```
**********************************************************
   77917 Wed Oct 17 21:48:36 2012
new/usr/src/cmd/truss/codes.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
______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)TIOCGPGRP,    "TIOCGPGRP",    NULL },
 361         { (uint_t)TIOCSPGRP,    "TIOCSPGRP",    NULL },
 362         { (uint_t)TIOCGSID,     "TIOCGSID",     NULL },
 363         { (uint_t)TIOCSTI,      "TIOCSTI",      NULL },
 364         { (uint_t)TIOCMSET,     "TIOCMSET",     NULL },
 365         { (uint_t)TIOCMBIS,     "TIOCMBIS",     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)TIOCOUTQ,     "TIOCOUTQ",     NULL },
 375         { (uint_t)TIOCGLTC,     "TIOCGLTC",     NULL },
 376         { (uint_t)TIOCSLTC,     "TIOCSLTC",     NULL },
 377         { (uint_t)TIOCCDTR,     "TIOCCDTR",     NULL },
 378         { (uint_t)TIOCSDTR,     "TIOCSDTR",     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,   "TIOCSILOOP",   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)DLIOCLOWLINK, "DLIOCLOWLINK", 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)STTHROW,      "STTHROW",      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_SRDOPT,     "I_SRDOPT",     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",
541           NULL },
542         { (uint_t)AUDIO_MIXER_SINGLE_OPEN,      "AUDIO_MIXER_SINGLE_OPEN",
543           NULL },
544         { (uint_t)AUDIO_MIXER_GET_SAMPLE_RATES, "AUDIO_MIXER_GET_SAMPLE_RATES",
545           NULL },
546         { (uint_t)AUDIO_MIXERCTL_GETINFO,       "AUDIO_MIXERCTL_GETINFO",
547           NULL },
548         { (uint_t)AUDIO_MIXERCTL_SETINFO,       "AUDIO_MIXERCTL_SETINFO",
549           NULL },
550         { (uint_t)AUDIO_MIXERCTL_GET_CHINFO,    "AUDIO_MIXERCTL_GET_CHINFO",
551           NULL },
552         { (uint_t)AUDIO_MIXERCTL_SET_CHINFO,    "AUDIO_MIXERCTL_SET_CHINFO",
553           NULL },
554         { (uint_t)AUDIO_MIXERCTL_GET_MODE,      "AUDIO_MIXERCTL_GET_MODE",
555           NULL },
556         { (uint_t)AUDIO_MIXERCTL_SET_MODE,      "AUDIO_MIXERCTL_SET_MODE",
557           NULL },
558         /* new style Boomer (OSS) ioctls */
559         { (uint_t)SNDCTL_SYSINFO,       "SNDCTL_SYSINFO",       NULL },
560         { (uint_t)SNDCTL_AUDIOINFO,     "SNDCTL_AUDIOINFO",     NULL },
561         { (uint_t)SNDCTL_AUDIOINFO_EX,  "SNDCTL_AUDIOINFO_EX",  NULL },
562         { (uint_t)SNDCTL_MIXERINFO,     "SNDCTL_MIXERINFO",     NULL },
563         { (uint_t)SNDCTL_CARDINFO,      "SNDCTL_CARDINFO",      NULL },
564         { (uint_t)SNDCTL_ENGINEINFO,    "SNDCTL_ENGINEINFO",    NULL },
565         { (uint_t)SNDCTL_MIX_NRMIX,     "SNDCTL_MIX_NRMIX",     NULL },
566         { (uint_t)SNDCTL_MIX_NREXT,     "SNDCTL_MIX_NREXT",     NULL },
567         { (uint_t)SNDCTL_MIX_EXTINFO,   "SNDCTL_MIX_EXTINFO",   NULL },
568         { (uint_t)SNDCTL_MIX_READ,      "SNDCTL_MIX_READ",      NULL },
569         { (uint_t)SNDCTL_MIX_WRITE,     "SNDCTL_MIX_WRITE",     NULL },
570         { (uint_t)SNDCTL_MIX_ENUMINFO,  "SNDCTL_MIX_ENUMINFO",  NULL },
571         { (uint_t)SNDCTL_MIX_DESCRIPTION,       "SNDCTL_MIX_DESCRIPTION",
572           NULL },
573         { (uint_t)SNDCTL_SETSONG,       "SNDCTL_SETSONG",       NULL },
574         { (uint_t)SNDCTL_GETSONG,       "SNDCTL_GETSONG",       NULL },
575         { (uint_t)SNDCTL_SETNAME,       "SNDCTL_SETNAME",       NULL },
576         { (uint_t)SNDCTL_SETLABEL,      "SNDCTL_SETLABEL",      NULL },
577         { (uint_t)SNDCTL_GETLABEL,      "SNDCTL_GETLABEL",      NULL },
578         { (uint_t)SNDCTL_DSP_HALT,      "SNDCTL_DSP_HALT",      NULL },
579         { (uint_t)SNDCTL_DSP_RESET,     "SNDCTL_DSP_RESET",     NULL },
580         { (uint_t)SNDCTL_DSP_SYNC,      "SNDCTL_DSP_SYNC",      NULL },
581         { (uint_t)SNDCTL_DSP_SPEED,     "SNDCTL_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_GETOPTR,   "SNDCTL_DSP_GETOPTR",   NULL },
603         { (uint_t)SNDCTL_DSP_SETSYNCRO, "SNDCTL_DSP_SETSYNCRO", 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_RECMASK,     "SOUND_MIXER_READ_RECMASK",
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 },

696         /* STREAMS redirection ioctls */
697         { (uint_t)SRIOCSREDIR,          "SRIOCSREDIR",  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),     "PIOCGTRACE",   NULL },
712         { (uint_t)(PIOC|6),     "PIOCSTRACE",   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),    "PIOCSHOLD",    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)SIOCSPGRP,    "SIOCSPGRP",    NULL },
768         { (uint_t)SIOCGPGRP,    "SIOCGPGRP",    NULL },
769         { (uint_t)SIOCADDRT,    "SIOCADDRT",    "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)SIOCDARP,     "SIOCDARP",     "arpreq" },
794         { (uint_t)SIOCUPPER,    "SIOCUPPER",    "ifreq" },
795         { (uint_t)SIOCLOWER,    "SIOCLOWER",    "ifreq" },
796         { (uint_t)SIOCSETSYNC,  "SIOCSETSYNC",  "ifreq" },
797         { (uint_t)SIOCGETSYNC,  "SIOCGETSYNC",  "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)SIOCGLIFCONF,         "SIOCGLIFCONF",         "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)SIOCGLIFNETMASK,      "SIOCGLIFNETMASK",      "lifreq" },
835         { (uint_t)SIOCSLIFNETMASK,      "SIOCSLIFNETMASK",      "lifreq" },
836         { (uint_t)SIOCGLIFMETRIC,       "SIOCGLIFMETRIC",       "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)SIOCGLIFSUBNET,       "SIOCGLIFSUBNET",       "lifreq" },
848         { (uint_t)SIOCSLIFLNKINFO,      "SIOCSLIFLNKINFO",      "lifreq" },
849         { (uint_t)SIOCGLIFLNKINFO,      "SIOCGLIFLNKINFO",      "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)SIOCGLIFGROUPNAME,    "SIOCGLIFGROUPNAME",    "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)SIOCGLIFZONE,         "SIOCGLIFZONE",         "lifreq" },
871         { (uint_t)SIOCSLIFZONE,         "SIOCSLIFZONE",         "lifreq" },
872         { (uint_t)SIOCSCTPSOPT,         "SIOCSCTPSOPT",         NULL },
873         { (uint_t)SIOCSCTPGOPT,         "SIOCSCTPGOPT",         NULL },
874         { (uint_t)SIOCSCTPPEELOFF,      "SIOPCSCTPPEELOFF",     "int" },
875         { (uint_t)SIOCGLIFUSESRC,       "SIOCGLIFUSESRC",       "lifreq" },
876         { (uint_t)SIOCSLIFUSESRC,       "SIOCSLIFUSESRC",       "lifreq" },
877         { (uint_t)SIOCGLIFSRCOF,        "SIOCGLIFSRCOF",        "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)SIOCGLIFDADSTATE,     "SIOCGLIFDADSTATE",  "lifreq" },
883         { (uint_t)SIOCSLIFPREFIX,       "SIOCSLIFPREFIX", "lifreq" },
884         { (uint_t)SIOCGSTAMP,           "SIOCGSTAMP",           "timeval" },
885         { (uint_t)SIOCGIFHWADDR,        "SIOCGIFHWADDR",        "ifreq" },
886         { (uint_t)SIOCGLIFHWADDR,       "SIOCGLIFHWADDR",       "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,
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_AGGDESC,    "DTRACEIOC_AGGDESC",    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)KIOCGETKEY,           "KIOCGETKEY",   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)KIOCGRPTDELAY,        "KIOCGRPTDELAY",        NULL },
1067         { (uint_t)KIOCSRPTDELAY,        "KIOCSRPTDELAY",        NULL },
1068         { (uint_t)KIOCGRPTRATE,         "KIOCGRPTRATE", NULL },
1069         { (uint_t)KIOCSRPTRATE,         "KIOCSRPTRATE", 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",
1085             "laioc_add_rem"},
1086         { (uint_t)LAIOC_REMOVE,         "LAIOC_REMOVE",
1087             "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",
1093                 "dld_ioc_phys_attr"},
1094         { (uint_t)DLDIOC_DOORSERVER,    "DLDIOC_DOORSERVER", "dld_ioc_door"},
1095         { (uint_t)DLDIOC_RENAME,        "DLDIOC_RENAME", "dld_ioc_rename"},
1096         { (uint_t)DLDIOC_SECOBJ_GET,            "DLDIOC_SECOBJ_GET",
1097                 "dld_ioc_secobj_get"},
1098         { (uint_t)DLDIOC_SECOBJ_SET,            "DLDIOC_SECOBJ_SET",
1099                 "dld_ioc_secobj_set"},
1100         { (uint_t)DLDIOC_SECOBJ_UNSET,          "DLDIOC_SECOBJ_UNSET",
1101                 "dld_ioc_secobj_unset"},
1102         { (uint_t)DLDIOC_MACADDRGET,            "DLDIOC_MACADDRGET",
1103                 "dld_ioc_macaddrget"},
1104         { (uint_t)DLDIOC_SETMACPROP,            "DLDIOC_SETMACPROP",
1105                 "dld_ioc_macprop_s"},
1106         { (uint_t)DLDIOC_GETMACPROP,            "DLDIOC_GETMACPROP",
1107                 "dld_ioc_macprop_s"},
1108         { (uint_t)DLDIOC_ADDFLOW,               "DLDIOC_ADDFLOW",
1109                 "dld_ioc_addflow"},
1110         { (uint_t)DLDIOC_REMOVEFLOW,            "DLDIOC_REMOVEFLOW",
1111                 "dld_ioc_removeflow"},
1112         { (uint_t)DLDIOC_MODIFYFLOW,            "DLDIOC_MODIFYFLOW",
1113                 "dld_ioc_modifyflow"},
```

```
1114         { (uint_t)DLDIOC_WALKFLOW,             "DLDIOC_WALKFLOW",
1115                 "dld_ioc_walkflow"},
1116         { (uint_t)DLDIOC_USAGELOG,             "DLDIOC_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_FITS_SEND,           "ZFS_IOC_FITS_SEND",
1197                 "zfs_cmd_t" },
1198 #endif /* ! codereview */
1199         { (uint_t)ZFS_IOC_INJECT_FAULT,        "ZFS_IOC_INJECT_FAULT",
1200                 "zfs_cmd_t" },
1201         { (uint_t)ZFS_IOC_CLEAR_FAULT,         "ZFS_IOC_CLEAR_FAULT",
1202                 "zfs_cmd_t" },
1203         { (uint_t)ZFS_IOC_INJECT_LIST_NEXT,    "ZFS_IOC_INJECT_LIST_NEXT",
1204                 "zfs_cmd_t" },
1205         { (uint_t)ZFS_IOC_ERROR_LOG,           "ZFS_IOC_ERROR_LOG",
1206                 "zfs_cmd_t" },
1207         { (uint_t)ZFS_IOC_CLEAR,               "ZFS_IOC_CLEAR",
1208                 "zfs_cmd_t" },
1209         { (uint_t)ZFS_IOC_PROMOTE,             "ZFS_IOC_PROMOTE",
1210                 "zfs_cmd_t" },
1211         { (uint_t)ZFS_IOC_SNAPSHOT,            "ZFS_IOC_SNAPSHOT",
1212                 "zfs_cmd_t" },
1213         { (uint_t)ZFS_IOC_DSOBJ_TO_DSNAME,     "ZFS_IOC_DSOBJ_TO_DSNAME",
1214                 "zfs_cmd_t" },
1215         { (uint_t)ZFS_IOC_OBJ_TO_PATH,         "ZFS_IOC_OBJ_TO_PATH",
1216                 "zfs_cmd_t" },
1217         { (uint_t)ZFS_IOC_POOL_SET_PROPS,      "ZFS_IOC_POOL_SET_PROPS",
1218                 "zfs_cmd_t" },
1219         { (uint_t)ZFS_IOC_POOL_GET_PROPS,      "ZFS_IOC_POOL_GET_PROPS",
1220                 "zfs_cmd_t" },
1221         { (uint_t)ZFS_IOC_SET_FSACL,           "ZFS_IOC_SET_FSACL",
1222                 "zfs_cmd_t" },
1223         { (uint_t)ZFS_IOC_GET_FSACL,           "ZFS_IOC_GET_FSACL",
1224                 "zfs_cmd_t" },
1225         { (uint_t)ZFS_IOC_SHARE,               "ZFS_IOC_SHARE",
1226                 "zfs_cmd_t" },
1227         { (uint_t)ZFS_IOC_INHERIT_PROP,        "ZFS_IOC_INHERIT_PROP",
1228                 "zfs_cmd_t" },
1229         { (uint_t)ZFS_IOC_SMB_ACL,             "ZFS_IOC_SMB_ACL",
1230                 "zfs_cmd_t" },
1231         { (uint_t)ZFS_IOC_USERSPACE_ONE,       "ZFS_IOC_USERSPACE_ONE",
1232                 "zfs_cmd_t" },
1233         { (uint_t)ZFS_IOC_USERSPACE_MANY,      "ZFS_IOC_USERSPACE_MANY",
1234                 "zfs_cmd_t" },
1235         { (uint_t)ZFS_IOC_USERSPACE_UPGRADE,   "ZFS_IOC_USERSPACE_UPGRADE",
1236                 "zfs_cmd_t" },
1237         { (uint_t)ZFS_IOC_HOLD,                "ZFS_IOC_HOLD",
1238                 "zfs_cmd_t" },
1239         { (uint_t)ZFS_IOC_RELEASE,             "ZFS_IOC_RELEASE",
1240                 "zfs_cmd_t" },
1241         { (uint_t)ZFS_IOC_GET_HOLDS,           "ZFS_IOC_GET_HOLDS",
1242                 "zfs_cmd_t" },
1243         { (uint_t)ZFS_IOC_OBJSET_RECVD_PROPS,  "ZFS_IOC_OBJSET_RECVD_PROPS",
1244                 "zfs_cmd_t" },
1245         { (uint_t)ZFS_IOC_VDEV_SPLIT,          "ZFS_IOC_VDEV_SPLIT",
```

```
1246                "zfs_cmd_t" },
1247        { (uint_t)ZFS_IOC_NEXT_OBJ,             "ZFS_IOC_NEXT_OBJ",
1248                "zfs_cmd_t" },
1249        { (uint_t)ZFS_IOC_DIFF,                 "ZFS_IOC_DIFF",
1250                "zfs_cmd_t" },
1251        { (uint_t)ZFS_IOC_TMP_SNAPSHOT,         "ZFS_IOC_TMP_SNAPSHOT",
1252                "zfs_cmd_t" },
1253        { (uint_t)ZFS_IOC_OBJ_TO_STATS,         "ZFS_IOC_OBJ_TO_STATS",
1254                "zfs_cmd_t" },
1255        { (uint_t)ZFS_IOC_SPACE_WRITTEN,        "ZFS_IOC_SPACE_WRITTEN",
1256                "zfs_cmd_t" },
1257        { (uint_t)ZFS_IOC_DESTROY_SNAPS,        "ZFS_IOC_DESTROY_SNAPS",
1258                "zfs_cmd_t" },
1259        { (uint_t)ZFS_IOC_POOL_REGUID,          "ZFS_IOC_POOL_REGUID",
1260                "zfs_cmd_t" },
1261        { (uint_t)ZFS_IOC_POOL_REOPEN,          "ZFS_IOC_POOL_REOPEN",
1262                "zfs_cmd_t" },
1263        { (uint_t)ZFS_IOC_SEND_PROGRESS,        "ZFS_IOC_SEND_PROGRESS",
1264                "zfs_cmd_t" },
1265        { (uint_t)ZFS_IOC_LOG_HISTORY,          "ZFS_IOC_LOG_HISTORY",
1266                "zfs_cmd_t" },
1267        { (uint_t)ZFS_IOC_SEND_NEW,             "ZFS_IOC_SEND_NEW",
1268                "zfs_cmd_t" },
1269        { (uint_t)ZFS_IOC_SEND_SPACE,           "ZFS_IOC_SEND_SPACE",
1270                "zfs_cmd_t" },
1271        { (uint_t)ZFS_IOC_CLONE,                "ZFS_IOC_CLONE",
1272                "zfs_cmd_t" },

1274        /* kssl ioctls */
1275        { (uint_t)KSSL_ADD_ENTRY,               "KSSL_ADD_ENTRY",
1276                "kssl_params_t"},
1277        { (uint_t)KSSL_DELETE_ENTRY,            "KSSL_DELETE_ENTRY",
1278                "sockaddr_in"},

1280        /* disk ioctls - (0x04 << 8) - dkio.h */
1281        { (uint_t)DKIOCGGEOM,           "DKIOCGGEOM",
1282                "struct dk_geom"},
1283        { (uint_t)DKIOCINFO,            "DKIOCINFO",
1284                "struct dk_info"},
1285        { (uint_t)DKIOCEJECT,           "DKIOCEJECT",
1286                NULL},
1287        { (uint_t)DKIOCGVTOC,           "DKIOCGVTOC",
1288                "struct vtoc"},
1289        { (uint_t)DKIOCSVTOC,           "DKIOCSVTOC",
1290                "struct vtoc"},
1291        { (uint_t)DKIOCGEXTVTOC,        "DKIOCGEXTVTOC",
1292                "struct extvtoc"},
1293        { (uint_t)DKIOCSEXTVTOC,        "DKIOCSEXTVTOC",
1294                "struct extvtoc"},
1295        { (uint_t)DKIOCFLUSHWRITECACHE, "DKIOCFLUSHWRITECACHE",
1296                NULL},
1297        { (uint_t)DKIOCGETWCE,          "DKIOCGETWCE",
1298                NULL},
1299        { (uint_t)DKIOCSETWCE,          "DKIOCSETWCE",
1300                NULL},
1301        { (uint_t)DKIOCSGEOM,           "DKIOCSGEOM",
1302                "struct dk_geom"},
1303        { (uint_t)DKIOCSAPART,          "DKIOCSAPART",
1304                "struct dk_allmap"},
1305        { (uint_t)DKIOCGAPART,          "DKIOCGAPART",
1306                "struct dk_allmap"},
1307        { (uint_t)DKIOCG_PHYGEOM,       "DKIOCG_PHYGEOM",
1308                "struct dk_geom"},
1309        { (uint_t)DKIOCG_VIRTGEOM,      "DKIOCG_VIRTGEOM",
1310                "struct dk_geom"},
1311        { (uint_t)DKIOCLOCK,            "DKIOCLOCK",
```



```
1312                NULL},
1313        { (uint_t)DKIOCUNLOCK,          "DKIOCUNLOCK",
1314                NULL},
1315        { (uint_t)DKIOCSTATE,           "DKIOCSTATE",
1316                NULL},
1317        { (uint_t)DKIOCREMOVABLE,       "DKIOCREMOVABLE",
1318                NULL},
1319        { (uint_t)DKIOCHOTPLUGGABLE,    "DKIOCHOTPLUGGABLE",
1320                NULL},
1321        { (uint_t)DKIOCADDBAD,          "DKIOCADDBAD",
1322                NULL},
1323        { (uint_t)DKIOCGETDEF,          "DKIOCGETDEF",
1324                NULL},
1325        { (uint_t)DKIOCPARTINFO,        "DKIOCPARTINFO",
1326                "struct part_info"},
1327        { (uint_t)DKIOCEXTPARTINFO,     "DKIOCEXTPARTINFO",
1328                "struct extpart_info"},
1329        { (uint_t)DKIOCGMEDIAINFO,      "DKIOCGMEDIAINFO",
1330                "struct dk_minfo"},
1331        { (uint_t)DKIOCGMBOOT,          "DKIOCGMBOOT",
1332                NULL},
1333        { (uint_t)DKIOCSMBOOT,          "DKIOCSMBOOT",
1334                NULL},
1335        { (uint_t)DKIOCSETEFI,          "DKIOCSETEFI",
1336                "struct dk_efi"},
1337        { (uint_t)DKIOCGETEFI,          "DKIOCGETEFI",
1338                "struct dk_efi"},
1339        { (uint_t)DKIOCPARTITION,       "DKIOCPARTITION",
1340                "struct partition64"},
1341        { (uint_t)DKIOCGETVOLCAP,       "DKIOCGETVOLCAP",
1342                "struct volcap_t"},
1343        { (uint_t)DKIOCSETVOLCAP,       "DKIOCSETVOLCAP",
1344                "struct volcap_t"},
1345        { (uint_t)DKIOCDMR,             "DKIOCDMR",
1346                "struct vol_directed_rd"},
1347        { (uint_t)DKIOCDUMPINIT,        "DKIOCDUMPINIT",
1348                NULL},
1349        { (uint_t)DKIOCDUMPFINI,        "DKIOCDUMPFINI",
1350                NULL},
1351        { (uint_t)DKIOCREADONLY,        "DKIOCREADONLY",
1352                NULL},

1354        /* disk ioctls - (0x04 << 8) - fdio.h */
1355        { (uint_t)FDIOGCHAR,            "FDIOGCHAR",
1356                "struct fd_char"},
1357        { (uint_t)FDIOSCHAR,            "FDIOSCHAR",
1358                "struct fd_char"},
1359        { (uint_t)FDEJECT,              "FDEJECT",
1360                NULL},
1361        { (uint_t)FDGETCHANGE,          "FDGETCHANGE",
1362                NULL},
1363        { (uint_t)FDGETDRIVECHAR,       "FDGETDRIVECHAR",
1364                "struct fd_drive"},
1365        { (uint_t)FDSETDRIVECHAR,       "FDSETDRIVECHAR",
1366                "struct fd_drive"},
1367        { (uint_t)FDGETSEARCH,          "FDGETSEARCH",
1368                NULL},
1369        { (uint_t)FDSETSEARCH,          "FDSETSEARCH",
1370                NULL},
1371        { (uint_t)FDIOCMD,              "FDIOCMD",
1372                "struct fd_cmd"},
1373        { (uint_t)FDRAW,                "FDRAW",
1374                "struct fd_raw"},
1375        { (uint_t)FDDEFGEOCHAR,         "FDDEFGEOCHAR",
1376                NULL},
```

```
1378        /* disk ioctls - (0x04 << 8) - cdio.h */
1379        { (uint_t)CDROMPAUSE,           "CDROMPAUSE",
1380              NULL},
1381        { (uint_t)CDROMRESUME,          "CDROMRESUME",
1382              NULL},
1383        { (uint_t)CDROMPLAYMSF,         "CDROMPLAYMSF",
1384              "struct cdrom_msf"},
1385        { (uint_t)CDROMPLAYTRKIND,      "CDROMPLAYTRKIND",
1386              "struct cdrom_ti"},
1387        { (uint_t)CDROMREADTOCHDR,      "CDROMREADTOCHDR",
1388              "struct cdrom_tochdr"},
1389        { (uint_t)CDROMREADTOCENTRY,    "CDROMREADTOCENTRY",
1390              "struct cdrom_tocentry"},
1391        { (uint_t)CDROMSTOP,            "CDROMSTOP",
1392              NULL},
1393        { (uint_t)CDROMSTART,           "CDROMSTART",
1394              NULL},
1395        { (uint_t)CDROMEJECT,           "CDROMEJECT",
1396              NULL},
1397        { (uint_t)CDROMVOLCTRL,         "CDROMVOLCTRL",
1398              "struct cdrom_volctrl"},
1399        { (uint_t)CDROMSUBCHNL,         "CDROMSUBCHNL",
1400              "struct cdrom_subchnl"},
1401        { (uint_t)CDROMREADMODE2,       "CDROMREADMODE2",
1402              "struct cdrom_read"},
1403        { (uint_t)CDROMREADMODE1,       "CDROMREADMODE1",
1404              "struct cdrom_read"},
1405        { (uint_t)CDROMREADOFFSET,      "CDROMREADOFFSET",
1406              NULL},
1407        { (uint_t)CDROMGBLKMODE,        "CDROMGBLKMODE",
1408              NULL},
1409        { (uint_t)CDROMSBLKMODE,        "CDROMSBLKMODE",
1410              NULL},
1411        { (uint_t)CDROMCDDA,            "CDROMCDDA",
1412              "struct cdrom_cdda"},
1413        { (uint_t)CDROMCDXA,            "CDROMCDXA",
1414              "struct cdrom_cdxa"},
1415        { (uint_t)CDROMSUBCODE,         "CDROMSUBCODE",
1416              "struct cdrom_subcode"},
1417        { (uint_t)CDROMGDRVSPEED,       "CDROMGDRVSPEED",
1418              NULL},
1419        { (uint_t)CDROMSDRVSPEED,       "CDROMSDRVSPEED",
1420              NULL},
1421        { (uint_t)CDROMCLOSETRAY,       "CDROMCLOSETRAY",
1422              NULL},

1424        /* disk ioctls - (0x04 << 8) - uscsi.h */
1425        { (uint_t)USCSICMD,             "USCSICMD",
1426              "struct uscsi_cmd"},

1428        /* dumpadm ioctls - (0xdd << 8) */
1429        { (uint_t)DIOCGETDEV,   "DIOCGETDEV",
1430              NULL},

1432        /* mntio ioctls - ('m' << 8) */
1433        { (uint_t)MNTIOC_NMNTS,         "MNTIOC_NMNTS",
1434              NULL},
1435        { (uint_t)MNTIOC_GETDEVLIST,    "MNTIOC_GETDEVLIST",
1436              NULL},
1437        { (uint_t)MNTIOC_SETTAG,        "MNTIOC_SETTAG",
1438              "struct mnttagdesc"},
1439        { (uint_t)MNTIOC_CLRTAG,        "MNTIOC_CLRTAG",
1440              "struct mnttagdesc"},
1441        { (uint_t)MNTIOC_SHOWHIDDEN,    "MNTIOC_SHOWHIDDEN",
1442              NULL},
1443        { (uint_t)MNTIOC_GETMNTENT,     "MNTIOC_GETMNTENT",
```



```
1444              "struct mnttab"},
1445        { (uint_t)MNTIOC_GETEXTMNTENT,  "MNTIOC_GETEXTMNTENT",
1446              "struct extmnttab"},
1447        { (uint_t)MNTIOC_GETMNTANY,     "MNTIOC_GETMNTANY",
1448              "struct mnttab"},

1450        /* devinfo ioctls - ('df' << 8) - devinfo_impl.h */
1451        { (uint_t)DINFOUSRLD,           "DINFOUSRLD",
1452              NULL},
1453        { (uint_t)DINFOLODRV,           "DINFOLODRV",
1454              NULL},
1455        { (uint_t)DINFOIDENT,           "DINFOIDENT",
1456              NULL},

1458        { (uint_t)IPTUN_CREATE, "IPTUN_CREATE", "iptun_kparams_t"},
1459        { (uint_t)IPTUN_DELETE, "IPTUN_DELETE", "datalink_id_t"},
1460        { (uint_t)IPTUN_MODIFY, "IPTUN_MODIFY", "iptun_kparams_t"},
1461        { (uint_t)IPTUN_INFO,   "IPTUN_INFO",   NULL},
1462        { (uint_t)IPTUN_SET_6TO4RELAY, "IPTUN_SET_6TO4RELAY",   NULL},
1463        { (uint_t)IPTUN_GET_6TO4RELAY, "IPTUN_GET_6TO4RELAY",   NULL},

1465        /* zcons ioctls */
1466        { (uint_t)ZC_HOLDSLAVE,         "ZC_HOLDSLAVE",         NULL },
1467        { (uint_t)ZC_RELEASESLAVE,      "ZC_RELEASESLAVE",      NULL },

1469        /* hid ioctls - ('h' << 8) - hid.h */
1470        { (uint_t)HIDIOCKMGDIRECT,      "HIDIOCKMGDIRECT",      NULL },
1471        { (uint_t)HIDIOCKMSDIRECT,      "HIDIOCKMSDIRECT",      NULL },

1473        /* pm ioctls */
1474        { (uint_t)PM_SCHEDULE,          "PM_SCHEDULE",          NULL },
1475        { (uint_t)PM_GET_IDLE_TIME,     "PM_GET_IDLE_TIME",     NULL },
1476        { (uint_t)PM_GET_NUM_CMPTS,     "PM_GET_NUM_CMPTS",     NULL },
1477        { (uint_t)PM_GET_THRESHOLD,     "PM_GET_THRESHOLD",     NULL },
1478        { (uint_t)PM_SET_THRESHOLD,     "PM_SET_THRESHOLD",     NULL },
1479        { (uint_t)PM_GET_NORM_PWR,      "PM_GET_NORM_PWR",      NULL },
1480        { (uint_t)PM_SET_CUR_PWR,       "PM_SET_CUR_PWR",       NULL },
1481        { (uint_t)PM_GET_CUR_PWR,       "PM_GET_CUR_PWR",       NULL },
1482        { (uint_t)PM_GET_NUM_DEPS,      "PM_GET_NUM_DEPS",      NULL },
1483        { (uint_t)PM_GET_DEP,           "PM_GET_DEP",           NULL },
1484        { (uint_t)PM_ADD_DEP,           "PM_ADD_DEP",           NULL },
1485        { (uint_t)PM_REM_DEP,           "PM_REM_DEP",           NULL },
1486        { (uint_t)PM_REM_DEVICE,        "PM_REM_DEVICE",        NULL },
1487        { (uint_t)PM_REM_DEVICES,       "PM_REM_DEVICES",       NULL },
1488        { (uint_t)PM_DISABLE_AUTOPM,    "PM_DISABLE_AUTOPM",    NULL },
1489        { (uint_t)PM_REENABLE_AUTOPM,   "PM_REENABLE_AUTOPM",   NULL },
1490        { (uint_t)PM_SET_NORM_PWR,      "PM_SET_NORM_PWR",      NULL },
1491        { (uint_t)PM_GET_SYSTEM_THRESHOLD,      "PM_GET_SYSTEM_THRESHOLD",
1492              NULL },
1493        { (uint_t)PM_GET_DEFAULT_SYSTEM_THRESHOLD,
1494              "PM_GET_DEFAULT_SYSTEM_THRESHOLD", NULL },
1495        { (uint_t)PM_SET_SYSTEM_THRESHOLD,      "PM_SET_SYSTEM_THRESHOLD",
1496              NULL },
1497        { (uint_t)PM_START_PM,          "PM_START_PM",          NULL },
1498        { (uint_t)PM_STOP_PM,           "PM_STOP_PM",           NULL },
1499        { (uint_t)PM_RESET_PM,          "PM_RESET_PM",          NULL },
1500        { (uint_t)PM_GET_PM_STATE,      "PM_GET_PM_STATE",      NULL },
1501        { (uint_t)PM_GET_AUTOS3_STATE,  "PM_GET_AUTOS3_STATE",  NULL },
1502        { (uint_t)PM_GET_S3_SUPPORT_STATE,      "PM_GET_S3_SUPPORT_STATE",
1503              NULL },
1504        { (uint_t)PM_IDLE_DOWN,         "PM_IDLE_DOWN",         NULL },
1505        { (uint_t)PM_START_CPUPM,       "PM_START_CPUPM",       NULL },
1506        { (uint_t)PM_START_CPUPM_EV,    "PM_START_CPUPM_EV",    NULL },
1507        { (uint_t)PM_START_CPUPM_POLL,  "PM_START_CPUPM_POLL",  NULL },
1508        { (uint_t)PM_STOP_CPUPM,        "PM_STOP_CPUPM",        NULL },
1509        { (uint_t)PM_GET_CPU_THRESHOLD, "PM_GET_CPU_THRESHOLD", NULL },
```

```
1510         { (uint_t)PM_SET_CPU_THRESHOLD, "PM_SET_CPU_THRESHOLD", NULL },
1511         { (uint_t)PM_GET_CPUPM_STATE,   "PM_GET_CPUPM_STATE",   NULL },
1512         { (uint_t)PM_START_AUTOS3,      "PM_START_AUTOS3",      NULL },
1513         { (uint_t)PM_STOP_AUTOS3,       "PM_STOP_AUTOS3",       NULL },
1514         { (uint_t)PM_ENABLE_S3,         "PM_ENABLE_S3",         NULL },
1515         { (uint_t)PM_DISABLE_S3,        "PM_DISABLE_S3",        NULL },
1516         { (uint_t)PM_ENTER_S3,          "PM_ENTER_S3",          NULL },
1517         { (uint_t)PM_DISABLE_CPU_DEEP_IDLE,     "PM_DISABLE_CPU_DEEP_IDLE",
1518                 NULL },
1519         { (uint_t)PM_ENABLE_CPU_DEEP_IDLE,      "PM_START_CPU_DEEP_IDLE",
1520                 NULL },
1521         { (uint_t)PM_DEFAULT_CPU_DEEP_IDLE,     "PM_DFLT_CPU_DEEP_IDLE",
1522                 NULL },
1523 #ifdef _SYSCALL32
1524         { (uint_t)PM_GET_STATE_CHANGE,          "PM_GET_STATE_CHANGE",
1525                 "pm_state_change32_t" },
1526         { (uint_t)PM_GET_STATE_CHANGE_WAIT,     "PM_GET_STATE_CHANGE_WAIT",
1527                 "pm_state_change32_t" },
1528         { (uint_t)PM_DIRECT_NOTIFY,             "PM_DIRECT_NOTIFY",
1529                 "pm_state_change32_t" },
1530         { (uint_t)PM_DIRECT_NOTIFY_WAIT,        "PM_DIRECT_NOTIFY_WAIT",
1531                 "pm_state_change32_t" },
1532         { (uint_t)PM_REPARSE_PM_PROPS,          "PM_REPARSE_PM_PROPS",
1533                 "pm_req32_t" },
1534         { (uint_t)PM_SET_DEVICE_THRESHOLD,      "PM_SET_DEVICE_THRESHOLD",
1535                 "pm_req32_t" },
1536         { (uint_t)PM_GET_STATS,                 "PM_GET_STATS",
1537                 "pm_req32_t" },
1538         { (uint_t)PM_GET_DEVICE_THRESHOLD,      "PM_GET_DEVICE_THRESHOLD",
1539                 "pm_req32_t" },
1540         { (uint_t)PM_GET_POWER_NAME,            "PM_GET_POWER_NAME",
1541                 "pm_req32_t" },
1542         { (uint_t)PM_GET_POWER_LEVELS,          "PM_GET_POWER_LEVELS",
1543                 "pm_req32_t" },
1544         { (uint_t)PM_GET_NUM_COMPONENTS,        "PM_GET_NUM_COMPONENTS",
1545                 "pm_req32_t" },
1546         { (uint_t)PM_GET_COMPONENT_NAME,        "PM_GET_COMPONENT_NAME",
1547                 "pm_req32_t" },
1548         { (uint_t)PM_GET_NUM_POWER_LEVELS,      "PM_GET_NUM_POWER_LEVELS",
1549                 "pm_req32_t" },
1550         { (uint_t)PM_DIRECT_PM,                 "PM_DIRECT_PM",
1551                 "pm_req32_t" },
1552         { (uint_t)PM_RELEASE_DIRECT_PM,         "PM_RELEASE_DIRECT_PM",
1553                 "pm_req32_t" },
1554         { (uint_t)PM_RESET_DEVICE_THRESHOLD,    "PM_RESET_DEVICE_THRESHOLD",
1555                 "pm_req32_t" },
1556         { (uint_t)PM_GET_DEVICE_TYPE,           "PM_GET_DEVICE_TYPE",
1557                 "pm_req32_t" },
1558         { (uint_t)PM_SET_COMPONENT_THRESHOLDS,  "PM_SET_COMPONENT_THRESHOLDS",
1559                 "pm_req32_t" },
1560         { (uint_t)PM_GET_COMPONENT_THRESHOLDS,  "PM_GET_COMPONENT_THRESHOLDS",
1561                 "pm_req32_t" },
1562         { (uint_t)PM_GET_DEVICE_THRESHOLD_BASIS,
1563                 "PM_GET_DEVICE_THRESHOLD_BASIS",        "pm_req32_t" },
1564         { (uint_t)PM_SET_CURRENT_POWER,         "PM_SET_CURRENT_POWER",
1565                 "pm_req32_t" },
1566         { (uint_t)PM_GET_CURRENT_POWER,         "PM_GET_CURRENT_POWER",
1567                 "pm_req32_t" },
1568         { (uint_t)PM_GET_FULL_POWER,            "PM_GET_FULL_POWER",
1569                 "pm_req32_t" },
1570         { (uint_t)PM_ADD_DEPENDENT,             "PM_ADD_DEPENDENT",
1571                 "pm_req32_t" },
1572         { (uint_t)PM_GET_TIME_IDLE,             "PM_GET_TIME_IDLE",
1573                 "pm_req32_t" },
1574         { (uint_t)PM_ADD_DEPENDENT_PROPERTY,    "PM_ADD_DEPENDENT_PROPERTY",
1575                 "pm_req32_t" },
```



```
1576         { (uint_t)PM_GET_CMD_NAME,              "PM_GET_CMD_NAME",
1577                 "pm_req32_t" },
1578         { (uint_t)PM_SEARCH_LIST,               "PM_SEARCH_LIST",
1579                 "pm_searchargs32_t" },
1580 #else  /* _SYSCALL32 */
1581         { (uint_t)PM_GET_STATE_CHANGE,          "PM_GET_STATE_CHANGE",
1582                 "pm_state_change_t" },
1583         { (uint_t)PM_GET_STATE_CHANGE_WAIT,     "PM_GET_STATE_CHANGE_WAIT",
1584                 "pm_state_change_t" },
1585         { (uint_t)PM_DIRECT_NOTIFY,             "PM_DIRECT_NOTIFY",
1586                 "pm_state_change_t" },
1587         { (uint_t)PM_DIRECT_NOTIFY_WAIT,        "PM_DIRECT_NOTIFY_WAIT",
1588                 "pm_state_change_t" },
1589         { (uint_t)PM_REPARSE_PM_PROPS,          "PM_REPARSE_PM_PROPS",
1590                 "pm_req_t" },
1591         { (uint_t)PM_SET_DEVICE_THRESHOLD,      "PM_SET_DEVICE_THRESHOLD",
1592                 "pm_req_t" },
1593         { (uint_t)PM_GET_STATS,                 "PM_GET_STATS",
1594                 "pm_req_t" },
1595         { (uint_t)PM_GET_DEVICE_THRESHOLD,      "PM_GET_DEVICE_THRESHOLD",
1596                 "pm_req_t" },
1597         { (uint_t)PM_GET_POWER_NAME,            "PM_GET_POWER_NAME",
1598                 "pm_req_t" },
1599         { (uint_t)PM_GET_POWER_LEVELS,          "PM_GET_POWER_LEVELS",
1600                 "pm_req_t" },
1601         { (uint_t)PM_GET_NUM_COMPONENTS,        "PM_GET_NUM_COMPONENTS",
1602                 "pm_req_t" },
1603         { (uint_t)PM_GET_COMPONENT_NAME,        "PM_GET_COMPONENT_NAME",
1604                 "pm_req_t" },
1605         { (uint_t)PM_GET_NUM_POWER_LEVELS,      "PM_GET_NUM_POWER_LEVELS",
1606                 "pm_req_t" },
1607         { (uint_t)PM_DIRECT_PM,                 "PM_DIRECT_PM",
1608                 "pm_req_t" },
1609         { (uint_t)PM_RELEASE_DIRECT_PM,         "PM_RELEASE_DIRECT_PM",
1610                 "pm_req_t" },
1611         { (uint_t)PM_RESET_DEVICE_THRESHOLD,    "PM_RESET_DEVICE_THRESHOLD",
1612                 "pm_req_t" },
1613         { (uint_t)PM_GET_DEVICE_TYPE,           "PM_GET_DEVICE_TYPE",
1614                 "pm_req_t" },
1615         { (uint_t)PM_SET_COMPONENT_THRESHOLDS,  "PM_SET_COMPONENT_THRESHOLDS",
1616                 "pm_req_t" },
1617         { (uint_t)PM_GET_COMPONENT_THRESHOLDS,  "PM_GET_COMPONENT_THRESHOLDS",
1618                 "pm_req_t" },
1619         { (uint_t)PM_GET_DEVICE_THRESHOLD_BASIS,
1620                 "PM_GET_DEVICE_THRESHOLD_BASIS",        "pm_req_t" },
1621         { (uint_t)PM_SET_CURRENT_POWER,         "PM_SET_CURRENT_POWER",
1622                 "pm_req_t" },
1623         { (uint_t)PM_GET_CURRENT_POWER,         "PM_GET_CURRENT_POWER",
1624                 "pm_req_t" },
1625         { (uint_t)PM_GET_FULL_POWER,            "PM_GET_FULL_POWER",
1626                 "pm_req_t" },
1627         { (uint_t)PM_ADD_DEPENDENT,             "PM_ADD_DEPENDENT",
1628                 "pm_req_t" },
1629         { (uint_t)PM_GET_TIME_IDLE,             "PM_GET_TIME_IDLE",
1630                 "pm_req_t" },
1631         { (uint_t)PM_ADD_DEPENDENT_PROPERTY,    "PM_ADD_DEPENDENT_PROPERTY",
1632                 "pm_req_t" },
1633         { (uint_t)PM_GET_CMD_NAME,              "PM_GET_CMD_NAME",
1634                 "pm_req_t" },
1635         { (uint_t)PM_SEARCH_LIST,       "PM_SEARCH_LIST",
1636                 "pm_searchargs_t" },
1637 #endif /* _SYSCALL */

1639         { (uint_t)0, NULL, NULL }
1640 };
```

```
1642 void
1643 ioctl_ioccom(char *buf, size_t size, uint_t code, int nbytes, int x, int y)
1644 {
1645         const char *inoutstr;

1647         if (code & IOC_VOID)
1648                 inoutstr = "";
1649         else if ((code & IOC_INOUT) == IOC_INOUT)
1650                 inoutstr = "WR";
1651         else
1652                 inoutstr = code & IOC_IN ? "W" : "R";

1654         if (isascii(x) && isprint(x))
1655                 (void) snprintf(buf, size, "_IO%sN('%c', %d, %d)", inoutstr,
1656                     x, y, nbytes);
1657         else
1658                 (void) snprintf(buf, size, "_IO%sN(0x%x, %d, %d)", inoutstr,
1659                     x, y, nbytes);
1660 }


1663 const char *
1664 ioctlname(private_t *pri, uint_t code)
1665 {
1666         const struct ioc *ip;
1667         const char *str = NULL;

1669         for (ip = &ioc[0]; ip->name; ip++) {
1670                 if (code == ip->code) {
1671                         str = ip->name;
1672                         break;
1673                 }
1674         }

1676         /*
1677          * Developers hide ascii ioctl names in the ioctl subcode; for example
1678          * 0x445210 should be printed 'D'<<16|'R'<<8|10.  We allow for all
1679          * three high order bytes (called hi, mid and lo) to contain ascii
1680          * characters.
1681          */
1682         if (str == NULL) {
1683                 int c_hi = code >> 24;
1684                 int c_mid = (code >> 16) & 0xff;
1685                 int c_mid_nm = (code >> 16);
1686                 int c_lo = (code >> 8) & 0xff;
1687                 int c_lo_nm = code >> 8;

1689                 if (isascii(c_lo) && isprint(c_lo) &&
1690                     isascii(c_mid) && isprint(c_mid) &&
1691                     isascii(c_hi) && isprint(c_hi))
1692                         (void) sprintf(pri->code_buf,
1693                             "(('%c'<<24)|('%c'<<16)|('%c'<<8)|%d)",
1694                             c_hi, c_mid, c_lo, code & 0xff);
1695                 else if (isascii(c_lo) && isprint(c_lo) &&
1696                     isascii(c_mid_nm) && isprint(c_mid_nm))
1697                         (void) sprintf(pri->code_buf,
1698                             "(('%c'<<16)|('%c'<<8)|%d)", c_mid, c_lo,
1699                             code & 0xff);
1700                 else if (isascii(c_lo_nm) && isprint(c_lo_nm))
1701                         (void) sprintf(pri->code_buf, "(('%c'<<8)|%d)",
1702                             c_lo_nm, code & 0xff);
1703                 else if (code & (IOC_VOID|IOC_INOUT))
1704                         ioctl_ioccom(pri->code_buf, sizeof (pri->code_buf),
1705                             code, c_mid, c_lo, code & 0xff);
1706                 else
1707                         (void) sprintf(pri->code_buf, "0x%.4X", code);
```



```
1708                 str = (const char *)pri->code_buf;
1709         }

1711         return (str);
1712 }


1715 const char *
1716 ioctldatastruct(uint_t code)
1717 {
1718         const struct ioc *ip;
1719         const char *str = NULL;

1721         for (ip = &ioc[0]; ip->name != NULL; ip++) {
1722                 if (code == ip->code) {
1723                         str = ip->datastruct;
1724                         break;
1725                 }
1726         }
1727         return (str);
1728 }


1731 const char *
1732 fcntlname(int code)
1733 {
1734         const char *str = NULL;

1736         if (code >= FCNTLMIN && code <= FCNTLMAX)
1737                 str = FCNTLname[code-FCNTLMIN];
1738         return (str);
1739 }

1741 const char *
1742 sfsname(int code)
1743 {
1744         const char *str = NULL;

1746         if (code >= SYSFSMIN && code <= SYSFSMAX)
1747                 str = SYSFSname[code-SYSFSMIN];
1748         return (str);
1749 }

1751 /* ARGSUSED */
1752 const char *
1753 si86name(int code)
1754 {
1755         const char *str = NULL;

1757 #if defined(__i386) || defined(__amd64)
1758         switch (code) {
1759         case SI86SWPI:          str = "SI86SWPI";       break;
1760         case SI86SYM:           str = "SI86SYM";        break;
1761         case SI86CONF:          str = "SI86CONF";       break;
1762         case SI86BOOT:          str = "SI86BOOT";       break;
1763         case SI86AUTO:          str = "SI86AUTO";       break;
1764         case SI86EDT:           str = "SI86EDT";        break;
1765         case SI86SWAP:          str = "SI86SWAP";       break;
1766         case SI86FPHW:          str = "SI86FPHW";       break;
1767         case SI86FPSTART:       str = "SI86FPSTART";    break;
1768         case GRNON:             str = "GRNON";          break;
1769         case GRNFLASH:          str = "GRNFLASH";       break;
1770         case STIME:             str = "STIME";          break;
1771         case SETNAME:           str = "SETNAME";        break;
1772         case RNVR:              str = "RNVR";           break;
1773         case WNVR:              str = "WNVR";           break;
```

```
1774         case RTODC:             str = "RTODC";          break;
1775         case CHKSER:            str = "CHKSER";         break;
1776         case SI86NVPRT:         str = "SI86NVPRT";      break;
1777         case SANUPD:            str = "SANUPD";         break;
1778         case SI86KSTR:          str = "SI86KSTR";       break;
1779         case SI86MEM:           str = "SI86MEM";        break;
1780         case SI86TODEMON:       str = "SI86TODEMON";    break;
1781         case SI86CCDEMON:       str = "SI86CCDEMON";    break;
1782         case SI86CACHE:         str = "SI86CACHE";      break;
1783         case SI86DELMEM:        str = "SI86DELMEM";     break;
1784         case SI86ADDMEM:        str = "SI86ADDMEM";     break;
1785 /* 71 through 74 reserved for VPIX */
1786         case SI86V86:           str = "SI86V86";        break;
1787         case SI86SLTIME:        str = "SI86SLTIME";     break;
1788         case SI86DSCR:          str = "SI86DSCR";       break;
1789         case RDUBLK:            str = "RDUBLK";         break;
1790 /* NFA entry point */
1791         case SI86NFA:           str = "SI86NFA";        break;
1792         case SI86VM86:          str = "SI86VM86";       break;
1793         case SI86VMENABLE:      str = "SI86VMENABLE";   break;
1794         case SI86LIMUSER:       str = "SI86LIMUSER";    break;
1795         case SI86RDID:          str = "SI86RDID";       break;
1796         case SI86RDBOOT:        str = "SI86RDBOOT";     break;
1797 /* Merged Product defines */
1798         case SI86SHFIL:         str = "SI86SHFIL";      break;
1799         case SI86PCHRGN:        str = "SI86PCHRGN";     break;
1800         case SI86BADVISE:       str = "SI86BADVISE";    break;
1801         case SI86SHRGN:         str = "SI86SHRGN";      break;
1802         case SI86CHIDT:         str = "SI86CHIDT";      break;
1803         case SI86EMULRDA:       str = "SI86EMULRDA";    break;
1804 /* RTC commands */
1805         case WTODC:             str = "WTODC";          break;
1806         case SGMTL:             str = "SGMTL";          break;
1807         case GGMTL:             str = "GGMTL";          break;
1808         case RTCSYNC:           str = "RTCSYNC";        break;
1809         }
1810 #endif /* __i386 */

1812         return (str);
1813 }

1815 const char *
1816 utscode(int code)
1817 {
1818         const char *str = NULL;

1820         switch (code) {
1821         case UTS_UNAME:         str = "UNAME";  break;
1822         case UTS_USTAT:         str = "USTAT";  break;
1823         case UTS_FUSERS:        str = "FUSERS"; break;
1824         }

1826         return (str);
1827 }

1829 const char *
1830 rctlsyscode(int code)
1831 {
1832         const char *str = NULL;
1833         switch (code) {
1834         case 0:         str = "GETRCTL";        break;
1835         case 1:         str = "SETRCTL";        break;
1836         case 2:         str = "RCTLSYS_LST";    break;
1837         case 3:         str = "RCTLSYS_CTL";    break;
1838         case 4:         str = "RCTLSYS_SETPROJ";        break;
1839         default:        str = "UNKNOWN";        break;
```



```
1840         }
1841         return (str);
1842 }

1844 const char *
1845 rctl_local_action(private_t *pri, uint_t val)
1846 {
1847         uint_t action = val & (~RCTL_LOCAL_ACTION_MASK);

1849         char *s = pri->code_buf;

1851         *s = '\0';

1853         if (action & RCTL_LOCAL_NOACTION) {
1854                 action ^= RCTL_LOCAL_NOACTION;
1855                 (void) strlcat(s, "|RCTL_LOCAL_NOACTION",
1856                     sizeof (pri->code_buf));
1857         }
1858         if (action & RCTL_LOCAL_SIGNAL) {
1859                 action ^= RCTL_LOCAL_SIGNAL;
1860                 (void) strlcat(s, "|RCTL_LOCAL_SIGNAL",
1861                     sizeof (pri->code_buf));
1862         }
1863         if (action & RCTL_LOCAL_DENY) {
1864                 action ^= RCTL_LOCAL_DENY;
1865                 (void) strlcat(s, "|RCTL_LOCAL_DENY",
1866                     sizeof (pri->code_buf));
1867         }

1869         if ((action & (~RCTL_LOCAL_ACTION_MASK)) != 0)
1870                 return (NULL);
1871         else if (*s != '\0')
1872                 return (s+1);
1873         else
1874                 return (NULL);
1875 }


1878 const char *
1879 rctl_local_flags(private_t *pri, uint_t val)
1880 {
1881         uint_t pval = val & RCTL_LOCAL_ACTION_MASK;
1882         char *s = pri->code_buf;

1884         *s = '\0';

1886         if (pval & RCTL_LOCAL_MAXIMAL) {
1887                 pval ^= RCTL_LOCAL_MAXIMAL;
1888                 (void) strlcat(s, "|RCTL_LOCAL_MAXIMAL",
1889                     sizeof (pri->code_buf));
1890         }

1892         if ((pval & RCTL_LOCAL_ACTION_MASK) != 0)
1893                 return (NULL);
1894         else if (*s != '\0')
1895                 return (s+1);
1896         else
1897                 return (NULL);
1898 }


1901 const char *
1902 sconfname(int code)
1903 {
1904         const char *str = NULL;
```

```
1906         if (code >= SCONFMIN && code <= SCONFMAX)
1907                 str = SCONFname[code-SCONFMIN];
1908         return (str);
1909 }

1911 const char *
1912 pathconfname(int code)
1913 {
1914         const char *str = NULL;

1916         if (code >= PATHCONFMIN && code <= PATHCONFMAX)
1917                 str = PATHCONFname[code-PATHCONFMIN];
1918         return (str);
1919 }

1921 #define ALL_O_FLAGS \
1922         (O_NDELAY|O_APPEND|O_SYNC|O_DSYNC|O_NONBLOCK|O_CREAT|O_TRUNC\
1923         |O_EXCL|O_NOCTTY|O_LARGEFILE|O_RSYNC|O_XATTR|O_NOFOLLOW|O_NOLINKS\
1924         |FXATTRDIROPEN)

1926 const char *
1927 openarg(private_t *pri, int arg)
1928 {
1929         char *str = pri->code_buf;

1931         if ((arg & ~(O_ACCMODE | ALL_O_FLAGS)) != 0)
1932                 return (NULL);

1934         switch (arg & O_ACCMODE) {
1935         default:
1936                 return (NULL);
1937         case O_RDONLY:
1938                 (void) strcpy(str, "O_RDONLY");
1939                 break;
1940         case O_WRONLY:
1941                 (void) strcpy(str, "O_WRONLY");
1942                 break;
1943         case O_RDWR:
1944                 (void) strcpy(str, "O_RDWR");
1945                 break;
1946         case O_SEARCH:
1947                 (void) strcpy(str, "O_SEARCH");
1948                 break;
1949         case O_EXEC:
1950                 (void) strcpy(str, "O_EXEC");
1951                 break;
1952         }

1954         if (arg & O_NDELAY)
1955                 (void) strlcat(str, "|O_NDELAY", sizeof (pri->code_buf));
1956         if (arg & O_APPEND)
1957                 (void) strlcat(str, "|O_APPEND", sizeof (pri->code_buf));
1958         if (arg & O_SYNC)
1959                 (void) strlcat(str, "|O_SYNC", sizeof (pri->code_buf));
1960         if (arg & O_DSYNC)
1961                 (void) strlcat(str, "|O_DSYNC", sizeof (pri->code_buf));
1962         if (arg & O_NONBLOCK)
1963                 (void) strlcat(str, "|O_NONBLOCK", sizeof (pri->code_buf));
1964         if (arg & O_CREAT)
1965                 (void) strlcat(str, "|O_CREAT", sizeof (pri->code_buf));
1966         if (arg & O_TRUNC)
1967                 (void) strlcat(str, "|O_TRUNC", sizeof (pri->code_buf));
1968         if (arg & O_EXCL)
1969                 (void) strlcat(str, "|O_EXCL", sizeof (pri->code_buf));
1970         if (arg & O_NOCTTY)
1971                 (void) strlcat(str, "|O_NOCTTY", sizeof (pri->code_buf));
```



```
1972         if (arg & O_LARGEFILE)
1973                 (void) strlcat(str, "|O_LARGEFILE", sizeof (pri->code_buf));
1974         if (arg & O_RSYNC)
1975                 (void) strlcat(str, "|O_RSYNC", sizeof (pri->code_buf));
1976         if (arg & O_XATTR)
1977                 (void) strlcat(str, "|O_XATTR", sizeof (pri->code_buf));
1978         if (arg & O_NOFOLLOW)
1979                 (void) strlcat(str, "|O_NOFOLLOW", sizeof (pri->code_buf));
1980         if (arg & O_NOLINKS)
1981                 (void) strlcat(str, "|O_NOLINKS", sizeof (pri->code_buf));
1982         if (arg & FXATTRDIROPEN)
1983                 (void) strlcat(str, "|FXATTRDIROPEN", sizeof (pri->code_buf));

1985         return ((const char *)str);
1986 }

1988 const char *
1989 whencearg(int arg)
1990 {
1991         const char *str = NULL;

1993         switch (arg) {
1994         case SEEK_SET:  str = "SEEK_SET";       break;
1995         case SEEK_CUR:  str = "SEEK_CUR";       break;
1996         case SEEK_END:  str = "SEEK_END";       break;
1997         case SEEK_DATA: str = "SEEK_DATA";      break;
1998         case SEEK_HOLE: str = "SEEK_HOLE";      break;
1999         }

2001         return (str);
2002 }

2004 #define IPC_FLAGS       (IPC_ALLOC|IPC_CREAT|IPC_EXCL|IPC_NOWAIT)

2006 char *
2007 ipcflags(private_t *pri, int arg)
2008 {
2009         char *str = pri->code_buf;

2011         if (arg & 0777)
2012                 (void) sprintf(str, "0%.3o", arg&0777);
2013         else
2014                 *str = '\0';

2016         if (arg & IPC_ALLOC)
2017                 (void) strcat(str, "|IPC_ALLOC");
2018         if (arg & IPC_CREAT)
2019                 (void) strcat(str, "|IPC_CREAT");
2020         if (arg & IPC_EXCL)
2021                 (void) strcat(str, "|IPC_EXCL");
2022         if (arg & IPC_NOWAIT)
2023                 (void) strcat(str, "|IPC_NOWAIT");

2025         return (str);
2026 }

2028 const char *
2029 msgflags(private_t *pri, int arg)
2030 {
2031         char *str;

2033         if (arg == 0 || (arg & ~(IPC_FLAGS|MSG_NOERROR|0777)) != 0)
2034                 return ((char *)NULL);

2036         str = ipcflags(pri, arg);
```

```
2038         if (arg & MSG_NOERROR)
2039                 (void) strcat(str, "|MSG_NOERROR");

2041         if (*str == '|')
2042                 str++;
2043         return ((const char *)str);
2044 }

2046 const char *
2047 semflags(private_t *pri, int arg)
2048 {
2049         char *str;

2051         if (arg == 0 || (arg & ~(IPC_FLAGS|SEM_UNDO|0777)) != 0)
2052                 return ((char *)NULL);

2054         str = ipcflags(pri, arg);

2056         if (arg & SEM_UNDO)
2057                 (void) strcat(str, "|SEM_UNDO");

2059         if (*str == '|')
2060                 str++;
2061         return ((const char *)str);
2062 }

2064 const char *
2065 shmflags(private_t *pri, int arg)
2066 {
2067         char *str;

2069         if (arg == 0 || (arg & ~(IPC_FLAGS|SHM_RDONLY|SHM_RND|0777)) != 0)
2070                 return ((char *)NULL);

2072         str = ipcflags(pri, arg);

2074         if (arg & SHM_RDONLY)
2075                 (void) strcat(str, "|SHM_RDONLY");
2076         if (arg & SHM_RND)
2077                 (void) strcat(str, "|SHM_RND");

2079         if (*str == '|')
2080                 str++;
2081         return ((const char *)str);
2082 }

2084 #define MSGCMDMIN       0
2085 #define MSGCMDMAX       IPC_STAT64
2086 const char *const MSGCMDname[MSGCMDMAX+1] = {
2087         NULL, NULL, NULL, NULL, NULL,
2088         NULL, NULL, NULL, NULL, NULL,
2089         "IPC_RMID",     /* 10 */
2090         "IPC_SET",      /* 11 */
2091         "IPC_STAT",     /* 12 */
2092         "IPC_SET64",    /* 13 */
2093         "IPC_STAT64",   /* 14 */
2094 };

2096 #define SEMCMDMIN       0
2097 #define SEMCMDMAX       IPC_STAT64
2098 const char *const SEMCMDname[SEMCMDMAX+1] = {
2099         NULL,           /* 0 */
2100         NULL,           /* 1 */
2101         NULL,           /* 2 */
2102         "GETNCNT",      /* 3 */
2103         "GETPID",       /* 4 */
```



```
2104         "GETVAL",       /* 5 */
2105         "GETALL",       /* 6 */
2106         "GETZCNT",      /* 7 */
2107         "SETVAL",       /* 8 */
2108         "SETALL",       /* 9 */
2109         "IPC_RMID",     /* 10 */
2110         "IPC_SET",      /* 11 */
2111         "IPC_STAT",     /* 12 */
2112         "IPC_SET64",    /* 13 */
2113         "IPC_STAT64",   /* 14 */
2114 };

2116 #define SHMCMDMIN       0
2117 #define SHMCMDMAX       IPC_STAT64
2118 const char *const SHMCMDname[SHMCMDMAX+1] = {
2119         NULL,           /* 0 */
2120         NULL,           /* 1 */
2121         NULL,           /* 2 */
2122         "SHM_LOCK",     /* 3 */
2123         "SHM_UNLOCK",   /* 4 */
2124         NULL, NULL, NULL, NULL, NULL,                   /* 5 NULLs */
2125         "IPC_RMID",     /* 10 */
2126         "IPC_SET",      /* 11 */
2127         "IPC_STAT",     /* 12 */
2128         "IPC_SET64",    /* 13 */
2129         "IPC_STAT64",   /* 14 */
2130 };

2132 const char *
2133 msgcmd(int arg)
2134 {
2135         const char *str = NULL;

2137         if (arg >= MSGCMDMIN && arg <= MSGCMDMAX)
2138                 str = MSGCMDname[arg-MSGCMDMIN];
2139         return (str);
2140 }

2142 const char *
2143 semcmd(int arg)
2144 {
2145         const char *str = NULL;

2147         if (arg >= SEMCMDMIN && arg <= SEMCMDMAX)
2148                 str = SEMCMDname[arg-SEMCMDMIN];
2149         return (str);
2150 }

2152 const char *
2153 shmcmd(int arg)
2154 {
2155         const char *str = NULL;

2157         if (arg >= SHMCMDMIN && arg <= SHMCMDMAX)
2158                 str = SHMCMDname[arg-SHMCMDMIN];
2159         return (str);
2160 }

2162 const char *
2163 strrdopt(int arg)       /* streams read option (I_SRDOPT I_GRDOPT) */
2164 {
2165         const char *str = NULL;

2167         switch (arg) {
2168         case RNORM:     str = "RNORM";          break;
2169         case RMSGD:     str = "RMSGD";          break;
```

```
2170         case RMSGN:     str = "RMSGN";          break;
2171         }

2173         return (str);
2174 }

2176 /* bit map of streams events (I_SETSIG & I_GETSIG) */
2177 const char *
2178 strevents(private_t *pri, int arg)
2179 {
2180         char *str = pri->code_buf;

2182         if (arg & ~(S_INPUT|S_HIPRI|S_OUTPUT|S_MSG|S_ERROR|S_HANGUP))
2183                 return ((char *)NULL);

2185         *str = '\0';
2186         if (arg & S_INPUT)
2187                 (void) strcat(str, "|S_INPUT");
2188         if (arg & S_HIPRI)
2189                 (void) strcat(str, "|S_HIPRI");
2190         if (arg & S_OUTPUT)
2191                 (void) strcat(str, "|S_OUTPUT");
2192         if (arg & S_MSG)
2193                 (void) strcat(str, "|S_MSG");
2194         if (arg & S_ERROR)
2195                 (void) strcat(str, "|S_ERROR");
2196         if (arg & S_HANGUP)
2197                 (void) strcat(str, "|S_HANGUP");

2199         return ((const char *)(str+1));
2200 }

2202 const char *
2203 tiocflush(private_t *pri, int arg)      /* bit map passsed by TIOCFLUSH */
2204 {
2205         char *str = pri->code_buf;

2207         if (arg & ~(FREAD|FWRITE))
2208                 return ((char *)NULL);

2210         *str = '\0';
2211         if (arg & FREAD)
2212                 (void) strcat(str, "|FREAD");
2213         if (arg & FWRITE)
2214                 (void) strcat(str, "|FWRITE");

2216         return ((const char *)(str+1));
2217 }

2219 const char *
2220 strflush(int arg)       /* streams flush option (I_FLUSH) */
2221 {
2222         const char *str = NULL;

2224         switch (arg) {
2225         case FLUSHR:    str = "FLUSHR";         break;
2226         case FLUSHW:    str = "FLUSHW";         break;
2227         case FLUSHRW:   str = "FLUSHRW";        break;
2228         }

2230         return (str);
2231 }

2233 #define ALL_MOUNT_FLAGS (MS_RDONLY|MS_FSS|MS_DATA|MS_NOSUID|MS_REMOUNT| \
2234         MS_NOTRUNC|MS_OVERLAY|MS_OPTIONSTR|MS_GLOBAL|MS_FORCE|MS_NOMNTTAB)
```



```
2236 const char *
2237 mountflags(private_t *pri, int arg)     /* bit map of mount syscall flags */
2238 {
2239         char *str = pri->code_buf;
2240         size_t used = 0;

2242         if (arg & ~ALL_MOUNT_FLAGS)
2243                 return ((char *)NULL);

2245         *str = '\0';
2246         if (arg & MS_RDONLY)
2247                 used = strlcat(str, "|MS_RDONLY", sizeof (pri->code_buf));
2248         if (arg & MS_FSS)
2249                 used = strlcat(str, "|MS_FSS", sizeof (pri->code_buf));
2250         if (arg & MS_DATA)
2251                 used = strlcat(str, "|MS_DATA", sizeof (pri->code_buf));
2252         if (arg & MS_NOSUID)
2253                 used = strlcat(str, "|MS_NOSUID", sizeof (pri->code_buf));
2254         if (arg & MS_REMOUNT)
2255                 used = strlcat(str, "|MS_REMOUNT", sizeof (pri->code_buf));
2256         if (arg & MS_NOTRUNC)
2257                 used = strlcat(str, "|MS_NOTRUNC", sizeof (pri->code_buf));
2258         if (arg & MS_OVERLAY)
2259                 used = strlcat(str, "|MS_OVERLAY", sizeof (pri->code_buf));
2260         if (arg & MS_OPTIONSTR)
2261                 used = strlcat(str, "|MS_OPTIONSTR", sizeof (pri->code_buf));
2262         if (arg & MS_GLOBAL)
2263                 used = strlcat(str, "|MS_GLOBAL", sizeof (pri->code_buf));
2264         if (arg & MS_FORCE)
2265                 used = strlcat(str, "|MS_FORCE", sizeof (pri->code_buf));
2266         if (arg & MS_NOMNTTAB)
2267                 used = strlcat(str, "|MS_NOMNTTAB", sizeof (pri->code_buf));

2269         if (used == 0 || used >= sizeof (pri->code_buf))
2270                 return ((char *)NULL);                  /* use prt_hex() */

2272         return ((const char *)(str+1));
2273 }

2275 const char *
2276 svfsflags(private_t *pri, ulong_t arg)  /* bit map of statvfs syscall flags */
2277 {
2278         char *str = pri->code_buf;

2280         if (arg & ~(ST_RDONLY|ST_NOSUID|ST_NOTRUNC)) {
2281                 (void) sprintf(str, "0x%lx", arg);
2282                 return (str);
2283         }
2284         *str = '\0';
2285         if (arg & ST_RDONLY)
2286                 (void) strcat(str, "|ST_RDONLY");
2287         if (arg & ST_NOSUID)
2288                 (void) strcat(str, "|ST_NOSUID");
2289         if (arg & ST_NOTRUNC)
2290                 (void) strcat(str, "|ST_NOTRUNC");
2291         if (*str == '\0')
2292                 (void) strcat(str, "|0");
2293         return ((const char *)(str+1));
2294 }

2296 const char *
2297 fuiname(int arg)        /* fusers() input argument */
2298 {
2299         const char *str = NULL;

2301         switch (arg) {
```

```
2302         case F_FILE_ONLY:       str = "F_FILE_ONLY";            break;
2303         case F_CONTAINED:       str = "F_CONTAINED";            break;
2304         }

2306         return (str);
2307 }

2309 const char *
2310 fuflags(private_t *pri, int arg)        /* fusers() output flags */
2311 {
2312         char *str = pri->code_buf;

2314         if (arg & ~(F_CDIR|F_RDIR|F_TEXT|F_MAP|F_OPEN|F_TRACE|F_TTY)) {
2315                 (void) sprintf(str, "0x%x", arg);
2316                 return (str);
2317         }
2318         *str = '\0';
2319         if (arg & F_CDIR)
2320                 (void) strcat(str, "|F_CDIR");
2321         if (arg & F_RDIR)
2322                 (void) strcat(str, "|F_RDIR");
2323         if (arg & F_TEXT)
2324                 (void) strcat(str, "|F_TEXT");
2325         if (arg & F_MAP)
2326                 (void) strcat(str, "|F_MAP");
2327         if (arg & F_OPEN)
2328                 (void) strcat(str, "|F_OPEN");
2329         if (arg & F_TRACE)
2330                 (void) strcat(str, "|F_TRACE");
2331         if (arg & F_TTY)
2332                 (void) strcat(str, "|F_TTY");
2333         if (*str == '\0')
2334                 (void) strcat(str, "|0");
2335         return ((const char *)(str+1));
2336 }


2339 const char *
2340 ipprotos(int arg)       /* IP protocols cf. netinet/in.h */
2341 {
2342         switch (arg) {
2343         case IPPROTO_IP:        return ("IPPROTO_IP");
2344         case IPPROTO_ICMP:      return ("IPPROTO_ICMP");
2345         case IPPROTO_IGMP:      return ("IPPROTO_IGMP");
2346         case IPPROTO_GGP:       return ("IPPROTO_GGP");
2347         case IPPROTO_ENCAP:     return ("IPPROTO_ENCAP");
2348         case IPPROTO_TCP:       return ("IPPROTO_TCP");
2349         case IPPROTO_EGP:       return ("IPPROTO_EGP");
2350         case IPPROTO_PUP:       return ("IPPROTO_PUP");
2351         case IPPROTO_UDP:       return ("IPPROTO_UDP");
2352         case IPPROTO_IDP:       return ("IPPROTO_IDP");
2353         case IPPROTO_IPV6:      return ("IPPROTO_IPV6");
2354         case IPPROTO_ROUTING:   return ("IPPROTO_ROUTING");
2355         case IPPROTO_FRAGMENT:  return ("IPPROTO_FRAGMENT");
2356         case IPPROTO_RSVP:      return ("IPPROTO_RSVP");
2357         case IPPROTO_ESP:       return ("IPPROTO_ESP");
2358         case IPPROTO_AH:        return ("IPPROTO_AH");
2359         case IPPROTO_ICMPV6:    return ("IPPROTO_ICMPV6");
2360         case IPPROTO_NONE:      return ("IPPROTO_NONE");
2361         case IPPROTO_DSTOPTS:   return ("IPPROTO_DSTOPTS");
2362         case IPPROTO_HELLO:     return ("IPPROTO_HELLO");
2363         case IPPROTO_ND:        return ("IPPROTO_ND");
2364         case IPPROTO_EON:       return ("IPPROTO_EON");
2365         case IPPROTO_PIM:       return ("IPPROTO_PIM");
2366         case IPPROTO_SCTP:      return ("IPPROTO_SCTP");
2367         case IPPROTO_RAW:       return ("IPPROTO_RAW");
```



```
2368         default:                return (NULL);
2369         }
2370 }
```

```
**********************************************************
  163475 Wed Oct 17 21:48:36 2012
new/usr/src/cmd/zfs/zfs_main.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2012 Milan Jurik. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <libgen.h>
#include <libintl.h>
#include <libuutil.h>
#include <libnvpair.h>
#include <locale.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <fcntl.h>
#include <zone.h>
#include <grp.h>
#include <pwd.h>
#include <signal.h>
#include <sys/list.h>
#include <sys/mkdev.h>
#include <sys/mntent.h>
#include <sys/mnttab.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/fs/zfs.h>
#include <sys/types.h>
#include <time.h>

#include <libzfs.h>
```



```
#include <libzfs_core.h>
#include <zfs_prop.h>
#include <zfs_deleg.h>
#include <libuutil.h>
#include <aclutils.h>
#include <directory.h>

#include "zfs_iter.h"
#include "zfs_util.h"
#include "zfs_comutil.h"

libzfs_handle_t *g_zfs;

static FILE *mnttab_file;
static char history_str[HIS_MAX_RECORD_LEN];
static boolean_t log_history = B_TRUE;

static int zfs_do_clone(int argc, char **argv);
static int zfs_do_create(int argc, char **argv);
static int zfs_do_destroy(int argc, char **argv);
static int zfs_do_get(int argc, char **argv);
static int zfs_do_inherit(int argc, char **argv);
static int zfs_do_list(int argc, char **argv);
static int zfs_do_mount(int argc, char **argv);
static int zfs_do_rename(int argc, char **argv);
static int zfs_do_rollback(int argc, char **argv);
static int zfs_do_set(int argc, char **argv);
static int zfs_do_upgrade(int argc, char **argv);
static int zfs_do_snapshot(int argc, char **argv);
static int zfs_do_unmount(int argc, char **argv);
static int zfs_do_share(int argc, char **argv);
static int zfs_do_unshare(int argc, char **argv);
static int zfs_do_send(int argc, char **argv);
static int zfs_do_fits_send(int argc, char **argv);
#endif /* ! codereview */
static int zfs_do_receive(int argc, char **argv);
static int zfs_do_promote(int argc, char **argv);
static int zfs_do_userspace(int argc, char **argv);
static int zfs_do_allow(int argc, char **argv);
static int zfs_do_unallow(int argc, char **argv);
static int zfs_do_hold(int argc, char **argv);
static int zfs_do_holds(int argc, char **argv);
static int zfs_do_release(int argc, char **argv);
static int zfs_do_diff(int argc, char **argv);

/*
 * Enable a reasonable set of defaults for libumem debugging on DEBUG builds.
 */

#ifdef DEBUG
const char *
_umem_debug_init(void)
{
        return ("default,verbose"); /* $UMEM_DEBUG setting */
}

const char *
_umem_logging_init(void)
{
        return ("fail,contents"); /* $UMEM_LOGGING setting */
}
#endif

typedef enum {
        HELP_CLONE,
        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_FITS_SEND,
137 #endif /* ! codereview */
138         HELP_SET,
139         HELP_SHARE,
140         HELP_SNAPSHOT,
141         HELP_UNMOUNT,
142         HELP_UNSHARE,
143         HELP_ALLOW,
144         HELP_UNALLOW,
145         HELP_USERSPACE,
146         HELP_GROUPSPACE,
147         HELP_HOLD,
148         HELP_HOLDS,
149         HELP_RELEASE,
150         HELP_DIFF,
151 } zfs_help_t;

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

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



