_recvd_props = B_TRUE;
3892     }
3893
3894     /*
3895     * If new received properties are supplied, they are to
3896     * completely replace the existing received properties, so stash
3897     * away the existing ones.
3898     */
3899     if (dsl_prop_get_received(os, &origprops) == 0) {
3900         nvlist_t *errlist = NULL;
3901         /*
3902         * Don't bother writing a property if its value won't
3903         * change (and avoid the unnecessary security checks).
3904         *
3905         * The first receive after SPA_VERSION_RECVD_PROPS is a
3906         * special case where we blow away all local properties
3907         * regardless.
3908         */
3909         if (!first_recvd_props)
3910             props_reduce(props, origprops);
3911         if (zfs_check_clearable(tofs, origprops,
3912             &errlist) != 0)
3913             (void) nvlist_merge(errors, errlist, 0);
3914         nvlist_free(errlist);
3915     }
3916
3917     dmu_objset_rele(os, FTAG);
3918 }
3919
3920 if (zc->zc_string[0]) {
3921     error = dmu_objset_hold(zc->zc_string, FTAG, &origin);
3922     if (error)
3923         goto out;
3924 }
3925
3926 error = dmu_recv_begin(tofs, tosnap, zc->zc_top_ds,
3927     &zc->zc_begin_record, force, origin, &drc);
3928 if (origin)
3929     dmu_objset_rele(origin, FTAG);
3930 if (error)
3931     goto out;
3932
3933 /*
3934 * Set properties before we receive the stream so that they are applied
3935 * to the new data. Note that we must call dmu_recv_stream() if
3936 * dmu_recv_begin() succeeds.
3937 */
3938 if (props) {
3939     nvlist_t *errlist;
3940
3941     if (dmu_objset_from_ds(drc.drc_logical_ds, &os) == 0) {
3942         if (drc.drc_newfs) {
3943             if (spa_version(os->os_spa) >=
3944                 SPA_VERSION_RECVD_PROPS)
3945                 first_recvd_props = B_TRUE;
3946             } else if (origprops != NULL) {
3947                 if (clear_received_props(os, tofs, origprops,
3948                     first_recvd_props ? NULL : props) != 0)
3949                     zc->zc_obj |= ZPROP_ERR_NOCLEAR;
3950             } else {
3951                 zc->zc_obj |= ZPROP_ERR_NOCLEAR;
3952             }
3953         }
3954         dsl_prop_set_hasrecvd(os);
3955     } else if (!drc.drc_newfs) {

```

```

3953         zc->zc_obj |= ZPROP_ERR_NOCLEAR;
3954     }
3955
3956     (void) zfs_set_prop_nvlist(tofs, ZPROP_SRC_RECEIVED,
3957         props, errors);
3958     (void) nvlist_merge(errors, errlist, 0);
3959     nvlist_free(errlist);
3960 }
3961
3962 if (zc->zc_nvlist_dst_size != 0 &&
3963     (nvlist_smush(errors, zc->zc_nvlist_dst_size) != 0 ||
3964     put_nvlist(zc, errors) != 0)) {
3965     if (fit_error_list(zc, &errors) != 0 || put_nvlist(zc, errors) != 0) {
3966         /*
3967         * Caller made zc->zc_nvlist_dst less than the minimum expected
3968         * size or supplied an invalid address.
3969         */
3970         props_error = EINVAL;
3971     }
3972
3973     off = fp->f_offset;
3974     error = dmu_recv_stream(&drc, fp->f_vnode, &off, zc->zc_cleanup_fd,
3975         &zc->zc_action_handle);
3976
3977     if (error == 0) {
3978         zfsvfs_t *zfsvfs = NULL;
3979
3980         if (getzfsvfs(tofs, &zfsvfs) == 0) {
3981             /* online recv */
3982             int end_err;
3983
3984             error = zfs_suspend_fs(zfsvfs);
3985             /*
3986             * If the suspend fails, then the recv_end will
3987             * likely also fail, and clean up after itself.
3988             */
3989             end_err = dmu_recv_end(&drc);
3990             if (error == 0)
3991                 error = zfs_resume_fs(zfsvfs, tofs);
3992             error = error ? error : end_err;
3993             VFS_RELE(zfsvfs->z_vfs);
3994         } else {
3995             error = dmu_recv_end(&drc);
3996         }
3997     }
3998
3999     zc->zc_cookie = off - fp->f_offset;
4000     if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4001         fp->f_offset = off;
4002
4003 #ifdef DEBUG
4004     if (zfs_ioc_recv_inject_err) {
4005         zfs_ioc_recv_inject_err = B_FALSE;
4006         error = 1;
4007     }
4008 #endif
4009     /*
4010     * On error, restore the original props.
4011     */
4012     if (error && props) {
4013         if (dmu_objset_hold(tofs, FTAG, &os) == 0) {
4014             if (clear_received_props(os, tofs, props, NULL) != 0) {
4015                 /*
4016                 * We failed to clear the received properties.
4017                 * Since we may have left a $recvd value on the

```

```

4015         * system, we can't clear the $hasrecvd flag.
4016         */
4017         zc->zc_obj |= ZPROP_ERR_NOESTORE;
4018     } else if (first_recvd_props) {
4019         dsl_prop_unset_hasrecvd(os);
4020     }
4021     dmu_objset_rele(os, FTAG);
4022 } else if (!drc.drc_newfs) {
4023     /* We failed to clear the received properties. */
4024     zc->zc_obj |= ZPROP_ERR_NOESTORE;
4025 }
4026
4027 if (origprops == NULL && !drc.drc_newfs) {
4028     /* We failed to stash the original properties. */
4029     zc->zc_obj |= ZPROP_ERR_NOESTORE;
4030 }
4031
4032 /*
4033  * dsl_props_set() will not convert RECEIVED to LOCAL on or
4034  * after SPA_VERSION_RECVD_PROPS, so we need to specify LOCAL
4035  * explicitly if we're restoring local properties cleared in the
4036  * first new-style receive.
4037  */
4038 if (origprops != NULL &&
4039     zfs_set_prop_nvlist(tofs, (first_recvd_props ?
4040     ZPROP_SRC_LOCAL : ZPROP_SRC_RECEIVED),
4041     origprops, NULL) != 0) {
4042     /*
4043      * We stashed the original properties but failed to
4044      * restore them.
4045      */
4046     zc->zc_obj |= ZPROP_ERR_NOESTORE;
4047 }
4048 }
4049 out:
4050     nvlist_free(props);
4051     nvlist_free(origprops);
4052     nvlist_free(errors);
4053     releasef(fd);
4054
4055     if (error == 0)
4056         error = props_error;
4057
4058     return (error);
4059 }
4060
4061 /*
4062  * inputs:
4063  * zc_name     name of snapshot to send
4064  * zc_cookie   file descriptor to send stream to
4065  * zc_obj      fromorigin flag (mutually exclusive with zc_fromobj)
4066  * zc_sendobj  objsetid of snapshot to send
4067  * zc_fromobj  objsetid of incremental fromsnap (may be zero)
4068  * zc_guid     if set, estimate size of stream only.  zc_cookie is ignored.
4069  *             output size in zc_objset_type.
4070  *
4071  * outputs: none
4072  */
4073 static int
4074 zfs_ioc_send(zfs_cmd_t *zc)
4075 {
4076     objset_t *fromsnap = NULL;
4077     objset_t *tosnap;
4078     int error;
4079     offset_t off;
4080     dsl_dataset_t *ds;

```

```

4081     dsl_dataset_t *dsfrom = NULL;
4082     spa_t *spa;
4083     dsl_pool_t *dp;
4084     boolean_t estimate = (zc->zc_guid != 0);
4085
4086     error = spa_open(zc->zc_name, &spa, FTAG);
4087     if (error)
4088         return (error);
4089
4090     dp = spa_get_dsl(spa);
4091     rw_enter(&dp->dp_config_rwlock, RW_READER);
4092     error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
4093     rw_exit(&dp->dp_config_rwlock);
4094     if (error) {
4095         spa_close(spa, FTAG);
4096         if (error)
4097             #endif /* !codereview */
4098             return (error);
4099     }
4100
4101     error = dmu_objset_from_ds(ds, &tosnap);
4102     if (error) {
4103         dsl_dataset_rele(ds, FTAG);
4104         spa_close(spa, FTAG);
4105         return (error);
4106     }
4107
4108     if (zc->zc_fromobj != 0) {
4109         rw_enter(&dp->dp_config_rwlock, RW_READER);
4110         error = dsl_dataset_hold_obj(dp, zc->zc_fromobj, FTAG, &dsfrom);
4111         rw_exit(&dp->dp_config_rwlock);
4112         spa_close(spa, FTAG);
4113         if (error) {
4114             dsl_dataset_rele(ds, FTAG);
4115             return (error);
4116         }
4117         error = dmu_objset_from_ds(dsfrom, &fromsnap);
4118         if (error) {
4119             dsl_dataset_rele(dsfrom, FTAG);
4120             dsl_dataset_rele(ds, FTAG);
4121             return (error);
4122         }
4123     }
4124
4125     if (zc->zc_obj) {
4126         dsl_pool_t *dp = ds->ds_dir->dd_pool;
4127
4128         if (fromsnap != NULL) {
4129             dsl_dataset_rele(dsfrom, FTAG);
4130             dsl_dataset_rele(ds, FTAG);
4131             return (EINVAL);
4132         }
4133
4134         if (dsl_dir_is_clone(ds->ds_dir)) {
4135             rw_enter(&dp->dp_config_rwlock, RW_READER);
4136             error = dsl_dataset_hold_obj(dp,
4137             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &dsfrom);
4138             rw_exit(&dp->dp_config_rwlock);
4139             if (error) {
4140                 dsl_dataset_rele(ds, FTAG);
4141                 return (error);
4142             }
4143             error = dmu_objset_from_ds(dsfrom, &fromsnap);
4144             if (error) {
4145                 dsl_dataset_rele(dsfrom, FTAG);
4146                 dsl_dataset_rele(ds, FTAG);

```

```

4143         return (error);
4144     }
4145 } else {
3759     spa_close(spa, FTAG);
3760 }
4146
4148 if (estimate) {
4149     error = dmu_send_estimate(tosnap, fromsnap,
3764     error = dmu_send_estimate(tosnap, fromsnap, zc->zc_obj,
4150     &zc->zc_objset_type);
4151 } else {
4152     file_t *fp = getf(zc->zc_cookie);
4153     if (fp == NULL) {
4154         dsl_dataset_rele(ds, FTAG);
4155         if (dsfrom)
4156             dsl_dataset_rele(dsfrom, FTAG);
4157         return (EBADF);
4158     }
4160     off = fp->f_offset;
4161     error = dmu_send(tosnap, fromsnap,
3776     error = dmu_send(tosnap, fromsnap, zc->zc_obj,
4162     zc->zc_cookie, fp->f_vnode, &off);
4164     if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4165         fp->f_offset = off;
4166     releasef(zc->zc_cookie);
4167 }
4168 if (dsfrom)
4169     dsl_dataset_rele(dsfrom, FTAG);
4170 dsl_dataset_rele(ds, FTAG);
4171 return (error);
4172 }

```

unchanged portion omitted

```

4668 /*
4669 * inputs:
4670 * zc_name          name of filesystem
4671 * zc_value         prefix name for snapshot
4672 * zc_cleanup_fd   cleanup-on-exit file descriptor for calling process
4673 *
4674 * outputs:
4675 * zc_value         short name of new snapshot
4676 #endif /* ! codereview */
4677 */
4678 static int
4679 zfs_ioc_tmp_snapshot(zfs_cmd_t *zc)
4680 {
4681     char *snap_name;
4682     int error;
4684     snap_name = kmem_asprintf("%s%-016llx", zc->zc_name, zc->zc_value,
4290     snap_name = kmem_asprintf("%s%-016llx", zc->zc_value,
4685     (u_longlong_t)ddi_get_lbolt64());
4687     if (strlen(snap_name) >= MAXPATHLEN) {
4293     if (strlen(snap_name) >= MAXNAMELEN) {
4688         strfree(snap_name);
4689         return (E2BIG);
4690     }
4692     error = dmu_objset_snapshot_tmp(snap_name, "%temp", zc->zc_cleanup_fd);
4298     error = dmu_objset_snapshot(zc->zc_name, snap_name,
4299     NULL, B_FALSE, B_TRUE, zc->zc_cleanup_fd);
4693     if (error != 0) {

```

```

4694         strfree(snap_name);
4695         return (error);
4696     }
4698     (void) strcpy(zc->zc_value, strchr(snap_name, '@') + 1);
4305     (void) strcpy(zc->zc_value, snap_name);
4699     strfree(snap_name);
4700     return (0);
4701 }

```

unchanged portion omitted

```

5052 /*
5053 * innvl: {
5054 *     "firstsnap" -> snapshot name
5055 * }
5056 *
5057 * outnvl: {
5058 *     "used" -> space in bytes
5059 *     "compressed" -> compressed space in bytes
5060 *     "uncompressed" -> uncompressed space in bytes
5061 * }
4661 * inputs:
4662 * zc_name          full name of last snapshot
4663 * zc_value         full name of first snapshot
4664 *
4665 * outputs:
4666 * zc_cookie        space in bytes
4667 * zc_objset_type   compressed space in bytes
4668 * zc_perm_action   uncompressed space in bytes
5062 */
5063 static int
5064 zfs_ioc_space_snaps(const char *lastsnap, nvlist_t *innvl, nvlist_t *outnvl)
4671 zfs_ioc_space_snaps(zfs_cmd_t *zc)
5065 {
5066     int error;
5067     dsl_dataset_t *new, *old;
5068     char *firstsnap;
5069     uint64_t used, comp, uncomp;
5071     if (nvlist_lookup_string(innvl, "firstsnap", &firstsnap) != 0)
5072         return (EINVAL);
5073 #endif /* ! codereview */
5075     error = dsl_dataset_hold(lastsnap, FTAG, &new);
4675     error = dsl_dataset_hold(zc->zc_name, FTAG, &new);
5076     if (error != 0)
5077         return (error);
5078     error = dsl_dataset_hold(firstsnap, FTAG, &old);
4678     error = dsl_dataset_hold(zc->zc_value, FTAG, &old);
5079     if (error != 0) {
5080         dsl_dataset_rele(new, FTAG);
5081         return (error);
5082     }
5084     error = dsl_dataset_space_wouldfree(old, new, &used, &comp, &uncomp);
4684     error = dsl_dataset_space_wouldfree(old, new, &zc->zc_cookie,
4685     &zc->zc_objset_type, &zc->zc_perm_action);
5085     dsl_dataset_rele(old, FTAG);
5086     dsl_dataset_rele(new, FTAG);
5087     fnvlist_add_uint64(outnvl, "used", used);
5088     fnvlist_add_uint64(outnvl, "compressed", comp);
5089     fnvlist_add_uint64(outnvl, "uncompressed", uncomp);
5090 #endif /* ! codereview */
5091     return (error);
5092 }

```

```

5094 /*
5095  * innvl: {
5096  *   "fd" -> file descriptor to write stream to (int32)
5097  *   (optional) "fromsnap" -> full snap name to send an incremental from
5098  * }
5099  *
5100  * outnvl is unused
5101  */
5102 /* ARGSUSED */
5103 static int
5104 zfs_ioc_send_new(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5105 {
5106     objset_t *fromsnap = NULL;
5107     objset_t *tosnap;
5108     int error;
5109     offset_t off;
5110     char *fromname;
5111     int fd;
5112
5113     error = nvlist_lookup_int32(innvl, "fd", &fd);
5114     if (error != 0)
5115         return (EINVAL);
5116
5117     error = dmub_objset_hold(snapname, FTAG, &tosnap);
5118     if (error)
5119         return (error);
5120
5121     error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
5122     if (error == 0) {
5123         error = dmub_objset_hold(fromname, FTAG, &fromsnap);
5124         if (error) {
5125             dmub_objset_rele(tosnap, FTAG);
5126             return (error);
5127         }
5128     }
5129
5130     file_t *fp = getf(fd);
5131     if (fp == NULL) {
5132         dmub_objset_rele(tosnap, FTAG);
5133         if (fromsnap != NULL)
5134             dmub_objset_rele(fromsnap, FTAG);
5135         return (EBADF);
5136     }
5137
5138     off = fp->f_offset;
5139     error = dmub_send(tosnap, fromsnap, fd, fp->f_vnode, &off);
5140
5141     if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
5142         fp->f_offset = off;
5143     releasef(fd);
5144     if (fromsnap != NULL)
5145         dmub_objset_rele(fromsnap, FTAG);
5146     dmub_objset_rele(tosnap, FTAG);
5147     return (error);
5148 }
5149
5150 /*
5151  * Determine approximately how large a zfs send stream will be -- the number
5152  * of bytes that will be written to the fd supplied to zfs_ioc_send_new().
5153  *
5154  * innvl: {
5155  *   (optional) "fromsnap" -> full snap name to send an incremental from
5156  * }
5157  *
5158  * outnvl: {
5159  *   "space" -> bytes of space (uint64)

```

```

5160  * }
5161  */
5162 static int
5163 zfs_ioc_send_space(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5164 {
5165     objset_t *fromsnap = NULL;
5166     objset_t *tosnap;
5167     int error;
5168     char *fromname;
5169     uint64_t space;
5170
5171     error = dmub_objset_hold(snapname, FTAG, &tosnap);
5172     if (error)
5173         return (error);
5174
5175     error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
5176     if (error == 0) {
5177         error = dmub_objset_hold(fromname, FTAG, &fromsnap);
5178         if (error) {
5179             dmub_objset_rele(tosnap, FTAG);
5180             return (error);
5181         }
5182     }
5183
5184     error = dmub_send_estimate(tosnap, fromsnap, &space);
5185     fnvlist_add_uint64(outnvl, "space", space);
5186
5187     if (fromsnap != NULL)
5188         dmub_objset_rele(fromsnap, FTAG);
5189     dmub_objset_rele(tosnap, FTAG);
5190     return (error);
5191 }
5192
5193 static zfs_ioc_vec_t zfs_ioc_vec[ZFS_IOC_LAST - ZFS_IOC_FIRST];
5194
5195 static void
5196 zfs_ioctl_register_legacy(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5197     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5198     boolean_t log_history, zfs_ioc_poolcheck_t pool_check)
5199 {
5200     zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];
5201
5202     ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5203     ASSERT3U(ioc, <, ZFS_IOC_LAST);
5204     ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5205     ASSERT3P(vec->zvec_func, ==, NULL);
5206
5207     vec->zvec_legacy_func = func;
5208     vec->zvec_secpolicy = secpolicy;
5209     vec->zvec_namecheck = namecheck;
5210     vec->zvec_allow_log = log_history;
5211     vec->zvec_pool_check = pool_check;
5212 }
5213
5214 /*
5215  * See the block comment at the beginning of this file for details on
5216  * each argument to this function.
5217  */
5218 static void
5219 zfs_ioctl_register(const char *name, zfs_ioc_t ioc, zfs_ioc_func_t *func,
5220     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5221     zfs_ioc_poolcheck_t pool_check, boolean_t smush_outnvl,
5222     boolean_t allow_log)
5223 {
5224     zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];

```

```

5227     ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5228     ASSERT3U(ioc, <, ZFS_IOC_LAST);
5229     ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5230     ASSERT3P(vec->zvec_func, ==, NULL);

5232     /* if we are logging, the name must be valid */
5233     ASSERT(!allow_log || namecheck != NO_NAME);

5235     vec->zvec_name = name;
5236     vec->zvec_func = func;
5237     vec->zvec_secpolicy = secpolicy;
5238     vec->zvec_namecheck = namecheck;
5239     vec->zvec_pool_check = pool_check;
5240     vec->zvec_smush_outnvlst = smush_outnvlst;
5241     vec->zvec_allow_log = allow_log;
5242 }

5244 static void
5245 zfs_ioctl_register_pool(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5246     zfs_secpolicy_func_t *secpolicy, boolean_t log_history,
5247     zfs_ioc_poolcheck_t pool_check)
5248 {
5249     zfs_ioctl_register_legacy(ioc, func, secpolicy,
5250         POOL_NAME, log_history, pool_check);
5251 }

5253 static void
5254 zfs_ioctl_register_dataset_nolog(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5255     zfs_secpolicy_func_t *secpolicy, zfs_ioc_poolcheck_t pool_check)
5256 {
5257     zfs_ioctl_register_legacy(ioc, func, secpolicy,
5258         DATASET_NAME, B_FALSE, pool_check);
5259 }

5261 static void
5262 zfs_ioctl_register_pool_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5263 {
5264     zfs_ioctl_register_legacy(ioc, func, zfs_secpolicy_config,
5265         POOL_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5266 }

5268 static void
5269 zfs_ioctl_register_pool_meta(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5270     zfs_secpolicy_func_t *secpolicy)
5271 {
5272     zfs_ioctl_register_legacy(ioc, func, secpolicy,
5273         NO_NAME, B_FALSE, POOL_CHECK_NONE);
5274 }

5276 static void
5277 zfs_ioctl_register_dataset_read_secpolicy(zfs_ioc_t ioc,
5278     zfs_ioc_legacy_func_t *func, zfs_secpolicy_func_t *secpolicy)
5279 {
5280     zfs_ioctl_register_legacy(ioc, func, secpolicy,
5281         DATASET_NAME, B_FALSE, POOL_CHECK_SUSPENDED);
5282 }

5284 static void
5285 zfs_ioctl_register_dataset_read(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5286 {
5287     zfs_ioctl_register_dataset_read_secpolicy(ioc, func,
5288         zfs_secpolicy_read);
5289 }

5291 static void

```

```

5292 zfs_ioctl_register_dataset_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5293     zfs_secpolicy_func_t *secpolicy)
5294 {
5295     zfs_ioctl_register_legacy(ioc, func, secpolicy,
5296         DATASET_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5297 }

5299 static void
5300 zfs_ioctl_init(void)
5301 {
5302     zfs_ioctl_register("snapshot", ZFS_IOC_SNAPSHOT,
5303         zfs_ioc_snapshot, zfs_secpolicy_snapshot, POOL_NAME,
5304         POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5306     zfs_ioctl_register("log_history", ZFS_IOC_LOG_HISTORY,
5307         zfs_ioc_log_history, zfs_secpolicy_log_history, NO_NAME,
5308         POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_FALSE);

5310     zfs_ioctl_register("space_snaps", ZFS_IOC_SPACE_SNAPS,
5311         zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME,
5312         POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5314     zfs_ioctl_register("send", ZFS_IOC_SEND_NEW,
5315         zfs_ioc_send_new, zfs_secpolicy_send_new, DATASET_NAME,
5316         POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5318     zfs_ioctl_register("send_space", ZFS_IOC_SEND_SPACE,
5319         zfs_ioc_send_space, zfs_secpolicy_read, DATASET_NAME,
5320         POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5322     zfs_ioctl_register("create", ZFS_IOC_CREATE,
5323         zfs_ioc_create, zfs_secpolicy_create_clone, DATASET_NAME,
5324         POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5326     zfs_ioctl_register("clone", ZFS_IOC_CLONE,
5327         zfs_ioc_clone, zfs_secpolicy_create_clone, DATASET_NAME,
5328         POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5330     zfs_ioctl_register("destroy_snaps", ZFS_IOC_DESTROY_SNAPS,
5331         zfs_ioc_destroy_snaps, zfs_secpolicy_destroy_snaps, POOL_NAME,
5332         POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5334     /* IOCTLS that use the legacy function signature */

5336     zfs_ioctl_register_legacy(ZFS_IOC_POOL_FREEZE, zfs_ioc_pool_freeze,
5337         zfs_secpolicy_config, NO_NAME, B_FALSE, POOL_CHECK_READONLY);

5339     zfs_ioctl_register_pool(ZFS_IOC_POOL_CREATE, zfs_ioc_pool_create,
5340         zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5341     zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SCAN,
5342         zfs_ioc_pool_scan);
5343     zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_UPGRADE,
5344         zfs_ioc_pool_upgrade);
5345     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ADD,
5346         zfs_ioc_vdev_add);
5347     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_REMOVE,
5348         zfs_ioc_vdev_remove);
5349     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SET_STATE,
5350         zfs_ioc_vdev_set_state);
5351     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ATTACH,
5352         zfs_ioc_vdev_attach);
5353     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_DETACH,
5354         zfs_ioc_vdev_detach);
5355     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETPATH,
5356         zfs_ioc_vdev_setpath);
5357     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETFRU,

```

```

5358     zfs_ioc_vdev_setfru);
5359     zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SET_PROPS,
5360     zfs_ioc_pool_set_props);
5361     zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SPLIT,
5362     zfs_ioc_vdev_split);
5363     zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_REGUID,
5364     zfs_ioc_pool_reguid);

5366     zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_CONFIGS,
5367     zfs_ioc_pool_configs, zfs_secpolicy_none);
5368     zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_TRYIMPORT,
5369     zfs_ioc_pool_tryimport, zfs_secpolicy_config);
5370     zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_FAULT,
5371     zfs_ioc_inject_fault, zfs_secpolicy_inject);
5372     zfs_ioctl_register_pool_meta(ZFS_IOC_CLEAR_FAULT,
5373     zfs_ioc_clear_fault, zfs_secpolicy_inject);
5374     zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_LIST_NEXT,
5375     zfs_ioc_inject_list_next, zfs_secpolicy_inject);

5377     /*
5378     * pool destroy, and export don't log the history as part of
5379     * zfsdev_ioctl, but rather zfs_ioc_pool_export
5380     * does the logging of those commands.
5381     */
5382     zfs_ioctl_register_pool(ZFS_IOC_POOL_DESTROY, zfs_ioc_pool_destroy,
5383     zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
5384     zfs_ioctl_register_pool(ZFS_IOC_POOL_EXPORT, zfs_ioc_pool_export,
5385     zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);

5387     zfs_ioctl_register_pool(ZFS_IOC_POOL_STATS, zfs_ioc_pool_stats,
5388     zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
5389     zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_PROPS, zfs_ioc_pool_get_props,
5390     zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);

5392     zfs_ioctl_register_pool(ZFS_IOC_ERROR_LOG, zfs_ioc_error_log,
5393     zfs_secpolicy_inject, B_FALSE, POOL_CHECK_SUSPENDED);
5394     zfs_ioctl_register_pool(ZFS_IOC_DSOBJ_TO_DSNAME,
5395     zfs_ioc_dsobj_to_dsname,
5396     zfs_secpolicy_diff, B_FALSE, POOL_CHECK_SUSPENDED);
5397     zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_HISTORY,
5398     zfs_ioc_pool_get_history,
5399     zfs_secpolicy_config, B_FALSE, POOL_CHECK_SUSPENDED);

5401     zfs_ioctl_register_pool(ZFS_IOC_POOL_IMPORT, zfs_ioc_pool_import,
5402     zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);

5404     zfs_ioctl_register_pool(ZFS_IOC_CLEAR, zfs_ioc_clear,
5405     zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
5406     zfs_ioctl_register_pool(ZFS_IOC_POOL_REOPEN, zfs_ioc_pool_reopen,
5407     zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);

5409     zfs_ioctl_register_dataset_read(ZFS_IOC_SPACE_WRITTEN,
5410     zfs_ioc_space_written);
5411     zfs_ioctl_register_dataset_read(ZFS_IOC_GET_HOLDS,
5412     zfs_ioc_get_holds);
5413     zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_RECVD_PROPS,
5414     zfs_ioc_objset_recvd_props);
5415     zfs_ioctl_register_dataset_read(ZFS_IOC_NEXT_OBJ,
5416     zfs_ioc_next_obj);
5417     zfs_ioctl_register_dataset_read(ZFS_IOC_GET_FSACL,
5418     zfs_ioc_get_fsacl);
5419     zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_STATS,
5420     zfs_ioc_objset_stats);
5421     zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_ZPLPROPS,
5422     zfs_ioc_objset_zplprops);
5423     zfs_ioctl_register_dataset_read(ZFS_IOC_DATASET_LIST_NEXT,

```

```

5424     zfs_ioc_dataset_list_next);
5425     zfs_ioctl_register_dataset_read(ZFS_IOC_SNAPSHOT_LIST_NEXT,
5426     zfs_ioc_snapshot_list_next);
5427     zfs_ioctl_register_dataset_read(ZFS_IOC_SEND_PROGRESS,
5428     zfs_ioc_send_progress);

5430     zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_DIFF,
5431     zfs_ioc_diff, zfs_secpolicy_diff);
5432     zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_STATS,
5433     zfs_ioc_obj_to_stats, zfs_secpolicy_diff);
5434     zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_PATH,
5435     zfs_ioc_obj_to_path, zfs_secpolicy_diff);
5436     zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_ONE,
5437     zfs_ioc_userspace_one, zfs_secpolicy_userspace_one);
5438     zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_MANY,
5439     zfs_ioc_userspace_many, zfs_secpolicy_userspace_many);
5440     zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_SEND,
5441     zfs_ioc_send, zfs_secpolicy_send);

5443     zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_PROP, zfs_ioc_set_prop,
5444     zfs_secpolicy_none);
5445     zfs_ioctl_register_dataset_modify(ZFS_IOC_DESTROY, zfs_ioc_destroy,
5446     zfs_secpolicy_destroy);
5447     zfs_ioctl_register_dataset_modify(ZFS_IOC_ROLLBACK, zfs_ioc_rollback,
5448     zfs_secpolicy_rollback);
5449     zfs_ioctl_register_dataset_modify(ZFS_IOC_RENAME, zfs_ioc_rename,
5450     zfs_secpolicy_rename);
5451     zfs_ioctl_register_dataset_modify(ZFS_IOC_RECV, zfs_ioc_recv,
5452     zfs_secpolicy_recv);
5453     zfs_ioctl_register_dataset_modify(ZFS_IOC_PROMOTE, zfs_ioc_promote,
5454     zfs_secpolicy_promote);
5455     zfs_ioctl_register_dataset_modify(ZFS_IOC_HOLD, zfs_ioc_hold,
5456     zfs_secpolicy_hold);
5457     zfs_ioctl_register_dataset_modify(ZFS_IOC_RELEASE, zfs_ioc_release,
5458     zfs_secpolicy_release);
5459     zfs_ioctl_register_dataset_modify(ZFS_IOC_INHERIT_PROP,
5460     zfs_ioc_inherit_prop, zfs_secpolicy_inherit_prop);
5461     zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_FSACL, zfs_ioc_set_fsacl,
5462     zfs_secpolicy_set_fsacl);

5464     zfs_ioctl_register_dataset_nolog(ZFS_IOC_SHARE, zfs_ioc_share,
5465     zfs_secpolicy_share, POOL_CHECK_NONE);
5466     zfs_ioctl_register_dataset_nolog(ZFS_IOC_SMB_ACL, zfs_ioc_smb_acl,
5467     zfs_secpolicy_smb_acl, POOL_CHECK_NONE);
5468     zfs_ioctl_register_dataset_nolog(ZFS_IOC_USERSPACE_UPGRADE,
5469     zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
5470     POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5471     zfs_ioctl_register_dataset_nolog(ZFS_IOC_TMP_SNAPSHOT,
5472     zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot,
5473     POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5474 }
4688 * pool create, destroy, and export don't log the history as part of
4689 * zfsdev_ioctl, but rather zfs_ioc_pool_create, and zfs_ioc_pool_export
4690 * do the logging of those commands.
4691 */
4692 static zfs_ioc_vec_t zfs_ioc_vec[] = {
4693     { zfs_ioc_pool_create, zfs_secpolicy_config, POOL_NAME, B_FALSE,
4694     POOL_CHECK_NONE },
4695     { zfs_ioc_pool_destroy, zfs_secpolicy_config, POOL_NAME, B_FALSE,
4696     POOL_CHECK_NONE },
4697     { zfs_ioc_pool_import, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4698     POOL_CHECK_NONE },
4699     { zfs_ioc_pool_export, zfs_secpolicy_config, POOL_NAME, B_FALSE,
4700     POOL_CHECK_NONE },
4701     { zfs_ioc_pool_configs, zfs_secpolicy_none, NO_NAME, B_FALSE,
4702     POOL_CHECK_NONE },

```