```
191         { NULL },
192         { "send",       zfs_do_send,            HELP_SEND               },
193         { "receive",    zfs_do_receive,         HELP_RECEIVE            },
194         { NULL },
195         { "fits-send",  zfs_do_fits_send,       HELP_FITS_SEND          },
196         { NULL },
197 #endif /* ! codereview */
198         { "allow",      zfs_do_allow,           HELP_ALLOW              },
199         { NULL },
200         { "unallow",    zfs_do_unallow,         HELP_UNALLOW            },
201         { NULL },
202         { "hold",       zfs_do_hold,            HELP_HOLD               },
203         { "holds",      zfs_do_holds,           HELP_HOLDS              },
204         { "release",    zfs_do_release,         HELP_RELEASE            },
205         { "diff",       zfs_do_diff,            HELP_DIFF               },
206 };

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

210 zfs_command_t *current_command;

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

```
257                     "\trename [-f] -p <filesystem|volume> <filesystem|volume>\n"
258                     "\trename -r <snapshot> <snapshot>"));
259             case HELP_ROLLBACK:
260                     return (gettext("\trollback [-rRf] <snapshot>\n"));
261             case HELP_SEND:
262                     return (gettext("\tsend [-DnPpRv] [-[iI] snapshot] "
263                         "<snapshot>\n"));
264             case HELP_FITS_SEND:
265                     return (gettext("\tfits-send [-v] [-i snapshot] "
266                         "<snapshot>\n"));
267 #endif /* ! codereview */
268             case HELP_SET:
269                     return (gettext("\tset <property=value> "
270                         "<filesystem|volume|snapshot> ...\n"));
271             case HELP_SHARE:
272                     return (gettext("\tshare <-a | filesystem>\n"));
273             case HELP_SNAPSHOT:
274                     return (gettext("\tsnapshot [-r] [-o property=value] ... "
275                         "<filesystem@snapname|volume@snapname> ...\n"));
276             case HELP_UNMOUNT:
277                     return (gettext("\tunmount [-f] "
278                         "<-a | filesystem|mountpoint>\n"));
279             case HELP_UNSHARE:
280                     return (gettext("\tunshare "
281                         "<-a | filesystem|mountpoint>\n"));
282             case HELP_ALLOW:
283                     return (gettext("\tallow <filesystem|volume>\n"
284                         "\tallow [-ldug] "
285                         "<\"everyone\"|user|group>[,...] <perm|@setname>[,...]\n"
286                         "\t    <filesystem|volume>\n"
287                         "\tallow [-ld] -e <perm|@setname>[,...] "
288                         "<filesystem|volume>\n"
289                         "\tallow -c <perm|@setname>[,...] <filesystem|volume>\n"
290                         "\tallow -s @setname <perm|@setname>[,...] "
291                         "<filesystem|volume>\n"));
292             case HELP_UNALLOW:
293                     return (gettext("\tunallow [-rldug] "
294                         "<\"everyone\"|user|group>[,...]\n"
295                         "\t    [<perm|@setname>[,...]] <filesystem|volume>\n"
296                         "\tunallow [-rld] -e [<perm|@setname>[,...]] "
297                         "<filesystem|volume>\n"
298                         "\tunallow [-r] -c [<perm|@setname>[,...]] "
299                         "<filesystem|volume>\n"
300                         "\tunallow [-r] -s @setname [<perm|@setname>[,...]] "
301                         "<filesystem|volume>\n"));
302             case HELP_USERSPACE:
303                     return (gettext("\tuserspace [-Hinp] [-o field[,...]] "
304                         "[-s field] ...\n\t[-S field] ... "
305                         "[-t type[,...]] <filesystem|snapshot>\n"));
306             case HELP_GROUPSPACE:
307                     return (gettext("\tgroupspace [-Hinp] [-o field[,...]] "
308                         "[-s field] ...\n\t[-S field] ... "
309                         "[-t type[,...]] <filesystem|snapshot>\n"));
310             case HELP_HOLD:
311                     return (gettext("\thold [-r] <tag> <snapshot> ...\n"));
312             case HELP_HOLDS:
313                     return (gettext("\tholds [-r] <snapshot> ...\n"));
314             case HELP_RELEASE:
315                     return (gettext("\trelease [-r] <tag> <snapshot> ...\n"));
316             case HELP_DIFF:
317                     return (gettext("\tdiff [-FHt] <snapshot> "
318                         "[snapshot|filesystem]\n"));
319             }

321             abort();
322             /* NOTREACHED */
```



```
323 }

325 void
326 nomem(void)
327 {
328         (void) fprintf(stderr, gettext("internal error: out of memory\n"));
329         exit(1);
330 }

332 /*
333  * Utility function to guarantee malloc() success.
334  */

336 void *
337 safe_malloc(size_t size)
338 {
339         void *data;

341         if ((data = calloc(1, size)) == NULL)
342                 nomem();

344         return (data);
345 }

347 static char *
348 safe_strdup(char *str)
349 {
350         char *dupstr = strdup(str);

352         if (dupstr == NULL)
353                 nomem();

355         return (dupstr);
356 }

358 /*
359  * Callback routine that will print out information for each of
360  * the properties.
361  */
362 static int
363 usage_prop_cb(int prop, void *cb)
364 {
365         FILE *fp = cb;

367         (void) fprintf(fp, "\t%-15s ", zfs_prop_to_name(prop));

369         if (zfs_prop_readonly(prop))
370                 (void) fprintf(fp, " NO    ");
371         else
372                 (void) fprintf(fp, "YES    ");

374         if (zfs_prop_inheritable(prop))
375                 (void) fprintf(fp, "  YES   ");
376         else
377                 (void) fprintf(fp, "   NO   ");

379         if (zfs_prop_values(prop) == NULL)
380                 (void) fprintf(fp, "-\n");
381         else
382                 (void) fprintf(fp, "%s\n", zfs_prop_values(prop));

384         return (ZPROP_CONT);
385 }

387 /*
388  * Display usage message.  If we're inside a command, display only the usage for
```

```
 * that command.  Otherwise, iterate over the entire command table and display
 * a complete usage message.
 */
static void
usage(boolean_t requested)
{
        int i;
        boolean_t show_properties = B_FALSE;
        FILE *fp = requested ? stdout : stderr;

        if (current_command == NULL) {

                (void) fprintf(fp, gettext("usage: zfs command args ...\n"));
                (void) fprintf(fp,
                    gettext("where 'command' is one of the following:\n\n"));

                for (i = 0; i < NCOMMAND; i++) {
                        if (command_table[i].name == NULL)
                                (void) fprintf(fp, "\n");
                        else
                                (void) fprintf(fp, "%s",
                                    get_usage(command_table[i].usage));
                }

                (void) fprintf(fp, gettext("\nEach dataset is of the form: "
                    "pool/[dataset/]*dataset[@name]\n"));
        } else {
                (void) fprintf(fp, gettext("usage:\n"));
                (void) fprintf(fp, "%s", get_usage(current_command->usage));
        }

        if (current_command != NULL &&
            (strcmp(current_command->name, "set") == 0 ||
            strcmp(current_command->name, "get") == 0 ||
            strcmp(current_command->name, "inherit") == 0 ||
            strcmp(current_command->name, "list") == 0))
                show_properties = B_TRUE;

        if (show_properties) {
                (void) fprintf(fp,
                    gettext("\nThe following properties are supported:\n"));

                (void) fprintf(fp, "\n\t%-14s %s  %s   %s\n\n",
                    "PROPERTY", "EDIT", "INHERIT", "VALUES");

                /* Iterate over all properties */
                (void) zprop_iter(usage_prop_cb, fp, B_FALSE, B_TRUE,
                    ZFS_TYPE_DATASET);

                (void) fprintf(fp, "\t%-15s ", "userused@...");
                (void) fprintf(fp, " NO       NO   <size>\n");
                (void) fprintf(fp, "\t%-15s ", "groupused@...");
                (void) fprintf(fp, " NO       NO   <size>\n");
                (void) fprintf(fp, "\t%-15s ", "userquota@...");
                (void) fprintf(fp, "YES       NO   <size> | none\n");
                (void) fprintf(fp, "\t%-15s ", "groupquota@...");
                (void) fprintf(fp, "YES       NO   <size> | none\n");
                (void) fprintf(fp, "\t%-15s ", "written@<snap>");
                (void) fprintf(fp, " NO       NO   <size>\n");

                (void) fprintf(fp, gettext("\nSizes are specified in bytes "
                    "with standard units such as K, M, G, etc.\n"));
                (void) fprintf(fp, gettext("\nUser-defined properties can "
                    "be specified by using a name containing a colon (:).\n"));
                (void) fprintf(fp, gettext("\nThe {user|group}{used|quota}@ "
                    "properties must be appended with\n"
```



```
                    "a user or group specifier of one of these forms:\n"
                    "    POSIX name      (eg: \"matt\")\n"
                    "    POSIX id        (eg: \"126829\")\n"
                    "    SMB name@domain (eg: \"matt@sun\")\n"
                    "    SMB SID         (eg: \"S-1-234-567-89\")\n"));
        } else {
                (void) fprintf(fp,
                    gettext("\nFor the property list, run: %s\n"),
                    "zfs set|get");
                (void) fprintf(fp,
                    gettext("\nFor the delegated permission list, run: %s\n"),
                    "zfs allow|unallow");
        }

        /*
         * See comments at end of main().
         */
        if (getenv("ZFS_ABORT") != NULL) {
                (void) printf("dumping core by request\n");
                abort();
        }

        exit(requested ? 0 : 2);
}

static int
parseprop(nvlist_t *props)
{
        char *propname = optarg;
        char *propval, *strval;

        if ((propval = strchr(propname, '=')) == NULL) {
                (void) fprintf(stderr, gettext("missing "
                    "'=' for -o option\n"));
                return (-1);
        }
        *propval = '\0';
        propval++;
        if (nvlist_lookup_string(props, propname, &strval) == 0) {
                (void) fprintf(stderr, gettext("property '%s' "
                    "specified multiple times\n"), propname);
                return (-1);
        }
        if (nvlist_add_string(props, propname, propval) != 0)
                nomem();
        return (0);
}

static int
parse_depth(char *opt, int *flags)
{
        char *tmp;
        int depth;

        depth = (int)strtol(opt, &tmp, 0);
        if (*tmp) {
                (void) fprintf(stderr,
                    gettext("%s is not an integer\n"), optarg);
                usage(B_FALSE);
        }
        if (depth < 0) {
                (void) fprintf(stderr,
                    gettext("Depth can not be negative.\n"));
                usage(B_FALSE);
        }
        *flags |= (ZFS_ITER_DEPTH_LIMIT|ZFS_ITER_RECURSE);
```

```
521         return (depth);
522 }

524 #define PROGRESS_DELAY 2                /* seconds */

526 static char *pt_reverse = "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b";
527 static time_t pt_begin;
528 static char *pt_header = NULL;
529 static boolean_t pt_shown;

531 static void
532 start_progress_timer(void)
533 {
534         pt_begin = time(NULL) + PROGRESS_DELAY;
535         pt_shown = B_FALSE;
536 }

538 static void
539 set_progress_header(char *header)
540 {
541         assert(pt_header == NULL);
542         pt_header = safe_strdup(header);
543         if (pt_shown) {
544                 (void) printf("%s: ", header);
545                 (void) fflush(stdout);
546         }
547 }

549 static void
550 update_progress(char *update)
551 {
552         if (!pt_shown && time(NULL) > pt_begin) {
553                 int len = strlen(update);

555                 (void) printf("%s: %s%*.*s", pt_header, update, len, len,
556                     pt_reverse);
557                 (void) fflush(stdout);
558                 pt_shown = B_TRUE;
559         } else if (pt_shown) {
560                 int len = strlen(update);

562                 (void) printf("%s%*.*s", update, len, len, pt_reverse);
563                 (void) fflush(stdout);
564         }
565 }

567 static void
568 finish_progress(char *done)
569 {
570         if (pt_shown) {
571                 (void) printf("%s\n", done);
572                 (void) fflush(stdout);
573         }
574         free(pt_header);
575         pt_header = NULL;
576 }
577 /*
578  * zfs clone [-p] [-o prop=value] ... <snap> <fs | vol>
579  *
580  * Given an existing dataset, create a writable copy whose initial contents
581  * are the same as the source.  The newly created dataset maintains a
582  * dependency on the original; the original cannot be destroyed so long as
583  * the clone exists.
584  *
585  * The '-p' flag creates all the non-existing ancestors of the target first.
586  */
```



```
587 static int
588 zfs_do_clone(int argc, char **argv)
589 {
590         zfs_handle_t *zhp = NULL;
591         boolean_t parents = B_FALSE;
592         nvlist_t *props;
593         int ret = 0;
594         int c;

596         if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
597                 nomem();

599         /* check options */
600         while ((c = getopt(argc, argv, "o:p")) != -1) {
601                 switch (c) {
602                 case 'o':
603                         if (parseprop(props))
604                                 return (1);
605                         break;
606                 case 'p':
607                         parents = B_TRUE;
608                         break;
609                 case '?':
610                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
611                             optopt);
612                         goto usage;
613                 }
614         }

616         argc -= optind;
617         argv += optind;

619         /* check number of arguments */
620         if (argc < 1) {
621                 (void) fprintf(stderr, gettext("missing source dataset "
622                     "argument\n"));
623                 goto usage;
624         }
625         if (argc < 2) {
626                 (void) fprintf(stderr, gettext("missing target dataset "
627                     "argument\n"));
628                 goto usage;
629         }
630         if (argc > 2) {
631                 (void) fprintf(stderr, gettext("too many arguments\n"));
632                 goto usage;
633         }

635         /* open the source dataset */
636         if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_SNAPSHOT)) == NULL)
637                 return (1);

639         if (parents && zfs_name_valid(argv[1], ZFS_TYPE_FILESYSTEM |
640             ZFS_TYPE_VOLUME)) {
641                 /*
642                  * Now create the ancestors of the target dataset.  If the
643                  * target already exists and '-p' option was used we should not
644                  * complain.
645                  */
646                 if (zfs_dataset_exists(g_zfs, argv[1], ZFS_TYPE_FILESYSTEM |
647                     ZFS_TYPE_VOLUME))
648                         return (0);
649                 if (zfs_create_ancestors(g_zfs, argv[1]) != 0)
650                         return (1);
651         }
```

```
653         /* pass to libzfs */
654         ret = zfs_clone(zhp, argv[1], props);

656         /* create the mountpoint if necessary */
657         if (ret == 0) {
658                 zfs_handle_t *clone;

660                 clone = zfs_open(g_zfs, argv[1], ZFS_TYPE_DATASET);
661                 if (clone != NULL) {
662                         if (zfs_get_type(clone) != ZFS_TYPE_VOLUME)
663                                 if ((ret = zfs_mount(clone, NULL, 0)) == 0)
664                                         ret = zfs_share(clone);
665                         zfs_close(clone);
666                 }
667         }

669         zfs_close(zhp);
670         nvlist_free(props);

672         return (!!ret);

674 usage:
675         if (zhp)
676                 zfs_close(zhp);
677         nvlist_free(props);
678         usage(B_FALSE);
679         return (-1);
680 }

682 /*
683  * zfs create [-p] [-o prop=value] ... fs
684  * zfs create [-ps] [-b blocksize] [-o prop=value] ... -V vol size
685  *
686  * Create a new dataset.  This command can be used to create filesystems
687  * and volumes.  Snapshot creation is handled by 'zfs snapshot'.
688  * For volumes, the user must specify a size to be used.
689  *
690  * The '-s' flag applies only to volumes, and indicates that we should not try
691  * to set the reservation for this volume.  By default we set a reservation
692  * equal to the size for any volume.  For pools with SPA_VERSION >=
693  * SPA_VERSION_REFRESERVATION, we set a refreservation instead.
694  *
695  * The '-p' flag creates all the non-existing ancestors of the target first.
696  */
697 static int
698 zfs_do_create(int argc, char **argv)
699 {
700         zfs_type_t type = ZFS_TYPE_FILESYSTEM;
701         zfs_handle_t *zhp = NULL;
702         uint64_t volsize;
703         int c;
704         boolean_t noreserve = B_FALSE;
705         boolean_t bflag = B_FALSE;
706         boolean_t parents = B_FALSE;
707         int ret = 1;
708         nvlist_t *props;
709         uint64_t intval;
710         int canmount = ZFS_CANMOUNT_OFF;

712         if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
713                 nomem();

715         /* check options */
716         while ((c = getopt(argc, argv, ":V:b:so:p")) != -1) {
717                 switch (c) {
718                 case 'V':
```



```
719                         type = ZFS_TYPE_VOLUME;
720                         if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
721                                 (void) fprintf(stderr, gettext("bad volume "
722                                     "size '%s': %s\n"), optarg,
723                                     libzfs_error_description(g_zfs));
724                                 goto error;
725                         }

727                         if (nvlist_add_uint64(props,
728                             zfs_prop_to_name(ZFS_PROP_VOLSIZE), intval) != 0)
729                                 nomem();
730                         volsize = intval;
731                         break;
732                 case 'p':
733                         parents = B_TRUE;
734                         break;
735                 case 'b':
736                         bflag = B_TRUE;
737                         if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
738                                 (void) fprintf(stderr, gettext("bad volume "
739                                     "block size '%s': %s\n"), optarg,
740                                     libzfs_error_description(g_zfs));
741                                 goto error;
742                         }

744                         if (nvlist_add_uint64(props,
745                             zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
746                             intval) != 0)
747                                 nomem();
748                         break;
749                 case 'o':
750                         if (parseprop(props))
751                                 goto error;
752                         break;
753                 case 's':
754                         noreserve = B_TRUE;
755                         break;
756                 case ':':
757                         (void) fprintf(stderr, gettext("missing size "
758                             "argument\n"));
759                         goto badusage;
760                 case '?':
761                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
762                             optopt);
763                         goto badusage;
764                 }
765         }

767         if ((bflag || noreserve) && type != ZFS_TYPE_VOLUME) {
768                 (void) fprintf(stderr, gettext("'-s' and '-b' can only be "
769                     "used when creating a volume\n"));
770                 goto badusage;
771         }

773         argc -= optind;
774         argv += optind;

776         /* check number of arguments */
777         if (argc == 0) {
778                 (void) fprintf(stderr, gettext("missing %s argument\n"),
779                     zfs_type_to_name(type));
780                 goto badusage;
781         }
782         if (argc > 1) {
783                 (void) fprintf(stderr, gettext("too many arguments\n"));
784                 goto badusage;
```

```
785         }
786
787         if (type == ZFS_TYPE_VOLUME && !noreserve) {
788                 zpool_handle_t *zpool_handle;
789                 uint64_t spa_version;
790                 char *p;
791                 zfs_prop_t resv_prop;
792                 char *strval;
793
794                 if (p = strchr(argv[0], '/'))
795                         *p = '\0';
796                 zpool_handle = zpool_open(g_zfs, argv[0]);
797                 if (p != NULL)
798                         *p = '/';
799                 if (zpool_handle == NULL)
800                         goto error;
801                 spa_version = zpool_get_prop_int(zpool_handle,
802                     ZPOOL_PROP_VERSION, NULL);
803                 zpool_close(zpool_handle);
804                 if (spa_version >= SPA_VERSION_REFRESERVATION)
805                         resv_prop = ZFS_PROP_REFRESERVATION;
806                 else
807                         resv_prop = ZFS_PROP_RESERVATION;
808                 volsize = zvol_volsize_to_reservation(volsize, props);
809
810                 if (nvlist_lookup_string(props, zfs_prop_to_name(resv_prop),
811                     &strval) != 0) {
812                         if (nvlist_add_uint64(props,
813                             zfs_prop_to_name(resv_prop), volsize) != 0) {
814                                 nvlist_free(props);
815                                 nomem();
816                         }
817                 }
818         }
819
820         if (parents && zfs_name_valid(argv[0], type)) {
821                 /*
822                  * Now create the ancestors of target dataset.  If the target
823                  * already exists and '-p' option was used we should not
824                  * complain.
825                  */
826                 if (zfs_dataset_exists(g_zfs, argv[0], type)) {
827                         ret = 0;
828                         goto error;
829                 }
830                 if (zfs_create_ancestors(g_zfs, argv[0]) != 0)
831                         goto error;
832         }
833
834         /* pass to libzfs */
835         if (zfs_create(g_zfs, argv[0], type, props) != 0)
836                 goto error;
837
838         if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_DATASET)) == NULL)
839                 goto error;
840
841         ret = 0;
842         /*
843          * if the user doesn't want the dataset automatically mounted,
844          * then skip the mount/share step
845          */
846         if (zfs_prop_valid_for_type(ZFS_PROP_CANMOUNT, type))
847                 canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);
848
849         /*
850          * Mount and/or share the new filesystem as appropriate.  We provide a
```



```
851          * verbose error message to let the user know that their filesystem was
852          * in fact created, even if we failed to mount or share it.
853          */
854         if (canmount == ZFS_CANMOUNT_ON) {
855                 if (zfs_mount(zhp, NULL, 0) != 0) {
856                         (void) fprintf(stderr, gettext("filesystem "
857                             "successfully created, but not mounted\n"));
858                         ret = 1;
859                 } else if (zfs_share(zhp) != 0) {
860                         (void) fprintf(stderr, gettext("filesystem "
861                             "successfully created, but not shared\n"));
862                         ret = 1;
863                 }
864         }
865
866 error:
867         if (zhp)
868                 zfs_close(zhp);
869         nvlist_free(props);
870         return (ret);
871 badusage:
872         nvlist_free(props);
873         usage(B_FALSE);
874         return (2);
875 }
876
877 /*
878  * zfs destroy [-rRf] <fs, vol>
879  * zfs destroy [-rRd] <snap>
880  *
881  *      -r      Recursively destroy all children
882  *      -R      Recursively destroy all dependents, including clones
883  *      -f      Force unmounting of any dependents
884  *      -d      If we can't destroy now, mark for deferred destruction
885  *
886  * Destroys the given dataset.  By default, it will unmount any filesystems,
887  * and refuse to destroy a dataset that has any dependents.  A dependent can
888  * either be a child, or a clone of a child.
889  */
890 typedef struct destroy_cbdata {
891         boolean_t       cb_first;
892         boolean_t       cb_force;
893         boolean_t       cb_recurse;
894         boolean_t       cb_error;
895         boolean_t       cb_doclones;
896         zfs_handle_t    *cb_target;
897         boolean_t       cb_defer_destroy;
898         boolean_t       cb_verbose;
899         boolean_t       cb_parsable;
900         boolean_t       cb_dryrun;
901         nvlist_t        *cb_nvl;
902
903         /* first snap in contiguous run */
904         char            *cb_firstsnap;
905         /* previous snap in contiguous run */
906         char            *cb_prevsnap;
907         int64_t         cb_snapused;
908         char            *cb_snapspec;
909 } destroy_cbdata_t;
910
911 /*
912  * Check for any dependents based on the '-r' or '-R' flags.
913  */
914 static int
915 destroy_check_dependent(zfs_handle_t *zhp, void *data)
916 {
```

```
917         destroy_cbdata_t *cbp = data;
918         const char *tname = zfs_get_name(cbp->cb_target);
919         const char *name = zfs_get_name(zhp);

921         if (strncmp(tname, name, strlen(tname)) == 0 &&
922             (name[strlen(tname)] == '/' || name[strlen(tname)] == '@')) {
923                 /*
924                  * This is a direct descendant, not a clone somewhere else in
925                  * the hierarchy.
926                  */
927                 if (cbp->cb_recurse)
928                         goto out;

930                 if (cbp->cb_first) {
931                         (void) fprintf(stderr, gettext("cannot destroy '%s': "
932                             "%s has children\n"),
933                             zfs_get_name(cbp->cb_target),
934                             zfs_type_to_name(zfs_get_type(cbp->cb_target)));
935                         (void) fprintf(stderr, gettext("use '-r' to destroy "
936                             "the following datasets:\n"));
937                         cbp->cb_first = B_FALSE;
938                         cbp->cb_error = B_TRUE;
939                 }

941                 (void) fprintf(stderr, "%s\n", zfs_get_name(zhp));
942         } else {
943                 /*
944                  * This is a clone.  We only want to report this if the '-r'
945                  * wasn't specified, or the target is a snapshot.
946                  */
947                 if (!cbp->cb_recurse &&
948                     zfs_get_type(cbp->cb_target) != ZFS_TYPE_SNAPSHOT)
949                         goto out;

951                 if (cbp->cb_first) {
952                         (void) fprintf(stderr, gettext("cannot destroy '%s': "
953                             "%s has dependent clones\n"),
954                             zfs_get_name(cbp->cb_target),
955                             zfs_type_to_name(zfs_get_type(cbp->cb_target)));
956                         (void) fprintf(stderr, gettext("use '-R' to destroy "
957                             "the following datasets:\n"));
958                         cbp->cb_first = B_FALSE;
959                         cbp->cb_error = B_TRUE;
960                         cbp->cb_dryrun = B_TRUE;
961                 }

963                 (void) fprintf(stderr, "%s\n", zfs_get_name(zhp));
964         }

966 out:
967         zfs_close(zhp);
968         return (0);
969 }

971 static int
972 destroy_callback(zfs_handle_t *zhp, void *data)
973 {
974         destroy_cbdata_t *cb = data;
975         const char *name = zfs_get_name(zhp);

977         if (cb->cb_verbose) {
978                 if (cb->cb_parsable) {
979                         (void) printf("destroy\t%s\n", name);
980                 } else if (cb->cb_dryrun) {
981                         (void) printf(gettext("would destroy %s\n"),
982                             name);
```



```
983                 } else {
984                         (void) printf(gettext("will destroy %s\n"),
985                             name);
986                 }
987         }

989         /*
990          * Ignore pools (which we've already flagged as an error before getting
991          * here).
992          */
993         if (strchr(zfs_get_name(zhp), '/') == NULL &&
994             zfs_get_type(zhp) == ZFS_TYPE_FILESYSTEM) {
995                 zfs_close(zhp);
996                 return (0);
997         }

999         if (!cb->cb_dryrun) {
1000                if (zfs_unmount(zhp, NULL, cb->cb_force ? MS_FORCE : 0) != 0 ||
1001                    zfs_destroy(zhp, cb->cb_defer_destroy) != 0) {
1002                        zfs_close(zhp);
1003                        return (-1);
1004                }
1005        }

1007        zfs_close(zhp);
1008        return (0);
1009 }

1011 static int
1012 destroy_print_cb(zfs_handle_t *zhp, void *arg)
1013 {
1014        destroy_cbdata_t *cb = arg;
1015        const char *name = zfs_get_name(zhp);
1016        int err = 0;

1018        if (nvlist_exists(cb->cb_nvl, name)) {
1019                if (cb->cb_firstsnap == NULL)
1020                        cb->cb_firstsnap = strdup(name);
1021                if (cb->cb_prevsnap != NULL)
1022                        free(cb->cb_prevsnap);
1023                /* this snap continues the current range */
1024                cb->cb_prevsnap = strdup(name);
1025                if (cb->cb_firstsnap == NULL || cb->cb_prevsnap == NULL)
1026                        nomem();
1027                if (cb->cb_verbose) {
1028                        if (cb->cb_parsable) {
1029                                (void) printf("destroy\t%s\n", name);
1030                        } else if (cb->cb_dryrun) {
1031                                (void) printf(gettext("would destroy %s\n"),
1032                                    name);
1033                        } else {
1034                                (void) printf(gettext("will destroy %s\n"),
1035                                    name);
1036                        }
1037                }
1038        } else if (cb->cb_firstsnap != NULL) {
1039                /* end of this range */
1040                uint64_t used = 0;
1041                err = lzc_snaprange_space(cb->cb_firstsnap,
1042                    cb->cb_prevsnap, &used);
1043                cb->cb_snapused += used;
1044                free(cb->cb_firstsnap);
1045                cb->cb_firstsnap = NULL;
1046                free(cb->cb_prevsnap);
1047                cb->cb_prevsnap = NULL;
1048        }
```

```
1049         zfs_close(zhp);
1050         return (err);
1051 }

1053 static int
1054 destroy_print_snapshots(zfs_handle_t *fs_zhp, destroy_cbdata_t *cb)
1055 {
1056         int err = 0;
1057         assert(cb->cb_firstsnap == NULL);
1058         assert(cb->cb_prevsnap == NULL);
1059         err = zfs_iter_snapshots_sorted(fs_zhp, destroy_print_cb, cb);
1060         if (cb->cb_firstsnap != NULL) {
1061                 uint64_t used = 0;
1062                 if (err == 0) {
1063                         err = lzc_snaprange_space(cb->cb_firstsnap,
1064                             cb->cb_prevsnap, &used);
1065                 }
1066                 cb->cb_snapused += used;
1067                 free(cb->cb_firstsnap);
1068                 cb->cb_firstsnap = NULL;
1069                 free(cb->cb_prevsnap);
1070                 cb->cb_prevsnap = NULL;
1071         }
1072         return (err);
1073 }

1075 static int
1076 snapshot_to_nvl_cb(zfs_handle_t *zhp, void *arg)
1077 {
1078         destroy_cbdata_t *cb = arg;
1079         int err = 0;

1081         /* Check for clones. */
1082         if (!cb->cb_doclones && !cb->cb_defer_destroy) {
1083                 cb->cb_target = zhp;
1084                 cb->cb_first = B_TRUE;
1085                 err = zfs_iter_dependents(zhp, B_TRUE,
1086                     destroy_check_dependent, cb);
1087         }

1089         if (err == 0) {
1090                 if (nvlist_add_boolean(cb->cb_nvl, zfs_get_name(zhp)))
1091                         nomem();
1092         }
1093         zfs_close(zhp);
1094         return (err);
1095 }

1097 static int
1098 gather_snapshots(zfs_handle_t *zhp, void *arg)
1099 {
1100         destroy_cbdata_t *cb = arg;
1101         int err = 0;

1103         err = zfs_iter_snapspec(zhp, cb->cb_snapspec, snapshot_to_nvl_cb, cb);
1104         if (err == ENOENT)
1105                 err = 0;
1106         if (err != 0)
1107                 goto out;

1109         if (cb->cb_verbose) {
1110                 err = destroy_print_snapshots(zhp, cb);
1111                 if (err != 0)
1112                         goto out;
1113         }
```



```
1115         if (cb->cb_recurse)
1116                 err = zfs_iter_filesystems(zhp, gather_snapshots, cb);

1118 out:
1119         zfs_close(zhp);
1120         return (err);
1121 }

1123 static int
1124 destroy_clones(destroy_cbdata_t *cb)
1125 {
1126         nvpair_t *pair;
1127         for (pair = nvlist_next_nvpair(cb->cb_nvl, NULL);
1128             pair != NULL;
1129             pair = nvlist_next_nvpair(cb->cb_nvl, pair)) {
1130                 zfs_handle_t *zhp = zfs_open(g_zfs, nvpair_name(pair),
1131                     ZFS_TYPE_SNAPSHOT);
1132                 if (zhp != NULL) {
1133                         boolean_t defer = cb->cb_defer_destroy;
1134                         int err = 0;

1136                         /*
1137                          * We can't defer destroy non-snapshots, so set it to
1138                          * false while destroying the clones.
1139                          */
1140                         cb->cb_defer_destroy = B_FALSE;
1141                         err = zfs_iter_dependents(zhp, B_FALSE,
1142                             destroy_callback, cb);
1143                         cb->cb_defer_destroy = defer;
1144                         zfs_close(zhp);
1145                         if (err != 0)
1146                                 return (err);
1147                 }
1148         }
1149         return (0);
1150 }

1152 static int
1153 zfs_do_destroy(int argc, char **argv)
1154 {
1155         destroy_cbdata_t cb = { 0 };
1156         int c;
1157         zfs_handle_t *zhp;
1158         char *at;
1159         zfs_type_t type = ZFS_TYPE_DATASET;

1161         /* check options */
1162         while ((c = getopt(argc, argv, "vpndfrR")) != -1) {
1163                 switch (c) {
1164                 case 'v':
1165                         cb.cb_verbose = B_TRUE;
1166                         break;
1167                 case 'p':
1168                         cb.cb_verbose = B_TRUE;
1169                         cb.cb_parsable = B_TRUE;
1170                         break;
1171                 case 'n':
1172                         cb.cb_dryrun = B_TRUE;
1173                         break;
1174                 case 'd':
1175                         cb.cb_defer_destroy = B_TRUE;
1176                         type = ZFS_TYPE_SNAPSHOT;
1177                         break;
1178                 case 'f':
1179                         cb.cb_force = B_TRUE;
1180                         break;
```

```
1181                 case 'r':
1182                         cb.cb_recurse = B_TRUE;
1183                         break;
1184                 case 'R':
1185                         cb.cb_recurse = B_TRUE;
1186                         cb.cb_doclones = B_TRUE;
1187                         break;
1188                 case '?':
1189                 default:
1190                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1191                             optopt);
1192                         usage(B_FALSE);
1193                 }
1194         }

1196         argc -= optind;
1197         argv += optind;

1199         /* check number of arguments */
1200         if (argc == 0) {
1201                 (void) fprintf(stderr, gettext("missing dataset argument\n"));
1202                 usage(B_FALSE);
1203         }
1204         if (argc > 1) {
1205                 (void) fprintf(stderr, gettext("too many arguments\n"));
1206                 usage(B_FALSE);
1207         }

1209         at = strchr(argv[0], '@');
1210         if (at != NULL) {
1211                 int err = 0;

1213                 /* Build the list of snaps to destroy in cb_nvl. */
1214                 if (nvlist_alloc(&cb.cb_nvl, NV_UNIQUE_NAME, 0) != 0)
1215                         nomem();

1217                 *at = '\0';
1218                 zhp = zfs_open(g_zfs, argv[0],
1219                     ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
1220                 if (zhp == NULL)
1221                         return (1);

1223                 cb.cb_snapspec = at + 1;
1224                 if (gather_snapshots(zfs_handle_dup(zhp), &cb) != 0 ||
1225                     cb.cb_error) {
1226                         zfs_close(zhp);
1227                         nvlist_free(cb.cb_nvl);
1228                         return (1);
1229                 }

1231                 if (nvlist_empty(cb.cb_nvl)) {
1232                         (void) fprintf(stderr, gettext("could not find any "
1233                             "snapshots to destroy; check snapshot names.\n"));
1234                         zfs_close(zhp);
1235                         nvlist_free(cb.cb_nvl);
1236                         return (1);
1237                 }

1239                 if (cb.cb_verbose) {
1240                         char buf[16];
1241                         zfs_nicenum(cb.cb_snapused, buf, sizeof (buf));
1242                         if (cb.cb_parsable) {
1243                                 (void) printf("reclaim\t%llu\n",
1244                                     cb.cb_snapused);
1245                         } else if (cb.cb_dryrun) {
1246                                 (void) printf(gettext("would reclaim %s\n"),
```



```
1247                                     buf);
1248                         } else {
1249                                 (void) printf(gettext("will reclaim %s\n"),
1250                                     buf);
1251                         }
1252                 }

1254                 if (!cb.cb_dryrun) {
1255                         if (cb.cb_doclones)
1256                                 err = destroy_clones(&cb);
1257                         if (err == 0) {
1258                                 err = zfs_destroy_snaps_nvl(zhp, cb.cb_nvl,
1259                                     cb.cb_defer_destroy);
1260                         }
1261                 }

1263                 zfs_close(zhp);
1264                 nvlist_free(cb.cb_nvl);
1265                 if (err != 0)
1266                         return (1);
1267         } else {
1268                 /* Open the given dataset */
1269                 if ((zhp = zfs_open(g_zfs, argv[0], type)) == NULL)
1270                         return (1);

1272                 cb.cb_target = zhp;

1274                 /*
1275                  * Perform an explicit check for pools before going any further.
1276                  */
1277                 if (!cb.cb_recurse && strchr(zfs_get_name(zhp), '/') == NULL &&
1278                     zfs_get_type(zhp) == ZFS_TYPE_FILESYSTEM) {
1279                         (void) fprintf(stderr, gettext("cannot destroy '%s': "
1280                             "operation does not apply to pools\n"),
1281                             zfs_get_name(zhp));
1282                         (void) fprintf(stderr, gettext("use 'zfs destroy -r "
1283                             "%s' to destroy all datasets in the pool\n"),
1284                             zfs_get_name(zhp));
1285                         (void) fprintf(stderr, gettext("use 'zpool destroy %s' "
1286                             "to destroy the pool itself\n"), zfs_get_name(zhp));
1287                         zfs_close(zhp);
1288                         return (1);
1289                 }

1291                 /*
1292                  * Check for any dependents and/or clones.
1293                  */
1294                 cb.cb_first = B_TRUE;
1295                 if (!cb.cb_doclones &&
1296                     zfs_iter_dependents(zhp, B_TRUE, destroy_check_dependent,
1297                     &cb) != 0) {
1298                         zfs_close(zhp);
1299                         return (1);
1300                 }

1302                 if (cb.cb_error) {
1303                         zfs_close(zhp);
1304                         return (1);
1305                 }

1307                 if (zfs_iter_dependents(zhp, B_FALSE, destroy_callback,
1308                     &cb) != 0) {
1309                         zfs_close(zhp);
1310                         return (1);
1311                 }
```

```
1313                 /*
1314                  * Do the real thing.  The callback will close the
1315                  * handle regardless of whether it succeeds or not.
1316                  */
1317                 if (destroy_callback(zhp, &cb) != 0)
1318                         return (1);
1319         }

1321         return (0);
1322 }

1324 static boolean_t
1325 is_recvd_column(zprop_get_cbdata_t *cbp)
1326 {
1327         int i;
1328         zfs_get_column_t col;

1330         for (i = 0; i < ZFS_GET_NCOLS &&
1331             (col = cbp->cb_columns[i]) != GET_COL_NONE; i++)
1332                 if (col == GET_COL_RECVD)
1333                         return (B_TRUE);
1334         return (B_FALSE);
1335 }

1337 /*
1338  * zfs get [-rHp] [-o all | field[,field]...] [-s source[,source]...]
1339  *      < all | property[,property]... > < fs | snap | vol > ...
1340  *
1341  *      -r      recurse over any child datasets
1342  *      -H      scripted mode.  Headers are stripped, and fields are separated
1343  *              by tabs instead of spaces.
1344  *      -o      Set of fields to display.  One of "name,property,value,
1345  *              received,source". Default is "name,property,value,source".
1346  *              "all" is an alias for all five.
1347  *      -s      Set of sources to allow.  One of
1348  *              "local,default,inherited,received,temporary,none".  Default is
1349  *              all six.
1350  *      -p      Display values in parsable (literal) format.
1351  *
1352  *  Prints properties for the given datasets.  The user can control which
1353  *  columns to display as well as which property types to allow.
1354  */

1356 /*
1357  * Invoked to display the properties for a single dataset.
1358  */
1359 static int
1360 get_callback(zfs_handle_t *zhp, void *data)
1361 {
1362         char buf[ZFS_MAXPROPLEN];
1363         char rbuf[ZFS_MAXPROPLEN];
1364         zprop_source_t sourcetype;
1365         char source[ZFS_MAXNAMELEN];
1366         zprop_get_cbdata_t *cbp = data;
1367         nvlist_t *user_props = zfs_get_user_props(zhp);
1368         zprop_list_t *pl = cbp->cb_proplist;
1369         nvlist_t *propval;
1370         char *strval;
1371         char *sourceval;
1372         boolean_t received = is_recvd_column(cbp);

1374         for (; pl != NULL; pl = pl->pl_next) {
1375                 char *recvdval = NULL;
1376                 /*
1377                  * Skip the special fake placeholder.  This will also skip over
1378                  * the name property when 'all' is specified.
```



```
1379                  */
1380                 if (pl->pl_prop == ZFS_PROP_NAME &&
1381                     pl == cbp->cb_proplist)
1382                         continue;

1384                 if (pl->pl_prop != ZPROP_INVAL) {
1385                         if (zfs_prop_get(zhp, pl->pl_prop, buf,
1386                             sizeof (buf), &sourcetype, source,
1387                             sizeof (source),
1388                             cbp->cb_literal) != 0) {
1389                                 if (pl->pl_all)
1390                                         continue;
1391                                 if (!zfs_prop_valid_for_type(pl->pl_prop,
1392                                     ZFS_TYPE_DATASET)) {
1393                                         (void) fprintf(stderr,
1394                                             gettext("No such property '%s'\n"),
1395                                             zfs_prop_to_name(pl->pl_prop));
1396                                         continue;
1397                                 }
1398                                 sourcetype = ZPROP_SRC_NONE;
1399                                 (void) strlcpy(buf, "-", sizeof (buf));
1400                         }

1402                         if (received && (zfs_prop_get_recvd(zhp,
1403                             zfs_prop_to_name(pl->pl_prop), rbuf, sizeof (rbuf),
1404                             cbp->cb_literal) == 0))
1405                                 recvdval = rbuf;

1407                         zprop_print_one_property(zfs_get_name(zhp), cbp,
1408                             zfs_prop_to_name(pl->pl_prop),
1409                             buf, sourcetype, source, recvdval);
1410                 } else if (zfs_prop_userquota(pl->pl_user_prop)) {
1411                         sourcetype = ZPROP_SRC_LOCAL;

1413                         if (zfs_prop_get_userquota(zhp, pl->pl_user_prop,
1414                             buf, sizeof (buf), cbp->cb_literal) != 0) {
1415                                 sourcetype = ZPROP_SRC_NONE;
1416                                 (void) strlcpy(buf, "-", sizeof (buf));
1417                         }

1419                         zprop_print_one_property(zfs_get_name(zhp), cbp,
1420                             pl->pl_user_prop, buf, sourcetype, source, NULL);
1421                 } else if (zfs_prop_written(pl->pl_user_prop)) {
1422                         sourcetype = ZPROP_SRC_LOCAL;

1424                         if (zfs_prop_get_written(zhp, pl->pl_user_prop,
1425                             buf, sizeof (buf), cbp->cb_literal) != 0) {
1426                                 sourcetype = ZPROP_SRC_NONE;
1427                                 (void) strlcpy(buf, "-", sizeof (buf));
1428                         }

1430                         zprop_print_one_property(zfs_get_name(zhp), cbp,
1431                             pl->pl_user_prop, buf, sourcetype, source, NULL);
1432                 } else {
1433                         if (nvlist_lookup_nvlist(user_props,
1434                             pl->pl_user_prop, &propval) != 0) {
1435                                 if (pl->pl_all)
1436                                         continue;
1437                                 sourcetype = ZPROP_SRC_NONE;
1438                                 strval = "-";
1439                         } else {
1440                                 verify(nvlist_lookup_string(propval,
1441                                     ZPROP_VALUE, &strval) == 0);
1442                                 verify(nvlist_lookup_string(propval,
1443                                     ZPROP_SOURCE, &sourceval) == 0);
```

```
1445                                if (strcmp(sourceval,
1446                                    zfs_get_name(zhp)) == 0) {
1447                                        sourcetype = ZPROP_SRC_LOCAL;
1448                                } else if (strcmp(sourceval,
1449                                    ZPROP_SOURCE_VAL_RECVD) == 0) {
1450                                        sourcetype = ZPROP_SRC_RECEIVED;
1451                                } else {
1452                                        sourcetype = ZPROP_SRC_INHERITED;
1453                                        (void) strlcpy(source,
1454                                            sourceval, sizeof (source));
1455                                }
1456                        }

1458                        if (received && (zfs_prop_get_recvd(zhp,
1459                            pl->pl_user_prop, rbuf, sizeof (rbuf),
1460                            cbp->cb_literal) == 0))
1461                                recvdval = rbuf;

1463                        zprop_print_one_property(zfs_get_name(zhp), cbp,
1464                            pl->pl_user_prop, strval, sourcetype,
1465                            source, recvdval);
1466                }
1467        }

1469        return (0);
1470 }

1472 static int
1473 zfs_do_get(int argc, char **argv)
1474 {
1475        zprop_get_cbdata_t cb = { 0 };
1476        int i, c, flags = ZFS_ITER_ARGS_CAN_BE_PATHS;
1477        int types = ZFS_TYPE_DATASET;
1478        char *value, *fields;
1479        int ret = 0;
1480        int limit = 0;
1481        zprop_list_t fake_name = { 0 };

1483        /*
1484         * Set up default columns and sources.
1485         */
1486        cb.cb_sources = ZPROP_SRC_ALL;
1487        cb.cb_columns[0] = GET_COL_NAME;
1488        cb.cb_columns[1] = GET_COL_PROPERTY;
1489        cb.cb_columns[2] = GET_COL_VALUE;
1490        cb.cb_columns[3] = GET_COL_SOURCE;
1491        cb.cb_type = ZFS_TYPE_DATASET;

1493        /* check options */
1494        while ((c = getopt(argc, argv, ":d:o:s:rt:Hp")) != -1) {
1495                switch (c) {
1496                case 'p':
1497                        cb.cb_literal = B_TRUE;
1498                        break;
1499                case 'd':
1500                        limit = parse_depth(optarg, &flags);
1501                        break;
1502                case 'r':
1503                        flags |= ZFS_ITER_RECURSE;
1504                        break;
1505                case 'H':
1506                        cb.cb_scripted = B_TRUE;
1507                        break;
1508                case ':':
1509                        (void) fprintf(stderr, gettext("missing argument for "
1510                            "'%c' option\n"), optopt);
```



```
1511                        usage(B_FALSE);
1512                        break;
1513                case 'o':
1514                        /*
1515                         * Process the set of columns to display.  We zero out
1516                         * the structure to give us a blank slate.
1517                         */
1518                        bzero(&cb.cb_columns, sizeof (cb.cb_columns));
1519                        i = 0;
1520                        while (*optarg != '\0') {
1521                                static char *col_subopts[] =
1522                                    { "name", "property", "value", "received",
1523                                    "source", "all", NULL };

1525                                if (i == ZFS_GET_NCOLS) {
1526                                        (void) fprintf(stderr, gettext("too "
1527                                            "many fields given to -o "
1528                                            "option\n"));
1529                                        usage(B_FALSE);
1530                                }

1532                                switch (getsubopt(&optarg, col_subopts,
1533                                    &value)) {
1534                                case 0:
1535                                        cb.cb_columns[i++] = GET_COL_NAME;
1536                                        break;
1537                                case 1:
1538                                        cb.cb_columns[i++] = GET_COL_PROPERTY;
1539                                        break;
1540                                case 2:
1541                                        cb.cb_columns[i++] = GET_COL_VALUE;
1542                                        break;
1543                                case 3:
1544                                        cb.cb_columns[i++] = GET_COL_RECVD;
1545                                        flags |= ZFS_ITER_RECVD_PROPS;
1546                                        break;
1547                                case 4:
1548                                        cb.cb_columns[i++] = GET_COL_SOURCE;
1549                                        break;
1550                                case 5:
1551                                        if (i > 0) {
1552                                                (void) fprintf(stderr,
1553                                                    gettext("\"all\" conflicts "
1554                                                    "with specific fields "
1555                                                    "given to -o option\n"));
1556                                                usage(B_FALSE);
1557                                        }
1558                                        cb.cb_columns[0] = GET_COL_NAME;
1559                                        cb.cb_columns[1] = GET_COL_PROPERTY;
1560                                        cb.cb_columns[2] = GET_COL_VALUE;
1561                                        cb.cb_columns[3] = GET_COL_RECVD;
1562                                        cb.cb_columns[4] = GET_COL_SOURCE;
1563                                        flags |= ZFS_ITER_RECVD_PROPS;
1564                                        i = ZFS_GET_NCOLS;
1565                                        break;
1566                                default:
1567                                        (void) fprintf(stderr,
1568                                            gettext("invalid column name "
1569                                            "'%s'\n"), value);
1570                                        usage(B_FALSE);
1571                                }
1572                        }
1573                        break;

1575                case 's':
1576                        cb.cb_sources = 0;
```

```
1577                        while (*optarg != '\0') {
1578                                static char *source_subopts[] = {
1579                                        "local", "default", "inherited",
1580                                        "received", "temporary", "none",
1581                                        NULL };

1583                                switch (getsubopt(&optarg, source_subopts,
1584                                    &value)) {
1585                                case 0:
1586                                        cb.cb_sources |= ZPROP_SRC_LOCAL;
1587                                        break;
1588                                case 1:
1589                                        cb.cb_sources |= ZPROP_SRC_DEFAULT;
1590                                        break;
1591                                case 2:
1592                                        cb.cb_sources |= ZPROP_SRC_INHERITED;
1593                                        break;
1594                                case 3:
1595                                        cb.cb_sources |= ZPROP_SRC_RECEIVED;
1596                                        break;
1597                                case 4:
1598                                        cb.cb_sources |= ZPROP_SRC_TEMPORARY;
1599                                        break;
1600                                case 5:
1601                                        cb.cb_sources |= ZPROP_SRC_NONE;
1602                                        break;
1603                                default:
1604                                        (void) fprintf(stderr,
1605                                            gettext("invalid source "
1606                                            "'%s'\n"), value);
1607                                        usage(B_FALSE);
1608                                }
1609                        }
1610                        break;

1612                case 't':
1613                        types = 0;
1614                        flags &= ~ZFS_ITER_PROP_LISTSNAPS;
1615                        while (*optarg != '\0') {
1616                                static char *type_subopts[] = { "filesystem",
1617                                    "volume", "snapshot", "all", NULL };

1619                                switch (getsubopt(&optarg, type_subopts,
1620                                    &value)) {
1621                                case 0:
1622                                        types |= ZFS_TYPE_FILESYSTEM;
1623                                        break;
1624                                case 1:
1625                                        types |= ZFS_TYPE_VOLUME;
1626                                        break;
1627                                case 2:
1628                                        types |= ZFS_TYPE_SNAPSHOT;
1629                                        break;
1630                                case 3:
1631                                        types = ZFS_TYPE_DATASET;
1632                                        break;

1634                                default:
1635                                        (void) fprintf(stderr,
1636                                            gettext("invalid type '%s'\n"),
1637                                            value);
1638                                        usage(B_FALSE);
1639                                }
1640                        }
1641                        break;
```



```
1643                case '?':
1644                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1645                            optopt);
1646                        usage(B_FALSE);
1647                }
1648        }

1650        argc -= optind;
1651        argv += optind;

1653        if (argc < 1) {
1654                (void) fprintf(stderr, gettext("missing property "
1655                    "argument\n"));
1656                usage(B_FALSE);
1657        }

1659        fields = argv[0];

1661        if (zprop_get_list(g_zfs, fields, &cb.cb_proplist, ZFS_TYPE_DATASET)
1662            != 0)
1663                usage(B_FALSE);

1665        argc--;
1666        argv++;

1668        /*
1669         * As part of zfs_expand_proplist(), we keep track of the maximum column
1670         * width for each property.  For the 'NAME' (and 'SOURCE') columns, we
1671         * need to know the maximum name length.  However, the user likely did
1672         * not specify 'name' as one of the properties to fetch, so we need to
1673         * make sure we always include at least this property for
1674         * print_get_headers() to work properly.
1675         */
1676        if (cb.cb_proplist != NULL) {
1677                fake_name.pl_prop = ZFS_PROP_NAME;
1678                fake_name.pl_width = strlen(gettext("NAME"));
1679                fake_name.pl_next = cb.cb_proplist;
1680                cb.cb_proplist = &fake_name;
1681        }

1683        cb.cb_first = B_TRUE;

1685        /* run for each object */
1686        ret = zfs_for_each(argc, argv, flags, types, NULL,
1687            &cb.cb_proplist, limit, get_callback, &cb);

1689        if (cb.cb_proplist == &fake_name)
1690                zprop_free_list(fake_name.pl_next);
1691        else
1692                zprop_free_list(cb.cb_proplist);

1694        return (ret);
1695 }

1697 /*
1698  * inherit [-rS] <property> <fs|vol> ...
1699  *
1700  *      -r      Recurse over all children
1701  *      -S      Revert to received value, if any
1702  *
1703  * For each dataset specified on the command line, inherit the given property
1704  * from its parent.  Inheriting a property at the pool level will cause it to
1705  * use the default value.  The '-r' flag will recurse over all children, and is
1706  * useful for setting a property on a hierarchy-wide basis, regardless of any
1707  * local modifications for each dataset.
1708  */
```

```
1710 typedef struct inherit_cbdata {
1711         const char *cb_propname;
1712         boolean_t cb_received;
1713 } inherit_cbdata_t;

1715 static int
1716 inherit_recurse_cb(zfs_handle_t *zhp, void *data)
1717 {
1718         inherit_cbdata_t *cb = data;
1719         zfs_prop_t prop = zfs_name_to_prop(cb->cb_propname);

1721         /*
1722          * If we're doing it recursively, then ignore properties that
1723          * are not valid for this type of dataset.
1724          */
1725         if (prop != ZPROP_INVAL &&
1726             !zfs_prop_valid_for_type(prop, zfs_get_type(zhp)))
1727                 return (0);

1729         return (zfs_prop_inherit(zhp, cb->cb_propname, cb->cb_received) != 0);
1730 }

1732 static int
1733 inherit_cb(zfs_handle_t *zhp, void *data)
1734 {
1735         inherit_cbdata_t *cb = data;

1737         return (zfs_prop_inherit(zhp, cb->cb_propname, cb->cb_received) != 0);
1738 }

1740 static int
1741 zfs_do_inherit(int argc, char **argv)
1742 {
1743         int c;
1744         zfs_prop_t prop;
1745         inherit_cbdata_t cb = { 0 };
1746         char *propname;
1747         int ret = 0;
1748         int flags = 0;
1749         boolean_t received = B_FALSE;

1751         /* check options */
1752         while ((c = getopt(argc, argv, "rS")) != -1) {
1753                 switch (c) {
1754                 case 'r':
1755                         flags |= ZFS_ITER_RECURSE;
1756                         break;
1757                 case 'S':
1758                         received = B_TRUE;
1759                         break;
1760                 case '?':
1761                 default:
1762                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1763                             optopt);
1764                         usage(B_FALSE);
1765                 }
1766         }

1768         argc -= optind;
1769         argv += optind;

1771         /* check number of arguments */
1772         if (argc < 1) {
1773                 (void) fprintf(stderr, gettext("missing property argument\n"));
1774                 usage(B_FALSE);
```



```
1775         }
1776         if (argc < 2) {
1777                 (void) fprintf(stderr, gettext("missing dataset argument\n"));
1778                 usage(B_FALSE);
1779         }

1781         propname = argv[0];
1782         argc--;
1783         argv++;

1785         if ((prop = zfs_name_to_prop(propname)) != ZPROP_INVAL) {
1786                 if (zfs_prop_readonly(prop)) {
1787                         (void) fprintf(stderr, gettext(
1788                             "%s property is read-only\n"),
1789                             propname);
1790                         return (1);
1791                 }
1792                 if (!zfs_prop_inheritable(prop) && !received) {
1793                         (void) fprintf(stderr, gettext("'%s' property cannot "
1794                             "be inherited\n"), propname);
1795                         if (prop == ZFS_PROP_QUOTA ||
1796                             prop == ZFS_PROP_RESERVATION ||
1797                             prop == ZFS_PROP_REFQUOTA ||
1798                             prop == ZFS_PROP_REFRESERVATION)
1799                                 (void) fprintf(stderr, gettext("use 'zfs set "
1800                                     "%s=none' to clear\n"), propname);
1801                         return (1);
1802                 }
1803                 if (received && (prop == ZFS_PROP_VOLSIZE ||
1804                     prop == ZFS_PROP_VERSION)) {
1805                         (void) fprintf(stderr, gettext("'%s' property cannot "
1806                             "be reverted to a received value\n"), propname);
1807                         return (1);
1808                 }
1809         } else if (!zfs_prop_user(propname)) {
1810                 (void) fprintf(stderr, gettext("invalid property '%s'\n"),
1811                     propname);
1812                 usage(B_FALSE);
1813         }

1815         cb.cb_propname = propname;
1816         cb.cb_received = received;

1818         if (flags & ZFS_ITER_RECURSE) {
1819                 ret = zfs_for_each(argc, argv, flags, ZFS_TYPE_DATASET,
1820                     NULL, NULL, 0, inherit_recurse_cb, &cb);
1821         } else {
1822                 ret = zfs_for_each(argc, argv, flags, ZFS_TYPE_DATASET,
1823                     NULL, NULL, 0, inherit_cb, &cb);
1824         }

1826         return (ret);
1827 }

1829 typedef struct upgrade_cbdata {
1830         uint64_t cb_numupgraded;
1831         uint64_t cb_numsamegraded;
1832         uint64_t cb_numfailed;
1833         uint64_t cb_version;
1834         boolean_t cb_newer;
1835         boolean_t cb_foundone;
1836         char cb_lastfs[ZFS_MAXNAMELEN];
1837 } upgrade_cbdata_t;

1839 static int
1840 same_pool(zfs_handle_t *zhp, const char *name)
```

```
1841 {
1842         int len1 = strcspn(name, "/@");
1843         const char *zhname = zfs_get_name(zhp);
1844         int len2 = strcspn(zhname, "/@");
1845
1846         if (len1 != len2)
1847                 return (B_FALSE);
1848         return (strncmp(name, zhname, len1) == 0);
1849 }
1850
1851 static int
1852 upgrade_list_callback(zfs_handle_t *zhp, void *data)
1853 {
1854         upgrade_cbdata_t *cb = data;
1855         int version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1856
1857         /* list if it's old/new */
1858         if ((!cb->cb_newer && version < ZPL_VERSION) ||
1859             (cb->cb_newer && version > ZPL_VERSION)) {
1860                 char *str;
1861                 if (cb->cb_newer) {
1862                         str = gettext("The following filesystems are "
1863                             "formatted using a newer software version and\n"
1864                             "cannot be accessed on the current system.\n\n");
1865                 } else {
1866                         str = gettext("The following filesystems are "
1867                             "out of date, and can be upgraded.  After being\n"
1868                             "upgraded, these filesystems (and any 'zfs send' "
1869                             "streams generated from\n"
1870                             "subsequent snapshots) will no longer be "
1871                             "accessible by older software versions.\n\n");
1872                 }
1873
1874                 if (!cb->cb_foundone) {
1875                         (void) puts(str);
1876                         (void) printf(gettext("VER  FILESYSTEM\n"));
1877                         (void) printf(gettext("---  ------------\n"));
1878                         cb->cb_foundone = B_TRUE;
1879                 }
1880
1881                 (void) printf("%2u   %s\n", version, zfs_get_name(zhp));
1882         }
1883
1884         return (0);
1885 }
1886
1887 static int
1888 upgrade_set_callback(zfs_handle_t *zhp, void *data)
1889 {
1890         upgrade_cbdata_t *cb = data;
1891         int version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1892         int needed_spa_version;
1893         int spa_version;
1894
1895         if (zfs_spa_version(zhp, &spa_version) < 0)
1896                 return (-1);
1897
1898         needed_spa_version = zfs_spa_version_map(cb->cb_version);
1899
1900         if (needed_spa_version < 0)
1901                 return (-1);
1902
1903         if (spa_version < needed_spa_version) {
1904                 /* can't upgrade */
1905                 (void) printf(gettext("%s: can not be "
1906                     "upgraded; the pool version needs to first "
```



```
1907                     "be upgraded\nto version %d\n\n"),
1908                     zfs_get_name(zhp), needed_spa_version);
1909                 cb->cb_numfailed++;
1910                 return (0);
1911         }
1912
1913         /* upgrade */
1914         if (version < cb->cb_version) {
1915                 char verstr[16];
1916                 (void) snprintf(verstr, sizeof (verstr),
1917                     "%llu", cb->cb_version);
1918                 if (cb->cb_lastfs[0] && !same_pool(zhp, cb->cb_lastfs)) {
1919                         /*
1920                          * If they did "zfs upgrade -a", then we could
1921                          * be doing ioctls to different pools.  We need
1922                          * to log this history once to each pool, and bypass
1923                          * the normal history logging that happens in main().
1924                          */
1925                         (void) zpool_log_history(g_zfs, history_str);
1926                         log_history = B_FALSE;
1927                 }
1928                 if (zfs_prop_set(zhp, "version", verstr) == 0)
1929                         cb->cb_numupgraded++;
1930                 else
1931                         cb->cb_numfailed++;
1932                 (void) strcpy(cb->cb_lastfs, zfs_get_name(zhp));
1933         } else if (version > cb->cb_version) {
1934                 /* can't downgrade */
1935                 (void) printf(gettext("%s: can not be downgraded; "
1936                     "it is already at version %u\n"),
1937                     zfs_get_name(zhp), version);
1938                 cb->cb_numfailed++;
1939         } else {
1940                 cb->cb_numsamegraded++;
1941         }
1942         return (0);
1943 }
1944
1945 /*
1946  * zfs upgrade
1947  * zfs upgrade -v
1948  * zfs upgrade [-r] [-V <version>] <-a | filesystem>
1949  */
1950 static int
1951 zfs_do_upgrade(int argc, char **argv)
1952 {
1953         boolean_t all = B_FALSE;
1954         boolean_t showversions = B_FALSE;
1955         int ret = 0;
1956         upgrade_cbdata_t cb = { 0 };
1957         char c;
1958         int flags = ZFS_ITER_ARGS_CAN_BE_PATHS;
1959
1960         /* check options */
1961         while ((c = getopt(argc, argv, "rvV:a")) != -1) {
1962                 switch (c) {
1963                 case 'r':
1964                         flags |= ZFS_ITER_RECURSE;
1965                         break;
1966                 case 'v':
1967                         showversions = B_TRUE;
1968                         break;
1969                 case 'V':
1970                         if (zfs_prop_string_to_index(ZFS_PROP_VERSION,
1971                             optarg, &cb.cb_version) != 0) {
1972                                 (void) fprintf(stderr,
```

```
1973                                    gettext("invalid version %s\n"), optarg);
1974                                usage(B_FALSE);
1975                        }
1976                        break;
1977                case 'a':
1978                        all = B_TRUE;
1979                        break;
1980                case '?':
1981                default:
1982                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1983                            optopt);
1984                        usage(B_FALSE);
1985                }
1986        }

1988        argc -= optind;
1989        argv += optind;

1991        if ((!all && !argc) && ((flags & ZFS_ITER_RECURSE) | cb.cb_version))
1992                usage(B_FALSE);
1993        if (showversions && (flags & ZFS_ITER_RECURSE || all ||
1994            cb.cb_version || argc))
1995                usage(B_FALSE);
1996        if ((all || argc) && (showversions))
1997                usage(B_FALSE);
1998        if (all && argc)
1999                usage(B_FALSE);

2001        if (showversions) {
2002                /* Show info on available versions. */
2003                (void) printf(gettext("The following filesystem versions are "
2004                    "supported:\n\n"));
2005                (void) printf(gettext("VER  DESCRIPTION\n"));
2006                (void) printf("---  "
2007                    "--------------\n");
2008                (void) printf(gettext(" 1   Initial ZFS filesystem version\n"));
2009                (void) printf(gettext(" 2   Enhanced directory entries\n"));
2010                (void) printf(gettext(" 3   Case insensitive and filesystem "
2011                    "user identifier (FUID)\n"));
2012                (void) printf(gettext(" 4   userquota, groupquota "
2013                    "properties\n"));
2014                (void) printf(gettext(" 5   System attributes\n"));
2015                (void) printf(gettext("\nFor more information on a particular "
2016                    "version, including supported releases,\n"));
2017                (void) printf("see the ZFS Administration Guide.\n\n");
2018                ret = 0;
2019        } else if (argc || all) {
2020                /* Upgrade filesystems */
2021                if (cb.cb_version == 0)
2022                        cb.cb_version = ZPL_VERSION;
2023                ret = zfs_for_each(argc, argv, flags, ZFS_TYPE_FILESYSTEM,
2024                    NULL, NULL, 0, upgrade_set_callback, &cb);
2025                (void) printf(gettext("%llu filesystems upgraded\n"),
2026                    cb.cb_numupgraded);
2027                if (cb.cb_numsamegraded) {
2028                        (void) printf(gettext("%llu filesystems already at "
2029                            "this version\n"),
2030                            cb.cb_numsamegraded);
2031                }
2032                if (cb.cb_numfailed != 0)
2033                        ret = 1;
2034        } else {
2035                /* List old-version filesytems */
2036                boolean_t found;
2037                (void) printf(gettext("This system is currently running "
2038                    "ZFS filesystem version %llu.\n\n"), ZPL_VERSION);
```



```
2040                flags |= ZFS_ITER_RECURSE;
2041                ret = zfs_for_each(0, NULL, flags, ZFS_TYPE_FILESYSTEM,
2042                    NULL, NULL, 0, upgrade_list_callback, &cb);

2044                found = cb.cb_foundone;
2045                cb.cb_foundone = B_FALSE;
2046                cb.cb_newer = B_TRUE;

2048                ret = zfs_for_each(0, NULL, flags, ZFS_TYPE_FILESYSTEM,
2049                    NULL, NULL, 0, upgrade_list_callback, &cb);

2051                if (!cb.cb_foundone && !found) {
2052                        (void) printf(gettext("All filesystems are "
2053                            "formatted with the current version.\n"));
2054                }
2055        }

2057        return (ret);
2058 }

2060 /*
2061  * zfs userspace [-Hinp] [-o field[,...]] [-s field [-s field]...]
2062  *               [-S field [-S field]...] [-t type[,...]] filesystem | snapshot
2063  * zfs groupspace [-Hinp] [-o field[,...]] [-s field [-s field]...]
2064  *                [-S field [-S field]...] [-t type[,...]] filesystem | snapshot
2065  *
2066  *      -H      Scripted mode; elide headers and separate columns by tabs.
2067  *      -i      Translate SID to POSIX ID.
2068  *      -n      Print numeric ID instead of user/group name.
2069  *      -o      Control which fields to display.
2070  *      -p      Use exact (parseable) numeric output.
2071  *      -s      Specify sort columns, descending order.
2072  *      -S      Specify sort columns, ascending order.
2073  *      -t      Control which object types to display.
2074  *
2075  *      Displays space consumed by, and quotas on, each user in the specified
2076  *      filesystem or snapshot.
2077  */

2079 /* us_field_types, us_field_hdr and us_field_names should be kept in sync */
2080 enum us_field_types {
2081         USFIELD_TYPE,
2082         USFIELD_NAME,
2083         USFIELD_USED,
2084         USFIELD_QUOTA
2085 };
2086 static char *us_field_hdr[] = { "TYPE", "NAME", "USED", "QUOTA" };
2087 static char *us_field_names[] = { "type", "name", "used", "quota" };
2088 #define USFIELD_LAST    (sizeof (us_field_names) / sizeof (char *))

2090 #define USTYPE_PSX_GRP  (1 << 0)
2091 #define USTYPE_PSX_USR  (1 << 1)
2092 #define USTYPE_SMB_GRP  (1 << 2)
2093 #define USTYPE_SMB_USR  (1 << 3)
2094 #define USTYPE_ALL      \
2095         (USTYPE_PSX_GRP | USTYPE_PSX_USR | USTYPE_SMB_GRP | USTYPE_SMB_USR)

2097 static int us_type_bits[] = {
2098         USTYPE_PSX_GRP,
2099         USTYPE_PSX_USR,
2100         USTYPE_SMB_GRP,
2101         USTYPE_SMB_USR,
2102         USTYPE_ALL
2103 };
2104 static char *us_type_names[] = { "posixgroup", "posxiuser", "smbgroup",
```

```
2105         "smbuser", "all" };

2107 typedef struct us_node {
2108         nvlist_t        *usn_nvl;
2109         uu_avl_node_t   usn_avlnode;
2110         uu_list_node_t  usn_listnode;
2111 } us_node_t;

2113 typedef struct us_cbdata {
2114         nvlist_t        **cb_nvlp;
2115         uu_avl_pool_t   *cb_avl_pool;
2116         uu_avl_t        *cb_avl;
2117         boolean_t       cb_numname;
2118         boolean_t       cb_nicenum;
2119         boolean_t       cb_sid2posix;
2120         zfs_userquota_prop_t cb_prop;
2121         zfs_sort_column_t *cb_sortcol;
2122         size_t          cb_width[USFIELD_LAST];