```

4703 { zfs_ioc_pool_stats, zfs_secpolicy_read, POOL_NAME, B_FALSE,
4704   POOL_CHECK_NONE },
4705 { zfs_ioc_pool_tryimport, zfs_secpolicy_config, NO_NAME, B_FALSE,
4706   POOL_CHECK_NONE },
4707 { zfs_ioc_pool_scan, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4708   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4709 { zfs_ioc_pool_freeze, zfs_secpolicy_config, NO_NAME, B_FALSE,
4710   POOL_CHECK_READONLY },
4711 { zfs_ioc_pool_upgrade, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4712   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4713 { zfs_ioc_pool_get_history, zfs_secpolicy_config, POOL_NAME, B_FALSE,
4714   POOL_CHECK_NONE },
4715 { zfs_ioc_vdev_add, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4716   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4717 { zfs_ioc_vdev_remove, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4718   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4719 { zfs_ioc_vdev_set_state, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4720   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4721 { zfs_ioc_vdev_attach, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4722   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4723 { zfs_ioc_vdev_detach, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4724   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4725 { zfs_ioc_vdev_setpath, zfs_secpolicy_config, POOL_NAME, B_FALSE,
4726   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4727 { zfs_ioc_vdev_setfru, zfs_secpolicy_config, POOL_NAME, B_FALSE,
4728   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4729 { zfs_ioc_objset_stats, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4730   POOL_CHECK_SUSPENDED },
4731 { zfs_ioc_objset_zplprops, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4732   POOL_CHECK_NONE },
4733 { zfs_ioc_dataset_list_next, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4734   POOL_CHECK_SUSPENDED },
4735 { zfs_ioc_snapshot_list_next, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4736   POOL_CHECK_SUSPENDED },
4737 { zfs_ioc_set_prop, zfs_secpolicy_none, DATASET_NAME, B_TRUE,
4738   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4739 { zfs_ioc_create, zfs_secpolicy_create, DATASET_NAME, B_TRUE,
4740   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4741 { zfs_ioc_destroy, zfs_secpolicy_destroy, DATASET_NAME, B_TRUE,
4742   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4743 { zfs_ioc_rollback, zfs_secpolicy_rollback, DATASET_NAME, B_TRUE,
4744   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4745 { zfs_ioc_rename, zfs_secpolicy_rename, DATASET_NAME, B_TRUE,
4746   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4747 { zfs_ioc_recv, zfs_secpolicy_receive, DATASET_NAME, B_TRUE,
4748   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4749 { zfs_ioc_send, zfs_secpolicy_send, DATASET_NAME, B_FALSE,
4750   POOL_CHECK_NONE },
4751 { zfs_ioc_inject_fault, zfs_secpolicy_inject, NO_NAME, B_FALSE,
4752   POOL_CHECK_NONE },
4753 { zfs_ioc_clear_fault, zfs_secpolicy_inject, NO_NAME, B_FALSE,
4754   POOL_CHECK_NONE },
4755 { zfs_ioc_inject_list_next, zfs_secpolicy_inject, NO_NAME, B_FALSE,
4756   POOL_CHECK_NONE },
4757 { zfs_ioc_error_log, zfs_secpolicy_inject, POOL_NAME, B_FALSE,
4758   POOL_CHECK_NONE },
4759 { zfs_ioc_clear, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4760   POOL_CHECK_NONE },
4761 { zfs_ioc_promote, zfs_secpolicy_promote, DATASET_NAME, B_TRUE,
4762   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4763 { zfs_ioc_snapshot, zfs_secpolicy_snapshot, DATASET_NAME, B_TRUE,
4764   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4765 { zfs_ioc_dsobj_to_dsname, zfs_secpolicy_diff, POOL_NAME, B_FALSE,
4766   POOL_CHECK_NONE },
4767 { zfs_ioc_obj_to_path, zfs_secpolicy_diff, DATASET_NAME, B_FALSE,
4768   POOL_CHECK_SUSPENDED },

```

```

4769 { zfs_ioc_pool_set_props, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4770   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4771 { zfs_ioc_pool_get_props, zfs_secpolicy_read, POOL_NAME, B_FALSE,
4772   POOL_CHECK_NONE },
4773 { zfs_ioc_set_fsacl, zfs_secpolicy_fsacl, DATASET_NAME, B_TRUE,
4774   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4775 { zfs_ioc_get_fsacl, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4776   POOL_CHECK_NONE },
4777 { zfs_ioc_share, zfs_secpolicy_share, DATASET_NAME, B_FALSE,
4778   POOL_CHECK_NONE },
4779 { zfs_ioc_inherit_prop, zfs_secpolicy_inherit, DATASET_NAME, B_TRUE,
4780   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4781 { zfs_ioc_smb_acl, zfs_secpolicy_smb_acl, DATASET_NAME, B_FALSE,
4782   POOL_CHECK_NONE },
4783 { zfs_ioc_userspace_one, zfs_secpolicy_userspace_one, DATASET_NAME,
4784   B_FALSE, POOL_CHECK_NONE },
4785 { zfs_ioc_userspace_many, zfs_secpolicy_userspace_many, DATASET_NAME,
4786   B_FALSE, POOL_CHECK_NONE },
4787 { zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
4788   DATASET_NAME, B_FALSE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4789 { zfs_ioc_hold, zfs_secpolicy_hold, DATASET_NAME, B_TRUE,
4790   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4791 { zfs_ioc_release, zfs_secpolicy_release, DATASET_NAME, B_TRUE,
4792   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4793 { zfs_ioc_get_holds, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4794   POOL_CHECK_SUSPENDED },
4795 { zfs_ioc_objset_recvd_props, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4796   POOL_CHECK_NONE },
4797 { zfs_ioc_vdev_split, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4798   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4799 { zfs_ioc_next_obj, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4800   POOL_CHECK_NONE },
4801 { zfs_ioc_diff, zfs_secpolicy_diff, DATASET_NAME, B_FALSE,
4802   POOL_CHECK_NONE },
4803 { zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot, DATASET_NAME,
4804   B_FALSE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4805 { zfs_ioc_obj_to_stats, zfs_secpolicy_diff, DATASET_NAME, B_FALSE,
4806   POOL_CHECK_SUSPENDED },
4807 { zfs_ioc_space_written, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4808   POOL_CHECK_SUSPENDED },
4809 { zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4810   POOL_CHECK_SUSPENDED },
4811 { zfs_ioc_destroy_snaps_nv1, zfs_secpolicy_destroy_recursive,
4812   DATASET_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4813 { zfs_ioc_pool_reguid, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4814   POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY },
4815 { zfs_ioc_pool_reopen, zfs_secpolicy_config, POOL_NAME, B_TRUE,
4816   POOL_CHECK_SUSPENDED },
4817 { zfs_ioc_send_progress, zfs_secpolicy_read, DATASET_NAME, B_FALSE,
4818   POOL_CHECK_NONE },
4819 };

5476 int
5477 pool_status_check(const char *name, zfs_ioc_namecheck_t type,
5478   zfs_ioc_poolcheck_t check)
5479 {
5480     spa_t *spa;
5481     int error;

5483     ASSERT(type == POOL_NAME || type == DATASET_NAME);

5485     if (check & POOL_CHECK_NONE)
5486         return (0);

5488     error = spa_open(name, &spa, FTAG);
5489     if (error == 0) {

```



```

5490         if ((check & POOL_CHECK_SUSPENDED) && spa_suspended(spa))
5491             error = EAGAIN;
5492         else if ((check & POOL_CHECK_READONLY) && !spa_writeable(spa))
5493             error = EROFS;
5494         spa_close(spa, FTAG);
5495     }
5496     return (error);
5497 }
    _____
    unchanged_portion_omitted_

5607 static int
5608 zfsdev_ioctl(dev_t dev, int cmd, intp_t arg, int flag, cred_t *cr, int *rvalp)
5609 {
5610     zfs_cmd_t *zc;
5611     uint_t vecnum;
5612     int error, rc, len;
5613     uint_t vec;
5614     int error, rc;
5615     minor_t minor = getminor(dev);
5616     const zfs_ioc_vec_t *vec;
5617     char *saved_poolname = NULL;
5618     nvlist_t *innvl = NULL;
5619 #endif /* ! codereview */

5619     if (minor != 0 &&
5620         zfsdev_get_soft_state(minor, ZSST_CTLDEV) == NULL)
5621         return (zvol_ioctl(dev, cmd, arg, flag, cr, rvalp));

5622     vecnum = cmd - ZFS_IOC_FIRST;
5623     vec = cmd - ZFS_IOC;
5624     ASSERT3U(getmajor(dev), ==, ddi_driver_major(zfs_dip));

5625     if (vecnum >= sizeof (zfs_ioc_vec) / sizeof (zfs_ioc_vec[0]))
5626     if (vec >= sizeof (zfs_ioc_vec) / sizeof (zfs_ioc_vec[0]))
5627         return (EINVAL);
5628     vec = &zfs_ioc_vec[vecnum];
5629 #endif /* ! codereview */

5631     zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);

5633     error = ddi_copyin((void *)arg, zc, sizeof (zfs_cmd_t), flag);
5634     if (error != 0) {
5635         if (error != 0)
5636             error = EFAULT;
5637         goto out;
5638 #endif /* ! codereview */

5640     zc->zc_iflags = flag & FKIOCTL;
5641     if (zc->zc_nvlist_src_size != 0) {
5642         error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
5643             zc->zc_iflags, &innvl);
5644         if (error != 0)
5645             goto out;
5646     }
5647     if ((error == 0) && !(flag & FKIOCTL))
5648         error = zfs_ioc_vec[vec].zvec_secpolicy(zc, cr);

5648     /*
5649      * Ensure that all pool/dataset names are valid before we pass down to
5650      * the lower layers.
5651      */
5652     if (error == 0) {
5653         zc->zc_name[sizeof (zc->zc_name) - 1] = '\0';
5654         switch (vec->zvec_namecheck) {
5655             case POOL_NAME:
5656                 if (pool_namecheck(zc->zc_name, NULL, NULL) != 0)
5657                     error = EINVAL;
5658                 else
5659                     error = pool_status_check(zc->zc_name,
5660                         vec->zvec_namecheck, vec->zvec_pool_check);
5661                 break;
5662             case DATASET_NAME:
5663                 if (dataset_namecheck(zc->zc_name, NULL, NULL) != 0)
5664                     error = EINVAL;
5665                 else
5666                     error = pool_status_check(zc->zc_name,
5667                         vec->zvec_namecheck, vec->zvec_pool_check);
5668                 break;
5669             case NO_NAME:
5670                 break;
5671         }

5672         if (error == 0 && !(flag & FKIOCTL))
5673             error = vec->zvec_secpolicy(zc, innvl, cr);

5674         if (error != 0)
5675             goto out;

5676         /* legacy ioctls can modify zc_name */
5677         len = strcspn(zc->zc_name, "/@") + 1;
5678         saved_poolname = kmem_alloc(len, KM_SLEEP);
5679         (void) strncpy(saved_poolname, zc->zc_name, len);

5680         if (vec->zvec_func != NULL) {
5681             nvlist_t *outnvl;
5682             int puterror = 0;
5683             spa_t *spa;
5684             nvlist_t *lognv = NULL;

5685             ASSERT(vec->zvec_legacy_func == NULL);

5686             /*
5687              * Add the innvl to the lognv before calling the func,
5688              * in case the func changes the innvl.
5689              */
5690             if (vec->zvec_allow_log) {
5691                 lognv = fnvlist_alloc();
5692                 fnvlist_add_string(lognv, ZPOOL_HIST_IOCTL,
5693                     vec->zvec_name);
5694                 if (!nvlist_empty(innvl)) {
5695                     fnvlist_add_nvlist(lognv, ZPOOL_HIST_INPUT_NVLIST,
5696                         innvl);
5697                 }
5698             }

5699             outnvl = fnvlist_alloc();
5700             error = vec->zvec_func(zc->zc_name, innvl, outnvl);

5701             if (error == 0 && vec->zvec_allow_log &&
5702                 spa_open(zc->zc_name, &spa, FTAG) == 0) {

```

```

4976         switch (zfs_ioc_vec[vec].zvec_namecheck) {
4977             case POOL_NAME:
4978                 if (pool_namecheck(zc->zc_name, NULL, NULL) != 0)
4979                     error = EINVAL;
4980                 else
4981                     error = pool_status_check(zc->zc_name,
4982                         vec->zvec_namecheck, vec->zvec_pool_check);
4983                 break;
4984             case DATASET_NAME:
4985                 if (dataset_namecheck(zc->zc_name, NULL, NULL) != 0)
4986                     error = EINVAL;
4987                 else
4988                     error = pool_status_check(zc->zc_name,
4989                         vec->zvec_namecheck, vec->zvec_pool_check);
4990                 break;
4991             case NO_NAME:
4992                 break;
4993         }

4994         if (error == 0 && !(flag & FKIOCTL))
4995             error = vec->zvec_secpolicy(zc, innvl, cr);

4996         if (error != 0)
4997             goto out;

4998         /* legacy ioctls can modify zc_name */
4999         len = strcspn(zc->zc_name, "/@") + 1;
5000         saved_poolname = kmem_alloc(len, KM_SLEEP);
5001         (void) strncpy(saved_poolname, zc->zc_name, len);

5002         if (vec->zvec_func != NULL) {
5003             nvlist_t *outnvl;
5004             int puterror = 0;
5005             spa_t *spa;
5006             nvlist_t *lognv = NULL;

5007             ASSERT(vec->zvec_legacy_func == NULL);

5008             /*
5009              * Add the innvl to the lognv before calling the func,
5010              * in case the func changes the innvl.
5011              */
5012             if (vec->zvec_allow_log) {
5013                 lognv = fnvlist_alloc();
5014                 fnvlist_add_string(lognv, ZPOOL_HIST_IOCTL,
5015                     vec->zvec_name);
5016                 if (!nvlist_empty(innvl)) {
5017                     fnvlist_add_nvlist(lognv, ZPOOL_HIST_INPUT_NVLIST,
5018                         innvl);
5019                 }
5020             }

5021             outnvl = fnvlist_alloc();
5022             error = vec->zvec_func(zc->zc_name, innvl, outnvl);

5023             if (error == 0 && vec->zvec_allow_log &&
5024                 spa_open(zc->zc_name, &spa, FTAG) == 0) {

```

```

5715         if (!nvlst_empty(outnvl)) {
5716             fnvlst_add_nvlst(lognv, ZPOOL_HIST_OUTPUT_NVL,
5717                 outnvl);
5718         }
5719         (void) spa_history_log_nvl(spa, lognv);
5720         spa_close(spa, FTAG);
5721     }
5722     fnvlst_free(lognv);

5724     if (!nvlst_empty(outnvl) || zc->zc_nvlist_dst_size != 0) {
5725         int smusherror = 0;
5726         if (vec->zvec_smush_outnvlst) {
5727             smusherror = nvlst_smush(outnvl,
5728                 zc->zc_nvlist_dst_size);
5729         }
5730         if (smusherror == 0)
5731             puterror = put_nvlist(zc, outnvl);
5732     #endif /* ! codereview */
5733     }

5735     if (puterror != 0)
5736         error = puterror;

5738     nvlst_free(outnvl);
5739 } else {
5740     error = vec->zvec_legacy_func(zc);
5741 }
4994 if (error == 0)
4995     error = zfs_ioc_vec[vec].zvec_func(zc);

5743 out:
5744     nvlst_free(innvl);
5745 #endif /* ! codereview */
5746 rc = ddi_copyout(zc, (void *)arg, sizeof (zfs_cmd_t), flag);
5747 if (error == 0 && rc != 0)
4997 if (error == 0) {
4998     if (rc != 0)
5748         error = EFAULT;
5749 if (error == 0 && vec->zvec_allow_log) {
5750     char *s = tsd_get(zfs_allow_log_key);
5751     if (s != NULL)
5752         strfree(s);
5753     (void) tsd_set(zfs_allow_log_key, saved_poolname);
5754 } else {
5755     if (saved_poolname != NULL)
5756         strfree(saved_poolname);
5000     if (zfs_ioc_vec[vec].zvec_his_log)
5001         zfs_log_history(zc);
5757 }

5759     kmem_free(zc, sizeof (zfs_cmd_t));
5760     return (error);
5761 }
    unchanged_portion_omitted

5872 static void
5873 zfs_allow_log_destroy(void *arg)
5874 {
5875     char *poolname = arg;
5876     strfree(poolname);
5877 }

5118 uint_t zfs_fsycer_key;
5119 extern uint_t rrw_tsd_key;

5879 int

```

```

5880 _init(void)
5881 {
5882     int error;

5884     spa_init(FREAD | FWRITE);
5885     zfs_init();
5886     zvol_init();
5887     zfs_ioctl_init();
5888 #endif /* ! codereview */

5890     if ((error = mod_install(&modlinkage)) != 0) {
5891         zvol_fini();
5892         zfs_fini();
5893         spa_fini();
5894         return (error);
5895     }

5897     tsd_create(&zfs_fsycer_key, NULL);
5898     tsd_create(&rrw_tsd_key, rrw_tsd_destroy);
5899     tsd_create(&zfs_allow_log_key, zfs_allow_log_destroy);
5129     tsd_create(&rrw_tsd_key, NULL);

5901     error = ldi_ident_from_mod(&modlinkage, &zfs_li);
5902     ASSERT(error == 0);
5903     mutex_init(&zfs_share_lock, NULL, MUTEX_DEFAULT, NULL);

5905     return (0);
5906 }
    unchanged_portion_omitted

```

```

*****
58649 Thu Jun 28 15:57:21 2012
new/usr/src/uts/common/fs/zfs/zfs_vfsops.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 #endif /* ! codereview */
25 */
26
27 /* Portions Copyright 2010 Robert Milkowski */
28
29 #include <sys/types.h>
30 #include <sys/param.h>
31 #include <sys/system.h>
32 #include <sys/sysmacros.h>
33 #include <sys/kmem.h>
34 #include <sys/pathname.h>
35 #include <sys/vnode.h>
36 #include <sys/vfs.h>
37 #include <sys/vfs_opreg.h>
38 #include <sys/mntent.h>
39 #include <sys/mount.h>
40 #include <sys/cmn_err.h>
41 #include "fs/fs_subr.h"
42 #include <sys/zfs_znode.h>
43 #include <sys/zfs_dir.h>
44 #include <sys/zil.h>
45 #include <sys/fs/zfs.h>
46 #include <sys/dmu.h>
47 #include <sys/dsl_prop.h>
48 #include <sys/dsl_dataset.h>
49 #include <sys/dsl_deleg.h>
50 #include <sys/spa.h>
51 #include <sys/zap.h>
52 #include <sys/sa.h>
53 #include <sys/varargs.h>
54 #include <sys/policy.h>

```

```

55 #include <sys/atomic.h>
56 #include <sys/mkdev.h>
57 #include <sys/modctl.h>
58 #include <sys/refstr.h>
59 #include <sys/zfs_ioctl.h>
60 #include <sys/zfs_ctldir.h>
61 #include <sys/zfs_fuid.h>
62 #include <sys/bootconf.h>
63 #include <sys/sunddi.h>
64 #include <sys/dnld.h>
65 #include <sys/dmu_objset.h>
66 #include <sys/spa_boot.h>
67 #include <sys/sa.h>
68 #include "zfs_comutil.h"
69
70 int zfsfstype;
71 vfsops_t *zfs_vfsops = NULL;
72 static major_t zfs_major;
73 static minor_t zfs_minor;
74 static kmutex_t zfs_dev_mtx;
75
76 extern int sys_shutdown;
77
78 static int zfs_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr);
79 static int zfs_umount(vfs_t *vfsp, int fflag, cred_t *cr);
80 static int zfs_mountroot(vfs_t *vfsp, enum whymountroot);
81 static int zfs_root(vfs_t *vfsp, vnode_t **vpp);
82 static int zfs_statvfs(vfs_t *vfsp, struct statvfs64 *statp);
83 static int zfs_vget(vfs_t *vfsp, vnode_t **vpp, fid_t *fidp);
84 static void zfs_freevfs(vfs_t *vfsp);
85
86 static const fs_operation_def_t zfs_vfsops_template[] = {
87     VFSNAME_MOUNT,           { .zfs_mount = zfs_mount },
88     VFSNAME_MOUNTROOT,     { .zfs_mountroot = zfs_mountroot },
89     VFSNAME_UNMOUNT,       { .zfs_unmount = zfs_unmount },
90     VFSNAME_ROOT,          { .zfs_root = zfs_root },
91     VFSNAME_STATVFS,       { .zfs_statvfs = zfs_statvfs },
92     VFSNAME_SYNC,          { .zfs_sync = zfs_sync },
93     VFSNAME_VGET,          { .zfs_vget = zfs_vget },
94     VFSNAME_FREEVFS,       { .zfs_freevfs = zfs_freevfs },
95     NULL,                   NULL
96 };
97
98 static const fs_operation_def_t zfs_vfsops_eio_template[] = {
99     VFSNAME_FREEVFS,       { .zfs_freevfs = zfs_freevfs },
100    NULL,                   NULL
101 };
102
103 /*
104  * We need to keep a count of active fs's.
105  * This is necessary to prevent our module
106  * from being unloaded after a umount -f
107  */
108 static uint32_t zfs_active_fs_count = 0;
109
110 static char *noatime_cancel[] = { MNTOPT_ETIME, NULL };
111 static char *atime_cancel[] = { MNTOPT_NOATIME, NULL };
112 static char *noxattr_cancel[] = { MNTOPT_XATTR, NULL };
113 static char *xattr_cancel[] = { MNTOPT_NOXATTR, NULL };
114
115 /*
116  * MO_DEFAULT is not used since the default value is determined
117  * by the equivalent property.
118  */
119 static mntopt_t mntopts[] = {
120     { MNTOPT_NOXATTR, noxattr_cancel, NULL, 0, NULL },

```

```

121     { MNTOPT_XATTR, xattr_cancel, NULL, 0, NULL },
122     { MNTOPT_NOATIME, noatime_cancel, NULL, 0, NULL },
123     { MNTOPT_ATIME, atime_cancel, NULL, 0, NULL }
124 };

126 static mntopts_t zfs_mntopts = {
127     sizeof (mntopts) / sizeof (mntopt_t),
128     mntopts
129 };

131 /*ARGSUSED*/
132 int
133 zfs_sync(vfs_t *vfsp, short flag, cred_t *cr)
134 {
135     /*
136      * Data integrity is job one. We don't want a compromised kernel
137      * writing to the storage pool, so we never sync during panic.
138      */
139     if (panicstr)
140         return (0);

142     /*
143      * SYNC_ATTR is used by fsflush() to force old filesystems like UFS
144      * to sync metadata, which they would otherwise cache indefinitely.
145      * Semantically, the only requirement is that the sync be initiated.
146      * The DMU syncs out txgs frequently, so there's nothing to do.
147      */
148     if (flag & SYNC_ATTR)
149         return (0);

151     if (vfsp != NULL) {
152         /*
153          * Sync a specific filesystem.
154          */
155         zfsvfs_t *zfsvfs = vfsp->vfs_data;
156         dsl_pool_t *dp;

158         ZFS_ENTER(zfsvfs);
159         dp = dmub_objset_pool(zfsvfs->z_os);

161         /*
162          * If the system is shutting down, then skip any
163          * filesystems which may exist on a suspended pool.
164          */
165         if (sys_shutdown && spa_suspended(dp->dp_spa)) {
166             ZFS_EXIT(zfsvfs);
167             return (0);
168         }

170         if (zfsvfs->z_log != NULL)
171             zil_commit(zfsvfs->z_log, 0);

173         ZFS_EXIT(zfsvfs);
174     } else {
175         /*
176          * Sync all ZFS filesystems. This is what happens when you
177          * run sync(1M). Unlike other filesystems, ZFS honors the
178          * request by waiting for all pools to commit all dirty data.
179          */
180         spa_sync_allpools();
181     }

183     return (0);
184 }

186 static int

```

```

187 zfs_create_unique_device(dev_t *dev)
188 {
189     major_t new_major;

191     do {
192         ASSERT3U(zfs_minor, <=, MAXMIN32);
193         minor_t start = zfs_minor;
194         do {
195             mutex_enter(&zfs_dev_mtx);
196             if (zfs_minor >= MAXMIN32) {
197                 /*
198                  * If we're still using the real major
199                  * keep out of /dev/zfs and /dev/zvol minor
200                  * number space. If we're using a getudev()'ed
201                  * major number, we can use all of its minors.
202                  */
203                 if (zfs_major == ddi_name_to_major(ZFS_DRIVER))
204                     zfs_minor = ZFS_MIN_MINOR;
205                 else
206                     zfs_minor = 0;
207             } else {
208                 zfs_minor++;
209             }
210             *dev = makedevice(zfs_major, zfs_minor);
211             mutex_exit(&zfs_dev_mtx);
212         } while (vfs_devismounted(*dev) && zfs_minor != start);
213         if (zfs_minor == start) {
214             /*
215              * We are using all ~262,000 minor numbers for the
216              * current major number. Create a new major number.
217              */
218             if ((new_major = getudev()) == (major_t)-1) {
219                 cmn_err(CE_WARN,
220                     "zfs_mount: Can't get unique major "
221                     "device number.");
222                 return (-1);
223             }
224             mutex_enter(&zfs_dev_mtx);
225             zfs_major = new_major;
226             zfs_minor = 0;

228             mutex_exit(&zfs_dev_mtx);
229         } else {
230             break;
231         }
232         /* CONSTANTCONDITION */
233     } while (1);

235     return (0);
236 }

238 static void
239 atime_changed_cb(void *arg, uint64_t newval)
240 {
241     zfsvfs_t *zfsvfs = arg;

243     if (newval == TRUE) {
244         zfsvfs->z_atime = TRUE;
245         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NOATIME);
246         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_ATIME, NULL, 0);
247     } else {
248         zfsvfs->z_atime = FALSE;
249         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_ATIME);
250         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NOATIME, NULL, 0);
251     }
252 }

```

```

254 static void
255 xattr_changed_cb(void *arg, uint64_t newval)
256 {
257     zfsvfs_t *zfsvfs = arg;
258
259     if (newval == TRUE) {
260         /* XXX locking on vfs_flag? */
261         zfsvfs->z_vfs->vfs_flag |= VFS_XATTR;
262         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NOXATTR);
263         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_XATTR, NULL, 0);
264     } else {
265         /* XXX locking on vfs_flag? */
266         zfsvfs->z_vfs->vfs_flag &= ~VFS_XATTR;
267         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_XATTR);
268         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NOXATTR, NULL, 0);
269     }
270 }
271
272 static void
273 blkksz_changed_cb(void *arg, uint64_t newval)
274 {
275     zfsvfs_t *zfsvfs = arg;
276
277     if (newval < SPA_MINBLOCKSIZE ||
278         newval > SPA_MAXBLOCKSIZE || !ISP2(newval))
279         newval = SPA_MAXBLOCKSIZE;
280
281     zfsvfs->z_max_blkksz = newval;
282     zfsvfs->z_vfs->vfs_bsize = newval;
283 }
284
285 static void
286 readonly_changed_cb(void *arg, uint64_t newval)
287 {
288     zfsvfs_t *zfsvfs = arg;
289
290     if (newval) {
291         /* XXX locking on vfs_flag? */
292         zfsvfs->z_vfs->vfs_flag |= VFS_RDONLY;
293         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_RW);
294         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_RO, NULL, 0);
295     } else {
296         /* XXX locking on vfs_flag? */
297         zfsvfs->z_vfs->vfs_flag &= ~VFS_RDONLY;
298         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_RO);
299         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_RW, NULL, 0);
300     }
301 }
302
303 static void
304 devices_changed_cb(void *arg, uint64_t newval)
305 {
306     zfsvfs_t *zfsvfs = arg;
307
308     if (newval == FALSE) {
309         zfsvfs->z_vfs->vfs_flag |= VFS_NODEVICES;
310         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_DEVICES);
311         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NODEVICES, NULL, 0);
312     } else {
313         zfsvfs->z_vfs->vfs_flag &= ~VFS_NODEVICES;
314         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NODEVICES);
315         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_DEVICES, NULL, 0);
316     }
317 }

```

```

319 static void
320 setuid_changed_cb(void *arg, uint64_t newval)
321 {
322     zfsvfs_t *zfsvfs = arg;
323
324     if (newval == FALSE) {
325         zfsvfs->z_vfs->vfs_flag |= VFS_NOSETUID;
326         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_SETUID);
327         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NOSETUID, NULL, 0);
328     } else {
329         zfsvfs->z_vfs->vfs_flag &= ~VFS_NOSETUID;
330         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NOSETUID);
331         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_SETUID, NULL, 0);
332     }
333 }
334
335 static void
336 exec_changed_cb(void *arg, uint64_t newval)
337 {
338     zfsvfs_t *zfsvfs = arg;
339
340     if (newval == FALSE) {
341         zfsvfs->z_vfs->vfs_flag |= VFS_NOEXEC;
342         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_EXEC);
343         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NOEXEC, NULL, 0);
344     } else {
345         zfsvfs->z_vfs->vfs_flag &= ~VFS_NOEXEC;
346         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NOEXEC);
347         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_EXEC, NULL, 0);
348     }
349 }
350
351 /*
352  * The nbmand mount option can be changed at mount time.
353  * We can't allow it to be toggled on live file systems or incorrect
354  * behavior may be seen from cifs clients
355  *
356  * This property isn't registered via dsl_prop_register(), but this callback
357  * will be called when a file system is first mounted
358  */
359 static void
360 nbmand_changed_cb(void *arg, uint64_t newval)
361 {
362     zfsvfs_t *zfsvfs = arg;
363     if (newval == FALSE) {
364         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NBMAND);
365         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NONBMAND, NULL, 0);
366     } else {
367         vfs_clearmntopt(zfsvfs->z_vfs, MNTOPT_NONBMAND);
368         vfs_setmntopt(zfsvfs->z_vfs, MNTOPT_NBMAND, NULL, 0);
369     }
370 }
371
372 static void
373 snapdir_changed_cb(void *arg, uint64_t newval)
374 {
375     zfsvfs_t *zfsvfs = arg;
376
377     zfsvfs->z_show_ctldir = newval;
378 }
379
380 static void
381 vscan_changed_cb(void *arg, uint64_t newval)
382 {
383     zfsvfs_t *zfsvfs = arg;

```

```

385     zfsvfs->z_vscan = newval;
386 }

388 static void
389 acl_mode_changed_cb(void *arg, uint64_t newval)
390 {
391     zfsvfs_t *zfsvfs = arg;

393     zfsvfs->z_acl_mode = newval;
394 }

396 static void
397 acl_inherit_changed_cb(void *arg, uint64_t newval)
398 {
399     zfsvfs_t *zfsvfs = arg;

401     zfsvfs->z_acl_inherit = newval;
402 }

404 static int
405 zfs_register_callbacks(vfs_t *vfsp)
406 {
407     struct dsl_dataset *ds = NULL;
408     objset_t *os = NULL;
409     zfsvfs_t *zfsvfs = NULL;
410     uint64_t nbmand;
411     int readonly, do_readonly = B_FALSE;
412     int setuid, do_setuid = B_FALSE;
413     int exec, do_exec = B_FALSE;
414     int devices, do_devices = B_FALSE;
415     int xattr, do_xattr = B_FALSE;
416     int atime, do_atime = B_FALSE;
417     int error = 0;

419     ASSERT(vfsp);
420     zfsvfs = vfs->vfs_data;
421     ASSERT(zfsvfs);
422     os = zfsvfs->z_os;

424     /*
425      * The act of registering our callbacks will destroy any mount
426      * options we may have.  In order to enable temporary overrides
427      * of mount options, we stash away the current values and
428      * restore them after we register the callbacks.
429      */
430     if (vfs_optionisset(vfsp, MNTOPT_RO, NULL) ||
431         !spa_writeable(dmu_objset_spa(os))) {
432         readonly = B_TRUE;
433         do_readonly = B_TRUE;
434     } else if (vfs_optionisset(vfsp, MNTOPT_RW, NULL)) {
435         readonly = B_FALSE;
436         do_readonly = B_TRUE;
437     }
438     if (vfs_optionisset(vfsp, MNTOPT_NOSUID, NULL)) {
439         devices = B_FALSE;
440         setuid = B_FALSE;
441         do_devices = B_TRUE;
442         do_setuid = B_TRUE;
443     } else {
444         if (vfs_optionisset(vfsp, MNTOPT_NODEVICES, NULL)) {
445             devices = B_FALSE;
446             do_devices = B_TRUE;
447         } else if (vfs_optionisset(vfsp, MNTOPT_DEVICES, NULL)) {
448             devices = B_TRUE;
449             do_devices = B_TRUE;
450         }

```

```

452         if (vfs_optionisset(vfsp, MNTOPT_NOSETUID, NULL)) {
453             setuid = B_FALSE;
454             do_setuid = B_TRUE;
455         } else if (vfs_optionisset(vfsp, MNTOPT_SETUID, NULL)) {
456             setuid = B_TRUE;
457             do_setuid = B_TRUE;
458         }
459     }
460     if (vfs_optionisset(vfsp, MNTOPT_NOEXEC, NULL)) {
461         exec = B_FALSE;
462         do_exec = B_TRUE;
463     } else if (vfs_optionisset(vfsp, MNTOPT_EXEC, NULL)) {
464         exec = B_TRUE;
465         do_exec = B_TRUE;
466     }
467     if (vfs_optionisset(vfsp, MNTOPT_NOXATTR, NULL)) {
468         xattr = B_FALSE;
469         do_xattr = B_TRUE;
470     } else if (vfs_optionisset(vfsp, MNTOPT_XATTR, NULL)) {
471         xattr = B_TRUE;
472         do_xattr = B_TRUE;
473     }
474     if (vfs_optionisset(vfsp, MNTOPT_NOATIME, NULL)) {
475         atime = B_FALSE;
476         do_atime = B_TRUE;
477     } else if (vfs_optionisset(vfsp, MNTOPT_ATIME, NULL)) {
478         atime = B_TRUE;
479         do_atime = B_TRUE;
480     }

482     /*
483      * nbmand is a special property.  It can only be changed at
484      * mount time.
485      *
486      * This is weird, but it is documented to only be changeable
487      * at mount time.
488      */
489     if (vfs_optionisset(vfsp, MNTOPT_NONBMAND, NULL)) {
490         nbmand = B_FALSE;
491     } else if (vfs_optionisset(vfsp, MNTOPT_NBMAND, NULL)) {
492         nbmand = B_TRUE;
493     } else {
494         char osname[MAXNAMELEN];

496         dmu_objset_name(os, osname);
497         if (error = dsl_prop_get_integer(osname, "nbmand", &nbmand,
498             NULL)) {
499             return (error);
500         }
501     }

503     /*
504      * Register property callbacks.
505      *
506      * It would probably be fine to just check for i/o error from
507      * the first prop_register(), but I guess I like to go
508      * overboard...
509      */
510     ds = dmu_objset_ds(os);
511     error = dsl_prop_register(ds, "atime", atime_changed_cb, zfsvfs);
512     error = error ? error : dsl_prop_register(ds,
513         "xattr", xattr_changed_cb, zfsvfs);
514     error = error ? error : dsl_prop_register(ds,
515         "recordsize", blksz_changed_cb, zfsvfs);
516     error = error ? error : dsl_prop_register(ds,

```

```

517     "readonly", readonly_changed_cb, zfsvfs);
518     error = error ? error : dsl_prop_register(ds,
519     "devices", devices_changed_cb, zfsvfs);
520     error = error ? error : dsl_prop_register(ds,
521     "setuid", setuid_changed_cb, zfsvfs);
522     error = error ? error : dsl_prop_register(ds,
523     "exec", exec_changed_cb, zfsvfs);
524     error = error ? error : dsl_prop_register(ds,
525     "snapdir", snapdir_changed_cb, zfsvfs);
526     error = error ? error : dsl_prop_register(ds,
527     "aclmode", acl_mode_changed_cb, zfsvfs);
528     error = error ? error : dsl_prop_register(ds,
529     "aclinherit", acl_inherit_changed_cb, zfsvfs);
530     error = error ? error : dsl_prop_register(ds,
531     "vscan", vscan_changed_cb, zfsvfs);
532     if (error)
533         goto unregister;
534
535     /*
536     * Invoke our callbacks to restore temporary mount options.
537     */
538     if (do_readonly)
539         readonly_changed_cb(zfsvfs, readonly);
540     if (do_setuid)
541         setuid_changed_cb(zfsvfs, setuid);
542     if (do_exec)
543         exec_changed_cb(zfsvfs, exec);
544     if (do_devices)
545         devices_changed_cb(zfsvfs, devices);
546     if (do_xattr)
547         xattr_changed_cb(zfsvfs, xattr);
548     if (do_atime)
549         atime_changed_cb(zfsvfs, atime);
550
551     nbmand_changed_cb(zfsvfs, nbmand);
552
553     return (0);
554
555 unregister:
556     /*
557     * We may attempt to unregister some callbacks that are not
558     * registered, but this is OK; it will simply return ENOMSG,
559     * which we will ignore.
560     */
561     (void) dsl_prop_unregister(ds, "atime", atime_changed_cb, zfsvfs);
562     (void) dsl_prop_unregister(ds, "xattr", xattr_changed_cb, zfsvfs);
563     (void) dsl_prop_unregister(ds, "recordsize", blksize_changed_cb, zfsvfs);
564     (void) dsl_prop_unregister(ds, "readonly", readonly_changed_cb, zfsvfs);
565     (void) dsl_prop_unregister(ds, "devices", devices_changed_cb, zfsvfs);
566     (void) dsl_prop_unregister(ds, "setuid", setuid_changed_cb, zfsvfs);
567     (void) dsl_prop_unregister(ds, "exec", exec_changed_cb, zfsvfs);
568     (void) dsl_prop_unregister(ds, "snapdir", snapdir_changed_cb, zfsvfs);
569     (void) dsl_prop_unregister(ds, "aclmode", acl_mode_changed_cb, zfsvfs);
570     (void) dsl_prop_unregister(ds, "aclinherit", acl_inherit_changed_cb,
571     zfsvfs);
572     (void) dsl_prop_unregister(ds, "vscan", vscan_changed_cb, zfsvfs);
573     return (error);
574
575 }
576
577 static int
578 zfs_space_delta_cb(dmu_object_type_t bonustype, void *data,
579     uint64_t *userp, uint64_t *group)
580 {
581     znode_phys_t *znp = data;
582     int error = 0;

```

```

584     /*
585     * Is it a valid type of object to track?
586     */
587     if (bonustype != DMU_OT_ZNODE && bonustype != DMU_OT_SA)
588         return (ENOENT);
589
590     /*
591     * If we have a NULL data pointer
592     * then assume the id's aren't changing and
593     * return EEXIST to the dmuf to let it know to
594     * use the same ids
595     */
596     if (data == NULL)
597         return (EEXIST);
598
599     if (bonustype == DMU_OT_ZNODE) {
600         *userp = znp->zp_uid;
601         *group = znp->zp_gid;
602     } else {
603         int hdrsize;
604
605         ASSERT(bonustype == DMU_OT_SA);
606         hdrsize = sa_hdrsize(data);
607
608         if (hdrsize != 0) {
609             *userp = *((uint64_t *)((uintptr_t)data + hdrsize +
610             SA_UID_OFFSET));
611             *group = *((uint64_t *)((uintptr_t)data + hdrsize +
612             SA_GID_OFFSET));
613         } else {
614             /*
615             * This should only happen for newly created
616             * files that haven't had the znode data filled
617             * in yet.
618             */
619             *userp = 0;
620             *group = 0;
621         }
622     }
623     return (error);
624 }
625
626 static void
627 fuidstr_to_sid(zfsvfs_t *zfsvfs, const char *fuidstr,
628     char *domainbuf, int buflen, uid_t *ridp)
629 {
630     uint64_t fuid;
631     const char *domain;
632
633     fuid = strtoum(fuidstr, NULL);
634
635     domain = zfs_fuid_find_by_idx(zfsvfs, FUID_INDEX(fuid));
636     if (domain)
637         (void) strncpy(domainbuf, domain, buflen);
638     else
639         domainbuf[0] = '\0';
640     *ridp = FUID_RID(fuid);
641 }
642
643 static uint64_t
644 zfs_userquota_prop_to_obj(zfsvfs_t *zfsvfs, zfs_userquota_prop_t type)
645 {
646     switch (type) {
647     case ZFS_PROP_USERUSED:
648         return (DMU_USERUSED_OBJECT);

```

```

649     case ZFS_PROP_GROUPUSED:
650         return (DMU_GROUPUSED_OBJECT);
651     case ZFS_PROP_USERQUOTA:
652         return (zfsvfs->z_userquota_obj);
653     case ZFS_PROP_GROUPQUOTA:
654         return (zfsvfs->z_groupquota_obj);
655     }
656     return (0);
657 }

659 int
660 zfs_userspace_many(zfsvfs_t *zfsvfs, zfs_userquota_prop_t type,
661     uint64_t *cookiep, void *vbuf, uint64_t *bufsizep)
662 {
663     int error;
664     zap_cursor_t zc;
665     zap_attribute_t za;
666     zfs_useracct_t *buf = vbuf;
667     uint64_t obj;

669     if (!dmu_objset_userspace_present(zfsvfs->z_os))
670         return (ENOTSUP);

672     obj = zfs_userquota_prop_to_obj(zfsvfs, type);
673     if (obj == 0) {
674         *bufsizep = 0;
675         return (0);
676     }

678     for (zap_cursor_init_serialized(&zc, zfsvfs->z_os, obj, *cookiep);
679         (error = zap_cursor_retrieve(&zc, &za)) == 0;
680         zap_cursor_advance(&zc)) {
681         if ((uintptr_t)buf - (uintptr_t)vbuf + sizeof (zfs_useracct_t) >
682             *bufsizep)
683             break;

685         fuidstr_to_sid(zfsvfs, za.za_name,
686             buf->zu_domain, sizeof (buf->zu_domain), &buf->zu_rid);

688         buf->zu_space = za.za_first_integer;
689         buf++;
690     }
691     if (error == ENOENT)
692         error = 0;

694     ASSERT3U((uintptr_t)buf - (uintptr_t)vbuf, <=, *bufsizep);
695     *bufsizep = (uintptr_t)buf - (uintptr_t)vbuf;
696     *cookiep = zap_cursor_serialize(&zc);
697     zap_cursor_fini(&zc);
698     return (error);
699 }

701 /*
702  * buf must be big enough (eg, 32 bytes)
703  */
704 static int
705 id_to_fuidstr(zfsvfs_t *zfsvfs, const char *domain, uid_t rid,
706     char *buf, boolean_t addok)
707 {
708     uint64_t fuid;
709     int domainid = 0;

711     if (domain && domain[0]) {
712         domainid = zfs_fuid_find_by_domain(zfsvfs, domain, NULL, addok);
713         if (domainid == -1)
714             return (ENOENT);

```

```

715     }
716     fuid = FUID_ENCODE(domainid, rid);
717     (void) sprintf(buf, "%llx", (longlong_t)fuid);
718     return (0);
719 }

721 int
722 zfs_userspace_one(zfsvfs_t *zfsvfs, zfs_userquota_prop_t type,
723     const char *domain, uint64_t rid, uint64_t *valp)
724 {
725     char buf[32];
726     int err;
727     uint64_t obj;

729     *valp = 0;

731     if (!dmu_objset_userspace_present(zfsvfs->z_os))
732         return (ENOTSUP);

734     obj = zfs_userquota_prop_to_obj(zfsvfs, type);
735     if (obj == 0)
736         return (0);

738     err = id_to_fuidstr(zfsvfs, domain, rid, buf, B_FALSE);
739     if (err)
740         return (err);

742     err = zap_lookup(zfsvfs->z_os, obj, buf, 8, 1, valp);
743     if (err == ENOENT)
744         err = 0;
745     return (err);
746 }

748 int
749 zfs_set_userquota(zfsvfs_t *zfsvfs, zfs_userquota_prop_t type,
750     const char *domain, uint64_t rid, uint64_t quota)
751 {
752     char buf[32];
753     int err;
754     dmu_tx_t *tx;
755     uint64_t *objp;
756     boolean_t fuid_dirtied;

758     if (type != ZFS_PROP_USERQUOTA && type != ZFS_PROP_GROUPQUOTA)
759         return (EINVAL);

761     if (zfsvfs->z_version < ZPL_VERSION_USERSPACE)
762         return (ENOTSUP);

764     objp = (type == ZFS_PROP_USERQUOTA) ? &zfsvfs->z_userquota_obj :
765         &zfsvfs->z_groupquota_obj;

767     err = id_to_fuidstr(zfsvfs, domain, rid, buf, B_TRUE);
768     if (err)
769         return (err);
770     fuid_dirtied = zfsvfs->z_fuid_dirty;

772     tx = dmu_tx_create(zfsvfs->z_os);
773     dmu_tx_hold_zap(tx, *objp ? *objp : DMU_NEW_OBJECT, B_TRUE, NULL);
774     if (*objp == 0) {
775         dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, B_TRUE,
776             zfs_userquota_prop_prefixes[type]);
777     }
778     if (fuid_dirtied)
779         zfs_fuid_txhold(zfsvfs, tx);
780     err = dmu_tx_assign(tx, TXG_WAIT);

```



```

781     if (err) {
782         dmu_tx_abort(tx);
783         return (err);
784     }

786     mutex_enter(&zfsvfs->z_lock);
787     if (*objp == 0) {
788         *objp = zap_create(zfsvfs->z_os, DMU_OT_USERGROUP_QUOTA,
789             DMU_OT_NONE, 0, tx);
790         VERIFY(0 == zap_add(zfsvfs->z_os, MASTER_NODE_OBJ,
791             zfs_userquota_prop_prefixes[type], 8, 1, objp, tx));
792     }
793     mutex_exit(&zfsvfs->z_lock);

795     if (quota == 0) {
796         err = zap_remove(zfsvfs->z_os, *objp, buf, tx);
797         if (err == ENOENT)
798             err = 0;
799     } else {
800         err = zap_update(zfsvfs->z_os, *objp, buf, 8, 1, &quota, tx);
801     }
802     ASSERT(err == 0);
803     if (fuid_dirtied)
804         zfs_fuid_sync(zfsvfs, tx);
805     dmu_tx_commit(tx);
806     return (err);
807 }

809 boolean_t
810 zfs_fuid_overquota(zfsvfs_t *zfsvfs, boolean_t isgroup, uint64_t fuid)
811 {
812     char buf[32];
813     uint64_t used, quota, usedobj, quotaobj;
814     int err;

816     usedobj = isgroup ? DMU_GROUPUSED_OBJECT : DMU_USERUSED_OBJECT;
817     quotaobj = isgroup ? zfsvfs->z_groupquota_obj : zfsvfs->z_userquota_obj;

819     if (quotaobj == 0 || zfsvfs->z_replay)
820         return (B_FALSE);

822     (void) sprintf(buf, "%llx", (longlong_t)fuid);
823     err = zap_lookup(zfsvfs->z_os, quotaobj, buf, 8, 1, &quota);
824     if (err != 0)
825         return (B_FALSE);

827     err = zap_lookup(zfsvfs->z_os, usedobj, buf, 8, 1, &used);
828     if (err != 0)
829         return (B_FALSE);
830     return (used >= quota);
831 }

833 boolean_t
834 zfs_owner_overquota(zfsvfs_t *zfsvfs, znode_t *zp, boolean_t isgroup)
835 {
836     uint64_t fuid;
837     uint64_t quotaobj;

839     quotaobj = isgroup ? zfsvfs->z_groupquota_obj : zfsvfs->z_userquota_obj;

841     fuid = isgroup ? zp->z_gid : zp->z_uid;

843     if (quotaobj == 0 || zfsvfs->z_replay)
844         return (B_FALSE);

846     return (zfs_fuid_overquota(zfsvfs, isgroup, fuid));

```

```

847 }

849 int
850 zfsvfs_create(const char *osname, zfsvfs_t **zfpv)
851 {
852     objset_t *os;
853     zfsvfs_t *zfsvfs;
854     uint64_t zval;
855     int i, error;
856     uint64_t sa_obj;

858     zfsvfs = kmem_zalloc(sizeof (zfsvfs_t), KM_SLEEP);

860     /*
861      * We claim to always be readonly so we can open snapshots;
862      * other ZPL code will prevent us from writing to snapshots.
863      */
864     error = dmu_objset_own(osname, DMU_OST_ZFS, B_TRUE, zfsvfs, &os);
865     if (error) {
866         kmem_free(zfsvfs, sizeof (zfsvfs_t));
867         return (error);
868     }

870     /*
871      * Initialize the zfs-specific filesystem structure.
872      * Should probably make this a kmem cache, shuffle fields,
873      * and just bzero up to z_hold_mtx[].
874      */
875     zfsvfs->z_vfs = NULL;
876     zfsvfs->z_parent = zfsvfs;
877     zfsvfs->z_max_blkisz = SPA_MAXBLOCKSIZE;
878     zfsvfs->z_show_ctldir = ZFS_SNAPDIR_VISIBLE;
879     zfsvfs->z_os = os;

881     error = zfs_get_zplprop(os, ZFS_PROP_VERSION, &zfsvfs->z_version);
882     if (error) {
883         goto out;
884     } else if (>zfsvfs->z_version >
885         zfs_zpl_version_map(spa_version(dmu_objset_spa(os)))) {
886         (void) printf("Can't mount a version %lld file system "
887             "on a version %lld pool\n. Pool must be upgraded to mount "
888             "this file system.", (u_longlong_t)zfsvfs->z_version,
889             (u_longlong_t)spa_version(dmu_objset_spa(os)));
890         error = ENOTSUP;
891         goto out;
892     }
893     if ((error = zfs_get_zplprop(os, ZFS_PROP_NORMALIZE, &zval)) != 0)
894         goto out;
895     zfsvfs->z_norm = (int)zval;

897     if ((error = zfs_get_zplprop(os, ZFS_PROP_UTF8ONLY, &zval)) != 0)
898         goto out;
899     zfsvfs->z_utf8 = (zval != 0);

901     if ((error = zfs_get_zplprop(os, ZFS_PROP_CASE, &zval)) != 0)
902         goto out;
903     zfsvfs->z_case = (uint_t)zval;

905     /*
906      * Fold case on file systems that are always or sometimes case
907      * insensitive.
908      */
909     if (zfsvfs->z_case == ZFS_CASE_INSENSITIVE ||
910         zfsvfs->z_case == ZFS_CASE_MIXED)
911         zfsvfs->z_norm |= U8_TEXTPREP_TOUPPER;

```

```

913 zfsvfs->z_use_fuids = USE_FUIDS(zfsvfs->z_version, zfsvfs->z_os);
914 zfsvfs->z_use_sa = USE_SA(zfsvfs->z_version, zfsvfs->z_os);

916 if (zfsvfs->z_use_sa) {
917     /* should either have both of these objects or none */
918     error = zap_lookup(os, MASTER_NODE_OBJ, ZFS_SA_ATTRS, 8, 1,
919         &sa_obj);
920     if (error)
921         return (error);
922 } else {
923     /*
924      * Pre SA versions file systems should never touch
925      * either the attribute registration or layout objects.
926      */
927     sa_obj = 0;
928 }

930 error = sa_setup(os, sa_obj, zfs_attr_table, ZPL_END,
931     &zfsvfs->z_attr_table);
932 if (error)
933     goto out;

935 if (zfsvfs->z_version >= ZPL_VERSION_SA)
936     sa_register_update_callback(os, zfs_sa_upgrade);

938 error = zap_lookup(os, MASTER_NODE_OBJ, ZFS_ROOT_OBJ, 8, 1,
939     &zfsvfs->z_root);
940 if (error)
941     goto out;
942 ASSERT(zfsvfs->z_root != 0);

944 error = zap_lookup(os, MASTER_NODE_OBJ, ZFS_UNLINKED_SET, 8, 1,
945     &zfsvfs->z_unlinkedobj);
946 if (error)
947     goto out;

949 error = zap_lookup(os, MASTER_NODE_OBJ,
950     zfs_userquota_prop_prefixes[ZFS_PROP_USERQUOTA],
951     8, 1, &zfsvfs->z_userquota_obj);
952 if (error && error != ENOENT)
953     goto out;

955 error = zap_lookup(os, MASTER_NODE_OBJ,
956     zfs_userquota_prop_prefixes[ZFS_PROP_GROUPQUOTA],
957     8, 1, &zfsvfs->z_groupquota_obj);
958 if (error && error != ENOENT)
959     goto out;

961 error = zap_lookup(os, MASTER_NODE_OBJ, ZFS_FUID_TABLES, 8, 1,
962     &zfsvfs->z_fuid_obj);
963 if (error && error != ENOENT)
964     goto out;

966 error = zap_lookup(os, MASTER_NODE_OBJ, ZFS_SHARES_DIR, 8, 1,
967     &zfsvfs->z_shares_dir);
968 if (error && error != ENOENT)
969     goto out;

971 mutex_init(&zfsvfs->z_znodes_lock, NULL, MUTEX_DEFAULT, NULL);
972 mutex_init(&zfsvfs->z_lock, NULL, MUTEX_DEFAULT, NULL);
973 list_create(&zfsvfs->z_all_znodes, sizeof (znode_t),
974     offsetof(znode_t, z_link_node));
975 rrw_init(&zfsvfs->z_teardown_lock);
976 rw_init(&zfsvfs->z_teardown_inactive_lock, NULL, RW_DEFAULT, NULL);
977 rw_init(&zfsvfs->z_fuid_lock, NULL, RW_DEFAULT, NULL);
978 for (i = 0; i != ZFS_OBJ_MTX_SZ; i++)

```

```

979     mutex_init(&zfsvfs->z_hold_mtx[i], NULL, MUTEX_DEFAULT, NULL);

981     *zfvp = zfsvfs;
982     return (0);

984 out:
985     dmu_objset_disown(os, zfsvfs);
986     *zfvp = NULL;
987     kmem_free(zfsvfs, sizeof (zfsvfs_t));
988     return (error);
989 }

991 static int
992 zfsvfs_setup(zfsvfs_t *zfsvfs, boolean_t mounting)
993 {
994     int error;

996     error = zfs_register_callbacks(zfsvfs->z_vfs);
997     if (error)
998         return (error);

1000     /*
1001      * Set the objset user_ptr to track its zfsvfs.
1002      */
1003     mutex_enter(&zfsvfs->z_os->os_user_ptr_lock);
1004     dmu_objset_set_user(zfsvfs->z_os, zfsvfs);
1005     mutex_exit(&zfsvfs->z_os->os_user_ptr_lock);

1007     zfsvfs->z_log = zil_open(zfsvfs->z_os, zfs_get_data);

1009     /*
1010      * If we are not mounting (ie: online recv), then we don't
1011      * have to worry about replaying the log as we blocked all
1012      * operations out since we closed the ZIL.
1013      */
1014     if (mounting) {
1015         boolean_t readonly;

1017         /*
1018          * During replay we remove the read only flag to
1019          * allow replays to succeed.
1020          */
1021         readonly = zfsvfs->z_vfs->vfs_flag & VFS_RDONLY;
1022         if (readonly != 0)
1023             zfsvfs->z_vfs->vfs_flag &= ~VFS_RDONLY;
1024         else
1025             zfs_unlinked_drain(zfsvfs);

1027         /*
1028          * Parse and replay the intent log.
1029          *
1030          * Because of ziltest, this must be done after
1031          * zfs_unlinked_drain(). (Further note: ziltest
1032          * doesn't use readonly mounts, where
1033          * zfs_unlinked_drain() isn't called.) This is because
1034          * ziltest causes spa_sync() to think it's committed,
1035          * but actually it is not, so the intent log contains
1036          * many txg's worth of changes.
1037          *
1038          * In particular, if object N is in the unlinked set in
1039          * the last txg to actually sync, then it could be
1040          * actually freed in a later txg and then reallocated
1041          * in a yet later txg. This would write a "create
1042          * object N" record to the intent log. Normally, this
1043          * would be fine because the spa_sync() would have
1044          * written out the fact that object N is free, before

```

```

1045     * we could write the "create object N" intent log
1046     * record.
1047     *
1048     * But when we are in ziltest mode, we advance the "open
1049     * txg" without actually spa_sync()-ing the changes to
1050     * disk. So we would see that object N is still
1051     * allocated and in the unlinked set, and there is an
1052     * intent log record saying to allocate it.
1053     */
1054     if (spa_writeable(dmu_objset_spa(zfsvfs->z_os)) {
1055         if (zil_replay_disable) {
1056             zil_destroy(zfsvfs->z_log, B_FALSE);
1057         } else {
1058             zfsvfs->z_replay = B_TRUE;
1059             zil_replay(zfsvfs->z_os, zfsvfs,
1060                 zfs_replay_vector);
1061             zfsvfs->z_replay = B_FALSE;
1062         }
1063     }
1064     zfsvfs->z_vfs->vfs_flag |= readonly; /* restore readonly bit */
1065 }
1066
1067 return (0);
1068 }
1069
1070 void
1071 zfsvfs_free(zfsvfs_t *zfsvfs)
1072 {
1073     int i;
1074     extern krwlock_t zfsvfs_lock; /* in zfs_znode.c */
1075
1076     /*
1077     * This is a barrier to prevent the filesystem from going away in
1078     * zfs_znode_move() until we can safely ensure that the filesystem is
1079     * not unmounted. We consider the filesystem valid before the barrier
1080     * and invalid after the barrier.
1081     */
1082     rw_enter(&zfsvfs_lock, RW_READER);
1083     rw_exit(&zfsvfs_lock);
1084
1085     zfs_fuid_destroy(zfsvfs);
1086
1087     mutex_destroy(&zfsvfs->z_znodes_lock);
1088     mutex_destroy(&zfsvfs->z_lock);
1089     list_destroy(&zfsvfs->z_all_znodes);
1090     rrw_destroy(&zfsvfs->z_teardown_lock);
1091     rw_destroy(&zfsvfs->z_teardown_inactive_lock);
1092     rw_destroy(&zfsvfs->z_fuid_lock);
1093     for (i = 0; i != ZFS_OBJ_MTX_SZ; i++)
1094         mutex_destroy(&zfsvfs->z_hold_mtx[i]);
1095     kmem_free(zfsvfs, sizeof(zfsvfs_t));
1096 }
1097
1098 static void
1099 zfs_set_fuid_feature(zfsvfs_t *zfsvfs)
1100 {
1101     zfsvfs->z_use_fuids = USE_FUIDS(zfsvfs->z_version, zfsvfs->z_os);
1102     if (zfsvfs->z_vfs) {
1103         if (zfsvfs->z_use_fuids) {
1104             vfs_set_feature(zfsvfs->z_vfs, VFSFT_XVATTR);
1105             vfs_set_feature(zfsvfs->z_vfs, VFSFT_SYSATTR_VIEWS);
1106             vfs_set_feature(zfsvfs->z_vfs, VFSFT_ACEMASKONACCESS);
1107             vfs_set_feature(zfsvfs->z_vfs, VFSFT_ACLONCREATE);
1108             vfs_set_feature(zfsvfs->z_vfs, VFSFT_ACCESS_FILTER);
1109             vfs_set_feature(zfsvfs->z_vfs, VFSFT_REPARSE);
1110         } else {

```

```

1111         vfs_clear_feature(zfsvfs->z_vfs, VFSFT_XVATTR);
1112         vfs_clear_feature(zfsvfs->z_vfs, VFSFT_SYSATTR_VIEWS);
1113         vfs_clear_feature(zfsvfs->z_vfs, VFSFT_ACEMASKONACCESS);
1114         vfs_clear_feature(zfsvfs->z_vfs, VFSFT_ACLONCREATE);
1115         vfs_clear_feature(zfsvfs->z_vfs, VFSFT_ACCESS_FILTER);
1116         vfs_clear_feature(zfsvfs->z_vfs, VFSFT_REPARSE);
1117     }
1118 }
1119     zfsvfs->z_use_sa = USE_SA(zfsvfs->z_version, zfsvfs->z_os);
1120 }
1121
1122 static int
1123 zfs_domount(vfs_t *vfsp, char *osname)
1124 {
1125     dev_t mount_dev;
1126     uint64_t recordsize, fsid_guid;
1127     int error = 0;
1128     zfsvfs_t *zfsvfs;
1129
1130     ASSERT(vfsp);
1131     ASSERT(osname);
1132
1133     error = zfsvfs_create(osname, &zfsvfs);
1134     if (error)
1135         return (error);
1136     zfsvfs->z_vfs = vfsp;
1137
1138     /* Initialize the generic filesystem structure. */
1139     vfsp->vfs_bcount = 0;
1140     vfsp->vfs_data = NULL;
1141
1142     if (zfs_create_unique_device(&mount_dev) == -1) {
1143         error = ENODEV;
1144         goto out;
1145     }
1146     ASSERT(vfs_devismounted(mount_dev) == 0);
1147
1148     if (error = dsl_prop_get_integer(osname, "recordsize", &recordsize,
1149         NULL))
1150         goto out;
1151
1152     vfsp->vfs_dev = mount_dev;
1153     vfsp->vfs_fstype = zfsfstype;
1154     vfsp->vfs_bsize = recordsize;
1155     vfsp->vfs_flag |= VFS_NOTRUNC;
1156     vfsp->vfs_data = zfsvfs;
1157
1158     /*
1159     * The fsid is 64 bits, composed of an 8-bit fs type, which
1160     * separates our fsid from any other filesystem types, and a
1161     * 56-bit objset unique ID. The objset unique ID is unique to
1162     * all objsets open on this system, provided by unique_create().
1163     * The 8-bit fs type must be put in the low bits of fsid[1]
1164     * because that's where other Solaris filesystems put it.
1165     */
1166     fsid_guid = dmu_objset_fsid_guid(zfsvfs->z_os);
1167     ASSERT((fsid_guid & ~((1ULL<<56)-1)) == 0);
1168     vfsp->vfs_fsid.val[0] = fsid_guid;
1169     vfsp->vfs_fsid.val[1] = ((fsid_guid>>32) << 8) |
1170         zfsfstype & 0xFF;
1171
1172     /*
1173     * Set features for file system.
1174     */
1175     zfs_set_fuid_feature(zfsvfs);
1176     if (zfsvfs->z_case == ZFS_CASE_INSENSITIVE) {

```

```

1177     vfs_set_feature(vfsp, VFSFT_DIRENTFLAGS);
1178     vfs_set_feature(vfsp, VFSFT_CASEINSENSITIVE);
1179     vfs_set_feature(vfsp, VFSFT_NOCASESENSITIVE);
1180 } else if (zfsvfs->z_case == ZFS_CASE_MIXED) {
1181     vfs_set_feature(vfsp, VFSFT_DIRENTFLAGS);
1182     vfs_set_feature(vfsp, VFSFT_CASEINSENSITIVE);
1183 }
1184 vfs_set_feature(vfsp, VFSFT_ZEROCOPY_SUPPORTED);

1186 if (dmu_objset_is_snapshot(zfsvfs->z_os)) {
1187     uint64_t pval;

1189     atime_changed_cb(zfsvfs, B_FALSE);
1190     readonly_changed_cb(zfsvfs, B_TRUE);
1191     if (error = dsl_prop_get_integer(osname, "xattr", &pval, NULL))
1192         goto out;
1193     xattr_changed_cb(zfsvfs, pval);
1194     zfsvfs->z_issnap = B_TRUE;
1195     zfsvfs->z_os->os_sync = ZFS_SYNC_DISABLED;

1197     mutex_enter(&zfsvfs->z_os->os_user_ptr_lock);
1198     dmu_objset_set_user(zfsvfs->z_os, zfsvfs);
1199     mutex_exit(&zfsvfs->z_os->os_user_ptr_lock);
1200 } else {
1201     error = zfsvfs_setup(zfsvfs, B_TRUE);
1202 }

1204 if (!zfsvfs->z_issnap)
1205     zfsctl_create(zfsvfs);
1206 out:
1207 if (error) {
1208     dmu_objset_disown(zfsvfs->z_os, zfsvfs);
1209     zfsvfs_free(zfsvfs);
1210 } else {
1211     atomic_add_32(&zfs_active_fs_count, 1);
1212 }

1214 return (error);
1215 }

1217 void
1218 zfs_unregister_callbacks(zfsvfs_t *zfsvfs)
1219 {
1220     objset_t *os = zfsvfs->z_os;
1221     struct dsl_dataset *ds;

1223     /*
1224      * Unregister properties.
1225      */
1226     if (!dmu_objset_is_snapshot(os)) {
1227         ds = dmu_objset_ds(os);
1228         VERIFY(dsl_prop_unregister(ds, "atime", atime_changed_cb,
1229             zfsvfs) == 0);

1231         VERIFY(dsl_prop_unregister(ds, "xattr", xattr_changed_cb,
1232             zfsvfs) == 0);

1234         VERIFY(dsl_prop_unregister(ds, "recordsize", blksize_changed_cb,
1235             zfsvfs) == 0);

1237         VERIFY(dsl_prop_unregister(ds, "readonly", readonly_changed_cb,
1238             zfsvfs) == 0);

1240         VERIFY(dsl_prop_unregister(ds, "devices", devices_changed_cb,
1241             zfsvfs) == 0);

```

```

1243     VERIFY(dsl_prop_unregister(ds, "setuid", setuid_changed_cb,
1244         zfsvfs) == 0);

1246     VERIFY(dsl_prop_unregister(ds, "exec", exec_changed_cb,
1247         zfsvfs) == 0);

1249     VERIFY(dsl_prop_unregister(ds, "snapdir", snapdir_changed_cb,
1250         zfsvfs) == 0);

1252     VERIFY(dsl_prop_unregister(ds, "aclmode", acl_mode_changed_cb,
1253         zfsvfs) == 0);

1255     VERIFY(dsl_prop_unregister(ds, "aclinherit",
1256         acl_inherit_changed_cb, zfsvfs) == 0);

1258     VERIFY(dsl_prop_unregister(ds, "vscan",
1259         vscan_changed_cb, zfsvfs) == 0);
1260 }
1261 }

1263 /*
1264  * Convert a decimal digit string to a uint64_t integer.
1265  */
1266 static int
1267 str_to_uint64(char *str, uint64_t *objnum)
1268 {
1269     uint64_t num = 0;

1271     while (*str) {
1272         if (*str < '0' || *str > '9')
1273             return (EINVAL);

1275         num = num*10 + *str++ - '0';
1276     }

1278     *objnum = num;
1279     return (0);
1280 }

1282 /*
1283  * The boot path passed from the boot loader is in the form of
1284  * "rootpool-name/root-filesystem-object-number". Convert this
1285  * string to a dataset name: "rootpool-name/root-filesystem-name".
1286  */
1287 static int
1288 zfs_parse_bootfs(char *bpath, char *outpath)
1289 {
1290     char *slashp;
1291     uint64_t objnum;
1292     int error;

1294     if (*bpath == 0 || *bpath == '/')
1295         return (EINVAL);

1297     (void) strcpy(outpath, bpath);

1299     slashp = strchr(bpath, '/');

1301     /* if no '/', just return the pool name */
1302     if (slashp == NULL) {
1303         return (0);
1304     }

1306     /* if not a number, just return the root dataset name */
1307     if (str_to_uint64(slashp+1, &objnum)) {
1308         return (0);

```