2123 } us_cbdata_t;

2125 static boolean_t us_populated = B_FALSE;

2127 typedef struct {
2128         zfs_sort_column_t *si_sortcol;
2129         boolean_t       si_numname;
2130 } us_sort_info_t;

2132 static int
2133 us_field_index(char *field)
2134 {
2135         int i;

2137         for (i = 0; i < USFIELD_LAST; i++) {
2138                 if (strcmp(field, us_field_names[i]) == 0)
2139                         return (i);
2140         }

2142         return (-1);
2143 }

2145 static int
2146 us_compare(const void *larg, const void *rarg, void *unused)
2147 {
2148         const us_node_t *l = larg;
2149         const us_node_t *r = rarg;
2150         us_sort_info_t *si = (us_sort_info_t *)unused;
2151         zfs_sort_column_t *sortcol = si->si_sortcol;
2152         boolean_t numname = si->si_numname;
2153         nvlist_t *lnvl = l->usn_nvl;
2154         nvlist_t *rnvl = r->usn_nvl;
2155         int rc = 0;
2156         boolean_t lvb, rvb;

2158         for (; sortcol != NULL; sortcol = sortcol->sc_next) {
2159                 char *lvstr = "";
2160                 char *rvstr = "";
2161                 uint32_t lv32 = 0;
2162                 uint32_t rv32 = 0;
2163                 uint64_t lv64 = 0;
2164                 uint64_t rv64 = 0;
2165                 zfs_prop_t prop = sortcol->sc_prop;
2166                 const char *propname = NULL;
2167                 boolean_t reverse = sortcol->sc_reverse;

2169                 switch (prop) {
2170                 case ZFS_PROP_TYPE:
```



```
2171                         propname = "type";
2172                         (void) nvlist_lookup_uint32(lnvl, propname, &lv32);
2173                         (void) nvlist_lookup_uint32(rnvl, propname, &rv32);
2174                         if (rv32 != lv32)
2175                                 rc = (rv32 < lv32) ? 1 : -1;
2176                         break;
2177                 case ZFS_PROP_NAME:
2178                         propname = "name";
2179                         if (numname) {
2180                                 (void) nvlist_lookup_uint64(lnvl, propname,
2181                                     &lv64);
2182                                 (void) nvlist_lookup_uint64(rnvl, propname,
2183                                     &rv64);
2184                                 if (rv64 != lv64)
2185                                         rc = (rv64 < lv64) ? 1 : -1;
2186                         } else {
2187                                 (void) nvlist_lookup_string(lnvl, propname,
2188                                     &lvstr);
2189                                 (void) nvlist_lookup_string(rnvl, propname,
2190                                     &rvstr);
2191                                 rc = strcmp(lvstr, rvstr);
2192                         }
2193                         break;
2194                 case ZFS_PROP_USED:
2195                 case ZFS_PROP_QUOTA:
2196                         if (!us_populated)
2197                                 break;
2198                         if (prop == ZFS_PROP_USED)
2199                                 propname = "used";
2200                         else
2201                                 propname = "quota";
2202                         (void) nvlist_lookup_uint64(lnvl, propname, &lv64);
2203                         (void) nvlist_lookup_uint64(rnvl, propname, &rv64);
2204                         if (rv64 != lv64)
2205                                 rc = (rv64 < lv64) ? 1 : -1;
2206                         break;
2207                 }

2209                 if (rc != 0) {
2210                         if (rc < 0)
2211                                 return (reverse ? 1 : -1);
2212                         else
2213                                 return (reverse ? -1 : 1);
2214                 }
2215         }

2217         /*
2218          * If entries still seem to be the same, check if they are of the same
2219          * type (smbentity is added only if we are doing SID to POSIX ID
2220          * translation where we can have duplicate type/name combinations).
2221          */
2222         if (nvlist_lookup_boolean_value(lnvl, "smbentity", &lvb) == 0 &&
2223             nvlist_lookup_boolean_value(rnvl, "smbentity", &rvb) == 0 &&
2224             lvb != rvb)
2225                 return (lvb < rvb ? -1 : 1);

2227         return (0);
2228 }

2230 static inline const char *
2231 us_type2str(unsigned field_type)
2232 {
2233         switch (field_type) {
2234         case USTYPE_PSX_USR:
2235                 return ("POSIX User");
2236         case USTYPE_PSX_GRP:
```

```
2237                 return ("POSIX Group");
2238         case USTYPE_SMB_USR:
2239                 return ("SMB User");
2240         case USTYPE_SMB_GRP:
2241                 return ("SMB Group");
2242         default:
2243                 return ("Undefined");
2244         }
2245 }

2247 static int
2248 userspace_cb(void *arg, const char *domain, uid_t rid, uint64_t space)
2249 {
2250         us_cbdata_t *cb = (us_cbdata_t *)arg;
2251         zfs_userquota_prop_t prop = cb->cb_prop;
2252         char *name = NULL;
2253         char *propname;
2254         char sizebuf[32];
2255         us_node_t *node;
2256         uu_avl_pool_t *avl_pool = cb->cb_avl_pool;
2257         uu_avl_t *avl = cb->cb_avl;
2258         uu_avl_index_t idx;
2259         nvlist_t *props;
2260         us_node_t *n;
2261         zfs_sort_column_t *sortcol = cb->cb_sortcol;
2262         unsigned type;
2263         const char *typestr;
2264         size_t namelen;
2265         size_t typelen;
2266         size_t sizelen;
2267         int typeidx, nameidx, sizeidx;
2268         us_sort_info_t sortinfo = { sortcol, cb->cb_numname };
2269         boolean_t smbentity = B_FALSE;

2271         if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
2272                 nomem();
2273         node = safe_malloc(sizeof (us_node_t));
2274         uu_avl_node_init(node, &node->usn_avlnode, avl_pool);
2275         node->usn_nvl = props;

2277         if (domain != NULL && domain[0] != '\0') {
2278                 /* SMB */
2279                 char sid[ZFS_MAXNAMELEN + 32];
2280                 uid_t id;
2281                 uint64_t classes;
2282                 int err;
2283                 directory_error_t e;

2285                 smbentity = B_TRUE;

2287                 (void) snprintf(sid, sizeof (sid), "%s-%u", domain, rid);

2289                 if (prop == ZFS_PROP_GROUPUSED || prop == ZFS_PROP_GROUPQUOTA) {
2290                         type = USTYPE_SMB_GRP;
2291                         err = sid_to_id(sid, B_FALSE, &id);
2292                 } else {
2293                         type = USTYPE_SMB_USR;
2294                         err = sid_to_id(sid, B_TRUE, &id);
2295                 }

2297                 if (err == 0) {
2298                         rid = id;
2299                         if (!cb->cb_sid2posix) {
2300                                 e = directory_name_from_sid(NULL, sid, &name,
2301                                     &classes);
2302                                 if (e != NULL)
```



```
2303                                         directory_error_free(e);
2304                                 if (name == NULL)
2305                                         name = sid;
2306                         }
2307                 }
2308         }

2310         if (cb->cb_sid2posix || domain == NULL || domain[0] == '\0') {
2311                 /* POSIX or -i */
2312                 if (prop == ZFS_PROP_GROUPUSED || prop == ZFS_PROP_GROUPQUOTA) {
2313                         type = USTYPE_PSX_GRP;
2314                         if (!cb->cb_numname) {
2315                                 struct group *g;

2317                                 if ((g = getgrgid(rid)) != NULL)
2318                                         name = g->gr_name;
2319                         }
2320                 } else {
2321                         type = USTYPE_PSX_USR;
2322                         if (!cb->cb_numname) {
2323                                 struct passwd *p;

2325                                 if ((p = getpwuid(rid)) != NULL)
2326                                         name = p->pw_name;
2327                         }
2328                 }
2329         }

2331         /*
2332          * Make sure that the type/name combination is unique when doing
2333          * SID to POSIX ID translation (hence changing the type from SMB to
2334          * POSIX).
2335          */
2336         if (cb->cb_sid2posix &&
2337             nvlist_add_boolean_value(props, "smbentity", smbentity) != 0)
2338                 nomem();

2340         /* Calculate/update width of TYPE field */
2341         typestr = us_type2str(type);
2342         typelen = strlen(gettext(typestr));
2343         typeidx = us_field_index("type");
2344         if (typelen > cb->cb_width[typeidx])
2345                 cb->cb_width[typeidx] = typelen;
2346         if (nvlist_add_uint32(props, "type", type) != 0)
2347                 nomem();

2349         /* Calculate/update width of NAME field */
2350         if ((cb->cb_numname && cb->cb_sid2posix) || name == NULL) {
2351                 if (nvlist_add_uint64(props, "name", rid) != 0)
2352                         nomem();
2353                 namelen = snprintf(NULL, 0, "%u", rid);
2354         } else {
2355                 if (nvlist_add_string(props, "name", name) != 0)
2356                         nomem();
2357                 namelen = strlen(name);
2358         }
2359         nameidx = us_field_index("name");
2360         if (namelen > cb->cb_width[nameidx])
2361                 cb->cb_width[nameidx] = namelen;

2363         /*
2364          * Check if this type/name combination is in the list and update it;
2365          * otherwise add new node to the list.
2366          */
2367         if ((n = uu_avl_find(avl, node, &sortinfo, &idx)) == NULL) {
2368                 uu_avl_insert(avl, node, idx);
```

```
2369         } else {
2370                 nvlist_free(props);
2371                 free(node);
2372                 node = n;
2373                 props = node->usn_nvl;
2374         }

2376         /* Calculate/update width of USED/QUOTA fields */
2377         if (cb->cb_nicenum)
2378                 zfs_nicenum(space, sizebuf, sizeof (sizebuf));
2379         else
2380                 (void) snprintf(sizebuf, sizeof (sizebuf), "%llu", space);
2381         sizelen = strlen(sizebuf);
2382         if (prop == ZFS_PROP_USERUSED || prop == ZFS_PROP_GROUPUSED) {
2383                 propname = "used";
2384                 if (!nvlist_exists(props, "quota"))
2385                         (void) nvlist_add_uint64(props, "quota", 0);
2386         } else {
2387                 propname = "quota";
2388                 if (!nvlist_exists(props, "used"))
2389                         (void) nvlist_add_uint64(props, "used", 0);
2390         }
2391         sizeidx = us_field_index(propname);
2392         if (sizelen > cb->cb_width[sizeidx])
2393                 cb->cb_width[sizeidx] = sizelen;

2395         if (nvlist_add_uint64(props, propname, space) != 0)
2396                 nomem();

2398         return (0);
2399 }

2401 static void
2402 print_us_node(boolean_t scripted, boolean_t parsable, int *fields, int types,
2403     size_t *width, us_node_t *node)
2404 {
2405         nvlist_t *nvl = node->usn_nvl;
2406         char valstr[ZFS_MAXNAMELEN];
2407         boolean_t first = B_TRUE;
2408         int cfield = 0;
2409         int field;
2410         uint32_t ustype;

2412         /* Check type */
2413         (void) nvlist_lookup_uint32(nvl, "type", &ustype);
2414         if (!(ustype & types))
2415                 return;

2417         while ((field = fields[cfield]) != USFIELD_LAST) {
2418                 nvpair_t *nvp = NULL;
2419                 data_type_t type;
2420                 uint32_t val32;
2421                 uint64_t val64;
2422                 char *strval = NULL;

2424                 while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
2425                         if (strcmp(nvpair_name(nvp),
2426                             us_field_names[field]) == 0)
2427                                 break;
2428                 }

2430                 type = nvpair_type(nvp);
2431                 switch (type) {
2432                 case DATA_TYPE_UINT32:
2433                         (void) nvpair_value_uint32(nvp, &val32);
2434                         break;
```



```
2435                 case DATA_TYPE_UINT64:
2436                         (void) nvpair_value_uint64(nvp, &val64);
2437                         break;
2438                 case DATA_TYPE_STRING:
2439                         (void) nvpair_value_string(nvp, &strval);
2440                         break;
2441                 default:
2442                         (void) fprintf(stderr, "invalid data type\n");
2443                 }

2445                 switch (field) {
2446                 case USFIELD_TYPE:
2447                         strval = (char *)us_type2str(val32);
2448                         break;
2449                 case USFIELD_NAME:
2450                         if (type == DATA_TYPE_UINT64) {
2451                                 (void) sprintf(valstr, "%llu", val64);
2452                                 strval = valstr;
2453                         }
2454                         break;
2455                 case USFIELD_USED:
2456                 case USFIELD_QUOTA:
2457                         if (type == DATA_TYPE_UINT64) {
2458                                 if (parsable) {
2459                                         (void) sprintf(valstr, "%llu", val64);
2460                                 } else {
2461                                         zfs_nicenum(val64, valstr,
2462                                             sizeof (valstr));
2463                                 }
2464                                 if (field == USFIELD_QUOTA &&
2465                                     strcmp(valstr, "0") == 0)
2466                                         strval = "none";
2467                                 else
2468                                         strval = valstr;
2469                         }
2470                         break;
2471                 }

2473                 if (!first) {
2474                         if (scripted)
2475                                 (void) printf("\t");
2476                         else
2477                                 (void) printf("  ");
2478                 }
2479                 if (scripted)
2480                         (void) printf("%s", strval);
2481                 else if (field == USFIELD_TYPE || field == USFIELD_NAME)
2482                         (void) printf("%-*s", width[field], strval);
2483                 else
2484                         (void) printf("%*s", width[field], strval);

2486                 first = B_FALSE;
2487                 cfield++;
2488         }

2490         (void) printf("\n");
2491 }

2493 static void
2494 print_us(boolean_t scripted, boolean_t parsable, int *fields, int types,
2495     size_t *width, boolean_t rmnode, uu_avl_t *avl)
2496 {
2497         us_node_t *node;
2498         const char *col;
2499         int cfield = 0;
2500         int field;
```

```
2502         if (!scripted) {
2503                 boolean_t first = B_TRUE;

2505                 while ((field = fields[cfield]) != USFIELD_LAST) {
2506                         col = gettext(us_field_hdr[field]);
2507                         if (field == USFIELD_TYPE || field == USFIELD_NAME) {
2508                                 (void) printf(first ? "%-*s" : "  %-*s",
2509                                     width[field], col);
2510                         } else {
2511                                 (void) printf(first ? "%*s" : "  %*s",
2512                                     width[field], col);
2513                         }
2514                         first = B_FALSE;
2515                         cfield++;
2516                 }
2517                 (void) printf("\n");
2518         }

2520         for (node = uu_avl_first(avl); node; node = uu_avl_next(avl, node)) {
2521                 print_us_node(scripted, parsable, fields, types, width, node);
2522                 if (rmnode)
2523                         nvlist_free(node->usn_nvl);
2524         }
2525 }

2527 static int
2528 zfs_do_userspace(int argc, char **argv)
2529 {
2530         zfs_handle_t *zhp;
2531         zfs_userquota_prop_t p;
2532         uu_avl_pool_t *avl_pool;
2533         uu_avl_t *avl_tree;
2534         uu_avl_walk_t *walk;
2535         char *delim;
2536         char deffields[] = "type,name,used,quota";
2537         char *ofield = NULL;
2538         char *tfield = NULL;
2539         int cfield = 0;
2540         int fields[256];
2541         int i;
2542         boolean_t scripted = B_FALSE;
2543         boolean_t prtnum = B_FALSE;
2544         boolean_t parsable = B_FALSE;
2545         boolean_t sid2posix = B_FALSE;
2546         int ret = 0;
2547         int c;
2548         zfs_sort_column_t *sortcol = NULL;
2549         int types = USTYPE_PSX_USR | USTYPE_SMB_USR;
2550         us_cbdata_t cb;
2551         us_node_t *node;
2552         us_node_t *rmnode;
2553         uu_list_pool_t *listpool;
2554         uu_list_t *list;
2555         uu_avl_index_t idx = 0;
2556         uu_list_index_t idx2 = 0;

2558         if (argc < 2)
2559                 usage(B_FALSE);

2561         if (strcmp(argv[0], "groupspace") == 0)
2562                 /* Toggle default group types */
2563                 types = USTYPE_PSX_GRP | USTYPE_SMB_GRP;

2565         while ((c = getopt(argc, argv, "nHpo:s:S:t:i")) != -1) {
2566                 switch (c) {
```



```
2567                 case 'n':
2568                         prtnum = B_TRUE;
2569                         break;
2570                 case 'H':
2571                         scripted = B_TRUE;
2572                         break;
2573                 case 'p':
2574                         parsable = B_TRUE;
2575                         break;
2576                 case 'o':
2577                         ofield = optarg;
2578                         break;
2579                 case 's':
2580                 case 'S':
2581                         if (zfs_add_sort_column(&sortcol, optarg,
2582                             c == 's' ? B_FALSE : B_TRUE) != 0) {
2583                                 (void) fprintf(stderr,
2584                                     gettext("invalid field '%s'\n"), optarg);
2585                                 usage(B_FALSE);
2586                         }
2587                         break;
2588                 case 't':
2589                         tfield = optarg;
2590                         break;
2591                 case 'i':
2592                         sid2posix = B_TRUE;
2593                         break;
2594                 case ':':
2595                         (void) fprintf(stderr, gettext("missing argument for "
2596                             "'%c' option\n"), optopt);
2597                         usage(B_FALSE);
2598                         break;
2599                 case '?':
2600                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
2601                             optopt);
2602                         usage(B_FALSE);
2603                 }
2604         }

2606         argc -= optind;
2607         argv += optind;

2609         if (argc < 1) {
2610                 (void) fprintf(stderr, gettext("missing dataset name\n"));
2611                 usage(B_FALSE);
2612         }
2613         if (argc > 1) {
2614                 (void) fprintf(stderr, gettext("too many arguments\n"));
2615                 usage(B_FALSE);
2616         }

2618         /* Use default output fields if not specified using -o */
2619         if (ofield == NULL)
2620                 ofield = deffields;
2621         do {
2622                 if ((delim = strchr(ofield, ',')) != NULL)
2623                         *delim = '\0';
2624                 if ((fields[cfield++] = us_field_index(ofield)) == -1) {
2625                         (void) fprintf(stderr, gettext("invalid type '%s' "
2626                             "for -o option\n"), ofield);
2627                         return (-1);
2628                 }
2629                 if (delim != NULL)
2630                         ofield = delim + 1;
2631         } while (delim != NULL);
2632         fields[cfield] = USFIELD_LAST;
```

```
2634         /* Override output types (-t option) */
2635         if (tfield != NULL) {
2636                 types = 0;
2637
2638                 do {
2639                         boolean_t found = B_FALSE;
2640
2641                         if ((delim = strchr(tfield, ',')) != NULL)
2642                                 *delim = '\0';
2643                         for (i = 0; i < sizeof (us_type_bits) / sizeof (int);
2644                             i++) {
2645                                 if (strcmp(tfield, us_type_names[i]) == 0) {
2646                                         found = B_TRUE;
2647                                         types |= us_type_bits[i];
2648                                         break;
2649                                 }
2650                         }
2651                         if (!found) {
2652                                 (void) fprintf(stderr, gettext("invalid type "
2653                                     "'%s' for -t option\n"), tfield);
2654                                 return (-1);
2655                         }
2656                         if (delim != NULL)
2657                                 tfield = delim + 1;
2658                 } while (delim != NULL);
2659         }
2660
2661         if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_DATASET)) == NULL)
2662                 return (1);
2663
2664         if ((avl_pool = uu_avl_pool_create("us_avl_pool", sizeof (us_node_t),
2665             offsetof(us_node_t, usn_avlnode), us_compare, UU_DEFAULT)) == NULL)
2666                 nomem();
2667         if ((avl_tree = uu_avl_create(avl_pool, NULL, UU_DEFAULT)) == NULL)
2668                 nomem();
2669
2670         /* Always add default sorting columns */
2671         (void) zfs_add_sort_column(&sortcol, "type", B_FALSE);
2672         (void) zfs_add_sort_column(&sortcol, "name", B_FALSE);
2673
2674         cb.cb_sortcol = sortcol;
2675         cb.cb_numname = prtnum;
2676         cb.cb_nicenum = !parsable;
2677         cb.cb_avl_pool = avl_pool;
2678         cb.cb_avl = avl_tree;
2679         cb.cb_sid2posix = sid2posix;
2680
2681         for (i = 0; i < USFIELD_LAST; i++)
2682                 cb.cb_width[i] = strlen(gettext(us_field_hdr[i]));
2683
2684         for (p = 0; p < ZFS_NUM_USERQUOTA_PROPS; p++) {
2685                 if (((p == ZFS_PROP_USERUSED || p == ZFS_PROP_USERQUOTA) &&
2686                     !(types & (USTYPE_PSX_USR | USTYPE_SMB_USR))) ||
2687                     ((p == ZFS_PROP_GROUPUSED || p == ZFS_PROP_GROUPQUOTA) &&
2688                     !(types & (USTYPE_PSX_GRP | USTYPE_SMB_GRP))))
2689                         continue;
2690                 cb.cb_prop = p;
2691                 if ((ret = zfs_userspace(zhp, p, userspace_cb, &cb)) != 0)
2692                         return (ret);
2693         }
2694
2695         /* Sort the list */
2696         if ((node = uu_avl_first(avl_tree)) == NULL)
2697                 return (0);
```



```
2698
2699         us_populated = B_TRUE;
2700
2701         listpool = uu_list_pool_create("tmplist", sizeof (us_node_t),
2702             offsetof(us_node_t, usn_listnode), NULL, UU_DEFAULT);
2703         list = uu_list_create(listpool, NULL, UU_DEFAULT);
2704         uu_list_node_init(node, &node->usn_listnode, listpool);
2705
2706         while (node != NULL) {
2707                 rmnode = node;
2708                 node = uu_avl_next(avl_tree, node);
2709                 uu_avl_remove(avl_tree, rmnode);
2710                 if (uu_list_find(list, rmnode, NULL, &idx2) == NULL)
2711                         uu_list_insert(list, rmnode, idx2);
2712         }
2713
2714         for (node = uu_list_first(list); node != NULL;
2715             node = uu_list_next(list, node)) {
2716                 us_sort_info_t sortinfo = { sortcol, cb.cb_numname };
2717
2718                 if (uu_avl_find(avl_tree, node, &sortinfo, &idx) == NULL)
2719                         uu_avl_insert(avl_tree, node, idx);
2720         }
2721
2722         uu_list_destroy(list);
2723         uu_list_pool_destroy(listpool);
2724
2725         /* Print and free node nvlist memory */
2726         print_us(scripted, parsable, fields, types, cb.cb_width, B_TRUE,
2727             cb.cb_avl);
2728
2729         zfs_free_sort_columns(sortcol);
2730
2731         /* Clean up the AVL tree */
2732         if ((walk = uu_avl_walk_start(cb.cb_avl, UU_WALK_ROBUST)) == NULL)
2733                 nomem();
2734
2735         while ((node = uu_avl_walk_next(walk)) != NULL) {
2736                 uu_avl_remove(cb.cb_avl, node);
2737                 free(node);
2738         }
2739
2740         uu_avl_walk_end(walk);
2741         uu_avl_destroy(avl_tree);
2742         uu_avl_pool_destroy(avl_pool);
2743
2744         return (ret);
2745 }
2746
2747 /*
2748  * list [-r][-d max] [-H] [-o property[,property]...] [-t type[,type]...]
2749  *      [-s property [-s property]...] [-S property [-S property]...]
2750  *      <dataset> ...
2751  *
2752  *      -r      Recurse over all children
2753  *      -d      Limit recursion by depth.
2754  *      -H      Scripted mode; elide headers and separate columns by tabs
2755  *      -o      Control which fields to display.
2756  *      -t      Control which object types to display.
2757  *      -s      Specify sort columns, descending order.
2758  *      -S      Specify sort columns, ascending order.
2759  *
2760  * When given no arguments, lists all filesystems in the system.
2761  * Otherwise, list the specified datasets, optionally recursing down them if
2762  * '-r' is specified.
2763  */
2764 typedef struct list_cbdata {
```

```
2765         boolean_t       cb_first;
2766         boolean_t       cb_scripted;
2767         zprop_list_t    *cb_proplist;
2768 } list_cbdata_t;

2770 /*
2771  * Given a list of columns to display, output appropriate headers for each one.
2772  */
2773 static void
2774 print_header(zprop_list_t *pl)
2775 {
2776         char headerbuf[ZFS_MAXPROPLEN];
2777         const char *header;
2778         int i;
2779         boolean_t first = B_TRUE;
2780         boolean_t right_justify;

2782         for (; pl != NULL; pl = pl->pl_next) {
2783                 if (!first) {
2784                         (void) printf("  ");
2785                 } else {
2786                         first = B_FALSE;
2787                 }

2789                 right_justify = B_FALSE;
2790                 if (pl->pl_prop != ZPROP_INVAL) {
2791                         header = zfs_prop_column_name(pl->pl_prop);
2792                         right_justify = zfs_prop_align_right(pl->pl_prop);
2793                 } else {
2794                         for (i = 0; pl->pl_user_prop[i] != '\0'; i++)
2795                                 headerbuf[i] = toupper(pl->pl_user_prop[i]);
2796                         headerbuf[i] = '\0';
2797                         header = headerbuf;
2798                 }

2800                 if (pl->pl_next == NULL && !right_justify)
2801                         (void) printf("%s", header);
2802                 else if (right_justify)
2803                         (void) printf("%*s", pl->pl_width, header);
2804                 else
2805                         (void) printf("%-*s", pl->pl_width, header);
2806         }

2808         (void) printf("\n");
2809 }

2811 /*
2812  * Given a dataset and a list of fields, print out all the properties according
2813  * to the described layout.
2814  */
2815 static void
2816 print_dataset(zfs_handle_t *zhp, zprop_list_t *pl, boolean_t scripted)
2817 {
2818         boolean_t first = B_TRUE;
2819         char property[ZFS_MAXPROPLEN];
2820         nvlist_t *userprops = zfs_get_user_props(zhp);
2821         nvlist_t *propval;
2822         char *propstr;
2823         boolean_t right_justify;
2824         int width;

2826         for (; pl != NULL; pl = pl->pl_next) {
2827                 if (!first) {
2828                         if (scripted)
2829                                 (void) printf("\t");
2830                         else
```



```
2831                                 (void) printf("  ");
2832                 } else {
2833                         first = B_FALSE;
2834                 }

2836                 if (pl->pl_prop != ZPROP_INVAL) {
2837                         if (zfs_prop_get(zhp, pl->pl_prop, property,
2838                             sizeof (property), NULL, NULL, 0, B_FALSE) != 0)
2839                                 propstr = "-";
2840                         else
2841                                 propstr = property;

2843                         right_justify = zfs_prop_align_right(pl->pl_prop);
2844                 } else if (zfs_prop_userquota(pl->pl_user_prop)) {
2845                         if (zfs_prop_get_userquota(zhp, pl->pl_user_prop,
2846                             property, sizeof (property), B_FALSE) != 0)
2847                                 propstr = "-";
2848                         else
2849                                 propstr = property;
2850                         right_justify = B_TRUE;
2851                 } else if (zfs_prop_written(pl->pl_user_prop)) {
2852                         if (zfs_prop_get_written(zhp, pl->pl_user_prop,
2853                             property, sizeof (property), B_FALSE) != 0)
2854                                 propstr = "-";
2855                         else
2856                                 propstr = property;
2857                         right_justify = B_TRUE;
2858                 } else {
2859                         if (nvlist_lookup_nvlist(userprops,
2860                             pl->pl_user_prop, &propval) != 0)
2861                                 propstr = "-";
2862                         else
2863                                 verify(nvlist_lookup_string(propval,
2864                                     ZPROP_VALUE, &propstr) == 0);
2865                         right_justify = B_FALSE;
2866                 }

2868                 width = pl->pl_width;

2870                 /*
2871                  * If this is being called in scripted mode, or if this is the
2872                  * last column and it is left-justified, don't include a width
2873                  * format specifier.
2874                  */
2875                 if (scripted || (pl->pl_next == NULL && !right_justify))
2876                         (void) printf("%s", propstr);
2877                 else if (right_justify)
2878                         (void) printf("%*s", width, propstr);
2879                 else
2880                         (void) printf("%-*s", width, propstr);
2881         }

2883         (void) printf("\n");
2884 }

2886 /*
2887  * Generic callback function to list a dataset or snapshot.
2888  */
2889 static int
2890 list_callback(zfs_handle_t *zhp, void *data)
2891 {
2892         list_cbdata_t *cbp = data;

2894         if (cbp->cb_first) {
2895                 if (!cbp->cb_scripted)
2896                         print_header(cbp->cb_proplist);
```

```
2897                 cbp->cb_first = B_FALSE;
2898         }

2900         print_dataset(zhp, cbp->cb_proplist, cbp->cb_scripted);

2902         return (0);
2903 }

2905 static int
2906 zfs_do_list(int argc, char **argv)
2907 {
2908         int c;
2909         boolean_t scripted = B_FALSE;
2910         static char default_fields[] =
2911             "name,used,available,referenced,mountpoint";
2912         int types = ZFS_TYPE_DATASET;
2913         boolean_t types_specified = B_FALSE;
2914         char *fields = NULL;
2915         list_cbdata_t cb = { 0 };
2916         char *value;
2917         int limit = 0;
2918         int ret = 0;
2919         zfs_sort_column_t *sortcol = NULL;
2920         int flags = ZFS_ITER_PROP_LISTSNAPS | ZFS_ITER_ARGS_CAN_BE_PATHS;

2922         /* check options */
2923         while ((c = getopt(argc, argv, ":d:o:rt:Hs:S:")) != -1) {
2924                 switch (c) {
2925                 case 'o':
2926                         fields = optarg;
2927                         break;
2928                 case 'd':
2929                         limit = parse_depth(optarg, &flags);
2930                         break;
2931                 case 'r':
2932                         flags |= ZFS_ITER_RECURSE;
2933                         break;
2934                 case 'H':
2935                         scripted = B_TRUE;
2936                         break;
2937                 case 's':
2938                         if (zfs_add_sort_column(&sortcol, optarg,
2939                             B_FALSE) != 0) {
2940                                 (void) fprintf(stderr,
2941                                     gettext("invalid property '%s'\n"), optarg);
2942                                 usage(B_FALSE);
2943                         }
2944                         break;
2945                 case 'S':
2946                         if (zfs_add_sort_column(&sortcol, optarg,
2947                             B_TRUE) != 0) {
2948                                 (void) fprintf(stderr,
2949                                     gettext("invalid property '%s'\n"), optarg);
2950                                 usage(B_FALSE);
2951                         }
2952                         break;
2953                 case 't':
2954                         types = 0;
2955                         types_specified = B_TRUE;
2956                         flags &= ~ZFS_ITER_PROP_LISTSNAPS;
2957                         while (*optarg != '\0') {
2958                                 static char *type_subopts[] = { "filesystem",
2959                                     "volume", "snapshot", "all", NULL };

2961                                 switch (getsubopt(&optarg, type_subopts,
2962                                     &value)) {
```



```
2963                                 case 0:
2964                                         types |= ZFS_TYPE_FILESYSTEM;
2965                                         break;
2966                                 case 1:
2967                                         types |= ZFS_TYPE_VOLUME;
2968                                         break;
2969                                 case 2:
2970                                         types |= ZFS_TYPE_SNAPSHOT;
2971                                         break;
2972                                 case 3:
2973                                         types = ZFS_TYPE_DATASET;
2974                                         break;

2976                                 default:
2977                                         (void) fprintf(stderr,
2978                                             gettext("invalid type '%s'\n"),
2979                                             value);
2980                                         usage(B_FALSE);
2981                                 }
2982                         }
2983                         break;
2984                 case ':':
2985                         (void) fprintf(stderr, gettext("missing argument for "
2986                             "'%c' option\n"), optopt);
2987                         usage(B_FALSE);
2988                         break;
2989                 case '?':
2990                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
2991                             optopt);
2992                         usage(B_FALSE);
2993                 }
2994         }

2996         argc -= optind;
2997         argv += optind;

2999         if (fields == NULL)
3000                 fields = default_fields;

3002         /*
3003          * If "-o space" and no types were specified, don't display snapshots.
3004          */
3005         if (strcmp(fields, "space") == 0 && types_specified == B_FALSE)
3006                 types &= ~ZFS_TYPE_SNAPSHOT;

3008         /*
3009          * If the user specifies '-o all', the zprop_get_list() doesn't
3010          * normally include the name of the dataset.  For 'zfs list', we always
3011          * want this property to be first.
3012          */
3013         if (zprop_get_list(g_zfs, fields, &cb.cb_proplist, ZFS_TYPE_DATASET)
3014             != 0)
3015                 usage(B_FALSE);

3017         cb.cb_scripted = scripted;
3018         cb.cb_first = B_TRUE;

3020         ret = zfs_for_each(argc, argv, flags, types, sortcol, &cb.cb_proplist,
3021             limit, list_callback, &cb);

3023         zprop_free_list(cb.cb_proplist);
3024         zfs_free_sort_columns(sortcol);

3026         if (ret == 0 && cb.cb_first && !cb.cb_scripted)
3027                 (void) printf(gettext("no datasets available\n"));
```

```
3029         return (ret);
3030 }

3032 /*
3033  * zfs rename [-f] <fs | snap | vol> <fs | snap | vol>
3034  * zfs rename [-f] -p <fs | vol> <fs | vol>
3035  * zfs rename -r <snap> <snap>
3036  *
3037  * Renames the given dataset to another of the same type.
3038  *
3039  * The '-p' flag creates all the non-existing ancestors of the target first.
3040  */
3041 /* ARGSUSED */
3042 static int
3043 zfs_do_rename(int argc, char **argv)
3044 {
3045         zfs_handle_t *zhp;
3046         int c;
3047         int ret = 0;
3048         boolean_t recurse = B_FALSE;
3049         boolean_t parents = B_FALSE;
3050         boolean_t force_unmount = B_FALSE;

3052         /* check options */
3053         while ((c = getopt(argc, argv, "prf")) != -1) {
3054                 switch (c) {
3055                 case 'p':
3056                         parents = B_TRUE;
3057                         break;
3058                 case 'r':
3059                         recurse = B_TRUE;
3060                         break;
3061                 case 'f':
3062                         force_unmount = B_TRUE;
3063                         break;
3064                 case '?':
3065                 default:
3066                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3067                             optopt);
3068                         usage(B_FALSE);
3069                 }
3070         }

3072         argc -= optind;
3073         argv += optind;

3075         /* check number of arguments */
3076         if (argc < 1) {
3077                 (void) fprintf(stderr, gettext("missing source dataset "
3078                     "argument\n"));
3079                 usage(B_FALSE);
3080         }
3081         if (argc < 2) {
3082                 (void) fprintf(stderr, gettext("missing target dataset "
3083                     "argument\n"));
3084                 usage(B_FALSE);
3085         }
3086         if (argc > 2) {
3087                 (void) fprintf(stderr, gettext("too many arguments\n"));
3088                 usage(B_FALSE);
3089         }

3091         if (recurse && parents) {
3092                 (void) fprintf(stderr, gettext("-p and -r options are mutually "
3093                     "exclusive\n"));
3094                 usage(B_FALSE);
```



```
3095         }

3097         if (recurse && strchr(argv[0], '@') == 0) {
3098                 (void) fprintf(stderr, gettext("source dataset for recursive "
3099                     "rename must be a snapshot\n"));
3100                 usage(B_FALSE);
3101         }

3103         if ((zhp = zfs_open(g_zfs, argv[0], parents ? ZFS_TYPE_FILESYSTEM |
3104             ZFS_TYPE_VOLUME : ZFS_TYPE_DATASET)) == NULL)
3105                 return (1);

3107         /* If we were asked and the name looks good, try to create ancestors. */
3108         if (parents && zfs_name_valid(argv[1], zfs_get_type(zhp)) &&
3109             zfs_create_ancestors(g_zfs, argv[1]) != 0) {
3110                 zfs_close(zhp);
3111                 return (1);
3112         }

3114         ret = (zfs_rename(zhp, argv[1], recurse, force_unmount) != 0);

3116         zfs_close(zhp);
3117         return (ret);
3118 }

3120 /*
3121  * zfs promote <fs>
3122  *
3123  * Promotes the given clone fs to be the parent
3124  */
3125 /* ARGSUSED */
3126 static int
3127 zfs_do_promote(int argc, char **argv)
3128 {
3129         zfs_handle_t *zhp;
3130         int ret = 0;

3132         /* check options */
3133         if (argc > 1 && argv[1][0] == '-') {
3134                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3135                     argv[1][1]);
3136                 usage(B_FALSE);
3137         }

3139         /* check number of arguments */
3140         if (argc < 2) {
3141                 (void) fprintf(stderr, gettext("missing clone filesystem"
3142                     " argument\n"));
3143                 usage(B_FALSE);
3144         }
3145         if (argc > 2) {
3146                 (void) fprintf(stderr, gettext("too many arguments\n"));
3147                 usage(B_FALSE);
3148         }

3150         zhp = zfs_open(g_zfs, argv[1], ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3151         if (zhp == NULL)
3152                 return (1);

3154         ret = (zfs_promote(zhp) != 0);


3157         zfs_close(zhp);
3158         return (ret);
3159 }
```

```
3161 /*
3162  * zfs rollback [-rRf] <snapshot>
3163  *
3164  *      -r      Delete any intervening snapshots before doing rollback
3165  *      -R      Delete any snapshots and their clones
3166  *      -f      ignored for backwards compatability
3167  *
3168  * Given a filesystem, rollback to a specific snapshot, discarding any changes
3169  * since then and making it the active dataset.  If more recent snapshots exist,
3170  * the command will complain unless the '-r' flag is given.
3171  */
3172 typedef struct rollback_cbdata {
3173         uint64_t        cb_create;
3174         boolean_t       cb_first;
3175         int             cb_doclones;
3176         char            *cb_target;
3177         int             cb_error;
3178         boolean_t       cb_recurse;
3179         boolean_t       cb_dependent;
3180 } rollback_cbdata_t;

3182 /*
3183  * Report any snapshots more recent than the one specified.  Used when '-r' is
3184  * not specified.  We reuse this same callback for the snapshot dependents - if
3185  * 'cb_dependent' is set, then this is a dependent and we should report it
3186  * without checking the transaction group.
3187  */
3188 static int
3189 rollback_check(zfs_handle_t *zhp, void *data)
3190 {
3191         rollback_cbdata_t *cbp = data;

3193         if (cbp->cb_doclones) {
3194                 zfs_close(zhp);
3195                 return (0);
3196         }

3198         if (!cbp->cb_dependent) {
3199                 if (strcmp(zfs_get_name(zhp), cbp->cb_target) != 0 &&
3200                     zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
3201                     zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) >
3202                     cbp->cb_create) {

3204                         if (cbp->cb_first && !cbp->cb_recurse) {
3205                                 (void) fprintf(stderr, gettext("cannot "
3206                                     "rollback to '%s': more recent snapshots "
3207                                     "exist\n"),
3208                                     cbp->cb_target);
3209                                 (void) fprintf(stderr, gettext("use '-r' to "
3210                                     "force deletion of the following "
3211                                     "snapshots:\n"));
3212                                 cbp->cb_first = 0;
3213                                 cbp->cb_error = 1;
3214                         }

3216                         if (cbp->cb_recurse) {
3217                                 cbp->cb_dependent = B_TRUE;
3218                                 if (zfs_iter_dependents(zhp, B_TRUE,
3219                                     rollback_check, cbp) != 0) {
3220                                         zfs_close(zhp);
3221                                         return (-1);
3222                                 }
3223                                 cbp->cb_dependent = B_FALSE;
3224                         } else {
3225                                 (void) fprintf(stderr, "%s\n",
3226                                     zfs_get_name(zhp));
```



```
3227                         }
3228                 }
3229         } else {
3230                 if (cbp->cb_first && cbp->cb_recurse) {
3231                         (void) fprintf(stderr, gettext("cannot rollback to "
3232                             "'%s': clones of previous snapshots exist\n"),
3233                             cbp->cb_target);
3234                         (void) fprintf(stderr, gettext("use '-R' to "
3235                             "force deletion of the following clones and "
3236                             "dependents:\n"));
3237                         cbp->cb_first = 0;
3238                         cbp->cb_error = 1;
3239                 }

3241                 (void) fprintf(stderr, "%s\n", zfs_get_name(zhp));
3242         }

3244         zfs_close(zhp);
3245         return (0);
3246 }

3248 static int
3249 zfs_do_rollback(int argc, char **argv)
3250 {
3251         int ret = 0;
3252         int c;
3253         boolean_t force = B_FALSE;
3254         rollback_cbdata_t cb = { 0 };
3255         zfs_handle_t *zhp, *snap;
3256         char parentname[ZFS_MAXNAMELEN];
3257         char *delim;

3259         /* check options */
3260         while ((c = getopt(argc, argv, "rRf")) != -1) {
3261                 switch (c) {
3262                 case 'r':
3263                         cb.cb_recurse = 1;
3264                         break;
3265                 case 'R':
3266                         cb.cb_recurse = 1;
3267                         cb.cb_doclones = 1;
3268                         break;
3269                 case 'f':
3270                         force = B_TRUE;
3271                         break;
3272                 case '?':
3273                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3274                             optopt);
3275                         usage(B_FALSE);
3276                 }
3277         }

3279         argc -= optind;
3280         argv += optind;

3282         /* check number of arguments */
3283         if (argc < 1) {
3284                 (void) fprintf(stderr, gettext("missing dataset argument\n"));
3285                 usage(B_FALSE);
3286         }
3287         if (argc > 1) {
3288                 (void) fprintf(stderr, gettext("too many arguments\n"));
3289                 usage(B_FALSE);
3290         }

3292         /* open the snapshot */
```

```
3293         if ((snap = zfs_open(g_zfs, argv[0], ZFS_TYPE_SNAPSHOT)) == NULL)
3294                 return (1);

3296         /* open the parent dataset */
3297         (void) strlcpy(parentname, argv[0], sizeof (parentname));
3298         verify((delim = strrchr(parentname, '@')) != NULL);
3299         *delim = '\0';
3300         if ((zhp = zfs_open(g_zfs, parentname, ZFS_TYPE_DATASET)) == NULL) {
3301                 zfs_close(snap);
3302                 return (1);
3303         }

3305         /*
3306          * Check for more recent snapshots and/or clones based on the presence
3307          * of '-r' and '-R'.
3308          */
3309         cb.cb_target = argv[0];
3310         cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3311         cb.cb_first = B_TRUE;
3312         cb.cb_error = 0;
3313         if ((ret = zfs_iter_children(zhp, rollback_check, &cb)) != 0)
3314                 goto out;

3316         if ((ret = cb.cb_error) != 0)
3317                 goto out;

3319         /*
3320          * Rollback parent to the given snapshot.
3321          */
3322         ret = zfs_rollback(zhp, snap, force);

3324 out:
3325         zfs_close(snap);
3326         zfs_close(zhp);

3328         if (ret == 0)
3329                 return (0);
3330         else
3331                 return (1);
3332 }

3334 /*
3335  * zfs set property=value { fs | snap | vol } ...
3336  *
3337  * Sets the given property for all datasets specified on the command line.
3338  */
3339 typedef struct set_cbdata {
3340         char            *cb_propname;
3341         char            *cb_value;
3342 } set_cbdata_t;

3344 static int
3345 set_callback(zfs_handle_t *zhp, void *data)
3346 {
3347         set_cbdata_t *cbp = data;

3349         if (zfs_prop_set(zhp, cbp->cb_propname, cbp->cb_value) != 0) {
3350                 switch (libzfs_errno(g_zfs)) {
3351                 case EZFS_MOUNTFAILED:
3352                         (void) fprintf(stderr, gettext("property may be set "
3353                             "but unable to remount filesystem\n"));
3354                         break;
3355                 case EZFS_SHARENFSFAILED:
3356                         (void) fprintf(stderr, gettext("property may be set "
3357                             "but unable to reshare filesystem\n"));
3358                         break;
```



```
3359                 }
3360                 return (1);
3361         }
3362         return (0);
3363 }

3365 static int
3366 zfs_do_set(int argc, char **argv)
3367 {
3368         set_cbdata_t cb;
3369         int ret = 0;

3371         /* check for options */
3372         if (argc > 1 && argv[1][0] == '-') {
3373                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3374                     argv[1][1]);
3375                 usage(B_FALSE);
3376         }

3378         /* check number of arguments */
3379         if (argc < 2) {
3380                 (void) fprintf(stderr, gettext("missing property=value "
3381                     "argument\n"));
3382                 usage(B_FALSE);
3383         }
3384         if (argc < 3) {
3385                 (void) fprintf(stderr, gettext("missing dataset name\n"));
3386                 usage(B_FALSE);
3387         }

3389         /* validate property=value argument */
3390         cb.cb_propname = argv[1];
3391         if (((cb.cb_value = strchr(cb.cb_propname, '=')) == NULL) ||
3392             (cb.cb_value[1] == '\0')) {
3393                 (void) fprintf(stderr, gettext("missing value in "
3394                     "property=value argument\n"));
3395                 usage(B_FALSE);
3396         }

3398         *cb.cb_value = '\0';
3399         cb.cb_value++;

3401         if (*cb.cb_propname == '\0') {
3402                 (void) fprintf(stderr,
3403                     gettext("missing property in property=value argument\n"));
3404                 usage(B_FALSE);
3405         }

3407         ret = zfs_for_each(argc - 2, argv + 2, NULL,
3408             ZFS_TYPE_DATASET, NULL, NULL, 0, set_callback, &cb);

3410         return (ret);
3411 }

3413 typedef struct snap_cbdata {
3414         nvlist_t *sd_nvl;
3415         boolean_t sd_recursive;
3416         const char *sd_snapname;
3417 } snap_cbdata_t;

3419 static int
3420 zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
3421 {
3422         snap_cbdata_t *sd = arg;
3423         char *name;
3424         int rv = 0;
```

```
3425         int error;

3427         error = asprintf(&name, "%s@%s", zfs_get_name(zhp), sd->sd_snapname);
3428         if (error == -1)
3429                 nomem();
3430         fnvlist_add_boolean(sd->sd_nvl, name);
3431         free(name);

3433         if (sd->sd_recursive)
3434                 rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
3435         zfs_close(zhp);
3436         return (rv);
3437 }

3439 /*
3440  * zfs snapshot [-r] [-o prop=value] ... <fs@snap>
3441  *
3442  * Creates a snapshot with the given name.  While functionally equivalent to
3443  * 'zfs create', it is a separate command to differentiate intent.
3444  */
3445 static int
3446 zfs_do_snapshot(int argc, char **argv)
3447 {
3448         int ret = 0;
3449         char c;
3450         nvlist_t *props;
3451         snap_cbdata_t sd = { 0 };
3452         boolean_t multiple_snaps = B_FALSE;

3454         if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
3455                 nomem();
3456         if (nvlist_alloc(&sd.sd_nvl, NV_UNIQUE_NAME, 0) != 0)
3457                 nomem();

3459         /* check options */
3460         while ((c = getopt(argc, argv, "ro:")) != -1) {
3461                 switch (c) {
3462                 case 'o':
3463                         if (parseprop(props))
3464                                 return (1);
3465                         break;
3466                 case 'r':
3467                         sd.sd_recursive = B_TRUE;
3468                         multiple_snaps = B_TRUE;
3469                         break;
3470                 case '?':
3471                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3472                             optopt);
3473                         goto usage;
3474                 }
3475         }

3477         argc -= optind;
3478         argv += optind;

3480         /* check number of arguments */
3481         if (argc < 1) {
3482                 (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3483                 goto usage;
3484         }

3486         if (argc > 1)
3487                 multiple_snaps = B_TRUE;
3488         for (; argc > 0; argc--, argv++) {
3489                 char *atp;
3490                 zfs_handle_t *zhp;
```



```
3492                 atp = strchr(argv[0], '@');
3493                 if (atp == NULL)
3494                         goto usage;
3495                 *atp = '\0';
3496                 sd.sd_snapname = atp + 1;
3497                 zhp = zfs_open(g_zfs, argv[0],
3498                     ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3499                 if (zhp == NULL)
3500                         goto usage;
3501                 if (zfs_snapshot_cb(zhp, &sd) != 0)
3502                         goto usage;
3503         }

3505         ret = zfs_snapshot_nvl(g_zfs, sd.sd_nvl, props);
3506         nvlist_free(sd.sd_nvl);
3507         nvlist_free(props);
3508         if (ret != 0 && multiple_snaps)
3509                 (void) fprintf(stderr, gettext("no snapshots were created\n"));
3510         return (ret != 0);

3512 usage:
3513         nvlist_free(sd.sd_nvl);
3514         nvlist_free(props);
3515         usage(B_FALSE);
3516         return (-1);
3517 }

3519 /*
3520  * Send a backup stream to stdout.
3521  */
3522 static int
3523 zfs_do_send(int argc, char **argv)
3524 {
3525         char *fromname = NULL;
3526         char *toname = NULL;
3527         char *cp;
3528         zfs_handle_t *zhp;
3529         sendflags_t flags = { 0 };
3530         int c, err;
3531         nvlist_t *dbgnv = NULL;
3532         boolean_t extraverbose = B_FALSE;

3534         /* check options */
3535         while ((c = getopt(argc, argv, ":i:I:RDpvnP")) != -1) {
3536                 switch (c) {
3537                 case 'i':
3538                         if (fromname)
3539                                 usage(B_FALSE);
3540                         fromname = optarg;
3541                         break;
3542                 case 'I':
3543                         if (fromname)
3544                                 usage(B_FALSE);
3545                         fromname = optarg;
3546                         flags.doall = B_TRUE;
3547                         break;
3548                 case 'R':
3549                         flags.replicate = B_TRUE;
3550                         break;
3551                 case 'p':
3552                         flags.props = B_TRUE;
3553                         break;
3554                 case 'P':
3555                         flags.parsable = B_TRUE;
3556                         flags.verbose = B_TRUE;
```

```
3557                        break;
3558                case 'v':
3559                        if (flags.verbose)
3560                                extraverbose = B_TRUE;
3561                        flags.verbose = B_TRUE;
3562                        flags.progress = B_TRUE;
3563                        break;
3564                case 'D':
3565                        flags.dedup = B_TRUE;
3566                        break;
3567                case 'n':
3568                        flags.dryrun = B_TRUE;
3569                        break;
3570                case ':':
3571                        (void) fprintf(stderr, gettext("missing argument for "
3572                            "'%c' option\n"), optopt);
3573                        usage(B_FALSE);
3574                        break;
3575                case '?':
3576                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3577                            optopt);
3578                        usage(B_FALSE);
3579                }
3580        }

3582        argc -= optind;
3583        argv += optind;

3585        /* check number of arguments */
3586        if (argc < 1) {
3587                (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3588                usage(B_FALSE);
3589        }
3590        if (argc > 1) {
3591                (void) fprintf(stderr, gettext("too many arguments\n"));
3592                usage(B_FALSE);
3593        }

3595        if (!flags.dryrun && isatty(STDOUT_FILENO)) {
3596                (void) fprintf(stderr,
3597                    gettext("Error: Stream can not be written to a terminal.\n"
3598                    "You must redirect standard output.\n"));
3599                return (1);
3600        }

3602        cp = strchr(argv[0], '@');
3603        if (cp == NULL) {
3604                (void) fprintf(stderr,
3605                    gettext("argument must be a snapshot\n"));
3606                usage(B_FALSE);
3607        }
3608        *cp = '\0';
3609        toname = cp + 1;
3610        zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3611        if (zhp == NULL)
3612                return (1);

3614        /*
3615         * If they specified the full path to the snapshot, chop off
3616         * everything except the short name of the snapshot, but special
3617         * case if they specify the origin.
3618         */
3619        if (fromname && (cp = strchr(fromname, '@')) != NULL) {
3620                char origin[ZFS_MAXNAMELEN];
3621                zprop_source_t src;
```



```
3623                (void) zfs_prop_get(zhp, ZFS_PROP_ORIGIN,
3624                    origin, sizeof (origin), &src, NULL, 0, B_FALSE);

3626                if (strcmp(origin, fromname) == 0) {
3627                        fromname = NULL;
3628                        flags.fromorigin = B_TRUE;
3629                } else {
3630                        *cp = '\0';
3631                        if (cp != fromname && strcmp(argv[0], fromname)) {
3632                                (void) fprintf(stderr,
3633                                    gettext("incremental source must be "
3634                                    "in same filesystem\n"));
3635                                usage(B_FALSE);
3636                        }
3637                        fromname = cp + 1;
3638                        if (strchr(fromname, '@') || strchr(fromname, '/')) {
3639                                (void) fprintf(stderr,
3640                                    gettext("invalid incremental source\n"));
3641                                usage(B_FALSE);
3642                        }
3643                }
3644        }

3646        if (flags.replicate && fromname == NULL)
3647                flags.doall = B_TRUE;

3649        err = zfs_send(zhp, fromname, toname, &flags, STDOUT_FILENO, NULL, 0,
3650            extraverbose ? &dbgnv : NULL);

3652        if (extraverbose && dbgnv != NULL) {
3653                /*
3654                 * dump_nvlist prints to stdout, but that's been
3655                 * redirected to a file.  Make it print to stderr
3656                 * instead.
3657                 */
3658                (void) dup2(STDERR_FILENO, STDOUT_FILENO);
3659                dump_nvlist(dbgnv, 0);
3660                nvlist_free(dbgnv);
3661        }
3662        zfs_close(zhp);

3664        return (err != 0);
3665 }

3667 /*
3668  * zfs receive [-vnFu] [-d | -e] <fs@snap>
3669  *
3670  * Restore a backup stream from stdin.
3671  */
3672 static int
3673 zfs_do_receive(int argc, char **argv)
3674 {
3675        int c, err;
3676        recvflags_t flags = { 0 };

3678        /* check options */
3679        while ((c = getopt(argc, argv, ":denuvF")) != -1) {
3680                switch (c) {
3681                case 'd':
3682                        flags.isprefix = B_TRUE;
3683                        break;
3684                case 'e':
3685                        flags.isprefix = B_TRUE;
3686                        flags.istail = B_TRUE;
3687                        break;
3688                case 'n':
```

```
3689                        flags.dryrun = B_TRUE;
3690                        break;
3691                case 'u':
3692                        flags.nomount = B_TRUE;
3693                        break;
3694                case 'v':
3695                        flags.verbose = B_TRUE;
3696                        break;
3697                case 'F':
3698                        flags.force = B_TRUE;
3699                        break;
3700                case ':':
3701                        (void) fprintf(stderr, gettext("missing argument for "
3702                            "'%c' option\n"), optopt);
3703                        usage(B_FALSE);
3704                        break;
3705                case '?':
3706                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3707                            optopt);
3708                        usage(B_FALSE);
3709                }
3710        }

3712        argc -= optind;
3713        argv += optind;

3715        /* check number of arguments */
3716        if (argc < 1) {
3717                (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3718                usage(B_FALSE);
3719        }
3720        if (argc > 1) {
3721                (void) fprintf(stderr, gettext("too many arguments\n"));
3722                usage(B_FALSE);
3723        }

3725        if (isatty(STDIN_FILENO)) {
3726                (void) fprintf(stderr,
3727                    gettext("Error: Backup stream can not be read "
3728                    "from a terminal.\n"
3729                    "You must redirect standard input.\n"));
3730                return (1);
3731        }

3733        err = zfs_receive(g_zfs, argv[0], &flags, STDIN_FILENO, NULL);

3735        return (err != 0);
3736 }

3738 /*
3739  * Send a backup stream to stdout in fits format.
3740  */
3741 static int
3742 zfs_do_fits_send(int argc, char **argv)
3743 {
3744        char *fromname = NULL;
3745        char *toname = NULL;
3746        char *cp;
3747        zfs_handle_t *zhp;
3748        sendflags_t flags = { 0 };
3749        int c, err;

3751        /* check options */
3752        while ((c = getopt(argc, argv, ":i:v")) != -1) {
3753                switch (c) {
3754                case 'i':
```



```
3755                        if (fromname)
3756                                usage(B_FALSE);
3757                        fromname = optarg;
3758                        break;
3759                case 'v':
3760                        flags.verbose = B_TRUE;
3761                        break;
3762                case ':':
3763                        (void) fprintf(stderr, gettext("missing argument for "
3764                            "'%c' option\n"), optopt);
3765                        usage(B_FALSE);
3766                        break;
3767                case '?':
3768                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3769                            optopt);
3770                        usage(B_FALSE);
3771                }
3772        }

3774        argc -= optind;
3775        argv += optind;

3777        /* check number of arguments */
3778        if (argc < 1) {
3779                (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3780                usage(B_FALSE);
3781        }
3782        if (argc > 1) {
3783                (void) fprintf(stderr, gettext("too many arguments\n"));
3784                usage(B_FALSE);
3785        }

3787        if (isatty(STDOUT_FILENO)) {
3788                (void) fprintf(stderr,
3789                    gettext("Error: Stream can not be written to a terminal.\n"
3790                    "You must redirect standard output.\n"));
3791                return (1);
3792        }

3794        cp = strchr(argv[0], '@');
3795        if (cp == NULL) {
3796                (void) fprintf(stderr,
3797                    gettext("argument must be a snapshot\n"));
3798                usage(B_FALSE);
3799        }
3800        *cp = '\0';
3801        toname = cp + 1;
3802        zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_FILESYSTEM);
3803        if (zhp == NULL)
3804                return (1);

3806        /*
3807         * If they specified the full path to the snapshot, chop off
3808         * everything except the short name of the snapshot, but special
3809         * case if they specify the origin.
3810         */
3811        if (fromname && (cp = strchr(fromname, '@')) != NULL) {
3812                char origin[ZFS_MAXNAMELEN];
3813                zprop_source_t src;

3815                (void) zfs_prop_get(zhp, ZFS_PROP_ORIGIN,
3816                    origin, sizeof (origin), &src, NULL, 0, B_FALSE);

3818                if (strcmp(origin, fromname) == 0) {
3819                        fromname = NULL;
3820                        flags.fromorigin = B_TRUE;
```

```
3821                } else {
3822                        *cp = '\0';
3823                        if (cp != fromname && strcmp(argv[0], fromname)) {
3824                                (void) fprintf(stderr,
3825                                    gettext("incremental source must be "
3826                                    "in same filesystem\n"));
3827                                usage(B_FALSE);
3828                        }
3829                        fromname = cp + 1;
3830                        if (strchr(fromname, '@') || strchr(fromname, '/')) {
3831                                (void) fprintf(stderr,
3832                                    gettext("invalid incremental source\n"));
3833                                usage(B_FALSE);
3834                        }
3835                }
3836        }

3838        err = zfs_fits_send(zhp, fromname, toname, &flags, STDOUT_FILENO,
3839            NULL, 0);

3841        zfs_close(zhp);

3843        return (err != 0);
3844 }

3846 /*
3847 #endif /* ! codereview */
3848  * allow/unallow stuff
3849  */
3850 /* copied from zfs/sys/dsl_deleg.h */
3851 #define ZFS_DELEG_PERM_CREATE           "create"
3852 #define ZFS_DELEG_PERM_DESTROY          "destroy"
3853 #define ZFS_DELEG_PERM_SNAPSHOT         "snapshot"
3854 #define ZFS_DELEG_PERM_ROLLBACK         "rollback"
3855 #define ZFS_DELEG_PERM_CLONE            "clone"
3856 #define ZFS_DELEG_PERM_PROMOTE          "promote"
3857 #define ZFS_DELEG_PERM_RENAME           "rename"
3858 #define ZFS_DELEG_PERM_MOUNT            "mount"
3859 #define ZFS_DELEG_PERM_SHARE            "share"
3860 #define ZFS_DELEG_PERM_SEND             "send"
3861 #define ZFS_DELEG_PERM_RECEIVE          "receive"
3862 #define ZFS_DELEG_PERM_ALLOW            "allow"
3863 #define ZFS_DELEG_PERM_USERPROP         "userprop"
3864 #define ZFS_DELEG_PERM_VSCAN            "vscan" /* ??? */
3865 #define ZFS_DELEG_PERM_USERQUOTA        "userquota"
3866 #define ZFS_DELEG_PERM_GROUPQUOTA       "groupquota"
3867 #define ZFS_DELEG_PERM_USERUSED         "userused"
3868 #define ZFS_DELEG_PERM_GROUPUSED        "groupused"
3869 #define ZFS_DELEG_PERM_HOLD             "hold"
3870 #define ZFS_DELEG_PERM_RELEASE          "release"
3871 #define ZFS_DELEG_PERM_DIFF             "diff"

3873 #define ZFS_NUM_DELEG_NOTES ZFS_DELEG_NOTE_NONE

3875 static zfs_deleg_perm_tab_t zfs_deleg_perm_tbl[] = {
3876         { ZFS_DELEG_PERM_ALLOW, ZFS_DELEG_NOTE_ALLOW },
3877         { ZFS_DELEG_PERM_CLONE, ZFS_DELEG_NOTE_CLONE },
3878         { ZFS_DELEG_PERM_CREATE, ZFS_DELEG_NOTE_CREATE },
3879         { ZFS_DELEG_PERM_DESTROY, ZFS_DELEG_NOTE_DESTROY },
3880         { ZFS_DELEG_PERM_DIFF, ZFS_DELEG_NOTE_DIFF},
3881         { ZFS_DELEG_PERM_HOLD, ZFS_DELEG_NOTE_HOLD },
3882         { ZFS_DELEG_PERM_MOUNT, ZFS_DELEG_NOTE_MOUNT },
3883         { ZFS_DELEG_PERM_PROMOTE, ZFS_DELEG_NOTE_PROMOTE },
3884         { ZFS_DELEG_PERM_RECEIVE, ZFS_DELEG_NOTE_RECEIVE },
3885         { ZFS_DELEG_PERM_RELEASE, ZFS_DELEG_NOTE_RELEASE },
3886         { ZFS_DELEG_PERM_RENAME, ZFS_DELEG_NOTE_RENAME },
```



```
3887         { ZFS_DELEG_PERM_ROLLBACK, ZFS_DELEG_NOTE_ROLLBACK },
3888         { ZFS_DELEG_PERM_SEND, ZFS_DELEG_NOTE_SEND },
3889         { ZFS_DELEG_PERM_SHARE, ZFS_DELEG_NOTE_SHARE },
3890         { ZFS_DELEG_PERM_SNAPSHOT, ZFS_DELEG_NOTE_SNAPSHOT },

3892         { ZFS_DELEG_PERM_GROUPQUOTA, ZFS_DELEG_NOTE_GROUPQUOTA },
3893         { ZFS_DELEG_PERM_GROUPUSED, ZFS_DELEG_NOTE_GROUPUSED },
3894         { ZFS_DELEG_PERM_USERPROP, ZFS_DELEG_NOTE_USERPROP },
3895         { ZFS_DELEG_PERM_USERQUOTA, ZFS_DELEG_NOTE_USERQUOTA },
3896         { ZFS_DELEG_PERM_USERUSED, ZFS_DELEG_NOTE_USERUSED },
3897         { NULL, ZFS_DELEG_NOTE_NONE }
3898 };

3900 /* permission structure */
3901 typedef struct deleg_perm {
3902         zfs_deleg_who_type_t    dp_who_type;
3903         const char              *dp_name;
3904         boolean_t               dp_local;
3905         boolean_t               dp_descend;
3906 } deleg_perm_t;

3908 /* */
3909 typedef struct deleg_perm_node {
3910         deleg_perm_t            dpn_perm;

3912         uu_avl_node_t           dpn_avl_node;
3913 } deleg_perm_node_t;

3915 typedef struct fs_perm fs_perm_t;

3917 /* permissions set */
3918 typedef struct who_perm {
3919         zfs_deleg_who_type_t    who_type;
3920         const char              *who_name;              /* id */
3921         char                    who_ug_name[256];       /* user/group name */
3922         fs_perm_t               *who_fsperm;            /* uplink */

3924         uu_avl_t                *who_deleg_perm_avl;    /* permissions */
3925 } who_perm_t;

3927 /* */
3928 typedef struct who_perm_node {
3929         who_perm_t      who_perm;
3930         uu_avl_node_t   who_avl_node;
3931 } who_perm_node_t;

3933 typedef struct fs_perm_set fs_perm_set_t;
3934 /* fs permissions */
3935 struct fs_perm {
3936         const char              *fsp_name;

3938         uu_avl_t                *fsp_sc_avl;    /* sets,create */
3939         uu_avl_t                *fsp_uge_avl;   /* user,group,everyone */

3941         fs_perm_set_t           *fsp_set;       /* uplink */
3942 };

3944 /* */
3945 typedef struct fs_perm_node {
3946         fs_perm_t       fspn_fsperm;
3947         uu_avl_t        *fspn_avl;

3949         uu_list_node_t  fspn_list_node;
3950 } fs_perm_node_t;

3952 /* top level structure */
```

```
3953 struct fs_perm_set {
3954         uu_list_pool_t  *fsps_list_pool;
3955         uu_list_t       *fsps_list; /* list of fs_perms */

3957         uu_avl_pool_t   *fsps_named_set_avl_pool;
3958         uu_avl_pool_t   *fsps_who_perm_avl_pool;
3959         uu_avl_pool_t   *fsps_deleg_perm_avl_pool;
3960 };

3962 static inline const char *
3963 deleg_perm_type(zfs_deleg_note_t note)
3964 {
3965         /* subcommands */
3966         switch (note) {
3967                 /* SUBCOMMANDS */
3968                 /* OTHER */
3969         case ZFS_DELEG_NOTE_GROUPQUOTA:
3970         case ZFS_DELEG_NOTE_GROUPUSED:
3971         case ZFS_DELEG_NOTE_USERPROP:
3972         case ZFS_DELEG_NOTE_USERQUOTA:
3973         case ZFS_DELEG_NOTE_USERUSED:
3974                 /* other */
3975                 return (gettext("other"));
3976         default:
3977                 return (gettext("subcommand"));
3978         }
3979 }

3981 static int inline
3982 who_type2weight(zfs_deleg_who_type_t who_type)
3983 {
3984         int res;
3985         switch (who_type) {
3986                 case ZFS_DELEG_NAMED_SET_SETS:
3987                 case ZFS_DELEG_NAMED_SET:
3988                         res = 0;
3989                         break;
3990                 case ZFS_DELEG_CREATE_SETS:
3991                 case ZFS_DELEG_CREATE:
3992                         res = 1;
3993                         break;
3994                 case ZFS_DELEG_USER_SETS:
3995                 case ZFS_DELEG_USER:
3996                         res = 2;
3997                         break;
3998                 case ZFS_DELEG_GROUP_SETS:
3999                 case ZFS_DELEG_GROUP:
4000                         res = 3;
4001                         break;
4002                 case ZFS_DELEG_EVERYONE_SETS:
4003                 case ZFS_DELEG_EVERYONE:
4004                         res = 4;
4005                         break;
4006                 default:
4007                         res = -1;
4008         }

4010         return (res);
4011 }

4013 /* ARGSUSED */
4014 static int
4015 who_perm_compare(const void *larg, const void *rarg, void *unused)
4016 {
4017         const who_perm_node_t *l = larg;
4018         const who_perm_node_t *r = rarg;
```



```
4019         zfs_deleg_who_type_t ltype = l->who_perm.who_type;
4020         zfs_deleg_who_type_t rtype = r->who_perm.who_type;
4021         int lweight = who_type2weight(ltype);
4022         int rweight = who_type2weight(rtype);
4023         int res = lweight - rweight;
4024         if (res == 0)
4025                 res = strncmp(l->who_perm.who_name, r->who_perm.who_name,
4026                     ZFS_MAX_DELEG_NAME-1);

4028         if (res == 0)
4029                 return (0);
4030         if (res > 0)
4031                 return (1);
4032         else
4033                 return (-1);
4034 }

4036 /* ARGSUSED */
4037 static int
4038 deleg_perm_compare(const void *larg, const void *rarg, void *unused)
4039 {
4040         const deleg_perm_node_t *l = larg;
4041         const deleg_perm_node_t *r = rarg;
4042         int res =  strncmp(l->dpn_perm.dp_name, r->dpn_perm.dp_name,
4043             ZFS_MAX_DELEG_NAME-1);

4045         if (res == 0)
4046                 return (0);

4048         if (res > 0)
4049                 return (1);
4050         else
4051                 return (-1);
4052 }

4054 static inline void
4055 fs_perm_set_init(fs_perm_set_t *fspset)
4056 {
4057         bzero(fspset, sizeof (fs_perm_set_t));

4059         if ((fspset->fsps_list_pool = uu_list_pool_create("fsps_list_pool",
4060             sizeof (fs_perm_node_t), offsetof(fs_perm_node_t, fspn_list_node),
4061             NULL, UU_DEFAULT)) == NULL)
4062                 nomem();
4063         if ((fspset->fsps_list = uu_list_create(fspset->fsps_list_pool, NULL,
4064             UU_DEFAULT)) == NULL)
4065                 nomem();

4067         if ((fspset->fsps_named_set_avl_pool = uu_avl_pool_create(
4068             "named_set_avl_pool", sizeof (who_perm_node_t), offsetof(
4069             who_perm_node_t, who_avl_node), who_perm_compare,
4070             UU_DEFAULT)) == NULL)
4071                 nomem();

4073         if ((fspset->fsps_who_perm_avl_pool = uu_avl_pool_create(
4074             "who_perm_avl_pool", sizeof (who_perm_node_t), offsetof(
4075             who_perm_node_t, who_avl_node), who_perm_compare,
4076             UU_DEFAULT)) == NULL)
4077                 nomem();

4079         if ((fspset->fsps_deleg_perm_avl_pool = uu_avl_pool_create(
4080             "deleg_perm_avl_pool", sizeof (deleg_perm_node_t), offsetof(
4081             deleg_perm_node_t, dpn_avl_node), deleg_perm_compare, UU_DEFAULT))
4082             == NULL)
4083                 nomem();
4084 }
```

```
4086 static inline void fs_perm_fini(fs_perm_t *);
4087 static inline void who_perm_fini(who_perm_t *);

4089 static inline void
4090 fs_perm_set_fini(fs_perm_set_t *fspset)
4091 {
4092         fs_perm_node_t *node = uu_list_first(fspset->fsps_list);

4094         while (node != NULL) {
4095                 fs_perm_node_t *next_node =
4096                     uu_list_next(fspset->fsps_list, node);
4097                 fs_perm_t *fsperm = &node->fspn_fsperm;
4098                 fs_perm_fini(fsperm);
4099                 uu_list_remove(fspset->fsps_list, node);
4100                 free(node);
4101                 node = next_node;
4102         }

4104         uu_avl_pool_destroy(fspset->fsps_named_set_avl_pool);
4105         uu_avl_pool_destroy(fspset->fsps_who_perm_avl_pool);
4106         uu_avl_pool_destroy(fspset->fsps_deleg_perm_avl_pool);
4107 }

4109 static inline void
4110 deleg_perm_init(deleg_perm_t *deleg_perm, zfs_deleg_who_type_t type,
4111     const char *name)
4112 {
4113         deleg_perm->dp_who_type = type;
4114         deleg_perm->dp_name = name;
4115 }

4117 static inline void
4118 who_perm_init(who_perm_t *who_perm, fs_perm_t *fsperm,
4119     zfs_deleg_who_type_t type, const char *name)
4120 {
4121         uu_avl_pool_t   *pool;
4122         pool = fsperm->fsp_set->fsps_deleg_perm_avl_pool;

4124         bzero(who_perm, sizeof (who_perm_t));

4126         if ((who_perm->who_deleg_perm_avl = uu_avl_create(pool, NULL,