```

1309     }
1310
1311     *slashp = '\0';
1312     error = dsl_dsobj_to_dsname(bpath, objnum, outpath);
1313     *slashp = '/';
1314
1315     return (error);
1316 }
1317
1318 /*
1319  * zfs_check_global_label:
1320  * Check that the hex label string is appropriate for the dataset
1321  * being mounted into the global_zone proper.
1322  *
1323  * Return an error if the hex label string is not default or
1324  * admin_low/admin_high. For admin_low labels, the corresponding
1325  * dataset must be readonly.
1326  */
1327 int
1328 zfs_check_global_label(const char *dsname, const char *hexsl)
1329 {
1330     if (strcasecmp(hexsl, ZFS_MLSLABEL_DEFAULT) == 0)
1331         return (0);
1332     if (strcasecmp(hexsl, ADMIN_HIGH) == 0)
1333         return (0);
1334     if (strcasecmp(hexsl, ADMIN_LOW) == 0) {
1335         /* must be readonly */
1336         uint64_t ronly;
1337
1338         if (dsl_prop_get_integer(dsname,
1339             zfs_prop_to_name(ZFS_PROP_READONLY), &ronly, NULL))
1340             return (EACCES);
1341         return (ronly ? 0 : EACCES);
1342     }
1343     return (EACCES);
1344 }
1345
1346 /*
1347  * zfs_mount_label_policy:
1348  * Determine whether the mount is allowed according to MAC check.
1349  * by comparing (where appropriate) label of the dataset against
1350  * the label of the zone being mounted into. If the dataset has
1351  * no label, create one.
1352  *
1353  * Returns:
1354  *     0 : access allowed
1355  *     >0 : error code, such as EACCES
1356  */
1357 static int
1358 zfs_mount_label_policy(vfs_t *vfsp, char *osname)
1359 {
1360     int error, retv;
1361     zone_t *mntzone = NULL;
1362     ts_label_t *mnt_tsl;
1363     bslabel_t *mnt_sl;
1364     bslabel_t ds_sl;
1365     char ds_hexsl[MAXNAMELEN];
1366
1367     retv = EACCES; /* assume the worst */
1368
1369     /*
1370      * Start by getting the dataset label if it exists.
1371      */
1372     error = dsl_prop_get(osname, zfs_prop_to_name(ZFS_PROP_MLSLABEL),
1373         1, sizeof (ds_hexsl), &ds_hexsl, NULL);
1374     if (error)

```

```

1375         return (EACCES);
1376
1377     /*
1378      * If labeling is NOT enabled, then disallow the mount of datasets
1379      * which have a non-default label already. No other label checks
1380      * are needed.
1381      */
1382     if (!is_system_labeled()) {
1383         if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) == 0)
1384             return (0);
1385         return (EACCES);
1386     }
1387
1388     /*
1389      * Get the label of the mountpoint. If mounting into the global
1390      * zone (i.e. mountpoint is not within an active zone and the
1391      * zoned property is off), the label must be default or
1392      * admin_low/admin_high only; no other checks are needed.
1393      */
1394     mntzone = zone_find_by_any_path(refstr_value(vfsp->vfs_mntpt), B_FALSE);
1395     if (mntzone->zone_id == GLOBAL_ZONEID) {
1396         uint64_t zoned;
1397
1398         zone_rele(mntzone);
1399
1400         if (dsl_prop_get_integer(osname,
1401             zfs_prop_to_name(ZFS_PROP_ZONED), &zoned, NULL))
1402             return (EACCES);
1403         if (!zoned)
1404             return (zfs_check_global_label(osname, ds_hexsl));
1405         else
1406             /*
1407              * This is the case of a zone dataset being mounted
1408              * initially, before the zone has been fully created;
1409              * allow this mount into global zone.
1410              */
1411             return (0);
1412     }
1413
1414     mnt_tsl = mntzone->zone_sl;
1415     ASSERT(mnt_tsl != NULL);
1416     label_hold(mnt_tsl);
1417     mnt_sl = label2bslabel(mnt_tsl);
1418
1419     if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) == 0) {
1420         /*
1421          * The dataset doesn't have a real label, so fabricate one.
1422          */
1423         char *str = NULL;
1424
1425         if (l_to_str_internal(mnt_sl, &str) == 0 &&
1426             dsl_prop_set(osname, zfs_prop_to_name(ZFS_PROP_MLSLABEL),
1427                 ZPROP_SRC_LOCAL, 1, strlen(str) + 1, str) == 0)
1428             retv = 0;
1429         if (str != NULL)
1430             kmem_free(str, strlen(str) + 1);
1431     } else if (hexstr_to_label(ds_hexsl, &ds_sl) == 0) {
1432         /*
1433          * Now compare labels to complete the MAC check. If the
1434          * labels are equal then allow access. If the mountpoint
1435          * label dominates the dataset label, allow readonly access.
1436          * Otherwise, access is denied.
1437          */
1438         if (blequal(mnt_sl, &ds_sl))
1439             retv = 0;
1440         else if (bldominates(mnt_sl, &ds_sl)) {

```

```

1441         vfs_setmntopt(vfsp, MNTOPT_RO, NULL, 0);
1442         retv = 0;
1443     }
1444 }

1446     label_rele(mnt_tsl);
1447     zone_rele(mntzone);
1448     return (retv);
1449 }

1451 static int
1452 zfs_mountroot(vfs_t *vfsp, enum whymountroot why)
1453 {
1454     int error = 0;
1455     static int zfsrootdone = 0;
1456     zfsvfs_t *zfsvfs = NULL;
1457     znode_t *zp = NULL;
1458     vnode_t *vp = NULL;
1459     char *zfs_bootfs;
1460     char *zfs_devid;

1462     ASSERT(vfsp);

1464     /*
1465      * The filesystem that we mount as root is defined in the
1466      * boot property "zfs-bootfs" with a format of
1467      * "poolname/root-dataset-objnum".
1468      */
1469     if (why == ROOT_INIT) {
1470         if (zfsrootdone++)
1471             return (EBUSY);
1472         /*
1473          * the process of doing a spa_load will require the
1474          * clock to be set before we could (for example) do
1475          * something better by looking at the timestamp on
1476          * an uberblock, so just set it to -1.
1477          */
1478         clkset(-1);

1480         if ((zfs_bootfs = spa_get_bootprop("zfs-bootfs")) == NULL) {
1481             cmn_err(CE_NOTE, "spa_get_bootfs: can not get "
1482                  "bootfs name");
1483             return (EINVAL);
1484         }
1485         zfs_devid = spa_get_bootprop("diskdevid");
1486         error = spa_import_rootpool(rootfs.bo_name, zfs_devid);
1487         if (zfs_devid)
1488             spa_free_bootprop(zfs_devid);
1489         if (error) {
1490             spa_free_bootprop(zfs_bootfs);
1491             cmn_err(CE_NOTE, "spa_import_rootpool: error %d",
1492                  error);
1493             return (error);
1494         }
1495         if (error = zfs_parse_bootfs(zfs_bootfs, rootfs.bo_name)) {
1496             spa_free_bootprop(zfs_bootfs);
1497             cmn_err(CE_NOTE, "zfs_parse_bootfs: error %d",
1498                  error);
1499             return (error);
1500         }

1502         spa_free_bootprop(zfs_bootfs);

1504         if (error = vfs_lock(vfsp))
1505             return (error);

```

```

1507         if (error = zfs_domount(vfsp, rootfs.bo_name)) {
1508             cmn_err(CE_NOTE, "zfs_domount: error %d", error);
1509             goto out;
1510         }

1512         zfsvfs = (zfsvfs_t *)vfsp->vfs_data;
1513         ASSERT(zfsvfs);
1514         if (error = zfs_zget(zfsvfs, zfsvfs->z_root, &zp)) {
1515             cmn_err(CE_NOTE, "zfs_zget: error %d", error);
1516             goto out;
1517         }

1519         vp = ZTOV(zp);
1520         mutex_enter(&vp->v_lock);
1521         vp->v_flag |= VROOT;
1522         mutex_exit(&vp->v_lock);
1523         rootvp = vp;

1525         /*
1526          * Leave rootvp held. The root file system is never unmounted.
1527          */

1529         vfs_add((struct vnode *)0, vfsp,
1530              (vfsp->vfs_flag & VFS_RDONLY) ? MS_RDONLY : 0);
1531     out:
1532         vfs_unlock(vfsp);
1533         return (error);
1534     } else if (why == ROOT_REMOUNT) {
1535         readonly_changed_cb(vfsp->vfs_data, B_FALSE);
1536         vfsp->vfs_flag |= VFS_REMOUNT;

1538         /* refresh mount options */
1539         zfs_unregister_callbacks(vfsp->vfs_data);
1540         return (zfs_register_callbacks(vfsp));

1542     } else if (why == ROOT_UNMOUNT) {
1543         zfs_unregister_callbacks((zfsvfs_t *)vfsp->vfs_data);
1544         (void) zfs_sync(vfsp, 0, 0);
1545         return (0);
1546     }

1548     /*
1549      * if "why" is equal to anything else other than ROOT_INIT,
1550      * ROOT_REMOUNT, or ROOT_UNMOUNT, we do not support it.
1551      */
1552     return (ENOTSUP);
1553 }

1555 /*ARGSUSED*/
1556 static int
1557 zfs_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
1558 {
1559     char          *osname;
1560     pathname_t    spn;
1561     int           error = 0;
1562     uio_seg_t     fromspace = (uap->flags & MS_SYSSPACE) ?
1563         UIO_SYSSPACE : UIO_USERSPACE;
1564     int           canwrite;

1566     if (mvp->v_type != VDIR)
1567         return (ENOTDIR);

1569     mutex_enter(&mvp->v_lock);
1570     if ((uap->flags & MS_REMOUNT) == 0 &&
1571         (uap->flags & MS_OVERLAY) == 0 &&
1572         (mvp->v_count != 1 || (mvp->v_flag & VROOT))) {

```

```

1573         mutex_exit(&mvp->v_lock);
1574         return (EBUSY);
1575     }
1576     mutex_exit(&mvp->v_lock);

1578     /*
1579     * ZFS does not support passing unparsed data in via MS_DATA.
1580     * Users should use the MS_OPTIONSTR interface; this means
1581     * that all option parsing is already done and the options struct
1582     * can be interrogated.
1583     */
1584     if ((uap->flags & MS_DATA) && uap->datalen > 0)
1585         return (EINVAL);

1587     /*
1588     * Get the objset name (the "special" mount argument).
1589     */
1590     if (error = pn_get(uap->spec, fromspace, &spn))
1591         return (error);

1593     osname = spn.pn_path;

1595     /*
1596     * Check for mount privilege?
1597     *
1598     * If we don't have privilege then see if
1599     * we have local permission to allow it
1600     */
1601     error = secpolicy_fs_mount(cr, mvp, vfsp);
1602     if (error) {
1603         if (dsl_deleg_access(osname, ZFS_DELEG_PERM_MOUNT, cr) == 0) {
1604             vattr_t          vattr;

1606             /*
1607             * Make sure user is the owner of the mount point
1608             * or has sufficient privileges.
1609             */

1611             vattr.va_mask = AT_UID;

1613             if (VOP_GETATTR(mvp, &vattr, 0, cr, NULL)) {
1614                 goto out;
1615             }

1617             if (secpolicy_vnode_owner(cr, vattr.va_uid) != 0 &&
1618                 VOP_ACCESS(mvp, VWRITE, 0, cr, NULL) != 0) {
1619                 goto out;
1620             }
1621             secpolicy_fs_mount_clearopts(cr, vfsp);
1622         } else {
1623             goto out;
1624         }
1625     }

1627     /*
1628     * Refuse to mount a filesystem if we are in a local zone and the
1629     * dataset is not visible.
1630     */
1631     if (!INGLOBALZONE(curproc) &&
1632         (!zone_dataset_visible(osname, &canwrite) || !canwrite)) {
1633         error = EPERM;
1634         goto out;
1635     }

1637     error = zfs_mount_label_policy(vfsp, osname);
1638     if (error)

```

```

1639         goto out;

1641     /*
1642     * When doing a remount, we simply refresh our temporary properties
1643     * according to those options set in the current VFS options.
1644     */
1645     if (uap->flags & MS_REMOUNT) {
1646         /* refresh mount options */
1647         zfs_unregister_callbacks(vfsp->vfs_data);
1648         error = zfs_register_callbacks(vfsp);
1649         goto out;
1650     }

1652     error = zfs_domount(vfsp, osname);

1654     /*
1655     * Add an extra VFS_HOLD on our parent vfs so that it can't
1656     * disappear due to a forced unmount.
1657     */
1658     if (error == 0 && ((zfsvfs_t *)vfsp->vfs_data)->z_issnap)
1659         VFS_HOLD(mvp->v_vfsp);

1661 out:
1662     pn_free(&spn);
1663     return (error);
1664 }

1666 static int
1667 zfs_statvfs(vfs_t *vfsp, struct statvfs64 *statp)
1668 {
1669     zfsvfs_t *zfsvfs = vfsp->vfs_data;
1670     dev32_t d32;
1671     uint64_t refdbytes, availbytes, usedobjs, availobjs;

1673     ZFS_ENTER(zfsvfs);

1675     dmu_objset_space(zfsvfs->z_os,
1676         &refdbytes, &availbytes, &usedobjs, &availobjs);

1678     /*
1679     * The underlying storage pool actually uses multiple block sizes.
1680     * We report the fragsize as the smallest block size we support,
1681     * and we report our blocksize as the filesystem's maximum blocksize.
1682     */
1683     statp->f_frsize = 1UL << SPA_MINBLOCKSHIFT;
1684     statp->f_bsize = zfsvfs->z_max_blkz;

1686     /*
1687     * The following report "total" blocks of various kinds in the
1688     * file system, but reported in terms of f_frsize - the
1689     * "fragment" size.
1690     */

1692     statp->f_blocks = (refdbytes + availbytes) >> SPA_MINBLOCKSHIFT;
1693     statp->f_bfree = availbytes >> SPA_MINBLOCKSHIFT;
1694     statp->f_bavail = statp->f_bfree; /* no root reservation */

1696     /*
1697     * statvfs() should really be called statufs(), because it assumes
1698     * static metadata. ZFS doesn't preallocate files, so the best
1699     * we can do is report the max that could possibly fit in f_files,
1700     * and that minus the number actually used in f_ffree.
1701     * For f_ffree, report the smaller of the number of object available
1702     * and the number of blocks (each object will take at least a block).
1703     */
1704     statp->f_ffree = MIN(availobjs, statp->f_bfree);

```

```

1705 statp->f_favail = statp->f_ffree; /* no "root reservation" */
1706 statp->f_files = statp->f_ffree + usedobjs;

1708 (void) cmpldev(&d32, vfsp->vfs_dev);
1709 statp->f_fsid = d32;

1711 /*
1712  * We're a zfs filesystem.
1713  */
1714 (void) strcpy(statp->f_basetype, vfssw[vfsp->vfsfstype].vsw_name);

1716 statp->f_flag = vf_to_stf(vfsp->vfs_flag);

1718 statp->f_namemax = ZFS_MAXNAMELEN;

1720 /*
1721  * We have all of 32 characters to stuff a string here.
1722  * Is there anything useful we could/should provide?
1723  */
1724 bzero(statp->f_fstr, sizeof (statp->f_fstr));

1726 ZFS_EXIT(zfsvfs);
1727 return (0);
1728 }

1730 static int
1731 zfs_root(vfs_t *vfsp, vnode_t **vpp)
1732 {
1733     zfsvfs_t *zfsvfs = vfsp->vfs_data;
1734     znode_t *rootzp;
1735     int error;

1737     ZFS_ENTER(zfsvfs);

1739     error = zfs_zget(zfsvfs, zfsvfs->z_root, &rootzp);
1740     if (error == 0)
1741         *vpp = ZTOV(rootzp);

1743     ZFS_EXIT(zfsvfs);
1744     return (error);
1745 }

1747 /*
1748  * Teardown the zfsvfs::z_os.
1749  *
1750  * Note, if 'unmounting' is FALSE, we return with the 'z_teardown_lock'
1751  * and 'z_teardown_inactive_lock' held.
1752  */
1753 static int
1754 zfsvfs_teardown(zfsvfs_t *zfsvfs, boolean_t unmounting)
1755 {
1756     znode_t *zp;

1758     rrw_enter(&zfsvfs->z_teardown_lock, RW_WRITER, FTAG);

1760     if (!unmounting) {
1761         /*
1762          * We purge the parent filesystem's vfsp as the parent
1763          * filesystem and all of its snapshots have their vnode's
1764          * v_vfsp set to the parent's filesystem's vfsp. Note,
1765          * 'z_parent' is self referential for non-snapshots.
1766          */
1767         (void) dnlc_purge_vfsp(zfsvfs->z_parent->z_vfs, 0);
1768     }

1770     /*

```

```

1771     * Close the zil. NB: Can't close the zil while zfs_inactive
1772     * threads are blocked as zil_close can call zfs_inactive.
1773     */
1774     if (zfsvfs->z_log) {
1775         zil_close(zfsvfs->z_log);
1776         zfsvfs->z_log = NULL;
1777     }

1779     rw_enter(&zfsvfs->z_teardown_inactive_lock, RW_WRITER);

1781     /*
1782     * If we are not unmounting (ie: online recv) and someone already
1783     * unmounted this file system while we were doing the switchover,
1784     * or a reopen of z_os failed then just bail out now.
1785     */
1786     if (!unmounting && (zfsvfs->z_unmounted || zfsvfs->z_os == NULL)) {
1787         rw_exit(&zfsvfs->z_teardown_inactive_lock);
1788         rrw_exit(&zfsvfs->z_teardown_lock, FTAG);
1789         return (EIO);
1790     }

1792     /*
1793     * At this point there are no vops active, and any new vops will
1794     * fail with EIO since we have z_teardown_lock for writer (only
1795     * relevant for forced unmount).
1796     *
1797     * Release all holds on dbufs.
1798     */
1799     mutex_enter(&zfsvfs->z_znodes_lock);
1800     for (zp = list_head(&zfsvfs->z_all_znodes); zp != NULL;
1801          zp = list_next(&zfsvfs->z_all_znodes, zp))
1802         if (zp->z_sa_hdl) {
1803             ASSERT(ZTOV(zp)->v_count > 0);
1804             zfs_znode_dmu_fini(zp);
1805         }
1806     mutex_exit(&zfsvfs->z_znodes_lock);

1808     /*
1809     * If we are unmounting, set the unmounted flag and let new vops
1810     * unblock. zfs_inactive will have the unmounted behavior, and all
1811     * other vops will fail with EIO.
1812     */
1813     if (unmounting) {
1814         zfsvfs->z_unmounted = B_TRUE;
1815         rrw_exit(&zfsvfs->z_teardown_lock, FTAG);
1816         rw_exit(&zfsvfs->z_teardown_inactive_lock);
1817     }

1819     /*
1820     * z_os will be NULL if there was an error in attempting to reopen
1821     * zfsvfs, so just return as the properties had already been
1822     * unregistered and cached data had been evicted before.
1823     */
1824     if (zfsvfs->z_os == NULL)
1825         return (0);

1827     /*
1828     * Unregister properties.
1829     */
1830     zfs_unregister_callbacks(zfsvfs);

1832     /*
1833     * Evict cached data
1834     */
1835     if (dmu_objset_is_dirty_anywhere(zfsvfs->z_os))
1836         if (!(zfsvfs->z_vfs->vfs_flag & VFS_RDONLY))

```



```

1837         txg_wait_synced(dmu_objset_pool(zfsvfs->z_os), 0);
1838     (void) dmu_objset_evict_dbufs(zfsvfs->z_os);

1840     return (0);
1841 }

1843 /*ARGSUSED*/
1844 static int
1845 zfs_umount(vfs_t *vfsp, int fflag, cred_t *cr)
1846 {
1847     zfsvfs_t *zfsvfs = vfsp->vfs_data;
1848     objset_t *os;
1849     int ret;

1851     ret = secpolicy_fs_unmount(cr, vfsp);
1852     if (ret) {
1853         if (dsl_deleg_access((char *)refstr_value(vfsp->vfs_resource),
1854             ZFS_DELEG_PERM_MOUNT, cr))
1855             return (ret);
1856     }

1858     /*
1859     * We purge the parent filesystem's vfsp as the parent filesystem
1860     * and all of its snapshots have their vnode's v_vfsp set to the
1861     * parent's filesystem's vfsp. Note, 'z_parent' is self
1862     * referential for non-snapshots.
1863     */
1864     (void) dnlc_purge_vfsp(zfsvfs->z_parent->z_vfs, 0);

1866     /*
1867     * Unmount any snapshots mounted under .zfs before unmounting the
1868     * dataset itself.
1869     */
1870     if (zfsvfs->z_ctldir != NULL &&
1871         (ret = zfsctl_umount_snapshots(vfsp, fflag, cr)) != 0) {
1872         return (ret);
1873     }

1875     if (!(fflag & MS_FORCE)) {
1876         /*
1877         * Check the number of active vnodes in the file system.
1878         * Our count is maintained in the vfs structure, but the
1879         * number is off by 1 to indicate a hold on the vfs
1880         * structure itself.
1881         *
1882         * The '.zfs' directory maintains a reference of its
1883         * own, and any active references underneath are
1884         * reflected in the vnode count.
1885         */
1886         if (zfsvfs->z_ctldir == NULL) {
1887             if (vfsp->vfs_count > 1)
1888                 return (EBUSY);
1889         } else {
1890             if (vfsp->vfs_count > 2 ||
1891                 zfsvfs->z_ctldir->v_count > 1)
1892                 return (EBUSY);
1893         }
1894     }

1896     vfsp->vfs_flag |= VFS_UNMOUNTED;

1898     VERIFY(zfsvfs_tearardown(zfsvfs, B_TRUE) == 0);
1899     os = zfsvfs->z_os;

1901     /*
1902     * z_os will be NULL if there was an error in

```

```

1903     * attempting to reopen zfsvfs.
1904     */
1905     if (os != NULL) {
1906         /*
1907         * Unset the objset user_ptr.
1908         */
1909         mutex_enter(&os->os_user_ptr_lock);
1910         dmu_objset_set_user(os, NULL);
1911         mutex_exit(&os->os_user_ptr_lock);

1913         /*
1914         * Finally release the objset
1915         */
1916         dmu_objset_disown(os, zfsvfs);
1917     }

1919     /*
1920     * We can now safely destroy the '.zfs' directory node.
1921     */
1922     if (zfsvfs->z_ctldir != NULL)
1923         zfsctl_destroy(zfsvfs);

1925     return (0);
1926 }

1928 static int
1929 zfs_vget(vfs_t *vfsp, vnode_t **vpp, fid_t *fidp)
1930 {
1931     zfsvfs_t      *zfsvfs = vfsp->vfs_data;
1932     znode_t       *zp;
1933     uint64_t      object = 0;
1934     uint64_t      fid_gen = 0;
1935     uint64_t      gen_mask;
1936     uint64_t      zp_gen;
1937     int           i, err;

1939     *vpp = NULL;

1941     ZFS_ENTER(zfsvfs);

1943     if (fidp->fid_len == LONG_FID_LEN) {
1944         zfid_long_t *zlfid = (zfid_long_t *)fidp;
1945         uint64_t      objsetid = 0;
1946         uint64_t      setgen = 0;

1948         for (i = 0; i < sizeof (zlfid->zfid_setid); i++)
1949             objsetid |= ((uint64_t)zlfid->zfid_setid[i]) << (8 * i);

1951         for (i = 0; i < sizeof (zlfid->zfid_setgen); i++)
1952             setgen |= ((uint64_t)zlfid->zfid_setgen[i]) << (8 * i);

1954         ZFS_EXIT(zfsvfs);

1956         err = zfsctl_lookup_objset(vfsp, objsetid, &zfsvfs);
1957         if (err)
1958             return (EINVAL);
1959         ZFS_ENTER(zfsvfs);
1960     }

1962     if (fidp->fid_len == SHORT_FID_LEN || fidp->fid_len == LONG_FID_LEN) {
1963         zfid_short_t *zfid = (zfid_short_t *)fidp;

1965         for (i = 0; i < sizeof (zfid->zfid_object); i++)
1966             object |= ((uint64_t)zfid->zfid_object[i]) << (8 * i);

1968         for (i = 0; i < sizeof (zfid->zfid_gen); i++)

```

```

1969         fid_gen |= ((uint64_t)zfid->zfid_gen[i]) << (8 * i);
1970     } else {
1971         ZFS_EXIT(zfsvfs);
1972         return (EINVAL);
1973     }
1974
1975     /* A zero fid_gen means we are in the .zfs control directories */
1976     if (fid_gen == 0 &&
1977         (object == ZFSCCTL_INO_ROOT || object == ZFSCCTL_INO_SNAPDIR)) {
1978         *vpp = zfsvfs->z_ctldir;
1979         ASSERT(*vpp != NULL);
1980         if (object == ZFSCCTL_INO_SNAPDIR) {
1981             VERIFY(zfsctl_root_lookup(*vpp, "snapshot", vpp, NULL,
1982                                     0, NULL, NULL, NULL, NULL) == 0);
1983         } else {
1984             VN_HOLD(*vpp);
1985         }
1986         ZFS_EXIT(zfsvfs);
1987         return (0);
1988     }
1989
1990     gen_mask = -1ULL >> (64 - 8 * i);
1991
1992     dprintf("getting %llu [%u mask %llx]\n", object, fid_gen, gen_mask);
1993     if (err = zfs_zget(zfsvfs, object, &zp)) {
1994         ZFS_EXIT(zfsvfs);
1995         return (err);
1996     }
1997     (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_GEN(zfsvfs), &zp_gen,
1998                   sizeof (uint64_t));
1999     zp_gen = zp_gen & gen_mask;
2000     if (zp_gen == 0)
2001         zp_gen = 1;
2002     if (zp->z_unlinked || zp_gen != fid_gen) {
2003         dprintf("znode gen (%u) != fid gen (%u)\n", zp_gen, fid_gen);
2004         VN_RELE(ZTOV(zp));
2005         ZFS_EXIT(zfsvfs);
2006         return (EINVAL);
2007     }
2008
2009     *vpp = ZTOV(zp);
2010     ZFS_EXIT(zfsvfs);
2011     return (0);
2012 }
2013
2014 /*
2015  * Block out VOPs and close zfsvfs_t::z_os
2016  *
2017  * Note, if successful, then we return with the 'z_teardown_lock' and
2018  * 'z_teardown_inactive_lock' write held.
2019  */
2020 int
2021 zfs_suspend_fs(zfsvfs_t *zfsvfs)
2022 {
2023     int error;
2024
2025     if ((error = zfsvfs_teardown(zfsvfs, B_FALSE)) != 0)
2026         return (error);
2027     dmub_objset_disown(zfsvfs->z_os, zfsvfs);
2028
2029     return (0);
2030 }
2031
2032 /*
2033  * Reopen zfsvfs_t::z_os and release VOPs.
2034  */

```

```

2035 int
2036 zfs_resume_fs(zfsvfs_t *zfsvfs, const char *osname)
2037 {
2038     int err;
2039
2040     ASSERT(RRW_WRITE_HELD(&zfsvfs->z_teardown_lock));
2041     ASSERT(RRW_WRITE_HELD(&zfsvfs->z_teardown_inactive_lock));
2042
2043     err = dmub_objset_own(osname, DMU_OST_ZFS, B_FALSE, zfsvfs,
2044                          &zfsvfs->z_os);
2045     if (err) {
2046         zfsvfs->z_os = NULL;
2047     } else {
2048         znode_t *zp;
2049         uint64_t sa_obj = 0;
2050
2051         /*
2052          * Make sure version hasn't changed
2053          */
2054
2055         err = zfs_get_zplprop(zfsvfs->z_os, ZFS_PROP_VERSION,
2056                              &zfsvfs->z_version);
2057
2058         if (err)
2059             goto bail;
2060
2061         err = zap_lookup(zfsvfs->z_os, MASTER_NODE_OBJ,
2062                        ZFS_SA_ATTRS, 8, 1, &sa_obj);
2063
2064         if (err && zfsvfs->z_version >= ZPL_VERSION_SA)
2065             goto bail;
2066
2067         if ((err = sa_setup(zfsvfs->z_os, sa_obj,
2068                            zfs_attr_table, ZPL_END, &zfsvfs->z_attr_table)) != 0)
2069             goto bail;
2070
2071         if (zfsvfs->z_version >= ZPL_VERSION_SA)
2072             sa_register_update_callback(zfsvfs->z_os,
2073                                       zfs_sa_upgrade);
2074
2075         VERIFY(zfsvfs_setup(zfsvfs, B_FALSE) == 0);
2076
2077         zfs_set_fuid_feature(zfsvfs);
2078
2079         /*
2080          * Attempt to re-establish all the active znodes with
2081          * their dbufs. If a zfs_rezget() fails, then we'll let
2082          * any potential callers discover that via ZFS_ENTER_VERIFY_VP
2083          * when they try to use their znode.
2084          */
2085         mutex_enter(&zfsvfs->z_znodes_lock);
2086         for (zp = list_head(&zfsvfs->z_all_znodes); zp;
2087              zp = list_next(&zfsvfs->z_all_znodes, zp)) {
2088             (void) zfs_rezget(zp);
2089         }
2090         mutex_exit(&zfsvfs->z_znodes_lock);
2091     }
2092
2093 bail:
2094     /* release the VOPs */
2095     rw_exit(&zfsvfs->z_teardown_inactive_lock);
2096     rrw_exit(&zfsvfs->z_teardown_lock, FTAG);
2097
2098     if (err) {
2099         /*
2100          * Since we couldn't reopen zfsvfs::z_os, or

```

```

2101         * setup the sa framework force unmount this file system.
2102         */
2103         if (vn_vfswlock(zfsvfs->z_vfs->vfs_vnodecovered) == 0)
2104             (void) dounmount(zfsvfs->z_vfs, MS_FORCE, CRED());
2105     }
2106     return (err);
2107 }

2109 static void
2110 zfs_freevfs(vfs_t *vfsp)
2111 {
2112     zfsvfs_t *zfsvfs = vfsp->vfs_data;

2114     /*
2115      * If this is a snapshot, we have an extra VFS_HOLD on our parent
2116      * from zfs_mount(). Release it here. If we came through
2117      * zfs_mountroot() instead, we didn't grab an extra hold, so
2118      * skip the VFS_RELE for rootvfs.
2119      */
2120     if (zfsvfs->z_issnap && (vfsp != rootvfs))
2121         VFS_RELE(zfsvfs->z_parent->z_vfs);

2123     zfsvfs_free(zfsvfs);

2125     atomic_add_32(&zfs_active_fs_count, -1);
2126 }

2128 /*
2129 * VFS_INIT() initialization. Note that there is no VFS_FINI(),
2130 * so we can't safely do any non-idempotent initialization here.
2131 * Leave that to zfs_init() and zfs_fini(), which are called
2132 * from the module's _init() and _fini() entry points.
2133 */
2134 /*ARGSUSED*/
2135 static int
2136 zfs_vfsinit(int fstype, char *name)
2137 {
2138     int error;

2140     zfsfstype = fstype;

2142     /*
2143      * Setup vfsops and vnodeops tables.
2144      */
2145     error = vfs_setfsops(fstype, zfs_vfsops_template, &zfs_vfsops);
2146     if (error != 0) {
2147         cmn_err(CE_WARN, "zfs: bad vfs ops template");
2148     }

2150     error = zfs_create_op_tables();
2151     if (error) {
2152         zfs_remove_op_tables();
2153         cmn_err(CE_WARN, "zfs: bad vnode ops template");
2154         (void) vfs_freevfsops_by_type(zfsfstype);
2155         return (error);
2156     }

2158     mutex_init(&zfs_dev_mtx, NULL, MUTEX_DEFAULT, NULL);

2160     /*
2161      * Unique major number for all zfs mounts.
2162      * If we run out of 32-bit minors, we'll getudev() another major.
2163      */
2164     zfs_major = ddi_name_to_major(ZFS_DRIVER);
2165     zfs_minor = ZFS_MIN_MINOR;

```

```

2167     return (0);
2168 }

2170 void
2171 zfs_init(void)
2172 {
2173     /*
2174      * Initialize .zfs directory structures
2175      */
2176     zfsctl_init();

2178     /*
2179      * Initialize znode cache, vnode ops, etc...
2180      */
2181     zfs_znode_init();

2183     dmu_objset_register_type(DMU_OST_ZFS, zfs_space_delta_cb);
2184 }

2186 void
2187 zfs_fini(void)
2188 {
2189     zfsctl_fini();
2190     zfs_znode_fini();
2191 }

2193 int
2194 zfs_busy(void)
2195 {
2196     return (zfs_active_fs_count != 0);
2197 }

2199 int
2200 zfs_set_version(zfsvfs_t *zfsvfs, uint64_t newvers)
2201 {
2202     int error;
2203     objset_t *os = zfsvfs->z_os;
2204     dmu_tx_t *tx;

2206     if (newvers < ZPL_VERSION_INITIAL || newvers > ZPL_VERSION)
2207         return (EINVAL);

2209     if (newvers < zfsvfs->z_version)
2210         return (EINVAL);

2212     if (zfs_spa_version_map(newvers) >
2213         spa_version(dmu_objset_spa(zfsvfs->z_os)))
2214         return (ENOTSUP);

2216     tx = dmu_tx_create(os);
2217     dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, B_FALSE, ZPL_VERSION_STR);
2218     if (newvers >= ZPL_VERSION_SA && !zfsvfs->z_use_sa) {
2219         dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, B_TRUE,
2220             ZFS_SA_ATTRS);
2221         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
2222     }
2223     error = dmu_tx_assign(tx, TXG_WAIT);
2224     if (error) {
2225         dmu_tx_abort(tx);
2226         return (error);
2227     }

2229     error = zap_update(os, MASTER_NODE_OBJ, ZPL_VERSION_STR,
2230         8, 1, &newvers, tx);

2232     if (error) {

```

```
2233         dmu_tx_commit(tx);
2234         return (error);
2235     }
2237     if (newvers >= ZPL_VERSION_SA && !zfsvfs->z_use_sa) {
2238         uint64_t sa_obj;
2240         ASSERT3U(spa_version(dmu_objset_spa(zfsvfs->z_os)), >=,
2241             SPA_VERSION_SA);
2242         sa_obj = zap_create(os, DMU_OT_SA_MASTER_NODE,
2243             DMU_OT_NONE, 0, tx);
2245         error = zap_add(os, MASTER_NODE_OBJ,
2246             ZFS_SA_ATTRS, 8, 1, &sa_obj, tx);
2247         ASSERT3U(error, ==, 0);
2249         VERIFY(0 == sa_set_sa_object(os, sa_obj));
2250         sa_register_update_callback(os, zfs_sa_upgrade);
2251     }
2253     spa_history_log_internal_ds(dmu_objset_ds(os), "upgrade", tx,
2254         "from %llu to %llu", zfsvfs->z_version, newvers);
2255     spa_history_log_internal(LOG_DS_UPGRADE,
2256         dmu_objset_spa(os), tx, "oldver=%llu newver=%llu dataset = %llu",
2257         zfsvfs->z_version, newvers, dmu_objset_id(os));
2259     dmu_tx_commit(tx);
2261     zfsvfs->z_version = newvers;
2263     zfs_set_fuid_feature(zfsvfs);
2265     return (0);
2266 }
2267 unchanged_portion_omitted
```

new/usr/src/uts/common/fs/zfs/zvol.c

1

```
*****
49100 Thu Jun 28 15:57:22 2012
new/usr/src/uts/common/fs/zfs/zvol.c
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 *
24 * Portions Copyright 2010 Robert Milkowski
25 *
26 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
27 * Copyright (c) 2012 by Delphix. All rights reserved.
28 #endif /* !codereview */
29 */
31 /*
32  * ZFS volume emulation driver.
33  *
34  * Makes a DMU object look like a volume of arbitrary size, up to 2^64 bytes.
35  * Volumes are accessed through the symbolic links named:
36  *
37  * /dev/zvol/dsk/<pool_name>/<dataset_name>
38  * /dev/zvol/rdsk/<pool_name>/<dataset_name>
39  *
40  * These links are created by the /dev filesystem (sdev_zvolops.c).
41  * Volumes are persistent through reboot. No user command needs to be
42  * run before opening and using a device.
43  */
45 #include <sys/types.h>
46 #include <sys/param.h>
47 #include <sys/errno.h>
48 #include <sys/uio.h>
49 #include <sys/buf.h>
50 #include <sys/modctl.h>
51 #include <sys/open.h>
52 #include <sys/kmem.h>
53 #include <sys/conf.h>
54 #include <sys/cmn_err.h>
```

new/usr/src/uts/common/fs/zfs/zvol.c

2

```
55 #include <sys/stat.h>
56 #include <sys/zap.h>
57 #include <sys/spa.h>
58 #include <sys/zio.h>
59 #include <sys/dmu_traverse.h>
60 #include <sys/dnode.h>
61 #include <sys/dsl_dataset.h>
62 #include <sys/dsl_prop.h>
63 #include <sys/dkio.h>
64 #include <sys/efi_partition.h>
65 #include <sys/byteorder.h>
66 #include <sys/pathname.h>
67 #include <sys/ddi.h>
68 #include <sys/sunddi.h>
69 #include <sys/crc32.h>
70 #include <sys/dirent.h>
71 #include <sys/policy.h>
72 #include <sys/fs/zfs.h>
73 #include <sys/zfs_ioctl.h>
74 #include <sys/mkdev.h>
75 #include <sys/zil.h>
76 #include <sys/refcount.h>
77 #include <sys/zfs_znode.h>
78 #include <sys/zfs_rlock.h>
79 #include <sys/vdev_disk.h>
80 #include <sys/vdev_impl.h>
81 #include <sys/zvol.h>
82 #include <sys/dumphdr.h>
83 #include <sys/zil_impl.h>
85 #include "zfs_namecheck.h"
87 void *zfsdev_state;
88 static char *zvol_tag = "zvol_tag";
90 #define ZVOL_DUMP_SIZE "dumpsize"
92 /*
93  * This lock protects the zfsdev_state structure from being modified
94  * while it's being used, e.g. an open that comes in before a create
95  * finishes. It also protects temporary opens of the dataset so that,
96  * e.g., an open doesn't get a spurious EBUSY.
97  */
98 kmutex_t zfsdev_state_lock;
99 static uint32_t zvol_minors;
101 typedef struct zvol_extent {
102     list_node_t    ze_node;
103     dva_t          ze_dva;           /* dva associated with this extent */
104     uint64_t       ze_nblks;       /* number of blocks in extent */
105 } zvol_extent_t;
107 /*
108  * The in-core state of each volume.
109  */
110 typedef struct zvol_state {
111     char           zv_name[MAXPATHLEN]; /* pool/dd name */
112     uint64_t       zv_volsize;         /* amount of space we advertise */
113     uint64_t       zv_volblocksize;   /* volume block size */
114     minor_t        zv_minor;          /* minor number */
115     uint8_t        zv_min_bs;         /* minimum addressable block shift */
116     uint8_t        zv_flags;          /* readonly, dumpified, etc. */
117     objset_t       *zv_objset;        /* objset handle */
118     zvol_open_count_t zv_open_count; /* open counts */
119     uint32_t       zv_total_opens;    /* total open count */
120     zillog_t       *zv_zilog;         /* ZIL handle */
```

```

121     list_t      zv_extents; /* List of extents for dump */
122     znode_t     zv_znode;   /* for range locking */
123     dmu_buf_t   *zv_dbuf;   /* bonus handle */
124 } zvol_state_t;

126 /*
127  * zvol specific flags
128  */
129 #define ZVOL_RDONLY    0x1
130 #define ZVOL_DUMPIFIED 0x2
131 #define ZVOL_EXCL     0x4
132 #define ZVOL_WCE      0x8

134 /*
135  * zvol maximum transfer in one DMU tx.
136  */
137 int zvol_maxphys = DMU_MAX_ACCESS/2;

139 extern int zfs_set_prop_nvlist(const char *, zprop_source_t,
140     nvlist_t *, nvlist_t *);
141     nvlist_t *, nvlist_t **);
142 static int zvol_remove_zv(zvol_state_t *);
143 static int zvol_get_data(void *arg, lr_write_t *lr, char *buf, zio_t *zio);
144 static int zvol_dumpify(zvol_state_t *zv);
145 static int zvol_dump_fini(zvol_state_t *zv);
146 static int zvol_dump_init(zvol_state_t *zv, boolean_t resize);

147 static void
148 zvol_size_changed(uint64_t volsize, major_t maj, minor_t min)
149 {
150     dev_t dev = makedevice(maj, min);

152     VERIFY(ddi_prop_update_int64(dev, zfs_dip,
153         "Size", volsize) == DDI_SUCCESS);
154     VERIFY(ddi_prop_update_int64(dev, zfs_dip,
155         "Nblocks", lbtodb(volsize)) == DDI_SUCCESS);

157     /* Notify specfs to invalidate the cached size */
158     spec_size_invalidate(dev, VBLK);
159     spec_size_invalidate(dev, VCHR);
160 }
    unchanged portion omitted

1877 static int
1878 zvol_dumpify(zvol_state_t *zv)
1879 {
1880     int error = 0;
1881     uint64_t dumpsize = 0;
1882     dmu_tx_t *tx;
1883     objset_t *os = zv->zv_objset;

1885     if (zv->zv_flags & ZVOL_RDONLY)
1886         return (EROFS);

1888     if (zap_lookup(zv->zv_objset, ZVOL_ZAP_OBJ, ZVOL_DUMP_SIZE,
1889         8, 1, &dumpsize) != 0 || dumpsize != zv->zv_volsize) {
1890         boolean_t resize = (dumpsize > 0);
1777         boolean_t resize = (dumpsize > 0) ? B_TRUE : B_FALSE;

1892         if ((error = zvol_dump_init(zv, resize)) != 0) {
1893             (void) zvol_dump_fini(zv);
1894             return (error);
1895         }
1896     }

1898     /*

```

```

1899     * Build up our lba mapping.
1900     */
1901     error = zvol_get_lbas(zv);
1902     if (error) {
1903         (void) zvol_dump_fini(zv);
1904         return (error);
1905     }

1907     tx = dmu_tx_create(os);
1908     dmu_tx_hold_zap(tx, ZVOL_ZAP_OBJ, TRUE, NULL);
1909     error = dmu_tx_assign(tx, TXG_WAIT);
1910     if (error) {
1911         dmu_tx_abort(tx);
1912         (void) zvol_dump_fini(zv);
1913         return (error);
1914     }

1916     zv->zv_flags |= ZVOL_DUMPIFIED;
1917     error = zap_update(os, ZVOL_ZAP_OBJ, ZVOL_DUMP_SIZE, 8, 1,
1918         &zv->zv_volsize, tx);
1919     dmu_tx_commit(tx);

1921     if (error) {
1922         (void) zvol_dump_fini(zv);
1923         return (error);
1924     }

1926     txg_wait_synced(dmu_objset_pool(os), 0);
1927     return (0);
1928 }
    unchanged portion omitted

```

new/usr/src/uts/common/sys/fs/zfs.h

1

```
*****
28730 Thu Jun 28 15:57:22 2012
new/usr/src/uts/common/sys/fs/zfs.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
unchanged_portion_omitted

55 typedef enum dmu_objset_type {
56     DMU_OST_NONE,
57     DMU_OST_META,
58     DMU_OST_ZFS,
59     DMU_OST_ZVOL,
60     DMU_OST_OTHER,          /* For testing only! */
61     DMU_OST_ANY,           /* Be careful! */
62     DMU_OST_NUMTYPES
63 } dmu_objset_type_t;

64 #endif /* ! codereview */
65 #define ZFS_TYPE_DATASET \
66     (ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME | ZFS_TYPE_SNAPSHOT)

67

68 #define ZAP_MAXNAMELEN 256
69 #define ZAP_MAXVALUELEN (1024 * 8)
70 #define ZAP_OLDMAXVALUELEN 1024
71

72 /*
73  * Dataset properties are identified by these constants and must be added to
74  * the end of this list to ensure that external consumers are not affected
75  * by the change. If you make any changes to this list, be sure to update
76  * the property table in usr/src/common/zfs/zfs_prop.c.
77  */
78
79 typedef enum {
80     ZFS_PROP_TYPE,
81     ZFS_PROP_CREATION,
82     ZFS_PROP_USED,
83     ZFS_PROP_AVAILABLE,
84     ZFS_PROP_REFERENCED,
85     ZFS_PROP_COMPRESSRATIO,
86     ZFS_PROP_MOUNTED,
87     ZFS_PROP_ORIGIN,
88     ZFS_PROP_QUOTA,
89     ZFS_PROP_RESERVATION,
90     ZFS_PROP_VOLSIZE,
91     ZFS_PROP_VOLBLOCKSIZE,
92     ZFS_PROP_RECORDSIZE,
93     ZFS_PROP_MOUNTPOINT,
94     ZFS_PROP_SHARENFS,
95     ZFS_PROP_CHECKSUM,
96     ZFS_PROP_COMPRESSION,
97     ZFS_PROP_ETIME,
98     ZFS_PROP_DEVICES,
99     ZFS_PROP_EXEC,
100    ZFS_PROP_SETUID,
101    ZFS_PROP_READONLY,
102    ZFS_PROP_ZONED,
103    ZFS_PROP_SNAPDIR,
104    ZFS_PROP_ACLMODE,
105    ZFS_PROP_ACLINHERIT,
106    ZFS_PROP_CREATETXG,          /* not exposed to the user */

```

new/usr/src/uts/common/sys/fs/zfs.h

2

```
107     ZFS_PROP_NAME,             /* not exposed to the user */
108     ZFS_PROP_CANMOUNT,
109     ZFS_PROP_ISCSIPTIONS,     /* not exposed to the user */
110     ZFS_PROP_XATTR,
111     ZFS_PROP_NUMCLONES,       /* not exposed to the user */
112     ZFS_PROP_COPIES,
113     ZFS_PROP_VERSION,
114     ZFS_PROP_UTF8ONLY,
115     ZFS_PROP_NORMALIZE,
116     ZFS_PROP_CASE,
117     ZFS_PROP_VSCAN,
118     ZFS_PROP_NBMAND,
119     ZFS_PROP_SHARESMB,
120     ZFS_PROP_REFQUOTA,
121     ZFS_PROP_REFRESERVATION,
122     ZFS_PROP_GUID,
123     ZFS_PROP_PRIMARYCACHE,
124     ZFS_PROP_SECONDARYCACHE,
125     ZFS_PROP_USEDSNAP,
126     ZFS_PROP_USEDSDS,
127     ZFS_PROP_USEDCHILD,
128     ZFS_PROP_USEDREFRESERV,
129     ZFS_PROP_USERACCOUNTING,  /* not exposed to the user */
130     ZFS_PROP_STMF_SHAREINFO,  /* not exposed to the user */
131     ZFS_PROP_DEFER_DESTROY,
132     ZFS_PROP_USERREFS,
133     ZFS_PROP_LOGBIAS,
134     ZFS_PROP_UNIQUE,          /* not exposed to the user */
135     ZFS_PROP_OBJSETID,       /* not exposed to the user */
136     ZFS_PROP_DEDUP,
137     ZFS_PROP_MLSLABEL,
138     ZFS_PROP_SYNC,
139     ZFS_PROP_REFRATIO,
140     ZFS_PROP_WRITTEN,
141     ZFS_PROP_CLONES,
142     ZFS_NUM_PROPS
143 } zfs_prop_t;

144
145 typedef enum {
146     ZFS_PROP_USERUSED,
147     ZFS_PROP_USERQUOTA,
148     ZFS_PROP_GROUPUSED,
149     ZFS_PROP_GROUPQUOTA,
150     ZFS_NUM_USERQUOTA_PROPS
151 } zfs_userquota_prop_t;

152
153 extern const char *zfs_userquota_prop_prefixes[ZFS_NUM_USERQUOTA_PROPS];

154
155 /*
156  * Pool properties are identified by these constants and must be added to the
157  * end of this list to ensure that external consumers are not affected
158  * by the change. If you make any changes to this list, be sure to update
159  * the property table in usr/src/common/zfs/zpool_prop.c.
160  */
161 typedef enum {
162     ZPOOL_PROP_NAME,
163     ZPOOL_PROP_SIZE,
164     ZPOOL_PROP_CAPACITY,
165     ZPOOL_PROP_ALTROOT,
166     ZPOOL_PROP_HEALTH,
167     ZPOOL_PROP_GUID,
168     ZPOOL_PROP_VERSION,
169     ZPOOL_PROP_BOOTFS,
170     ZPOOL_PROP_DELEGATION,
171     ZPOOL_PROP_AUTOREPLACE,
172     ZPOOL_PROP_CACHEFILE,

```

```

173     ZPOOL_PROP_FAILUREMODE,
174     ZPOOL_PROP_LISTSNAPS,
175     ZPOOL_PROP_AUTOEXPAND,
176     ZPOOL_PROP_DEDUPDITTO,
177     ZPOOL_PROP_DEDUPRATIO,
178     ZPOOL_PROP_FREE,
179     ZPOOL_PROP_ALLOCATED,
180     ZPOOL_PROP_READONLY,
181     ZPOOL_PROP_COMMENT,
182     ZPOOL_PROP_EXPANDSZ,
183     ZPOOL_PROP_FREEING,
184     ZPOOL_NUM_PROPS
185 } zpool_prop_t;

187 /* Small enough to not hog a whole line of printout in zpool(1M). */
188 #define ZPROP_MAX_COMMENT      32

190 #define ZPROP_CONT             -2
191 #define ZPROP_INVAL           -1

193 #define ZPROP_VALUE            "value"
194 #define ZPROP_SOURCE           "source"

196 typedef enum {
197     ZPROP_SRC_NONE = 0x1,
198     ZPROP_SRC_DEFAULT = 0x2,
199     ZPROP_SRC_TEMPORARY = 0x4,
200     ZPROP_SRC_LOCAL = 0x8,
201     ZPROP_SRC_INHERITED = 0x10,
202     ZPROP_SRC_RECEIVED = 0x20
203 } zprop_source_t;

205 #define ZPROP_SRC_ALL      0x3f

207 #define ZPROP_SOURCE_VAL_RECVD  "$recvd"
208 #define ZPROP_N_MORE_ERRORS    "N_MORE_ERRORS"
209 /*
210  * Dataset flag implemented as a special entry in the props zap object
211  * indicating that the dataset has received properties on or after
212  * SPA_VERSION_RECVD_PROPS. The first such receive blows away local properties
213  * just as it did in earlier versions, and thereafter, local properties are
214  * preserved.
215  */
216 #define ZPROP_HAS_RECVD        "$hasrecvd"

218 typedef enum {
219     ZPROP_ERR_NOCLEAR = 0x1, /* failure to clear existing props */
220     ZPROP_ERR_NORESTORE = 0x2 /* failure to restore props on error */
221 } zprop_errflags_t;

223 typedef int (*zprop_func)(int, void *);

225 /*
226  * Properties to be set on the root file system of a new pool
227  * are stuffed into their own nvlist, which is then included in
228  * the properties nvlist with the pool properties.
229  */
230 #define ZPOOL_ROOTFS_PROPS      "root-props-nv1"

232 /*
233  * Dataset property functions shared between libzfs and kernel.
234  */
235 const char *zfs_prop_default_string(zfs_prop_t);
236 uint64_t zfs_prop_default_numeric(zfs_prop_t);
237 boolean_t zfs_prop_readonly(zfs_prop_t);
238 boolean_t zfs_prop_inheritable(zfs_prop_t);

```

```

239 boolean_t zfs_prop_setonce(zfs_prop_t);
240 const char *zfs_prop_to_name(zfs_prop_t);
241 zfs_prop_t zfs_name_to_prop(const char *);
242 boolean_t zfs_prop_user(const char *);
243 boolean_t zfs_prop_userquota(const char *);
244 boolean_t zfs_prop_written(const char *);
245 int zfs_prop_index_to_string(zfs_prop_t, uint64_t, const char **);
246 int zfs_prop_string_to_index(zfs_prop_t, const char *, uint64_t *);
247 uint64_t zfs_prop_random_value(zfs_prop_t, uint64_t seed);
248 boolean_t zfs_prop_valid_for_type(int, zfs_type_t);

250 /*
251  * Pool property functions shared between libzfs and kernel.
252  */
253 zpool_prop_t zpool_name_to_prop(const char *);
254 const char *zpool_prop_to_name(zpool_prop_t);
255 const char *zpool_prop_default_string(zpool_prop_t);
256 uint64_t zpool_prop_default_numeric(zpool_prop_t);
257 boolean_t zpool_prop_readonly(zpool_prop_t);
258 boolean_t zpool_prop_feature(const char *);
259 boolean_t zpool_prop_unsupported(const char *name);
260 int zpool_prop_index_to_string(zpool_prop_t, uint64_t, const char **);
261 int zpool_prop_string_to_index(zpool_prop_t, const char *, uint64_t *);
262 uint64_t zpool_prop_random_value(zpool_prop_t, uint64_t seed);

264 /*
265  * Definitions for the Delegation.
266  */
267 typedef enum {
268     ZFS_DELEG_WHO_UNKNOWN = 0,
269     ZFS_DELEG_USER = 'u',
270     ZFS_DELEG_USER_SETS = 'U',
271     ZFS_DELEG_GROUP = 'g',
272     ZFS_DELEG_GROUP_SETS = 'G',
273     ZFS_DELEG_EVERYONE = 'e',
274     ZFS_DELEG_EVERYONE_SETS = 'E',
275     ZFS_DELEG_CREATE = 'c',
276     ZFS_DELEG_CREATE_SETS = 'C',
277     ZFS_DELEG_NAMED_SET = 's',
278     ZFS_DELEG_NAMED_SET_SETS = 'S'
279 } zfs_deleg_who_type_t;

281 typedef enum {
282     ZFS_DELEG_NONE = 0,
283     ZFS_DELEG_PERM_LOCAL = 1,
284     ZFS_DELEG_PERM_DESCENDENT = 2,
285     ZFS_DELEG_PERM_LOCALDESCENDENT = 3,
286     ZFS_DELEG_PERM_CREATE = 4
287 } zfs_deleg_inherit_t;

289 #define ZFS_DELEG_PERM_UID      "uid"
290 #define ZFS_DELEG_PERM_GID      "gid"
291 #define ZFS_DELEG_PERM_GROUPS   "groups"

293 #define ZFS_MLSLABEL_DEFAULT    "none"

295 #define ZFS_SMB_ACL_SRC         "src"
296 #define ZFS_SMB_ACL_TARGET     "target"

298 typedef enum {
299     ZFS_CANMOUNT_OFF = 0,
300     ZFS_CANMOUNT_ON = 1,
301     ZFS_CANMOUNT_NOAUTO = 2
302 } zfs_canmount_type_t;

304 typedef enum {

```



```

305     ZFS_LOGBIAS_LATENCY = 0,
306     ZFS_LOGBIAS_THROUGHPUT = 1
307 } zfs_logbias_op_t;

309 typedef enum zfs_share_op {
310     ZFS_SHARE_NFS = 0,
311     ZFS_UNSHARE_NFS = 1,
312     ZFS_SHARE_SMB = 2,
313     ZFS_UNSHARE_SMB = 3
314 } zfs_share_op_t;

316 typedef enum zfs_smb_acl_op {
317     ZFS_SMB_ACL_ADD,
318     ZFS_SMB_ACL_REMOVE,
319     ZFS_SMB_ACL_RENAME,
320     ZFS_SMB_ACL_PURGE
321 } zfs_smb_acl_op_t;

323 typedef enum zfs_cache_type {
324     ZFS_CACHE_NONE = 0,
325     ZFS_CACHE_METADATA = 1,
326     ZFS_CACHE_ALL = 2
327 } zfs_cache_type_t;

329 typedef enum {
330     ZFS_SYNC_STANDARD = 0,
331     ZFS_SYNC_ALWAYS = 1,
332     ZFS_SYNC_DISABLED = 2
333 } zfs_sync_type_t;

336 /*
337  * On-disk version number.
338  */
339 #define SPA_VERSION_1           1ULL
340 #define SPA_VERSION_2           2ULL
341 #define SPA_VERSION_3           3ULL
342 #define SPA_VERSION_4           4ULL
343 #define SPA_VERSION_5           5ULL
344 #define SPA_VERSION_6           6ULL
345 #define SPA_VERSION_7           7ULL
346 #define SPA_VERSION_8           8ULL
347 #define SPA_VERSION_9           9ULL
348 #define SPA_VERSION_10          10ULL
349 #define SPA_VERSION_11          11ULL
350 #define SPA_VERSION_12          12ULL
351 #define SPA_VERSION_13          13ULL
352 #define SPA_VERSION_14          14ULL
353 #define SPA_VERSION_15          15ULL
354 #define SPA_VERSION_16          16ULL
355 #define SPA_VERSION_17          17ULL
356 #define SPA_VERSION_18          18ULL
357 #define SPA_VERSION_19          19ULL
358 #define SPA_VERSION_20          20ULL
359 #define SPA_VERSION_21          21ULL
360 #define SPA_VERSION_22          22ULL
361 #define SPA_VERSION_23          23ULL
362 #define SPA_VERSION_24          24ULL
363 #define SPA_VERSION_25          25ULL
364 #define SPA_VERSION_26          26ULL
365 #define SPA_VERSION_27          27ULL
366 #define SPA_VERSION_28          28ULL
367 #define SPA_VERSION_5000        5000ULL

369 /*
370  * When bumping up SPA_VERSION, make sure GRUB ZFS understands the on-disk

```

```

371  * format change. Go to usr/src/grub/grub-0.97/stage2/{zfs-include/, fsys_zfs*},
372  * and do the appropriate changes. Also bump the version number in
373  * usr/src/grub/capability.
374  */
375 #define SPA_VERSION              SPA_VERSION_5000
376 #define SPA_VERSION_STRING       "5000"

378 /*
379  * Symbolic names for the changes that caused a SPA_VERSION switch.
380  * Used in the code when checking for presence or absence of a feature.
381  * Feel free to define multiple symbolic names for each version if there
382  * were multiple changes to on-disk structures during that version.
383  *
384  * NOTE: When checking the current SPA_VERSION in your code, be sure
385  * to use spa_version() since it reports the version of the
386  * last synced uberblock. Checking the in-flight version can
387  * be dangerous in some cases.
388  */
389 #define SPA_VERSION_INITIAL      SPA_VERSION_1
390 #define SPA_VERSION_DITTO_BLOCKS SPA_VERSION_2
391 #define SPA_VERSION_SPARES      SPA_VERSION_3
392 #define SPA_VERSION_RAIDZ2     SPA_VERSION_3
393 #define SPA_VERSION_BPOBJ_ACCOUNT SPA_VERSION_3
394 #define SPA_VERSION_RAIDZ_DEFLATE SPA_VERSION_3
395 #define SPA_VERSION_DNODE_BYTES SPA_VERSION_3
396 #define SPA_VERSION_ZPOOL_HISTORY SPA_VERSION_4
397 #define SPA_VERSION_GZIP_COMPRESSION SPA_VERSION_5
398 #define SPA_VERSION_BOOTFS     SPA_VERSION_6
399 #define SPA_VERSION_SLOGS      SPA_VERSION_7
400 #define SPA_VERSION_DELEGATED_PERMS SPA_VERSION_8
401 #define SPA_VERSION_FUID       SPA_VERSION_9
402 #define SPA_VERSION_REFRESERVATION SPA_VERSION_9
403 #define SPA_VERSION_REFQUOTA   SPA_VERSION_9
404 #define SPA_VERSION_UNIQUE_ACCURATE SPA_VERSION_9
405 #define SPA_VERSION_L2CACHE     SPA_VERSION_10
406 #define SPA_VERSION_NEXT_CLONES SPA_VERSION_11
407 #define SPA_VERSION_ORIGIN     SPA_VERSION_11
408 #define SPA_VERSION_DSL_SCRUB  SPA_VERSION_11
409 #define SPA_VERSION_SNAP_PROPS SPA_VERSION_12
410 #define SPA_VERSION_USED_BREAKDOWN SPA_VERSION_13
411 #define SPA_VERSION_PASSTHROUGH_X SPA_VERSION_14
412 #define SPA_VERSION_USERSPACE  SPA_VERSION_15
413 #define SPA_VERSION_STMF_PROP  SPA_VERSION_16
414 #define SPA_VERSION_RAIDZ3     SPA_VERSION_17
415 #define SPA_VERSION_USERREFS   SPA_VERSION_18
416 #define SPA_VERSION_HOLES      SPA_VERSION_19
417 #define SPA_VERSION_ZLE_COMPRESSION SPA_VERSION_20
418 #define SPA_VERSION_DEDUP      SPA_VERSION_21
419 #define SPA_VERSION_RECVD_PROPS SPA_VERSION_22
420 #define SPA_VERSION_SLIM_ZIL   SPA_VERSION_23
421 #define SPA_VERSION_SA         SPA_VERSION_24
422 #define SPA_VERSION_SCAN       SPA_VERSION_25
423 #define SPA_VERSION_DIR_CLONES SPA_VERSION_26
424 #define SPA_VERSION_DEADLISTS  SPA_VERSION_26
425 #define SPA_VERSION_FAST_SNAP  SPA_VERSION_27
426 #define SPA_VERSION_MULTI_REPLACE SPA_VERSION_28
427 #define SPA_VERSION_BEFORE_FEATURES SPA_VERSION_28
428 #define SPA_VERSION_FEATURES   SPA_VERSION_5000

430 #define SPA_VERSION_IS_SUPPORTED(v) \
431     (((v) >= SPA_VERSION_INITIAL && (v) <= SPA_VERSION_BEFORE_FEATURES) || \
432      ((v) >= SPA_VERSION_FEATURES && (v) <= SPA_VERSION))

434 /*
435  * ZPL version - rev'd whenever an incompatible on-disk format change
436  * occurs. This is independent of SPA/DMU/ZAP versioning. You must

```

```

437 * also update the version_table[] and help message in zfs_prop.c.
438 *
439 * When changing, be sure to teach GRUB how to read the new format!
440 * See usr/src/grub/grub-0.97/stage2/{zfs-include/,fsys_zfs*}
441 */
442 #define ZPL_VERSION_1          1ULL
443 #define ZPL_VERSION_2          2ULL
444 #define ZPL_VERSION_3          3ULL
445 #define ZPL_VERSION_4          4ULL
446 #define ZPL_VERSION_5          5ULL
447 #define ZPL_VERSION            ZPL_VERSION_5
448 #define ZPL_VERSION_STRING     "5"

450 #define ZPL_VERSION_INITIAL     ZPL_VERSION_1
451 #define ZPL_VERSION_DIRENT_TYPE ZPL_VERSION_2
452 #define ZPL_VERSION_FUID       ZPL_VERSION_3
453 #define ZPL_VERSION_NORMALIZATION ZPL_VERSION_3
454 #define ZPL_VERSION_SYSATTR    ZPL_VERSION_3
455 #define ZPL_VERSION_USERSPACE  ZPL_VERSION_4
456 #define ZPL_VERSION_SA         ZPL_VERSION_5

458 /* Rewind request information */
459 #define ZPOOL_NO_REWIND        1 /* No policy - default behavior */
460 #define ZPOOL_NEVER_REWIND    2 /* Do not search for best txg or rewind */
461 #define ZPOOL_TRY_REWIND      4 /* Search for best txg, but do not rewind */
462 #define ZPOOL_DO_REWIND       8 /* Rewind to best txg w/in deferred frees */
463 #define ZPOOL_EXTREME_REWIND 16 /* Allow extreme measures to find best txg */
464 #define ZPOOL_REWIND_MASK     28 /* All the possible rewind bits */
465 #define ZPOOL_REWIND_POLICIES 31 /* All the possible policy bits */

467 typedef struct zpool_rewind_policy {
468     uint32_t    zrp_request; /* rewind behavior requested */
469     uint64_t    zrp_maxmeta; /* max acceptable meta-data errors */
470     uint64_t    zrp_maxdata; /* max acceptable data errors */
471     uint64_t    zrp_txg;     /* specific txg to load */
472 } zpool_rewind_policy_t;

474 /*
475 * The following are configuration names used in the nvlist describing a pool's
476 * configuration.
477 */
478 #define ZPOOL_CONFIG_VERSION     "version"
479 #define ZPOOL_CONFIG_POOL_NAME   "name"
480 #define ZPOOL_CONFIG_POOL_STATE  "state"
481 #define ZPOOL_CONFIG_POOL_TXG    "txg"
482 #define ZPOOL_CONFIG_POOL_GUID   "pool_guid"
483 #define ZPOOL_CONFIG_CREATE_TXG  "create_txg"
484 #define ZPOOL_CONFIG_TOP_GUID    "top_guid"
485 #define ZPOOL_CONFIG_VDEV_TREE   "vdev_tree"
486 #define ZPOOL_CONFIG_TYPE        "type"
487 #define ZPOOL_CONFIG_CHILDREN    "children"
488 #define ZPOOL_CONFIG_ID          "id"
489 #define ZPOOL_CONFIG_GUID        "guid"
490 #define ZPOOL_CONFIG_PATH        "path"
491 #define ZPOOL_CONFIG_DEVID       "devid"
492 #define ZPOOL_CONFIG_METASLAB_ARRAY "metaslab_array"
493 #define ZPOOL_CONFIG_METASLAB_SHIFT "metaslab_shift"
494 #define ZPOOL_CONFIG_ASHIFT      "ashift"
495 #define ZPOOL_CONFIG_ASIZE       "asize"
496 #define ZPOOL_CONFIG_DTL         "DTL"
497 #define ZPOOL_CONFIG_SCAN_STATS  "scan_stats" /* not stored on disk */
498 #define ZPOOL_CONFIG_VDEV_STATS  "vdev_stats" /* not stored on disk */
499 #define ZPOOL_CONFIG_WHOLE_DISK  "whole_disk"
500 #define ZPOOL_CONFIG_ERRCOUNT   "error_count"
501 #define ZPOOL_CONFIG_NOT_PRESENT "not_present"
502 #define ZPOOL_CONFIG_SPARES      "spares"

```