4127             UU_DEFAULT)) == NULL)
4128                 nomem();

4130         who_perm->who_type = type;
4131         who_perm->who_name = name;
4132         who_perm->who_fsperm = fsperm;
4133 }

4135 static inline void
4136 who_perm_fini(who_perm_t *who_perm)
4137 {
4138         deleg_perm_node_t *node = uu_avl_first(who_perm->who_deleg_perm_avl);

4140         while (node != NULL) {
4141                 deleg_perm_node_t *next_node =
4142                     uu_avl_next(who_perm->who_deleg_perm_avl, node);

4144                 uu_avl_remove(who_perm->who_deleg_perm_avl, node);
4145                 free(node);
4146                 node = next_node;
4147         }

4149         uu_avl_destroy(who_perm->who_deleg_perm_avl);
4150 }
```



```
4152 static inline void
4153 fs_perm_init(fs_perm_t *fsperm, fs_perm_set_t *fspset, const char *fsname)
4154 {
4155         uu_avl_pool_t   *nset_pool = fspset->fsps_named_set_avl_pool;
4156         uu_avl_pool_t   *who_pool = fspset->fsps_who_perm_avl_pool;

4158         bzero(fsperm, sizeof (fs_perm_t));

4160         if ((fsperm->fsp_sc_avl = uu_avl_create(nset_pool, NULL, UU_DEFAULT))
4161             == NULL)
4162                 nomem();

4164         if ((fsperm->fsp_uge_avl = uu_avl_create(who_pool, NULL, UU_DEFAULT))
4165             == NULL)
4166                 nomem();

4168         fsperm->fsp_set = fspset;
4169         fsperm->fsp_name = fsname;
4170 }

4172 static inline void
4173 fs_perm_fini(fs_perm_t *fsperm)
4174 {
4175         who_perm_node_t *node = uu_avl_first(fsperm->fsp_sc_avl);
4176         while (node != NULL) {
4177                 who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_sc_avl,
4178                     node);
4179                 who_perm_t *who_perm = &node->who_perm;
4180                 who_perm_fini(who_perm);
4181                 uu_avl_remove(fsperm->fsp_sc_avl, node);
4182                 free(node);
4183                 node = next_node;
4184         }

4186         node = uu_avl_first(fsperm->fsp_uge_avl);
4187         while (node != NULL) {
4188                 who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_uge_avl,
4189                     node);
4190                 who_perm_t *who_perm = &node->who_perm;
4191                 who_perm_fini(who_perm);
4192                 uu_avl_remove(fsperm->fsp_uge_avl, node);
4193                 free(node);
4194                 node = next_node;
4195         }

4197         uu_avl_destroy(fsperm->fsp_sc_avl);
4198         uu_avl_destroy(fsperm->fsp_uge_avl);
4199 }

4201 static void inline
4202 set_deleg_perm_node(uu_avl_t *avl, deleg_perm_node_t *node,
4203     zfs_deleg_who_type_t who_type, const char *name, char locality)
4204 {
4205         uu_avl_index_t idx = 0;

4207         deleg_perm_node_t *found_node = NULL;
4208         deleg_perm_t    *deleg_perm = &node->dpn_perm;

4210         deleg_perm_init(deleg_perm, who_type, name);

4212         if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4213             == NULL)
4214                 uu_avl_insert(avl, node, idx);
4215         else {
4216                 node = found_node;
```

```
4217                deleg_perm = &node->dpn_perm;
4218        }


4221        switch (locality) {
4222        case ZFS_DELEG_LOCAL:
4223                deleg_perm->dp_local = B_TRUE;
4224                break;
4225        case ZFS_DELEG_DESCENDENT:
4226                deleg_perm->dp_descend = B_TRUE;
4227                break;
4228        case ZFS_DELEG_NA:
4229                break;
4230        default:
4231                assert(B_FALSE); /* invalid locality */
4232        }
4233 }

4235 static inline int
4236 parse_who_perm(who_perm_t *who_perm, nvlist_t *nvl, char locality)
4237 {
4238        nvpair_t *nvp = NULL;
4239        fs_perm_set_t *fspset = who_perm->who_fsperm->fsp_set;
4240        uu_avl_t *avl = who_perm->who_deleg_perm_avl;
4241        zfs_deleg_who_type_t who_type = who_perm->who_type;

4243        while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4244                const char *name = nvpair_name(nvp);
4245                data_type_t type = nvpair_type(nvp);
4246                uu_avl_pool_t *avl_pool = fspset->fsps_deleg_perm_avl_pool;
4247                deleg_perm_node_t *node =
4248                    safe_malloc(sizeof (deleg_perm_node_t));

4250                assert(type == DATA_TYPE_BOOLEAN);

4252                uu_avl_node_init(node, &node->dpn_avl_node, avl_pool);
4253                set_deleg_perm_node(avl, node, who_type, name, locality);
4254        }

4256        return (0);
4257 }

4259 static inline int
4260 parse_fs_perm(fs_perm_t *fsperm, nvlist_t *nvl)
4261 {
4262        nvpair_t *nvp = NULL;
4263        fs_perm_set_t *fspset = fsperm->fsp_set;

4265        while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4266                nvlist_t *nvl2 = NULL;
4267                const char *name = nvpair_name(nvp);
4268                uu_avl_t *avl = NULL;
4269                uu_avl_pool_t *avl_pool;
4270                zfs_deleg_who_type_t perm_type = name[0];
4271                char perm_locality = name[1];
4272                const char *perm_name = name + 3;
4273                boolean_t is_set = B_TRUE;
4274                who_perm_t *who_perm = NULL;

4276                assert('$' == name[2]);

4278                if (nvpair_value_nvlist(nvp, &nvl2) != 0)
4279                        return (-1);

4281                switch (perm_type) {
4282                case ZFS_DELEG_CREATE:
```



```
4283                case ZFS_DELEG_CREATE_SETS:
4284                case ZFS_DELEG_NAMED_SET:
4285                case ZFS_DELEG_NAMED_SET_SETS:
4286                        avl_pool = fspset->fsps_named_set_avl_pool;
4287                        avl = fsperm->fsp_sc_avl;
4288                        break;
4289                case ZFS_DELEG_USER:
4290                case ZFS_DELEG_USER_SETS:
4291                case ZFS_DELEG_GROUP:
4292                case ZFS_DELEG_GROUP_SETS:
4293                case ZFS_DELEG_EVERYONE:
4294                case ZFS_DELEG_EVERYONE_SETS:
4295                        avl_pool = fspset->fsps_who_perm_avl_pool;
4296                        avl = fsperm->fsp_uge_avl;
4297                        break;
4298                }

4300                if (is_set) {
4301                        who_perm_node_t *found_node = NULL;
4302                        who_perm_node_t *node = safe_malloc(
4303                            sizeof (who_perm_node_t));
4304                        who_perm = &node->who_perm;
4305                        uu_avl_index_t idx = 0;

4307                        uu_avl_node_init(node, &node->who_avl_node, avl_pool);
4308                        who_perm_init(who_perm, fsperm, perm_type, perm_name);

4310                        if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4311                            == NULL) {
4312                                if (avl == fsperm->fsp_uge_avl) {
4313                                        uid_t rid = 0;
4314                                        struct passwd *p = NULL;
4315                                        struct group *g = NULL;
4316                                        const char *nice_name = NULL;

4318                                        switch (perm_type) {
4319                                        case ZFS_DELEG_USER_SETS:
4320                                        case ZFS_DELEG_USER:
4321                                                rid = atoi(perm_name);
4322                                                p = getpwuid(rid);
4323                                                if (p)
4324                                                        nice_name = p->pw_name;
4325                                                break;
4326                                        case ZFS_DELEG_GROUP_SETS:
4327                                        case ZFS_DELEG_GROUP:
4328                                                rid = atoi(perm_name);
4329                                                g = getgrgid(rid);
4330                                                if (g)
4331                                                        nice_name = g->gr_name;
4332                                                break;
4333                                        }

4335                                        if (nice_name != NULL)
4336                                                (void) strlcpy(
4337                                                    node->who_perm.who_ug_name,
4338                                                    nice_name, 256);
4339                                }

4341                                uu_avl_insert(avl, node, idx);
4342                        } else {
4343                                node = found_node;
4344                                who_perm = &node->who_perm;
4345                        }
4346                }

4348                (void) parse_who_perm(who_perm, nvl2, perm_locality);
```

```
4349         }

4351         return (0);
4352 }

4354 static inline int
4355 parse_fs_perm_set(fs_perm_set_t *fspset, nvlist_t *nvl)
4356 {
4357         nvpair_t *nvp = NULL;
4358         uu_avl_index_t idx = 0;

4360         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4361                 nvlist_t *nvl2 = NULL;
4362                 const char *fsname = nvpair_name(nvp);
4363                 data_type_t type = nvpair_type(nvp);
4364                 fs_perm_t *fsperm = NULL;
4365                 fs_perm_node_t *node = safe_malloc(sizeof (fs_perm_node_t));
4366                 if (node == NULL)
4367                         nomem();

4369                 fsperm = &node->fspn_fsperm;

4371                 assert(DATA_TYPE_NVLIST == type);

4373                 uu_list_node_init(node, &node->fspn_list_node,
4374                     fspset->fsps_list_pool);

4376                 idx = uu_list_numnodes(fspset->fsps_list);
4377                 fs_perm_init(fsperm, fspset, fsname);

4379                 if (nvpair_value_nvlist(nvp, &nvl2) != 0)
4380                         return (-1);

4382                 (void) parse_fs_perm(fsperm, nvl2);

4384                 uu_list_insert(fspset->fsps_list, node, idx);
4385         }

4387         return (0);
4388 }

4390 static inline const char *
4391 deleg_perm_comment(zfs_deleg_note_t note)
4392 {
4393         const char *str = "";

4395         /* subcommands */
4396         switch (note) {
4397                 /* SUBCOMMANDS */
4398         case ZFS_DELEG_NOTE_ALLOW:
4399                 str = gettext("Must also have the permission that is being"
4400                     "\n\t\t\t\tallowed");
4401                 break;
4402         case ZFS_DELEG_NOTE_CLONE:
4403                 str = gettext("Must also have the 'create' ability and 'mount'"
4404                     "\n\t\t\t\tability in the origin file system");
4405                 break;
4406         case ZFS_DELEG_NOTE_CREATE:
4407                 str = gettext("Must also have the 'mount' ability");
4408                 break;
4409         case ZFS_DELEG_NOTE_DESTROY:
4410                 str = gettext("Must also have the 'mount' ability");
4411                 break;
4412         case ZFS_DELEG_NOTE_DIFF:
4413                 str = gettext("Allows lookup of paths within a dataset;"
4414                     "\n\t\t\t\tgiven an object number. Ordinary users need this"
```



```
4415                     "\n\t\t\t\tin order to use zfs diff");
4416                 break;
4417         case ZFS_DELEG_NOTE_HOLD:
4418                 str = gettext("Allows adding a user hold to a snapshot");
4419                 break;
4420         case ZFS_DELEG_NOTE_MOUNT:
4421                 str = gettext("Allows mount/umount of ZFS datasets");
4422                 break;
4423         case ZFS_DELEG_NOTE_PROMOTE:
4424                 str = gettext("Must also have the 'mount'\n\t\t\t\tand"
4425                     " 'promote' ability in the origin file system");
4426                 break;
4427         case ZFS_DELEG_NOTE_RECEIVE:
4428                 str = gettext("Must also have the 'mount' and 'create'"
4429                     " ability");
4430                 break;
4431         case ZFS_DELEG_NOTE_RELEASE:
4432                 str = gettext("Allows releasing a user hold which\n\t\t\t\t"
4433                     "might destroy the snapshot");
4434                 break;
4435         case ZFS_DELEG_NOTE_RENAME:
4436                 str = gettext("Must also have the 'mount' and 'create'"
4437                     "\n\t\t\t\tability in the new parent");
4438                 break;
4439         case ZFS_DELEG_NOTE_ROLLBACK:
4440                 str = gettext("");
4441                 break;
4442         case ZFS_DELEG_NOTE_SEND:
4443                 str = gettext("");
4444                 break;
4445         case ZFS_DELEG_NOTE_SHARE:
4446                 str = gettext("Allows sharing file systems over NFS or SMB"
4447                     "\n\t\t\t\tprotocols");
4448                 break;
4449         case ZFS_DELEG_NOTE_SNAPSHOT:
4450                 str = gettext("");
4451                 break;
4452 /*
4453  *      case ZFS_DELEG_NOTE_VSCAN:
4454  *              str = gettext("");
4455  *              break;
4456  */
4457                 /* OTHER */
4458         case ZFS_DELEG_NOTE_GROUPQUOTA:
4459                 str = gettext("Allows accessing any groupquota@... property");
4460                 break;
4461         case ZFS_DELEG_NOTE_GROUPUSED:
4462                 str = gettext("Allows reading any groupused@... property");
4463                 break;
4464         case ZFS_DELEG_NOTE_USERPROP:
4465                 str = gettext("Allows changing any user property");
4466                 break;
4467         case ZFS_DELEG_NOTE_USERQUOTA:
4468                 str = gettext("Allows accessing any userquota@... property");
4469                 break;
4470         case ZFS_DELEG_NOTE_USERUSED:
4471                 str = gettext("Allows reading any userused@... property");
4472                 break;
4473                 /* other */
4474         default:
4475                 str = "";
4476         }

4478         return (str);
4479 }
```

```
4481 struct allow_opts {
4482         boolean_t local;
4483         boolean_t descend;
4484         boolean_t user;
4485         boolean_t group;
4486         boolean_t everyone;
4487         boolean_t create;
4488         boolean_t set;
4489         boolean_t recursive; /* unallow only */
4490         boolean_t prt_usage;

4492         boolean_t prt_perms;
4493         char *who;
4494         char *perms;
4495         const char *dataset;
4496 };

4498 static inline int
4499 prop_cmp(const void *a, const void *b)
4500 {
4501         const char *str1 = *(const char **)a;
4502         const char *str2 = *(const char **)b;
4503         return (strcmp(str1, str2));
4504 }

4506 static void
4507 allow_usage(boolean_t un, boolean_t requested, const char *msg)
4508 {
4509         const char *opt_desc[] = {
4510                 "-h", gettext("show this help message and exit"),
4511                 "-l", gettext("set permission locally"),
4512                 "-d", gettext("set permission for descents"),
4513                 "-u", gettext("set permission for user"),
4514                 "-g", gettext("set permission for group"),
4515                 "-e", gettext("set permission for everyone"),
4516                 "-c", gettext("set create time permission"),
4517                 "-s", gettext("define permission set"),
4518                 /* unallow only */
4519                 "-r", gettext("remove permissions recursively"),
4520         };
4521         size_t unallow_size = sizeof (opt_desc) / sizeof (char *);
4522         size_t allow_size = unallow_size - 2;
4523         const char *props[ZFS_NUM_PROPS];
4524         int i;
4525         size_t count = 0;
4526         FILE *fp = requested ? stdout : stderr;
4527         zprop_desc_t *pdtbl = zfs_prop_get_table();
4528         const char *fmt = gettext("%-16s %-14s\t%s\n");

4530         (void) fprintf(fp, gettext("Usage: %s\n"), get_usage(un ? HELP_UNALLOW :
4531             HELP_ALLOW));
4532         (void) fprintf(fp, gettext("Options:\n"));
4533         for (int i = 0; i < (un ? unallow_size : allow_size); i++) {
4534                 const char *opt = opt_desc[i++];
4535                 const char *optdsc = opt_desc[i];
4536                 (void) fprintf(fp, gettext("  %-10s  %s\n"), opt, optdsc);
4537         }

4539         (void) fprintf(fp, gettext("\nThe following permissions are "
4540             "supported:\n\n"));
4541         (void) fprintf(fp, fmt, gettext("NAME"), gettext("TYPE"),
4542             gettext("NOTES"));
4543         for (i = 0; i < ZFS_NUM_DELEG_NOTES; i++) {
4544                 const char *perm_name = zfs_deleg_perm_tbl[i].z_perm;
4545                 zfs_deleg_note_t perm_note = zfs_deleg_perm_tbl[i].z_note;
4546                 const char *perm_type = deleg_perm_type(perm_note);
```



```
4547                 const char *perm_comment = deleg_perm_comment(perm_note);
4548                 (void) fprintf(fp, fmt, perm_name, perm_type, perm_comment);
4549         }

4551         for (i = 0; i < ZFS_NUM_PROPS; i++) {
4552                 zprop_desc_t *pd = &pdtbl[i];
4553                 if (pd->pd_visible != B_TRUE)
4554                         continue;

4556                 if (pd->pd_attr == PROP_READONLY)
4557                         continue;

4559                 props[count++] = pd->pd_name;
4560         }
4561         props[count] = NULL;

4563         qsort(props, count, sizeof (char *), prop_cmp);

4565         for (i = 0; i < count; i++)
4566                 (void) fprintf(fp, fmt, props[i], gettext("property"), "");

4568         if (msg != NULL)
4569                 (void) fprintf(fp, gettext("\nzfs: error: %s"), msg);

4571         exit(requested ? 0 : 2);
4572 }

4574 static inline const char *
4575 munge_args(int argc, char **argv, boolean_t un, size_t expected_argc,
4576     char **permsp)
4577 {
4578         if (un && argc == expected_argc - 1)
4579                 *permsp = NULL;
4580         else if (argc == expected_argc)
4581                 *permsp = argv[argc - 2];
4582         else
4583                 allow_usage(un, B_FALSE,
4584                     gettext("wrong number of parameters\n"));

4586         return (argv[argc - 1]);
4587 }

4589 static void
4590 parse_allow_args(int argc, char **argv, boolean_t un, struct allow_opts *opts)
4591 {
4592         int uge_sum = opts->user + opts->group + opts->everyone;
4593         int csuge_sum = opts->create + opts->set + uge_sum;
4594         int ldcsuge_sum = csuge_sum + opts->local + opts->descend;
4595         int all_sum = un ? ldcsuge_sum + opts->recursive : ldcsuge_sum;

4597         if (uge_sum > 1)
4598                 allow_usage(un, B_FALSE,
4599                     gettext("-u, -g, and -e are mutually exclusive\n"));

4601         if (opts->prt_usage)
4602                 if (argc == 0 && all_sum == 0)
4603                         allow_usage(un, B_TRUE, NULL);
4604                 else
4605                         usage(B_FALSE);

4607         if (opts->set) {
4608                 if (csuge_sum > 1)
4609                         allow_usage(un, B_FALSE,
4610                             gettext("invalid options combined with -s\n"));

4612                 opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
```

```
4613                 if (argv[0][0] != '@')
4614                         allow_usage(un, B_FALSE,
4615                             gettext("invalid set name: missing '@' prefix\n"));
4616                 opts->who = argv[0];
4617         } else if (opts->create) {
4618                 if (ldcsuge_sum > 1)
4619                         allow_usage(un, B_FALSE,
4620                             gettext("invalid options combined with -c\n"));
4621                 opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4622         } else if (opts->everyone) {
4623                 if (csuge_sum > 1)
4624                         allow_usage(un, B_FALSE,
4625                             gettext("invalid options combined with -e\n"));
4626                 opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4627         } else if (uge_sum == 0 && argc > 0 && strcmp(argv[0], "everyone")
4628             == 0) {
4629                 opts->everyone = B_TRUE;
4630                 argc--;
4631                 argv++;
4632                 opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4633         } else if (argc == 1 && !un) {
4634                 opts->prt_perms = B_TRUE;
4635                 opts->dataset = argv[argc-1];
4636         } else {
4637                 opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4638                 opts->who = argv[0];
4639         }

4641         if (!opts->local && !opts->descend) {
4642                 opts->local = B_TRUE;
4643                 opts->descend = B_TRUE;
4644         }
4645 }

4647 static void
4648 store_allow_perm(zfs_deleg_who_type_t type, boolean_t local, boolean_t descend,
4649     const char *who, char *perms, nvlist_t *top_nvl)
4650 {
4651         int i;
4652         char ld[2] = { '\0', '\0' };
4653         char who_buf[ZFS_MAXNAMELEN+32];
4654         char base_type;
4655         char set_type;
4656         nvlist_t *base_nvl = NULL;
4657         nvlist_t *set_nvl = NULL;
4658         nvlist_t *nvl;

4660         if (nvlist_alloc(&base_nvl, NV_UNIQUE_NAME, 0) != 0)
4661                 nomem();
4662         if (nvlist_alloc(&set_nvl, NV_UNIQUE_NAME, 0) !=  0)
4663                 nomem();

4665         switch (type) {
4666         case ZFS_DELEG_NAMED_SET_SETS:
4667         case ZFS_DELEG_NAMED_SET:
4668                 set_type = ZFS_DELEG_NAMED_SET_SETS;
4669                 base_type = ZFS_DELEG_NAMED_SET;
4670                 ld[0] = ZFS_DELEG_NA;
4671                 break;
4672         case ZFS_DELEG_CREATE_SETS:
4673         case ZFS_DELEG_CREATE:
4674                 set_type = ZFS_DELEG_CREATE_SETS;
4675                 base_type = ZFS_DELEG_CREATE;
4676                 ld[0] = ZFS_DELEG_NA;
4677                 break;
4678         case ZFS_DELEG_USER_SETS:
```



```
4679         case ZFS_DELEG_USER:
4680                 set_type = ZFS_DELEG_USER_SETS;
4681                 base_type = ZFS_DELEG_USER;
4682                 if (local)
4683                         ld[0] = ZFS_DELEG_LOCAL;
4684                 if (descend)
4685                         ld[1] = ZFS_DELEG_DESCENDENT;
4686                 break;
4687         case ZFS_DELEG_GROUP_SETS:
4688         case ZFS_DELEG_GROUP:
4689                 set_type = ZFS_DELEG_GROUP_SETS;
4690                 base_type = ZFS_DELEG_GROUP;
4691                 if (local)
4692                         ld[0] = ZFS_DELEG_LOCAL;
4693                 if (descend)
4694                         ld[1] = ZFS_DELEG_DESCENDENT;
4695                 break;
4696         case ZFS_DELEG_EVERYONE_SETS:
4697         case ZFS_DELEG_EVERYONE:
4698                 set_type = ZFS_DELEG_EVERYONE_SETS;
4699                 base_type = ZFS_DELEG_EVERYONE;
4700                 if (local)
4701                         ld[0] = ZFS_DELEG_LOCAL;
4702                 if (descend)
4703                         ld[1] = ZFS_DELEG_DESCENDENT;
4704         }

4706         if (perms != NULL) {
4707                 char *curr = perms;
4708                 char *end = curr + strlen(perms);

4710                 while (curr < end) {
4711                         char *delim = strchr(curr, ',');
4712                         if (delim == NULL)
4713                                 delim = end;
4714                         else
4715                                 *delim = '\0';

4717                         if (curr[0] == '@')
4718                                 nvl = set_nvl;
4719                         else
4720                                 nvl = base_nvl;

4722                         (void) nvlist_add_boolean(nvl, curr);
4723                         if (delim != end)
4724                                 *delim = ',';
4725                         curr = delim + 1;
4726                 }

4728                 for (i = 0; i < 2; i++) {
4729                         char locality = ld[i];
4730                         if (locality == 0)
4731                                 continue;

4733                         if (!nvlist_empty(base_nvl)) {
4734                                 if (who != NULL)
4735                                         (void) snprintf(who_buf,
4736                                             sizeof (who_buf), "%c%c$%s",
4737                                             base_type, locality, who);
4738                                 else
4739                                         (void) snprintf(who_buf,
4740                                             sizeof (who_buf), "%c%c$",
4741                                             base_type, locality);

4743                                 (void) nvlist_add_nvlist(top_nvl, who_buf,
4744                                     base_nvl);
```

```
4745                        }


4748                        if (!nvlist_empty(set_nvl)) {
4749                                if (who != NULL)
4750                                        (void) snprintf(who_buf,
4751                                            sizeof (who_buf), "%c%c$%s",
4752                                            set_type, locality, who);
4753                                else
4754                                        (void) snprintf(who_buf,
4755                                            sizeof (who_buf), "%c%c$",
4756                                            set_type, locality);

4758                                (void) nvlist_add_nvlist(top_nvl, who_buf,
4759                                    set_nvl);
4760                        }
4761                }
4762        } else {
4763                for (i = 0; i < 2; i++) {
4764                        char locality = ld[i];
4765                        if (locality == 0)
4766                                continue;

4768                        if (who != NULL)
4769                                (void) snprintf(who_buf, sizeof (who_buf),
4770                                    "%c%c$%s", base_type, locality, who);
4771                        else
4772                                (void) snprintf(who_buf, sizeof (who_buf),
4773                                    "%c%c$", base_type, locality);
4774                        (void) nvlist_add_boolean(top_nvl, who_buf);

4776                        if (who != NULL)
4777                                (void) snprintf(who_buf, sizeof (who_buf),
4778                                    "%c%c$%s", set_type, locality, who);
4779                        else
4780                                (void) snprintf(who_buf, sizeof (who_buf),
4781                                    "%c%c$", set_type, locality);
4782                        (void) nvlist_add_boolean(top_nvl, who_buf);
4783                }
4784        }
4785 }

4787 static int
4788 construct_fsacl_list(boolean_t un, struct allow_opts *opts, nvlist_t **nvlp)
4789 {
4790        if (nvlist_alloc(nvlp, NV_UNIQUE_NAME, 0) != 0)
4791                nomem();

4793        if (opts->set) {
4794                store_allow_perm(ZFS_DELEG_NAMED_SET, opts->local,
4795                    opts->descend, opts->who, opts->perms, *nvlp);
4796        } else if (opts->create) {
4797                store_allow_perm(ZFS_DELEG_CREATE, opts->local,
4798                    opts->descend, NULL, opts->perms, *nvlp);
4799        } else if (opts->everyone) {
4800                store_allow_perm(ZFS_DELEG_EVERYONE, opts->local,
4801                    opts->descend, NULL, opts->perms, *nvlp);
4802        } else {
4803                char *curr = opts->who;
4804                char *end = curr + strlen(curr);

4806                while (curr < end) {
4807                        const char *who;
4808                        zfs_deleg_who_type_t who_type;
4809                        char *endch;
4810                        char *delim = strchr(curr, ',');
```



```
4811                        char errbuf[256];
4812                        char id[64];
4813                        struct passwd *p = NULL;
4814                        struct group *g = NULL;

4816                        uid_t rid;
4817                        if (delim == NULL)
4818                                delim = end;
4819                        else
4820                                *delim = '\0';

4822                        rid = (uid_t)strtol(curr, &endch, 0);
4823                        if (opts->user) {
4824                                who_type = ZFS_DELEG_USER;
4825                                if (*endch != '\0')
4826                                        p = getpwnam(curr);
4827                                else
4828                                        p = getpwuid(rid);

4830                                if (p != NULL)
4831                                        rid = p->pw_uid;
4832                                else {
4833                                        (void) snprintf(errbuf, 256, gettext(
4834                                            "invalid user %s"), curr);
4835                                        allow_usage(un, B_TRUE, errbuf);
4836                                }
4837                        } else if (opts->group) {
4838                                who_type = ZFS_DELEG_GROUP;
4839                                if (*endch != '\0')
4840                                        g = getgrnam(curr);
4841                                else
4842                                        g = getgrgid(rid);

4844                                if (g != NULL)
4845                                        rid = g->gr_gid;
4846                                else {
4847                                        (void) snprintf(errbuf, 256, gettext(
4848                                            "invalid group %s"),  curr);
4849                                        allow_usage(un, B_TRUE, errbuf);
4850                                }
4851                        } else {
4852                                if (*endch != '\0') {
4853                                        p = getpwnam(curr);
4854                                } else {
4855                                        p = getpwuid(rid);
4856                                }

4858                                if (p == NULL)
4859                                        if (*endch != '\0') {
4860                                                g = getgrnam(curr);
4861                                        } else {
4862                                                g = getgrgid(rid);
4863                                        }

4865                                if (p != NULL) {
4866                                        who_type = ZFS_DELEG_USER;
4867                                        rid = p->pw_uid;
4868                                } else if (g != NULL) {
4869                                        who_type = ZFS_DELEG_GROUP;
4870                                        rid = g->gr_gid;
4871                                } else {
4872                                        (void) snprintf(errbuf, 256, gettext(
4873                                            "invalid user/group %s"), curr);
4874                                        allow_usage(un, B_TRUE, errbuf);
4875                                }
4876                        }
```

```
4878                        (void) sprintf(id, "%u", rid);
4879                        who = id;

4881                        store_allow_perm(who_type, opts->local,
4882                            opts->descend, who, opts->perms, *nvlp);
4883                        curr = delim + 1;
4884                }
4885        }

4887        return (0);
4888 }

4890 static void
4891 print_set_creat_perms(uu_avl_t *who_avl)
4892 {
4893        const char *sc_title[] = {
4894                gettext("Permission sets:\n"),
4895                gettext("Create time permissions:\n"),
4896                NULL
4897        };
4898        const char **title_ptr = sc_title;
4899        who_perm_node_t *who_node = NULL;
4900        int prev_weight = -1;

4902        for (who_node = uu_avl_first(who_avl); who_node != NULL;
4903            who_node = uu_avl_next(who_avl, who_node)) {
4904                uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4905                zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4906                const char *who_name = who_node->who_perm.who_name;
4907                int weight = who_type2weight(who_type);
4908                boolean_t first = B_TRUE;
4909                deleg_perm_node_t *deleg_node;

4911                if (prev_weight != weight) {
4912                        (void) printf(*title_ptr++);
4913                        prev_weight = weight;
4914                }

4916                if (who_name == NULL || strnlen(who_name, 1) == 0)
4917                        (void) printf("\t");
4918                else
4919                        (void) printf("\t%s ", who_name);

4921                for (deleg_node = uu_avl_first(avl); deleg_node != NULL;
4922                    deleg_node = uu_avl_next(avl, deleg_node)) {
4923                        if (first) {
4924                                (void) printf("%s",
4925                                    deleg_node->dpn_perm.dp_name);
4926                                first = B_FALSE;
4927                        } else
4928                                (void) printf(",%s",
4929                                    deleg_node->dpn_perm.dp_name);
4930                }

4932                (void) printf("\n");
4933        }
4934 }

4936 static void inline
4937 print_uge_deleg_perms(uu_avl_t *who_avl, boolean_t local, boolean_t descend,
4938     const char *title)
4939 {
4940        who_perm_node_t *who_node = NULL;
4941        boolean_t prt_title = B_TRUE;
4942        uu_avl_walk_t *walk;
```



```
4944        if ((walk = uu_avl_walk_start(who_avl, UU_WALK_ROBUST)) == NULL)
4945                nomem();

4947        while ((who_node = uu_avl_walk_next(walk)) != NULL) {
4948                const char *who_name = who_node->who_perm.who_name;
4949                const char *nice_who_name = who_node->who_perm.who_ug_name;
4950                uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4951                zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4952                char delim = ' ';
4953                deleg_perm_node_t *deleg_node;
4954                boolean_t prt_who = B_TRUE;

4956                for (deleg_node = uu_avl_first(avl);
4957                    deleg_node != NULL;
4958                    deleg_node = uu_avl_next(avl, deleg_node)) {
4959                        if (local != deleg_node->dpn_perm.dp_local ||
4960                            descend != deleg_node->dpn_perm.dp_descend)
4961                                continue;

4963                        if (prt_who) {
4964                                const char *who = NULL;
4965                                if (prt_title) {
4966                                        prt_title = B_FALSE;
4967                                        (void) printf(title);
4968                                }

4970                                switch (who_type) {
4971                                case ZFS_DELEG_USER_SETS:
4972                                case ZFS_DELEG_USER:
4973                                        who = gettext("user");
4974                                        if (nice_who_name)
4975                                                who_name  = nice_who_name;
4976                                        break;
4977                                case ZFS_DELEG_GROUP_SETS:
4978                                case ZFS_DELEG_GROUP:
4979                                        who = gettext("group");
4980                                        if (nice_who_name)
4981                                                who_name  = nice_who_name;
4982                                        break;
4983                                case ZFS_DELEG_EVERYONE_SETS:
4984                                case ZFS_DELEG_EVERYONE:
4985                                        who = gettext("everyone");
4986                                        who_name = NULL;
4987                                }

4989                                prt_who = B_FALSE;
4990                                if (who_name == NULL)
4991                                        (void) printf("\t%s", who);
4992                                else
4993                                        (void) printf("\t%s %s", who, who_name);
4994                        }

4996                        (void) printf("%c%s", delim,
4997                            deleg_node->dpn_perm.dp_name);
4998                        delim = ',';
4999                }

5001                if (!prt_who)
5002                        (void) printf("\n");
5003        }

5005        uu_avl_walk_end(walk);
5006 }

5008 static void
```

```
5009 print_fs_perms(fs_perm_set_t *fspset)
5010 {
5011         fs_perm_node_t *node = NULL;
5012         char buf[ZFS_MAXNAMELEN+32];
5013         const char *dsname = buf;

5015         for (node = uu_list_first(fspset->fsps_list); node != NULL;
5016             node = uu_list_next(fspset->fsps_list, node)) {
5017                 uu_avl_t *sc_avl = node->fspn_fsperm.fsp_sc_avl;
5018                 uu_avl_t *uge_avl = node->fspn_fsperm.fsp_uge_avl;
5019                 int left = 0;

5021                 (void) snprintf(buf, ZFS_MAXNAMELEN+32,
5022                     gettext("---- Permissions on %s "),
5023                     node->fspn_fsperm.fsp_name);
5024                 (void) printf(dsname);
5025                 left = 70 - strlen(buf);
5026                 while (left-- > 0)
5027                         (void) printf("-");
5028                 (void) printf("\n");

5030                 print_set_creat_perms(sc_avl);
5031                 print_uge_deleg_perms(uge_avl, B_TRUE, B_FALSE,
5032                     gettext("Local permissions:\n"));
5033                 print_uge_deleg_perms(uge_avl, B_FALSE, B_TRUE,
5034                     gettext("Descendent permissions:\n"));
5035                 print_uge_deleg_perms(uge_avl, B_TRUE, B_TRUE,
5036                     gettext("Local+Descendent permissions:\n"));
5037         }
5038 }

5040 static fs_perm_set_t fs_perm_set = { NULL, NULL, NULL, NULL };

5042 struct deleg_perms {
5043         boolean_t un;
5044         nvlist_t *nvl;
5045 };

5047 static int
5048 set_deleg_perms(zfs_handle_t *zhp, void *data)
5049 {
5050         struct deleg_perms *perms = (struct deleg_perms *)data;
5051         zfs_type_t zfs_type = zfs_get_type(zhp);

5053         if (zfs_type != ZFS_TYPE_FILESYSTEM && zfs_type != ZFS_TYPE_VOLUME)
5054                 return (0);

5056         return (zfs_set_fsacl(zhp, perms->un, perms->nvl));
5057 }

5059 static int
5060 zfs_do_allow_unallow_impl(int argc, char **argv, boolean_t un)
5061 {
5062         zfs_handle_t *zhp;
5063         nvlist_t *perm_nvl = NULL;
5064         nvlist_t *update_perm_nvl = NULL;
5065         int error = 1;
5066         int c;
5067         struct allow_opts opts = { 0 };

5069         const char *optstr = un ? "ldugecsrh" : "ldugecsh";

5071         /* check opts */
5072         while ((c = getopt(argc, argv, optstr)) != -1) {
5073                 switch (c) {
5074                 case 'l':
```



```
5075                         opts.local = B_TRUE;
5076                         break;
5077                 case 'd':
5078                         opts.descend = B_TRUE;
5079                         break;
5080                 case 'u':
5081                         opts.user = B_TRUE;
5082                         break;
5083                 case 'g':
5084                         opts.group = B_TRUE;
5085                         break;
5086                 case 'e':
5087                         opts.everyone = B_TRUE;
5088                         break;
5089                 case 's':
5090                         opts.set = B_TRUE;
5091                         break;
5092                 case 'c':
5093                         opts.create = B_TRUE;
5094                         break;
5095                 case 'r':
5096                         opts.recursive = B_TRUE;
5097                         break;
5098                 case ':':
5099                         (void) fprintf(stderr, gettext("missing argument for "
5100                             "'%c' option\n"), optopt);
5101                         usage(B_FALSE);
5102                         break;
5103                 case 'h':
5104                         opts.prt_usage = B_TRUE;
5105                         break;
5106                 case '?':
5107                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5108                             optopt);
5109                         usage(B_FALSE);
5110                 }
5111         }

5113         argc -= optind;
5114         argv += optind;

5116         /* check arguments */
5117         parse_allow_args(argc, argv, un, &opts);

5119         /* try to open the dataset */
5120         if ((zhp = zfs_open(g_zfs, opts.dataset, ZFS_TYPE_FILESYSTEM |
5121             ZFS_TYPE_VOLUME)) == NULL) {
5122                 (void) fprintf(stderr, "Failed to open dataset: %s\n",
5123                     opts.dataset);
5124                 return (-1);
5125         }

5127         if (zfs_get_fsacl(zhp, &perm_nvl) != 0)
5128                 goto cleanup2;

5130         fs_perm_set_init(&fs_perm_set);
5131         if (parse_fs_perm_set(&fs_perm_set, perm_nvl) != 0) {
5132                 (void) fprintf(stderr, "Failed to parse fsacl permissions\n");
5133                 goto cleanup1;
5134         }

5136         if (opts.prt_perms)
5137                 print_fs_perms(&fs_perm_set);
5138         else {
5139                 (void) construct_fsacl_list(un, &opts, &update_perm_nvl);
5140                 if (zfs_set_fsacl(zhp, un, update_perm_nvl) != 0)
```

```
5141                        goto cleanup0;

5143                if (un && opts.recursive) {
5144                        struct deleg_perms data = { un, update_perm_nvl };
5145                        if (zfs_iter_filesystems(zhp, set_deleg_perms,
5146                            &data) != 0)
5147                                goto cleanup0;
5148                }
5149        }

5151        error = 0;

5153 cleanup0:
5154        nvlist_free(perm_nvl);
5155        if (update_perm_nvl != NULL)
5156                nvlist_free(update_perm_nvl);
5157 cleanup1:
5158        fs_perm_set_fini(&fs_perm_set);
5159 cleanup2:
5160        zfs_close(zhp);

5162        return (error);
5163 }

5165 /*
5166  * zfs allow [-r] [-t] <tag> <snap> ...
5167  *
5168  *      -r      Recursively hold
5169  *      -t      Temporary hold (hidden option)
5170  *
5171  * Apply a user-hold with the given tag to the list of snapshots.
5172  */
5173 static int
5174 zfs_do_allow(int argc, char **argv)
5175 {
5176        return (zfs_do_allow_unallow_impl(argc, argv, B_FALSE));
5177 }

5179 /*
5180  * zfs unallow [-r] [-t] <tag> <snap> ...
5181  *
5182  *      -r      Recursively hold
5183  *      -t      Temporary hold (hidden option)
5184  *
5185  * Apply a user-hold with the given tag to the list of snapshots.
5186  */
5187 static int
5188 zfs_do_unallow(int argc, char **argv)
5189 {
5190        return (zfs_do_allow_unallow_impl(argc, argv, B_TRUE));
5191 }

5193 static int
5194 zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
5195 {
5196        int errors = 0;
5197        int i;
5198        const char *tag;
5199        boolean_t recursive = B_FALSE;
5200        boolean_t temphold = B_FALSE;
5201        const char *opts = holding ? "rt" : "r";
5202        int c;

5204        /* check options */
5205        while ((c = getopt(argc, argv, opts)) != -1) {
5206                switch (c) {
```



```
5207                case 'r':
5208                        recursive = B_TRUE;
5209                        break;
5210                case 't':
5211                        temphold = B_TRUE;
5212                        break;
5213                case '?':
5214                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5215                            optopt);
5216                        usage(B_FALSE);
5217                }
5218        }

5220        argc -= optind;
5221        argv += optind;

5223        /* check number of arguments */
5224        if (argc < 2)
5225                usage(B_FALSE);

5227        tag = argv[0];
5228        --argc;
5229        ++argv;

5231        if (holding && tag[0] == '.') {
5232                /* tags starting with '.' are reserved for libzfs */
5233                (void) fprintf(stderr, gettext("tag may not start with '.'\n"));
5234                usage(B_FALSE);
5235        }

5237        for (i = 0; i < argc; ++i) {
5238                zfs_handle_t *zhp;
5239                char parent[ZFS_MAXNAMELEN];
5240                const char *delim;
5241                char *path = argv[i];

5243                delim = strchr(path, '@');
5244                if (delim == NULL) {
5245                        (void) fprintf(stderr,
5246                            gettext("'%s' is not a snapshot\n"), path);
5247                        ++errors;
5248                        continue;
5249                }
5250                (void) strncpy(parent, path, delim - path);
5251                parent[delim - path] = '\0';

5253                zhp = zfs_open(g_zfs, parent,
5254                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
5255                if (zhp == NULL) {
5256                        ++errors;
5257                        continue;
5258                }
5259                if (holding) {
5260                        if (zfs_hold(zhp, delim+1, tag, recursive,
5261                            temphold, B_FALSE, -1, 0, 0) != 0)
5262                                ++errors;
5263                } else {
5264                        if (zfs_release(zhp, delim+1, tag, recursive) != 0)
5265                                ++errors;
5266                }
5267                zfs_close(zhp);
5268        }

5270        return (errors != 0);
5271 }
```

```
5273 /*
5274  * zfs hold [-r] [-t] <tag> <snap> ...
5275  *
5276  *      -r      Recursively hold
5277  *      -t      Temporary hold (hidden option)
5278  *
5279  * Apply a user-hold with the given tag to the list of snapshots.
5280  */
5281 static int
5282 zfs_do_hold(int argc, char **argv)
5283 {
5284         return (zfs_do_hold_rele_impl(argc, argv, B_TRUE));
5285 }

5287 /*
5288  * zfs release [-r] <tag> <snap> ...
5289  *
5290  *      -r      Recursively release
5291  *
5292  * Release a user-hold with the given tag from the list of snapshots.
5293  */
5294 static int
5295 zfs_do_release(int argc, char **argv)
5296 {
5297         return (zfs_do_hold_rele_impl(argc, argv, B_FALSE));
5298 }

5300 typedef struct holds_cbdata {
5301         boolean_t       cb_recursive;
5302         const char      *cb_snapname;
5303         nvlist_t        **cb_nvlp;
5304         size_t          cb_max_namelen;
5305         size_t          cb_max_taglen;
5306 } holds_cbdata_t;

5308 #define STRFTIME_FMT_STR "%a %b %e %k:%M %Y"
5309 #define DATETIME_BUF_LEN (32)
5310 /*
5311  *
5312  */
5313 static void
5314 print_holds(boolean_t scripted, size_t nwidth, size_t tagwidth, nvlist_t *nvl)
5315 {
5316         int i;
5317         nvpair_t *nvp = NULL;
5318         char *hdr_cols[] = { "NAME", "TAG", "TIMESTAMP" };
5319         const char *col;

5321         if (!scripted) {
5322                 for (i = 0; i < 3; i++) {
5323                         col = gettext(hdr_cols[i]);
5324                         if (i < 2)
5325                                 (void) printf("%-*s  ", i ? tagwidth : nwidth,
5326                                     col);
5327                         else
5328                                 (void) printf("%s\n", col);
5329                 }
5330         }

5332         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
5333                 char *zname = nvpair_name(nvp);
5334                 nvlist_t *nvl2;
5335                 nvpair_t *nvp2 = NULL;
5336                 (void) nvpair_value_nvlist(nvp, &nvl2);
5337                 while ((nvp2 = nvlist_next_nvpair(nvl2, nvp2)) != NULL) {
5338                         char tsbuf[DATETIME_BUF_LEN];
```



```
5339                         char *tagname = nvpair_name(nvp2);
5340                         uint64_t val = 0;
5341                         time_t time;
5342                         struct tm t;
5343                         char sep = scripted ? '\t' : ' ';
5344                         size_t sepnum = scripted ? 1 : 2;

5346                         (void) nvpair_value_uint64(nvp2, &val);
5347                         time = (time_t)val;
5348                         (void) localtime_r(&time, &t);
5349                         (void) strftime(tsbuf, DATETIME_BUF_LEN,
5350                             gettext(STRFTIME_FMT_STR), &t);

5352                         (void) printf("%-*s%*c%-*s%*c%s\n", nwidth, zname,
5353                             sepnum, sep, tagwidth, tagname, sepnum, sep, tsbuf);
5354                 }
5355         }
5356 }

5358 /*
5359  * Generic callback function to list a dataset or snapshot.
5360  */
5361 static int
5362 holds_callback(zfs_handle_t *zhp, void *data)
5363 {
5364         holds_cbdata_t *cbp = data;
5365         nvlist_t *top_nvl = *cbp->cb_nvlp;
5366         nvlist_t *nvl = NULL;
5367         nvpair_t *nvp = NULL;
5368         const char *zname = zfs_get_name(zhp);
5369         size_t znamelen = strnlen(zname, ZFS_MAXNAMELEN);

5371         if (cbp->cb_recursive) {
5372                 const char *snapname;
5373                 char *delim  = strchr(zname, '@');
5374                 if (delim == NULL)
5375                         return (0);

5377                 snapname = delim + 1;
5378                 if (strcmp(cbp->cb_snapname, snapname))
5379                         return (0);
5380         }

5382         if (zfs_get_holds(zhp, &nvl) != 0)
5383                 return (-1);

5385         if (znamelen > cbp->cb_max_namelen)
5386                 cbp->cb_max_namelen  = znamelen;

5388         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
5389                 const char *tag = nvpair_name(nvp);
5390                 size_t taglen = strnlen(tag, MAXNAMELEN);
5391                 if (taglen > cbp->cb_max_taglen)
5392                         cbp->cb_max_taglen  = taglen;
5393         }

5395         return (nvlist_add_nvlist(top_nvl, zname, nvl));
5396 }

5398 /*
5399  * zfs holds [-r] <snap> ...
5400  *
5401  *      -r      Recursively hold
5402  */
5403 static int
5404 zfs_do_holds(int argc, char **argv)
```

```
5405 {
5406         int errors = 0;
5407         int c;
5408         int i;
5409         boolean_t scripted = B_FALSE;
5410         boolean_t recursive = B_FALSE;
5411         const char *opts = "rH";
5412         nvlist_t *nvl;

5414         int types = ZFS_TYPE_SNAPSHOT;
5415         holds_cbdata_t cb = { 0 };

5417         int limit = 0;
5418         int ret = 0;
5419         int flags = 0;

5421         /* check options */
5422         while ((c = getopt(argc, argv, opts)) != -1) {
5423                 switch (c) {
5424                 case 'r':
5425                         recursive = B_TRUE;
5426                         break;
5427                 case 'H':
5428                         scripted = B_TRUE;
5429                         break;
5430                 case '?':
5431                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5432                             optopt);
5433                         usage(B_FALSE);
5434                 }
5435         }

5437         if (recursive) {
5438                 types |= ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME;
5439                 flags |= ZFS_ITER_RECURSE;
5440         }

5442         argc -= optind;
5443         argv += optind;

5445         /* check number of arguments */
5446         if (argc < 1)
5447                 usage(B_FALSE);

5449         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
5450                 nomem();

5452         for (i = 0; i < argc; ++i) {
5453                 char *snapshot = argv[i];
5454                 const char *delim;
5455                 const char *snapname;

5457                 delim = strchr(snapshot, '@');
5458                 if (delim == NULL) {
5459                         (void) fprintf(stderr,
5460                             gettext("'%s' is not a snapshot\n"), snapshot);
5461                         ++errors;
5462                         continue;
5463                 }
5464                 snapname = delim + 1;
5465                 if (recursive)
5466                         snapshot[delim - snapshot] = '\0';

5468                 cb.cb_recursive = recursive;
5469                 cb.cb_snapname = snapname;
5470                 cb.cb_nvlp = &nvl;
```



```
5472                 /*
5473                  *  1. collect holds data, set format options
5474                  */
5475                 ret = zfs_for_each(argc, argv, flags, types, NULL, NULL, limit,
5476                     holds_callback, &cb);
5477                 if (ret != 0)
5478                         ++errors;
5479         }

5481         /*
5482          *  2. print holds data
5483          */
5484         print_holds(scripted, cb.cb_max_namelen, cb.cb_max_taglen, nvl);

5486         if (nvlist_empty(nvl))
5487                 (void) printf(gettext("no datasets available\n"));

5489         nvlist_free(nvl);

5491         return (0 != errors);
5492 }

5494 #define CHECK_SPINNER 30
5495 #define SPINNER_TIME 3          /* seconds */
5496 #define MOUNT_TIME 5            /* seconds */

5498 static int
5499 get_one_dataset(zfs_handle_t *zhp, void *data)
5500 {
5501         static char *spin[] = { "-", "\\", "|", "/" };
5502         static int spinval = 0;
5503         static int spincheck = 0;
5504         static time_t last_spin_time = (time_t)0;
5505         get_all_cb_t *cbp = data;
5506         zfs_type_t type = zfs_get_type(zhp);

5508         if (cbp->cb_verbose) {
5509                 if (--spincheck < 0) {
5510                         time_t now = time(NULL);
5511                         if (last_spin_time + SPINNER_TIME < now) {
5512                                 update_progress(spin[spinval++ % 4]);
5513                                 last_spin_time = now;
5514                         }
5515                         spincheck = CHECK_SPINNER;
5516                 }
5517         }

5519         /*
5520          * Interate over any nested datasets.
5521          */
5522         if (zfs_iter_filesystems(zhp, get_one_dataset, data) != 0) {
5523                 zfs_close(zhp);
5524                 return (1);
5525         }

5527         /*
5528          * Skip any datasets whose type does not match.
5529          */
5530         if ((type & ZFS_TYPE_FILESYSTEM) == 0) {
5531                 zfs_close(zhp);
5532                 return (0);
5533         }
5534         libzfs_add_handle(cbp, zhp);
5535         assert(cbp->cb_used <= cbp->cb_alloc);
```

```
5537         return (0);
5538 }

5540 static void
5541 get_all_datasets(zfs_handle_t ***dslist, size_t *count, boolean_t verbose)
5542 {
5543         get_all_cb_t cb = { 0 };
5544         cb.cb_verbose = verbose;
5545         cb.cb_getone = get_one_dataset;

5547         if (verbose)
5548                 set_progress_header(gettext("Reading ZFS config"));
5549         (void) zfs_iter_root(g_zfs, get_one_dataset, &cb);

5551         *dslist = cb.cb_handles;
5552         *count = cb.cb_used;

5554         if (verbose)
5555                 finish_progress(gettext("done."));
5556 }

5558 /*
5559  * Generic callback for sharing or mounting filesystems.  Because the code is so
5560  * similar, we have a common function with an extra parameter to determine which
5561  * mode we are using.
5562  */
5563 #define OP_SHARE        0x1
5564 #define OP_MOUNT        0x2

5566 /*
5567  * Share or mount a dataset.
5568  */
5569 static int
5570 share_mount_one(zfs_handle_t *zhp, int op, int flags, char *protocol,
5571     boolean_t explicit, const char *options)
5572 {
5573         char mountpoint[ZFS_MAXPROPLEN];
5574         char shareopts[ZFS_MAXPROPLEN];
5575         char smbshareopts[ZFS_MAXPROPLEN];
5576         const char *cmdname = op == OP_SHARE ? "share" : "mount";
5577         struct mnttab mnt;
5578         uint64_t zoned, canmount;
5579         boolean_t shared_nfs, shared_smb;

5581         assert(zfs_get_type(zhp) & ZFS_TYPE_FILESYSTEM);

5583         /*
5584          * Check to make sure we can mount/share this dataset.  If we
5585          * are in the global zone and the filesystem is exported to a
5586          * local zone, or if we are in a local zone and the
5587          * filesystem is not exported, then it is an error.
5588          */
5589         zoned = zfs_prop_get_int(zhp, ZFS_PROP_ZONED);

5591         if (zoned && getzoneid() == GLOBAL_ZONEID) {
5592                 if (!explicit)
5593                         return (0);

5595                 (void) fprintf(stderr, gettext("cannot %s '%s': "
5596                     "dataset is exported to a local zone\n"), cmdname,
5597                     zfs_get_name(zhp));
5598                 return (1);

5600         } else if (!zoned && getzoneid() != GLOBAL_ZONEID) {
5601                 if (!explicit)
5602                         return (0);
```



```
5604                 (void) fprintf(stderr, gettext("cannot %s '%s': "
5605                     "permission denied\n"), cmdname,
5606                     zfs_get_name(zhp));
5607                 return (1);
5608         }

5610         /*
5611          * Ignore any filesystems which don't apply to us. This
5612          * includes those with a legacy mountpoint, or those with
5613          * legacy share options.
5614          */
5615         verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
5616             sizeof (mountpoint), NULL, NULL, 0, B_FALSE) == 0);
5617         verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS, shareopts,
5618             sizeof (shareopts), NULL, NULL, 0, B_FALSE) == 0);
5619         verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB, smbshareopts,
5620             sizeof (smbshareopts), NULL, NULL, 0, B_FALSE) == 0);

5622         if (op == OP_SHARE && strcmp(shareopts, "off") == 0 &&
5623             strcmp(smbshareopts, "off") == 0) {
5624                 if (!explicit)
5625                         return (0);

5627                 (void) fprintf(stderr, gettext("cannot share '%s': "
5628                     "legacy share\n"), zfs_get_name(zhp));
5629                 (void) fprintf(stderr, gettext("use share(1M) to "
5630                     "share this filesystem, or set "
5631                     "sharenfs property on\n"));
5632                 return (1);
5633         }

5635         /*
5636          * We cannot share or mount legacy filesystems. If the
5637          * shareopts is non-legacy but the mountpoint is legacy, we
5638          * treat it as a legacy share.
5639          */
5640         if (strcmp(mountpoint, "legacy") == 0) {
5641                 if (!explicit)
5642                         return (0);

5644                 (void) fprintf(stderr, gettext("cannot %s '%s': "
5645                     "legacy mountpoint\n"), cmdname, zfs_get_name(zhp));
5646                 (void) fprintf(stderr, gettext("use %s(1M) to "
5647                     "%s this filesystem\n"), cmdname, cmdname);
5648                 return (1);
5649         }

5651         if (strcmp(mountpoint, "none") == 0) {
5652                 if (!explicit)
5653                         return (0);

5655                 (void) fprintf(stderr, gettext("cannot %s '%s': no "
5656                     "mountpoint set\n"), cmdname, zfs_get_name(zhp));
5657                 return (1);
5658         }

5660         /*
5661          * canmount     explicit        outcome
5662          * on           no              pass through
5663          * on           yes             pass through
5664          * off          no              return 0
5665          * off          yes             display error, return 1
5666          * noauto       no              return 0
5667          * noauto       yes             pass through
5668          */
```

```
5669         canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);
5670         if (canmount == ZFS_CANMOUNT_OFF) {
5671                 if (!explicit)
5672                         return (0);

5674                 (void) fprintf(stderr, gettext("cannot %s '%s': "
5675                     "'canmount' property is set to 'off'\n"), cmdname,
5676                     zfs_get_name(zhp));
5677                 return (1);
5678         } else if (canmount == ZFS_CANMOUNT_NOAUTO && !explicit) {
5679                 return (0);
5680         }

5682         /*
5683          * At this point, we have verified that the mountpoint and/or
5684          * shareopts are appropriate for auto management. If the
5685          * filesystem is already mounted or shared, return (failing
5686          * for explicit requests); otherwise mount or share the
5687          * filesystem.
5688          */
5689         switch (op) {
5690         case OP_SHARE:

5692                 shared_nfs = zfs_is_shared_nfs(zhp, NULL);
5693                 shared_smb = zfs_is_shared_smb(zhp, NULL);

5695                 if (shared_nfs && shared_smb ||
5696                     (shared_nfs && strcmp(shareopts, "on") == 0 &&
5697                     strcmp(smbshareopts, "off") == 0) ||
5698                     (shared_smb && strcmp(smbshareopts, "on") == 0 &&
5699                     strcmp(shareopts, "off") == 0)) {
5700                         if (!explicit)
5701                                 return (0);

5703                         (void) fprintf(stderr, gettext("cannot share "
5704                             "'%s': filesystem already shared\n"),
5705                             zfs_get_name(zhp));
5706                         return (1);
5707                 }

5709                 if (!zfs_is_mounted(zhp, NULL) &&
5710                     zfs_mount(zhp, NULL, 0) != 0)
5711                         return (1);

5713                 if (protocol == NULL) {
5714                         if (zfs_shareall(zhp) != 0)
5715                                 return (1);
5716                 } else if (strcmp(protocol, "nfs") == 0) {
5717                         if (zfs_share_nfs(zhp))
5718                                 return (1);
5719                 } else if (strcmp(protocol, "smb") == 0) {
5720                         if (zfs_share_smb(zhp))
5721                                 return (1);
5722                 } else {
5723                         (void) fprintf(stderr, gettext("cannot share "
5724                             "'%s': invalid share type '%s' "
5725                             "specified\n"),
5726                             zfs_get_name(zhp), protocol);
5727                         return (1);
5728                 }

5730                 break;

5732         case OP_MOUNT:
5733                 if (options == NULL)
5734                         mnt.mnt_mntopts = "";
```



```
5735                 else
5736                         mnt.mnt_mntopts = (char *)options;

5738                 if (!hasmntopt(&mnt, MNTOPT_REMOUNT) &&
5739                     zfs_is_mounted(zhp, NULL)) {
5740                         if (!explicit)
5741                                 return (0);

5743                         (void) fprintf(stderr, gettext("cannot mount "
5744                             "'%s': filesystem already mounted\n"),
5745                             zfs_get_name(zhp));
5746                         return (1);
5747                 }

5749                 if (zfs_mount(zhp, options, flags) != 0)
5750                         return (1);
5751                 break;
5752         }

5754         return (0);
5755 }

5757 /*
5758  * Reports progress in the form "(current/total)".  Not thread-safe.
5759  */
5760 static void
5761 report_mount_progress(int current, int total)
5762 {
5763         static time_t last_progress_time = 0;
5764         time_t now = time(NULL);
5765         char info[32];

5767         /* report 1..n instead of 0..n-1 */
5768         ++current;

5770         /* display header if we're here for the first time */
5771         if (current == 1) {
5772                 set_progress_header(gettext("Mounting ZFS filesystems"));
5773         } else if (current != total && last_progress_time + MOUNT_TIME >= now) {
5774                 /* too soon to report again */
5775                 return;
5776         }

5778         last_progress_time = now;

5780         (void) sprintf(info, "(%d/%d)", current, total);

5782         if (current == total)
5783                 finish_progress(info);
5784         else
5785                 update_progress(info);
5786 }

5788 static void
5789 append_options(char *mntopts, char *newopts)
5790 {
5791         int len = strlen(mntopts);

5793         /* original length plus new string to append plus 1 for the comma */
5794         if (len + 1 + strlen(newopts) >= MNT_LINE_MAX) {
5795                 (void) fprintf(stderr, gettext("the opts argument for "
5796                     "'%c' option is too long (more than %d chars)\n"),
5797                     "-o", MNT_LINE_MAX);
5798                 usage(B_FALSE);
5799         }
```

```
5801         if (*mntopts)
5802                 mntopts[len++] = ',';
5803
5804         (void) strcpy(&mntopts[len], newopts);
5805 }
5806
5807 static int
5808 share_mount(int op, int argc, char **argv)
5809 {
5810         int do_all = 0;
5811         boolean_t verbose = B_FALSE;
5812         int c, ret = 0;
5813         char *options = NULL;
5814         int flags = 0;
5815
5816         /* check options */
5817         while ((c = getopt(argc, argv, op == OP_MOUNT ? ":avo:O" : "a"))
5818             != -1) {
5819                 switch (c) {
5820                 case 'a':
5821                         do_all = 1;
5822                         break;
5823                 case 'v':
5824                         verbose = B_TRUE;
5825                         break;
5826                 case 'o':
5827                         if (*optarg == '\0') {
5828                                 (void) fprintf(stderr, gettext("empty mount "
5829                                     "options (-o) specified\n"));
5830                                 usage(B_FALSE);
5831                         }
5832
5833                         if (options == NULL)
5834                                 options = safe_malloc(MNT_LINE_MAX + 1);
5835
5836                         /* option validation is done later */
5837                         append_options(options, optarg);
5838                         break;
5839
5840                 case 'O':
5841                         flags |= MS_OVERLAY;
5842                         break;
5843                 case ':':
5844                         (void) fprintf(stderr, gettext("missing argument for "
5845                             "'%c' option\n"), optopt);
5846                         usage(B_FALSE);
5847                         break;
5848                 case '?':
5849                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5850                             optopt);
5851                         usage(B_FALSE);
5852                 }
5853         }
5854
5855         argc -= optind;
5856         argv += optind;
5857
5858         /* check number of arguments */
5859         if (do_all) {
5860                 zfs_handle_t **dslist = NULL;
5861                 size_t i, count = 0;
5862                 char *protocol = NULL;
5863
5864                 if (op == OP_SHARE && argc > 0) {
5865                         if (strcmp(argv[0], "nfs") != 0 &&
5866                             strcmp(argv[0], "smb") != 0) {
```



```
5867                                 (void) fprintf(stderr, gettext("share type "
5868                                     "must be 'nfs' or 'smb'\n"));
5869                                 usage(B_FALSE);
5870                         }
5871                         protocol = argv[0];
5872                         argc--;
5873                         argv++;
5874                 }
5875
5876                 if (argc != 0) {
5877                         (void) fprintf(stderr, gettext("too many arguments\n"));
5878                         usage(B_FALSE);
5879                 }
5880
5881                 start_progress_timer();
5882                 get_all_datasets(&dslist, &count, verbose);
5883
5884                 if (count == 0)
5885                         return (0);
5886
5887                 qsort(dslist, count, sizeof (void *), libzfs_dataset_cmp);
5888
5889                 for (i = 0; i < count; i++) {
5890                         if (verbose)
5891                                 report_mount_progress(i, count);
5892
5893                         if (share_mount_one(dslist[i], op, flags, protocol,
5894                             B_FALSE, options) != 0)
5895                                 ret = 1;
5896                         zfs_close(dslist[i]);
5897                 }
5898
5899                 free(dslist);
5900         } else if (argc == 0) {
5901                 struct mnttab entry;
5902
5903                 if ((op == OP_SHARE) || (options != NULL)) {
5904                         (void) fprintf(stderr, gettext("missing filesystem "
5905                             "argument (specify -a for all)\n"));
5906                         usage(B_FALSE);
5907                 }
5908
5909                 /*
5910                  * When mount is given no arguments, go through /etc/mnttab and
5911                  * display any active ZFS mounts.  We hide any snapshots, since
5912                  * they are controlled automatically.
5913                  */
5914                 rewind(mnttab_file);
5915                 while (getmntent(mnttab_file, &entry) == 0) {
5916                         if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0 ||
5917                             strchr(entry.mnt_special, '@') != NULL)
5918                                 continue;
5919
5920                         (void) printf("%-30s  %s\n", entry.mnt_special,
5921                             entry.mnt_mountp);
5922                 }
5923
5924         } else {
5925                 zfs_handle_t *zhp;
5926
5927                 if (argc > 1) {
5928                         (void) fprintf(stderr,
5929                             gettext("too many arguments\n"));
5930                         usage(B_FALSE);
5931                 }
```

```
5933                 if ((zhp = zfs_open(g_zfs, argv[0],
5934                     ZFS_TYPE_FILESYSTEM)) == NULL) {
5935                         ret = 1;
5936                 } else {
5937                         ret = share_mount_one(zhp, op, flags, NULL, B_TRUE,
5938                             options);
5939                         zfs_close(zhp);
5940                 }
5941         }

5943         return (ret);
5944 }

5946 /*
5947  * zfs mount -a [nfs]
5948  * zfs mount filesystem
5949  *
5950  * Mount all filesystems, or mount the given filesystem.
5951  */
5952 static int
5953 zfs_do_mount(int argc, char **argv)
5954 {
5955         return (share_mount(OP_MOUNT, argc, argv));
5956 }

5958 /*
5959  * zfs share -a [nfs | smb]
5960  * zfs share filesystem
5961  *
5962  * Share all filesystems, or share the given filesystem.
5963  */
5964 static int
5965 zfs_do_share(int argc, char **argv)
5966 {
5967         return (share_mount(OP_SHARE, argc, argv));
5968 }

5970 typedef struct unshare_unmount_node {
5971         zfs_handle_t    *un_zhp;
5972         char            *un_mountp;
5973         uu_avl_node_t   un_avlnode;
5974 } unshare_unmount_node_t;

5976 /* ARGSUSED */
5977 static int
5978 unshare_unmount_compare(const void *larg, const void *rarg, void *unused)
5979 {
5980         const unshare_unmount_node_t *l = larg;
5981         const unshare_unmount_node_t *r = rarg;

5983         return (strcmp(l->un_mountp, r->un_mountp));
5984 }

5986 /*
5987  * Convenience routine used by zfs_do_umount() and manual_unmount().  Given an
5988  * absolute path, find the entry /etc/mnttab, verify that its a ZFS filesystem,
5989  * and unmount it appropriately.
5990  */
5991 static int
5992 unshare_unmount_path(int op, char *path, int flags, boolean_t is_manual)
5993 {
5994         zfs_handle_t *zhp;
5995         int ret = 0;
5996         struct stat64 statbuf;
5997         struct extmnttab entry;
5998         const char *cmdname = (op == OP_SHARE) ? "unshare" : "unmount";
```



```
5999         ino_t path_inode;

6001         /*
6002          * Search for the path in /etc/mnttab.  Rather than looking for the
6003          * specific path, which can be fooled by non-standard paths (i.e. ".."
6004          * or "//"), we stat() the path and search for the corresponding
6005          * (major,minor) device pair.
6006          */
6007         if (stat64(path, &statbuf) != 0) {
6008                 (void) fprintf(stderr, gettext("cannot %s '%s': %s\n"),
6009                     cmdname, path, strerror(errno));
6010                 return (1);
6011         }
6012         path_inode = statbuf.st_ino;

6014         /*
6015          * Search for the given (major,minor) pair in the mount table.
6016          */
6017         rewind(mnttab_file);
6018         while ((ret = getextmntent(mnttab_file, &entry, 0)) == 0) {
6019                 if (entry.mnt_major == major(statbuf.st_dev) &&
6020                     entry.mnt_minor == minor(statbuf.st_dev))
6021                         break;
6022         }
6023         if (ret != 0) {
6024                 if (op == OP_SHARE) {
6025                         (void) fprintf(stderr, gettext("cannot %s '%s': not "
6026                             "currently mounted\n"), cmdname, path);
6027                         return (1);
6028                 }
6029                 (void) fprintf(stderr, gettext("warning: %s not in mnttab\n"),
6030                     path);
6031                 if ((ret = umount2(path, flags)) != 0)
6032                         (void) fprintf(stderr, gettext("%s: %s\n"), path,
6033                             strerror(errno));
6034                 return (ret != 0);
6035         }

6037         if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0) {
6038                 (void) fprintf(stderr, gettext("cannot %s '%s': not a ZFS "
6039                     "filesystem\n"), cmdname, path);
6040                 return (1);
6041         }

6043         if ((zhp = zfs_open(g_zfs, entry.mnt_special,
6044             ZFS_TYPE_FILESYSTEM)) == NULL)
6045                 return (1);

6047         ret = 1;
6048         if (stat64(entry.mnt_mountp, &statbuf) != 0) {
6049                 (void) fprintf(stderr, gettext("cannot %s '%s': %s\n"),
6050                     cmdname, path, strerror(errno));
6051                 goto out;
6052         } else if (statbuf.st_ino != path_inode) {
6053                 (void) fprintf(stderr, gettext("cannot "
6054                     "%s '%s': not a mountpoint\n"), cmdname, path);
6055                 goto out;
6056         }

6058         if (op == OP_SHARE) {
6059                 char nfs_mnt_prop[ZFS_MAXPROPLEN];
6060                 char smbshare_prop[ZFS_MAXPROPLEN];

6062                 verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS, nfs_mnt_prop,
6063                     sizeof (nfs_mnt_prop), NULL, NULL, 0, B_FALSE) == 0);
6064                 verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB, smbshare_prop,
```

```
6065                    sizeof (smbshare_prop), NULL, NULL, 0, B_FALSE) == 0);

6067                if (strcmp(nfs_mnt_prop, "off") == 0 &&
6068                    strcmp(smbshare_prop, "off") == 0) {
6069                        (void) fprintf(stderr, gettext("cannot unshare "
6070                            "'%s': legacy share\n"), path);
6071                        (void) fprintf(stderr, gettext("use "
6072                            "unshare(1M) to unshare this filesystem\n"));
6073                } else if (!zfs_is_shared(zhp)) {
6074                        (void) fprintf(stderr, gettext("cannot unshare '%s': "
6075                            "not currently shared\n"), path);
6076                } else {
6077                        ret = zfs_unshareall_bypath(zhp, path);
6078                }
6079        } else {
6080                char mtpt_prop[ZFS_MAXPROPLEN];

6082                verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mtpt_prop,
6083                    sizeof (mtpt_prop), NULL, NULL, 0, B_FALSE) == 0);

6085                if (is_manual) {
6086                        ret = zfs_unmount(zhp, NULL, flags);
6087                } else if (strcmp(mtpt_prop, "legacy") == 0) {