```

503 #define ZPOOL_CONFIG_IS_SPARE    "is_spare"
504 #define ZPOOL_CONFIG_NPARITY    "nparity"
505 #define ZPOOL_CONFIG_HOSTID     "hostid"
506 #define ZPOOL_CONFIG_HOSTNAME   "hostname"
507 #define ZPOOL_CONFIG_LOADED_TIME "initial_load_time"
508 #define ZPOOL_CONFIG_UNSPARE    "unspare"
509 #define ZPOOL_CONFIG_PHYS_PATH  "phys_path"
510 #define ZPOOL_CONFIG_IS_LOG     "is_log"
511 #define ZPOOL_CONFIG_L2CACHE    "l2cache"
512 #define ZPOOL_CONFIG_HOLE_ARRAY "hole_array"
513 #define ZPOOL_CONFIG_VDEV_CHILDREN "vdev_children"
514 #define ZPOOL_CONFIG_IS_HOLE    "is_hole"
515 #define ZPOOL_CONFIG_DDT_HISTOGRAM "ddt_histogram"
516 #define ZPOOL_CONFIG_DDT_OBJ_STATS "ddt_object_stats"
517 #define ZPOOL_CONFIG_DDT_STATS  "ddt_stats"
518 #define ZPOOL_CONFIG_SPLIT      "splitcfg"
519 #define ZPOOL_CONFIG_ORIG_GUID  "orig_guid"
520 #define ZPOOL_CONFIG_SPLIT_GUID "split_guid"
521 #define ZPOOL_CONFIG_SPLIT_LIST "guid_list"
522 #define ZPOOL_CONFIG_REMOVING   "removing"
523 #define ZPOOL_CONFIG_RESILVERING "resilvering"
524 #define ZPOOL_CONFIG_COMMENT    "comment"
525 #define ZPOOL_CONFIG_SUSPENDED  "suspended" /* not stored on disk */
526 #define ZPOOL_CONFIG_TIMESTAMP  "timestamp" /* not stored on disk */
527 #define ZPOOL_CONFIG_BOOTFS     "bootfs" /* not stored on disk */
528 #define ZPOOL_CONFIG_MISSING_DEVICES "missing_vdevs" /* not stored on disk */
529 #define ZPOOL_CONFIG_LOAD_INFO  "load_info" /* not stored on disk */
530 #define ZPOOL_CONFIG_REWIND_INFO "rewind_info" /* not stored on disk */
531 #define ZPOOL_CONFIG_UNSUP_FEAT "unsup_feat" /* not stored on disk */
532 #define ZPOOL_CONFIG_CAN_RDONLY "can_RDONLY" /* not stored on disk */
533 #define ZPOOL_CONFIG_FEATURES_FOR_READ "features_for_read"
534 #define ZPOOL_CONFIG_FEATURE_STATS "feature_stats" /* not stored on disk */
535 /*
536 * The persistent vdev state is stored as separate values rather than a single
537 * 'vdev_state' entry. This is because a device can be in multiple states, such
538 * as offline and degraded.
539 */
540 #define ZPOOL_CONFIG_OFFLINE     "offline"
541 #define ZPOOL_CONFIG_FAULTED    "faulted"
542 #define ZPOOL_CONFIG_DEGRADED   "degraded"
543 #define ZPOOL_CONFIG_REMOVED    "removed"
544 #define ZPOOL_CONFIG_FRU        "fru"
545 #define ZPOOL_CONFIG_AUX_STATE  "aux_state"

547 /* Rewind policy parameters */
548 #define ZPOOL_REWIND_POLICY      "rewind-policy"
549 #define ZPOOL_REWIND_REQUEST    "rewind-request"
550 #define ZPOOL_REWIND_REQUEST_TXG "rewind-request-txg"
551 #define ZPOOL_REWIND_META_THRESH "rewind-meta-thresh"
552 #define ZPOOL_REWIND_DATA_THRESH "rewind-data-thresh"

554 /* Rewind data discovered */
555 #define ZPOOL_CONFIG_LOAD_TIME   "rewind_txg_ts"
556 #define ZPOOL_CONFIG_LOAD_DATA_ERRORS "verify_data_errors"
557 #define ZPOOL_CONFIG_REWIND_TIME "seconds_of_rewind"

559 #define VDEV_TYPE_ROOT           "root"
560 #define VDEV_TYPE_MIRROR        "mirror"
561 #define VDEV_TYPE REPLACING     "replacing"
562 #define VDEV_TYPE RAIDZ        "raidz"
563 #define VDEV_TYPE DISK         "disk"
564 #define VDEV_TYPE FILE         "file"
565 #define VDEV_TYPE MISSING      "missing"
566 #define VDEV_TYPE HOLE         "hole"
567 #define VDEV_TYPE SPARE        "spare"
568 #define VDEV_TYPE LOG          "log"

```

```

569 #define VDEV_TYPE_L2CACHE          "l2cache"

571 /*
572  * This is needed in userland to report the minimum necessary device size.
573  */
574 #define SPA_MINDEVSIZE              (64ULL << 20)

576 /*
577  * The location of the pool configuration repository, shared between kernel and
578  * userland.
579  */
580 #define ZPOOL_CACHE                 "/etc/zfs/zpool.cache"

582 /*
583  * vdev states are ordered from least to most healthy.
584  * A vdev that's CANT_OPEN or below is considered unusable.
585  */
586 typedef enum vdev_state {
587     VDEV_STATE_UNKNOWN = 0, /* Uninitialized vdev          */
588     VDEV_STATE_CLOSED,     /* Not currently open    */
589     VDEV_STATE_OFFLINE,    /* Not allowed to open   */
590     VDEV_STATE_REMOVED,    /* Explicitly removed from system */
591     VDEV_STATE_CANT_OPEN,  /* Tried to open, but failed */
592     VDEV_STATE_FAULTED,    /* External request to fault device */
593     VDEV_STATE_DEGRADED,   /* Replicated vdev with unhealthy kids */
594     VDEV_STATE_HEALTHY     /* Presumed good         */
595 } vdev_state_t;

597 #define VDEV_STATE_ONLINE           VDEV_STATE_HEALTHY

599 /*
600  * vdev aux states. When a vdev is in the CANT_OPEN state, the aux field
601  * of the vdev stats structure uses these constants to distinguish why.
602  */
603 typedef enum vdev_aux {
604     VDEV_AUX_NONE,          /* no error                */
605     VDEV_AUX_OPEN_FAILED,  /* ldi_open*() or vn_open() failed */
606     VDEV_AUX_CORRUPT_DATA, /* bad label or disk contents */
607     VDEV_AUX_NO_REPLICAS,  /* insufficient number of replicas */
608     VDEV_AUX_BAD_GUID_SUM, /* vdev guid sum doesn't match */
609     VDEV_AUX_TOO_SMALL,    /* vdev size is too small   */
610     VDEV_AUX_BAD_LABEL,    /* the label is OK but invalid */
611     VDEV_AUX_VERSION_NEWER, /* on-disk version is too new */
612     VDEV_AUX_VERSION_OLDER, /* on-disk version is too old */
613     VDEV_AUX_UNSUP_FEAT,   /* unsupported features     */
614     VDEV_AUX_SPARED,       /* hot spare used in another pool */
615     VDEV_AUX_ERR_EXCEEDED, /* too many errors          */
616     VDEV_AUX_IO_FAILURE,   /* experienced I/O failure  */
617     VDEV_AUX_BAD_LOG,      /* cannot read log chain(s) */
618     VDEV_AUX_EXTERNAL,     /* external diagnosis       */
619     VDEV_AUX_SPLIT_POOL    /* vdev was split off into another pool */
620 } vdev_aux_t;

622 /*
623  * pool state. The following states are written to disk as part of the normal
624  * SPA lifecycle: ACTIVE, EXPORTED, DESTROYED, SPARE, L2CACHE. The remaining
625  * states are software abstractions used at various levels to communicate
626  * pool state.
627  */
628 typedef enum pool_state {
629     POOL_STATE_ACTIVE = 0, /* In active use          */
630     POOL_STATE_EXPORTED,  /* Explicitly exported    */
631     POOL_STATE_DESTROYED, /* Explicitly destroyed   */
632     POOL_STATE_SPARE,     /* Reserved for hot spare use */
633     POOL_STATE_L2CACHE,   /* Level 2 ARC device     */
634     POOL_STATE_UNINITIALIZED, /* Internal spa_t state */

```

```

635     POOL_STATE_UNAVAIL, /* Internal libzfs state */
636     POOL_STATE_POTENTIALLY_ACTIVE /* Internal libzfs state */
637 } pool_state_t;

639 /*
640  * Scan Functions.
641  */
642 typedef enum pool_scan_func {
643     POOL_SCAN_NONE,
644     POOL_SCAN_SCRUB,
645     POOL_SCAN_RESILVER,
646     POOL_SCAN_FUNCS
647 } pool_scan_func_t;

649 /*
650  * ZIO types. Needed to interpret vdev statistics below.
651  */
652 typedef enum zio_type {
653     ZIO_TYPE_NULL = 0,
654     ZIO_TYPE_READ,
655     ZIO_TYPE_WRITE,
656     ZIO_TYPE_FREE,
657     ZIO_TYPE_CLAIM,
658     ZIO_TYPE_IOCTL,
659     ZIO_TYPES
660 } zio_type_t;

662 /*
663  * Pool statistics. Note: all fields should be 64-bit because this
664  * is passed between kernel and userland as an nvlist uint64 array.
665  */
666 typedef struct pool_scan_stat {
667     /* values stored on disk */
668     uint64_t pss_func; /* pool_scan_func_t */
669     uint64_t pss_state; /* dsl_scan_state_t */
670     uint64_t pss_start_time; /* scan start time */
671     uint64_t pss_end_time; /* scan end time */
672     uint64_t pss_to_examine; /* total bytes to scan */
673     uint64_t pss_examined; /* total examined bytes */
674     uint64_t pss_to_process; /* total bytes to process */
675     uint64_t pss_processed; /* total processed bytes */
676     uint64_t pss_errors; /* scan errors */

678     /* values not stored on disk */
679     uint64_t pss_pass_exam; /* examined bytes per scan pass */
680     uint64_t pss_pass_start; /* start time of a scan pass */
681 } pool_scan_stat_t;

683 typedef enum dsl_scan_state {
684     DSS_NONE,
685     DSS_SCANNING,
686     DSS_FINISHED,
687     DSS_CANCELED,
688     DSS_NUM_STATES
689 } dsl_scan_state_t;

692 /*
693  * Vdev statistics. Note: all fields should be 64-bit because this
694  * is passed between kernel and userland as an nvlist uint64 array.
695  */
696 typedef struct vdev_stat {
697     hrtime_t vs_timestamp; /* time since vdev load */
698     uint64_t vs_state; /* vdev state */
699     uint64_t vs_aux; /* see vdev_aux_t */
700     uint64_t vs_alloc; /* space allocated */

```

```

701     uint64_t     vs_space;          /* total capacity */
702     uint64_t     vs_dspace;        /* deflated capacity */
703     uint64_t     vs_rsize;        /* replaceable dev size */
704     uint64_t     vs_esize;        /* expandable dev size */
705     uint64_t     vs_ops[ZIO_TYPES]; /* operation count */
706     uint64_t     vs_bytes[ZIO_TYPES]; /* bytes read/written */
707     uint64_t     vs_read_errors;   /* read errors */
708     uint64_t     vs_write_errors;  /* write errors */
709     uint64_t     vs_checksum_errors; /* checksum errors */
710     uint64_t     vs_self_healed;   /* self-healed bytes */
711     uint64_t     vs_scan_removing; /* removing? */
712     uint64_t     vs_scan_processed; /* scan processed bytes */
713 } vdev_stat_t;

715 /*
716  * DDT statistics. Note: all fields should be 64-bit because this
717  * is passed between kernel and userland as an nvlist uint64 array.
718  */
719 typedef struct ddt_object {
720     uint64_t     ddo_count;        /* number of elements in ddt */
721     uint64_t     ddo_dspace;      /* size of ddt on disk */
722     uint64_t     ddo_mspace;     /* size of ddt in-core */
723 } ddt_object_t;

725 typedef struct ddt_stat {
726     uint64_t     dds_blocks;      /* blocks */
727     uint64_t     dds_lsize;      /* logical size */
728     uint64_t     dds_psize;      /* physical size */
729     uint64_t     dds_dsize;      /* deflated allocated size */
730     uint64_t     dds_ref_blocks; /* referenced blocks */
731     uint64_t     dds_ref_lsize;  /* referenced lsize * refcnt */
732     uint64_t     dds_ref_psize;  /* referenced psize * refcnt */
733     uint64_t     dds_ref_dsize;  /* referenced dsize * refcnt */
734 } ddt_stat_t;

736 typedef struct ddt_histogram {
737     ddt_stat_t     ddh_stat[64]; /* power-of-two histogram buckets */
738 } ddt_histogram_t;

740 #define ZVOL_DRIVER      "zvol"
741 #define ZFS_DRIVER      "zfs"
742 #define ZFS_DEV         "/dev/zfs"

744 /* general zvol path */
745 #define ZVOL_DIR         "/dev/zvol"
746 /* expansion */
747 #define ZVOL_PSEUDO_DEV  "/devices/pseudo/zfs@0:"
748 /* for dump and swap */
749 #define ZVOL_FULL_DEV_DIR ZVOL_DIR "/dsk/"
750 #define ZVOL_FULL_RDEV_DIR ZVOL_DIR "/rdsk/"

752 #define ZVOL_PROP_NAME   "name"
753 #define ZVOL_DEFAULT_BLOCKSIZE 8192

755 /*
756  * /dev/zfs ioctl numbers.
757  */
758 #define ZFS_IOC          ('Z' << 8)

758 typedef enum zfs_ioc {
759     ZFS_IOC_FIRST = ('Z' << 8),
760     ZFS_IOC = ZFS_IOC_FIRST,
761     ZFS_IOC_POOL_CREATE = ZFS_IOC_FIRST,
762     ZFS_IOC_POOL_DESTROY,
763     ZFS_IOC_POOL_IMPORT,

```

```

764     ZFS_IOC_POOL_EXPORT,
765     ZFS_IOC_POOL_CONFIGS,
766     ZFS_IOC_POOL_STATS,
767     ZFS_IOC_POOL_TRYIMPORT,
768     ZFS_IOC_POOL_SCAN,
769     ZFS_IOC_POOL_FREEZE,
770     ZFS_IOC_POOL_UPGRADE,
771     ZFS_IOC_POOL_GET_HISTORY,
772     ZFS_IOC_VDEV_ADD,
773     ZFS_IOC_VDEV_REMOVE,
774     ZFS_IOC_VDEV_SET_STATE,
775     ZFS_IOC_VDEV_ATTACH,
776     ZFS_IOC_VDEV_DETACH,
777     ZFS_IOC_VDEV_SETPATH,
778     ZFS_IOC_VDEV_SETFRU,
779     ZFS_IOC_OBJSET_STATS,
780     ZFS_IOC_OBJSET_ZPLPROPS,
781     ZFS_IOC_DATASET_LIST_NEXT,
782     ZFS_IOC_SNAPSHOT_LIST_NEXT,
783     ZFS_IOC_SET_PROP,
784     ZFS_IOC_CREATE,
785     ZFS_IOC_DESTROY,
786     ZFS_IOC_ROLLBACK,
787     ZFS_IOC_RENAME,
788     ZFS_IOC_RECV,
789     ZFS_IOC_SEND,
790     ZFS_IOC_INJECT_FAULT,
791     ZFS_IOC_CLEAR_FAULT,
792     ZFS_IOC_INJECT_LIST_NEXT,
793     ZFS_IOC_ERROR_LOG,
794     ZFS_IOC_CLEAR,
795     ZFS_IOC_PROMOTE,
796     ZFS_IOC_SNAPSHOT,
797     ZFS_IOC_DSOBJ_TO_DSNAME,
798     ZFS_IOC_OBJ_TO_PATH,
799     ZFS_IOC_POOL_SET_PROPS,
800     ZFS_IOC_POOL_GET_PROPS,
801     ZFS_IOC_SET_FSACL,
802     ZFS_IOC_GET_FSACL,
803     ZFS_IOC_SHARE,
804     ZFS_IOC_INHERIT_PROP,
805     ZFS_IOC_SMB_ACL,
806     ZFS_IOC_USERSPACE_ONE,
807     ZFS_IOC_USERSPACE_MANY,
808     ZFS_IOC_USERSPACE_UPGRADE,
809     ZFS_IOC_HOLD,
810     ZFS_IOC_RELEASE,
811     ZFS_IOC_GET_HOLDS,
812     ZFS_IOC_OBJSET_RECVD_PROPS,
813     ZFS_IOC_VDEV_SPLIT,
814     ZFS_IOC_NEXT_OBJ,
815     ZFS_IOC_DIFF,
816     ZFS_IOC_TMP_SNAPSHOT,
817     ZFS_IOC_OBJ_TO_STATS,
818     ZFS_IOC_SPACE_WRITTEN,
819     ZFS_IOC_SPACE_SNAPS,
820     ZFS_IOC_DESTROY_SNAPS,
821     ZFS_IOC_DESTROY_SNAPS_NVLS,
822     ZFS_IOC_POOL_REGUID,
823     ZFS_IOC_POOL_REOPEN,
824     ZFS_IOC_SEND_PROGRESS,
825     ZFS_IOC_LOG_HISTORY,
826     ZFS_IOC_SEND_NEW,
827     ZFS_IOC_SEND_SPACE,
828     ZFS_IOC_CLONE,
829     ZFS_IOC_LAST

```

```

120     ZFS_IOC_SEND_PROGRESS
829 } zfs_ioc_t;
    unchanged_portion_omitted_

843 /*
844  * Bookmark name values.
845  */
846 #define ZPOOL_ERR_LIST      "error list"
847 #define ZPOOL_ERR_DATASET  "dataset"
848 #define ZPOOL_ERR_OBJECT   "object"

850 #define HIS_MAX_RECORD_LEN  (MAXPATHLEN + MAXPATHLEN + 1)

852 /*
853  * The following are names used in the nvlist describing
854  * the pool's history log.
855  */
856 #define ZPOOL_HIST_RECORD   "history record"
857 #define ZPOOL_HIST_TIME     "history time"
858 #define ZPOOL_HIST_CMD      "history command"
859 #define ZPOOL_HIST_WHO      "history who"
860 #define ZPOOL_HIST_ZONE     "history zone"
861 #define ZPOOL_HIST_HOST     "history hostname"
862 #define ZPOOL_HIST_TXG      "history txg"
863 #define ZPOOL_HIST_INT_EVENT "history internal event"
864 #define ZPOOL_HIST_INT_STR  "history internal str"
865 #define ZPOOL_HIST_INT_NAME "internal_name"
866 #define ZPOOL_HIST_IOCTL    "ioctl"
867 #define ZPOOL_HIST_INPUT_NVL "in_nvl"
868 #define ZPOOL_HIST_OUTPUT_NVL "out_nvl"
869 #define ZPOOL_HIST_DSNAME   "dsname"
870 #define ZPOOL_HIST_DSID     "dsid"
871 #endif /* ! codereview */

873 /*
874  * Flags for ZFS_IOC_VDEV_SET_STATE
875  */
876 #define ZFS_ONLINE_CHECKREMOVE 0x1
877 #define ZFS_ONLINE_UNSPARE     0x2
878 #define ZFS_ONLINE_FORCEFAULT 0x4
879 #define ZFS_ONLINE_EXPAND     0x8
880 #define ZFS_OFFLINE_TEMPORARY 0x1

882 /*
883  * Flags for ZFS_IOC_POOL_IMPORT
884  */
885 #define ZFS_IMPORT_NORMAL      0x0
886 #define ZFS_IMPORT_VERBATIM   0x1
887 #define ZFS_IMPORT_ANY_HOST   0x2
888 #define ZFS_IMPORT_MISSING_LOG 0x4
889 #define ZFS_IMPORT_ONLY       0x8

891 /*
892  * Sysevent payload members. ZFS will generate the following sysevents with the
893  * given payloads:
894  */
895 *     ESC_ZFS_RESILVER_START
896 *     ESC_ZFS_RESILVER_END
897 *     ESC_ZFS_POOL_DESTROY
898 *     ESC_ZFS_POOL_REGUID
899 *
900 *     ZFS_EV_POOL_NAME      DATA_TYPE_STRING
901 *     ZFS_EV_POOL_GUID      DATA_TYPE_UINT64
902 *
903 *     ESC_ZFS_VDEV_REMOVE
904 *     ESC_ZFS_VDEV_CLEAR

```

```

905 *     ESC_ZFS_VDEV_CHECK
906 *
907 *     ZFS_EV_POOL_NAME      DATA_TYPE_STRING
908 *     ZFS_EV_POOL_GUID      DATA_TYPE_UINT64
909 *     ZFS_EV_VDEV_PATH      DATA_TYPE_STRING      (optional)
910 *     ZFS_EV_VDEV_GUID      DATA_TYPE_UINT64
911 */
912 #define ZFS_EV_POOL_NAME     "pool_name"
913 #define ZFS_EV_POOL_GUID     "pool_guid"
914 #define ZFS_EV_VDEV_PATH     "vdev_path"
915 #define ZFS_EV_VDEV_GUID     "vdev_guid"

157 /*
158  * Note: This is encoded on-disk, so new events must be added to the
159  * end, and unused events can not be removed. Be sure to edit
160  * libzfs_pool.c: hist_event_table[].
161  */
162 typedef enum history_internal_events {
163     LOG_NO_EVENT = 0,
164     LOG_POOL_CREATE,
165     LOG_POOL_VDEV_ADD,
166     LOG_POOL_REMOVE,
167     LOG_POOL_DESTROY,
168     LOG_POOL_EXPORT,
169     LOG_POOL_IMPORT,
170     LOG_POOL_VDEV_ATTACH,
171     LOG_POOL_VDEV_REPLACE,
172     LOG_POOL_VDEV_DETACH,
173     LOG_POOL_VDEV_ONLINE,
174     LOG_POOL_VDEV_OFFLINE,
175     LOG_POOL_UPGRADE,
176     LOG_POOL_CLEAR,
177     LOG_POOL_SCAN,
178     LOG_POOL_PROPSSET,
179     LOG_DS_CREATE,
180     LOG_DS_CLONE,
181     LOG_DS_DESTROY,
182     LOG_DS_DESTROY_BEGIN,
183     LOG_DS_INHERIT,
184     LOG_DS_PROPSSET,
185     LOG_DS_QUOTA,
186     LOG_DS_PERM_UPDATE,
187     LOG_DS_PERM_REMOVE,
188     LOG_DS_PERM_WHO_REMOVE,
189     LOG_DS_PROMOTE,
190     LOG_DS_RECEIVE,
191     LOG_DS_RENAME,
192     LOG_DS_RESERVATION,
193     LOG_DS_REPLAY_INC_SYNC,
194     LOG_DS_REPLAY_FULL_SYNC,
195     LOG_DS_ROLLBACK,
196     LOG_DS_SNAPSHOT,
197     LOG_DS_UPGRADE,
198     LOG_DS_REFQUOTA,
199     LOG_DS_REFRESERV,
200     LOG_POOL_SCAN_DONE,
201     LOG_DS_USER_HOLD,
202     LOG_DS_USER_RELEASE,
203     LOG_POOL_SPLIT,
204     LOG_END
205 } history_internal_events_t;

917 #ifdef __cplusplus
918 }
    unchanged_portion_omitted_

```

new/usr/src/uts/common/sys/sunddi.h

1

```
*****
62824 Thu Jun 28 15:57:22 2012
new/usr/src/uts/common/sys/sunddi.h
2882 implement libzfs_core
2883 changing "canmount" property to "on" should not always remount dataset
2900 "zfs snapshot" should be able to create multiple, arbitrary snapshots at on
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Chris Siden <christopher.siden@delphix.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Bill Pijewski <wdp@joyent.com>
Reviewed by: Dan Kruchinin <dan.kruchinin@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21
22 /*
23  * Copyright (c) 1990, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright 2012 Garrett D'Amore <garrett@damore.org>. All rights reserved.
25  * Copyright (c) 2012 by Delphix. All rights reserved.
26  *#endif /* ! codereview */
27 */
28
29 #ifndef _SYS_SUNDDI_H
30 #define _SYS_SUNDDI_H
31
32 /*
33  * Sun Specific DDI definitions
34  */
35
36 #include <sys/isa_defs.h>
37 #include <sys/dditypes.h>
38 #include <sys/ddi_propdefs.h>
39 #include <sys/devops.h>
40 #include <sys/time.h>
41 #include <sys/cmn_err.h>
42 #include <sys/ddidevmap.h>
43 #include <sys/ddi_impldefs.h>
44 #include <sys/ddi_implfuncs.h>
45 #include <sys/ddi_isa.h>
46 #include <sys/model.h>
47 #include <sys/devctl.h>
48 #if defined(__i386) || defined(__amd64)
49 #include <sys/dma_engine.h>
50 #endif
51 #include <sys/sunpm.h>
52 #include <sys/nvpair.h>
53 #include <sys/sysevent.h>
54 #include <sys/thread.h>
```

new/usr/src/uts/common/sys/sunddi.h

2

```
55 #include <sys/stream.h>
56 #if defined(__GNUC__) && defined(_ASM_INLINES) && defined(_KERNEL)
57 #include <asm/sunddi.h>
58 #endif
59 #ifdef _KERNEL
60 #include <sys/ddi_obsolete.h>
61 #endif
62 #include <sys/u8_textprep.h>
63 #include <sys/kiconv.h>
64
65 #ifdef __cplusplus
66 extern "C" {
67 #endif
68
69 /*
70  * Generic Sun DDI definitions.
71  */
72
73 #define DDI_SUCCESS (0) /* successful return */
74 #define DDI_FAILURE (-1) /* unsuccessful return */
75 #define DDI_NOT_WELL_FORMED (-2) /* A dev_info node is not valid */
76 #define DDI_EAGAIN (-3) /* not enough interrupt resources */
77 #define DDI_EINVAL (-4) /* invalid request or arguments */
78 #define DDI_ENOTSUP (-5) /* operation is not supported */
79 #define DDI_EPENDING (-6) /* operation or an event is pending */
80 #define DDI_EALREADY (-7) /* operation already in progress */
81
82 /*
83  * General-purpose DDI error return value definitions
84  */
85 #define DDI_ENOMEM 1 /* memory not available */
86 #define DDI_EBUSY 2 /* busy */
87 #define DDI_ETransport 3 /* transport down */
88 #define DDI_ECONTEXT 4 /* context error */
89
90 /*
91  * General DDI sleep/nosleep allocation flags
92  */
93 #define DDI_SLEEP 0
94 #define DDI_NOSLEEP 1
95
96 /*
97  * The following special nodeid values are reserved for use when creating
98  * nodes ONLY. They specify the attributes of the DDI_NC_PSEUDO class node
99  * being created:
100  *
101  * o DEVI_PSEUDO_NODEID specifies a node without persistence.
102  * o DEVI_SID_NODEID specifies a node with persistence.
103  * o DEVI_SID_HIDDEN_NODEID specifies a hidden node with persistence.
104  *
105  * A node with the 'hidden' attribute will not show up in devinfo snapshots
106  * or in /devices file system.
107  *
108  * A node with the 'persistent' attribute will not be automatically removed by
109  * the framework in the current implementation - driver.conf nodes are without
110  * persistence.
111  *
112  * The actual nodeid value may be assigned by the framework and may be
113  * different than these special values. Drivers may not make assumptions
114  * about the nodeid value that is actually assigned to the node.
115  */
116
117 #define DEVI_PSEUDO_NODEID ((int)-1)
118 #define DEVI_SID_NODEID ((int)-2)
119 #define DEVI_SID_HIDDEN_NODEID ((int)-3)
```

```

121 #define DEVI_SID_HP_NODEID      ((int)-4)
122 #define DEVI_SID_HP_HIDDEN_NODEID ((int)-5)

124 #define DEVI_PSEUDO_NEXNAME     "pseudo"
125 #define DEVI_ISA_NEXNAME       "isa"
126 #define DEVI_EISA_NEXNAME      "eisa"

128 /*
129  * ddi_create_minor_node flags
130  */
131 #define CLONE_DEV                1      /* device is a clone device */
132 #define PRIVONLY_DEV            0x10   /* policy-based permissions only */

134 /*
135  * Historical values used for the flag field in ddi_create_minor_node.
136  * Future use of flag bits should avoid these fields to keep binary
137  * compatibility
138  * #define      GLOBAL_DEV          0x2
139  * #define      NODEBOUND_DEV       0x4
140  * #define      NODESPECIFIC_DEV    0x6
141  * #define      ENUMERATED_DEV      0x8
142  */

144 /*
145  * Device type defines which are used by the 'node_type' element of the
146  * ddi_minor_data structure
147  */
148 #define DDI_NT_SERIAL           "ddi_serial"      /* Serial port */
149 #define DDI_NT_SERIAL_MB       "ddi_serial:mb"    /* the 'built-in' serial */
150 /* ports (the old ttya, b */
151 /* (,c ,d) */
152 #define DDI_NT_SERIAL_DO       "ddi_serial:dialout" /* dialout ports */
153 #define DDI_NT_SERIAL_MB_DO    "ddi_serial:dialout,mb" /* dialout for onboard */
154 /* ports */
155 #define DDI_NT_SERIAL_LOMCON    "ddi_serial:lomcon" /* LOMlite2 console port */

157 /*
158  * * CHAN disk type devices have channel numbers or target numbers.
159  * (i.e. ipi and scsi devices)
160  */
161 #define DDI_NT_BLOCK           "ddi_block"        /* hard disks */
162 /*
163  * The next define is for block type devices that can possible exist on
164  * a sub-bus like the scsi bus or the ipi channel. The 'disks' program
165  * will pick up on this and create logical names like c0t0d0s0 instead of
166  * c0d0s0
167  */
168 #define DDI_NT_BLOCK_CHAN      "ddi_block:channel"
169 #define DDI_NT_BLOCK_WWN       "ddi_block:wwn"
170 #define DDI_NT_CD              "ddi_block:cdrom"  /* rom drives (cd-rom) */
171 #define DDI_NT_CD_CHAN         "ddi_block:cdrom:channel" /* rom drives (scsi type) */
172 #define DDI_NT_FD              "ddi_block:diskette" /* floppy disks */

174 #define DDI_NT_ENCLOSURE        "ddi_enclosure"
175 #define DDI_NT_SCSI_ENCLOSURE  "ddi_enclosure:scsi"

177 #define DDI_NT_BLOCK_SAS       "ddi_block:sas"

179 /*
180  * xVM virtual block devices
181  */
182 #define DDI_NT_BLOCK_XVMD      "ddi_block:xvmd"
183 #define DDI_NT_CD_XVMD         "ddi_block:cdrom:xvmd"

186 #define DDI_NT_TAPE            "ddi_byte:tape"    /* tape drives */

```

```

188 #define DDI_NT_NET            "ddi_network"      /* DLPI network devices */
190 #define DDI_NT_NET_WIFI      "ddi_network:wifi"  /* wifi devices */
192 #define DDI_NT_DISPLAY       "ddi_display"      /* display devices */
194 #define DDI_NT_DISPLAY_DRM   "ddi_display:drm"  /* drm display devices */
196 #define DDI_PSEUDO           "ddi_pseudo"       /* general pseudo devices */
198 #define DDI_NT_AUDIO         "ddi_audio"        /* audio device */
200 #define DDI_NT_MOUSE         "ddi_mouse"        /* mouse device */
202 #define DDI_NT_KEYBOARD      "ddi_keyboard"     /* keyboard device */
204 #define DDI_NT_PARALLEL      "ddi_parallel"     /* parallel port */
206 #define DDI_NT_PRINTER       "ddi_printer"     /* printer device */
208 #define DDI_NT_UGEN          "ddi_generic:usb"  /* USB generic drv */
210 #define DDI_NT_SMP           "ddi_sas_smp"      /* smp devcies */
212 #define DDI_NT_NEXUS         "ddi_ctl:devctl"    /* nexus drivers */
214 #define DDI_NT_SCSI_NEXUS     "ddi_ctl:devctl:scsi" /* nexus drivers */
216 #define DDI_NT_SATA_NEXUS     "ddi_ctl:devctl:sata" /* nexus drivers */
218 #define DDI_NT_IB_NEXUS      "ddi_ctl:devctl:ib" /* nexus drivers */
220 #define DDI_NT_ATTACHMENT_POINT "ddi_ctl:attachment_point" /* attachment pt */
222 #define DDI_NT_SCSI_ATTACHMENT_POINT "ddi_ctl:attachment_point:scsi" /* scsi attachment pt */
223 /* sata attachment pt */
225 #define DDI_NT_SATA_ATTACHMENT_POINT "ddi_ctl:attachment_point:sata" /* sata attachment pt */
226 /* sata attachment pt */
228 #define DDI_NT_SDCARD_ATTACHMENT_POINT "ddi_ctl:attachment_point:sdcard" /* sdcard attachment pt */
229 /* sdcard attachment pt */
231 #define DDI_NT_PCI_ATTACHMENT_POINT "ddi_ctl:attachment_point:pci" /* PCI attachment pt */
232 /* PCI attachment pt */
233 #define DDI_NT_SBD_ATTACHMENT_POINT "ddi_ctl:attachment_point:sbd" /* generic bd attachment pt */
234 /* generic bd attachment pt */
235 #define DDI_NT_FC_ATTACHMENT_POINT "ddi_ctl:attachment_point:fc" /* FC attachment pt */
236 /* FC attachment pt */
237 #define DDI_NT_USB_ATTACHMENT_POINT "ddi_ctl:attachment_point:usb" /* USB devices */
238 /* USB devices */
239 #define DDI_NT_BLOCK_FABRIC    "ddi_block:fabric" /* Fabric Devices */
240 /* Fabric Devices */
241 #define DDI_NT_IB_ATTACHMENT_POINT "ddi_ctl:attachment_point:ib" /* IB devices */
242 /* IB devices */

244 #define DDI_NT_AV_ASYNC       "ddi_av:async"    /* asynchronous AV device */
245 #define DDI_NT_AV_ISOCH      "ddi_av:isoch"    /* isochronous AV device */

247 /* Device types used for agpgart driver related devices */
248 #define DDI_NT_AGP_PSEUDO     "ddi_agp:pseudo"  /* agpgart pseudo device */
249 #define DDI_NT_AGP_MASTER     "ddi_agp:master"  /* agp master device */
250 #define DDI_NT_AGP_TARGET     "ddi_agp:target"  /* agp target device */
251 #define DDI_NT_AGP_CPUGART    "ddi_agp:cpugart" /* amd64 on-cpu gart device */

```

```

253 #define DDI_NT_REGACC      "ddi_tool_reg" /* tool register access */
254 #define DDI_NT_INTRCTL    "ddi_tool_intr" /* tool intr access */

256 /*
257  * DDI event definitions
258  */
259 #define EC_DEVFS          "EC_devfs"      /* Event class devfs */
260 #define EC_DDI           "EC_ddi"        /* Event class ddi */

262 /* Class devfs subclasses */
263 #define ESC_DEVFS_MINOR_CREATE "ESC_devfs_minor_create"
264 #define ESC_DEVFS_MINOR_REMOVE "ESC_devfs_minor_remove"
265 #define ESC_DEVFS_DEVI_ADD "ESC_devfs_devi_add"
266 #define ESC_DEVFS_DEVI_REMOVE "ESC_devfs_devi_remove"
267 #define ESC_DEVFS_INSTANCE_MOD "ESC_devfs_instance_mod"
268 #define ESC_DEVFS_BRANCH_ADD "ESC_devfs_branch_add"
269 #define ESC_DEVFS_BRANCH_REMOVE "ESC_devfs_branch_remove"
270 #define ESC_DEVFS_START "ESC_devfs_start"

272 /* Class ddi subclasses */
273 #define ESC_DDI_INITIATOR_REGISTER "ESC_ddi_initiator_register"
274 #define ESC_DDI_INITIATOR_UNREGISTER "ESC_ddi_initiator_unregister"

276 /* DDI/NDI event publisher */
277 #define EP_DDI SUNW_KERN_PUB"ddi"

279 /*
280  * devfs event class attributes
281  */
282 * The following attributes are private to EC_DEVFS event data.
283 */
284 #define DEVFS_DRIVER_NAME "di.driver"
285 #define DEVFS_INSTANCE "di.instance"
286 #define DEVFS_PATHNAME "di.path"
287 #define DEVFS_DEVI_CLASS "di.devi_class"
288 #define DEVFS_BRANCH_EVENT "di.branch_event"
289 #define DEVFS_MINOR_NAME "mi.name"
290 #define DEVFS_MINOR_NODETYPE "mi.nodetype"
291 #define DEVFS_MINOR_ISCLONE "mi.isclone"
292 #define DEVFS_MINOR_MAJNUM "mi.majorno"
293 #define DEVFS_MINOR_MINORNUM "mi.minorno"

295 /*
296  * ddi event class payload
297  */
298 * The following attributes are private to EC_DDI event data.
299 */
300 #define DDI_DRIVER_NAME "ddi.driver"
301 #define DDI_DRIVER_MAJOR "ddi.major"
302 #define DDI_INSTANCE "ddi.instance"
303 #define DDI_PATHNAME "ddi.path"
304 #define DDI_CLASS "ddi.class"

306 /*
307  * Fault-related definitions
308  */
309 * The specific numeric values have been chosen to be ordered, but
310 * not consecutive, to allow for future interpolation if required.
311 */
312 typedef enum {
313     DDI_SERVICE_LOST = -32,
314     DDI_SERVICE_DEGRADED = -16,
315     DDI_SERVICE_UNAFFECTED = 0,
316     DDI_SERVICE_RESTORED = 16
317 } ddi_fault_impact_t;

```

```

319 typedef enum {
320     DDI_DATAPATH_FAULT = -32,
321     DDI_DEVICE_FAULT = -16,
322     DDI_EXTERNAL_FAULT = 0
323 } ddi_fault_location_t;

325 typedef enum {
326     DDI_DEVSTATE_OFFLINE = -32,
327     DDI_DEVSTATE_DOWN = -16,
328     DDI_DEVSTATE_QUIESCED = 0,
329     DDI_DEVSTATE_DEGRADED = 16,
330     DDI_DEVSTATE_UP = 32
331 } ddi_devstate_t;

333 #ifdef _KERNEL

335 /*
336  * Common property definitions
337  */
338 #define DDI_FORCEATTACH "ddi-forceattach"
339 #define DDI_NO_AUTODETACH "ddi-no-autodetach"
340 #define DDI_VHCI_CLASS "ddi-vhci-class"
341 #define DDI_NO_ROOT_SUPPORT "ddi-no-root-support"
342 #define DDI_OPEN_RETURNS_EINTR "ddi-open-returns-eintr"
343 #define DDI_DEVID_REGISTRANT "ddi-devid-registrant"

345 /*
346  * Values that the function supplied to the dev_info
347  * tree traversal functions defined below must return.
348  */

350 /*
351  * Continue search, if appropriate.
352  */
353 #define DDI_WALK_CONTINUE 0

355 /*
356  * Terminate current depth of traversal. That is, terminate
357  * the current traversal of children nodes, but continue
358  * traversing sibling nodes and their children (if any).
359  */

361 #define DDI_WALK_PRUNECCHILD -1

363 /*
364  * Terminate current width of traversal. That is, terminate
365  * the current traversal of sibling nodes, but continue with
366  * traversing children nodes and their siblings (if appropriate).
367  */

369 #define DDI_WALK_PRUNESIB -2

371 /*
372  * Terminate the entire search.
373  */

375 #define DDI_WALK_TERMINATE -3

377 /*
378  * Terminate the entire search because an error occurred in function
379  */
380 #define DDI_WALK_ERROR -4

382 /*
383  * Drivers that are prepared to support full driver layering
384  * should create and export a null-valued property of the following

```



```

385 * name.
386 *
387 * Such drivers should be prepared to be called with FKLYR in
388 * the 'flag' argument of their open(9E), close(9E) routines, and
389 * with FKIOCTL in the 'mode' argument of their ioctl(9E) routines.
390 *
391 * See ioctl(9E) and ddi_copyin(9F) for details.
392 */
393 #define DDI_KERNEL_IOCTL      "ddi-kernel-ioctl"

395 /*
396 * Model definitions for ddi_mmap_get_model(9F) and ddi_model_convert_from(9F).
397 */
398 #define DDI_MODEL_MASK        DATAMODEL_MASK /* Note: 0x0FF00000 */
399 #define DDI_MODEL_ILP32       DATAMODEL_ILP32
400 #define DDI_MODEL_LP64        DATAMODEL_LP64
401 #define DDI_MODEL_NATIVE      DATAMODEL_NATIVE
402 #define DDI_MODEL_NONE        DATAMODEL_NONE

404 /* if set to B_TRUE is DER_MODE is equivalent to DERE_PANIC */
405 extern boolean_t ddi_err_panic;

407 /*
408 * Defines for ddi_err().
409 */
410 typedef enum {
411     DER_INVALID = 0,          /* must be 0 */
412     DER_CONT = 1,
413     DER_CONS,
414     DER_LOG,
415     DER_VERB,
416     DER_NOTE,
417     DER_WARN,
418     DER_PANIC,
419     DER_MODE,
420     DER_DEBUG
421 } ddi_err_t;

423 extern void ddi_err(ddi_err_t de, dev_info_t *rdip, const char *fmt, ...);

426 extern char *ddi_strdup(const char *str, int flag);
427 extern char *strdup(const char *str);
428 extern void strfree(char *str);

430 /*
431 * Functions and data references which really should be in <sys/ddi.h>
432 */

434 extern int maxphys;
435 extern void minphys(struct buf *);
436 extern int physio(int (*)(struct buf *), struct buf *, dev_t,
437     int, void (*)(struct buf *), struct uio *);
438 extern void disksort(struct diskhd *, struct buf *);

440 extern size_t strlen(const char *) __PURE;
441 extern size_t strlen(const char *, size_t) __PURE;
442 extern char *strcpy(char *, const char *);
443 extern char *strncpy(char *, const char *, size_t);
444 /* Need to be consistent with <string.h> C++ definition for strchr() */
445 #if __cplusplus >= 199711L
446 extern const char *strchr(const char *, int);
447 #ifndef _STRCHR_INLINE
448 #define _STRCHR_INLINE
449 extern "C++" {
450     inline char *strchr(char *__s, int __c) {

```

```

451         return (char *)strchr((const char *)__s, __c);
452     }
453 }
454 #endif /* _STRCHR_INLINE */
455 #else
456 extern char *strchr(const char *, int);
457 #endif /* __cplusplus >= 199711L */
458 #define DDI_STRSAME(s1, s2)    ((*s1) == *(s2)) && (strcmp((s1), (s2)) == 0)
459 extern int strcmp(const char *, const char *) __PURE;
460 extern int strncmp(const char *, const char *, size_t) __PURE;
461 extern char *strncat(char *, const char *, size_t);
462 extern size_t strlcat(char *, const char *, size_t);
463 extern size_t strlcpy(char *, const char *, size_t);
464 extern size_t strspn(const char *, const char *);
465 extern size_t strcspn(const char *, const char *);
466 #endif /* ! codereview */
467 extern int bcmp(const void *, const void *, size_t) __PURE;
468 extern int stoi(char **);
469 extern void numtos(ulong_t, char *);
470 extern void bcopy(const void *, void *, size_t);
471 extern void bzero(void *, size_t);

473 extern void *memcpy(void *, const void *, size_t);
474 extern void *memset(void *, int, size_t);
475 extern void *memmove(void *, const void *, size_t);
476 extern int memcmp(const void *, const void *, size_t) __PURE;
477 /* Need to be consistent with <string.h> C++ definition for memchr() */
478 #if __cplusplus >= 199711L
479 extern const void *memchr(const void *, int, size_t);
480 #ifndef _MEMCHR_INLINE
481 #define _MEMCHR_INLINE
482 extern "C++" {
483     inline void *memchr(void *__s, int __c, size_t __n) {
484         return (void *)memchr((const void *)__s, __c, __n);
485     }
486 }
487 #endif /* _MEMCHR_INLINE */
488 #else
489 extern void *memchr(const void *, int, size_t);
490 #endif /* __cplusplus >= 199711L */

492 extern int ddi_strtol(const char *, char **, int, long *);
493 extern int ddi_strtoul(const char *, char **, int, unsigned long *);
494 extern int ddi_strtoll(const char *, char **, int, longlong_t *);
495 extern int ddi_strtoull(const char *, char **, int, u_longlong_t *);

497 /*
498 * kiconv functions and their macros.
499 */
500 #define KICONV_IGNORE_NULL    (0x0001)
501 #define KICONV_REPLACE_INVALID (0x0002)

503 extern kiconv_t kiconv_open(const char *, const char *);
504 extern size_t kiconv(kiconv_t, char **, size_t *, char **, size_t *, int *);
505 extern int kiconv_close(kiconv_t);
506 extern size_t kiconvstr(const char *, const char *, char *, size_t *, char *,
507     size_t *, int, int *);

509 /*
510 * ddi_map_regs
511 *
512 * Map in the register set given by rnumber.
513 * The register number determine which register
514 * set will be mapped if more than one exists.
515 * The parent driver gets the information
516 * from parent private data and sets up the

```

```

517 *      appropriate mappings and returns the kernel
518 *      virtual address of the register set in *kaddrp.
519 *      The offset specifies an offset into the register
520 *      space to start from and len indicates the size
521 *      of the area to map. If len and offset are 0 then
522 *      the entire space is mapped. It returns DDI_SUCCESS on
523 *      success or DDI_FAILURE otherwise.
524 *
525 */
526 int
527 ddi_map_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp,
528             off_t offset, off_t len);

530 /*
531 * ddi_unmap_regs
532 *
533 *      Undo mappings set up by ddi_map_regs.
534 *      The register number determines which register
535 *      set will be unmapped if more than one exists.
536 *      This is provided for drivers preparing
537 *      to detach themselves from the system to
538 *      allow them to release allocated mappings.
539 *
540 *      The kaddrp and len specify the area to be
541 *      unmapped. *kaddrp was returned from ddi_map_regs
542 *      and len should match what ddi_map_regs was called
543 *      with.
544 */

546 void
547 ddi_unmap_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp,
548             off_t offset, off_t len);

550 int
551 ddi_map(dev_info_t *dp, ddi_map_req_t *mp, off_t offset, off_t len,
552         caddr_t *addrp);

554 int
555 ddi_apply_range(dev_info_t *dip, dev_info_t *rdip, struct regspec *rp);

557 /*
558 * ddi_rnumber_to_regspec: Not for use by leaf drivers.
559 */
560 struct regspec *
561 ddi_rnumber_to_regspec(dev_info_t *dip, int rnumber);

563 int
564 ddi_bus_map(dev_info_t *dip, dev_info_t *rdip, ddi_map_req_t *mp, off_t offset,
565            off_t len, caddr_t *vaddrp);

567 int
568 nullbusmap(dev_info_t *dip, dev_info_t *rdip, ddi_map_req_t *mp, off_t offset,
569            off_t len, caddr_t *vaddrp);

571 int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *val_p);
572 int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *val_p);
573 int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *val_p);
574 int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *val_p);

576 int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t val);
577 int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t val);
578 int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t val);
579 int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t val);

581 /*
582 * Peek and poke to and from a uio structure in xfersize pieces,

```

```

583 * using the parent nexi.
584 */
585 int ddi_peekpokeio(dev_info_t *devi, struct uio *uio, enum uio_rw rw,
586                  caddr_t addr, size_t len, uint_t xfersize);

588 /*
589 * Pagesize conversions using the parent nexi
590 */
591 unsigned long ddi_btop(dev_info_t *dip, unsigned long bytes);
592 unsigned long ddi_btopr(dev_info_t *dip, unsigned long bytes);
593 unsigned long ddi_ptob(dev_info_t *dip, unsigned long pages);

595 /*
596 * There are no more "block" interrupt functions, per se.
597 * All thread of control should be done with MP/MT lockings.
598 *
599 * However, there are certain times in which a driver needs
600 * absolutely a critical guaranteed non-preemptable time
601 * in which to execute a few instructions.
602 *
603 * The following pair of functions attempt to guarantee this,
604 * but they are dangerous to use. That is, use them with
605 * extreme care. They do not guarantee to stop other processors
606 * from executing, but they do guarantee that the caller
607 * of ddi_enter_critical will continue to run until the
608 * caller calls ddi_exit_critical. No intervening DDI functions
609 * may be called between an entry and an exit from a critical
610 * region.
611 *
612 * ddi_enter_critical returns an integer identifier which must
613 * be passed to ddi_exit_critical.
614 *
615 * Be very sparing in the use of these functions since it is
616 * likely that absolutely nothing else can occur in the system
617 * whilst in the critical region.
618 */

620 unsigned int
621 ddi_enter_critical(void);

623 void
624 ddi_exit_critical(unsigned int);

626 /*
627 * devmap functions
628 */
629 int
630 devmap_setup(dev_t dev, off_t off, ddi_as_handle_t as, caddr_t *addrp,
631             size_t len, uint_t prot, uint_t maxprot, uint_t flags,
632             struct cred *cred);

634 int
635 ddi_devmap_segmap(dev_t dev, off_t off, ddi_as_handle_t as, caddr_t *addrp,
636                 off_t len, uint_t prot, uint_t maxprot, uint_t flags,
637                 struct cred *cred);

639 int
640 devmap_load(devmap_cookie_t dhp, off_t offset, size_t len, uint_t type,
641            uint_t rw);

643 int
644 devmap_unload(devmap_cookie_t dhp, off_t offset, size_t len);

646 int
647 devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip,
648                    struct devmap_callback_ctl *callback_ops,

```

```

649     uint_t rnumber, offset_t roff, size_t len, uint_t maxprot,
650     uint_t flags, ddi_device_acc_attr_t *accattrp);

652 int
653 devmap_umem_setup(devmap_cookie_t dhp, dev_info_t *dip,
654     struct devmap_callback_ctl *callback_ops,
655     ddi_umem_cookie_t cookie, offset_t off, size_t len, uint_t maxprot,
656     uint_t flags, ddi_device_acc_attr_t *accattrp);

658 int
659 devmap_devmem_remap(devmap_cookie_t dhp, dev_info_t *dip,
660     uint_t rnumber, offset_t roff, size_t len, uint_t maxprot,
661     uint_t flags, ddi_device_acc_attr_t *accattrp);

663 int
664 devmap_umem_remap(devmap_cookie_t dhp, dev_info_t *dip,
665     ddi_umem_cookie_t cookie, offset_t off, size_t len, uint_t maxprot,
666     uint_t flags, ddi_device_acc_attr_t *accattrp);

668 void
669 devmap_set_ctx_timeout(devmap_cookie_t dhp, clock_t ticks);

671 int
672 devmap_default_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
673     size_t len, uint_t type, uint_t rw);

675 int
676 devmap_do_ctxmgt(devmap_cookie_t dhp, void *pvtp, offset_t off, size_t len,
677     uint_t type, uint_t rw, int (*ctxmgt)(devmap_cookie_t, void *, offset_t,
678     size_t, uint_t, uint_t));

681 void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t *cookiep);

683 void ddi_umem_free(ddi_umem_cookie_t cookie);

685 /*
686  * Functions to lock user memory and do repeated I/O or do devmap_umem_setup
687  */
688 int
689 ddi_umem_lock(caddr_t addr, size_t size, int flags, ddi_umem_cookie_t *cookie);

691 void
692 ddi_umem_unlock(ddi_umem_cookie_t cookie);

694 struct buf *
695 ddi_umem_iosetup(ddi_umem_cookie_t cookie, off_t off, size_t len, int direction,
696     dev_t dev, daddr_t blkno, int (*iodone)(struct buf *), int sleepflag);

698 /*
699  * Mapping functions
700  */
701 int
702 ddi_segmap(dev_t dev, off_t offset, struct as *asp, caddr_t *addrp, off_t len,
703     uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp);

705 int
706 ddi_segmap_setup(dev_t dev, off_t offset, struct as *as, caddr_t *addrp,
707     off_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cred,
708     ddi_device_acc_attr_t *accattrp, uint_t rnumber);

710 int
711 ddi_map_fault(dev_info_t *dip, struct hat *hat, struct seg *seg, caddr_t addr,
712     struct devpage *dp, pfn_t pfn, uint_t prot, uint_t lock);

714 int

```

```

715 ddi_device_mapping_check(dev_t dev, ddi_device_acc_attr_t *accattrp,
716     uint_t rnumber, uint_t *hat_flags);

718 /*
719  * Property functions: See also, ddi_propdefs.h.
720  * In general, the underlying driver MUST be held
721  * to call it's property functions.
722  */

724 /*
725  * Used to create, modify, and lookup integer properties
726  */
727 int ddi_prop_get_int(dev_t match_dev, dev_info_t *dip, uint_t flags,
728     char *name, int defvalue);
729 int64_t ddi_prop_get_int64(dev_t match_dev, dev_info_t *dip, uint_t flags,
730     char *name, int64_t defvalue);
731 int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip, uint_t flags,
732     char *name, int **data, uint_t *nelements);
733 int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip, uint_t flags,
734     char *name, int64_t **data, uint_t *nelements);
735 int ddi_prop_update_int(dev_t match_dev, dev_info_t *dip,
736     char *name, int data);
737 int ddi_prop_update_int64(dev_t match_dev, dev_info_t *dip,
738     char *name, int64_t data);
739 int ddi_prop_update_int_array(dev_t match_dev, dev_info_t *dip,
740     char *name, int *data, uint_t nelements);
741 int ddi_prop_update_int64_array(dev_t match_dev, dev_info_t *dip,
742     char *name, int64_t *data, uint_t nelements);
743 /*
744  * Used to create, modify, and lookup string properties
745  */
746 int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip, uint_t flags,
747     char *name, char **data);
748 int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip, uint_t flags,
749     char *name, char ***data, uint_t *nelements);
750 int ddi_prop_update_string(dev_t match_dev, dev_info_t *dip,
751     char *name, char *data);
752 int ddi_prop_update_string_array(dev_t match_dev, dev_info_t *dip,
753     char *name, char **data, uint_t nelements);

755 /*
756  * Used to create, modify, and lookup byte properties
757  */
758 int ddi_prop_lookup_byte_array(dev_t match_dev, dev_info_t *dip, uint_t flags,
759     char *name, uchar_t **data, uint_t *nelements);
760 int ddi_prop_update_byte_array(dev_t match_dev, dev_info_t *dip,
761     char *name, uchar_t *data, uint_t nelements);

763 /*
764  * Used to verify the existence of a property or to see if a boolean
765  * property exists.
766  */
767 int ddi_prop_exists(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name);

769 /*
770  * Used to free the data returned by the above property routines.
771  */
772 void ddi_prop_free(void *data);

774 /*
775  * nopropop: For internal use in 'dummy' cb_prop_op functions only
776  */

778 int
779 nopropop(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op, int mod_flags,
780     char *name, caddr_t valuep, int *lengthp);

```

```

782 /*
783 * ddi_prop_op: The basic property operator for drivers.
784 *
785 * In ddi_prop_op, the type of valuep is interpreted based on prop_op:
786 *
787 *     prop_op          valuep
788 *     -----          -
789 *
790 *     PROP_LEN          <unused>
791 *
792 *     PROP_LEN_AND_VAL_BUF  Pointer to callers buffer
793 *
794 *     PROP_LEN_AND_VAL_ALLOC  Address of callers pointer (will be set to
795 *                             address of allocated buffer, if successful)
796 */

798 int
799 ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op, int mod_flags,
800            char *name, caddr_t valuep, int *lengthp);

802 /* ddi_prop_op_size: for drivers that implement size in bytes */
803 int
804 ddi_prop_op_size(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
805                int mod_flags, char *name, caddr_t valuep, int *lengthp,
806                uint64_t size64);

808 /* ddi_prop_op_size_blksize: like ddi_prop_op_size, in blksize blocks */
809 int
810 ddi_prop_op_size_blksize(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
811                        int mod_flags, char *name, caddr_t valuep, int *lengthp,
812                        uint64_t size64, uint_t blksize);

814 /* ddi_prop_op_nblocks: for drivers that implement size in DEV_BSIZE blocks */
815 int
816 ddi_prop_op_nblocks(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
817                   int mod_flags, char *name, caddr_t valuep, int *lengthp,
818                   uint64_t nblocks64);

820 /* ddi_prop_op_nblocks_blksize: like ddi_prop_op_nblocks, in blksize blocks */
821 int
822 ddi_prop_op_nblocks_blksize(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
823                          int mod_flags, char *name, caddr_t valuep, int *lengthp,
824                          uint64_t nblocks64, uint_t blksize);

826 /*
827 * Variable length props...
828 */

830 /*
831 * ddi_getlongprop: Get variable length property len+val into a buffer
832 *                  allocated by property provider via kmem_alloc. Requester
833 *                  is responsible for freeing returned property via kmem_free.
834 *
835 * Arguments:
836 *
837 * dev:   Input: dev_t of property.
838 * dip:   Input: dev_info_t pointer of child.
839 * flags: Input: Possible flag modifiers are:
840 *         DDI_PROP_DONTPASS: Don't pass to parent if prop not found.
841 *         DDI_PROP_CANSLEEP: Memory allocation may sleep.
842 * name:  Input: name of property.
843 * valuep: Output: Addr of callers buffer pointer.
844 * lengthp: Output: *lengthp will contain prop length on exit.
845 *
846 * Possible Returns:

```

```

847 *
848 *         DDI_PROP_SUCCESS: Prop found and returned.
849 *         DDI_PROP_NOT_FOUND: Prop not found
850 *         DDI_PROP_UNDEFINED: Prop explicitly undefined.
851 *         DDI_PROP_NO_MEMORY: Prop found, but unable to alloc mem.
852 */

854 int
855 ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags,
856               char *name, caddr_t valuep, int *lengthp);

858 /*
859 *
860 * ddi_getlongprop_buf: Get long prop into pre-allocated callers
861 *                       buffer. (no memory allocation by provider).
862 *
863 * dev:   Input: dev_t of property.
864 * dip:   Input: dev_info_t pointer of child.
865 * flags: Input: DDI_PROP_DONTPASS or NULL
866 * name:  Input: name of property
867 * valuep: Input: ptr to callers buffer.
868 * lengthp:I/O: ptr to length of callers buffer on entry,
869 *              actual length of property on exit.
870 *
871 * Possible returns:
872 *
873 *         DDI_PROP_SUCCESS Prop found and returned
874 *         DDI_PROP_NOT_FOUND Prop not found
875 *         DDI_PROP_UNDEFINED Prop explicitly undefined.
876 *         DDI_PROP_BUF_TOO_SMALL Prop found, callers buf too small,
877 *                                no value returned, but actual prop
878 *                                length returned in *lengthp
879 *
880 */

882 int
883 ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags,
884                   char *name, caddr_t valuep, int *lengthp);

886 /*
887 * Integer/boolean sized props.
888 *
889 * Call is value only... returns found boolean or int sized prop value or
890 * defvalue if prop not found or is wrong length or is explicitly undefined.
891 * Only flag is DDI_PROP_DONTPASS...
892 *
893 * By convention, this interface returns boolean (0) sized properties
894 * as value (int)1.
895 */

897 int
898 ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name, int defvalue);

900 /*
901 * Get prop length interface: flags are 0 or DDI_PROP_DONTPASS
902 * if returns DDI_PROP_SUCCESS, length returned in *lengthp.
903 */

905 int
906 ddi_getpropplen(dev_t dev, dev_info_t *dip, int flags, char *name, int *lengthp);

909 /*
910 * Interface to create/modify a managed property on child's behalf...
911 * Only flag is DDI_PROP_CANSLEEP to allow memory allocation to sleep
912 * if no memory available for internal prop structure. Long property

```

```

913 * (non integer sized) value references are not copied.
914 *
915 * Define property with DDI_DEV_T_NONE dev_t for properties not associated
916 * with any particular dev_t. Use the same dev_t when modifying or undefining
917 * a property.
918 *
919 * No guarantee on order of property search, so don't mix the same
920 * property name with wildcard and non-wildcard dev_t's.
921 */

923 /*
924 * ddi_prop_create:    Define a managed property:
925 */

927 int
928 ddi_prop_create(dev_t dev, dev_info_t *dip, int flag,
929                char *name, caddr_t value, int length);

931 /*
932 * ddi_prop_modify:   Modify a managed property value
933 */

935 int
936 ddi_prop_modify(dev_t dev, dev_info_t *dip, int flag,
937                char *name, caddr_t value, int length);

939 /*
940 * ddi_prop_remove:   Undefine a managed property:
941 */

943 int
944 ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);

946 /*
947 * ddi_prop_remove_all:    Used before unloading a driver to remove
948 *                          all properties. (undefines all dev_t's props.)
949 *                          Also removes 'undefined' prop defs.
950 */

952 void
953 ddi_prop_remove_all(dev_info_t *dip);

956 /*
957 * ddi_prop_undefine:    Explicitly undefine a property. Property
958 *                       searches which match this property return
959 *                       the error code DDI_PROP_UNDEFINED.
960 *
961 *                       Use ddi_prop_remove to negate effect of
962 *                       ddi_prop_undefine
963 */

965 int
966 ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flag, char *name);

969 /*
970 * ddi_prop_cache_invalidate
971 *   Invalidate a property in the current cached
972 *   devinfo snapshot - next cached snapshot will
973 *   return the latest property value available.
974 */
975 void
976 ddi_prop_cache_invalidate(dev_t dev, dev_info_t *dip, char *name, int flags);

978 */

```

```

979 * The default ddi_bus_prop_op wrapper...
980 */

982 int
983 ddi_bus_prop_op(dev_t dev, dev_info_t *dip, dev_info_t *ch_dip,
984                ddi_prop_op_t prop_op, int mod_flags,
985                char *name, caddr_t valuep, int *lengthp);

988 /*
989 * Routines to traverse the tree of dev_info nodes.
990 * The general idea of these functions is to provide
991 * various tree traversal utilities. For each node
992 * that the tree traversal function finds, a caller
993 * supplied function is called with arguments of
994 * the current node and a caller supplied argument.
995 * The caller supplied function should return one
996 * of the integer values defined below which will
997 * indicate to the tree traversal function whether
998 * the traversal should be continued, and if so, how,
999 * or whether the traversal should terminate.
1000 */

1002 /*
1003 * This general-purpose routine traverses the tree of dev_info nodes,
1004 * starting from the given node, and calls the given function for each
1005 * node that it finds with the current node and the pointer arg (which
1006 * can point to a structure of information that the function
1007 * needs) as arguments.
1008 *
1009 * It does the walk a layer at a time, not depth-first.
1010 *
1011 * The given function must return one of the values defined above.
1012 */
1013 */

1015 void
1016 ddi_walk_devs(dev_info_t *, int (*)(dev_info_t *, void *), void *);

1018 /*
1019 * Routines to get at elements of the dev_info structure
1020 */

1022 /*
1023 * ddi_node_name gets the device's 'name' from the device node.
1024 *
1025 * ddi_binding_name gets the string the OS used to bind the node to a driver,
1026 * in certain cases, the binding name may be different from the node name,
1027 * if the node name does not name a specific device driver.
1028 *
1029 * ddi_get_name is a synonym for ddi_binding_name().
1030 */
1031 char *
1032 ddi_get_name(dev_info_t *dip);

1034 char *
1035 ddi_binding_name(dev_info_t *dip);

1037 const char *
1038 ddi_driver_name(dev_info_t *dip);

1040 major_t
1041 ddi_driver_major(dev_info_t *dip);

1043 major_t
1044 ddi_compatible_driver_major(dev_info_t *dip, char **formp);

```

```

1046 char *
1047 ddi_node_name(dev_info_t *dip);

1049 int
1050 ddi_get_nodeid(dev_info_t *dip);

1052 int
1053 ddi_get_instance(dev_info_t *dip);

1055 struct dev_ops *
1056 ddi_get_driver(dev_info_t *dip);

1058 void
1059 ddi_set_driver(dev_info_t *dip, struct dev_ops *devo);

1061 void
1062 ddi_set_driver_private(dev_info_t *dip, void *data);

1064 void *
1065 ddi_get_driver_private(dev_info_t *dip);

1067 /*
1068  * ddi_dev_is_needed tells system that a device is about to use a
1069  * component. Returns when component is ready.
1070  */
1071 int
1072 ddi_dev_is_needed(dev_info_t *dip, int cmpt, int level);

1074 /*
1075  * check if DDI_SUSPEND may result in power being removed from a device.
1076  */
1077 int
1078 ddi_removing_power(dev_info_t *dip);

1080 /*
1081  * (Obsolete) power entry point
1082  */
1083 int
1084 ddi_power(dev_info_t *dip, int cmpt, int level);

1086 /*
1087  * ddi_get_parent requires that the branch of the tree with the
1088  * node be held (ddi_hold_installed_driver) or that the devinfo tree
1089  * lock be held
1090  */
1091 dev_info_t *
1092 ddi_get_parent(dev_info_t *dip);

1094 /*
1095  * ddi_get_child and ddi_get_next_sibling require that the devinfo
1096  * tree lock be held
1097  */
1098 dev_info_t *
1099 ddi_get_child(dev_info_t *dip);

1101 dev_info_t *
1102 ddi_get_next_sibling(dev_info_t *dip);

1104 dev_info_t *
1105 ddi_get_next(dev_info_t *dip);

1107 void
1108 ddi_set_next(dev_info_t *dip, dev_info_t *nextdip);

1110 /*

```

```

1111  * dev_info manipulation functions
1112  */

1114 /*
1115  * Add and remove child devices. These are part of the system framework.
1116  *
1117  * ddi_add_child creates a dev_info structure with the passed name,
1118  * nodeid and instance arguments and makes it a child of pdip. Devices
1119  * that are known directly by the hardware have real nodeids; devices
1120  * that are software constructs use the defined DEVI_PSEUDO_NODEID
1121  * for the node id.
1122  *
1123  * ddi_remove_node removes the node from the tree. This fails if this
1124  * child has children. Parent and driver private data should already
1125  * be released (freed) prior to calling this function. If flag is
1126  * non-zero, the child is removed from it's linked list of instances.
1127  */
1128 dev_info_t *
1129 ddi_add_child(dev_info_t *pdip, char *name, uint_t nodeid, uint_t instance);

1131 int
1132 ddi_remove_child(dev_info_t *dip, int flag);