6088                        (void) fprintf(stderr, gettext("cannot unmount "
6089                            "'%s': legacy mountpoint\n"),
6090                            zfs_get_name(zhp));
6091                        (void) fprintf(stderr, gettext("use umount(1M) "
6092                            "to unmount this filesystem\n"));
6093                } else {
6094                        ret = zfs_unmountall(zhp, flags);
6095                }
6096        }

6098 out:
6099        zfs_close(zhp);

6101        return (ret != 0);
6102 }

6104 /*
6105  * Generic callback for unsharing or unmounting a filesystem.
6106  */
6107 static int
6108 unshare_unmount(int op, int argc, char **argv)
6109 {
6110        int do_all = 0;
6111        int flags = 0;
6112        int ret = 0;
6113        int c;
6114        zfs_handle_t *zhp;
6115        char nfs_mnt_prop[ZFS_MAXPROPLEN];
6116        char sharesmb[ZFS_MAXPROPLEN];

6118        /* check options */
6119        while ((c = getopt(argc, argv, op == OP_SHARE ? "a" : "af")) != -1) {
6120                switch (c) {
6121                case 'a':
6122                        do_all = 1;
6123                        break;
6124                case 'f':
6125                        flags = MS_FORCE;
6126                        break;
6127                case '?':
6128                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6129                            optopt);
6130                        usage(B_FALSE);
```



```
6131                }
6132        }

6134        argc -= optind;
6135        argv += optind;

6137        if (do_all) {
6138                /*
6139                 * We could make use of zfs_for_each() to walk all datasets in
6140                 * the system, but this would be very inefficient, especially
6141                 * since we would have to linearly search /etc/mnttab for each
6142                 * one.  Instead, do one pass through /etc/mnttab looking for
6143                 * zfs entries and call zfs_unmount() for each one.
6144                 *
6145                 * Things get a little tricky if the administrator has created
6146                 * mountpoints beneath other ZFS filesystems.  In this case, we
6147                 * have to unmount the deepest filesystems first.  To accomplish
6148                 * this, we place all the mountpoints in an AVL tree sorted by
6149                 * the special type (dataset name), and walk the result in
6150                 * reverse to make sure to get any snapshots first.
6151                 */
6152                struct mnttab entry;
6153                uu_avl_pool_t *pool;
6154                uu_avl_t *tree;
6155                unshare_unmount_node_t *node;
6156                uu_avl_index_t idx;
6157                uu_avl_walk_t *walk;

6159                if (argc != 0) {
6160                        (void) fprintf(stderr, gettext("too many arguments\n"));
6161                        usage(B_FALSE);
6162                }

6164                if (((pool = uu_avl_pool_create("unmount_pool",
6165                    sizeof (unshare_unmount_node_t),
6166                    offsetof(unshare_unmount_node_t, un_avlnode),
6167                    unshare_unmount_compare, UU_DEFAULT)) == NULL) ||
6168                    ((tree = uu_avl_create(pool, NULL, UU_DEFAULT)) == NULL))
6169                        nomem();

6171                rewind(mnttab_file);
6172                while (getmntent(mnttab_file, &entry) == 0) {

6174                        /* ignore non-ZFS entries */
6175                        if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
6176                                continue;

6178                        /* ignore snapshots */
6179                        if (strchr(entry.mnt_special, '@') != NULL)
6180                                continue;

6182                        if ((zhp = zfs_open(g_zfs, entry.mnt_special,
6183                            ZFS_TYPE_FILESYSTEM)) == NULL) {
6184                                ret = 1;
6185                                continue;
6186                        }

6188                        switch (op) {
6189                        case OP_SHARE:
6190                                verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS,
6191                                    nfs_mnt_prop,
6192                                    sizeof (nfs_mnt_prop),
6193                                    NULL, NULL, 0, B_FALSE) == 0);
6194                                if (strcmp(nfs_mnt_prop, "off") != 0)
6195                                        break;
6196                                verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB,
```

```
6197                                    nfs_mnt_prop,
6198                                    sizeof (nfs_mnt_prop),
6199                                    NULL, NULL, 0, B_FALSE) == 0);
6200                                if (strcmp(nfs_mnt_prop, "off") == 0)
6201                                        continue;
6202                                break;
6203                        case OP_MOUNT:
6204                                /* Ignore legacy mounts */
6205                                verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT,
6206                                    nfs_mnt_prop,
6207                                    sizeof (nfs_mnt_prop),
6208                                    NULL, NULL, 0, B_FALSE) == 0);
6209                                if (strcmp(nfs_mnt_prop, "legacy") == 0)
6210                                        continue;
6211                                /* Ignore canmount=noauto mounts */
6212                                if (zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT) ==
6213                                    ZFS_CANMOUNT_NOAUTO)
6214                                        continue;
6215                        default:
6216                                break;
6217                        }

6219                        node = safe_malloc(sizeof (unshare_unmount_node_t));
6220                        node->un_zhp = zhp;
6221                        node->un_mountp = safe_strdup(entry.mnt_mountp);

6223                        uu_avl_node_init(node, &node->un_avlnode, pool);

6225                        if (uu_avl_find(tree, node, NULL, &idx) == NULL) {
6226                                uu_avl_insert(tree, node, idx);
6227                        } else {
6228                                zfs_close(node->un_zhp);
6229                                free(node->un_mountp);
6230                                free(node);
6231                        }
6232                }

6234                /*
6235                 * Walk the AVL tree in reverse, unmounting each filesystem and
6236                 * removing it from the AVL tree in the process.
6237                 */
6238                if ((walk = uu_avl_walk_start(tree,
6239                    UU_WALK_REVERSE | UU_WALK_ROBUST)) == NULL)
6240                        nomem();

6242                while ((node = uu_avl_walk_next(walk)) != NULL) {
6243                        uu_avl_remove(tree, node);

6245                        switch (op) {
6246                        case OP_SHARE:
6247                                if (zfs_unshareall_bypath(node->un_zhp,
6248                                    node->un_mountp) != 0)
6249                                        ret = 1;
6250                                break;

6252                        case OP_MOUNT:
6253                                if (zfs_unmount(node->un_zhp,
6254                                    node->un_mountp, flags) != 0)
6255                                        ret = 1;
6256                                break;
6257                        }

6259                        zfs_close(node->un_zhp);
6260                        free(node->un_mountp);
6261                        free(node);
6262                }
```



```
6264                uu_avl_walk_end(walk);
6265                uu_avl_destroy(tree);
6266                uu_avl_pool_destroy(pool);

6268        } else {
6269                if (argc != 1) {
6270                        if (argc == 0)
6271                                (void) fprintf(stderr,
6272                                    gettext("missing filesystem argument\n"));
6273                        else
6274                                (void) fprintf(stderr,
6275                                    gettext("too many arguments\n"));
6276                        usage(B_FALSE);
6277                }

6279                /*
6280                 * We have an argument, but it may be a full path or a ZFS
6281                 * filesystem.  Pass full paths off to unmount_path() (shared by
6282                 * manual_unmount), otherwise open the filesystem and pass to
6283                 * zfs_unmount().
6284                 */
6285                if (argv[0][0] == '/')
6286                        return (unshare_unmount_path(op, argv[0],
6287                            flags, B_FALSE));

6289                if ((zhp = zfs_open(g_zfs, argv[0],
6290                    ZFS_TYPE_FILESYSTEM)) == NULL)
6291                        return (1);

6293                verify(zfs_prop_get(zhp, op == OP_SHARE ?
6294                    ZFS_PROP_SHARENFS : ZFS_PROP_MOUNTPOINT,
6295                    nfs_mnt_prop, sizeof (nfs_mnt_prop), NULL,
6296                    NULL, 0, B_FALSE) == 0);

6298                switch (op) {
6299                case OP_SHARE:
6300                        verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS,
6301                            nfs_mnt_prop,
6302                            sizeof (nfs_mnt_prop),
6303                            NULL, NULL, 0, B_FALSE) == 0);
6304                        verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB,
6305                            sharesmb, sizeof (sharesmb), NULL, NULL,
6306                            0, B_FALSE) == 0);

6308                        if (strcmp(nfs_mnt_prop, "off") == 0 &&
6309                            strcmp(sharesmb, "off") == 0) {
6310                                (void) fprintf(stderr, gettext("cannot "
6311                                    "unshare '%s': legacy share\n"),
6312                                    zfs_get_name(zhp));
6313                                (void) fprintf(stderr, gettext("use "
6314                                    "unshare(1M) to unshare this "
6315                                    "filesystem\n"));
6316                                ret = 1;
6317                        } else if (!zfs_is_shared(zhp)) {
6318                                (void) fprintf(stderr, gettext("cannot "
6319                                    "unshare '%s': not currently "
6320                                    "shared\n"), zfs_get_name(zhp));
6321                                ret = 1;
6322                        } else if (zfs_unshareall(zhp) != 0) {
6323                                ret = 1;
6324                        }
6325                        break;

6327                case OP_MOUNT:
6328                        if (strcmp(nfs_mnt_prop, "legacy") == 0) {
```

```
6329                                (void) fprintf(stderr, gettext("cannot "
6330                                    "unmount '%s': legacy "
6331                                    "mountpoint\n"), zfs_get_name(zhp));
6332                                (void) fprintf(stderr, gettext("use "
6333                                    "umount(1M) to unmount this "
6334                                    "filesystem\n"));
6335                                ret = 1;
6336                        } else if (!zfs_is_mounted(zhp, NULL)) {
6337                                (void) fprintf(stderr, gettext("cannot "
6338                                    "unmount '%s': not currently "
6339                                    "mounted\n"),
6340                                    zfs_get_name(zhp));
6341                                ret = 1;
6342                        } else if (zfs_unmountall(zhp, flags) != 0) {
6343                                ret = 1;
6344                        }
6345                        break;
6346                }

6348                zfs_close(zhp);
6349        }

6351        return (ret);
6352 }

6354 /*
6355  * zfs unmount -a
6356  * zfs unmount filesystem
6357  *
6358  * Unmount all filesystems, or a specific ZFS filesystem.
6359  */
6360 static int
6361 zfs_do_unmount(int argc, char **argv)
6362 {
6363        return (unshare_unmount(OP_MOUNT, argc, argv));
6364 }

6366 /*
6367  * zfs unshare -a
6368  * zfs unshare filesystem
6369  *
6370  * Unshare all filesystems, or a specific ZFS filesystem.
6371  */
6372 static int
6373 zfs_do_unshare(int argc, char **argv)
6374 {
6375        return (unshare_unmount(OP_SHARE, argc, argv));
6376 }

6378 /*
6379  * Called when invoked as /etc/fs/zfs/mount.  Do the mount if the mountpoint is
6380  * 'legacy'.  Otherwise, complain that use should be using 'zfs mount'.
6381  */
6382 static int
6383 manual_mount(int argc, char **argv)
6384 {
6385        zfs_handle_t *zhp;
6386        char mountpoint[ZFS_MAXPROPLEN];
6387        char mntopts[MNT_LINE_MAX] = { '\0' };
6388        int ret = 0;
6389        int c;
6390        int flags = 0;
6391        char *dataset, *path;

6393        /* check options */
6394        while ((c = getopt(argc, argv, ":mo:O")) != -1) {
```



```
6395                switch (c) {
6396                case 'o':
6397                        (void) strlcpy(mntopts, optarg, sizeof (mntopts));
6398                        break;
6399                case 'O':
6400                        flags |= MS_OVERLAY;
6401                        break;
6402                case 'm':
6403                        flags |= MS_NOMNTTAB;
6404                        break;
6405                case ':':
6406                        (void) fprintf(stderr, gettext("missing argument for "
6407                            "'%c' option\n"), optopt);
6408                        usage(B_FALSE);
6409                        break;
6410                case '?':
6411                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6412                            optopt);
6413                        (void) fprintf(stderr, gettext("usage: mount [-o opts] "
6414                            "<path>\n"));
6415                        return (2);
6416                }
6417        }

6419        argc -= optind;
6420        argv += optind;

6422        /* check that we only have two arguments */
6423        if (argc != 2) {
6424                if (argc == 0)
6425                        (void) fprintf(stderr, gettext("missing dataset "
6426                            "argument\n"));
6427                else if (argc == 1)
6428                        (void) fprintf(stderr,
6429                            gettext("missing mountpoint argument\n"));
6430                else
6431                        (void) fprintf(stderr, gettext("too many arguments\n"));
6432                (void) fprintf(stderr, "usage: mount <dataset> <mountpoint>\n");
6433                return (2);
6434        }

6436        dataset = argv[0];
6437        path = argv[1];

6439        /* try to open the dataset */
6440        if ((zhp = zfs_open(g_zfs, dataset, ZFS_TYPE_FILESYSTEM)) == NULL)
6441                return (1);

6443        (void) zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
6444            sizeof (mountpoint), NULL, NULL, 0, B_FALSE);

6446        /* check for legacy mountpoint and complain appropriately */
6447        ret = 0;
6448        if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
6449                if (mount(dataset, path, MS_OPTIONSTR | flags, MNTTYPE_ZFS,
6450                    NULL, 0, mntopts, sizeof (mntopts)) != 0) {
6451                        (void) fprintf(stderr, gettext("mount failed: %s\n"),
6452                            strerror(errno));
6453                        ret = 1;
6454                }
6455        } else {
6456                (void) fprintf(stderr, gettext("filesystem '%s' cannot be "
6457                    "mounted using 'mount -F zfs'\n"), dataset);
6458                (void) fprintf(stderr, gettext("Use 'zfs set mountpoint=%s' "
6459                    "instead.\n"), path);
6460                (void) fprintf(stderr, gettext("If you must use 'mount -F zfs' "
```

```
6461                    "or /etc/vfstab, use 'zfs set mountpoint=legacy'.\n"));
6462                (void) fprintf(stderr, gettext("See zfs(1M) for more "
6463                    "information.\n"));
6464                ret = 1;
6465        }

6467        return (ret);
6468 }

6470 /*
6471  * Called when invoked as /etc/fs/zfs/umount.  Unlike a manual mount, we allow
6472  * unmounts of non-legacy filesystems, as this is the dominant administrative
6473  * interface.
6474  */
6475 static int
6476 manual_unmount(int argc, char **argv)
6477 {
6478        int flags = 0;
6479        int c;

6481        /* check options */
6482        while ((c = getopt(argc, argv, "f")) != -1) {
6483                switch (c) {
6484                case 'f':
6485                        flags = MS_FORCE;
6486                        break;
6487                case '?':
6488                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6489                            optopt);
6490                        (void) fprintf(stderr, gettext("usage: unmount [-f] "
6491                            "<path>\n"));
6492                        return (2);
6493                }
6494        }

6496        argc -= optind;
6497        argv += optind;

6499        /* check arguments */
6500        if (argc != 1) {
6501                if (argc == 0)
6502                        (void) fprintf(stderr, gettext("missing path "
6503                            "argument\n"));
6504                else
6505                        (void) fprintf(stderr, gettext("too many arguments\n"));
6506                (void) fprintf(stderr, gettext("usage: unmount [-f] <path>\n"));
6507                return (2);
6508        }

6510        return (unshare_unmount_path(OP_MOUNT, argv[0], flags, B_TRUE));
6511 }

6513 static int
6514 find_command_idx(char *command, int *idx)
6515 {
6516        int i;

6518        for (i = 0; i < NCOMMAND; i++) {
6519                if (command_table[i].name == NULL)
6520                        continue;

6522                if (strcmp(command, command_table[i].name) == 0) {
6523                        *idx = i;
6524                        return (0);
6525                }
6526        }
```



```
6527        return (1);
6528 }

6530 static int
6531 zfs_do_diff(int argc, char **argv)
6532 {
6533        zfs_handle_t *zhp;
6534        int flags = 0;
6535        char *tosnap = NULL;
6536        char *fromsnap = NULL;
6537        char *atp, *copy;
6538        int err = 0;
6539        int c;

6541        while ((c = getopt(argc, argv, "FHt")) != -1) {
6542                switch (c) {
6543                case 'F':
6544                        flags |= ZFS_DIFF_CLASSIFY;
6545                        break;
6546                case 'H':
6547                        flags |= ZFS_DIFF_PARSEABLE;
6548                        break;
6549                case 't':
6550                        flags |= ZFS_DIFF_TIMESTAMP;
6551                        break;
6552                default:
6553                        (void) fprintf(stderr,
6554                            gettext("invalid option '%c'\n"), optopt);
6555                        usage(B_FALSE);
6556                }
6557        }

6559        argc -= optind;
6560        argv += optind;

6562        if (argc < 1) {
6563                (void) fprintf(stderr,
6564                gettext("must provide at least one snapshot name\n"));
6565                usage(B_FALSE);
6566        }

6568        if (argc > 2) {
6569                (void) fprintf(stderr, gettext("too many arguments\n"));
6570                usage(B_FALSE);
6571        }

6573        fromsnap = argv[0];
6574        tosnap = (argc == 2) ? argv[1] : NULL;

6576        copy = NULL;
6577        if (*fromsnap != '@')
6578                copy = strdup(fromsnap);
6579        else if (tosnap)
6580                copy = strdup(tosnap);
6581        if (copy == NULL)
6582                usage(B_FALSE);

6584        if (atp = strchr(copy, '@'))
6585                *atp = '\0';

6587        if ((zhp = zfs_open(g_zfs, copy, ZFS_TYPE_FILESYSTEM)) == NULL)
6588                return (1);

6590        free(copy);

6592        /*
```

```
6593          * Ignore SIGPIPE so that the library can give us
6594          * information on any failure
6595          */
6596         (void) sigignore(SIGPIPE);

6598         err = zfs_show_diffs(zhp, STDOUT_FILENO, fromsnap, tosnap, flags);

6600         zfs_close(zhp);

6602         return (err != 0);
6603 }

6605 int
6606 main(int argc, char **argv)
6607 {
6608         int ret = 0;
6609         int i;
6610         char *progname;
6611         char *cmdname;

6613         (void) setlocale(LC_ALL, "");
6614         (void) textdomain(TEXT_DOMAIN);

6616         opterr = 0;

6618         if ((g_zfs = libzfs_init()) == NULL) {
6619                 (void) fprintf(stderr, gettext("internal error: failed to "
6620                     "initialize ZFS library\n"));
6621                 return (1);
6622         }

6624         zfs_save_arguments(argc, argv, history_str, sizeof (history_str));

6626         libzfs_print_on_error(g_zfs, B_TRUE);

6628         if ((mnttab_file = fopen(MNTTAB, "r")) == NULL) {
6629                 (void) fprintf(stderr, gettext("internal error: unable to "
6630                     "open %s\n"), MNTTAB);
6631                 return (1);
6632         }

6634         /*
6635          * This command also doubles as the /etc/fs mount and unmount program.
6636          * Determine if we should take this behavior based on argv[0].
6637          */
6638         progname = basename(argv[0]);
6639         if (strcmp(progname, "mount") == 0) {
6640                 ret = manual_mount(argc, argv);
6641         } else if (strcmp(progname, "umount") == 0) {
6642                 ret = manual_unmount(argc, argv);
6643         } else {
6644                 /*
6645                  * Make sure the user has specified some command.
6646                  */
6647                 if (argc < 2) {
6648                         (void) fprintf(stderr, gettext("missing command\n"));
6649                         usage(B_FALSE);
6650                 }

6652                 cmdname = argv[1];

6654                 /*
6655                  * The 'umount' command is an alias for 'unmount'
6656                  */
6657                 if (strcmp(cmdname, "umount") == 0)
6658                         cmdname = "unmount";
```



```
6660                 /*
6661                  * The 'recv' command is an alias for 'receive'
6662                  */
6663                 if (strcmp(cmdname, "recv") == 0)
6664                         cmdname = "receive";

6666                 /*
6667                  * Special case '-?'
6668                  */
6669                 if (strcmp(cmdname, "-?") == 0)
6670                         usage(B_TRUE);

6672                 /*
6673                  * Run the appropriate command.
6674                  */
6675                 libzfs_mnttab_cache(g_zfs, B_TRUE);
6676                 if (find_command_idx(cmdname, &i) == 0) {
6677                         current_command = &command_table[i];
6678                         ret = command_table[i].func(argc - 1, argv + 1);
6679                 } else if (strchr(cmdname, '=') != NULL) {
6680                         verify(find_command_idx("set", &i) == 0);
6681                         current_command = &command_table[i];
6682                         ret = command_table[i].func(argc, argv);
6683                 } else {
6684                         (void) fprintf(stderr, gettext("unrecognized "
6685                             "command '%s'\n"), cmdname);
6686                         usage(B_FALSE);
6687                 }
6688                 libzfs_mnttab_cache(g_zfs, B_FALSE);
6689         }

6691         (void) fclose(mnttab_file);

6693         if (ret == 0 && log_history)
6694                 (void) zpool_log_history(g_zfs, history_str);

6696         libzfs_fini(g_zfs);

6698         /*
6699          * The 'ZFS_ABORT' environment variable causes us to dump core on exit
6700          * for the purposes of running ::findleaks.
6701          */
6702         if (getenv("ZFS_ABORT") != NULL) {
6703                 (void) printf("dumping core by request\n");
6704                 abort();
6705         }

6707         return (ret);
6708 }
```

```
**********************************************************
   27018 Wed Oct 17 21:48:37 2012
new/usr/src/lib/libzfs/common/libzfs.h
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

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

 592 extern int zfs_send(zfs_handle_t *, const char *, const char *,
 593     sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);
 594 extern int zfs_fits_send(zfs_handle_t *, const char *, const char *,
 595     sendflags_t *, int, snapfilter_cb_t, void *);
 596 #endif /* ! codereview */

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

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

 608 extern int zfs_userspace(zfs_handle_t *, zfs_userquota_prop_t,
 609     zfs_userspace_cb_t, void *);

 611 extern int zfs_get_fsacl(zfs_handle_t *, nvlist_t **);
 612 extern int zfs_set_fsacl(zfs_handle_t *, boolean_t, nvlist_t *);

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

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

 621         /*
 622          * Only the tail of the sent snapshot path is appended to the
 623          * destination to determine the received snapshot name (ie, -e).
 624          */
 625         boolean_t istail;

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

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

 633         /* set "canmount=off" on all modified filesystems */
 634         boolean_t canmountoff;

 636         /* byteswap flag is used internally; callers need not specify */
 637         boolean_t byteswap;

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

 643 extern int zfs_receive(libzfs_handle_t *, const char *, recvflags_t *,
 644     int, avl_tree_t *);
```



```
 646 typedef enum diff_flags {
 647         ZFS_DIFF_PARSEABLE = 0x1,
 648         ZFS_DIFF_TIMESTAMP = 0x2,
 649         ZFS_DIFF_CLASSIFY = 0x4
 650 } diff_flags_t;

 652 extern int zfs_show_diffs(zfs_handle_t *, int, const char *, const char *,
 653     int);

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

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

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

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

 699 /*
 700  * When dealing with nvlists, verify() is extremely useful
 701  */
 702 #ifdef NDEBUG
 703 #define verify(EX)      ((void)(EX))
 704 #else
 705 #define verify(EX)      assert(EX)
 706 #endif

 708 /*
 709  * Utility function to convert a number to a human-readable form.
 710  */
 711 extern void zfs_nicenum(uint64_t, char *, size_t);
```

```
712 extern int zfs_nicestrtonum(libzfs_handle_t *, const char *, uint64_t *);
713
714 /*
715  * Given a device or file, determine if it is part of a pool.
716  */
717 extern int zpool_in_use(libzfs_handle_t *, int, pool_state_t *, char **,
718     boolean_t *);
719
720 /*
721  * Label manipulation.
722  */
723 extern int zpool_read_label(int, nvlist_t **);
724 extern int zpool_clear_label(int);
725
726 /* is this zvol valid for use as a dump device? */
727 extern int zvol_check_dump_config(char *);
728
729 /*
730  * Management interfaces for SMB ACL files
731  */
732
733 int zfs_smb_acl_add(libzfs_handle_t *, char *, char *, char *);
734 int zfs_smb_acl_remove(libzfs_handle_t *, char *, char *, char *);
735 int zfs_smb_acl_purge(libzfs_handle_t *, char *, char *);
736 int zfs_smb_acl_rename(libzfs_handle_t *, char *, char *, char *, char *);
737
738 /*
739  * Enable and disable datasets within a pool by mounting/unmounting and
740  * sharing/unsharing them.
741  */
742 extern int zpool_enable_datasets(zpool_handle_t *, const char *, int);
743 extern int zpool_disable_datasets(zpool_handle_t *, boolean_t);
744
745 /*
746  * Mappings between vdev and FRU.
747  */
748 extern void libzfs_fru_refresh(libzfs_handle_t *);
749 extern const char *libzfs_fru_lookup(libzfs_handle_t *, const char *);
750 extern const char *libzfs_fru_devpath(libzfs_handle_t *, const char *);
751 extern boolean_t libzfs_fru_compare(libzfs_handle_t *, const char *,
752     const char *);
753 extern boolean_t libzfs_fru_notself(libzfs_handle_t *, const char *);
754 extern int zpool_fru_set(zpool_handle_t *, uint64_t, const char *);
755
756 #ifdef  __cplusplus
757 }
758 #endif
759
760 #endif  /* _LIBZFS_H */
```

```
**********************************************************
   87476 Wed Oct 17 21:48:37 2012
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

3201 /*
3202  * Generate a fits stream for the dataset identified by the argument zhp.
3203  *
3204  * The content of the send stream is the snapshot identified by
3205  * 'tosnap'.  Incremental streams are requested from the snapshot identified
3206  * by "fromsnap" (if non-null)
3207  * Currently no recursive send is implemented
3208  */
3209 int
3210 zfs_fits_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
3211     sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
3212     void *cb_arg)
3213 {
3214         char errbuf[1024];
3215         char name[MAXPATHLEN];
3216         zfs_cmd_t zc = { 0 };
3217         libzfs_handle_t *hdl = zhp->zfs_hdl;
3218         zfs_handle_t *thdl;

3220         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3221             "cannot send '%s'"), zhp->zfs_name);

3223         if (fromsnap && fromsnap[0] == '\0') {
3224                 zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3225                     "zero-length incremental source"));
3226                 return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
3227         }

3229         (void) snprintf(name, sizeof (name), "%s@%s", zhp->zfs_name, tosnap);
3230         if ((thdl = zfs_open(hdl, name, ZFS_TYPE_SNAPSHOT)) == NULL) {
3231                 (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
3232                     "WARNING: could not send %s@%s: does not exist\n"),
3233                     zhp->zfs_name, tosnap);
3234                 return (B_TRUE);
3235         }
3236         zc.zc_sendobj = zfs_prop_get_int(thdl, ZFS_PROP_OBJSETID);
3237         zfs_close(thdl);

3239         if (fromsnap) {
3240                 (void) snprintf(name, sizeof (name), "%s@%s",
3241                     zhp->zfs_name, fromsnap);
3242                 if ((thdl = zfs_open(hdl, name, ZFS_TYPE_SNAPSHOT)) == NULL) {
3243                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
3244                             "WARNING: could not send %s@%s:\n"
3245                             "incremental source (%s@%s) does not exist\n"),
3246                             zhp->zfs_name, tosnap,
3247                             zhp->zfs_name, fromsnap);
3248                         return (B_TRUE);
3249                 }
3250                 zc.zc_fromobj = zfs_prop_get_int(thdl, ZFS_PROP_OBJSETID);
3251                 zfs_close(thdl);
3252         }

3254         assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

3256         (void) snprintf(zc.zc_name, sizeof (zc.zc_name), "%s@%s",
```



```
3257             zhp->zfs_name, tosnap);
3258         zc.zc_cookie = outfd;
3259         zc.zc_obj = 0;

3261         if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_FITS_SEND, &zc) != 0) {
3262                 char errbuf[1024];
3263                 (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3264                     "warning: cannot send '%s'"), zhp->zfs_name);

3266                 switch (errno) {
3267                 case EXDEV:
3268                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3269                             "not an earlier snapshot from the same fs"));
3270                         return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
3271                 case ENOENT:
3272                         if (zfs_dataset_exists(hdl, zc.zc_name,
3273                             ZFS_TYPE_SNAPSHOT)) {
3274                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3275                                     "incremental source (@%s) does not exist"),
3276                                     zc.zc_value);
3277                         }
3278                         return (zfs_error(hdl, EZFS_NOENT, errbuf));
3279                 case EDQUOT:
3280                 case EFBIG:
3281                 case EIO:
3282                 case ENOLINK:
3283                 case ENOSPC:
3284                 case ENOSTR:
3285                 case ENXIO:
3286                 case EPIPE:
3287                 case ERANGE:
3288                 case EFAULT:
3289                 case EROFS:
3290                         zfs_error_aux(hdl, strerror(errno));
3291                         return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));
3292                 default:
3293                         return (zfs_standard_error(hdl, errno, errbuf));
3294                 }
3295         }

3297         return (0);
3298 }
3299 #endif /* ! codereview */
```

```
**********************************************************
    5399 Wed Oct 17 21:48:37 2012
new/usr/src/lib/libzfs/common/mapfile-vers
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2011 Nexenta Systems, Inc. All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#
# MAPFILE HEADER START
#
# WARNING:  STOP NOW.  DO NOT MODIFY THIS FILE.
# Object versioning must comply with the rules detailed in
#
#       usr/src/lib/README.mapfiles
#
# You should not be making modifications here until you've read the most current
# copy of that file. If you need help, contact a gatekeeper for guidance.
#
# MAPFILE HEADER END
#

$mapfile_version 2

SYMBOL_VERSION SUNWprivate_1.1 {
    global:
        fletcher_2_native;
        fletcher_2_byteswap;
        fletcher_4_native;
        fletcher_4_byteswap;
        fletcher_4_incremental_native;
        fletcher_4_incremental_byteswap;
        libzfs_add_handle;
        libzfs_dataset_cmp;
        libzfs_errno;
        libzfs_error_action;
        libzfs_error_description;
        libzfs_fini;
        libzfs_fru_compare;
        libzfs_fru_devpath;
        libzfs_fru_lookup;
        libzfs_fru_notself;
        libzfs_fru_refresh;
```



```
        libzfs_init;
        libzfs_mnttab_cache;
        libzfs_print_on_error;
        spa_feature_table;
        zfs_allocatable_devs;
        zfs_asprintf;
        zfs_clone;
        zfs_close;
        zfs_create;
        zfs_create_ancestors;
        zfs_dataset_exists;
        zfs_deleg_share_nfs;
        zfs_destroy;
        zfs_destroy_snaps;
        zfs_destroy_snaps_nvl;
        zfs_expand_proplist;
        zfs_fits_send;
#endif /* ! codereview */
        zfs_get_handle;
        zfs_get_holds;
        zfs_get_name;
        zfs_get_pool_handle;
        zfs_get_user_props;
        zfs_get_type;
        zfs_handle_dup;
        zfs_history_event_names;
        zfs_hold;
        zfs_is_mounted;
        zfs_is_shared;
        zfs_is_shared_nfs;
        zfs_is_shared_smb;
        zfs_iter_children;
        zfs_iter_dependents;
        zfs_iter_filesystems;
        zfs_iter_root;
        zfs_iter_snapshots;
        zfs_iter_snapshots_sorted;
        zfs_iter_snapspec;
        zfs_mount;
        zfs_name_to_prop;
        zfs_name_valid;
        zfs_nicenum;
        zfs_nicestrtonum;
        zfs_open;
        zfs_path_to_zhandle;
        zfs_promote;
        zfs_prop_align_right;
        zfs_prop_column_name;
        zfs_prop_default_numeric;
        zfs_prop_default_string;
        zfs_prop_get;
        zfs_prop_get_int;
        zfs_prop_get_numeric;
        zfs_prop_get_recvd;
        zfs_prop_get_table;
        zfs_prop_get_userquota_int;
        zfs_prop_get_userquota;
        zfs_prop_get_written_int;
        zfs_prop_get_written;
        zfs_prop_inherit;
        zfs_prop_inheritable;
        zfs_prop_init;
        zfs_prop_is_string;
        zfs_prop_readonly;
        zfs_prop_set;
        zfs_prop_string_to_index;
```

```
125         zfs_prop_to_name;
126         zfs_prop_user;
127         zfs_prop_userquota;
128         zfs_prop_valid_for_type;
129         zfs_prop_values;
130         zfs_prop_written;
131         zfs_prune_proplist;
132         zfs_receive;
133         zfs_refresh_properties;
134         zfs_release;
135         zfs_rename;
136         zfs_rollback;
137         zfs_save_arguments;
138         zfs_send;
139         zfs_share;
140         zfs_shareall;
141         zfs_share_nfs;
142         zfs_share_smb;
143         zfs_show_diffs;
144         zfs_smb_acl_add;
145         zfs_smb_acl_purge;
146         zfs_smb_acl_remove;
147         zfs_smb_acl_rename;
148         zfs_snapshot;
149         zfs_snapshot_nvl;
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_features;
188         zpool_get_handle;
189         zpool_get_history;
190         zpool_get_name;
```



```
191         zpool_get_physpath;
192         zpool_get_prop;
193         zpool_get_prop_int;
194         zpool_get_state;
195         zpool_get_status;
196         zpool_history_unpack;
197         zpool_import;
198         zpool_import_props;
199         zpool_import_status;
200         zpool_in_use;
201         zpool_is_bootable;
202         zpool_iter;
203         zpool_label_disk;
204         zpool_log_history;
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_prop;
226         zpool_state_to_name;
227         zpool_unmount_datasets;
228         zpool_upgrade;
229         zpool_vdev_attach;
230         zpool_vdev_clear;
231         zpool_vdev_degrade;
232         zpool_vdev_detach;
233         zpool_vdev_fault;
234         zpool_vdev_name;
235         zpool_vdev_offline;
236         zpool_vdev_online;
237         zpool_vdev_remove;
238         zpool_vdev_split;
239         zprop_free_list;
240         zprop_get_list;
241         zprop_iter;
242         zprop_print_one_property;
243         zprop_width;
244         zvol_check_dump_config;
245         zvol_volsize_to_reservation;
246     local:
247         *;
248 };
```

```
**********************************************************
   42960 Wed Oct 17 21:48:38 2012
new/usr/src/uts/common/Makefile.files
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#

#
# This Makefile defines all file modules for the directory uts/common
# and its children. These are the source files which may be considered
# common to all SunOS systems.

i386_CORE_OBJS += \
                atomic.o        \
                avintr.o        \
                pic.o

sparc_CORE_OBJS +=

COMMON_CORE_OBJS +=             \
                beep.o          \
                bitset.o        \
                bp_map.o        \
                brand.o         \
                cpucaps.o       \
                cmt.o           \
                cmt_policy.o    \
                cpu.o           \
                cpu_event.o     \
                cpu_intr.o      \
                cpu_pm.o        \
                cpupart.o       \
                cap_util.o      \
                disp.o          \
                group.o         \
                kstat_fr.o      \
                iscsiboot_prop.o        \
                lgrp.o          \
```



```
                lgrp_topo.o     \
                mmapobj.o       \
                mutex.o         \
                page_lock.o     \
                page_retire.o   \
                panic.o         \
                param.o         \
                pg.o            \
                pghw.o          \
                putnext.o       \
                rctl_proc.o     \
                rwlock.o        \
                seg_kmem.o      \
                softint.o       \
                string.o        \
                strtol.o        \
                strtoul.o       \
                strtoll.o       \
                strtoull.o      \
                thread_intr.o   \
                vm_page.o       \
                vm_pagelist.o   \
                zlib_obj.o      \
                clock_tick.o

CORE_OBJS +=    $(COMMON_CORE_OBJS) $($(MACH)_CORE_OBJS)

ZLIB_OBJS =     zutil.o zmod.o zmod_subr.o \
                adler32.o crc32.o deflate.o inffast.o \
                inflate.o inftrees.o trees.o

GENUNIX_OBJS += \
                access.o        \
                acl.o           \
                acl_common.o    \
                adjtime.o       \
                alarm.o         \
                aio_subr.o      \
                auditsys.o      \
                audit_core.o    \
                audit_zone.o    \
                audit_memory.o  \
                autoconf.o      \
                avl.o           \
                bdev_dsort.o    \
                bio.o           \
                bitmap.o        \
                blabel.o        \
                brandsys.o      \
                bz2blocksort.o  \
                bz2compress.o   \
                bz2decompress.o \
                bz2randtable.o  \
                bz2bzlib.o      \
                bz2crctable.o   \
                bz2huffman.o    \
                callb.o         \
                callout.o       \
                chdir.o         \
                chmod.o         \
                chown.o         \
                cladm.o         \
                class.o         \
                clock.o         \
                clock_highres.o \
                clock_realtime.o\
```

```
125                 close.o         \
126                 compress.o      \
127                 condvar.o       \
128                 conf.o          \
129                 console.o       \
130                 contract.o      \
131                 copyops.o       \
132                 core.o          \
133                 corectl.o       \
134                 cred.o          \
135                 cs_stubs.o      \
136                 dacf.o          \
137                 dacf_clnt.o     \
138                 damap.o \
139                 cyclic.o        \
140                 ddi.o           \
141                 ddifm.o         \
142                 ddi_hp_impl.o   \
143                 ddi_hp_ndi.o    \
144                 ddi_intr.o      \
145                 ddi_intr_impl.o \
146                 ddi_intr_irm.o  \
147                 ddi_nodeid.o    \
148                 ddi_timer.o     \
149                 devcfg.o        \
150                 devcache.o      \
151                 device.o        \
152                 devid.o         \
153                 devid_cache.o   \
154                 devid_scsi.o    \
155                 devid_smp.o     \
156                 devpolicy.o     \
157                 disp_lock.o     \
158                 dnlc.o          \
159                 driver.o        \
160                 dumpsubr.o      \
161                 driver_lyr.o    \
162                 dtrace_subr.o   \
163                 errorq.o        \
164                 etheraddr.o     \
165                 evchannels.o    \
166                 exacct.o        \
167                 exacct_core.o   \
168                 exec.o          \
169                 exit.o          \
170                 fbio.o          \
171                 fcntl.o         \
172                 fdbuffer.o      \
173                 fdsync.o        \
174                 fem.o           \
175                 ffs.o           \
176                 fio.o           \
177                 flock.o         \
178                 fm.o            \
179                 fork.o          \
180                 vpm.o           \
181                 fs_reparse.o    \
182                 fs_subr.o       \
183                 fsflush.o       \
184                 ftrace.o        \
185                 getcwd.o        \
186                 getdents.o      \
187                 getloadavg.o    \
188                 getpagesizes.o  \
189                 getpid.o        \
190                 gfs.o           \
```



```
191                 rusagesys.o     \
192                 gid.o           \
193                 groups.o        \
194                 grow.o          \
195                 hat_refmod.o    \
196                 id32.o          \
197                 id_space.o      \
198                 inet_ntop.o     \
199                 instance.o      \
200                 ioctl.o         \
201                 ip_cksum.o      \
202                 issetugid.o     \
203                 ippconf.o       \
204                 kcpc.o          \
205                 kdi.o           \
206                 kiconv.o        \
207                 klpd.o          \
208                 kmem.o          \
209                 ksyms_snapshot.o        \
210                 l_strplumb.o    \
211                 labelsys.o      \
212                 link.o          \
213                 list.o          \
214                 lockstat_subr.o \
215                 log_sysevent.o  \
216                 logsubr.o       \
217                 lookup.o        \
218                 lseek.o         \
219                 ltos.o          \
220                 lwp.o           \
221                 lwp_create.o    \
222                 lwp_info.o      \
223                 lwp_self.o      \
224                 lwp_sobj.o      \
225                 lwp_timer.o     \
226                 lwpsys.o        \
227                 main.o          \
228                 mmapobjsys.o    \
229                 memcntl.o       \
230                 memstr.o        \
231                 lgrpsys.o       \
232                 mkdir.o         \
233                 mknod.o         \
234                 mount.o         \
235                 move.o          \
236                 msacct.o        \
237                 multidata.o     \
238                 nbmlock.o       \
239                 ndifm.o         \
240                 nice.o          \
241                 netstack.o      \
242                 ntptime.o       \
243                 nvpair.o        \
244                 nvpair_alloc_system.o   \
245                 nvpair_alloc_fixed.o    \
246                 fnvpair.o       \
247                 octet.o         \
248                 open.o          \
249                 p_online.o      \
250                 pathconf.o      \
251                 pathname.o      \
252                 pause.o         \
253                 serializer.o    \
254                 pci_intr_lib.o  \
255                 pci_cap.o       \
256                 pcifm.o         \
```

```
257                 pgrp.o          \
258                 pgrpsys.o       \
259                 pid.o           \
260                 pkp_hash.o      \
261                 policy.o        \
262                 poll.o          \
263                 pool.o          \
264                 pool_pset.o     \
265                 port_subr.o     \
266                 ppriv.o         \
267                 printf.o        \
268                 priocntl.o      \
269                 priv.o          \
270                 priv_const.o    \
271                 proc.o          \
272                 procset.o       \
273                 processor_bind.o        \
274                 processor_info.o        \
275                 profil.o        \
276                 project.o       \
277                 qsort.o         \
278                 rctl.o          \
279                 rctlsys.o       \
280                 readlink.o      \
281                 refstr.o        \
282                 rename.o        \
283                 resolvepath.o   \
284                 retire_store.o  \
285                 process.o       \
286                 rlimit.o        \
287                 rmap.o          \
288                 rw.o            \
289                 rwstlock.o      \
290                 sad_conf.o      \
291                 sid.o           \
292                 sidsys.o        \
293                 sched.o         \
294                 schedctl.o      \
295                 sctp_crc32.o    \
296                 seg_dev.o       \
297                 seg_kp.o        \
298                 seg_kpm.o       \
299                 seg_map.o       \
300                 seg_vn.o        \
301                 seg_spt.o       \
302                 semaphore.o     \
303                 sendfile.o      \
304                 session.o       \
305                 share.o         \
306                 shuttle.o       \
307                 sig.o           \
308                 sigaction.o     \
309                 sigaltstack.o   \
310                 signotify.o     \
311                 sigpending.o    \
312                 sigprocmask.o   \
313                 sigqueue.o      \
314                 sigsendset.o    \
315                 sigsuspend.o    \
316                 sigtimedwait.o  \
317                 sleepq.o        \
318                 sock_conf.o     \
319                 space.o         \
320                 sscanf.o        \
321                 stat.o          \
322                 statfs.o        \
```



```
323                 statvfs.o       \
324                 stol.o          \
325                 str_conf.o      \
326                 strcalls.o      \
327                 stream.o        \
328                 streamio.o      \
329                 strext.o        \
330                 strsubr.o       \
331                 strsun.o        \
332                 subr.o          \
333                 sunddi.o        \
334                 sunmdi.o        \
335                 sunndi.o        \
336                 sunpci.o        \
337                 sunpm.o         \
338                 sundlpi.o       \
339                 suntpi.o        \
340                 swap_subr.o     \
341                 swap_vnops.o    \
342                 symlink.o       \
343                 sync.o          \
344                 sysclass.o      \
345                 sysconfig.o     \
346                 sysent.o        \
347                 sysfs.o         \
348                 systeminfo.o    \
349                 task.o          \
350                 taskq.o         \
351                 tasksys.o       \
352                 time.o          \
353                 timer.o         \
354                 times.o         \
355                 timers.o        \
356                 thread.o        \
357                 tlabel.o        \
358                 tnf_res.o       \
359                 turnstile.o     \
360                 tty_common.o    \
361                 u8_textprep.o   \
362                 uadmin.o        \
363                 uconv.o         \
364                 ucredsys.o      \
365                 uid.o           \
366                 umask.o         \
367                 umount.o        \
368                 uname.o         \
369                 unix_bb.o       \
370                 unlink.o        \
371                 urw.o           \
372                 utime.o         \
373                 utssys.o        \
374                 uucopy.o        \
375                 vfs.o           \
376                 vfs_conf.o      \
377                 vmem.o          \
378                 vm_anon.o       \
379                 vm_as.o         \
380                 vm_meter.o      \
381                 vm_pageout.o    \
382                 vm_pvn.o        \
383                 vm_rm.o         \
384                 vm_seg.o        \
385                 vm_subr.o       \
386                 vm_swap.o       \
387                 vm_usage.o      \
388                 vnode.o         \
```

```
389                 vuid_queue.o    \
390                 vuid_store.o    \
391                 waitq.o         \
392                 watchpoint.o    \
393                 yield.o         \
394                 scsi_confdata.o \
395                 xattr.o         \
396                 xattr_common.o  \
397                 xdr_mblk.o      \
398                 xdr_mem.o       \
399                 xdr.o           \
400                 xdr_array.o     \
401                 xdr_refer.o     \
402                 xhat.o          \
403                 zone.o

405 #
406 #       Stubs for the stand-alone linker/loader
407 #
408 sparc_GENSTUBS_OBJS =   \
409         kobj_stubs.o

411 i386_GENSTUBS_OBJS =

413 COMMON_GENSTUBS_OBJS =

415 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($(MACH)_GENSTUBS_OBJS)

417 #
418 #       DTrace and DTrace Providers
419 #
420 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

422 SDT_OBJS += sdt_subr.o

424 PROFILE_OBJS += profile.o

426 SYSTRACE_OBJS += systrace.o

428 LOCKSTAT_OBJS += lockstat.o

430 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

432 DCPC_OBJS += dcpc.o

434 #
435 #       Driver (pseudo-driver) Modules
436 #
437 IPP_OBJS +=     ippctl.o

439 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
440             audio_fltdata.o audio_format.o audio_ctrl.o \
441             audio_grc3.o audio_output.o audio_input.o \
442             audio_oss.o audio_sun.o

444 AUDIOEMU10K_OBJS += audioemu10k.o

446 AUDIOENS_OBJS += audioens.o

448 AUDIOVIA823X_OBJS += audiovia823x.o

450 AUDIOVIA97_OBJS += audiovia97.o

452 AUDIO1575_OBJS += audio1575.o

454 AUDIO810_OBJS += audio810.o
```



```
456 AUDIOCMI_OBJS += audiocmi.o

458 AUDIOCMIHD_OBJS += audiocmihd.o

460 AUDIOHD_OBJS += audiohd.o

462 AUDIOIXP_OBJS += audioixp.o

464 AUDIOLS_OBJS += audiols.o

466 AUDIOP16X_OBJS += audiop16x.o

468 AUDIOPCI_OBJS += audiopci.o

470 AUDIOSOLO_OBJS += audiosolo.o

472 AUDIOTS_OBJS += audiots.o

474 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

476 BLKDEV_OBJS += blkdev.o

478 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

480 CONSKBD_OBJS += conskbd.o

482 CONSMS_OBJS +=  consms.o

484 OLDPTY_OBJS +=  tty_ptyconf.o

486 PTC_OBJS +=     tty_pty.o

488 PTSL_OBJS +=    tty_pts.o

490 PTM_OBJS +=     ptm.o

492 MII_OBJS +=     mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
493                 mii_marvell.o mii_realtek.o mii_other.o

495 PTS_OBJS +=     pts.o

497 PTY_OBJS +=     ptms_conf.o

499 SAD_OBJS +=     sad.o

501 MD4_OBJS +=     md4.o md4_mod.o

503 MD5_OBJS +=     md5.o md5_mod.o

505 SHA1_OBJS +=    sha1.o sha1_mod.o

507 SHA2_OBJS +=    sha2.o sha2_mod.o

509 IPGPC_OBJS +=   classifierddi.o classifier.o filters.o trie.o table.o \
510                 ba_table.o

512 DSCPMK_OBJS +=  dscpmk.o dscpmkddi.o

514 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

516 FLOWACCT_OBJS +=        flowacctddi.o flowacct.o

518 TOKENMT_OBJS += tokenmt.o tokenmtddi.o

520 TSWTCL_OBJS +=  tswtcl.o tswtclddi.o
```

```
522 ARP_OBJS +=     arpddi.o

524 ICMP_OBJS +=    icmpddi.o

526 ICMP6_OBJS +=   icmp6ddi.o

528 RTS_OBJS +=     rtsddi.o

530 IP_ICMP_OBJS =  icmp.o icmp_opt_data.o
531 IP_RTS_OBJS =   rts.o rts_opt_data.o
532 IP_TCP_OBJS =   tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
533                 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
534                 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
535 IP_UDP_OBJS =   udp.o udp_opt_data.o udp_tunables.o udp_stats.o
536 IP_SCTP_OBJS =  sctp.o sctp_opt_data.o sctp_output.o \
537                 sctp_init.o sctp_input.o sctp_cookie.o \
538                 sctp_conn.o sctp_error.o sctp_snmp.o \
539                 sctp_tunables.o sctp_shutdown.o sctp_common.o \
540                 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
541                 sctp_bind.o sctp_notify.o sctp_asconf.o \
542                 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
543                 sctp_misc.o
544 IP_ILB_OBJS =   ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o

546 IP_OBJS +=      igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
547                 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
548                 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
549                 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
550                 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_squeue.o \
551                 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
552                 ip_helper_stream.o ip_tunables.o \
553                 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
554                 conn_opt.o ip_attr.o ip_dce.o \
555                 $(IP_ICMP_OBJS) \
556                 $(IP_RTS_OBJS) \
557                 $(IP_TCP_OBJS) \
558                 $(IP_UDP_OBJS) \
559                 $(IP_SCTP_OBJS) \
560                 $(IP_ILB_OBJS)

562 IP6_OBJS +=     ip6ddi.o

564 HOOK_OBJS +=    hook.o

566 NETI_OBJS +=    neti_impl.o neti_mod.o neti_stack.o

568 KEYSOCK_OBJS += keysockddi.o keysock.o keysock_opt_data.o

570 IPNET_OBJS +=   ipnet.o ipnet_bpf.o

572 SPDSOCK_OBJS += spdsockddi.o spdsock.o spdsock_opt_data.o

574 IPSECESP_OBJS += ipsecespddi.o ipsecesp.o

576 IPSECAH_OBJS += ipsecahddi.o ipsecah.o sadb.o

578 SPPP_OBJS +=    sppp.o sppp_dlpi.o sppp_mod.o s_common.o

580 SPPPTUN_OBJS += sppptun.o sppptun_mod.o

582 SPPPASYN_OBJS += spppasyn.o spppasyn_mod.o

584 SPPPCOMP_OBJS += spppcomp.o spppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
585                 zlib.o
```



```
587 TCP_OBJS +=     tcpddi.o

589 TCP6_OBJS +=    tcp6ddi.o

591 NCA_OBJS +=     ncaddi.o

593 SDP_SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o

595 SCTP_SOCK_MOD_OBJS += sockmod_sctp.o socksctp.o socksctpsubr.o

597 PFP_SOCK_MOD_OBJS += sockmod_pfp.o

599 RDS_SOCK_MOD_OBJS += sockmod_rds.o

601 RDS_OBJS +=     rdsddi.o rdssubr.o rds_opt.o rds_ioctl.o

603 RDSIB_OBJS +=   rdsib.o rdsib_ib.o rdsib_cm.o rdsib_ep.o rdsib_buf.o \
604                 rdsib_debug.o rdsib_sc.o

606 RDSV3_OBJS +=   af_rds.o rdsv3_ddi.o bind.o loop.o threads.o connection.o \
607                 transport.o cong.o sysctl.o message.o rds_recv.o send.o \
608                 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
609                 ib_recv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
610                 rdsv3_sc.o rdsv3_debug.o rdsv3_impl.o rdma.o rdsv3_af_thr.o

612 ISER_OBJS +=    iser.o iser_cm.o iser_cq.o iser_ib.o iser_idm.o \
613                 iser_resource.o iser_xfer.o

615 UDP_OBJS +=     udpddi.o

617 UDP6_OBJS +=    udp6ddi.o

619 SY_OBJS +=      gentty.o

621 TCO_OBJS +=     ticots.o

623 TCOO_OBJS +=    ticotsord.o

625 TCL_OBJS +=     ticlts.o

627 TL_OBJS +=      tl.o

629 DUMP_OBJS +=    dump.o

631 BPF_OBJS +=     bpf.o bpf_filter.o bpf_mod.o bpf_dlt.o bpf_mac.o

633 CLONE_OBJS +=   clone.o

635 CN_OBJS +=      cons.o

637 DLD_OBJS +=     dld_drv.o dld_proto.o dld_str.o dld_flow.o

639 DLS_OBJS +=     dls.o dls_link.o dls_mod.o dls_stat.o dls_mgmt.o

641 GLD_OBJS +=     gld.o gldutil.o

643 MAC_OBJS +=     mac.o  mac_bcast.o mac_client.o mac_datapath_setup.o mac_flow.o
644                 mac_hio.o mac_mod.o mac_ndd.o mac_provider.o mac_sched.o \
645                 mac_protect.o mac_soft_ring.o mac_stat.o mac_util.o

647 MAC_6TO4_OBJS +=        mac_6to4.o

649 MAC_ETHER_OBJS +=       mac_ether.o

651 MAC_IPV4_OBJS +=        mac_ipv4.o
```

```
653 MAC_IPV6_OBJS +=        mac_ipv6.o

655 MAC_WIFI_OBJS +=        mac_wifi.o

657 MAC_IB_OBJS +=          mac_ib.o

659 IPTUN_OBJS +=   iptun_dev.o iptun_ctl.o iptun.o

661 AGGR_OBJS +=    aggr_dev.o aggr_ctl.o aggr_grp.o aggr_port.o \
662                 aggr_send.o aggr_recv.o aggr_lacp.o

664 SOFTMAC_OBJS += softmac_main.o softmac_ctl.o softmac_capab.o \
665                 softmac_dev.o softmac_stat.o softmac_pkt.o softmac_fp.o

667 NET80211_OBJS += net80211.o net80211_proto.o net80211_input.o \
668                  net80211_output.o net80211_node.o net80211_crypto.o \
669                  net80211_crypto_none.o net80211_crypto_wep.o net80211_ioctl.o \
670                  net80211_crypto_tkip.o net80211_crypto_ccmp.o  \
671                  net80211_ht.o

673 VNIC_OBJS +=    vnic_ctl.o vnic_dev.o

675 SIMNET_OBJS +=  simnet.o

677 IB_OBJS +=      ibnex.o ibnex_ioctl.o ibnex_hca.o

679 IBCM_OBJS +=    ibcm_impl.o ibcm_sm.o ibcm_ti.o ibcm_utils.o ibcm_path.o \
680                 ibcm_arp.o ibcm_arp_link.o

682 IBDM_OBJS +=    ibdm.o

684 IBDMA_OBJS +=   ibdma.o

686 IBMF_OBJS +=    ibmf.o ibmf_impl.o ibmf_dr.o ibmf_wqe.o ibmf_ud_dest.o ibmf_mod.
687                 ibmf_send.o ibmf_recv.o ibmf_handlers.o ibmf_trans.o \
688                 ibmf_timers.o ibmf_msg.o ibmf_utils.o ibmf_rmpp.o \
689                 ibmf_saa.o ibmf_saa_impl.o ibmf_saa_utils.o ibmf_saa_events.o

691 IBTL_OBJS +=    ibtl_impl.o ibtl_util.o ibtl_mem.o ibtl_handlers.o ibtl_qp.o \
692                 ibtl_cq.o ibtl_wr.o ibtl_hca.o ibtl_chan.o ibtl_cm.o \
693                 ibtl_mcg.o ibtl_ibnex.o ibtl_srq.o ibtl_part.o

695 TAVOR_OBJS +=   tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
696                 tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
697                 tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
698                 tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o

700 HERMON_OBJS +=  hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
701                 hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
702                 hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
703                 hermon_srq.o hermon_stats.o hermon_umap.o hermon_wr.o \
704                 hermon_fcoib.o hermon_fm.o

706 DAPLT_OBJS +=   daplt.o

708 SOL_OFS_OBJS += sol_cma.o sol_ib_cma.o sol_uobj.o \
709                 sol_ofs_debug_util.o sol_ofs_gen_util.o \
710                 sol_kverbs.o

712 SOL_UCMA_OBJS +=        sol_ucma.o

714 SOL_UVERBS_OBJS +=      sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
715                         sol_uverbs_hca.o sol_uverbs_qp.o

717 SOL_UMAD_OBJS += sol_umad.o
```



```
719 KSTAT_OBJS +=   kstat.o

721 KSYMS_OBJS +=   ksyms.o

723 INSTANCE_OBJS += inst_sync.o

725 IWSCN_OBJS +=   iwscons.o

727 LOFI_OBJS +=    lofi.o LzmaDec.o

729 FSSNAP_OBJS +=  fssnap.o

731 FSSNAPIF_OBJS += fssnap_if.o

733 MM_OBJS +=      mem.o

735 PHYSMEM_OBJS += physmem.o

737 OPTIONS_OBJS += options.o

739 WINLOCK_OBJS += winlockio.o

741 PM_OBJS +=      pm.o
742 SRN_OBJS +=     srn.o

744 PSEUDO_OBJS +=  pseudonex.o

746 RAMDISK_OBJS += ramdisk.o

748 LLC1_OBJS += llc1.o

750 USBKBM_OBJS += usbkbm.o

752 USBWCM_OBJS += usbwcm.o

754 BOFI_OBJS += bofi.o

756 HID_OBJS += hid.o

758 HWA_RC_OBJS += hwarc.o

760 USBSKEL_OBJS += usbskel.o

762 USBVC_OBJS += usbvc.o usbvc_v4l2.o

764 HIDPARSER_OBJS += hidparser.o

766 USB_AC_OBJS += usb_ac.o

768 USB_AS_OBJS += usb_as.o

770 USB_AH_OBJS += usb_ah.o

772 USBMS_OBJS += usbms.o

774 USBPRN_OBJS += usbprn.o

776 UGEN_OBJS += ugen.o

778 USBSER_OBJS += usbser.o usbser_rseq.o

780 USBSACM_OBJS += usbsacm.o

782 USBSER_KEYSPAN_OBJS += usbser_keyspan.o keyspan_dsd.o keyspan_pipe.o

784 USBS49_FW_OBJS += keyspan_49fw.o
```

```
786 USBSPRL_OBJS += usbser_pl2303.o pl2303_dsd.o

788 WUSB_CA_OBJS += wusb_ca.o

790 USBFTDI_OBJS += usbser_uftdi.o uftdi_dsd.o

792 USBECM_OBJS += usbecm.o

794 WC_OBJS += wscons.o vcons.o

796 VCONS_CONF_OBJS += vcons_conf.o

798 SCSI_OBJS +=    scsi_capabilities.o scsi_confsubr.o scsi_control.o \
799                 scsi_data.o scsi_fm.o scsi_hba.o scsi_reset_notify.o \
800                 scsi_resource.o scsi_subr.o scsi_transport.o scsi_watch.o \
801                 smp_transport.o

803 SCSI_VHCI_OBJS +=               scsi_vhci.o mpapi_impl.o scsi_vhci_tpgs.o

805 SCSI_VHCI_F_SYM_OBJS +=         sym.o

807 SCSI_VHCI_F_TPGS_OBJS +=        tpgs.o

809 SCSI_VHCI_F_ASYM_SUN_OBJS +=    asym_sun.o

811 SCSI_VHCI_F_SYM_HDS_OBJS +=     sym_hds.o

813 SCSI_VHCI_F_TAPE_OBJS +=        tape.o

815 SCSI_VHCI_F_TPGS_TAPE_OBJS +=   tpgs_tape.o

817 SGEN_OBJS +=    sgen.o

819 SMP_OBJS +=     smp.o

821 SATA_OBJS +=    sata.o

823 USBA_OBJS +=    hcdi.o  usba.o  usbai.o hubdi.o parser.o genconsole.o \
824                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
825                 usba_devdb.o usba10_calls.o usba_ugen.o whcdi.o wa.o
826 USBA_WITHOUT_WUSB_OBJS +=       hcdi.o  usba.o  usbai.o hubdi.o parser.o gencons
827                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
828                 usba_devdb.o usba10_calls.o usba_ugen.o

830 USBA10_OBJS +=  usba10.o

832 RSM_OBJS +=     rsm.o   rsmka_pathmanager.o     rsmka_util.o

834 RSMOPS_OBJS +=  rsmops.o

836 S1394_OBJS +=   t1394.o t1394_errmsg.o s1394.o s1394_addr.o s1394_asynch.o \
837                 s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
838                 s1394_fa.o s1394_fcp.o \
839                 s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o

841 HCI1394_OBJS += hci1394.o hci1394_async.o hci1394_attach.o hci1394_buf.o \
842                 hci1394_csr.o hci1394_detach.o hci1394_extern.o \
843                 hci1394_ioctl.o hci1394_isoch.o hci1394_isr.o \
844                 hci1394_ixl_comp.o hci1394_ixl_isr.o hci1394_ixl_misc.o \
845                 hci1394_ixl_update.o hci1394_misc.o hci1394_ohci.o \
846                 hci1394_q.o hci1394_s1394if.o hci1394_tlabel.o \
847                 hci1394_tlist.o hci1394_vendor.o

849 AV1394_OBJS +=  av1394.o av1394_as.o av1394_async.o av1394_cfgrom.o \
850                 av1394_cmp.o av1394_fcp.o av1394_isoch.o av1394_isoch_chan.o \
```



```
851                 av1394_isoch_recv.o av1394_isoch_xmit.o av1394_list.o \
852                 av1394_queue.o

854 DCAM1394_OBJS += dcam.o dcam_frame.o dcam_param.o dcam_reg.o \
855                 dcam_ring_buff.o

857 SCSA1394_OBJS += hba.o sbp2_driver.o sbp2_bus.o

859 SBP2_OBJS += cfgrom.o sbp2.o

861 PMODEM_OBJS += pmodem.o pmodem_cis.o cis.o cis_callout.o cis_handlers.o cis_para

863 DSW_OBJS +=     dsw.o dsw_dev.o ii_tree.o

865 NCALL_OBJS +=   ncall.o \
866                 ncall_stub.o

868 RDC_OBJS +=     rdc.o \
869                 rdc_dev.o \
870                 rdc_io.o \
871                 rdc_clnt.o \
872                 rdc_prot_xdr.o \
873                 rdc_svc.o \
874                 rdc_bitmap.o \
875                 rdc_health.o \
876                 rdc_subr.o \
877                 rdc_diskq.o

879 RDCSRV_OBJS +=  rdcsrv.o

881 RDCSTUB_OBJS += rdc_stub.o

883 SDBC_OBJS +=    sd_bcache.o \
884                 sd_bio.o \
885                 sd_conf.o \
886                 sd_ft.o \
887                 sd_hash.o \
888                 sd_io.o \
889                 sd_misc.o \
890                 sd_pcu.o \
891                 sd_tdaemon.o \
892                 sd_trace.o \
893                 sd_iob_impl0.o \
894                 sd_iob_impl1.o \
895                 sd_iob_impl2.o \
896                 sd_iob_impl3.o \
897                 sd_iob_impl4.o \
898                 sd_iob_impl5.o \
899                 sd_iob_impl6.o \
900                 sd_iob_impl7.o \
901                 safestore.o \
902                 safestore_ram.o

904 NSCTL_OBJS +=   nsctl.o \
905                 nsc_cache.o \
906                 nsc_disk.o \
907                 nsc_dev.o \
908                 nsc_freeze.o \
909                 nsc_gen.o \
910                 nsc_mem.o \
911                 nsc_ncallio.o \
912                 nsc_power.o \
913                 nsc_resv.o \
914                 nsc_rmspin.o \
915                 nsc_solaris.o \
916                 nsc_trap.o \
```

```
917                 nsc_list.o
918 UNISTAT_OBJS += spuni.o \
919                 spcs_s_k.o

921 NSKERN_OBJS +=  nsc_ddi.o \
922                 nsc_proc.o \
923                 nsc_raw.o \
924                 nsc_thread.o \
925                 nskernd.o

927 SV_OBJS +=      sv.o

929 PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
930                 pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

932 PMCS8001FW_C_OBJS +=    pmcs_fw_hdr.o
933 PMCS8001FW_OBJS +=              $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

935 #
936 #       Build up defines and paths.

938 ST_OBJS +=      st.o    st_conf.o

940 EMLXS_OBJS +=   emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
941                 emlxs_download.o emlxs_dump.o emlxs_els.o emlxs_event.o \
942                 emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
943                 emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
944                 emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
945                 emlxs_thread.o

947 EMLXS_FW_OBJS +=        emlxs_fw.o

949 OCE_OBJS +=     oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
950                 oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
951                 oce_utils.o

953 FCT_OBJS += discovery.o fct.o

955 QLT_OBJS += 2400.o 2500.o 8100.o qlt.o qlt_dma.o

957 SRPT_OBJS += srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

959 FCOE_OBJS += fcoe.o fcoe_eth.o fcoe_fc.o

961 FCOET_OBJS += fcoet.o fcoet_eth.o fcoet_fc.o

963 FCOEI_OBJS += fcoei.o fcoei_eth.o fcoei_lv.o

965 ISCSIT_SHARED_OBJS += \
966                 iscsit_common.o

968 ISCSIT_OBJS +=  $(ISCSIT_SHARED_OBJS) \
969                 iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
970                 iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
971                 iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

973 PPPT_OBJS +=    alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o

975 STMF_OBJS += lun_map.o stmf.o

977 STMF_SBD_OBJS += sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

979 SYSMSG_OBJS +=  sysmsg.o

981 SES_OBJS +=     ses.o ses_sen.o ses_safte.o ses_ses.o
```



```
983 TNF_OBJS +=     tnf_buf.o       tnf_trace.o     tnf_writer.o    trace_init.o \
984                 trace_funcs.o   tnf_probe.o     tnf.o

986 LOGINDMUX_OBJS += logindmux.o

988 DEVINFO_OBJS += devinfo.o

990 DEVPOLL_OBJS += devpoll.o

992 DEVPOOL_OBJS += devpool.o

994 I8042_OBJS +=   i8042.o

996 KB8042_OBJS +=  \
997                 at_keyprocess.o \
998                 kb8042.o        \
999                 kb8042_keytables.o

1001 MOUSE8042_OBJS += mouse8042.o

1003 FDC_OBJS +=     fdc.o

1005 ASY_OBJS +=     asy.o

1007 ECPP_OBJS +=    ecpp.o

1009 VUIDM3P_OBJS += vuidmice.o vuidm3p.o

1011 VUIDM4P_OBJS += vuidmice.o vuidm4p.o

1013 VUIDM5P_OBJS += vuidmice.o vuidm5p.o

1015 VUIDPS2_OBJS += vuidmice.o vuidps2.o

1017 HPCSVC_OBJS += hpcsvc.o

1019 PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

1021 PCIHPNEXUS_OBJS += pcihp.o

1023 OPENEEPR_OBJS += openprom.o

1025 RANDOM_OBJS += random.o

1027 PSHOT_OBJS += pshot.o

1029 GEN_DRV_OBJS += gen_drv.o

1031 TCLIENT_OBJS += tclient.o

1033 TPHCI_OBJS += tphci.o

1035 TVHCI_OBJS += tvhci.o

1037 EMUL64_OBJS += emul64.o emul64_bsd.o

1039 FCP_OBJS += fcp.o

1041 FCIP_OBJS += fcip.o

1043 FCSM_OBJS += fcsm.o

1045 FCTL_OBJS += fctl.o

1047 FP_OBJS += fp.o
```

```
1049 QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
1050         ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o

1052 QLC_FW_2200_OBJS += ql_fw_2200.o

1054 QLC_FW_2300_OBJS += ql_fw_2300.o

1056 QLC_FW_2400_OBJS += ql_fw_2400.o

1058 QLC_FW_2500_OBJS += ql_fw_2500.o

1060 QLC_FW_6322_OBJS += ql_fw_6322.o

1062 QLC_FW_8100_OBJS += ql_fw_8100.o

1064 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o

1066 ZCONS_OBJS += zcons.o

1068 NV_SATA_OBJS += nv_sata.o

1070 SI3124_OBJS += si3124.o

1072 AHCI_OBJS += ahci.o

1074 PCIIDE_OBJS += pci-ide.o

1076 PCEPP_OBJS += pcepp.o

1078 CPC_OBJS += cpc.o

1080 CPUID_OBJS += cpuid_drv.o

1082 SYSEVENT_OBJS += sysevent.o

1084 BL_OBJS += bl.o

1086 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1087             drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1088             drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1089             drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1090             drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o

1092 FM_OBJS += devfm.o devfm_machdep.o

1094 RTLS_OBJS +=    rtls.o

1096 #
1097 #                       exec modules
1098 #
1099 AOUTEXEC_OBJS +=aout.o

1101 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o

1103 INTPEXEC_OBJS +=intp.o

1105 SHBINEXEC_OBJS +=shbin.o

1107 JAVAEXEC_OBJS +=java.o

1109 #
1110 #                       file system modules
1111 #
1112 AUTOFS_OBJS +=  auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o

1114 CACHEFS_OBJS += cachefs_cnode.o         cachefs_cod.o \
```



```
1115                 cachefs_dir.o          cachefs_dlog.o  cachefs_filegrp.o \
1116                 cachefs_fscache.o      cachefs_ioctl.o cachefs_log.o \
1117                 cachefs_module.o \
1118                 cachefs_noopc.o        cachefs_resource.o \
1119                 cachefs_strict.o \
1120                 cachefs_subr.o         cachefs_vfsops.o \
1121                 cachefs_vnops.o

1123 DCFS_OBJS +=    dc_vnops.o

1125 DEVFS_OBJS +=   devfs_subr.o    devfs_vfsops.o  devfs_vnops.o

1127 DEV_OBJS  +=    sdev_subr.o     sdev_vfsops.o   sdev_vnops.o    \
1128                 sdev_ptsops.o   sdev_zvolops.o  sdev_comm.o     \
1129                 sdev_profile.o  sdev_ncache.o   sdev_netops.o   \
1130                 sdev_ipnetops.o \
1131                 sdev_vtops.o

1133 CTFS_OBJS +=    ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1134                 ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o

1136 OBJFS_OBJS +=   objfs_vfs.o     objfs_root.o    objfs_common.o \
1137                 objfs_odir.o    objfs_data.o

1139 FDFS_OBJS +=    fdops.o

1141 FIFO_OBJS +=    fifosubr.o      fifovnops.o

1143 PIPE_OBJS +=    pipe.o

1145 HSFS_OBJS +=    hsfs_node.o     hsfs_subr.o     hsfs_vfsops.o   hsfs_vnops.o \
1146                 hsfs_susp.o     hsfs_rrip.o     hsfs_susp_subr.o

1148 LOFS_OBJS +=    lofs_subr.o     lofs_vfsops.o   lofs_vnops.o

1150 NAMEFS_OBJS +=  namevfs.o       namevno.o

1152 NFS_OBJS +=     nfs_client.o    nfs_common.o    nfs_dump.o \
1153                 nfs_subr.o      nfs_vfsops.o    nfs_vnops.o \
1154                 nfs_xdr.o       nfs_sys.o       nfs_strerror.o \
1155                 nfs3_vfsops.o   nfs3_vnops.o    nfs3_xdr.o \
1156                 nfs_acl_vnops.o nfs_acl_xdr.o   nfs4_vfsops.o \
1157                 nfs4_vnops.o    nfs4_xdr.o      nfs4_idmap.o \
1158                 nfs4_shadow.o   nfs4_subr.o \
1159                 nfs4_attr.o     nfs4_rnode.o    nfs4_client.o \
1160                 nfs4_acache.o   nfs4_common.o   nfs4_client_state.o \
1161                 nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1162                 nfs4_client_debug.o     nfs_stats.o \
1163                 nfs4_acl.o      nfs4_stub_vnops.o       nfs_cmd.o

1165 NFSSRV_OBJS +=  nfs_server.o    nfs_srv.o       nfs3_srv.o \
1166                 nfs_acl_srv.o   nfs_auth.o      nfs_auth_xdr.o \
1167                 nfs_export.o    nfs_log.o       nfs_log_xdr.o \
1168                 nfs4_srv.o      nfs4_state.o    nfs4_srv_attr.o \
1169                 nfs4_srv_ns.o   nfs4_db.o       nfs4_srv_deleg.o \
1170                 nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o

1172 SMBSRV_SHARED_OBJS += \
1173                 smb_inet.o \
1174                 smb_match.o \
1175                 smb_msgbuf.o \
1176                 smb_oem.o \
1177                 smb_string.o \
1178                 smb_utf8.o \
1179                 smb_door_legacy.o \
1180                 smb_xdr.o \
```

```
1181                    smb_token.o \
1182                    smb_token_xdr.o \
1183                    smb_sid.o \
1184                    smb_native.o \
1185                    smb_netbios_util.o

1187 SMBSRV_OBJS +=     $(SMBSRV_SHARED_OBJS)                   \
1188                    smb_acl.o                               \
1189                    smb_alloc.o                             \
1190                    smb_close.o                             \
1191                    smb_common_open.o                       \
1192                    smb_common_transact.o                   \
1193                    smb_create.o                            \
1194                    smb_delete.o                            \
1195                    smb_directory.o                         \
1196                    smb_dispatch.o                          \
1197                    smb_echo.o                              \
1198                    smb_fem.o                               \
1199                    smb_find.o                              \
1200                    smb_flush.o                             \
1201                    smb_fsinfo.o                            \
1202                    smb_fsops.o                             \
1203                    smb_init.o                              \
1204                    smb_kdoor.o                             \
1205                    smb_kshare.o                            \
1206                    smb_kutil.o                             \
1207                    smb_lock.o                              \