1134 /*
1135  * Given the major number for a driver, make sure that dev_info nodes
1136  * are created form the driver's hwconf file, the driver for the named
1137  * device is loaded and attached, as well as any drivers for parent devices.
1138  * Return a pointer to the driver's dev_ops struct with the dev_ops held.
1139  * Note - Callers must release the dev_ops with ddi_rele_driver.
1140  *
1141  * When a driver is held, the branch of the devinfo tree from any of the
1142  * drivers devinfos to the root node are automatically held. This only
1143  * applies to tree traversals up (and back down) the tree following the
1144  * parent pointers.
1145  *
1146  * Use of this interface is discouraged, it may be removed in a future release.
1147  */
1148 struct dev_ops *
1149 ddi_hold_installed_driver(major_t major);

1151 void
1152 ddi_rele_driver(major_t major);

1154 /*
1155  * Attach and hold the specified instance of a driver. The flags argument
1156  * should be zero.
1157  */
1158 dev_info_t *
1159 ddi_hold_devi_by_instance(major_t major, int instance, int flags);

1161 void
1162 ddi_release_devi(dev_info_t *);

1164 /*
1165  * Associate a streams queue with a devinfo node
1166  */
1167 void
1168 ddi_assoc_queue_with_devi(queue_t *, dev_info_t *);

1170 /*
1171  * Given the identifier string passed, make sure that dev_info nodes
1172  * are created form the driver's hwconf file, the driver for the named
1173  * device is loaded and attached, as well as any drivers for parent devices.
1174  *
1175  * Note that the driver is not held and is subject to being removed the instant
1176  * this call completes. You probably really want ddi_hold_installed_driver.

```

```

1177 */
1178 int
1179 ddi_install_driver(char *idstring);

1181 /*
1182  * Routines that return specific nodes
1183  */

1185 dev_info_t *
1186 ddi_root_node(void);

1188 /*
1189  * Given a name and an instance number, find and return the
1190  * dev_info from the current state of the device tree.
1191  *
1192  * If instance number is -1, return the first named instance.
1193  *
1194  * If attached is 1, exclude all nodes that are < DS_ATTACHED
1195  *
1196  * Requires that the devinfo tree be locked.
1197  * If attached is 1, the driver must be held.
1198  */
1199 dev_info_t *
1200 ddi_find_devinfo(char *name, int instance, int attached);

1202 /*
1203  * Synchronization of I/O with respect to various
1204  * caches and system write buffers.
1205  *
1206  * Done at varying points during an I/O transfer (including at the
1207  * removal of an I/O mapping).
1208  *
1209  * Due to the support of systems with write buffers which may
1210  * not be able to be turned off, this function *must* be used at
1211  * any point in which data consistency might be required.
1212  *
1213  * Generally this means that if a memory object has multiple mappings
1214  * (both for I/O, as described by the handle, and the IU, via, e.g.
1215  * a call to ddi_dma_kvaddrp), and one mapping may have been
1216  * used to modify the memory object, this function must be called
1217  * to ensure that the modification of the memory object is
1218  * complete, as well as possibly to inform other mappings of
1219  * the object that any cached references to the object are
1220  * now stale (and flush or invalidate these stale cache references
1221  * as necessary).
1222  *
1223  * The function ddi_dma_sync() provides the general interface with
1224  * respect to this capability. Generally, ddi_dma_free() (below) may
1225  * be used in preference to ddi_dma_sync() as ddi_dma_free() calls
1226  * ddi_dma_sync().
1227  *
1228  * Returns 0 if all caches that exist and are specified by cache_flags
1229  * are successfully operated on, else -1.
1230  *
1231  * The argument offset specifies an offset into the mapping of the mapped
1232  * object in which to perform the synchronization. It will be silently
1233  * truncated to the granularity of underlying cache line sizes as
1234  * appropriate.
1235  *
1236  * The argument len specifies a length starting from offset in which to
1237  * perform the synchronization. A value of (uint_t) -1 means that the length
1238  * proceeds from offset to the end of the mapping. The length argument
1239  * will silently rounded up to the granularity of underlying cache line
1240  * sizes as appropriate.
1241  *
1242  * The argument flags specifies what to synchronize (the device's view of

```

```

1243  * the object or the cpu's view of the object).
1244  *
1245  * Inquiring minds want to know when ddi_dma_sync should be used:
1246  *
1247  * +   When an object is mapped for dma, assume that an
1248  *     implicit ddi_dma_sync() is done for you.
1249  *
1250  * +   When an object is unmapped (ddi_dma_free()), assume
1251  *     that an implicit ddi_dma_sync() is done for you.
1252  *
1253  * +   At any time between the two times above that the
1254  *     memory object may have been modified by either
1255  *     the DMA device or a processor and you wish that
1256  *     the change be noticed by the master that didn't
1257  *     do the modifying.
1258  *
1259  * Clearly, only the third case above requires the use of ddi_dma_sync.
1260  *
1261  * Inquiring minds also want to know which flag to use:
1262  *
1263  * +   If you *modify* with a cpu the object, you use
1264  *     ddi_dma_sync(...DDI_DMA_SYNC_FORDEV) (you are making sure
1265  *     that the DMA device sees the changes you made).
1266  *
1267  * +   If you are checking, with the processor, an area
1268  *     of the object that the DMA device *may* have modified,
1269  *     you use ddi_dma_sync(...DDI_DMA_SYNC_FORCPU) (you are
1270  *     making sure that the processor(s) will see the changes
1271  *     that the DMA device may have made).
1272  */

1274 int
1275 ddi_dma_sync(ddi_dma_handle_t handle, off_t offset, size_t len, uint_t flags);

1277 /*
1278  * Return the allowable DMA burst size for the object mapped by handle.
1279  * The burst sizes will returned in an integer that encodes power
1280  * of two burst sizes that are allowed in bit encoded format. For
1281  * example, a transfer that could allow 1, 2, 4, 8 and 32 byte bursts
1282  * would be encoded as 0x2f. A transfer that could be allowed as solely
1283  * a halfword (2 byte) transfers would be returned as 0x2.
1284  */

1286 int
1287 ddi_dma_burstsizes(ddi_dma_handle_t handle);

1289 /*
1290  * Merge DMA attributes
1291  */

1293 void
1294 ddi_dma_attr_merge(ddi_dma_attr_t *attr, ddi_dma_attr_t *mod);

1296 /*
1297  * Allocate a DMA handle
1298  */

1300 int
1301 ddi_dma_alloc_handle(dev_info_t *dip, ddi_dma_attr_t *attr,
1302                     int (*waitfp)(caddr_t), caddr_t arg,
1303                     ddi_dma_handle_t *handlep);

1305 /*
1306  * Free DMA handle
1307  */

```

```

1309 void
1310 ddi_dma_free_handle(ddi_dma_handle_t *handlep);

1312 /*
1313  * Allocate memory for DMA transfers
1314  */

1316 int
1317 ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length,
1318                 ddi_device_acc_attr_t *accattrp, uint_t xfermodes,
1319                 int (*waitfp)(caddr_t), caddr_t arg, caddr_t *kaddrp,
1320                 size_t *real_length, ddi_acc_handle_t *handlep);

1322 /*
1323  * Free DMA memory
1324  */

1326 void
1327 ddi_dma_mem_free(ddi_acc_handle_t *hp);

1329 /*
1330  * bind address to a DMA handle
1331  */

1333 int
1334 ddi_dma_addr_bind_handle(ddi_dma_handle_t handle, struct as *as,
1335                          caddr_t addr, size_t len, uint_t flags,
1336                          int (*waitfp)(caddr_t), caddr_t arg,
1337                          ddi_dma_cookie_t *cookiep, uint_t *ccountp);

1339 /*
1340  * bind buffer to DMA handle
1341  */

1343 int
1344 ddi_dma_buf_bind_handle(ddi_dma_handle_t handle, struct buf *bp,
1345                          uint_t flags, int (*waitfp)(caddr_t), caddr_t arg,
1346                          ddi_dma_cookie_t *cookiep, uint_t *ccountp);

1348 /*
1349  * unbind mapping object to handle
1350  */

1352 int
1353 ddi_dma_unbind_handle(ddi_dma_handle_t handle);

1355 /*
1356  * get next DMA cookie
1357  */

1359 void
1360 ddi_dma_nextcookie(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep);

1362 /*
1363  * get number of DMA windows
1364  */

1366 int
1367 ddi_dma_numwin(ddi_dma_handle_t handle, uint_t *nwinp);

1369 /*
1370  * get specific DMA window
1371  */

1373 int
1374 ddi_dma_getwin(ddi_dma_handle_t handle, uint_t win, off_t *offp,

```

```

1375         size_t *lenp, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

1377 /*
1378  * activate 64 bit SBus support
1379  */

1381 int
1382 ddi_dma_set_sbus64(ddi_dma_handle_t handle, ulong_t burstsizes);

1384 /*
1385  * Miscellaneous functions
1386  */

1388 /*
1389  * ddi_report_dev:      Report a successful attach.
1390  */

1392 void
1393 ddi_report_dev(dev_info_t *dev);

1395 /*
1396  * ddi_dev_regsizes
1397  *
1398  * If the device has h/w register(s), report
1399  * the size, in bytes, of the specified one into *resultp.
1400  *
1401  * Returns DDI_FAILURE if there are not registers,
1402  * or the specified register doesn't exist.
1403  */

1405 int
1406 ddi_dev_regsizes(dev_info_t *dev, uint_t rnumber, off_t *resultp);

1408 /*
1409  * ddi_dev_nregs
1410  *
1411  * If the device has h/w register(s), report
1412  * how many of them that there are into resultp.
1413  * Return DDI_FAILURE if the device has no registers.
1414  */

1416 int
1417 ddi_dev_nregs(dev_info_t *dev, int *resultp);

1419 /*
1420  * ddi_dev_is_sid
1421  *
1422  * If the device is self-identifying, i.e.,
1423  * has already been probed by a smart PROM
1424  * (and thus registers are known to be valid)
1425  * return DDI_SUCCESS, else DDI_FAILURE.
1426  */

1429 int
1430 ddi_dev_is_sid(dev_info_t *dev);

1432 /*
1433  * ddi_slaveonly
1434  *
1435  * If the device is on a bus that precludes
1436  * the device from being either a dma master or
1437  * a dma slave, return DDI_SUCCESS.
1438  */

1440 int

```



```

1441 ddi_slaveonly(dev_info_t *);

1444 /*
1445 * ddi_dev_affinity
1446 *
1447 *     Report, via DDI_SUCCESS, whether there exists
1448 *     an 'affinity' between two dev_info_t's. An
1449 *     affinity is defined to be either a parent-child,
1450 *     or a sibling relationship such that the siblings
1451 *     or in the same part of the bus they happen to be
1452 *     on.
1453 */

1455 int
1456 ddi_dev_affinity(dev_info_t *deva, dev_info_t *devb);

1459 /*
1460 * ddi_set_callback
1461 *
1462 *     Set a function/arg pair into the callback list identified
1463 *     by listid. *listid must always initially start out as zero.
1464 */

1466 void
1467 ddi_set_callback(int (*funcp)(caddr_t), caddr_t arg, uintptr_t *listid);

1469 /*
1470 * ddi_run_callback
1471 *
1472 *     Run the callback list identified by listid.
1473 */

1475 void
1476 ddi_run_callback(uintptr_t *listid);

1478 /*
1479 * More miscellaneous
1480 */

1482 int
1483 nochpoll(dev_t dev, short events, int anyyet, short *reventsp,
1484          struct pollhead **phpp);

1486 dev_info_t *
1487 nodevinfo(dev_t dev, int otyp);

1489 int
1490 ddi_no_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result);

1492 int
1493 ddi_getinfo_ltol(dev_info_t *dip, ddi_info_cmd_t infocmd,
1494                void *arg, void **result);

1496 int
1497 ddifail(dev_info_t *devi, ddi_attach_cmd_t cmd);

1499 int
1500 ddi_no_dma_map(dev_info_t *dip, dev_info_t *rdip,
1501               struct ddi_dma_req *dmareqp, ddi_dma_handle_t *handlep);

1503 int
1504 ddi_no_dma_allochdl(dev_info_t *dip, dev_info_t *rdip, ddi_dma_attr_t *attr,
1505                    int (*waitfp)(caddr_t), caddr_t arg, ddi_dma_handle_t *handlep);

```

```

1507 int
1508 ddi_no_dma_freehdl(dev_info_t *dip, dev_info_t *rdip,
1509                  ddi_dma_handle_t handle);

1511 int
1512 ddi_no_dma_bindhdl(dev_info_t *dip, dev_info_t *rdip,
1513                   ddi_dma_handle_t handle, struct ddi_dma_req *dmareq,
1514                   ddi_dma_cookie_t *cp, uint_t *ccountp);

1516 int
1517 ddi_no_dma_unbindhdl(dev_info_t *dip, dev_info_t *rdip,
1518                     ddi_dma_handle_t handle);

1520 int
1521 ddi_no_dma_flush(dev_info_t *dip, dev_info_t *rdip,
1522                 ddi_dma_handle_t handle, off_t off, size_t len,
1523                 uint_t cache_flags);

1525 int
1526 ddi_no_dma_win(dev_info_t *dip, dev_info_t *rdip,
1527               ddi_dma_handle_t handle, uint_t win, off_t *offp,
1528               size_t *lenp, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

1530 int
1531 ddi_no_dma_mctl(register dev_info_t *dip, dev_info_t *rdip,
1532                ddi_dma_handle_t handle, enum ddi_dma_ctlops request,
1533                off_t *offp, size_t *lenp, caddr_t *objp, uint_t flags);

1535 void
1536 ddivoid();

1538 cred_t *
1539 ddi_get_cred(void);

1541 time_t
1542 ddi_get_time(void);

1544 pid_t
1545 ddi_get_pid(void);

1547 kt_did_t
1548 ddi_get_kt_did(void);

1550 boolean_t
1551 ddi_can_receive_sig(void);

1553 void
1554 swab(void *src, void *dst, size_t nbytes);

1556 int
1557 ddi_create_minor_node(dev_info_t *dip, char *name, int spec_type,
1558                      minor_t minor_num, char *node_type, int flag);

1560 int
1561 ddi_create_priv_minor_node(dev_info_t *dip, char *name, int spec_type,
1562                            minor_t minor_num, char *node_type, int flag,
1563                            const char *rdpriv, const char *wrpriv, mode_t priv_mode);

1565 void
1566 ddi_remove_minor_node(dev_info_t *dip, char *name);

1568 int
1569 ddi_in_panic(void);

1571 int
1572 ddi_streams_driver(dev_info_t *dip);

```

```

1574 /*
1575  * DDI wrappers for ffs and fls
1576  */
1577 int
1578 ddi_ffs(long mask);

1580 int
1581 ddi_fls(long mask);

1583 /*
1584  * The ddi_soft_state* routines comprise generic storage management utilities
1585  * for driver soft state structures. Two types of soft_state indexes are
1586  * supported: 'integer index', and 'string index'.
1587  */
1588 typedef struct __ddi_soft_state_bystr ddi_soft_state_bystr;

1590 /*
1591  * Initialize a soft_state set, establishing the 'size' of soft state objects
1592  * in the set.
1593  */
1594 * For an 'integer indexed' soft_state set, the initial set will accommodate
1595 * 'n_items' objects - 'n_items' is a hint (i.e. zero is allowed), allocations
1596 * that exceed 'n_items' have additional overhead.
1597 *
1598 * For a 'string indexed' soft_state set, 'n_items' should be the typical
1599 * number of soft state objects in the set - 'n_items' is a hint, there may
1600 * be additional overhead if the hint is too small (and wasted memory if the
1601 * hint is too big).
1602 */
1603 int
1604 ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
1605 int
1606 ddi_soft_state_bystr_init(ddi_soft_state_bystr **state_p,
1607     size_t size, int n_items);

1609 /*
1610  * Allocate a soft state object associated with either 'integer index' or
1611  * 'string index' from a soft_state set.
1612  */
1613 int
1614 ddi_soft_state_zalloc(void *state, int item);
1615 int
1616 ddi_soft_state_bystr_zalloc(ddi_soft_state_bystr *state, const char *str);

1618 /*
1619  * Get the pointer to the allocated soft state object associated with
1620  * either 'integer index' or 'string index'.
1621  */
1622 void *
1623 ddi_get_soft_state(void *state, int item);
1624 void *
1625 ddi_soft_state_bystr_get(ddi_soft_state_bystr *state, const char *str);

1627 /*
1628  * Free the soft state object associated with either 'integer index'
1629  * or 'string index'.
1630  */
1631 void
1632 ddi_soft_state_free(void *state, int item);
1633 void
1634 ddi_soft_state_bystr_free(ddi_soft_state_bystr *state, const char *str);

1636 /*
1637  * Free the soft state set and any associated soft state objects.
1638  */

```

```

1639 void
1640 ddi_soft_state_fini(void **state_p);
1641 void
1642 ddi_soft_state_bystr_fini(ddi_soft_state_bystr **state_p);

1644 /*
1645  * The ddi_strid* routines provide string-to-index management utilities.
1646  */
1647 typedef struct __ddi_strid ddi_strid;
1648 int
1649 ddi_strid_init(ddi_strid **strid_p, int n_items);
1650 id_t
1651 ddi_strid_alloc(ddi_strid *strid, char *str);
1652 id_t
1653 ddi_strid_str2id(ddi_strid *strid, char *str);
1654 char *
1655 ddi_strid_id2str(ddi_strid *strid, id_t id);
1656 void
1657 ddi_strid_free(ddi_strid *strid, id_t id);
1658 void
1659 ddi_strid_fini(ddi_strid **strid_p);

1661 /*
1662  * Set the addr field of the name in dip to name
1663  */
1664 void
1665 ddi_set_name_addr(dev_info_t *dip, char *name);

1667 /*
1668  * Get the address part of the name.
1669  */
1670 char *
1671 ddi_get_name_addr(dev_info_t *dip);

1673 void
1674 ddi_set_parent_data(dev_info_t *dip, void *pd);

1676 void *
1677 ddi_get_parent_data(dev_info_t *dip);

1679 int
1680 ddi_initchild(dev_info_t *parent, dev_info_t *proto);

1682 int
1683 ddi_unitchild(dev_info_t *dip);

1685 major_t
1686 ddi_name_to_major(char *name);

1688 char *
1689 ddi_major_to_name(major_t major);

1691 char *
1692 ddi_deviname(dev_info_t *dip, char *name);

1694 char *
1695 ddi_pathname(dev_info_t *dip, char *path);

1697 char *
1698 ddi_pathname_minor(struct ddi_minor_data *dmdp, char *path);

1700 char *
1701 ddi_pathname_obp(dev_info_t *dip, char *path);

1703 int
1704 ddi_pathname_obp_set(dev_info_t *dip, char *component);

```



```

1837 extern ddi_devstate_t ddi_get_devstate(dev_info_t *);

1839 /*
1840  * Miscellaneous redefines
1841  */
1842 #define uiophysio      physio

1844 /*
1845  * utilities - "reg" mapping and all common portable data access functions
1846  */

1848 /*
1849  * error code from ddi_regs_map_setup
1850  */

1852 #define DDI_REGS_ACC_CONFLICT    (-10)

1854 /*
1855  * Device address advance flags
1856  */

1858 #define DDI_DEV_NO_AUTOINCR      0x0000
1859 #define DDI_DEV_AUTOINCR        0x0001

1861 int
1862 ddi_regs_map_setup(dev_info_t *dip, uint_t rnumber, caddr_t *addrp,
1863                  offset_t offset, offset_t len, ddi_device_acc_attr_t *accattrp,
1864                  ddi_acc_handle_t *handle);

1866 void
1867 ddi_regs_map_free(ddi_acc_handle_t *handle);

1869 /*
1870  * these are the prototypes for the common portable data access functions
1871  */

1873 uint8_t
1874 ddi_get8(ddi_acc_handle_t handle, uint8_t *addr);

1876 uint16_t
1877 ddi_get16(ddi_acc_handle_t handle, uint16_t *addr);

1879 uint32_t
1880 ddi_get32(ddi_acc_handle_t handle, uint32_t *addr);

1882 uint64_t
1883 ddi_get64(ddi_acc_handle_t handle, uint64_t *addr);

1885 void
1886 ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr,
1887             size_t reccount, uint_t flags);

1889 void
1890 ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr,
1891              size_t reccount, uint_t flags);

1893 void
1894 ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr,
1895              size_t reccount, uint_t flags);

1897 void
1898 ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr,
1899              size_t reccount, uint_t flags);

1901 void
1902 ddi_put8(ddi_acc_handle_t handle, uint8_t *addr, uint8_t value);

```

```

1904 void
1905 ddi_put16(ddi_acc_handle_t handle, uint16_t *addr, uint16_t value);

1907 void
1908 ddi_put32(ddi_acc_handle_t handle, uint32_t *addr, uint32_t value);

1910 void
1911 ddi_put64(ddi_acc_handle_t handle, uint64_t *addr, uint64_t value);

1913 void
1914 ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr,
1915             size_t reccount, uint_t flags);
1916 void
1917 ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr,
1918              size_t reccount, uint_t flags);
1919 void
1920 ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr,
1921              size_t reccount, uint_t flags);

1923 void
1924 ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr,
1925              size_t reccount, uint_t flags);

1927 /*
1928  * these are special device handling functions
1929  */
1930 int
1931 ddi_device_zero(ddi_acc_handle_t handle, caddr_t dev_addr,
1932               size_t bytecount, ssize_t dev_advcnt, uint_t dev_datasz);

1934 int
1935 ddi_device_copy(
1936     ddi_acc_handle_t src_handle, caddr_t src_addr, ssize_t src_advcnt,
1937     ddi_acc_handle_t dest_handle, caddr_t dest_addr, ssize_t dest_advcnt,
1938     size_t bytecount, uint_t dev_datasz);

1940 /*
1941  * these are software byte swapping functions
1942  */
1943 uint16_t
1944 ddi_swap16(uint16_t value);

1946 uint32_t
1947 ddi_swap32(uint32_t value);

1949 uint64_t
1950 ddi_swap64(uint64_t value);

1952 /*
1953  * these are the prototypes for PCI local bus functions
1954  */
1955 /*
1956  * PCI power management capabilities reporting in addition to those
1957  * provided by the PCI Power Management Specification.
1958  */
1959 #define PCI_PM_IDLESPPEED      0x1          /* clock for idle dev - cap */
1960 #define PCI_PM_IDLESPPEED_ANY (void *)-1  /* any clock for idle dev */
1961 #define PCI_PM_IDLESPPEED_NONE (void *)-2  /* regular clock for idle dev */

1963 int
1964 pci_config_setup(dev_info_t *dip, ddi_acc_handle_t *handle);

1966 void
1967 pci_config_teardown(ddi_acc_handle_t *handle);

```

```

1969 uint8_t
1970 pci_config_get8(ddi_acc_handle_t handle, off_t offset);

1972 uint16_t
1973 pci_config_get16(ddi_acc_handle_t handle, off_t offset);

1975 uint32_t
1976 pci_config_get32(ddi_acc_handle_t handle, off_t offset);

1978 uint64_t
1979 pci_config_get64(ddi_acc_handle_t handle, off_t offset);

1981 void
1982 pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);

1984 void
1985 pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);

1987 void
1988 pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);

1990 void
1991 pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

1993 int
1994 pci_report_pmcaps(dev_info_t *dip, int cap, void *arg);

1996 int
1997 pci_restore_config_regs(dev_info_t *dip);

1999 int
2000 pci_save_config_regs(dev_info_t *dip);

2002 void
2003 pci_ereport_setup(dev_info_t *dip);

2005 void
2006 pci_ereport_teardown(dev_info_t *dip);

2008 void
2009 pci_ereport_post(dev_info_t *dip, ddi_fm_error_t *derr, uint16_t *status);

2011 #if defined(__i386) || defined(__amd64)
2012 int
2013 pci_peekpoke_check(dev_info_t *, dev_info_t *, ddi_ctl_enum_t, void *, void *,
2014 int (*handler)(dev_info_t *, dev_info_t *, ddi_ctl_enum_t, void *,
2015 void *), kmutex_t *, kmutex_t *,
2016 void (*scan)(dev_info_t *, ddi_fm_error_t *));
2017 #endif

2019 void
2020 pci_target_enqueue(uint64_t, char *, char *, uint64_t);

2022 void
2023 pci_targetq_init(void);

2025 int
2026 pci_post_suspend(dev_info_t *dip);

2028 int
2029 pci_pre_resume(dev_info_t *dip);

2031 /*
2032  * the prototype for the C Language Type Model inquiry.
2033  */
2034 model_t ddi_mmap_get_model(void);

```

```

2035 model_t ddi_model_convert_from(model_t);

2037 /*
2038  * these are the prototypes for device id functions.
2039  */
2040 int
2041 ddi_devid_valid(ddi_devid_t devid);

2043 int
2044 ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);

2046 void
2047 ddi_devid_unregister(dev_info_t *dip);

2049 int
2050 ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t nbytes,
2051 void *id, ddi_devid_t *ret_devid);

2053 int
2054 ddi_devid_get(dev_info_t *dip, ddi_devid_t *ret_devid);

2056 size_t
2057 ddi_devid_sizeof(ddi_devid_t devid);

2059 void
2060 ddi_devid_free(ddi_devid_t devid);

2062 int
2063 ddi_devid_compare(ddi_devid_t id1, ddi_devid_t id2);

2065 int
2066 ddi_devid_scsi_encode(int version, char *driver_name,
2067 uchar_t *inq, size_t inq_len, uchar_t *inq80, size_t inq80_len,
2068 uchar_t *inq83, size_t inq83_len, ddi_devid_t *ret_devid);

2070 int
2071 ddi_devid_smp_encode(int version, char *driver_name,
2072 char *wwnstr, uchar_t *srmir_buf, size_t srmir_len,
2073 ddi_devid_t *ret_devid);

2075 char
2076 *ddi_devid_to_guid(ddi_devid_t devid);

2078 void
2079 ddi_devid_free_guid(char *guid);

2081 int
2082 ddi_lyr_get_devid(dev_t dev, ddi_devid_t *ret_devid);

2084 int
2085 ddi_lyr_get_minor_name(dev_t dev, int spec_type, char **minor_name);

2087 int
2088 ddi_lyr_devid_to_devlist(ddi_devid_t devid, char *minor_name, int *retndevs,
2089 dev_t **retdevs);

2091 void
2092 ddi_lyr_free_devlist(dev_t *devlist, int ndevs);

2094 char *
2095 ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);

2097 int
2098 ddi_devid_str_decode(char *devidstr, ddi_devid_t *devidp, char **minor_namep);

2100 void

```

```

2101 ddi_devid_str_free(char *devidstr);

2103 int
2104 ddi_devid_str_compare(char *id1_str, char *id2_str);

2106 /*
2107  * Event to post to when a devinfo node is removed.
2108  */
2109 #define DDI_DEVI_REMOVE_EVENT          "DDI:DEVI_REMOVE"
2110 #define DDI_DEVI_INSERT_EVENT          "DDI:DEVI_INSERT"
2111 #define DDI_DEVI_BUS_RESET_EVENT       "DDI:DEVI_BUS_RESET"
2112 #define DDI_DEVI_DEVICE_RESET_EVENT    "DDI:DEVI_DEVICE_RESET"

2114 /*
2115  * Invoke bus nexus driver's implementation of the
2116  * (*bus_remove_eventcall)() interface to remove a registered
2117  * callback handler for "event".
2118  */
2119 int
2120 ddi_remove_event_handler(ddi_callback_id_t id);

2122 /*
2123  * Invoke bus nexus driver's implementation of the
2124  * (*bus_add_eventcall)() interface to register a callback handler
2125  * for "event".
2126  */
2127 int
2128 ddi_add_event_handler(dev_info_t *dip, ddi_eventcookie_t event,
2129                      void (*handler)(dev_info_t *, ddi_eventcookie_t, void *, void *),
2130                      void *arg, ddi_callback_id_t *id);

2132 /*
2133  * Return a handle for event "name" by calling up the device tree
2134  * hierarchy via (*bus_get_eventcookie)() interface until claimed
2135  * by a bus nexus or top of dev_info tree is reached.
2136  */
2137 int
2138 ddi_get_eventcookie(dev_info_t *dip, char *name,
2139                    ddi_eventcookie_t *event_cookiep);

2141 /*
2142  * log a system event
2143  */
2144 int
2145 ddi_log_sysevent(dev_info_t *dip, char *vendor, char *class_name,
2146                 char *subclass_name, nvlist_t *attr_list, sysevent_id_t *eidp,
2147                 int sleep_flag);

2149 /*
2150  * ddi_log_sysevent() vendors
2151  */
2152 #define DDI_VENDOR_SUNW          "SUNW"

2154 /*
2155  * Opaque task queue handle.
2156  */
2157 typedef struct ddi_taskq ddi_taskq_t;

2159 /*
2160  * Use default system priority.
2161  */
2162 #define TASKQ_DEFAULTPRI -1

2164 /*
2165  * Create a task queue
2166  */

```

```

2167 ddi_taskq_t *ddi_taskq_create(dev_info_t *dip, const char *name,
2168                               int nthreads, pri_t pri, uint_t cflags);

2170 /*
2171  * destroy a task queue
2172  */
2173 void ddi_taskq_destroy(ddi_taskq_t *tq);

2175 /*
2176  * Dispatch a task to a task queue
2177  */
2178 int ddi_taskq_dispatch(ddi_taskq_t *tq, void (* func)(void *),
2179                       void *arg, uint_t dflags);

2181 /*
2182  * Wait for all previously scheduled tasks to complete.
2183  */
2184 void ddi_taskq_wait(ddi_taskq_t *tq);

2186 /*
2187  * Suspend all task execution.
2188  */
2189 void ddi_taskq_suspend(ddi_taskq_t *tq);

2191 /*
2192  * Resume task execution.
2193  */
2194 void ddi_taskq_resume(ddi_taskq_t *tq);

2196 /*
2197  * Is task queue suspended?
2198  */
2199 boolean_t ddi_taskq_suspended(ddi_taskq_t *tq);

2201 /*
2202  * Parse an interface name of the form <alphanumeric>##<numeric> where
2203  * <numeric> is maximal.
2204  */
2205 int ddi_parse(const char *, char *, uint_t *);

2207 /*
2208  * DDI interrupt priority level
2209  */
2210 #define DDI_IPL_0          (0)    /* kernel context */
2211 #define DDI_IPL_1          (1)    /* interrupt priority level 1 */
2212 #define DDI_IPL_2          (2)    /* interrupt priority level 2 */
2213 #define DDI_IPL_3          (3)    /* interrupt priority level 3 */
2214 #define DDI_IPL_4          (4)    /* interrupt priority level 4 */
2215 #define DDI_IPL_5          (5)    /* interrupt priority level 5 */
2216 #define DDI_IPL_6          (6)    /* interrupt priority level 6 */
2217 #define DDI_IPL_7          (7)    /* interrupt priority level 7 */
2218 #define DDI_IPL_8          (8)    /* interrupt priority level 8 */
2219 #define DDI_IPL_9          (9)    /* interrupt priority level 9 */
2220 #define DDI_IPL_10         (10)   /* interrupt priority level 10 */

2222 /*
2223  * DDI periodic timeout interface
2224  */
2225 ddi_periodic_t ddi_periodic_add(void (*)(void *), void *, hrtime_t, int);
2226 void ddi_periodic_delete(ddi_periodic_t);

2228 /*
2229  * Default quiesce(9E) implementation for drivers that don't need to do
2230  * anything.
2231  */
2232 int ddi_quiesce_not_needed(dev_info_t *);

```

```
2234 /*
2235  * Default quiesce(9E) initialization function for drivers that should
2236  * implement quiesce but haven't yet.
2237  */
2238 int ddi_quiesce_not_supported(dev_info_t *);

2240 /*
2241  * DDI generic callback interface
2242  */

2244 typedef struct __ddi_cb **ddi_cb_handle_t;

2246 int      ddi_cb_register(dev_info_t *dip, ddi_cb_flags_t flags,
2247                        ddi_cb_func_t cbfunc, void *arg1, void *arg2,
2248                        ddi_cb_handle_t *ret_hdlp);
2249 int      ddi_cb_unregister(ddi_cb_handle_t hdl);

2251 /* Notify DDI of memory added */
2252 void ddi_mem_update(uint64_t addr, uint64_t size);

2254 /* Path alias interfaces */
2255 typedef struct plat_alias {
2256     char *pali_current;
2257     uint64_t pali_naliases;
2258     char **pali_aliases;
2259 } plat_alias_t;

2261 typedef struct alias_pair {
2262     char *pair_alias;
2263     char *pair_curr;
2264 } alias_pair_t;

2266 extern boolean_t ddi_aliases_present;

2268 typedef struct ddi_alias {
2269     alias_pair_t      *dali_alias_pairs;
2270     alias_pair_t      *dali_curr_pairs;
2271     int               dali_num_pairs;
2272     mod_hash_t        *dali_alias_TLB;
2273     mod_hash_t        *dali_curr_TLB;
2274 } ddi_alias_t;

2276 extern ddi_alias_t ddi_aliases;

2278 void ddi_register_aliases(plat_alias_t *pali, uint64_t npali);
2279 dev_info_t *ddi_alias_redirect(char *alias);
2280 char *ddi_curr_redirect(char *curr);

2282 #endif /* _KERNEL */

2284 #ifdef __cplusplus
2285 }
2286 #endif

2288 #endif /* _SYS_SUNDDI_H */
```