1208                    smb_lock_byte_range.o                   \
1209                    smb_locking_andx.o                      \
1210                    smb_logoff_andx.o                       \
1211                    smb_mangle_name.o                       \
1212                    smb_mbuf_marshaling.o                   \
1213                    smb_mbuf_util.o                         \
1214                    smb_negotiate.o                         \
1215                    smb_net.o                               \
1216                    smb_node.o                              \
1217                    smb_nt_cancel.o                         \
1218                    smb_nt_create_andx.o                    \
1219                    smb_nt_transact_create.o                \
1220                    smb_nt_transact_ioctl.o                 \
1221                    smb_nt_transact_notify_change.o         \
1222                    smb_nt_transact_quota.o                 \
1223                    smb_nt_transact_security.o              \
1224                    smb_odir.o                              \
1225                    smb_ofile.o                             \
1226                    smb_open_andx.o                         \
1227                    smb_opipe.o                             \
1228                    smb_oplock.o                            \
1229                    smb_pathname.o                          \
1230                    smb_print.o                             \
1231                    smb_process_exit.o                      \
1232                    smb_query_fileinfo.o                    \
1233                    smb_read.o                              \
1234                    smb_rename.o                            \
1235                    smb_sd.o                                \
1236                    smb_seek.o                              \
1237                    smb_server.o                            \
1238                    smb_session.o                           \
1239                    smb_session_setup_andx.o                \
1240                    smb_set_fileinfo.o                      \
1241                    smb_signing.o                           \
1242                    smb_tree.o                              \
1243                    smb_trans2_create_directory.o           \
1244                    smb_trans2_dfs.o                        \
1245                    smb_trans2_find.o                       \
1246                    smb_tree_connect.o                      \
```



```
1247                    smb_unlock_byte_range.o                 \
1248                    smb_user.o                              \
1249                    smb_vfs.o                               \
1250                    smb_vops.o                              \
1251                    smb_vss.o                               \
1252                    smb_write.o                             \
1253                    smb_write_raw.o

1255 PCFS_OBJS +=      pc_alloc.o      pc_dir.o        pc_node.o       pc_subr.o \
1256                    pc_vfsops.o     pc_vnops.o

1258 PROC_OBJS +=      prcontrol.o     prioctl.o       prsubr.o        prusrio.o \
1259                    prvfsops.o      prvnops.o

1261 MNTFS_OBJS +=     mntvfsops.o     mntvnops.o

1263 SHAREFS_OBJS +=   sharetab.o      sharefs_vfsops.o        sharefs_vnops.o

1265 SPEC_OBJS +=      specsubr.o      specvfsops.o    specvnops.o

1267 SOCK_OBJS +=      socksubr.o      sockvfsops.o    sockparams.o    \
1268                    socksyscalls.o  socktpi.o       sockstr.o \
1269                    sockcommon_vnops.o      sockcommon_subr.o \
1270                    sockcommon_sops.o       sockcommon.o    \
1271                    sock_notsupp.o  socknotify.o \
1272                    nl7c.o          nl7curi.o       nl7chttp.o      nl7clogd.o \
1273                    nl7cnca.o       sodirect.o      sockfilter.o

1275 TMPFS_OBJS +=     tmp_dir.o       tmp_subr.o      tmp_tnode.o     tmp_vfsops.o \
1276                    tmp_vnops.o

1278 UDFS_OBJS +=      udf_alloc.o     udf_bmap.o      udf_dir.o       \
1279                    udf_inode.o     udf_subr.o      udf_vfsops.o    \
1280                    udf_vnops.o

1282 UFS_OBJS +=       ufs_alloc.o     ufs_bmap.o      ufs_dir.o       ufs_xattr.o \
1283                    ufs_inode.o     ufs_subr.o      ufs_tables.o    ufs_vfsops.o \
1284                    ufs_vnops.o     quota.o         quotacalls.o    quota_ufs.o \
1285                    ufs_filio.o     ufs_lockfs.o    ufs_thread.o    ufs_trans.o \
1286                    ufs_acl.o       ufs_panic.o     ufs_directio.o  ufs_log.o \
1287                    ufs_extvnops.o  ufs_snap.o      lufs.o          lufs_thread.o \
1288                    lufs_log.o      lufs_map.o      lufs_top.o      lufs_debug.o
1289 VSCAN_OBJS +=     vscan_drv.o     vscan_svc.o vscan_door.o

1291 NSMB_OBJS +=      smb_conn.o      smb_dev.o       smb_iod.o       smb_pass.o \
1292                    smb_rq.o        smb_sign.o      smb_smb.o       smb_subrs.o \
1293                    smb_time.o      smb_tran.o      smb_trantcp.o   smb_usr.o \
1294                    subr_mchain.o

1296 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1297 SMBFS_OBJS +=     smbfs_vfsops.o  smbfs_vnops.o   smbfs_node.o    \
1298                    smbfs_acl.o     smbfs_client.o  smbfs_smb.o     \
1299                    smbfs_subr.o    smbfs_subr2.o   \
1300                    smbfs_rwlock.o  smbfs_xattr.o   \
1301                    $(SMBFS_COMMON_OBJS)


1304 #
1305 #                       LVM modules
1306 #
1307 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1308         md_med.o md_rename.o md_subr.o

1310 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o

1312 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o
```

```
1314 SOFTPART_OBJS += sp.o sp_ioctl.o

1316 STRIPE_OBJS += stripe.o stripe_ioctl.o

1318 HOTSPARES_OBJS += hotspares.o

1320 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o

1322 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o

1324 NOTIFY_OBJS += md_notify.o

1326 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o

1328 ZFS_COMMON_OBJS +=              \
1329         arc.o                   \
1330         bplist.o                \
1331         bpobj.o                 \
1332         bptree.o                \
1333         dbuf.o                  \
1334         ddt.o                   \
1335         ddt_zap.o               \
1336         dmu.o                   \
1337         dmu_diff.o              \
1338         dmu_send.o              \
1339         dmu_object.o            \
1340         dmu_objset.o            \
1341         dmu_traverse.o          \
1342         dmu_tx.o                \
1343         fits.o                  \
1344         fits_pass1.o            \
1345         fits_pass2.o            \
1346         fits_send.o             \
1347         fits_crc32c.o           \
1348         fits_count.o            \
1349 #endif /* ! codereview */
1350         dnode.o                 \
1351         dnode_sync.o            \
1352         dsl_dir.o               \
1353         dsl_dataset.o           \
1354         dsl_deadlist.o          \
1355         dsl_pool.o              \
1356         dsl_synctask.o          \
1357         dmu_zfetch.o            \
1358         dsl_deleg.o             \
1359         dsl_prop.o              \
1360         dsl_scan.o              \
1361         zfeature.o              \
1362         gzip.o                  \
1363         lzjb.o                  \
1364         metaslab.o              \
1365         refcount.o              \
1366         sa.o                    \
1367         sha256.o                \
1368         spa.o                   \
1369         spa_config.o            \
1370         spa_errlog.o            \
1371         spa_history.o           \
1372         spa_misc.o              \
1373         space_map.o             \
1374         txg.o                   \
1375         uberblock.o             \
1376         unique.o                \
1377         vdev.o                  \
1378         vdev_cache.o            \
```



```
1379         vdev_file.o             \
1380         vdev_label.o            \
1381         vdev_mirror.o           \
1382         vdev_missing.o          \
1383         vdev_queue.o            \
1384         vdev_raidz.o            \
1385         vdev_root.o             \
1386         zap.o                   \
1387         zap_leaf.o              \
1388         zap_micro.o             \
1389         zfs_byteswap.o          \
1390         zfs_debug.o             \
1391         zfs_fm.o                \
1392         zfs_fuid.o              \
1393         zfs_sa.o                \
1394         zfs_znode.o             \
1395         zil.o                   \
1396         zio.o                   \
1397         zio_checksum.o          \
1398         zio_compress.o          \
1399         zio_inject.o            \
1400         zle.o                   \
1401         zrlock.o

1403 ZFS_SHARED_OBJS +=              \
1404         zfeature_common.o       \
1405         zfs_comutil.o           \
1406         zfs_deleg.o             \
1407         zfs_fletcher.o          \
1408         zfs_namecheck.o         \
1409         zfs_prop.o              \
1410         zpool_prop.o            \
1411         zprop_common.o

1413 ZFS_OBJS +=                     \
1414         $(ZFS_COMMON_OBJS)      \
1415         $(ZFS_SHARED_OBJS)      \
1416         vdev_disk.o             \
1417         zfs_acl.o               \
1418         zfs_ctldir.o            \
1419         zfs_dir.o               \
1420         zfs_ioctl.o             \
1421         zfs_log.o               \
1422         zfs_onexit.o            \
1423         zfs_replay.o            \
1424         zfs_rlock.o             \
1425         rrwlock.o               \
1426         zfs_vfsops.o            \
1427         zfs_vnops.o             \
1428         zvol.o

1430 ZUT_OBJS +=                     \
1431         zut.o

1433 #
1434 #                       streams modules
1435 #
1436 BUFMOD_OBJS     +=      bufmod.o

1438 CONNLD_OBJS +=  connld.o

1440 DEDUMP_OBJS +=  dedump.o

1442 DRCOMPAT_OBJS +=        drcompat.o

1444 LDLINUX_OBJS += ldlinux.o
```

```
1446 LDTERM_OBJS +=  ldterm.o uwidth.o

1448 PCKT_OBJS +=    pckt.o

1450 PFMOD_OBJS +=   pfmod.o

1452 PTEM_OBJS +=    ptem.o

1454 REDIRMOD_OBJS += strredirm.o

1456 TIMOD_OBJS +=   timod.o

1458 TIRDWR_OBJS +=  tirdwr.o

1460 TTCOMPAT_OBJS +=ttcompat.o

1462 LOG_OBJS +=     log.o

1464 PIPEMOD_OBJS += pipemod.o

1466 RPCMOD_OBJS +=  rpcmod.o        clnt_cots.o     clnt_clts.o \
1467                 clnt_gen.o      clnt_perr.o     mt_rpcinit.o    rpc_calmsg.o \
1468                 rpc_prot.o      rpc_sztypes.o   rpc_subr.o      rpcb_prot.o \
1469                 svc.o           svc_clts.o      svc_gen.o       svc_cots.o \
1470                 rpcsys.o        xdr_sizeof.o    clnt_rdma.o     svc_rdma.o \
1471                 xdr_rdma.o      rdma_subr.o     xdrrdma_sizeof.o

1473 TLIMOD_OBJS +=  tlimod.o        t_kalloc.o      t_kbind.o       t_kclose.o \
1474                 t_kconnect.o    t_kfree.o       t_kgtstate.o    t_kopen.o \
1475                 t_krcvudat.o    t_ksndudat.o    t_kspoll.o      t_kunbind.o \
1476                 t_kutil.o

1478 RLMOD_OBJS += rlmod.o

1480 TELMOD_OBJS += telmod.o

1482 CRYPTMOD_OBJS += cryptmod.o

1484 KB_OBJS +=      kbd.o           keytables.o

1486 #
1487 #                       ID mapping module
1488 #
1489 IDMAP_OBJS +=   idmap_mod.o     idmap_kapi.o    idmap_xdr.o     idmap_cache.o

1491 #
1492 #                       scheduling class modules
1493 #
1494 SDC_OBJS +=             sysdc.o

1496 RT_OBJS +=              rt.o
1497 RT_DPTBL_OBJS +=        rt_dptbl.o

1499 TS_OBJS +=              ts.o
1500 TS_DPTBL_OBJS +=        ts_dptbl.o

1502 IA_OBJS +=              ia.o

1504 FSS_OBJS +=             fss.o

1506 FX_OBJS +=              fx.o
1507 FX_DPTBL_OBJS +=        fx_dptbl.o

1509 #
1510 #                       Inter-Process Communication (IPC) modules
```



```
1511 #
1512 IPC_OBJS +=     ipc.o

1514 IPCMSG_OBJS +=  msg.o

1516 IPCSEM_OBJS +=  sem.o

1518 IPCSHM_OBJS +=  shm.o

1520 #
1521 #                       bignum module
1522 #
1523 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1525 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1527 #
1528 #                       kernel cryptographic framework
1529 #
1530 KCF_OBJS +=     kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1531                 kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1532                 kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1533                 kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1534                 kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1535                 kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1536                 fips_random.o

1538 CRYPTOADM_OBJS += cryptoadm.o

1540 CRYPTO_OBJS +=  crypto.o

1542 DPROV_OBJS +=   dprov.o

1544 DCA_OBJS +=     dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1545                 dca_rsa.o

1547 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o

1549 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1551 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1553 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1554                 ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1555                 ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1556                 ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1557                 mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1558                 secitem.o ec2_test.o ecp_test.o

1560 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o

1562 SWRANDPROV_OBJS += swrand.o

1564 #
1565 #                       kernel SSL
1566 #
1567 KSSL_OBJS +=    kssl.o kssl_ioctl.o

1569 KSSL_SOCKFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1571 #
1572 #                       misc. modules
1573 #

1575 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \
1576                 audit_path.o audit_start.o audit_syscalls.o audit_token.o \
```

```
1577                audit_mem.o

1579 PCIC_OBJS +=    pcic.o

1581 RPCSEC_OBJS +=  secmod.o        sec_clnt.o      sec_svc.o       sec_gen.o \
1582                 auth_des.o      auth_kern.o     auth_none.o     auth_loopb.o\
1583                 authdesprt.o    authdesubr.o    authu_prot.o \
1584                 key_call.o      key_prot.o      svc_authu.o     svcauthdes.o

1586 RPCSEC_GSS_OBJS +=      rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1587                 rpcsec_gss_utils.o svc_rpcsec_gss.o

1589 CONSCONFIG_OBJS += consconfig.o

1591 CONSCONFIG_DACF_OBJS  += consconfig_dacf.o consplat.o

1593 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1595 KBTRANS_OBJS +=                        \
1596                 kbtrans.o               \
1597                 kbtrans_keytables.o     \
1598                 kbtrans_polled.o        \
1599                 kbtrans_streams.o       \
1600                 usb_keytables.o

1602 KGSSD_OBJS +=   gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1603                 gss_display_name.o gss_release_name.o gss_import_name.o \
1604                 gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1606 KGSSD_DERIVED_OBJS = gssd_xdr.o

1608 KGSS_DUMMY_OBJS += dmech.o

1610 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1612 CRYPTO= cksumtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1613         nfold.o verify_checksum.o prng.o block_size.o make_checksum.o\
1614         checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1616 # crypto/des
1617 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1619 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1621 CRYPTO_ARCFOUR= k5_arcfour.o

1623 # crypto/enc_provider
1624 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1626 # crypto/hash_provider
1627 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_ksha1.o

1629 # crypto/keyhash_provider
1630 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1632 # crypto/crc32
1633 CRYPTO_CRC32= crc32.o

1635 # crypto/old
1636 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1638 # crypto/raw
1639 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1641 K5_KRB= kfree.o copy_key.o \
1642         parse.o init_ctx.o \
```



```
1643         ser_adata.o ser_addr.o \
1644         ser_auth.o ser_cksum.o \
1645         ser_key.o ser_princ.o \
1646         serialize.o unparse.o \
1647         ser_actx.o

1649 K5_OS=  timeofday.o toffset.o \
1650         init_os_ctx.o c_ustime.o

1652 SEAL=
1653 # EXPORT DELETE START
1654 SEAL=   seal.o unseal.o
1655 # EXPORT DELETE END

1657 MECH=   delete_sec_context.o \
1658         import_sec_context.o \
1659         gssapi_krb5.o \
1660         k5seal.o k5unseal.o k5sealv3.o \
1661         ser_sctx.o \
1662         sign.o \
1663         util_crypt.o  \
1664         util_validate.o  util_ordering.o  \
1665         util_seqnum.o util_set.o util_seed.o \
1666         wrap_size_limit.o verify.o



1670 MECH_GEN= util_token.o


1673 KGSS_KRB5_OBJS += krb5mech.o \
1674         $(MECH) $(SEAL) $(MECH_GEN) \
1675         $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1676         $(CRYPTO_ENC) $(CRYPTO_HASH) \
1677         $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1678         $(CRYPTO_OLD) \
1679         $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1681 DES_OBJS +=     des_crypt.o des_impl.o des_ks.o des_soft.o

1683 DLBOOT_OBJS +=  bootparam_xdr.o nfs_dlinet.o scan.o

1685 KRTLD_OBJS +=   kobj_bootflags.o getoptstr.o \
1686                 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1688 MOD_OBJS +=     modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1690 STRPLUMB_OBJS += strplumb.o

1692 CPR_OBJS +=     cpr_driver.o cpr_dump.o \
1693                 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1694                 cpr_uthread.o

1696 PROF_OBJS +=    prf.o

1698 SE_OBJS += se_driver.o

1700 SYSACCT_OBJS += acct.o

1702 ACCTCTL_OBJS += acctctl.o

1704 EXACCTSYS_OBJS += exacctsys.o

1706 KAIO_OBJS += aio.o

1708 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o
```

```
1710 BUSRA_OBJS += busra.o

1712 PCS_OBJS += pcs.o

1714 PCAN_OBJS += pcan.o

1716 PCATA_OBJS += pcide.o pcdisk.o pclabel.o pcata.o

1718 PCSER_OBJS += pcser.o pcser_cis.o

1720 PCWL_OBJS += pcwl.o

1722 PSET_OBJS +=    pset.o

1724 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o

1726 UHCI_OBJS += uhci.o uhciutil.o uhcitgt.o uhcihub.o uhcipolled.o

1728 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o

1730 HUBD_OBJS += hubd.o

1732 USB_MID_OBJS += usb_mid.o

1734 USB_IA_OBJS += usb_ia.o

1736 UWBA_OBJS += uwba.o uwbai.o

1738 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o

1740 HWAHC_OBJS += hwahc.o hwahc_util.o

1742 WUSB_DF_OBJS += wusb_df.o
1743 WUSB_FWMOD_OBJS += wusb_fwmod.o

1745 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1746             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1747             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o

1749 IBD_OBJS +=     ibd.o ibd_cm.o

1751 EIBNX_OBJS +=   enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1752                 enx_misc.o enx_q.o enx_ctl.o

1754 EOIB_OBJS +=    eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1755                 eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1756                 eib_rsrc.o eib_svc.o eib_vnic.o

1758 DLPISTUB_OBJS += dlpistub.o

1760 SDP_OBJS +=     sdpddi.o

1762 TRILL_OBJS +=   trill.o

1764 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1765         ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o

1767 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o

1769 RPCIB_OBJS += rpcib.o

1771 KMDB_OBJS += kdrv.o

1773 AFE_OBJS += afe.o
```



```
1775 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1776                 bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o

1778 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o

1780 EFE_OBJS += efe.o

1782 ELXL_OBJS += elxl.o

1784 HME_OBJS += hme.o

1786 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1787                 ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o

1789 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1790                 nge_log.o nge_rx.o nge_tx.o nge_xmii.o

1792 PCN_OBJS += pcn.o

1794 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o

1796 URTW_OBJS += urtw.o

1798 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1799                 arn_main.o arn_recv.o arn_xmit.o arn_rc.o

1801 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o

1803 ATU_OBJS += atu.o

1805 IPW_OBJS += ipw2100_hw.o ipw2100.o

1807 IWI_OBJS += ipw2200_hw.o ipw2200.o

1809 IWH_OBJS += iwh.o

1811 IWK_OBJS += iwk2.o

1813 IWP_OBJS += iwp.o

1815 MWL_OBJS += mwl.o

1817 MWLFW_OBJS += mwlfw_mode.o

1819 WPI_OBJS += wpi.o

1821 RAL_OBJS += rt2560.o ral_rate.o

1823 RUM_OBJS += rum.o

1825 RWD_OBJS += rt2661.o

1827 RWN_OBJS += rt2860.o

1829 UATH_OBJS += uath.o

1831 UATHFW_OBJS += uathfw_mod.o

1833 URAL_OBJS += ural.o

1835 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o

1837 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o

1839 MXFE_OBJS += mxfe.o
```

```
1841 MPTSAS_OBJS += mptsas.o mptsas_impl.o mptsas_init.o mptsas_raid.o mptsas_smhba.o

1843 SFE_OBJS += sfe.o sfe_util.o

1845 BFE_OBJS += bfe.o

1847 BRIDGE_OBJS += bridge.o

1849 IDM_SHARED_OBJS += base64.o

1851 IDM_OBJS +=     $(IDM_SHARED_OBJS) \
1852                 idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o

1854 VR_OBJS += vr.o

1856 ATGE_OBJS += atge_main.o atge_l1e.o atge_mii.o atge_l1.o atge_l1c.o

1858 YGE_OBJS = yge.o

1860 #
1861 #       Build up defines and paths.
1862 #
1863 LINT_DEFS       += -Dunix

1865 #
1866 #       This duality can be removed when the native and target compilers
1867 #       are the same (or at least recognize the same command line syntax!)
1868 #       It is a bug in the current compilation system that the assember
1869 #       can't process the -Y I, flag.
1870 #
1871 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1872 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1873 INCLUDE_PATH    += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common

1875 PCIEB_OBJS += pcieb.o

1877 #       Chelsio N110 10G NIC driver module
1878 #
1879 CH_OBJS = ch.o glue.o pe.o sge.o

1881 CH_COM_OBJS =   ch_mac.o ch_subr.o cspi.o espi.o ixf1010.o mc3.o mc4.o mc5.o \
1882                 mv88e1xxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1883                 vsc7321.o vsc7326.o xpak.o

1885 #
1886 #       PCI strings file
1887 #
1888 PCI_STRING_OBJS = pci_strings.o

1890 NET_DACF_OBJS += net_dacf.o

1892 #
1893 #       Xframe 10G NIC driver module
1894 #
1895 XGE_OBJS = xge.o xgell.o

1897 XGE_HAL_OBJS =  xgehal-channel.o xgehal-fifo.o xgehal-ring.o  xgehal-config.o \
1898                 xgehal-driver.o  xgehal-mm.o xgehal-stats.o  xgehal-device.o \
1899                 xge-queue.o  xgehal-mgmt.o xgehal-mgmtaux.o

1901 #
1902 #       e1000g module
1903 #
1904 E1000G_OBJS +=  e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1905                 e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
1906                 e1000_mac.o e1000_manage.o e1000_nvm.o e1000_osdep.o \
```



```
1907                 e1000_phy.o e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1908                 e1000g_tx.o e1000g_rx.o e1000g_stat.o

1910 #
1911 #       Intel 82575 1G NIC driver module
1912 #
1913 IGB_OBJS =      igb_82575.o igb_api.o igb_mac.o igb_manage.o \
1914                 igb_nvm.o igb_osdep.o igb_phy.o igb_buf.o \
1915                 igb_debug.o igb_gld.o igb_log.o igb_main.o \
1916                 igb_rx.o igb_stat.o igb_tx.o

1918 #
1919 #       Intel Pro/100 NIC driver module
1920 #
1921 IPRB_OBJS =     iprb.o

1923 #
1924 #       Intel 10GbE PCIE NIC driver module
1925 #
1926 IXGBE_OBJS =    ixgbe_82598.o ixgbe_82599.o ixgbe_api.o         \
1927                 ixgbe_common.o ixgbe_phy.o                      \
1928                 ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o           \
1929                 ixgbe_log.o ixgbe_main.o                        \
1930                 ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o           \
1931                 ixgbe_tx.o  ixgbe_x540.o ixgbe_mbx.o

1933 #
1934 #       NIU 10G/1G driver module
1935 #
1936 NXGE_OBJS =     nxge_mac.o nxge_ipp.o nxge_rxdma.o               \
1937                 nxge_txdma.o nxge_txc.o nxge_main.o             \
1938                 nxge_hw.o nxge_fzc.o nxge_virtual.o             \
1939                 nxge_send.o nxge_classify.o nxge_fflp.o         \
1940                 nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o       \
1941                 nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o      \
1942                 nxge_hio.o nxge_hio_guest.o nxge_intr.o

1944 NXGE_NPI_OBJS = \
1945                 npi.o npi_mac.o npi_ipp.o                       \
1946                 npi_txdma.o npi_rxdma.o npi_txc.o               \
1947                 npi_zcp.o npi_espc.o npi_fflp.o                 \
1948                 npi_vir.o

1950 NXGE_HCALL_OBJS =       \
1951                 nxge_hcall.o

1953 #
1954 #       kiconv modules
1955 #
1956 KICONV_EMEA_OBJS += kiconv_emea.o

1958 KICONV_JA_OBJS += kiconv_ja.o

1960 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o

1962 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o

1964 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o

1966 #
1967 #       AAC module
1968 #
1969 AAC_OBJS = aac.o aac_ioctl.o

1971 #
1972 #       sdcard modules
```

```
#
SDA_OBJS =      sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
SDHOST_OBJS =   sdhost.o

#
#       hxge 10G driver module
#
HXGE_OBJS =     hxge_main.o hxge_vmac.o hxge_send.o             \
                hxge_txdma.o hxge_rxdma.o hxge_virtual.o        \
                hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o    \
                hxge_ndd.o hxge_pfc.o                           \
                hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o        \
                hpi_vir.o hpi_pfc.o

#
#       MEGARAID_SAS module
#
MEGA_SAS_OBJS = megaraid_sas.o

#
#       MR_SAS module
#
MR_SAS_OBJS = mr_sas.o

#
#       ISCSI_INITIATOR module
#
ISCSI_INITIATOR_OBJS =  chap.o iscsi_io.o iscsi_thread.o        \
                        iscsi_ioctl.o iscsid.o iscsi.o          \
                        iscsi_login.o isns_client.o iscsiAuthClient.o   \
                        iscsi_lun.o iscsiAuthClientGlue.o       \
                        iscsi_net.o nvfile.o iscsi_cmd.o        \
                        iscsi_queue.o persistent.o iscsi_conn.o \
                        iscsi_sess.o radius_auth.o iscsi_crc.o  \
                        iscsi_stats.o radius_packet.o iscsi_doorclt.o   \
                        iscsi_targetparam.o utils.o kifconf.o

#
#       ntxn 10Gb/1Gb NIC driver module
#
NTXN_OBJS =     unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
                        unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

#
#       Myricom 10Gb NIC driver module
#
MYRI10GE_OBJS = myri10ge.o myri10ge_lro.o

#       nulldriver module
#
NULLDRIVER_OBJS =       nulldriver.o

TPM_OBJS =      tpm.o tpm_hcall.o
```

```
**********************************************************
  117504 Wed Oct 17 21:48:38 2012
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

 683 int
 683 static int
 684 dsl_dataset_namelen(dsl_dataset_t *ds)
 685 {
 686         int result;

 688         if (ds == NULL) {
 689                 result = 3;     /* "mos" */
 690         } else {
 691                 result = dsl_dir_namelen(ds->ds_dir);
 692                 VERIFY(0 == dsl_dataset_get_snapname(ds));
 693                 if (ds->ds_snapname[0]) {
 694                         ++result;       /* adding one for the @-sign */
 695                         if (!MUTEX_HELD(&ds->ds_lock)) {
 696                                 mutex_enter(&ds->ds_lock);
 697                                 result += strlen(ds->ds_snapname);
 698                                 mutex_exit(&ds->ds_lock);
 699                         } else {
 700                                 result += strlen(ds->ds_snapname);
 701                         }
 702                 }
 703         }

 705         return (result);
 706 }
______unchanged_portion_omitted_
```

```
**********************************************************
   30408 Wed Oct 17 21:48:38 2012
new/usr/src/uts/common/fs/zfs/fits.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <sys/mkdev.h>
#include <sys/debug.h>
#include <sys/open.h>
#include <sys/zfs_ioctl.h>
#include <zfs_namecheck.h>
#include <sys/policy.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_prop.h>
#include <sys/zvol.h>
#include <sys/zap.h>
#include <sys/dsl_dataset.h>
#include <sys/dmu_traverse.h>
#include <sys/dsl_dir.h>
#include <sys/arc.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/sa.h>
#include <sys/sa_impl.h>
#include <sys/zfs_acl.h>
#include <sys/zfs_sa.h>
#include <sys/zfs_znode.h>
#include <sys/dbuf.h>
#include <sys/fits.h>
#include <sys/fits_impl.h>

/*
 * fits_send generates a stream of filesystem data analogous to dmu_send.
 * The main difference is that the fits-stream does not contain zfs-specific
 * data and can be replayed on any filesystem. It just contains commands like
 * MKDIR, CHMOD, RENAME etc.
 * The stream is generated in two passes. The first pass, PASS_LINK basically
 * creates all new files/directories and links, while the second pass,
```



```
 * PASS_UNLINK, does all the removal of old stuff.
 * Each pass enumerates all objects in inode order.
 * There are some corner cases:
 *   Files / directories can only be created if the parent already exists or
 * already has been created. If an object is encountered which parent does not
 * satisfy this condition, it is put back and its creation will be trigger
 * by the creation of the parent.
 *   A similar case applies on deletion. A directory can only be removed after
 * the last contained object has been removed. If a directory is not empty,
 * it is put back and the deletion of the last object in it triggers the
 * deletion.
 *   If an objects gets deleted, and a new object is created under the same
 * name, pass1 cannot create the object directly. So it is created under a
 * temporary name and gets renamed in pass2.
 *   If an object is deleted and a new object (of possibly diffent type)
 * created under the same inode and the same name, this change cannot be
 * detected by enumerating the containing directory (as name + inode are
 * unchanged). It is detected by a change of the inode generation number and
 * a flag is set for pass2. Creation is postponed. In pass2, all enumerated
 * directories are checked for this inode (although the entry is unchanged,
 * the directory has a bumped txg). If it is encountered, delete + create
 * happen both in pass2.
 *
 * There are lots of TODOs left:
 *  - add XATTR support
 *  - add path-caching
 *  - add a cache for brute-force parent search
 *  - add a cache for inode-search in a directory
 *  - use a hash instead of the linear list in fits_count
 */
static int
fits_dnode_changed(spa_t *spa, fits_t *f, uint64_t dnobj,
    dnode_phys_t *from, arc_buf_t *frombuf, dnode_phys_t *to, arc_buf_t *tobuf);

/* copied from zfs_znode.c */
static int
fits_sa_setup(objset_t *osp, sa_attr_type_t **sa_table)
{
        uint64_t sa_obj = 0;
        int error;

        error = zap_lookup(osp, MASTER_NODE_OBJ, ZFS_SA_ATTRS, 8, 1, &sa_obj);
        if (error != 0 && error != ENOENT)
                return (error);

        error = sa_setup(osp, sa_obj, zfs_attr_table, ZPL_END, sa_table);
        return (error);
}

static int
fits_grab_sa_handle(objset_t *osp, uint64_t obj, sa_handle_t **hdlp,
    dmu_buf_t **db, void *tag)
{
        dmu_object_info_t doi;
        int error;

        if ((error = sa_buf_hold(osp, obj, tag, db)) != 0)
                return (error);

        dmu_object_info_from_db(*db, &doi);
        if ((doi.doi_bonus_type != DMU_OT_SA &&
            doi.doi_bonus_type != DMU_OT_ZNODE) ||
            (doi.doi_bonus_type == DMU_OT_ZNODE &&
            doi.doi_bonus_size < sizeof (znode_phys_t))) {
                sa_buf_rele(*db, tag);
                return (ENOTSUP);
```

```
125         }
127         error = sa_handle_get(osp, obj, NULL, SA_HDL_PRIVATE, hdlp);
128         if (error != 0) {
129                 sa_buf_rele(*db, tag);
130                 return (error);
131         }
133         return (0);
134 }
136 static void
137 fits_release_sa_handle(sa_handle_t *hdl, dmu_buf_t *db, void *tag)
138 {
139         sa_handle_destroy(hdl);
140         sa_buf_rele(db, tag);
141 }
143 static int
144 fits_find_from_bp(spa_t *spa, dnode_phys_t *dnp, blklevel_t *bl,
145     const zbookmark_t *zb, blkptr_t **bpp, arc_buf_t **pbuf)
146 {
147         uint32_t flags;
148         int epbs = dnp->dn_indblkshift - SPA_BLKPTRSHIFT;
149         int epbmask = (1 << epbs) - 1;
150         int level;
151         int slot;
152         uint64_t blkid;
153         uint64_t blk;
154         blklevel_t *blp;
155         zbookmark_t czb;
156         int i;
158         *bpp = NULL;
159         for (level = dnp->dn_nlevels - 1; level >= zb->zb_level; --level) {
160                 blkid = zb->zb_blkid >> (epbs * (level - zb->zb_level));
161                 blk = blkid >> epbs;
162                 slot = blk & epbmask;
163                 blp = bl + level;
165                 if (blp->bl_blk == blk)
166                         continue;
168                 for (i = 0; i <= level; ++i) {
169                         blklevel_t *b = bl + i;
171                         if (b->bl_buf)
172                                 arc_buf_remove_ref(b->bl_buf, &b->bl_buf);
173                         b->bl_bp = NULL;
174                         b->bl_buf = NULL;
175                         b->bl_blk = -1;
176                 }
177                 ASSERT(slot < blp[1].bl_nslots);
178                 if (BP_IS_HOLE(blp[1].bl_bp + slot)) {
179                         *bpp = NULL;
180                         return (0);
181                 }
182                 /*
183                  * load indblk
184                  */
185                 flags = ARC_WAIT;
186                 SET_BOOKMARK(&czb, zb->zb_objset, zb->zb_object, level, blkid);
187                 if (dsl_read(NULL, spa, blp[1].bl_bp + slot, blp[1].bl_buf,
188                     arc_getbuf_func, &blp->bl_buf, ZIO_PRIORITY_ASYNC_READ,
189                     ZIO_FLAG_CANFAIL, &flags, &czb) != 0)
190                         return (EIO);
```



```
191                 blp->bl_bp = blp->bl_buf->b_data;
192                 blp->bl_nslots = 1 << epbs;
193                 blp->bl_blk = blk;
194         }
195         slot = zb->zb_blkid & epbmask;
196         blp = bl + zb->zb_level;
197         ASSERT(slot < blp->bl_nslots);
198         *bpp = blp->bl_bp + slot;
199         *pbuf = blp->bl_buf;
200         if (BP_IS_HOLE(*bpp))
201                 *bpp = NULL;
203         return (0);
204 }
206 static int
207 fits_file_cb(spa_t *spa, fits_t *f, zbookmark_t *zb,
208     blkptr_t *bp, arc_buf_t *pbuf, void *ctx)
209 {
210         int err = 0;
211         blkptr_t *fbp;
213         if (issig(JUSTLOOKING) && issig(FORREAL))
214                 return (EINTR);
216         if (f->f_fromds && zb->zb_objset == f->f_fromds->ds_object)
217                 return (0);
219         if (bp == NULL) {
220                 arc_buf_t *fpbuf = NULL;
221                 zbookmark_t czb;
223                 ASSERT(f->f_fromds);
224                 SET_BOOKMARK(&czb, f->f_fromds->ds_object, zb->zb_object,
225                     zb->zb_level, zb->zb_blkid);
226                 err = fits_find_from_bp(spa, f->f_dnp, f->f_filebl,
227                     &czb, &fbp, &fpbuf);
228                 if (err)
229                         return (err);
230                 if (fbp) {
231 #if 0
232                         /* XXX TODO callback for newly created hole */
233                         err = fits_enum_bp(spa, da, &czb, fbp, fpbuf);
234                         if (err)
235                                 return (err);
236 #endif
237                 }
238         } else if (zb->zb_level == 0) {
239                 arc_buf_t *tbuf;
240                 uint32_t tflags = ARC_WAIT;
241                 int blksz = BP_GET_LSIZE(bp);
243                 if (dsl_read(NULL, spa, bp, pbuf,
244                     arc_getbuf_func, &tbuf, ZIO_PRIORITY_ASYNC_READ,
245                     ZIO_FLAG_CANFAIL, &tflags, zb) != 0)
246                         return (EIO);
248                 if (f->f_ops->fits_file_data)
249                         err = f->f_ops->fits_file_data(ctx, tbuf->b_data,
250                             zb->zb_blkid * blksz, blksz);
252                 (void) arc_buf_remove_ref(tbuf, &tbuf);
253         }
254         return (err);
255 }
```

```
257 static int
258 fits_enum_bp(spa_t *spa, fits_t *da, zbookmark_t *zb,
259     blkptr_t *bp, arc_buf_t *pbuf, uint64_t min_txg, void *ctx)
260 {
261         int err = 0;
262         arc_buf_t *buf = NULL;
263         uint32_t flags = ARC_WAIT;
264
265         if (BP_IS_HOLE(bp))
266                 return (0);
267
268         if (bp->blk_birth <= min_txg)
269                 return (0);
270
271         if (BP_GET_LEVEL(bp) > 0) {
272                 int i;
273                 int epb = BP_GET_LSIZE(bp) >> SPA_BLKPTRSHIFT;
274                 blkptr_t *cbp;
275                 zbookmark_t czb;
276
277                 if (dsl_read(NULL, spa, bp, pbuf, arc_getbuf_func, &buf,
278                     ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL, &flags, zb) != 0)
279                         return (EIO);
280                 cbp = buf->b_data;
281                 for (i = 0; i < epb; ++i, ++cbp) {
282                         SET_BOOKMARK(&czb, zb->zb_objset, zb->zb_object,
283                             zb->zb_level - 1, zb->zb_blkid * epb + i);
284                         err = fits_enum_bp(spa, da, &czb, cbp, buf, min_txg,
285                             ctx);
286                         if (err)
287                                 goto out;
288                 }
289         } else if (BP_GET_TYPE(bp) == DMU_OT_DNODE) {
290                 int i;
291                 int epb = BP_GET_LSIZE(bp) >> DNODE_SHIFT;
292                 dnode_phys_t *dnp;
293
294                 if (dsl_read(NULL, spa, bp, pbuf, arc_getbuf_func, &buf,
295                     ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL,
296                     &flags, zb) != 0) {
297                         err = EIO;
298                         goto out;
299                 }
300                 dnp = buf->b_data;
301                 for (i = 0; i < epb; ++i, ++dnp) {
302                         uint64_t dnobj = zb->zb_blkid * epb + i;
303                         if (dnp->dn_type == DMU_OT_NONE)
304                                 continue;
305                         err = fits_dnode_changed(spa, da, dnobj, dnp, buf,
306                             NULL, NULL);
307                         if (err)
308                                 goto out;
309                 }
310         } else {
311                 err = fits_file_cb(spa, da, zb, bp, pbuf, ctx);
312         }
313 out:
314         if (buf)
315                 (void) arc_buf_remove_ref(buf, &buf);
316
317         return (err);
318 }
319
320 static int
321 fits_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp, arc_buf_t *pbuf,
322     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
```



```
323 {
324         int err = 0;
325         fits_t *f = arg;
326         blkptr_t *fbp = NULL;
327         zbookmark_t czb;
328
329         if (issig(JUSTLOOKING) && issig(FORREAL))
330                 return (EINTR);
331
332         if (f->f_fromds)
333                 SET_BOOKMARK(&czb, f->f_fromds->ds_object, zb->zb_object,
334                     zb->zb_level, zb->zb_blkid);
335
336         if (zb->zb_object != DMU_META_DNODE_OBJECT)
337                 return (0);
338
339         if (bp == NULL) {
340                 arc_buf_t *fpbuf = NULL;
341
342                 if (!f->f_fromds)
343                         return (0);
344
345                 err = fits_find_from_bp(spa, f->f_dnp, f->f_bl,
346                     &czb, &fbp, &fpbuf);
347                 if (err)
348                         return (EIO);
349                 if (fbp) {
350                         err = fits_enum_bp(spa, f, &czb, fbp, fpbuf, 0, NULL);
351                         if (err)
352                                 return (EIO);
353                 }
354                 return (0);
355         } else if (zb->zb_level == 0) {
356                 dnode_phys_t *tblk;
357                 dnode_phys_t *fblk = NULL;
358                 arc_buf_t *tbuf;
359                 arc_buf_t *fbuf = NULL;
360                 arc_buf_t *fpbuf = NULL;
361                 uint32_t fflags = ARC_WAIT;
362                 uint32_t tflags = ARC_WAIT;
363                 int blksz = BP_GET_LSIZE(bp);
364                 int i;
365
366                 if (dsl_read(NULL, spa, bp, pbuf,
367                     arc_getbuf_func, &tbuf, ZIO_PRIORITY_ASYNC_READ,
368                     ZIO_FLAG_CANFAIL, &tflags, zb) != 0)
369                         return (EIO);
370                 tblk = tbuf->b_data;
371
372                 if (f->f_fromds) {
373                         err = fits_find_from_bp(spa, f->f_dnp, f->f_bl, zb,
374                             &fbp, &fpbuf);
375                         if (err)
376                                 return (EIO);
377                 }
378                 if (fbp) {
379                         if (dsl_read(NULL, spa, fbp, fpbuf,
380                             arc_getbuf_func, &fbuf, ZIO_PRIORITY_ASYNC_READ,
381                             ZIO_FLAG_CANFAIL, &fflags, &czb) != 0) {
382                                 (void) arc_buf_remove_ref(tbuf, &tbuf);
383                                 return (EIO);
384                         }
385                         fblk = fbuf->b_data;
386                         if (blksz != BP_GET_LSIZE(fbp))
387                                 return (EIO);
388                 }
```

```
389                 for (i = 0; i < blksz >> DNODE_SHIFT; i++) {
390                         uint64_t dnobj = (zb->zb_blkid <<
391                             (DNODE_BLOCK_SHIFT - DNODE_SHIFT)) + i;
392                         err = 0;
393                         if (fbuf && (tblk[i].dn_type == DMU_OT_NONE) &&
394                             fblk[i].dn_type != DMU_OT_NONE) {
395                                 err = fits_dnode_changed(spa, f, dnobj,
396                                     fblk + i, fbuf, NULL, NULL);
397                         } else if (fbuf) {
398                                 if (memcmp(tblk + i, fblk + i, sizeof (*tblk)))
399                                         err = fits_dnode_changed(spa, f,
400                                             dnobj, fblk + i, fbuf, tblk + i,
401                                             tbuf);
402                         } else {
403                                 if (tblk[i].dn_type != DMU_OT_NONE)
404                                         err = fits_dnode_changed(spa, f,
405                                             dnobj, NULL, NULL, tblk + i, tbuf);
406                         }
407                         if (err)
408                                 break;
409                 }
410                 (void) arc_buf_remove_ref(tbuf, &tbuf);
411                 if (fbuf)
412                         (void) arc_buf_remove_ref(fbuf, &fbuf);

414                 if (err)
415                         return (EIO);
416                 /* Don't care about the data blocks */
417                 return (TRAVERSE_VISIT_NO_CHILDREN);
418         }
419         return (0);
420 }

422 #define DIR_FROM        1
423 #define DIR_TO          2
424 static int
425 fits_diff_dir(fits_t *f, uint64_t dnobj, int dir, void *ctx)
426 {
427         zap_cursor_t zc;
428         zap_attribute_t *za;
429         int err;
430         objset_t *os1;
431         objset_t *os2;
432         uint64_t mask =  ZFS_DIRENT_OBJ(-1ULL);
433         uint64_t num;
434         uint64_t ix = 0;

436         if (dir == DIR_FROM) {
437                 os1 = f->f_fromsnap;
438                 os2 = f->f_tosnap;
439         } else {
440                 os1 = f->f_tosnap;
441                 os2 = f->f_fromsnap;
442         }

444         za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
445         for (zap_cursor_init(&zc, os1, dnobj);
446             (err = zap_cursor_retrieve(&zc, za)) == 0;
447             zap_cursor_advance(&zc), ++ix) {
448                 err = zap_lookup(os2, dnobj, za->za_name, sizeof (num), 1,
449                     &num);
450                 if (err && err != ENOENT)
451                         break;
452                 if (err == ENOENT) {
453                         if (dir == DIR_FROM) {
454                                 if (f->f_ops->fits_dirent_del) {
```



```
455                                         err = f->f_ops->fits_dirent_del(ctx,
456                                             za->za_name,
457                                             za->za_first_integer & mask);
458                                         if (err)
459                                                 goto out;
460                                 }
461                         } else {
462                                 if (f->f_ops->fits_dirent_add) {
463                                         err = f->f_ops->fits_dirent_add(ctx,
464                                             za->za_name,
465                                             za->za_first_integer & mask);
466                                         if (err)
467                                                 goto out;
468                                 }
469                         }
470                 } else if ((za->za_first_integer & mask) != (num & mask)) {
471                         if (dir == DIR_TO) {
472                                 /* report only once */
473                                 if (f->f_ops->fits_dirent_mod) {
474                                         err = f->f_ops->fits_dirent_mod(ctx,
475                                             za->za_name, num & mask,
476                                             za->za_first_integer & mask);
477                                         if (err)
478                                                 goto out;
479                                 }
480                         }
481                 } else {
482                         if (dir == DIR_TO) {
483                                 /* report only once */
484                                 if (f->f_ops->fits_dirent_unmod) {
485                                         err = f->f_ops->fits_dirent_unmod(ctx,
486                                             za->za_name, num & mask);
487                                         if (err)
488                                                 goto out;
489                                 }
490                         }
491                 }
492         }
493         err = 0;
494 out:
495         zap_cursor_fini(&zc);
496         kmem_free(za, sizeof (zap_attribute_t));

498         return (err);
499 }
500 static int
501 fits_enum_dir(fits_t *f, uint64_t dnobj, int dir, void *ctx)
502 {
503         zap_cursor_t zc;
504         zap_attribute_t *za;
505         int err;
506         objset_t *os;
507         uint64_t mask =  ZFS_DIRENT_OBJ(-1ULL);

509         if (dir == DIR_FROM)
510                 os = f->f_fromsnap;
511         else
512                 os = f->f_tosnap;

514         za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
515         for (zap_cursor_init(&zc, os, dnobj);
516             (err = zap_cursor_retrieve(&zc, za)) == 0;
517             zap_cursor_advance(&zc)) {
518                 if (dir == DIR_FROM) {
519                         if (f->f_ops->fits_dirent_del) {
520                                 err = f->f_ops->fits_dirent_del(ctx,
```

```
521                                     za->za_name, za->za_first_integer & mask);
522                             if (err)
523                                     break;
524                     }
525             } else {
526                     if (f->f_ops->fits_dirent_add) {
527                             err = f->f_ops->fits_dirent_add(ctx,
528                                 za->za_name, za->za_first_integer & mask);
529                             if (err)
530                                     break;
531                     }
532             }
533     }
534     if (err == ENOENT)
535             err = 0;

537     zap_cursor_fini(&zc);
538     kmem_free(za, sizeof (zap_attribute_t));

540     return (err);
541 }

543 static int
544 fits_dnode_changed(spa_t *spa, fits_t *f, uint64_t dnobj,
545     dnode_phys_t *from, arc_buf_t *frombuf, dnode_phys_t *to, arc_buf_t *tobuf)
546 {
547     int err = 0;
548     int type = 0;
549     fits_info_t si;

551     if (dnobj == f->f_shares_dir)
552             return (0);

554     if (to && to->dn_type != DMU_OT_PLAIN_FILE_CONTENTS &&
555         to->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
556             to = NULL;
557     }
558     if (from && from->dn_type != DMU_OT_PLAIN_FILE_CONTENTS &&
559         from->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
560             from = NULL;
561     }

563     if (from) {
564             err = fits_get_info(f, dnobj, FITS_OLD, &si, FI_ATTR_LINKS);
565             if (err)
566                     return (err);
567             if (si.si_nlinks == 0)
568                     from = NULL;
569     }
570     if (to) {
571             err = fits_get_info(f, dnobj, FITS_NEW, &si, FI_ATTR_LINKS);
572             if (err)
573                     return (err);
574             if (si.si_nlinks == 0)
575                     to = NULL;
576     }

578     if (!to && !from)
579             return (0);

581     if (from) {
582             if (from->dn_bonustype != DMU_OT_SA &&
583                 from->dn_bonustype != DMU_OT_ZNODE)
584                     return (EINVAL);
585     }
586     if (to) {
```



```
587             if (to->dn_bonustype != DMU_OT_SA &&
588                 to->dn_bonustype != DMU_OT_ZNODE)
589                     return (EINVAL);
590     }

592     if (from)
593             type = from->dn_type;
594     else if (to)
595             type = to->dn_type;

597     err = 0;
598     if (type == DMU_OT_DIRECTORY_CONTENTS) {
599             if (from && to) {
600                     if (f->f_ops->fits_dir_mod)
601                             err = f->f_ops->fits_dir_mod(f, dnobj);
602             } else if (from) {
603                     if (f->f_ops->fits_dir_del)
604                             err = f->f_ops->fits_dir_del(f, dnobj);
605             } else if (to) {
606                     if (f->f_ops->fits_dir_add)
607                             err = f->f_ops->fits_dir_add(f, dnobj);
608             }
609     } else if (type == DMU_OT_PLAIN_FILE_CONTENTS) {
610             if (from && to) {
611                     if (f->f_ops->fits_file_mod)
612                             err = f->f_ops->fits_file_mod(f, dnobj);
613             } else if (from) {
614                     if (f->f_ops->fits_file_del)
615                             err = f->f_ops->fits_file_del(f, dnobj);
616             } else if (to) {
617                     if (f->f_ops->fits_file_add)
618                             err = f->f_ops->fits_file_add(f, dnobj);
619             }
620     } else {
621             /* TODO other types, symlinks? */
622             err = 0;
623     }
624     return (err);
625 }

627 typedef struct _fits_search {
628     fits_t *zs_f;
629     uint64_t zs_dnobj;
630     uint64_t zs_parent;
631     objset_t *zs_osp;
632 } fits_search_t;

634 static int
635 search_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp, arc_buf_t *pbuf,
636     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
637 {
638     fits_search_t *zs = arg;
639     fits_t *f = zs->zs_f;
640     arc_buf_t *buf;
641     uint32_t flags = ARC_WAIT;
642     int ebp;
643     int i;
644     int ret;

646     if (issig(JUSTLOOKING) && issig(FORREAL))
647             return (EINTR);

649     if (zb->zb_object != DMU_META_DNODE_OBJECT)
650             return (0);

652     if (zb->zb_level != 0)
```

```
653                 return (0);

655         if (!bp || BP_IS_HOLE(bp))
656                 return (0);

658         if (BP_GET_TYPE(bp) != DMU_OT_DNODE)
659                 return (0);

661         ebp = BP_GET_LSIZE(bp) >> DNODE_SHIFT;

663         if (dsl_read(NULL, spa, bp, pbuf,
664             arc_getbuf_func, &buf, ZIO_PRIORITY_ASYNC_READ,
665             ZIO_FLAG_CANFAIL, &flags, zb) != 0)
666                 return (EIO);
667         dnp = buf->b_data;

669         for (i = 0; i < ebp; ++i) {
670                 zap_cursor_t zc;
671                 zap_attribute_t *za;
672                 uint64_t mask = ZFS_DIRENT_OBJ(-1ULL);
673                 uint64_t ix = 0;
674                 uint64_t dnobj = (zb->zb_blkid <<
675                         (DNODE_BLOCK_SHIFT - DNODE_SHIFT)) + i;

677                 if (dnp[i].dn_type != DMU_OT_DIRECTORY_CONTENTS)
678                         continue;
679                 if (dnobj == f->f_shares_dir)
680                         continue;

682                 za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
683                 for (zap_cursor_init(&zc, zs->zs_osp, dnobj);
684                     (ret = zap_cursor_retrieve(&zc, za)) == 0;
685                     zap_cursor_advance(&zc), ++ix) {
686                         if ((za->za_first_integer & mask) ==
687                             (zs->zs_dnobj & mask)) {
688                                 zs->zs_parent = dnobj;
689                                 break;
690                         }
691                 }
692                 zap_cursor_fini(&zc);
693                 kmem_free(za, sizeof (zap_attribute_t));
694         }

696         (void) arc_buf_remove_ref(buf, &buf);

698         if (zs->zs_parent)
699                 return (EIO);   /* abort search */

701         return (TRAVERSE_VISIT_NO_CHILDREN);
702 }

704 static int
705 fits_search_parent(fits_t *f, uint64_t dnobj, fits_which_t which,
706     uint64_t *parent)
707 {
708         dsl_dataset_t *ds;
709         fits_search_t zs;
710         int ret;

712         if (which == FITS_OLD) {
713                 ds = f->f_fromds;
714                 zs.zs_osp = f->f_fromsnap;
715         } else {
716                 ds = f->f_tods;
717                 zs.zs_osp = f->f_tosnap;
718         }
```



```
720         zs.zs_f = f;
721         zs.zs_dnobj = dnobj;
722         zs.zs_parent = 0;
723         ret = traverse_dataset(ds, 0, TRAVERSE_PRE, search_cb, &zs);
724         if (zs.zs_parent) {
725                 *parent = zs.zs_parent;
726                 return (0);
727         }

729         return (ret ? ret : ENOENT);
730 }

732 int
733 fits_get_info(fits_t *f, uint64_t dnobj, fits_which_t which,
734     fits_info_t *sp, uint64_t flags)
735 {
736         int ret;
737         sa_handle_t *hdl = NULL;
738         dmu_buf_t *db;
739         objset_t *osp;
740         sa_bulk_attr_t bulk[13];
741         int count = 0;
742         sa_attr_type_t *sa_table;

744         if (which == FITS_OLD) {
745                 osp = f->f_fromsnap;
746                 if (!osp)
747                         return (ENOENT);
748                 sa_table = f->f_from_sa_table;
749         } else if (which == FITS_NEW) {
750                 osp = f->f_tosnap;
751                 sa_table = f->f_to_sa_table;
752         } else {
753                 return (EINVAL);
754         }

756         ret = fits_grab_sa_handle(osp, dnobj, &hdl, &db, FTAG);
757         if (ret)
758                 return (ret);

760         if (flags & FI_ATTR_ATIME) {
761                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_ATIME], NULL,
762                     &sp->si_atime, sizeof (sp->si_atime));
763         }
764         if (flags & FI_ATTR_MTIME) {
765                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_MTIME], NULL,
766                     &sp->si_mtime, sizeof (sp->si_mtime));
767         }
768         if (flags & FI_ATTR_CTIME) {
769                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_CTIME], NULL,
770                     &sp->si_ctime, sizeof (sp->si_ctime));
771         }
772         if (flags & FI_ATTR_OTIME) {
773                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_CRTIME], NULL,
774                     &sp->si_otime, sizeof (sp->si_otime));
775         }
776         if (flags & FI_ATTR_MODE) {
777                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_MODE], NULL,
778                     &sp->si_mode, sizeof (sp->si_mode));
779         }
780         if (flags & FI_ATTR_SIZE) {
781                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_SIZE], NULL,
782                     &sp->si_size, sizeof (sp->si_size));
783         }
784         if (flags & FI_ATTR_PARENT) {
```

```
785                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_PARENT], NULL,
786                     &sp->si_parent, sizeof (sp->si_parent));
787         }
788         if (flags & FI_ATTR_LINKS) {
789                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_LINKS], NULL,
790                     &sp->si_nlinks, sizeof (sp->si_nlinks));
791         }
792         if (flags & FI_ATTR_RDEV) {
793                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_RDEV], NULL,
794                     &sp->si_rdev, sizeof (sp->si_rdev));
795         }
796         if (flags & FI_ATTR_UID) {
797                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_UID], NULL,
798                     &sp->si_uid, sizeof (sp->si_uid));
799         }
800         if (flags & FI_ATTR_GID) {
801                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_GID], NULL,
802                     &sp->si_gid, sizeof (sp->si_gid));
803         }
804         if (flags & FI_ATTR_GEN) {
805                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_GEN], NULL,
806                     &sp->si_gen, sizeof (sp->si_gen));
807         }
808         /* XXX if you add things, also bump the size of bulk */
809         /* XXX XATTR */

811         /* XXX TODO get flags to check for xattrdir */
812         if (count) {
813                 ret = sa_bulk_lookup(hdl, bulk, count);
814                 if (ret)
815                         goto out;
816         }

818         if ((flags & FI_ATTR_PARENT) && sp->si_parent != dnobj) {
819                 fits_info_t si;
820                 int good = 0;
821                 /*
822                  * verify parent. this is very expensive and only a workaround
823                  */
824                 ret = fits_get_info(f, sp->si_parent, which, &si, FI_ATTR_MODE);
825                 if (ret && ret != ENOENT)
826                         goto out;
827                 if (ret == 0 && S_ISDIR(si.si_mode)) {
828                         ret = fits_find_entry(f, sp->si_parent, dnobj, which,
829                             NULL);
830                         if (ret && ret != ENOENT)
831                                 goto out;
832                         if (ret == 0)
833                                 good = 1;
834                 }
835                 if (!good) {
836                         uint64_t parent;

838                         cmn_err(CE_NOTE, "parent wrong, do a brute force "
839                             "search for ino %"PRIu64"\n", dnobj);
840                         ret = fits_search_parent(f, dnobj, which, &parent);
841                         if (ret == ENOENT) {
842                                 cmn_err(CE_NOTE, "no parent found\n");
843                                 ret = EINVAL;
844                                 goto out;
845                         }
846                         if (ret)
847                                 goto out;
848                         sp->si_parent = parent;
849                         cmn_err(CE_NOTE, "parent found, use %"PRIu64"\n",
850                             parent);
```



```
851                         /*
852                          * TODO add a bad parent cache to prevent additional
853                          * lookup in pass 2
854                          */
855                 }
856         }

858 out:
859         fits_release_sa_handle(hdl, db, FTAG);
860         return (ret);
861 }

863 int
864 fits_file_contents(fits_t *f, uint64_t dnobj, void *ctx)
865 {
866         dnode_t *from = NULL;
867         dnode_t *to = NULL;
868         int err;
869         int i;
870         zbookmark_t czb;
871         spa_t *spa = f->f_tods->ds_dir->dd_pool->dp_spa;

873         if (f->f_fromds) {
874                 err = dnode_hold(f->f_fromsnap, dnobj, FTAG, &from);
875                 if (err && err != ENOENT)
876                         return (err);
877         }
878         if (from && from->dn_type != DMU_OT_PLAIN_FILE_CONTENTS) {
879                 dnode_rele(from, FTAG);
880                 from = NULL;
881         }
882         err = dnode_hold(f->f_tosnap, dnobj, FTAG, &to);
883         if (err)
884                 goto out;
885         if (to->dn_type != DMU_OT_PLAIN_FILE_CONTENTS) {
886                 err = EINVAL;
887                 goto out;
888         }
889         if (from) {
890                 f->f_filebl = kmem_zalloc(sizeof (blklevel_t)*from->dn_nlevels,
891                                           KM_SLEEP);
892                 for (i = 0; i < from->dn_nlevels; ++i)
893                         f->f_filebl[i].bl_blk = -1;
894                 i = from->dn_nlevels - 1;
895                 f->f_filebl[i].bl_nslots = from->dn_nblkptr;
896                 f->f_filebl[i].bl_bp = &from->dn_phys->dn_blkptr[0];
897                 f->f_filebl[i].bl_blk = 0;
898                 f->f_filebl[i].bl_buf = from->dn_dbuf->db_parent->db_buf;
899         }
900         for (i = 0; i < to->dn_nblkptr; ++i) {
901                 SET_BOOKMARK(&czb, f->f_tods->ds_object, dnobj,
902                     to->dn_nlevels - 1, i);
903                 err = fits_enum_bp(spa, f, &czb, to->dn_phys->dn_blkptr + i,
904                     NULL, f->f_fromtxg, ctx);
905                 if (err)
906                         goto out;
907         }
908 out:
909         if (f->f_filebl) {
910                 kmem_free(f->f_filebl, sizeof (blklevel_t) * from->dn_nlevels);
911                 f->f_filebl = NULL;
912         }
913         if (from)
914                 dnode_rele(from, FTAG);
915         if (to)
916                 dnode_rele(to, FTAG);
```

```
        return (err);
}

int
fits_dir_contents(fits_t *f, uint64_t dnobj, void *ctx)
{
        dnode_t *from = NULL;
        dnode_t *to = NULL;
        int err;

        if (f->f_fromds) {
                err = dnode_hold(f->f_fromsnap, dnobj, FTAG, &from);
                if (err && err != ENOENT)
                        return (err);
        }
        if (from && from->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
                dnode_rele(from, FTAG);
                from = NULL;
        }
        err = dnode_hold(f->f_tosnap, dnobj, FTAG, &to);
        if (err && err != ENOENT)
                return (err);
        if (to && to->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
                dnode_rele(to, FTAG);
                to = NULL;
        }

        if (to && from) {
                err = fits_diff_dir(f, dnobj, DIR_TO, ctx);
                if (err)
                        goto out;
                err = fits_diff_dir(f, dnobj, DIR_FROM, ctx);
        } else if (to) {
                err = fits_enum_dir(f, dnobj, DIR_TO, ctx);
        } else if (from) {
                err = fits_enum_dir(f, dnobj, DIR_FROM, ctx);
        }
out:
        if (from)
                dnode_rele(from, FTAG);
        if (to)
                dnode_rele(to, FTAG);

        return (err);
}

int
fits_find_entry(fits_t *f, uint64_t dirobj, uint64_t dnobj,
    fits_which_t which, char **name)
{
        zap_cursor_t zc;
        zap_attribute_t *za;
        int err;
        uint64_t mask = ZFS_DIRENT_OBJ(-1ULL);
        struct objset *os;

        if (which == FITS_OLD) {
                os = f->f_fromsnap;
                if (!os)
                        return (ENOENT);
        } else if (which == FITS_NEW) {
                os = f->f_tosnap;
        } else {
                return (EINVAL);
        }
```



```
        if (name)
                *name = NULL;
        za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
        for (zap_cursor_init(&zc, os, dirobj);
            (err = zap_cursor_retrieve(&zc, za)) == 0;
            zap_cursor_advance(&zc)) {
                if ((za->za_first_integer & mask) == (dnobj & mask)) {
                        if (name)
                                *name = za->za_name;
                        break;
                }
        }
        zap_cursor_fini(&zc);
        return (err);
}

void
fits_free_name(char *name)
{
        zap_attribute_t *za;

        if (!name)
                return;

        za = (zap_attribute_t *)(name - offsetof(zap_attribute_t, za_name));
        kmem_free(za, sizeof (*za));
}

int
fits_lookup_entry(fits_t *f, uint64_t dirobj, char *name,
    fits_which_t which, uint64_t *dnobj)
{
        struct objset *osp;
        int ret;

        if (which == FITS_OLD) {
                osp = f->f_fromsnap;
                if (!osp)
                        return (ENOENT);
        } else if (which == FITS_NEW) {
                osp = f->f_tosnap;
        } else {
                return (EINVAL);
        }

        ret =  zap_lookup(osp, dirobj, name, sizeof (*dnobj), 1, dnobj);
        if (ret)
                return (ret);
        *dnobj = ZFS_DIRENT_OBJ(*dnobj);

        return (0);
}

int
fits_write(fits_t *f, const uint8_t *data, int len)
{
        ssize_t resid; /* have to get resid to get detailed errno */
        int err;

        err = vn_rdwr(UIO_WRITE, f->f_vp, (caddr_t)data,
            len, 0, UIO_SYSSPACE, FAPPEND, RLIM64_INFINITY, CRED(), &resid);
        *f->f_offp += len;

        return (err);
```

```
}

int
fits_get_uuid(fits_t *f, fits_which_t which, uint8_t data[16])
{
        if (which == FITS_OLD && !f->f_fromds)
                return (ENOENT);

        LE_OUT64(data, f->f_tods->ds_dir->dd_pool->dp_spa->spa_config_guid);
        if (which == FITS_OLD) {
                LE_OUT64(data + 8, f->f_fromds->ds_phys->ds_guid);
        } else {
                LE_OUT64(data + 8, f->f_tods->ds_phys->ds_guid);
        }
        return (0);
}

int
fits_get_ctransid(fits_t *f, fits_which_t which, uint64_t *ctransid)
{
        if (which == FITS_OLD && !f->f_fromds)
                return (ENOENT);

        if (which == FITS_OLD)
                *ctransid = f->f_fromds->ds_phys->ds_creation_txg;
        else
                *ctransid = f->f_tods->ds_phys->ds_creation_txg;
        return (0);
}

int
fits_get_snapname(fits_t *f, fits_which_t which,
    char **name, int *len)
{
        dsl_dataset_t *ds;

        if (which == FITS_OLD && !f->f_fromds)
                return (ENOENT);

        if (which == FITS_OLD)
                ds = f->f_fromds;
        else
                ds = f->f_tods;

        *len = dsl_dataset_namelen(ds) + 1;
        *name = kmem_alloc(*len, KM_SLEEP);
        dsl_dataset_name(ds, *name);
        return (0);
}

int
fits_read_symlink(fits_t *f, uint64_t dnobj, fits_which_t which,
    char **target, int *plen)
{
        int err;
        int ret;
        sa_handle_t *hdl = NULL;
        dmu_buf_t *db;
        objset_t *osp;
        dmu_object_info_t doi;
        sa_attr_type_t *sa_table;

        if (which == FITS_OLD) {
                osp = f->f_fromsnap;
                if (!osp)
                        return (EINVAL);
```



```
                sa_table = f->f_from_sa_table;
        } else if (which == FITS_NEW) {
                osp = f->f_tosnap;
                sa_table = f->f_to_sa_table;
        } else {
                return (EINVAL);
        }

        err = fits_grab_sa_handle(osp, dnobj, &hdl, &db, FTAG);
        if (err)
                return (err);

        dmu_object_info_from_db(db, &doi);
        if (doi.doi_bonus_type == DMU_OT_SA) {
                int len;

                ret = sa_size(hdl, sa_table[ZPL_SYMLINK], &len);
                if (ret)
                        goto out;
                *target = kmem_alloc(len + 1, KM_SLEEP);
                *plen = len;
                (*target)[len] = 0;
                ret = sa_lookup(hdl, sa_table[ZPL_SYMLINK], *target, len + 1);
                if (ret)
                        kmem_free(*target, len + 1);
        } else {
                /*
                 * TODO read target from file data, the old way
                 * see zfs_readlink
                 */
                ret = EINVAL;
        }

out:
        fits_release_sa_handle(hdl, db, FTAG);

        return (ret);
}

int
fits_send(objset_t *tosnap, objset_t *fromsnap, int outfd, vnode_t *vp,
    offset_t *off)
{
        dsl_dataset_t *ds;
        dsl_dataset_t *fromds = NULL;
        int err = 0;
        fits_t f;
        arc_buf_t *buf = NULL;
        uint32_t flags;
        objset_phys_t *osp;
        int i;
        zbookmark_t zb;

        memset(&f, 0, sizeof (f));
        ds = tosnap->os_dsl_dataset;
        if (fromsnap)
                fromds = fromsnap->os_dsl_dataset;

        /* make certain we are looking at snapshots */
        if (!dsl_dataset_is_snapshot(ds) ||
            (fromds && !dsl_dataset_is_snapshot(fromds)))
                return (EINVAL);

        /* fromsnap must be earlier and from the same lineage as tosnap */
        if (fromds) {
                if (fromds->ds_phys->ds_creation_txg >=
```

```
1181                    ds->ds_phys->ds_creation_txg)
1182                        return (EXDEV);
1183
1184                if (fromds->ds_dir != ds->ds_dir)
1185                        return (EXDEV);
1186
1187                /*
1188                 * read root dnode from from-dataset
1189                 */
1190                flags = ARC_WAIT;
1191                SET_BOOKMARK(&zb, fromds->ds_object, ZB_ROOT_OBJECT,
1192                    ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1193                err = dsl_read_nolock(NULL, fromds->ds_dir->dd_pool->dp_spa,
1194                    &fromds->ds_phys->ds_bp, arc_getbuf_func, &buf,
1195                    ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL, &flags, &zb);
1196                if (err)
1197                        return (err);
1198                osp = buf->b_data;
1199                f.f_dnp = &osp->os_meta_dnode;
1200        }
1201        f.f_vp = vp;
1202        f.f_offp = off;
1203        f.f_err = 0;
1204        f.f_fromds = fromds;
1205        f.f_tods = ds;
1206        f.f_fromsnap = fromsnap;
1207        f.f_tosnap = tosnap;
1208        if (fromds) {
1209                f.f_fromtxg = fromds->ds_phys->ds_creation_txg;
1210                f.f_bl = kmem_zalloc(sizeof (blklevel_t) *
1211                    f.f_dnp->dn_nlevels, KM_SLEEP);
1212                for (i = 0; i < f.f_dnp->dn_nlevels; ++i)
1213                        f.f_bl[i].bl_blk = -1;
1214                i = f.f_dnp->dn_nlevels - 1;
1215                f.f_bl[i].bl_nslots = f.f_dnp->dn_nblkptr;
1216                f.f_bl[i].bl_bp = &f.f_dnp->dn_blkptr[0];
1217                f.f_bl[i].bl_blk = 0;
1218
1219                err = fits_sa_setup(fromsnap, &f.f_from_sa_table);
1220                if (err)
1221                        goto out;
1222        }
1223        err = fits_sa_setup(tosnap, &f.f_to_sa_table);
1224        if (err)
1225                goto out;
1226
1227        err = zap_lookup(tosnap, MASTER_NODE_OBJ, ZFS_SHARES_DIR, 8, 1,
1228            &f.f_shares_dir);
1229        if (err && err != ENOENT)
1230                goto out;
1231
1232        err = fits_start(&f, &f.f_ops);
1233        if (err)
1234                goto out;
1235
1236        err = traverse_dataset(ds, f.f_fromtxg,
1237            TRAVERSE_PRE | TRAVERSE_PREFETCH_METADATA, fits_cb, &f);
1238        if (err) {
1239                fits_abort(&f);
1240                goto out;
1241        }
1242        err = fits_start2(&f, &f.f_ops);
1243        if (err) {
1244                goto out;
1245        }
1246        err = traverse_dataset(ds, f.f_fromtxg,
```



```
1247            TRAVERSE_PRE | TRAVERSE_PREFETCH_METADATA, fits_cb, &f);
1248        if (err) {
1249                fits_abort(&f);
1250                goto out;
1251        }
1252
1253        err = fits_end(&f);
1254        if (err)
1255                goto out;
1256
1257 out:
1258        if (fromds) {
1259                for (i = 0; i < f.f_dnp->dn_nlevels - 1; ++i) {
1260                        blklevel_t *b = f.f_bl + i;
1261                        if (b->bl_buf)
1262                                arc_buf_remove_ref(b->bl_buf, &b->bl_buf);
1263                }
1264                kmem_free(f.f_bl, sizeof (blklevel_t) * f.f_dnp->dn_nlevels);
1265        }
1266
1267        if (buf)
1268                arc_buf_remove_ref(buf, &buf);
1269
1270        return (err);
1271 }
1272 #endif /* ! codereview */
```

```
**********************************************************
    2798 Wed Oct 17 21:48:38 2012
new/usr/src/uts/common/fs/zfs/fits_count.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/errno.h>
#include <sys/fits_impl.h>

int
fits_add_count(fits_counter_t *fc, uint64_t ino, uint64_t inc,
               uint64_t aux, uint64_t *new_count, uint64_t *old_aux)
{
        fits_count_elem_t *fce = fc->fc_head;

        while (fce && fce->fce_ino != ino)
                fce = fce->fce_next;

        if (!fce) {
                fce = kmem_alloc(sizeof(*fce), KM_SLEEP);
                fce->fce_ino = ino;
                fce->fce_count = 0;
                fce->fce_next = fc->fc_head;
                fc->fc_head = fce;
        }

        if (old_aux) {
                *old_aux = fce->fce_aux;
                fce->fce_aux = aux;
        }
        fce->fce_count += inc;

        if (new_count)
                *new_count = fce->fce_count;

        return 0;
}

int
fits_get_count(fits_counter_t *fc, uint64_t ino, uint64_t *new_count,
```



```
               uint64_t *old_aux)
{
        fits_count_elem_t *fce = fc->fc_head;

        while (fce && fce->fce_ino != ino)
                fce = fce->fce_next;

        if (!fce) {
                if (new_count)
                        *new_count = 0;
                if (old_aux)
                        *old_aux = 0;
                return ENOENT;
        } else {
                if (new_count)
                        *new_count = fce->fce_count;
                if (old_aux)
                        *old_aux = fce->fce_aux;
        }

        return 0;
}

void
fits_free_count(fits_counter_t *fc, uint64_t ino)
{
        fits_count_elem_t *fce = fc->fc_head;
        fits_count_elem_t *prev = NULL;

        while (fce && fce->fce_ino != ino) {
                prev = fce;
                fce = fce->fce_next;
        }

        if (!fce)
                return;

        if (prev)
                prev->fce_next = fce->fce_next;
        else
                fc->fc_head = fce->fce_next;

        kmem_free(fce, sizeof(*fce));
}

int
fits_assert_count_empty(fits_counter_t *fc)
{
        fits_count_elem_t *fce = fc->fc_head;
        int ret = 0;

        while (fce) {
                printf("!fits_assert_count_empty: %s ino %"PRIu64" count %"PRIu6
                        fc->fc_name, fce->fce_ino, fce->fce_count);
                if (fce->fce_count != 0)
                        ++ret;
                fce = fce->fce_next;
                /* XXX TODO free count */
                /* XXX known leftover: if a file had > 1 links and be replaced
                 * by a file with > 1 link, but no same_name replacements, the
                 * link_add_cnt leaks with being 0
                 */
        }

        return ret;
}
```

```
125 #endif /* ! codereview */
```

```
**********************************************************
    3894 Wed Oct 17 21:48:38 2012
new/usr/src/uts/common/fs/zfs/fits_crc32c.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * The crc32c algorithms are taken from sctp_crc32 implementation
 * common/inet/sctp_crc32.{c,h}, which in turn were taken from nxge_fflp_hash.c
 */

#include <sys/zfs_context.h>
#include <sys/fits_crc32c.h>

static void fits_crc32c_word(uint32_t *crcptr, const uint32_t *buf, int len);

/*
 * Fast CRC32C calculation algorithm.  The basic idea is to look at it
 * four bytes (one word) at a time, using four tables.  The
 * standard algorithm in RFC 3309 uses one table.
 */

#define CRC_32C_POLY 0x1EDC6F41L

/* The four CRC32c tables. */
static uint32_t crc32c_tab[4][256];
static int initialized;


static uint32_t
reflect_32(uint32_t b)
{
        int i;
        uint32_t rw = 0;

        for (i = 0; i < 32; i++) {
                if (b & 1) {
                        rw |= 1 << (31 - i);
                }
                b >>= 1;
        }
        return (rw);
}
```



```
#ifdef _BIG_ENDIAN
static uint32_t
flip32(uint32_t w)
{
        return (((w >> 24) | ((w >> 8) & 0xff00) |
            ((w << 8) & 0xff0000) | (w << 24)));
}
#endif

/*
 * Initialize the crc32c tables.
 */

void
fits_crc32c_init(void)
{
        uint32_t index, bit, byte, crc;

        for (index = 0; index < 256; index++) {
                crc = reflect_32(index);
                for (byte = 0; byte < 4; byte++) {
                        for (bit = 0; bit < 8; bit++) {
                                crc = (crc & 0x80000000) ?
                                    (crc << 1) ^ CRC_32C_POLY : crc << 1;
                        }
#ifdef _BIG_ENDIAN
                        crc32c_tab[3 - byte][index] = flip32(reflect_32(crc));
#else
                        crc32c_tab[byte][index] = reflect_32(crc);
#endif
                }
        }
}

/*
 * Lookup  the crc32c for a byte stream
 */
static void
fits_crc32c_byte(uint32_t *crcptr, const uint8_t *buf, int len)
{
        uint32_t crc;
        int i;

        crc = *crcptr;
        for (i = 0; i < len; i++) {
#ifdef _BIG_ENDIAN
                crc = (crc << 8) ^ crc32c_tab[3][buf[i] ^ (crc >> 24)];
#else
                crc = (crc >> 8) ^ crc32c_tab[0][buf[i] ^ (crc & 0xff)];
#endif
        }
        *crcptr = crc;
}

/*
 * Lookup  the crc32c for a 32 bit word stream
 * Lookup is done fro the 4 bytes in parallel
 * from the tables computed earlier
 *
 */
static void
fits_crc32c_word(uint32_t *crcptr, const uint32_t *buf, int len)
{
        uint32_t w, crc;
        int i;
```

```
125         crc = *crcptr;
126         for (i = 0; i < len; i++) {
127                 w = crc ^ buf[i];
128                 crc = crc32c_tab[0][w >> 24] ^
129                     crc32c_tab[1][(w >> 16) & 0xff] ^
130                     crc32c_tab[2][(w >> 8) & 0xff] ^
131                     crc32c_tab[3][w & 0xff];
132         }
133         *crcptr = crc;
134 }

136 /*
137  * Lookup  the crc32c for a stream of bytes
138  *
139  * Tries to lookup the CRC on 4 byte words
140  * If the buffer is not 4 byte aligned, first compute
141  * with byte lookup until aligned. Then compute crc
142  * for each 4 bytes. If there are bytes left at the end of
143  * the buffer, then perform a byte lookup for the remaining bytes
144  *
145  *
146  */
147 uint32_t
148 fits_crc32c(uint32_t crc32, const uint8_t *buf, int len)
149 {
150         int rem;

152         if (!initialized) {
153                 fits_crc32c_init();
154                 initialized = 1;
155         }

157         rem = 4 - (((uintptr_t)buf) & 3);
158         if (rem != 0) {
159                 if (len < rem) {
160                         rem = len;
161                 }
162                 fits_crc32c_byte(&crc32, buf, rem);
163                 buf = buf + rem;
164                 len = len - rem;
165         }
166         if (len > 3) {
167                 fits_crc32c_word(&crc32, (const uint32_t *) buf, len / 4);
168         }
169         rem = len & 3;
170         if (rem != 0) {
171                 fits_crc32c_byte(&crc32, buf + len - rem, rem);
172         }
173         return (crc32);
174 }
175 #endif /* ! codereview */
```

```
**********************************************************
   11533 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/fits_pass1.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/fits.h>
#include <sys/fits_impl.h>

struct fits_enum {
        fits_t *fe_fits;
        uint64_t fe_parent_ino;
        fits_dirent_t *fe_dirent_chain;
};

struct fits_file {
        fits_t *ff_fits;
        uint64_t ff_len;
        uint64_t ff_last_byte;
        uint64_t ff_ino;
        fits_path_t *ff_path;
        fits_dirent_t *ff_dirent;
};

static int fits_file_data_pass1(void *fits_filep, void *data, uint64_t off,
    uint64_t len);
static int fits_dirent_add_pass1(void *fits_enump, char *name, uint64_t ino);
static int fits_dirent_mod_pass1(void *fits_enump, char *name,
    uint64_t ino_old, uint64_t ino_new);
static int fits_dir_add_pass1(fits_t *f, uint64_t ino);
static int fits_mod_pass1(fits_t *f, uint64_t ino);

static fits_ops_t _ops = {
        .fits_dir_add = fits_dir_add_pass1,
        .fits_dir_mod = fits_mod_pass1,
        .fits_dirent_add = fits_dirent_add_pass1,
        .fits_dirent_mod = fits_dirent_mod_pass1,
        .fits_file_mod = fits_mod_pass1,
```



```
        .fits_file_data = fits_file_data_pass1
};

static int fits_file_add_genchange(fits_t *f, uint64_t ino);

int
fits_start(fits_t *f, fits_ops_t **ops)
{
        int ret;

        f->f_pass = PASS_LINK;
        f->f_link_add_cnt.fc_head = NULL;
        f->f_link_add_cnt.fc_name = "link_add_cnt";
        f->f_del_dir_cnt.fc_head = NULL;
        f->f_del_dir_cnt.fc_name = "del_dir_cnt";
        f->f_put_back_cnt.fc_head = NULL;
        f->f_put_back_cnt.fc_name = "put_back_cnt";
        fits_send_init(f);

        *ops = &_ops;

        ret = fits_send_start(f);
        if (ret) {
                fits_abort(f);
                return (ret);
        }

        return (0);
}

static int
enum_dir(fits_t *f, uint64_t ino, fits_dirent_t *chain)
{
        struct fits_enum fe = {
                .fe_fits = f,
                .fe_parent_ino = ino,
                .fe_dirent_chain = chain
        };

        return (fits_dir_contents(f, ino, &fe));
}

static int
fits_file_data_pass1(void *fits_filep, void *data, uint64_t off, uint64_t len)
{
        struct fits_file *ff = fits_filep;

        if (off + len > ff->ff_len)
                len = ff->ff_len - off;

        ff->ff_last_byte = off + len;

        return fits_send_file_data(ff->ff_fits, &ff->ff_path,
            ff->ff_dirent, ff->ff_ino, off, len, data);
}

static int
dirent_add_dir(fits_t *f, fits_dirent_t *dirent, uint64_t ino, int exists)
{
        fits_info_t si_old;
        fits_info_t si_new;
        int ret;

        ret = fits_get_info(f, ino, FITS_OLD, &si_old,
                            FI_ATTR_PARENT | FI_ATTR_GEN | FI_ATTR_MODE);
        if (ret && ret != ENOENT)
```

```
125                 return (ret);
127         if (ret == 0) {
128                 ret = fits_get_info(f, ino, FITS_NEW, &si_new,
129                                     FI_ATTR_GEN | FI_ATTR_MODE);
130                 if (ret)
131                         return (ret);
133                 if (si_old.si_gen == si_new.si_gen ||
134                     (S_ISDIR(si_new.si_mode) && S_ISDIR(si_old.si_mode)))
135                         return fits_send_rename(f, dirent, ino,
136                                                 si_old.si_parent, exists);
137                 return (0);
138         }
140         if (ino > f->f_current_ino)
141                 return (0);
143         /* dir is new */
144         ret = fits_send_mkdir(f, dirent, ino, exists);
145         if (ret)
146                 return (ret);
148         return (enum_dir(f, ino, dirent));
149 }
151 int
152 fits_dirent_add_file(fits_t *f, fits_dirent_t *dirent,
153     uint64_t ino, uint64_t mode, int exists)
154 {
155         fits_info_t si_old;
156         fits_info_t si_new;
157         int ret;
158         fits_path_t *fits_path;
159         uint64_t new_count;
160         uint64_t old_aux;
162         ret = fits_get_info(f, ino, FITS_OLD, &si_old,
163                             FI_ATTR_GEN | FI_ATTR_PARENT);
164         if (ret && ret != ENOENT)
165                 return (ret);
167         if (ret == 0) {
168                 ret = fits_get_info(f, ino, FITS_NEW, &si_new,
169                                     FI_ATTR_GEN);
170                 if (ret)
171                         return (ret);
173                 if (si_old.si_gen == si_new.si_gen)
174                         return fits_send_link(f, dirent, ino, si_old.si_parent,
175                             FITS_OLD, exists);
176         }
178         /* file is new */
179         ret = fits_add_count(&f->f_link_add_cnt, ino, 1, dirent->fd_parent_ino,
180             &new_count, &old_aux);
181         if (ret)
182                 return (ret);
184         if (new_count == 1)
185                 ret = fits_send_create_file(f, dirent, ino,
186                                             exists, &fits_path);
187         else
188                 ret = fits_send_link(f, dirent, ino, old_aux, FITS_NEW, exists);
189         if (ret)
190                 return (ret);
```



```
192         ret = fits_get_info(f, ino, FITS_NEW, &si_new,
193                             FI_ATTR_SIZE | FI_ATTR_LINKS);
194         ASSERT(ret == 0);
195         if (new_count == 1 && S_ISREG(mode)) {
196                 struct fits_file ff;
198                 ff.ff_ino = ino;
199                 ff.ff_len = si_new.si_size;
200                 ff.ff_fits = f;
201                 ff.ff_path = fits_path;
202                 ff.ff_dirent = dirent;
203                 ff.ff_last_byte = 0;
204                 ret = fits_file_contents(f, ino, &ff);
205                 fits_path_free(ff.ff_path);
206                 if (ret)
207                         return (ret);
208                 if (ff.ff_last_byte != si_new.si_size) {
209                         /* sparse end */
210                         ret = fits_send_truncate(f, NULL, ino, si_new.si_size);
211                         if (ret)
212                                 return (ret);
213                 }
214         }
215         if (new_count == si_new.si_nlinks)
216                 fits_free_count(&f->f_link_add_cnt, ino);
218         return (0);
219 }
221 static int
222 dirent_add(fits_t *f, fits_dirent_t *dirent, uint64_t ino, int exists)
223 {
224         fits_info_t si;
225         int ret;
227         ret = fits_get_info(f, ino, FITS_NEW, &si, FI_ATTR_MODE);
228         if (ret)
229                 return (ret);
231         if (S_ISDIR(si.si_mode)) {
232                 return (dirent_add_dir(f, dirent, ino, exists));
233         } else {
234                 return (fits_dirent_add_file(f, dirent, ino, si.si_mode,
235                     exists));
236         }
237 }
239 static int
240 fits_dirent_add_pass1(void *fits_enump, char *name, uint64_t ino)
241 {
242         struct fits_enum *fe = fits_enump;
243         fits_dirent_t dirent = {
244                 .fd_name = name,
245                 .fd_parent_ino = fe->fe_parent_ino,
246                 .fd_prev = fe->fe_dirent_chain,
247         };
249         return (dirent_add(fe->fe_fits, &dirent, ino, 0));
250 }
252 static int
253 fits_dirent_mod_pass1(void *fits_enump, char *name,
254     uint64_t ino_old, uint64_t ino_new)
255 {
256         struct fits_enum *fe = fits_enump;
```

```
257         fits_dirent_t dirent = {
258                 .fd_name = name,
259                 .fd_parent_ino = fe->fe_parent_ino,
260                 .fd_prev = fe->fe_dirent_chain,
261         };

263         return (dirent_add(fe->fe_fits, &dirent, ino_new, 1));
264 }

266 static int
267 fits_file_add_genchange(fits_t *f, uint64_t ino)
268 {
269         int ret;
270         char *name = NULL;

272         f->f_current_ino = ino;
273         f->f_current_path = NULL;

275         /*
276          * only called when generation has changed. TODO: move to own
277          * function
278          */
279         fits_info_t si_old;
280         fits_info_t si_new;
281         int same_name = 0;

283         ret = fits_get_info(f, ino, FITS_OLD, &si_old, FI_ATTR_MODE |
284                             FI_ATTR_LINKS | FI_ATTR_PARENT);
285         if (ret)
286                 return (ret);
287         ret = fits_get_info(f, ino, FITS_NEW, &si_new, FI_ATTR_MODE |
288                             FI_ATTR_LINKS | FI_ATTR_PARENT);
289         if (ret)
290                 return (ret);

292         if (si_old.si_nlinks > 1 && si_new.si_nlinks > 1)
293                 return fits_add_count(&f->f_link_add_cnt, ino, 0, 0, NULL,
294                     NULL);

296         if (S_ISDIR(si_old.si_mode))
297                 return (0);

299         fits_which_t from;
300         fits_which_t to;
301         uint64_t new_ino;
302         uint64_t parent = si_new.si_parent;

304         if (si_old.si_nlinks == 1) {
305                 from = FITS_OLD;
306                 to = FITS_NEW;
307                 parent = si_old.si_parent;
308         } else if (si_old.si_nlinks > 1 && si_new.si_nlinks == 1) {
309                 from = FITS_NEW;
310                 to = FITS_OLD;
311                 parent = si_new.si_parent;
312         } else {
313                 return (EINVAL);
314         }

316         ret = fits_find_entry(f, parent, ino, from, &name);
317         if (ret)
318                 return (ret);

320         ret = fits_lookup_entry(f, parent, name, to, &new_ino);
321         if (ret && ret != ENOENT)
322                 goto out;
```



```
323         if (ret == 0 && new_ino == ino)
324                 same_name = 1;

326         if ((si_old.si_nlinks == 1 || si_new.si_nlinks == 1) && !same_name) {
327                 ret = 0;
328                 goto out;
329         }

331         fits_dirent_t dirent = {
332                 .fd_parent_ino = parent,
333                 .fd_name = name,
334                 .fd_prev = NULL
335         };

337         ret = fits_send_unlink(f, &dirent, ino);
338         if (ret)
339                 goto out;
340         ret = fits_dirent_add_file(f, &dirent, ino, si_new.si_mode, 0);

342 out:
343         fits_free_name(name);
344         return (ret);
345 }

347 static int
348 fits_dir_add_pass1(fits_t *f, uint64_t ino)
349 {
350         int ret;
351         uint64_t parent;
352         uint64_t first_parent = FITS_NO_INO;
353         fits_info_t si;
354         fits_dirent_t dirent;
355         int same_name = 0;
356         char *name = NULL;

358         f->f_current_ino = ino;
359         f->f_current_path = NULL;

361         parent = ino;
362         while (1) {
363                 /* the new parent must exist, otherwise the fs is wrong */
364                 ret = fits_get_info(f, parent, FITS_NEW, &si,
365                                     FI_ATTR_PARENT);
366                 if (ret)
367                         return (ret);
368                 if (first_parent == FITS_NO_INO)
369                         first_parent = si.si_parent;

371                 ret = fits_get_info(f, parent, FITS_OLD, &si, 0);
372                 if (ret && ret != ENOENT)
373                         return (ret);
374                 if (ret != ENOENT)
375                         break;

377                 /*
378                  * this check is only needed for a full send, on all
379                  * incrementals the parent already exists and it breaks out
380                  * above
381                  */
382                 if (parent == si.si_parent) {
383                         first_parent = FITS_NO_INO;
384                         break;
385                 }
386                 parent = si.si_parent;

388                 if (parent > ino)
```

```
389                         return (0);
390         }

392         /*
393          * check for same-name
394          */
395         if (first_parent != FITS_NO_INO) {
396                 ret = fits_get_info(f, first_parent, FITS_OLD, &si,
397                     FI_ATTR_MODE);
398                 if (ret && ret != ENOENT)
399                         return (ret);
400                 if (ret == 0 && S_ISDIR(si.si_mode)) {
401                         uint64_t old_ino;

403                         ret = fits_find_entry(f, first_parent, ino,
404                             FITS_NEW, &name);
405                         if (ret)
406                                 return (ret);

408                         ret = fits_lookup_entry(f, first_parent, name,
409                             FITS_OLD, &old_ino);
410                         if (ret && ret != ENOENT) {
411                                 goto out;
412                         }
413                         if (ret == 0) {
414                                 same_name = 1;
415                                 dirent.fd_name = name;
416                                 dirent.fd_parent_ino = first_parent;
417                                 dirent.fd_prev = NULL;
418                                 if (old_ino == ino) {
419                                         ret = fits_add_count(&f->f_link_add_cnt,
420                                             ino, 0, 0, NULL, NULL);
421                                         if (ret)
422                                                 goto out;
423                                 }
424                         }
425                 }
426         }
427         /* dir is new */
428         ret = fits_send_mkdir(f, same_name ? &dirent : NULL, ino, same_name);
429         if (ret)
430                 goto out;

432         ret = enum_dir(f, ino, NULL);

434 out:
435         fits_free_name(name);
436         return (ret);
437 }

439 static int
440 fits_mod_pass1(fits_t *f, uint64_t ino)
441 {
442         fits_info_t si_old;
443         fits_info_t si_new;
444         int ret;

446         f->f_current_ino = ino;
447         f->f_current_path = NULL;

449         ret = fits_get_info(f, ino, FITS_NEW, &si_new, FI_ATTR_SIZE |
450                             FI_ATTR_MODE | FI_ATTR_GEN | FI_ATTR_UID |
451                             FI_ATTR_GID | FI_ATTR_SIZE);
452         if (ret)
453                 return (ret);
454         ret = fits_get_info(f, ino, FITS_OLD, &si_old, FI_ATTR_GEN |
```



```
455                             FI_ATTR_MODE | FI_ATTR_UID |
456                             FI_ATTR_GID | FI_ATTR_SIZE);
457         if (ret)
458                 return (ret);

460         if (!(S_ISDIR(si_old.si_mode) && S_ISDIR(si_new.si_mode)) &&
461             si_new.si_gen != si_old.si_gen) {
462                 if (S_ISDIR(si_new.si_mode))
463                         return (fits_dir_add_pass1(f, ino));
464                 else
465                         return (fits_file_add_genchange(f, ino));
466         }

468         if (S_ISDIR(si_new.si_mode)) {
469                 ret = enum_dir(f, ino, NULL);
470                 if (ret)
471                         return (ret);
472         }

474         if (S_ISREG(si_new.si_mode)) {
475                 struct fits_file ff;
476                 ff.ff_ino = ino;
477                 ff.ff_len = si_new.si_size;
478                 ff.ff_fits = f;
479                 ff.ff_path = NULL;
480                 ff.ff_dirent = NULL;
481                 ff.ff_last_byte = 0;

483                 ret = fits_file_contents(f, ino, &ff);
484                 fits_path_free(ff.ff_path);
485                 if (ret)
486                         return (ret);
487                 if (si_new.si_size < si_old.si_size ||
488                     (si_new.si_size != si_old.si_size &&
489                     si_new.si_size != ff.ff_last_byte)) {
490                         ret = fits_send_truncate(f, NULL, ino, si_new.si_size);
491                         if (ret)
492                                 return (ret);
493                 }
494         }

496         if (si_old.si_uid != si_new.si_uid || si_old.si_gid != si_new.si_gid) {
497                 ret = fits_send_chown(f, NULL, ino, si_new.si_uid,
498                     si_new.si_gid);
499                 if (ret)
500                         return (ret);
501         }
502         if (si_old.si_mode != si_new.si_mode) {
503                 ret = fits_send_chmod(f, NULL, ino, si_new.si_mode);
504                 if (ret)
505                         return (ret);
506         }

508         return (ret);
509 }
510 #endif /* ! codereview */
```

```
**********************************************************
   11719 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/fits_pass2.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/fits.h>
#include <sys/fits_impl.h>

struct fits_enum {
        fits_t *fe_fits;
        uint64_t fe_parent_ino;
        uint64_t fe_del_dir_cnt;
        uint64_t fe_put_back_cnt;
        fits_dirent_t *fe_dirent_chain;
};

struct fits_file {
        fits_t *ff_fits;
        uint64_t ff_len;
        uint64_t ff_last_byte;
        uint64_t ff_ino;
        fits_path_t *ff_path;
        fits_dirent_t *ff_dirent;
};

static int fits_dirent_del_pass2(void *fits_enump, char *name, uint64_t ino);
static int fits_dirent_mod_pass2(void *fits_enump, char *name,
    uint64_t ino_old, uint64_t ino_new);
static int fits_dirent_unmod_pass2(void *fits_enump, char *name, uint64_t ino);
static int fits_dir_del_pass2(fits_t *f, uint64_t ino);
static int fits_add_pass2(fits_t *f, uint64_t ino);
static int fits_mod_pass2(fits_t *f, uint64_t ino);

fits_ops_t _ops = {
        .fits_dirent_del = fits_dirent_del_pass2,
        .fits_dirent_mod = fits_dirent_mod_pass2,
        .fits_dirent_unmod = fits_dirent_unmod_pass2,
```



```
        .fits_file_add = fits_add_pass2,
        .fits_file_mod = fits_mod_pass2,
        .fits_dir_add = fits_add_pass2,
        .fits_dir_del = fits_dir_del_pass2,
        .fits_dir_mod = fits_mod_pass2
};

int
fits_start2(fits_t *f, fits_ops_t **ops)
{
        f->f_pass = PASS_UNLINK;
        *ops = &_ops;

        return (0);
}

int
fits_abort(fits_t *f)
{
        fits_send_fini(f);

        return (0);
}

int
fits_end(fits_t *f)
{
        int ret;
        int ret2;

        fits_send_end(f);

        ret = fits_assert_count_empty(&f->f_link_add_cnt);
        ret += fits_assert_count_empty(&f->f_del_dir_cnt);
        ret += fits_assert_count_empty(&f->f_put_back_cnt);

        printf("!fits: fits_end calling fits_abort\n");
        ret2 = fits_abort(f);

        return (ret ? ret : ret2);
}

static int
dir_del(fits_t *f, uint64_t ino, uint64_t removed_entries)
{
        fits_info_t si;
        uint64_t parent;
        int ret;
        int exists;
        uint64_t new_count;
        uint64_t left;

        while (1) {
                ret = fits_get_info(f, ino, FITS_OLD, &si, FI_ATTR_GEN |
                                    FI_ATTR_PARENT | FI_ATTR_NENTRIES);
                if (ret)
                        return (ret);

                if (si.si_parent > f->f_current_ino)
                        return (0);

                if (removed_entries + 2 < si.si_nentries) /* '.' and '..' */
                        return (0);

                fits_free_count(&f->f_del_dir_cnt, ino);
                ret = fits_send_rmdir(f, NULL, ino);
```

```
125                 if (ret)
126                         return (ret);
127
128                 parent = si.si_parent;
129                 exists = fits_get_info(f, parent, FITS_NEW, &si,
130                     FI_ATTR_MODE);
131                 if (exists && exists != ENOENT)
132                         return (exists);
133
134                 ret = fits_get_count(&f->f_put_back_cnt, parent, &left, NULL);
135                 if (ret && ret != ENOENT)
136                         return (ret);
137                 if (left > 0) {
138                         ret = fits_add_count(&f->f_put_back_cnt, parent, -1, 0,
139                             &left, NULL);
140                         if (ret)
141                                 return (ret);
142                 }
143                 if ((int64_t)left < 0)
144                         return (EINVAL);
145
146                 if (exists == 0 && S_ISDIR(si.si_mode)) {
147                         char *name;
148                         uint64_t new_ino;
149                         fits_info_t si_new;
150                         int new;
151
152                         new = fits_get_info(f, ino, FITS_NEW, &si_new,
153                                             FI_ATTR_MODE);
154                         if (new && new != ENOENT)
155                                 return (new);
156                         ret = fits_find_entry(f, parent, ino, FITS_OLD, &name);
157                         if (ret)
158                                 return (ret);
159
160                         ret = fits_lookup_entry(f, parent, name,
161                                                 FITS_NEW, &new_ino);
162                         if (ret && ret != ENOENT) {
163                                 fits_free_name(name);
164                                 return (ret);
165                         }
166                         if (ret == 0 && new_ino != ino) {
167                                 fits_dirent_t dirent = {
168                                         .fd_name = name,
169                                         .fd_parent_ino = parent
170                                 };
171
172                                 ret = fits_send_rename_from_tempname(f, &dirent,
173                                     ino, new_ino);
174                                 if (ret) {
175                                         fits_free_name(name);
176                                         return (ret);
177                                 }
178                         } else if (ret == 0 && new_ino == ino &&
179                             !S_ISDIR(si_new.si_mode)) {
180                                 fits_dirent_t dirent = {
181                                         .fd_name = name,
182                                         .fd_parent_ino = parent
183                                 };
184
185                                 ret = fits_dirent_add_file(f, &dirent, ino,
186                                     si_new.si_mode, 0);
187                                 if (ret) {
188                                         fits_free_name(name);
189                                         return (ret);
190                                 }
```



```
191                                 }
192                                 fits_free_name(name);
193                         }
194
195                         if (left == 0) {
196                                 fits_free_count(&f->f_put_back_cnt, parent);
197                                 if (exists == 0 && S_ISDIR(si.si_mode)) {
198                                         ret = fits_send_mtime_update(f, NULL, parent);
199                                         if (ret)
200                                                 return (ret);
201                                 }
202                         }
203
204                         if (exists == 0 && S_ISDIR(si.si_mode))
205                                 return (0);
206
207                         /* propagate deletion */
208                         ret = fits_add_count(&f->f_del_dir_cnt, parent, 1, 0,
209                             &new_count, NULL);
210                         if (ret)
211                                 return (ret);
212
213                         ino = parent;
214                         removed_entries = new_count;
215                 }
216         }
217
218         static int
219         enum_dir(fits_t *f, uint64_t ino, uint64_t *pput_back_cnt,
220             uint64_t *pdel_dir_cnt)
221         {
222                 int ret;
223                 struct fits_enum fe = {
224                         .fe_fits = f,
225                         .fe_parent_ino = ino,
226                         .fe_del_dir_cnt = 0,
227                         .fe_put_back_cnt = 0
228                 };
229
230                 ret = fits_dir_contents(f, ino, &fe);
231                 if (ret)
232                         return (ret);
233
234                 if (pput_back_cnt)
235                         *pput_back_cnt = fe.fe_put_back_cnt;
236                 if (pdel_dir_cnt)
237                         *pdel_dir_cnt = fe.fe_del_dir_cnt;
238
239                 if (fe.fe_put_back_cnt) {
240                         ret = fits_add_count(&f->f_put_back_cnt, ino,
241                             fe.fe_put_back_cnt, 0, NULL, NULL);
242                         if (ret)
243                                 return (ret);
244                 }
245
246                 return (0);
247         }
248
249         static int
250         dirent_del_file(struct fits_enum *fe, fits_dirent_t *dirent,
251                         uint64_t ino, uint64_t remains)
252         {
253                 int ret;
254
255                 ret = fits_send_unlink(fe->fe_fits, dirent, ino);
256                 if (ret)
```

```
257                 return (ret);

259         if (remains != FITS_NO_INO) {
260                 ret = fits_send_rename_from_tempname(fe->fe_fits, dirent,
261                     ino, remains);
262                 if (ret)
263                         return (ret);
264         }

266         fe->fe_del_dir_cnt++;

268         return (0);
269 }

271 static int
272 dirent_del_dir(struct fits_enum *fe, fits_dirent_t *dirent, uint64_t ino,
273     uint64_t remains)
274 {
275         int ret;
276         fits_info_t si;
277         fits_info_t si_old;
278         int new;
279         int old;
280         fits_t *f = fe->fe_fits;

282         new = fits_get_info(f, ino, FITS_NEW, &si, FI_ATTR_GEN |
283                             FI_ATTR_MODE);
284         if (new && new != ENOENT)
285                 return (new);
286         old = fits_get_info(f, ino, FITS_OLD, &si_old,
287                             FI_ATTR_NENTRIES | FI_ATTR_GEN |
288                             FI_ATTR_PARENT);
289         if (old)
290                 return (old);

292         /* new == 0 means the dir was renamed, which happened during pass 1 */
293         if (new == ENOENT ||
294             (si.si_gen != si_old.si_gen && !S_ISDIR(si.si_mode))) {
295                 uint64_t cnt;

297                 if (ino > f->f_current_ino) {
298                         ++fe->fe_put_back_cnt;
299                         return (0);
300                 }

302                 ret = fits_get_count(&f->f_del_dir_cnt, ino, &cnt,
303                                      NULL);
304                 if (ret && ret != ENOENT)
305                         return (ret);
306                 /* 2 for '.' and '..' */
307                 if (cnt + 2 < si_old.si_nentries) {
308                         ++fe->fe_put_back_cnt;
309                         return (0);
310                 }

312                 fits_free_count(&f->f_del_dir_cnt, ino);
313                 ret = fits_send_rmdir(f, dirent, ino);
314                 if (ret)
315                         return (ret);
316         }
317         if (remains != FITS_NO_INO) {
318                 ret = fits_send_rename_from_tempname(f, dirent, ino, remains);
319                 if (ret)
320                         return (ret);
321         }
322         fe->fe_del_dir_cnt++;
```



```
323         if (new == 0 && si.si_gen != si_old.si_gen && !S_ISDIR(si.si_mode) &&
324             si_old.si_parent == dirent->fd_parent_ino) {
325                 uint64_t parent = si_old.si_parent;
326                 fits_info_t sip;
327                 char *name = NULL;

329                 ret = fits_get_info(f, parent, FITS_OLD, &sip,
330                                     FI_ATTR_MODE);
331                 if (ret && ret != ENOENT)
332                         return (ret);
333                 if (ret == 0 && S_ISDIR(sip.si_mode)) {
334                         uint64_t old_ino;

336                         ret = fits_find_entry(f, parent, ino, FITS_OLD, &name);
337                         if (ret)
338                                 return (ret);

340                         ret = fits_lookup_entry(f, parent, name,
341                                                 FITS_NEW, &old_ino);
342                         if (ret && ret != ENOENT) {
343                                 fits_free_name(name);
344                                 return (ret);
345                         }
346                         if (ret == 0) {
347                                 ret = fits_dirent_add_file(f, dirent, ino,
348                                     si.si_mode, 0);
349                                 if (ret)
350                                         return (ret);
351                         }
352                 }
353         }

355         return (0);
356 }

358 static int
359 dirent_del(struct fits_enum *fe, fits_dirent_t *dirent,
360     uint64_t ino, uint64_t remains)
361 {
362         fits_info_t si;
363         int ret;

365         ret = fits_get_info(fe->fe_fits, ino, FITS_OLD, &si, FI_ATTR_MODE);
366         if (ret)
367                 return (ret);

369         if (S_ISDIR(si.si_mode)) {
370                 return (dirent_del_dir(fe, dirent, ino, remains));
371         } else {
372                 return (dirent_del_file(fe, dirent, ino, remains));
373         }
374 }

376 static int
377 fits_dirent_del_pass2(void *fits_enump, char *name, uint64_t ino)
378 {
379         struct fits_enum *fe = fits_enump;
380         fits_dirent_t dirent = {
381                 .fd_name = name,
382                 .fd_parent_ino = fe->fe_parent_ino,
383                 .fd_prev = fe->fe_dirent_chain,
384         };

386         return (dirent_del(fe, &dirent, ino, FITS_NO_INO));
387 }
```

```
389 static int
390 fits_dirent_mod_pass2(void *fits_enump, char *name,
391     uint64_t ino_old, uint64_t ino_new)
392 {
393         struct fits_enum *fe = fits_enump;
394         fits_dirent_t dirent = {
395                 .fd_name = name,
396                 .fd_parent_ino = fe->fe_parent_ino,
397                 .fd_prev = fe->fe_dirent_chain,
398         };
399
400         return (dirent_del(fe, &dirent, ino_old, ino_new));
401 }
402
403 static int
404 fits_dirent_unmod_pass2(void *fits_enump, char *name, uint64_t ino)
405 {
406         struct fits_enum *fe = fits_enump;
407         fits_t *f = fe->fe_fits;
408         int ret;
409         uint64_t cnt;
410         fits_info_t si_old;
411         fits_info_t si_new;
412         fits_dirent_t dirent = {
413                 .fd_name = name,
414                 .fd_parent_ino = fe->fe_parent_ino,
415                 .fd_prev = fe->fe_dirent_chain,
416         };
417
418         ret = fits_get_count(&f->f_link_add_cnt, ino, &cnt, NULL);
419         if (ret)
420                 return (ret == ENOENT ? 0 : ret);
421
422         ret = fits_get_info(f, ino, FITS_OLD, &si_old, FI_ATTR_MODE);
423         if (ret)
424                 return (ret);
425
426         if (S_ISDIR(si_old.si_mode)) {
427                 return (dirent_del_dir(fe, &dirent, ino, 0));
428         }
429
430         ret = fits_get_info(f, ino, FITS_NEW, &si_new, FI_ATTR_MODE);
431         if (ret)
432                 return (ret);
433         ret = fits_send_unlink(f, &dirent, ino);
434         if (ret)
435                 return (ret);
436         if (S_ISDIR(si_new.si_mode)) {
437                 fits_free_count(&f->f_link_add_cnt, ino);
438                 ret = fits_send_rename_from_tempname(f, &dirent, ino, ino);
439                 if (ret)
440                         return (ret);
441                 ret = fits_send_mtime_update(f, &dirent, ino);
442         } else {
443                 ret = fits_dirent_add_file(f, &dirent, ino, si_new.si_mode, 0);
444         }
445
446         return (ret);
447 }
448
449 static int
450 fits_add_pass2(fits_t *f, uint64_t ino)
451 {
452         f->f_current_ino = ino;
453         f->f_current_path = NULL;
```



```
455         return (fits_send_mtime_update(f, NULL, ino));
456 }
457
458 static int
459 fits_dir_del_pass2(fits_t *f, uint64_t ino)
460 {
461         uint64_t put_back_cnt;
462         uint64_t del_dir_cnt;
463         uint64_t new_count;
464         int ret;
465
466         f->f_current_ino = ino;
467         f->f_current_path = NULL;
468
469         ret = enum_dir(f, ino, &put_back_cnt, &del_dir_cnt);
470         if (ret)
471                 return (ret);
472
473         ret = fits_add_count(&f->f_del_dir_cnt, ino, del_dir_cnt,
474             0, &new_count, NULL);
475         if (ret)
476                 return (ret);
477
478         if (put_back_cnt == 0) {
479                 ret = dir_del(f, ino, new_count);
480                 if (ret)
481                         return (ret);
482         }
483
484         return (0);
485 }
486
487 static int
488 fits_mod_pass2(fits_t *f, uint64_t ino)
489 {
490         fits_info_t si_old;
491         fits_info_t si_new;
492         int ret;
493         uint64_t put_back_cnt = 0;
494
495         f->f_current_ino = ino;
496         f->f_current_path = NULL;
497
498         ret = fits_get_info(f, ino, FITS_NEW, &si_new, FI_ATTR_SIZE |
499                             FI_ATTR_MODE | FI_ATTR_GEN | FI_ATTR_UID |
500                             FI_ATTR_GID | FI_ATTR_SIZE);
501         if (ret)
502                 return (ret);
503         ret = fits_get_info(f, ino, FITS_OLD, &si_old, FI_ATTR_GEN |
504                             FI_ATTR_MODE | FI_ATTR_UID |
505                             FI_ATTR_GID | FI_ATTR_SIZE);
506         if (ret)
507                 return (ret);
508
509         if (!(S_ISDIR(si_old.si_mode) && S_ISDIR(si_new.si_mode)) &&
510             si_new.si_gen != si_old.si_gen) {
511                 if (S_ISDIR(si_old.si_mode)) {
512                         ret = fits_dir_del_pass2(f, ino);
513                         if (ret)
514                                 return (ret);
515                 }
516                 return (fits_add_pass2(f, ino));
517         }
518
519         if (S_ISDIR(si_new.si_mode)) {
520                 ret = enum_dir(f, ino, &put_back_cnt, NULL);
```

```
521                 if (ret)
522                         return (ret);
523         }

525         if (put_back_cnt)
526                 return (0);

528         return (fits_send_mtime_update(f, NULL, ino));
529 }
530 #endif /* ! codereview */
```

```
**********************************************************
   21228 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/fits_send.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
 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 STRATO AG. All rights reserved.
23  */
24 #include <sys/zfs_context.h>
25 #include <sys/stat.h>
26 #include <sys/mkdev.h>
27 #include <sys/errno.h>
28 #include <sys/types.h>
29 #include <sys/fits.h>
30 #include <sys/fits_impl.h>
31 #include <sys/fits_crc32c.h>

33 #define TEMPNAME_PREFIX "fits-tempname-"
34 /* 2^128 needs 39 digits in decimal */
35 #define TEMPNAME_SIZE (sizeof (TEMPNAME_PREFIX) + 39)

37 void
38 fits_send_init(fits_t *f)
39 {
40         f->f_alloc_len = FITS_SEND_BUF_SIZE;
41         f->f_buf = kmem_alloc(f->f_alloc_len, KM_SLEEP);
42         f->f_size = 0;
43 }

45 void
46 fits_send_fini(fits_t *f)
47 {
48         kmem_free(f->f_buf, f->f_alloc_len);
49 }

51 static int
52 fits_send_reserve(fits_t *f, void **buf, int len)
53 {
54         int res = f->f_alloc_len - f->f_size;
55         if (len > res)
56                 return (-E2BIG);
57         *buf = f->f_buf + f->f_size;
58         f->f_size += len;
```



```
60         return (0);
61 }

63 static int
64 fits_send_put(fits_t *f, void *buf, int len)
65 {
66         int ret;
67         void *p;

69         ret = fits_send_reserve(f, &p, len);
70         if (ret)
71                 return (ret);

73         memcpy(p, buf, len);

75         return (0);
76 }

78 static int
79 fits_send_put_attr(fits_t *f, uint16_t attr, void *buf, int len)
80 {
81         fits_attr_header_t hdr;
82         int ret;

84         LE_OUT16(&hdr.fa_type, attr);
85         LE_OUT16(&hdr.fa_len, len);

87         ret = fits_send_put(f, &hdr, sizeof (hdr));
88         if (ret)
89                 return (ret);
90         return (fits_send_put(f, buf, len));
91 }

93 static int
94 fits_send_reserve_attr(fits_t *f, uint16_t attr, void **buf, int len)
95 {
96         fits_attr_header_t hdr;
97         int ret;

99         LE_OUT16(&hdr.fa_type, attr);
100        LE_OUT16(&hdr.fa_len, len);

102        ret = fits_send_put(f, &hdr, sizeof (hdr));
103        if (ret)
104                return (ret);
105        return (fits_send_reserve(f, buf, len));
106 }

108 static int
109 fits_send_put_u64(fits_t *f, uint16_t attr, uint64_t val)
110 {
111        uint64_t v;

113        LE_OUT64(&v, val);
114        return (fits_send_put_attr(f, attr, &v, sizeof (v)));
115 }

117 static int
118 fits_send_put_time(fits_t *f, uint16_t attr, fits_time_t *t)
119 {
120        char buf[12];

122        LE_OUT64(buf, t->st_sec);
123        LE_OUT32(buf + 8, t->st_nsec);
```

```
125         return (fits_send_put_attr(f, attr, buf, sizeof (buf)));
126 }

128 static int
129 fits_cmd_start(fits_t *f, uint16_t cmd)
130 {
131         fits_cmd_header_t ch;

133         memset(&ch, 0, sizeof (ch));
134         LE_OUT16(&ch.fc_cmd, cmd);
135         f->f_size = 0;
136         return (fits_send_put(f, &ch, sizeof (ch)));
137 }

139 static int
140 fits_cmd_send(fits_t *f)
141 {
142         fits_cmd_header_t *ch;
143         uint32_t crc;
144         int ret;

146         ch = (fits_cmd_header_t *)f->f_buf;
147         LE_OUT32(&ch->fc_len, f->f_size - sizeof (*ch));
148         ch->fc_crc = 0;

150         crc = fits_crc32c(0, f->f_buf, f->f_size);
151         LE_OUT32(&ch->fc_crc, crc);

153         ret = fits_write(f, f->f_buf, f->f_size);
154         f->f_size = 0;

156         return (ret);
157 }

159 static int
160 fits_send_stream_header(fits_t *f)
161 {
162         fits_stream_header_t header;

164         strcpy(header.fs_magic, FITS_SEND_STREAM_MAGIC);
165         LE_OUT32(&header.fs_version, FITS_SEND_STREAM_VERSION);

167         return (fits_write(f, (uint8_t *)&header, sizeof (header)));
168 }

170 static void
171 tempname(uint64_t ino, char *buf, int maxlen)
172 {
173         int l = sizeof (TEMPNAME_PREFIX) - 1;
174         memcpy(buf, TEMPNAME_PREFIX, MIN(maxlen, l));
175         snprintf(buf + l, maxlen - l, "%llu", (long long)ino);
176 }

178 static void
179 path_add_name(fits_path_t **fp, char *name, int namelen)
180 {
181         fits_path_t *new;

183         new = kmem_alloc(sizeof (*new) + namelen + 1, KM_SLEEP);
184         new->fp_next = *fp;
185         new->fp_len = namelen + 1;
186         new->fp_total_len = namelen + 1;
187         if (*fp)
188                 new->fp_total_len += (*fp)->fp_total_len;
189         memcpy(new->fp_buf, name, namelen);
190         new->fp_buf[namelen] = '\0';
```



```
191         *fp = new;
192 }

194 static void
195 path_copy(fits_path_t *fp, char *b)
196 {
197         fits_path_t *cur;

199         for (cur = fp; cur; cur = cur->fp_next) {
200                 *b = '/';
201                 memcpy(b + 1, cur->fp_buf, cur->fp_len - 1);
202                 b += cur->fp_len;
203         }
204 }

206 static void
207 path2buf(fits_path_t *fp, char **buf, int *buf_len)
208 {
209         char *b;

211         *buf_len = fp->fp_total_len + 1; /* one for the trailing 0-byte */
212         *buf = b = kmem_alloc(*buf_len, KM_SLEEP);

214         path_copy(fp, b);

216         b[*buf_len - 1] = '\0';
217 }

219 static int
220 put_path(fits_t *f, uint16_t attr, fits_path_t *fp)
221 {
222         int ret;
223         void *p;

225         ret = fits_send_reserve_attr(f, attr, &p, fp->fp_total_len);
226         if (ret)
227                 return (ret);
228         path_copy(fp, p);

230         return (0);
231 }

233 void
234 fits_path_free(fits_path_t *fp)
235 {
236         fits_path_t *next;
237         while (fp) {
238                 next = fp->fp_next;
239                 kmem_free(fp, fp->fp_len + sizeof (*fp));
240                 fp = next;
241         }
242 }

244 static int
245 is_ino_run(fits_t *f, uint64_t ino)
246 {
247         int ret;
248         fits_info_t si;

250         while (1) {
251                 if (ino > f->f_current_ino)
252                         return (0);
253                 ret = fits_get_info(f, ino, FITS_OLD, &si, 0);
254                 if (ret && ret != ENOENT)
255                         return (ret);
256                 if (ret != ENOENT)
```

```
257                         break;
258                 ret = fits_get_info(f, ino, FITS_NEW, &si, FI_ATTR_PARENT);
259                 if (ret && ret != ENOENT)
260                         return (ret);
261                 if (ret)
262                         return (0);     /* ignore for now */
263                 ino = si.si_parent;
264         }
265         return (1);
266 }

268 static int
269 build_path(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
270     int devise_tempname, fits_which_t which_in, fits_path_t **fp)
271 {
272         int ret = 0;
273         fits_dirent_t *de;
274         fits_info_t si;
275         fits_which_t which;
276         fits_dirent_t temp_dirent;
277         char temp_buf[TEMPNAME_SIZE];

279         if (devise_tempname) {
280                 if (!dirent)
281                         return (EINVAL);
282                 temp_dirent = *dirent;
283                 tempname(ino, temp_buf, sizeof (temp_buf));
284                 temp_dirent.fd_name = temp_buf;
285                 dirent = &temp_dirent;
286         }

288         *fp = NULL;

290         for (de = dirent; de; de = de->fd_prev) {
291                 path_add_name(fp, de->fd_name, strlen(de->fd_name));
292                 ino = de->fd_parent_ino;
293         }

295         /*
296          * XXX TODO check if f->f_current_path is set. if yes, use it instead.
297          * otherwise save result of loop below to f_current_path
298          */
299         while (1) {
300                 int namebuflen;
301                 char *name;
302                 char *t_name;
303                 uint64_t old_parent;
304                 uint64_t new_parent;
305                 uint64_t old_gen = 0;
306                 uint64_t new_gen = 0;
307                 uint64_t parent;
308                 int check_tempname;
309                 uint64_t old_mode = 0;

311                 old_parent = 0;
312                 new_parent = 0;
313                 check_tempname = 0;
314                 ret = fits_get_info(f, ino, FITS_OLD, &si, FI_ATTR_PARENT |
315                     FI_ATTR_GEN | FI_ATTR_MODE);
316                 if (ret && ret != ENOENT)
317                         return (ret);
318                 if (ret == 0) {
319                         old_parent = si.si_parent;
320                         old_gen = si.si_gen;
321                         old_mode = si.si_mode;
322                 }
```



```
323                 ret = fits_get_info(f, ino, FITS_NEW, &si, FI_ATTR_PARENT |
324                     FI_ATTR_GEN | FI_ATTR_MODE);
325                 if (ret && ret != ENOENT)
326                         return (ret);
327                 if (ret == 0) {
328                         new_parent = si.si_parent;
329                         new_gen = si.si_gen;
330                 }
331                 if (old_parent && new_parent && old_gen != new_gen &&
332                     !(S_ISDIR(old_mode) && S_ISDIR(si.si_mode))) {
333                         if (which_in == FITS_OLD) {
334                                 new_parent = 0;
335                         } else if (which_in == FITS_NEW) {
336                                 old_parent = 0;
337                                 if (S_ISDIR(si.si_mode))
338                                         check_tempname = 1;
339                         }
340                 }

342                 if (f->f_pass == PASS_LINK) {
343                         if (old_parent && !new_parent) {
344                                 which = FITS_OLD;
345                         } else if (!old_parent && new_parent) {
346                                 which = FITS_NEW;
347                         } else if (is_ino_run(f, new_parent)) {
348                                 check_tempname = 1;
349                                 which = FITS_NEW;
350                         } else {
351                                 which = FITS_OLD;
352                         }
353                 } else {
354                         if (old_parent && !new_parent) {
355                                 which = FITS_OLD;
356                         } else {
357                                 check_tempname = 1;
358                                 which = FITS_NEW;
359                         }
360                 }
361                 if (which == FITS_OLD)
362                         parent = old_parent;
363                 else
364                         parent = new_parent;
365                 if (parent == ino)
366                         break;
367                 ret = fits_find_entry(f, parent, ino, which, &t_name);
368                 if (ret)
369                         return (ret);
370                 name = strdup(t_name);
371                 fits_free_name(t_name);
372                 namebuflen = strlen(name) + 1;
373                 if (check_tempname) {
374                         fits_info_t si_old;
375                         fits_info_t si_new;

377                         ret = fits_get_info(f, parent, FITS_OLD,
378                                             &si_old, FI_ATTR_GEN);
379                         if (ret && ret != ENOENT)
380                                 return (ret);
381                         if (ret == 0) {
382                                 ret = fits_get_info(f, parent, FITS_NEW,
383                                                     &si_new, FI_ATTR_GEN);
384                                 if (ret)
385                                         return (ret);
386                                 if (si_old.si_gen != si_new.si_gen)
387                                         check_tempname = 0;
388                         } else {
```

```
389                                 check_tempname = 0;
390                         }
391                 }
392                 if (check_tempname) {
393                         uint64_t old_ino;

395                         ret = fits_lookup_entry(f, parent, name,
396                                                 FITS_OLD, &old_ino);
397                         if (ret && ret != ENOENT) {
398                                 fits_free_name(name);
399                                 return (ret);
400                         }
401                         if ((ret == 0 && old_ino != ino) ||
402                             (ret == 0 && S_ISDIR(si.si_mode) &&
403                             !S_ISDIR(old_mode) && old_ino == ino)) {
404                                 int ret;
405                                 uint64_t cnt = 1;

407                                 if (f->f_pass == PASS_UNLINK &&
408                                     new_parent < f->f_current_ino) {
409                                         ret = fits_get_count(&f->f_put_back_cnt,
410                                             old_ino, &cnt, NULL);
411                                         if (ret && ret != ENOENT)
412                                                 return (ret);
413                                 }
414                                 if (cnt) {
415                                         kmem_free(name, namebuflen);
416                                         namebuflen = TEMPNAME_SIZE;
417                                         name = kmem_alloc(namebuflen, KM_SLEEP);
418                                         tempname(ino, name, namebuflen);
419                                 }
420                         }
421                 }
422                 ino = parent;
423                 path_add_name(fp, name, strlen(name));
424                 kmem_free(name, namebuflen);
425         }
426         if (*fp == NULL)
427                 path_add_name(fp, "", 0);

429         return (0);
430 }

432 int
433 fits_send_start(fits_t *f)
434 {
435         int ret;
436         uint8_t o_uuid[16];
437         uint8_t n_uuid[16];
438         uint64_t o_ctrans;
439         uint64_t n_ctrans;
440         char *path = NULL;
441         int len;
442         char *p;
443         int cmd = FITS_CMD_SUBVOL;

445         ret = fits_send_stream_header(f);
446         if (ret) {
447                 fits_abort(f);
448                 return (ret);
449         }

451         if ((ret = fits_get_uuid(f, FITS_NEW, n_uuid)) ||
452             (ret = fits_get_ctransid(f, FITS_NEW, &n_ctrans)) ||
453             (ret = fits_get_snapname(f, FITS_NEW, &path, &len)))
454                 goto out;
```



```
455         /* for now, strip the pool name */
456         if ((p = strchr(path, '/')))
457                 ++p;
458         else
459                 p = path;
460         ret = fits_get_uuid(f, FITS_OLD, o_uuid);
461         if (ret && ret != ENOENT)
462                 goto out;
463         if (ret == 0) {
464                 ret = fits_get_ctransid(f, FITS_OLD, &o_ctrans);
465                 if (ret)
466                         goto out;
467                 cmd = FITS_CMD_SNAPSHOT;
468         }
469         if ((ret = fits_cmd_start(f, cmd)) ||
470             (ret = fits_send_put_attr(f, FITS_ATTR_PATH, p, strlen(p))) ||
471             (ret = fits_send_put_u64(f, FITS_ATTR_CTRANSID, n_ctrans)) ||
472             (ret = fits_send_put_attr(f, FITS_ATTR_UUID, n_uuid, 16)))
473                 goto out;
474         if (cmd == FITS_CMD_SNAPSHOT) {
475                 if ((ret = fits_send_put_u64(f, FITS_ATTR_CLONE_CTRANSID,
476                     o_ctrans)) ||
477                     (ret = fits_send_put_attr(f, FITS_ATTR_CLONE_UUID,
478                     o_uuid, 16)))
479                         goto out;
480         }
481         ret = fits_cmd_send(f);

483 out:
484         kmem_free(path, len);

486         return (ret);
487 }

489 int
490 fits_send_create_file(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
491     int devise_tempname, fits_path_t **path_ret)
492 {
493         fits_path_t *path = NULL;
494         fits_info_t si;
495         int ret;
496         int send_rdev = 0;
497         int cmd;
498         uint64_t rdev = 0;
499         char *symlink = NULL;
500         int symlen = 0;

502         ret = build_path(f, dirent, ino, devise_tempname, FITS_NEW, &path);
503         if (ret)
504                 goto out;

506         ret = fits_get_info(f, ino, FITS_NEW, &si,
507                             FI_ATTR_MODE | FI_ATTR_UID | FI_ATTR_GID);
508         if (ret)
509                 goto out;

511         if (S_ISREG(si.si_mode)) {
512                 cmd = FITS_CMD_MKFILE;
513         } else if (S_ISDIR(si.si_mode)) {
514                 cmd = FITS_CMD_MKDIR;
515         } else if (S_ISLNK(si.si_mode)) {
516                 cmd = FITS_CMD_SYMLINK;
517                 ret = fits_read_symlink(f, ino, FITS_NEW, &symlink, &symlen);
518                 if (ret)
519                         goto out;
520         } else if (S_ISCHR(si.si_mode) || S_ISBLK(si.si_mode)) {
```

```
521                 cmd = FITS_CMD_MKNOD;
522                 send_rdev = 1;
523         } else if (S_ISFIFO(si.si_mode)) {
524                 cmd = FITS_CMD_MKFIFO;
525         } else if (S_ISSOCK(si.si_mode)) {
526                 cmd = FITS_CMD_MKSOCK;
527         } else {
528                 /* unknown file type, ignore for now */
529                 printf("!send_create_file: ignore file with mode 0%lo\n",
530                         (unsigned long)si.si_mode);
531                 return (0);
532         }

534         if (send_rdev) {
535                 fits_info_t sirdev;
536                 uint64_t r_major;
537                 uint64_t r_minor;
538                 ret = fits_get_info(f, ino, FITS_NEW, &sirdev, FI_ATTR_RDEV);
539                 if (ret)
540                         goto out;
541                 rdev = sirdev.si_rdev;

543                 /* XXX hardcodedly transform rdev to linux form */
544                 r_major = rdev >> 32;
545                 r_minor = rdev & 0xffffffful;
546                 rdev = ((r_minor & 0xff) | ((r_major & 0xfff) << 8) |
547                     ((r_minor >> 8) << 20) | ((r_major >> 12) << 44));
548         }
549         /* send MKFILE */
550         if ((ret = fits_cmd_start(f, cmd)) ||
551             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
552             (ret = fits_send_put_u64(f, FITS_ATTR_INO, ino)))
553                 goto out;
554         if (send_rdev) {
555                 ret = fits_send_put_u64(f, FITS_ATTR_RDEV, rdev);
556                 if (ret)
557                         goto out;
558                 ret = fits_send_put_u64(f, FITS_ATTR_MODE, si.si_mode);
559                 if (ret)
560                         goto out;
561         }
562         if (S_ISLNK(si.si_mode)) {
563                 ret = fits_send_put_attr(f, FITS_ATTR_PATH_LINK,
564                                          symlink, strlen(symlink));
565                 if (ret)
566                         goto out;
567         }
568         if ((ret = fits_cmd_send(f)))
569                 goto out;

571         /* send CHOWN */
572         if ((ret = fits_cmd_start(f, FITS_CMD_CHOWN)) ||
573             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
574             (ret = fits_send_put_u64(f, FITS_ATTR_UID, si.si_uid)) ||
575             (ret = fits_send_put_u64(f, FITS_ATTR_GID, si.si_gid)) ||
576             (ret = fits_cmd_send(f)))
577                 goto out;

579         /* send CHMOD */
580         if ((ret = fits_cmd_start(f, FITS_CMD_CHMOD)) ||
581             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
582             (ret = fits_send_put_u64(f, FITS_ATTR_MODE,
583             si.si_mode & 0xfff)) ||
584             (ret = fits_cmd_send(f)))
585                 goto out;
```



```
587 out:
588         if (ret == 0 && path_ret)
589                 *path_ret = path;
590         else
591                 fits_path_free(path);
592         if (symlink)
593                 kmem_free(symlink, symlen);
594         return (ret);
595 }

597 int
598 fits_send_link(fits_t *f, fits_dirent_t *new_dirent, uint64_t ino,
599     uint64_t old_parent_ino, fits_which_t which, int devise_tempname)
600 {
601         fits_path_t *new_path = NULL;
602         fits_path_t *old_path = NULL;
603         int ret;
604         fits_dirent_t old_dirent = {
605                 .fd_name = NULL,
606                 .fd_parent_ino = old_parent_ino,
607                 .fd_prev = NULL,
608         };

610         ret = fits_find_entry(f, old_parent_ino, ino, which,
611             &old_dirent.fd_name);
612         if (ret)
613                 return (ret);
614         ret = build_path(f, &old_dirent, ino, 0, FITS_OLD, &old_path);
615         if (ret)
616                 goto out;

618         ret = build_path(f, new_dirent, ino, devise_tempname, FITS_NEW,
619             &new_path);
620         if (ret)
621                 goto out;

623         if ((ret = fits_cmd_start(f, FITS_CMD_LINK)) ||
624             (ret = put_path(f, FITS_ATTR_PATH_LINK, old_path)) ||
625             (ret = put_path(f, FITS_ATTR_PATH, new_path)) ||
626             (ret = fits_cmd_send(f)))
627                 goto out;

629         if (f->f_pass == PASS_UNLINK)
630                 ret = fits_send_mtime_update(f, new_dirent, ino);
631 out:
632         if (old_dirent.fd_name)
633                 kmem_free(old_dirent.fd_name, strlen(old_dirent.fd_name) + 1);
634         fits_path_free(old_path);
635         fits_path_free(new_path);
636         return (ret);
637 }

639 int
640 fits_send_mkdir(fits_t *f, fits_dirent_t *dirent,
641                 uint64_t ino, int devise_tempname)
642 {
643         fits_path_t *path = NULL;
644         fits_info_t si;
645         int ret;

647         ret = build_path(f, dirent, ino, devise_tempname, FITS_NEW, &path);
648         if (ret)
649                 goto out;

651         ret = fits_get_info(f, ino, FITS_NEW, &si,
652                             FI_ATTR_UID | FI_ATTR_GID | FI_ATTR_MODE);
```

```
653         if (ret)
654                 goto out;

656         if (path->fp_total_len != 1) {
657                 /* don't send an mkdir for the root, but send chown/chmod */
658                 if ((ret = fits_cmd_start(f, FITS_CMD_MKDIR)) ||
659                     (ret = put_path(f, FITS_ATTR_PATH, path)) ||
660                     (ret = fits_send_put_u64(f, FITS_ATTR_INO, ino)) ||
661                     (ret = fits_cmd_send(f)))
662                         goto out;
663         }

665         /* send CHOWN */
666         if ((ret = fits_cmd_start(f, FITS_CMD_CHOWN)) ||
667             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
668             (ret = fits_send_put_u64(f, FITS_ATTR_UID, si.si_uid)) ||
669             (ret = fits_send_put_u64(f, FITS_ATTR_GID, si.si_gid)) ||
670             (ret = fits_cmd_send(f)))
671                 goto out;

673         /* send CHMOD */
674         if ((ret = fits_cmd_start(f, FITS_CMD_CHMOD)) ||
675             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
676             (ret = fits_send_put_u64(f, FITS_ATTR_MODE,
677             si.si_mode & 0xfff)) ||
678             (ret = fits_cmd_send(f)))
679                 goto out;

681 out:
682         fits_path_free(path);
683         return (ret);
684 }

686 /* this one is only used for directory renames */
687 int
688 fits_send_rename(fits_t *f, fits_dirent_t *new_dirent, uint64_t ino,
689     uint64_t old_parent_ino, int devise_tempname)
690 {
691         fits_path_t *new_path = NULL;
692         fits_path_t *old_path = NULL;
693         int ret;
694         fits_dirent_t old_dirent = {
695                 .fd_name = NULL,
696                 .fd_parent_ino = old_parent_ino,
697                 .fd_prev = NULL,
698         };

700         ret = fits_find_entry(f, old_parent_ino, ino, FITS_OLD,
701             &old_dirent.fd_name);
702         if (ret)
703                 return (ret);
704         ret = build_path(f, &old_dirent, ino, 0, FITS_OLD, &old_path);
705         if (ret)
706                 goto out;

708         ret = build_path(f, new_dirent, ino, devise_tempname, FITS_NEW,
709             &new_path);
710         if (ret)
711                 goto out;

713         if ((ret = fits_cmd_start(f, FITS_CMD_RENAME)) ||
714             (ret = put_path(f, FITS_ATTR_PATH, old_path)) ||
715             (ret = put_path(f, FITS_ATTR_PATH_TO, new_path)) ||
716             (ret = fits_cmd_send(f)))
717                 goto out;
718 out:
```



```
719         fits_path_free(old_path);
720         fits_path_free(new_path);
721         return (ret);
722 }

724 int
725 fits_send_rename_from_tempname(fits_t *f, fits_dirent_t *dirent,
726     uint64_t ino, uint64_t old)
727 {
728         char buf[TEMPNAME_SIZE];
729         fits_path_t *new_path = NULL;
730         fits_path_t *old_path = NULL;
731         int ret;
732         fits_dirent_t old_dirent;

734         tempname(old, buf, sizeof (buf));
735         old_dirent = *dirent;
736         old_dirent.fd_name = buf;

738         ret = build_path(f, &old_dirent, old, 0, FITS_OLD, &old_path);
739         if (ret)
740                 goto out;
741         ret = build_path(f, dirent, ino, 0, FITS_NEW, &new_path);
742         if (ret)
743                 goto out;

745         if ((ret = fits_cmd_start(f, FITS_CMD_RENAME)) ||
746             (ret = put_path(f, FITS_ATTR_PATH, old_path)) ||
747             (ret = put_path(f, FITS_ATTR_PATH_TO, new_path)) ||
748             (ret = fits_cmd_send(f)))
749                 goto out;

751         ret = fits_send_mtime_update(f, dirent, old);

753 out:
754         fits_path_free(old_path);
755         fits_path_free(new_path);
756         return (ret);
757 }

759 int
760 fits_send_unlink(fits_t *f, fits_dirent_t *dirent, uint64_t ino)
761 {
762         fits_path_t *path = NULL;
763         int ret;

765         ret = build_path(f, dirent, ino, 0, FITS_OLD, &path);
766         if (ret)
767                 goto out;

769         if ((ret = fits_cmd_start(f, FITS_CMD_UNLINK)) ||
770             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
771             (ret = fits_cmd_send(f)))
772                 goto out;

774 out:
775         fits_path_free(path);
776         return (ret);
777 }

779 int
780 fits_send_rmdir(fits_t *f, fits_dirent_t *dirent, uint64_t ino)
781 {
782         fits_path_t *path;
783         int ret;
```

```
785         ret = build_path(f, dirent, ino, 0, FITS_OLD, &path);
786         if (ret)
787                 goto out;

789         if ((ret = fits_cmd_start(f, FITS_CMD_RMDIR)) ||
790             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
791             (ret = fits_cmd_send(f)))
792                 goto out;

794 out:
795         fits_path_free(path);
796         return (ret);
797 }

799 int
800 fits_send_file_data(fits_t *f, fits_path_t **path_p,
801                     fits_dirent_t *dirent, uint64_t ino,
802                     uint64_t off, uint64_t len, void *data)
803 {
804         int ret = 0;

806         if (!*path_p) {
807                 ret = build_path(f, dirent, ino, 0, FITS_NEW, path_p);
808                 if (ret)
809                         return (ret);
810         }

812         while (len) {
813                 uint64_t l = MIN(len, FITS_SEND_READ_SIZE);

815                 if ((ret = fits_cmd_start(f, FITS_CMD_WRITE)) ||
816                     (ret = put_path(f, FITS_ATTR_PATH, *path_p)) ||
817                     (ret = fits_send_put_u64(f, FITS_ATTR_FILE_OFFSET, off))||
818                     (ret = fits_send_put_attr(f, FITS_ATTR_DATA, data, l)) ||
819                     (ret = fits_cmd_send(f)))
820                         goto out;
821                 data += l;
822                 off += l;
823                 len -= l;
824         }

826 out:
827         return (ret);
828 }

830 int
831 fits_send_mtime_update(fits_t *f, fits_dirent_t *dirent, uint64_t ino)
832 {
833         fits_path_t *path = NULL;
834         int ret;
835         fits_info_t si;

837         ret = fits_get_info(f, ino, FITS_NEW, &si,
838                             FI_ATTR_ATIME | FI_ATTR_MTIME |
839                             FI_ATTR_CTIME | FI_ATTR_OTIME);
840         if (ret) {
841                 if (ret == ENOENT)
842                         ret = 0;
843                 goto out;
844         }

846         ret = build_path(f, dirent, ino, 0, FITS_NEW, &path);
847         if (ret)
848                 goto out;

850         if ((ret = fits_cmd_start(f, FITS_CMD_UTIMES)) ||
```



```
851             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
852             (ret = fits_send_put_time(f, FITS_ATTR_ATIME, &si.si_atime)) ||
853             (ret = fits_send_put_time(f, FITS_ATTR_MTIME, &si.si_mtime)) ||
854             (ret = fits_send_put_time(f, FITS_ATTR_CTIME, &si.si_ctime)) ||
855             (ret = fits_send_put_time(f, FITS_ATTR_OTIME, &si.si_otime)) ||
856             (ret = fits_cmd_send(f)))
857                 goto out;

859 out:
860         fits_path_free(path);
861         return (ret);
862 }

864 int
865 fits_send_truncate(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
866     uint64_t new_size)
867 {
868         fits_path_t *path = NULL;
869         int ret;

871         ret = build_path(f, dirent, ino, 0, FITS_NEW, &path);
872         if (ret)
873                 return (ret);

875         if ((ret = fits_cmd_start(f, FITS_CMD_TRUNCATE)) ||
876             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
877             (ret = fits_send_put_u64(f, FITS_ATTR_SIZE, new_size)) ||
878             (ret = fits_cmd_send(f)))
879                 goto out;

881 out:
882         fits_path_free(path);
883         return (ret);
884 }

886 int
887 fits_send_chown(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
888                     uint64_t new_uid, uint64_t new_gid)
889 {
890         fits_path_t *path = NULL;
891         int ret;

893         ret = build_path(f, dirent, ino, 0, FITS_NEW, &path);
894         if (ret)
895                 return (ret);

897         if ((ret = fits_cmd_start(f, FITS_CMD_CHOWN)) ||
898             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
899             (ret = fits_send_put_u64(f, FITS_ATTR_UID, new_uid)) ||
900             (ret = fits_send_put_u64(f, FITS_ATTR_GID, new_gid)) ||
901             (ret = fits_cmd_send(f)))
902                 goto out;

904 out:
905         fits_path_free(path);
906         return (ret);
907 }

909 int
910 fits_send_chmod(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
911                     uint64_t new_mode)
912 {
913         fits_path_t *path = NULL;
914         int ret;

916         ret = build_path(f, dirent, ino, 0, FITS_NEW, &path);
```

```
917         if (ret)
918                 return (ret);

920         if ((ret = fits_cmd_start(f, FITS_CMD_CHMOD)) ||
921             (ret = put_path(f, FITS_ATTR_PATH, path)) ||
922             (ret = fits_send_put_u64(f, FITS_ATTR_MODE, new_mode)) ||
923             (ret = fits_cmd_send(f)))
924                 goto out;

926 out:
927         fits_path_free(path);
928         return (ret);
929 }

931 int
932 fits_send_end(fits_t *f)
933 {
934         int ret;

936         if ((ret = fits_cmd_start(f, FITS_CMD_END)) ||
937             (ret = fits_cmd_send(f)))
938                 goto out;

940 out:
941         return (ret);
942 }
943 #endif /* ! codereview */
```

```
**********************************************************
   10469 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

 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 int dsl_dataset_namelen(dsl_dataset_t *ds);
 206 #endif /* ! codereview */
 207 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
 208 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
 209 void dsl_dataset_drop_ref(dsl_dataset_t *ds, void *tag);
 210 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, boolean_t inconsistentok,
 211     void *tag);
 212 void dsl_dataset_make_exclusive(dsl_dataset_t *ds, void *tag);
 213 void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
 214     minor_t minor);
 215 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
 216     dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
 217 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
 218     uint64_t flags, dmu_tx_t *tx);
 219 int dsl_dataset_destroy(dsl_dataset_t *ds, void *tag, boolean_t defer);
 220 dsl_checkfunc_t dsl_dataset_destroy_check;
 221 dsl_syncfunc_t dsl_dataset_destroy_sync;
 222 dsl_syncfunc_t dsl_dataset_user_hold_sync;
 223 int dsl_dataset_snapshot_check(dsl_dataset_t *ds, const char *, dmu_tx_t *tx);
 224 void dsl_dataset_snapshot_sync(dsl_dataset_t *ds, const char *, dmu_tx_t *tx);
 225 int dsl_dataset_rename(char *name, const char *newname, boolean_t recursive);
 226 int dsl_dataset_promote(const char *name, char *conflsnap);
 227 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
 228     boolean_t force);
 229 int dsl_dataset_user_hold(char *dsname, char *snapname, char *htag,
 230     boolean_t recursive, boolean_t temphold, int cleanup_fd);
 231 int dsl_dataset_user_hold_for_send(dsl_dataset_t *ds, char *htag,
 232     boolean_t temphold);
 233 int dsl_dataset_user_release(char *dsname, char *snapname, char *htag,
 234     boolean_t recursive);
 235 int dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
 236     char *htag, boolean_t retry);
 237 int dsl_dataset_get_holds(const char *dsname, nvlist_t **nvp);

 239 blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
 240 void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

 242 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

 244 boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

 246 void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);
```



```
 248 void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
 249     dmu_tx_t *tx);
 250 int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
 251     dmu_tx_t *tx, boolean_t async);
 252 boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
 253     uint64_t blk_birth);
 254 uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

 256 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
 257 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
 258 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
 259 void dsl_dataset_space(dsl_dataset_t *ds,
 260     uint64_t *refdbytesp, uint64_t *availbytesp,
 261     uint64_t *usedobjsp, uint64_t *availobjsp);
 262 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
 263 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
 264     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
 265 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
 266     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
 267 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

 269 int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

 271 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
 272     uint64_t asize, uint64_t inflight, uint64_t *used,
 273     uint64_t *ref_rsrv);
 274 int dsl_dataset_set_quota(const char *dsname, zprop_source_t source,
 275     uint64_t quota);
 276 dsl_syncfunc_t dsl_dataset_set_quota_sync;
 277 int dsl_dataset_set_reservation(const char *dsname, zprop_source_t source,
 278     uint64_t reservation);

 280 int dsl_destroy_inconsistent(const char *dsname, void *arg);

 282 #ifdef ZFS_DEBUG
 283 #define dprintf_ds(ds, fmt, ...) do { \
 284         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
 285         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
 286         dsl_dataset_name(ds, __ds_name); \
 287         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
 288         kmem_free(__ds_name, MAXNAMELEN); \
 289         } \
 290 _NOTE(CONSTCOND) } while (0)
 291 #else
 292 #define dprintf_ds(dd, fmt, ...)
 293 #endif

 295 #ifdef  __cplusplus
 296 }
 297 #endif

 299 #endif /* _SYS_DSL_DATASET_H */
```

```
**********************************************************
    3383 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/sys/fits.h
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2012 Alexander Block. All rights reserved.
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */

#ifndef _SYS_FITS_H
#define _SYS_FITS_H

#include <sys/inttypes.h>
#include <sys/types.h>
#include <sys/dmu.h>
#include <sys/vnode.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define FITS_SEND_STREAM_MAGIC "btrfs-stream"
#define FITS_SEND_STREAM_VERSION 1

#define FITS_SEND_BUF_SIZE 65536
#define FITS_SEND_READ_SIZE 49152

typedef struct _fits_stream_header {
        char            fs_magic[sizeof (FITS_SEND_STREAM_MAGIC)];
        uint32_t        fs_version;
} __attribute__((__packed__)) fits_stream_header_t;

typedef struct _fits_cmd_header {
        /* len of the payload, not including header */
        uint32_t        fc_len;
        uint16_t        fc_cmd;
        /* the crc includes the header, but with fc_crc assumed as 0 */
        uint32_t        fc_crc;
} __attribute__((__packed__)) fits_cmd_header_t;

typedef struct _fits_attr_header {
```



```
        uint16_t        fa_type;
        /* len of the payload, not including header */
        uint16_t        fa_len;
} __attribute__((__packed__)) fits_attr_header_t;

/* commands */
#define FITS_CMD_SUBVOL         1
#define FITS_CMD_SNAPSHOT       2
#define FITS_CMD_MKFILE         3
#define FITS_CMD_MKDIR          4
#define FITS_CMD_MKNOD          5
#define FITS_CMD_MKFIFO         6
#define FITS_CMD_MKSOCK         7
#define FITS_CMD_SYMLINK        8
#define FITS_CMD_RENAME         9
#define FITS_CMD_LINK           10
#define FITS_CMD_UNLINK         11
#define FITS_CMD_RMDIR          12
#define FITS_CMD_SET_XATTR      13
#define FITS_CMD_REMOVE_XATTR   14
#define FITS_CMD_WRITE          15
#define FITS_CMD_CLONE          16
#define FITS_CMD_TRUNCATE       17
#define FITS_CMD_CHMOD          18
#define FITS_CMD_CHOWN          19
#define FITS_CMD_UTIMES         20
#define FITS_CMD_END            21
#define FITS_CMD_MAX            21

/* attributes */
#define FITS_ATTR_UUID                  1
#define FITS_ATTR_CTRANSID              2
#define FITS_ATTR_INO                   3
#define FITS_ATTR_SIZE                  4
#define FITS_ATTR_MODE                  5
#define FITS_ATTR_UID                   6
#define FITS_ATTR_GID                   7
#define FITS_ATTR_RDEV                  8
#define FITS_ATTR_CTIME                 9
#define FITS_ATTR_MTIME                 10
#define FITS_ATTR_ATIME                 11
#define FITS_ATTR_OTIME                 12
#define FITS_ATTR_XATTR_NAME            13
#define FITS_ATTR_XATTR_DATA            14
#define FITS_ATTR_PATH                  15
#define FITS_ATTR_PATH_TO               16
#define FITS_ATTR_PATH_LINK             17
#define FITS_ATTR_FILE_OFFSET           18
#define FITS_ATTR_DATA                  19
#define FITS_ATTR_CLONE_UUID            20
#define FITS_ATTR_CLONE_CTRANSID        21
#define FITS_ATTR_CLONE_PATH            22
#define FITS_ATTR_CLONE_OFFSET          23
#define FITS_ATTR_CLONE_LEN             24
#define FITS_ATTR_MAX                   24

int fits_send(objset_t *tosnap, objset_t *fromsnap, int outfd, vnode_t *vp,
    offset_t *off);

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_FITS_H */
#endif /* ! codereview */
```

****
1026 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/sys/fits_crc32c.h
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

#ifndef _SYS_FITS_CRC32C_H
#define _SYS_FITS_CRC32C_H

#include <sys/inttypes.h>
#include <sys/types.h>

uint32_t fits_crc32c(uint32_t seed, const uint8_t *data, int len);

#endif /* _SYS_FITS_CRC32C_H */
#endif /* ! codereview */
```

```
**********************************************************
    7223 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/sys/fits_impl.h
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */

#ifndef _SYS_FITS_IMPL_H
#define _SYS_FITS_IMPL_H

#include <sys/inttypes.h>
#include <sys/types.h>
#include <sys/cmn_err.h>
#include <sys/spa.h>
#include <sys/arc.h>
#include <sys/dsl_dataset.h>
#include <sys/dnode.h>
#include <sys/sa.h>

#define FITS_NO_INO     0

enum pass {
        PASS_LINK,
        PASS_UNLINK
};

typedef struct _fits_count_elem {
        struct _fits_count_elem *fce_next;
        uint64_t fce_ino;
        uint64_t fce_count;
        uint64_t fce_aux;
} fits_count_elem_t;

typedef struct _fits_counter {
        fits_count_elem_t *fc_head;
        const char *fc_name;
} fits_counter_t;

typedef struct blklevel {
        uint64_t        bl_blk;
```



```
        int             bl_nslots;
        blkptr_t        *bl_bp;
        arc_buf_t       *bl_buf;
} blklevel_t;

typedef struct _fits {
        enum pass f_pass;
        struct _fits_ops        *f_ops;
        fits_counter_t f_del_dir_cnt;
        fits_counter_t f_put_back_cnt;
        fits_counter_t f_link_add_cnt;
        uint64_t f_current_ino;
        struct _fits_path *f_current_path;
        int             f_alloc_len;
        uint8_t         *f_buf;
        int             f_size;
        struct vnode    *f_vp;          /* file to which we are reporting */
        offset_t        *f_offp;
        int             f_err;          /* error that stopped diff search */
        dsl_dataset_t   *f_fromds;
        dsl_dataset_t   *f_tods;
        objset_t        *f_fromsnap;
        objset_t        *f_tosnap;
        dnode_phys_t    *f_dnp;
        blklevel_t      *f_bl;
        blklevel_t      *f_filebl;
        uint64_t        f_fromtxg;
        sa_attr_type_t  *f_from_sa_table;
        sa_attr_type_t  *f_to_sa_table;
        uint64_t        f_shares_dir;
} fits_t;

typedef struct _fits_ops {
        int (*fits_dir_add)(fits_t *f, uint64_t ino);
        int (*fits_dir_del)(fits_t *f, uint64_t ino);
        int (*fits_dir_mod)(fits_t *f, uint64_t ino);
        int (*fits_dirent_add)(void *fits_enump, char *name, uint64_t ino);
        int (*fits_dirent_del)(void *fits_enump, char *name, uint64_t ino);
        int (*fits_dirent_mod)(void *fits_enump, char *name,
            uint64_t ino_old, uint64_t ino_new);
        int (*fits_dirent_unmod)(void *fits_enump, char *name, uint64_t ino);
        int (*fits_file_add)(fits_t *f, uint64_t ino);
        int (*fits_file_del)(fits_t *f, uint64_t ino);
        int (*fits_file_mod)(fits_t *f, uint64_t ino);
        int (*fits_file_data)(void *fits_filep, void *data, uint64_t off,
            uint64_t len);
} fits_ops_t;

typedef struct _fits_path {
        struct _fits_path *fp_next;
        int fp_len;
        int fp_total_len;
        char fp_buf[0];
} fits_path_t;

typedef struct _fits_dirent {
        char *fd_name;
        uint64_t fd_parent_ino;
        struct _fits_dirent *fd_prev;
} fits_dirent_t;

typedef enum _fits_which {
        FITS_UNDEF,
        FITS_OLD,
        FITS_NEW
} fits_which_t;
```

```
126 typedef struct _fits_time {
127         uint64_t        st_sec;
128         uint64_t        st_nsec;
129 } fits_time_t;

131 #define FI_ATTR_ATIME           (1 <<  0)
132 #define FI_ATTR_MTIME           (1 <<  1)
133 #define FI_ATTR_CTIME           (1 <<  2)
134 #define FI_ATTR_OTIME           (1 <<  3)
135 #define FI_ATTR_MODE            (1 <<  4)
136 #define FI_ATTR_SIZE            (1 <<  5)
137 #define FI_ATTR_NENTRIES        (1 <<  5)
138 #define FI_ATTR_PARENT          (1 <<  6)
139 #define FI_ATTR_LINKS           (1 <<  7)
140 #define FI_ATTR_RDEV            (1 <<  8)
141 #define FI_ATTR_UID             (1 <<  9)
142 #define FI_ATTR_GID             (1 << 10)
143 #define FI_ATTR_GEN             (1 << 11)
144 /* XXX TODO xattr, acl, dacl */

146 #undef si_uid   /* XXX defined in siginfo.h */
147 #undef si_gid   /* XXX defined in siginfo.h */
148 typedef struct _fits_info {
149         uint64_t        si_nlinks;
150         uint64_t        si_parent;
151         union {
152                 uint64_t        si_nentries;
153                 uint64_t        si_size;
154         };
155         fits_time_t     si_atime;
156         fits_time_t     si_mtime;
157         fits_time_t     si_ctime;
158         fits_time_t     si_otime;
159         uint64_t        si_mode;
160         /* XXX TODO xattr */
161         uint64_t        si_rdev;
162         uint64_t        si_uid;
163         uint64_t        si_gid;
164         uint64_t        si_gen;
165 } fits_info_t;

167 int fits_start(fits_t *f, fits_ops_t **);
168 int fits_start2(fits_t *f, fits_ops_t **);
169 int fits_abort(fits_t *f);
170 int fits_end(fits_t *f);

172 int fits_dirent_add_file(fits_t *f, fits_dirent_t *dirent,
173     uint64_t ino, uint64_t mode, int exists);
174 void fits_path_free(fits_path_t *fp);

176 int fits_get_info(fits_t *f, uint64_t dnobj, fits_which_t which,
177     fits_info_t *sp, uint64_t flags);

179 typedef int (*fits_file_cb_t)(void *ctx, void *data, int len);
180 int fits_file_contents(fits_t *f, uint64_t dnobj, void *ctx);
181 int fits_dir_contents(fits_t *f, uint64_t dnobj, void *ctx);
182 int fits_find_entry(fits_t *f, uint64_t dirobj, uint64_t dnobj,
183     fits_which_t which, char **name);
184 void fits_free_name(char *name);
185 int fits_lookup_entry(fits_t *f, uint64_t dirobj, char *name,
186     fits_which_t which, uint64_t *dnobj);
187 int fits_write(fits_t *f, const uint8_t *data, int len);
188 int fits_get_uuid(fits_t *f, fits_which_t which, uint8_t data[16]);
189 int fits_get_ctransid(fits_t *f, fits_which_t which,
190     uint64_t *ctransid);
```



```
191 int fits_get_snapname(fits_t *f, fits_which_t which,
192     char **name, int *len);
193 int fits_read_symlink(fits_t *f, uint64_t dnobj, fits_which_t which,
194     char **target, int *plen);

196 int fits_send_start(fits_t *f);
197 int fits_send_create_file(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
198 int devise_tempname, fits_path_t **path_ret);
199 int fits_send_link(fits_t *f, fits_dirent_t *new_dirent, uint64_t ino,
200     uint64_t old_parent_ino, fits_which_t which,
201 int devise_tempname);
202 int fits_send_mkdir(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
203 int devise_tempname);
204 int fits_send_rename(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
205     uint64_t old_parent_ino, int devise_tempname);
206 int fits_send_rename_from_tempname(fits_t *f, fits_dirent_t *dirent,
207     uint64_t ino, uint64_t old);
208 int fits_send_unlink(fits_t *f, fits_dirent_t *dirent, uint64_t ino);
209 int fits_send_rmdir(fits_t *f, fits_dirent_t *dirent, uint64_t ino);
210 /* TODO: make **path *path or do we really still alloc it? */
211 int fits_send_file_data(fits_t *f,
212     fits_path_t **path, fits_dirent_t *dirent,
213     uint64_t ino, uint64_t off, uint64_t len, void *data);
214 int fits_send_mtime_update(fits_t *f, fits_dirent_t *dirent, uint64_t ino);
215 int fits_send_truncate(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
216     uint64_t new_size);
217 int fits_send_chown(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
218     uint64_t new_uid, uint64_t new_gid);
219 int fits_send_chmod(fits_t *f, fits_dirent_t *dirent, uint64_t ino,
220     uint64_t new_mode);
221 int fits_send_end(fits_t *f);
222 int fits_add_count(fits_counter_t *fc, uint64_t ino, uint64_t inc,
223     uint64_t aux, uint64_t *new_count, uint64_t *old_aux);
224 void fits_free_count(fits_counter_t *fc, uint64_t ino);
225 int fits_get_count(fits_counter_t *fc, uint64_t ino, uint64_t *new_count,
226     uint64_t *old_aux);
227 int fits_assert_count_empty(fits_counter_t *fc);

229 void fits_send_init(fits_t *f);
230 void fits_send_fini(fits_t *f);

232 #endif /* _SYS_FITS_IMPL_H */
233 #endif /* ! codereview */
```

```
**********************************************************
  145361 Wed Oct 17 21:48:39 2012
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Portions Copyright 2011 Martin Matuska
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

/*
 * ZFS ioctls.
 *
 * This file handles the ioctls to /dev/zfs, used for configuring ZFS storage
 * pools and filesystems, e.g. with /sbin/zfs and /sbin/zpool.
 *
 * There are two ways that we handle ioctls: the legacy way where almost
 * all of the logic is in the ioctl callback, and the new way where most
 * of the marshalling is handled in the common entry point, zfsdev_ioctl().
 *
 * Non-legacy ioctls should be registered by calling
 * zfs_ioctl_register() from zfs_ioctl_init().  The ioctl is invoked
 * from userland by lzc_ioctl().
 *
 * The registration arguments are as follows:
 *
 * const char *name
 *   The name of the ioctl.  This is used for history logging.  If the
 *   ioctl returns successfully (the callback returns 0), and allow_log
 *   is true, then a history log entry will be recorded with the input &
 *   output nvlists.  The log entry can be printed with "zpool history -i".
 *
 * zfs_ioc_t ioc
 *   The ioctl request number, which userland will pass to ioctl(2).
 *   The ioctl numbers can change from release to release, because
 *   the caller (libzfs) must be matched to the kernel.
 *
 * zfs_secpolicy_func_t *secpolicy
 *   This function will be called before the zfs_ioc_func_t, to
```



```
 *   determine if this operation is permitted.  It should return EPERM
 *   on failure, and 0 on success.  Checks include determining if the
 *   dataset is visible in this zone, and if the user has either all
 *   zfs privileges in the zone (SYS_MOUNT), or has been granted permission
 *   to do this operation on this dataset with "zfs allow".
 *
 * zfs_ioc_namecheck_t namecheck
 *   This specifies what to expect in the zfs_cmd_t:zc_name -- a pool
 *   name, a dataset name, or nothing.  If the name is not well-formed,
 *   the ioctl will fail and the callback will not be called.
 *   Therefore, the callback can assume that the name is well-formed
 *   (e.g. is null-terminated, doesn't have more than one '@' character,
 *   doesn't have invalid characters).
 *
 * zfs_ioc_poolcheck_t pool_check
 *   This specifies requirements on the pool state.  If the pool does
 *   not meet them (is suspended or is readonly), the ioctl will fail
 *   and the callback will not be called.  If any checks are specified
 *   (i.e. it is not POOL_CHECK_NONE), namecheck must not be NO_NAME.
 *   Multiple checks can be or-ed together (e.g. POOL_CHECK_SUSPENDED |
 *   POOL_CHECK_READONLY).
 *
 * boolean_t smush_outnvlist
 *   If smush_outnvlist is true, then the output is presumed to be a
 *   list of errors, and it will be "smushed" down to fit into the
 *   caller's buffer, by removing some entries and replacing them with a
 *   single "N_MORE_ERRORS" entry indicating how many were removed.  See
 *   nvlist_smush() for details.  If smush_outnvlist is false, and the
 *   outnvlist does not fit into the userland-provided buffer, then the
 *   ioctl will fail with ENOMEM.
 *
 * zfs_ioc_func_t *func
 *   The callback function that will perform the operation.
 *
 *   The callback should return 0 on success, or an error number on
 *   failure.  If the function fails, the userland ioctl will return -1,
 *   and errno will be set to the callback's return value.  The callback
 *   will be called with the following arguments:
 *
 *   const char *name
 *     The name of the pool or dataset to operate on, from
 *     zfs_cmd_t:zc_name.  The 'namecheck' argument specifies the
 *     expected type (pool, dataset, or none).
 *
 *   nvlist_t *innvl
 *     The input nvlist, deserialized from zfs_cmd_t:zc_nvlist_src.  Or
 *     NULL if no input nvlist was provided.  Changes to this nvlist are
 *     ignored.  If the input nvlist could not be deserialized, the
 *     ioctl will fail and the callback will not be called.
 *
 *   nvlist_t *outnvl
 *     The output nvlist, initially empty.  The callback can fill it in,
 *     and it will be returned to userland by serializing it into
 *     zfs_cmd_t:zc_nvlist_dst.  If it is non-empty, and serialization
 *     fails (e.g. because the caller didn't supply a large enough
 *     buffer), then the overall ioctl will fail.  See the
 *     'smush_nvlist' argument above for additional behaviors.
 *
 *     There are two typical uses of the output nvlist:
 *       - To return state, e.g. property values.  In this case,
 *         smush_outnvlist should be false.  If the buffer was not large
 *         enough, the caller will reallocate a larger buffer and try
 *         the ioctl again.
 *
 *       - To return multiple errors from an ioctl which makes on-disk
 *         changes.  In this case, smush_outnvlist should be true.
```

```
 *         Ioctls which make on-disk modifications should generally not
 *         use the outnvl if they succeed, because the caller can not
 *         distinguish between the operation failing, and
 *         deserialization failing.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/modctl.h>
#include <sys/open.h>
#include <sys/file.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/cmn_err.h>
#include <sys/stat.h>
#include <sys/zfs_ioctl.h>
#include <sys/zfs_vfsops.h>
#include <sys/zfs_znode.h>
#include <sys/zap.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/vdev.h>
#include <sys/priv_impl.h>
#include <sys/dmu.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_deleg.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_impl.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>
#include <sys/policy.h>
#include <sys/zone.h>
#include <sys/nvpair.h>
#include <sys/pathname.h>
#include <sys/mount.h>
#include <sys/sdt.h>
#include <sys/fs/zfs.h>
#include <sys/zfs_ctldir.h>
#include <sys/zfs_dir.h>
#include <sys/zfs_onexit.h>
#include <sys/zvol.h>
#include <sys/dsl_scan.h>
#include <sharefs/share.h>
#include <sys/dmu_objset.h>
#include <sys/fits.h>
#endif /* ! codereview */

#include "zfs_namecheck.h"
#include "zfs_prop.h"
#include "zfs_deleg.h"
#include "zfs_comutil.h"

extern struct modlfs zfs_modlfs;

extern void zfs_init(void);
extern void zfs_fini(void);

ldi_ident_t zfs_li = NULL;
dev_info_t *zfs_dip;
```



```
uint_t zfs_fsyncer_key;
extern uint_t rrw_tsd_key;
static uint_t zfs_allow_log_key;

typedef int zfs_ioc_legacy_func_t(zfs_cmd_t *);
typedef int zfs_ioc_func_t(const char *, nvlist_t *, nvlist_t *);
typedef int zfs_secpolicy_func_t(zfs_cmd_t *, nvlist_t *, cred_t *);

typedef enum {
        NO_NAME,
        POOL_NAME,
        DATASET_NAME
} zfs_ioc_namecheck_t;

typedef enum {
        POOL_CHECK_NONE         = 1 << 0,
        POOL_CHECK_SUSPENDED    = 1 << 1,
        POOL_CHECK_READONLY     = 1 << 2,
} zfs_ioc_poolcheck_t;

typedef struct zfs_ioc_vec {
        zfs_ioc_legacy_func_t   *zvec_legacy_func;
        zfs_ioc_func_t          *zvec_func;
        zfs_secpolicy_func_t    *zvec_secpolicy;
        zfs_ioc_namecheck_t     zvec_namecheck;
        boolean_t               zvec_allow_log;
        zfs_ioc_poolcheck_t     zvec_pool_check;
        boolean_t               zvec_smush_outnvlist;
        const char              *zvec_name;
} zfs_ioc_vec_t;

/* This array is indexed by zfs_userquota_prop_t */
static const char *userquota_perms[] = {
        ZFS_DELEG_PERM_USERUSED,
        ZFS_DELEG_PERM_USERQUOTA,
        ZFS_DELEG_PERM_GROUPUSED,
        ZFS_DELEG_PERM_GROUPQUOTA,
};

static int zfs_ioc_userspace_upgrade(zfs_cmd_t *zc);
static int zfs_check_settable(const char *name, nvpair_t *property,
    cred_t *cr);
static int zfs_check_clearable(char *dataset, nvlist_t *props,
    nvlist_t **errors);
static int zfs_fill_zplprops_root(uint64_t, nvlist_t *, nvlist_t *,
    boolean_t *);
int zfs_set_prop_nvlist(const char *, zprop_source_t, nvlist_t *, nvlist_t *);
static int get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp);

/* _NOTE(PRINTFLIKE(4)) - this is printf-like, but lint is too whiney */
void
__dprintf(const char *file, const char *func, int line, const char *fmt, ...)
{
        const char *newfile;
        char buf[512];
        va_list adx;

        /*
         * Get rid of annoying "../common/" prefix to filename.
         */
        newfile = strrchr(file, '/');
        if (newfile != NULL) {
                newfile = newfile + 1; /* Get rid of leading / */
        } else {
                newfile = file;
        }
```

```
258         va_start(adx, fmt);
259         (void) vsnprintf(buf, sizeof (buf), fmt, adx);
260         va_end(adx);

262         /*
263          * To get this data, use the zfs-dprintf probe as so:
264          * dtrace -q -n 'zfs-dprintf \
265          *      /stringof(arg0) == "dbuf.c"/ \
266          *      {printf("%s: %s", stringof(arg1), stringof(arg3))}'
267          * arg0 = file name
268          * arg1 = function name
269          * arg2 = line number
270          * arg3 = message
271          */
272         DTRACE_PROBE4(zfs__dprintf,
273             char *, newfile, char *, func, int, line, char *, buf);
274 }

276 static void
277 history_str_free(char *buf)
278 {
279         kmem_free(buf, HIS_MAX_RECORD_LEN);
280 }

282 static char *
283 history_str_get(zfs_cmd_t *zc)
284 {
285         char *buf;

287         if (zc->zc_history == NULL)
288                 return (NULL);

290         buf = kmem_alloc(HIS_MAX_RECORD_LEN, KM_SLEEP);
291         if (copyinstr((void *)(uintptr_t)zc->zc_history,
292             buf, HIS_MAX_RECORD_LEN, NULL) != 0) {
293                 history_str_free(buf);
294                 return (NULL);
295         }

297         buf[HIS_MAX_RECORD_LEN -1] = '\0';

299         return (buf);
300 }

302 /*
303  * Check to see if the named dataset is currently defined as bootable
304  */
305 static boolean_t
306 zfs_is_bootfs(const char *name)
307 {
308         objset_t *os;

310         if (dmu_objset_hold(name, FTAG, &os) == 0) {
311                 boolean_t ret;
312                 ret = (dmu_objset_id(os) == spa_bootfs(dmu_objset_spa(os)));
313                 dmu_objset_rele(os, FTAG);
314                 return (ret);
315         }
316         return (B_FALSE);
317 }

319 /*
320  * zfs_earlier_version
321  *
322  *      Return non-zero if the spa version is less than requested version.
```



```
323  */
324 static int
325 zfs_earlier_version(const char *name, int version)
326 {
327         spa_t *spa;

329         if (spa_open(name, &spa, FTAG) == 0) {
330                 if (spa_version(spa) < version) {
331                         spa_close(spa, FTAG);
332                         return (1);
333                 }
334                 spa_close(spa, FTAG);
335         }
336         return (0);
337 }

339 /*
340  * zpl_earlier_version
341  *
342  * Return TRUE if the ZPL version is less than requested version.
343  */
344 static boolean_t
345 zpl_earlier_version(const char *name, int version)
346 {
347         objset_t *os;
348         boolean_t rc = B_TRUE;

350         if (dmu_objset_hold(name, FTAG, &os) == 0) {
351                 uint64_t zplversion;

353                 if (dmu_objset_type(os) != DMU_OST_ZFS) {
354                         dmu_objset_rele(os, FTAG);
355                         return (B_TRUE);
356                 }
357                 /* XXX reading from non-owned objset */
358                 if (zfs_get_zplprop(os, ZFS_PROP_VERSION, &zplversion) == 0)
359                         rc = zplversion < version;
360                 dmu_objset_rele(os, FTAG);
361         }
362         return (rc);
363 }

365 static void
366 zfs_log_history(zfs_cmd_t *zc)
367 {
368         spa_t *spa;
369         char *buf;

371         if ((buf = history_str_get(zc)) == NULL)
372                 return;

374         if (spa_open(zc->zc_name, &spa, FTAG) == 0) {
375                 if (spa_version(spa) >= SPA_VERSION_ZPOOL_HISTORY)
376                         (void) spa_history_log(spa, buf);
377                 spa_close(spa, FTAG);
378         }
379         history_str_free(buf);
380 }

382 /*
383  * Policy for top-level read operations (list pools).  Requires no privileges,
384  * and can be used in the local zone, as there is no associated dataset.
385  */
386 /* ARGSUSED */
387 static int
388 zfs_secpolicy_none(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
```

```
389 {
390         return (0);
391 }

393 /*
394  * Policy for dataset read operations (list children, get statistics).  Requires
395  * no privileges, but must be visible in the local zone.
396  */
397 /* ARGSUSED */
398 static int
399 zfs_secpolicy_read(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
400 {
401         if (INGLOBALZONE(curproc) ||
402             zone_dataset_visible(zc->zc_name, NULL))
403                 return (0);

405         return (ENOENT);
406 }

408 static int
409 zfs_dozonecheck_impl(const char *dataset, uint64_t zoned, cred_t *cr)
410 {
411         int writable = 1;

413         /*
414          * The dataset must be visible by this zone -- check this first
415          * so they don't see EPERM on something they shouldn't know about.
416          */
417         if (!INGLOBALZONE(curproc) &&
418             !zone_dataset_visible(dataset, &writable))
419                 return (ENOENT);

421         if (INGLOBALZONE(curproc)) {
422                 /*
423                  * If the fs is zoned, only root can access it from the
424                  * global zone.
425                  */
426                 if (secpolicy_zfs(cr) && zoned)
427                         return (EPERM);
428         } else {
429                 /*
430                  * If we are in a local zone, the 'zoned' property must be set.
431                  */
432                 if (!zoned)
433                         return (EPERM);

435                 /* must be writable by this zone */
436                 if (!writable)
437                         return (EPERM);
438         }
439         return (0);
440 }

442 static int
443 zfs_dozonecheck(const char *dataset, cred_t *cr)
444 {
445         uint64_t zoned;

447         if (dsl_prop_get_integer(dataset, "zoned", &zoned, NULL))
448                 return (ENOENT);

450         return (zfs_dozonecheck_impl(dataset, zoned, cr));
451 }

453 static int
454 zfs_dozonecheck_ds(const char *dataset, dsl_dataset_t *ds, cred_t *cr)
```



```
455 {
456         uint64_t zoned;

458         rw_enter(&ds->ds_dir->dd_pool->dp_config_rwlock, RW_READER);
459         if (dsl_prop_get_ds(ds, "zoned", 8, 1, &zoned, NULL)) {
460                 rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);
461                 return (ENOENT);
462         }
463         rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);

465         return (zfs_dozonecheck_impl(dataset, zoned, cr));
466 }

468 static int
469 zfs_secpolicy_write_perms(const char *name, const char *perm, cred_t *cr)
470 {
471         int error;
472         dsl_dataset_t *ds;

474         error = dsl_dataset_hold(name, FTAG, &ds);
475         if (error != 0)
476                 return (error);

478         error = zfs_dozonecheck_ds(name, ds, cr);
479         if (error == 0) {
480                 error = secpolicy_zfs(cr);
481                 if (error)
482                         error = dsl_deleg_access_impl(ds, perm, cr);
483         }

485         dsl_dataset_rele(ds, FTAG);
486         return (error);
487 }

489 static int
490 zfs_secpolicy_write_perms_ds(const char *name, dsl_dataset_t *ds,
491     const char *perm, cred_t *cr)
492 {
493         int error;

495         error = zfs_dozonecheck_ds(name, ds, cr);
496         if (error == 0) {
497                 error = secpolicy_zfs(cr);
498                 if (error)
499                         error = dsl_deleg_access_impl(ds, perm, cr);
500         }
501         return (error);
502 }

504 /*
505  * Policy for setting the security label property.
506  *
507  * Returns 0 for success, non-zero for access and other errors.
508  */
509 static int
510 zfs_set_slabel_policy(const char *name, char *strval, cred_t *cr)
511 {
512         char            ds_hexsl[MAXNAMELEN];
513         bslabel_t       ds_sl, new_sl;
514         boolean_t       new_default = FALSE;
515         uint64_t        zoned;
516         int             needed_priv = -1;
517         int             error;

519         /* First get the existing dataset label. */
520         error = dsl_prop_get(name, zfs_prop_to_name(ZFS_PROP_MLSLABEL),
```

```
521             1, sizeof (ds_hexsl), &ds_hexsl, NULL);
522         if (error)
523                 return (EPERM);

525         if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
526                 new_default = TRUE;

528         /* The label must be translatable */
529         if (!new_default && (hexstr_to_label(strval, &new_sl) != 0))
530                 return (EINVAL);

532         /*
533          * In a non-global zone, disallow attempts to set a label that
534          * doesn't match that of the zone; otherwise no other checks
535          * are needed.
536          */
537         if (!INGLOBALZONE(curproc)) {
538                 if (new_default || !blequal(&new_sl, CR_SL(CRED())))
539                         return (EPERM);
540                 return (0);
541         }

543         /*
544          * For global-zone datasets (i.e., those whose zoned property is
545          * "off", verify that the specified new label is valid for the
546          * global zone.
547          */
548         if (dsl_prop_get_integer(name,
549             zfs_prop_to_name(ZFS_PROP_ZONED), &zoned, NULL))
550                 return (EPERM);
551         if (!zoned) {
552                 if (zfs_check_global_label(name, strval) != 0)
553                         return (EPERM);
554         }

556         /*
557          * If the existing dataset label is nondefault, check if the
558          * dataset is mounted (label cannot be changed while mounted).
559          * Get the zfsvfs; if there isn't one, then the dataset isn't
560          * mounted (or isn't a dataset, doesn't exist, ...).
561          */
562         if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) != 0) {
563                 objset_t *os;
564                 static char *setsl_tag = "setsl_tag";

566                 /*
567                  * Try to own the dataset; abort if there is any error,
568                  * (e.g., already mounted, in use, or other error).
569                  */
570                 error = dmu_objset_own(name, DMU_OST_ZFS, B_TRUE,
571                     setsl_tag, &os);
572                 if (error)
573                         return (EPERM);

575                 dmu_objset_disown(os, setsl_tag);

577                 if (new_default) {
578                         needed_priv = PRIV_FILE_DOWNGRADE_SL;
579                         goto out_check;
580                 }

582                 if (hexstr_to_label(strval, &new_sl) != 0)
583                         return (EPERM);

585                 if (blstrictdom(&ds_sl, &new_sl))
586                         needed_priv = PRIV_FILE_DOWNGRADE_SL;
```



```
587                 else if (blstrictdom(&new_sl, &ds_sl))
588                         needed_priv = PRIV_FILE_UPGRADE_SL;
589         } else {
590                 /* dataset currently has a default label */
591                 if (!new_default)
592                         needed_priv = PRIV_FILE_UPGRADE_SL;
593         }

595 out_check:
596         if (needed_priv != -1)
597                 return (PRIV_POLICY(cr, needed_priv, B_FALSE, EPERM, NULL));
598         return (0);
599 }

601 static int
602 zfs_secpolicy_setprop(const char *dsname, zfs_prop_t prop, nvpair_t *propval,
603     cred_t *cr)
604 {
605         char *strval;

607         /*
608          * Check permissions for special properties.
609          */
610         switch (prop) {
611         case ZFS_PROP_ZONED:
612                 /*
613                  * Disallow setting of 'zoned' from within a local zone.
614                  */
615                 if (!INGLOBALZONE(curproc))
616                         return (EPERM);
617                 break;

619         case ZFS_PROP_QUOTA:
620                 if (!INGLOBALZONE(curproc)) {
621                         uint64_t zoned;
622                         char setpoint[MAXNAMELEN];
623                         /*
624                          * Unprivileged users are allowed to modify the
625                          * quota on things *under* (ie. contained by)
626                          * the thing they own.
627                          */
628                         if (dsl_prop_get_integer(dsname, "zoned", &zoned,
629                             setpoint))
630                                 return (EPERM);
631                         if (!zoned || strlen(dsname) <= strlen(setpoint))
632                                 return (EPERM);
633                 }
634                 break;

636         case ZFS_PROP_MLSLABEL:
637                 if (!is_system_labeled())
638                         return (EPERM);

640                 if (nvpair_value_string(propval, &strval) == 0) {
641                         int err;

643                         err = zfs_set_slabel_policy(dsname, strval, CRED());
644                         if (err != 0)
645                                 return (err);
646                 }
647                 break;
648         }

650         return (zfs_secpolicy_write_perms(dsname, zfs_prop_to_name(prop), cr));
651 }
```

```
/* ARGSUSED */
static int
zfs_secpolicy_set_fsacl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        int error;

        error = zfs_dozonecheck(zc->zc_name, cr);
        if (error)
                return (error);

        /*
         * permission to set permissions will be evaluated later in
         * dsl_deleg_can_allow()
         */
        return (0);
}

/* ARGSUSED */
static int
zfs_secpolicy_rollback(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        return (zfs_secpolicy_write_perms(zc->zc_name,
            ZFS_DELEG_PERM_ROLLBACK, cr));
}

/* ARGSUSED */
static int
zfs_secpolicy_send(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        spa_t *spa;
        dsl_pool_t *dp;
        dsl_dataset_t *ds;
        char *cp;
        int error;

        /*
         * Generate the current snapshot name from the given objsetid, then
         * use that name for the secpolicy/zone checks.
         */
        cp = strchr(zc->zc_name, '@');
        if (cp == NULL)
                return (EINVAL);
        error = spa_open(zc->zc_name, &spa, FTAG);
        if (error)
                return (error);

        dp = spa_get_dsl(spa);
        rw_enter(&dp->dp_config_rwlock, RW_READER);
        error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
        rw_exit(&dp->dp_config_rwlock);
        spa_close(spa, FTAG);
        if (error)
                return (error);

        dsl_dataset_name(ds, zc->zc_name);

        error = zfs_secpolicy_write_perms_ds(zc->zc_name, ds,
            ZFS_DELEG_PERM_SEND, cr);
        dsl_dataset_rele(ds, FTAG);

        return (error);
}

/* ARGSUSED */
static int
zfs_secpolicy_send_new(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
```



```
{
        return (zfs_secpolicy_write_perms(zc->zc_name,
            ZFS_DELEG_PERM_SEND, cr));
}

/* ARGSUSED */
static int
zfs_secpolicy_deleg_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        vnode_t *vp;
        int error;

        if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
            NO_FOLLOW, NULL, &vp)) != 0)
                return (error);

        /* Now make sure mntpnt and dataset are ZFS */

        if (vp->v_vfsp->vfs_fstype != zfsfstype ||
            (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
            zc->zc_name) != 0)) {
                VN_RELE(vp);
                return (EPERM);
        }

        VN_RELE(vp);
        return (dsl_deleg_access(zc->zc_name,
            ZFS_DELEG_PERM_SHARE, cr));
}

int
zfs_secpolicy_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        if (!INGLOBALZONE(curproc))
                return (EPERM);

        if (secpolicy_nfs(cr) == 0) {
                return (0);
        } else {
                return (zfs_secpolicy_deleg_share(zc, innvl, cr));
        }
}

int
zfs_secpolicy_smb_acl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        if (!INGLOBALZONE(curproc))
                return (EPERM);

        if (secpolicy_smb(cr) == 0) {
                return (0);
        } else {
                return (zfs_secpolicy_deleg_share(zc, innvl, cr));
        }
}

static int
zfs_get_parent(const char *datasetname, char *parent, int parentsize)
{
        char *cp;

        /*
         * Remove the @bla or /bla from the end of the name to get the parent.
         */
        (void) strncpy(parent, datasetname, parentsize);
        cp = strrchr(parent, '@');
```

```
785         if (cp != NULL) {
786                 cp[0] = '\0';
787         } else {
788                 cp = strrchr(parent, '/');
789                 if (cp == NULL)
790                         return (ENOENT);
791                 cp[0] = '\0';
792         }
793
794         return (0);
795 }
796
797 int
798 zfs_secpolicy_destroy_perms(const char *name, cred_t *cr)
799 {
800         int error;
801
802         if ((error = zfs_secpolicy_write_perms(name,
803             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
804                 return (error);
805
806         return (zfs_secpolicy_write_perms(name, ZFS_DELEG_PERM_DESTROY, cr));
807 }
808
809 /* ARGSUSED */
810 static int
811 zfs_secpolicy_destroy(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
812 {
813         return (zfs_secpolicy_destroy_perms(zc->zc_name, cr));
814 }
815
816 /*
817  * Destroying snapshots with delegated permissions requires
818  * descendant mount and destroy permissions.
819  */
820 /* ARGSUSED */
821 static int
822 zfs_secpolicy_destroy_snaps(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
823 {
824         nvlist_t *snaps;
825         nvpair_t *pair, *nextpair;
826         int error = 0;
827
828         if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
829                 return (EINVAL);
830         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
831             pair = nextpair) {
832                 dsl_dataset_t *ds;
833
834                 nextpair = nvlist_next_nvpair(snaps, pair);
835                 error = dsl_dataset_hold(nvpair_name(pair), FTAG, &ds);
836                 if (error == 0) {
837                         dsl_dataset_rele(ds, FTAG);
838                 } else if (error == ENOENT) {
839                         /*
840                          * Ignore any snapshots that don't exist (we consider
841                          * them "already destroyed").  Remove the name from the
842                          * nvl here in case the snapshot is created between
843                          * now and when we try to destroy it (in which case
844                          * we don't want to destroy it since we haven't
845                          * checked for permission).
846                          */
847                         fnvlist_remove_nvpair(snaps, pair);
848                         error = 0;
849                         continue;
850                 } else {
```



```
851                         break;
852                 }
853                 error = zfs_secpolicy_destroy_perms(nvpair_name(pair), cr);
854                 if (error != 0)
855                         break;
856         }
857
858         return (error);
859 }
860
861 int
862 zfs_secpolicy_rename_perms(const char *from, const char *to, cred_t *cr)
863 {
864         char    parentname[MAXNAMELEN];
865         int     error;
866
867         if ((error = zfs_secpolicy_write_perms(from,
868             ZFS_DELEG_PERM_RENAME, cr)) != 0)
869                 return (error);
870
871         if ((error = zfs_secpolicy_write_perms(from,
872             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
873                 return (error);
874
875         if ((error = zfs_get_parent(to, parentname,
876             sizeof (parentname))) != 0)
877                 return (error);
878
879         if ((error = zfs_secpolicy_write_perms(parentname,
880             ZFS_DELEG_PERM_CREATE, cr)) != 0)
881                 return (error);
882
883         if ((error = zfs_secpolicy_write_perms(parentname,
884             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
885                 return (error);
886
887         return (error);
888 }
889
890 /* ARGSUSED */
891 static int
892 zfs_secpolicy_rename(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
893 {
894         return (zfs_secpolicy_rename_perms(zc->zc_name, zc->zc_value, cr));
895 }
896
897 /* ARGSUSED */
898 static int
899 zfs_secpolicy_promote(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
900 {
901         char    parentname[MAXNAMELEN];
902         objset_t *clone;
903         int error;
904
905         error = zfs_secpolicy_write_perms(zc->zc_name,
906             ZFS_DELEG_PERM_PROMOTE, cr);
907         if (error)
908                 return (error);
909
910         error = dmu_objset_hold(zc->zc_name, FTAG, &clone);
911
912         if (error == 0) {
913                 dsl_dataset_t *pclone = NULL;
914                 dsl_dir_t *dd;
915                 dd = clone->os_dsl_dataset->ds_dir;
```

```
917                 rw_enter(&dd->dd_pool->dp_config_rwlock, RW_READER);
918                 error = dsl_dataset_hold_obj(dd->dd_pool,
919                     dd->dd_phys->dd_origin_obj, FTAG, &pclone);
920                 rw_exit(&dd->dd_pool->dp_config_rwlock);
921                 if (error) {
922                         dmu_objset_rele(clone, FTAG);
923                         return (error);
924                 }
926                 error = zfs_secpolicy_write_perms(zc->zc_name,
927                     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 static int
941 zfs_secpolicy_recv(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
942 {
943         int error;
945         if ((error = zfs_secpolicy_write_perms(zc->zc_name,
946             ZFS_DELEG_PERM_RECEIVE, cr)) != 0)
947                 return (error);
949         if ((error = zfs_secpolicy_write_perms(zc->zc_name,
950             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
951                 return (error);
953         return (zfs_secpolicy_write_perms(zc->zc_name,
954             ZFS_DELEG_PERM_CREATE, cr));
955 }
957 int
958 zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr)
959 {
960         return (zfs_secpolicy_write_perms(name,
961             ZFS_DELEG_PERM_SNAPSHOT, cr));
962 }
964 /*
965  * Check for permission to create each snapshot in the nvlist.
966  */
967 /* ARGSUSED */
968 static int
969 zfs_secpolicy_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
970 {
971         nvlist_t *snaps;
972         int error;
973         nvpair_t *pair;
975         if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
976                 return (EINVAL);
977         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
978             pair = nvlist_next_nvpair(snaps, pair)) {
979                 char *name = nvpair_name(pair);
980                 char *atp = strchr(name, '@');
982                 if (atp == NULL) {
```



```
983                         error = EINVAL;
984                         break;
985                 }
986                 *atp = '\0';
987                 error = zfs_secpolicy_snapshot_perms(name, cr);
988                 *atp = '@';
989                 if (error != 0)
990                         break;
991         }
992         return (error);
993 }
995 /* ARGSUSED */
996 static int
997 zfs_secpolicy_log_history(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
998 {
999         /*
1000          * Even root must have a proper TSD so that we know what pool
1001          * to log to.
1002          */
1003         if (tsd_get(zfs_allow_log_key) == NULL)
1004                 return (EPERM);
1005         return (0);
1006 }
1008 static int
1009 zfs_secpolicy_create_clone(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1010 {
1011         char    parentname[MAXNAMELEN];
1012         int     error;
1013         char    *origin;
1015         if ((error = zfs_get_parent(zc->zc_name, parentname,
1016             sizeof (parentname))) != 0)
1017                 return (error);
1019         if (nvlist_lookup_string(innvl, "origin", &origin) == 0 &&
1020             (error = zfs_secpolicy_write_perms(origin,
1021             ZFS_DELEG_PERM_CLONE, cr)) != 0)
1022                 return (error);
1024         if ((error = zfs_secpolicy_write_perms(parentname,
1025             ZFS_DELEG_PERM_CREATE, cr)) != 0)
1026                 return (error);
1028         return (zfs_secpolicy_write_perms(parentname,
1029             ZFS_DELEG_PERM_MOUNT, cr));
1030 }
1032 /*
1033  * Policy for pool operations - create/destroy pools, add vdevs, etc.  Requires
1034  * SYS_CONFIG privilege, which is not available in a local zone.
1035  */
1036 /* ARGSUSED */
1037 static int
1038 zfs_secpolicy_config(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1039 {
1040         if (secpolicy_sys_config(cr, B_FALSE) != 0)
1041                 return (EPERM);
1043         return (0);
1044 }
1046 /*
1047  * Policy for object to name lookups.
1048  */
```

```
1049 /* ARGSUSED */
1050 static int
1051 zfs_secpolicy_diff(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1052 {
1053         int error;

1055         if ((error = secpolicy_sys_config(cr, B_FALSE)) == 0)
1056                 return (0);

1058         error = zfs_secpolicy_write_perms(zc->zc_name, ZFS_DELEG_PERM_DIFF, cr);
1059         return (error);
1060 }

1062 /*
1063  * Policy for fault injection.  Requires all privileges.
1064  */
1065 /* ARGSUSED */
1066 static int
1067 zfs_secpolicy_inject(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1068 {
1069         return (secpolicy_zinject(cr));
1070 }

1072 /* ARGSUSED */
1073 static int
1074 zfs_secpolicy_inherit_prop(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1075 {
1076         zfs_prop_t prop = zfs_name_to_prop(zc->zc_value);

1078         if (prop == ZPROP_INVAL) {
1079                 if (!zfs_prop_user(zc->zc_value))
1080                         return (EINVAL);
1081                 return (zfs_secpolicy_write_perms(zc->zc_name,
1082                     ZFS_DELEG_PERM_USERPROP, cr));
1083         } else {
1084                 return (zfs_secpolicy_setprop(zc->zc_name, prop,
1085                     NULL, cr));
1086         }
1087 }

1089 static int
1090 zfs_secpolicy_userspace_one(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1091 {
1092         int err = zfs_secpolicy_read(zc, innvl, cr);
1093         if (err)
1094                 return (err);

1096         if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
1097                 return (EINVAL);

1099         if (zc->zc_value[0] == 0) {
1100                 /*
1101                  * They are asking about a posix uid/gid.  If it's
1102                  * themself, allow it.
1103                  */
1104                 if (zc->zc_objset_type == ZFS_PROP_USERUSED ||
1105                     zc->zc_objset_type == ZFS_PROP_USERQUOTA) {
1106                         if (zc->zc_guid == crgetuid(cr))
1107                                 return (0);
1108                 } else {
1109                         if (groupmember(zc->zc_guid, cr))
1110                                 return (0);
1111                 }
1112         }

1114         return (zfs_secpolicy_write_perms(zc->zc_name,
```



```
1115             userquota_perms[zc->zc_objset_type], cr));
1116 }

1118 static int
1119 zfs_secpolicy_userspace_many(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1120 {
1121         int err = zfs_secpolicy_read(zc, innvl, cr);
1122         if (err)
1123                 return (err);

1125         if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
1126                 return (EINVAL);

1128         return (zfs_secpolicy_write_perms(zc->zc_name,
1129             userquota_perms[zc->zc_objset_type], cr));
1130 }

1132 /* ARGSUSED */
1133 static int
1134 zfs_secpolicy_userspace_upgrade(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1135 {
1136         return (zfs_secpolicy_setprop(zc->zc_name, ZFS_PROP_VERSION,
1137             NULL, cr));
1138 }

1140 /* ARGSUSED */
1141 static int
1142 zfs_secpolicy_hold(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1143 {
1144         return (zfs_secpolicy_write_perms(zc->zc_name,
1145             ZFS_DELEG_PERM_HOLD, cr));
1146 }

1148 /* ARGSUSED */
1149 static int
1150 zfs_secpolicy_release(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1151 {
1152         return (zfs_secpolicy_write_perms(zc->zc_name,
1153             ZFS_DELEG_PERM_RELEASE, cr));
1154 }

1156 /*
1157  * Policy for allowing temporary snapshots to be taken or released
1158  */
1159 static int
1160 zfs_secpolicy_tmp_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1161 {
1162         /*
1163          * A temporary snapshot is the same as a snapshot,
1164          * hold, destroy and release all rolled into one.
1165          * Delegated diff alone is sufficient that we allow this.
1166          */
1167         int error;

1169         if ((error = zfs_secpolicy_write_perms(zc->zc_name,
1170             ZFS_DELEG_PERM_DIFF, cr)) == 0)
1171                 return (0);

1173         error = zfs_secpolicy_snapshot_perms(zc->zc_name, cr);
1174         if (!error)
1175                 error = zfs_secpolicy_hold(zc, innvl, cr);
1176         if (!error)
1177                 error = zfs_secpolicy_release(zc, innvl, cr);
1178         if (!error)
1179                 error = zfs_secpolicy_destroy(zc, innvl, cr);
1180         return (error);
```

```
}

/*
 * Returns the nvlist as specified by the user in the zfs_cmd_t.
 */
static int
get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp)
{
        char *packed;
        int error;
        nvlist_t *list = NULL;

        /*
         * Read in and unpack the user-supplied nvlist.
         */
        if (size == 0)
                return (EINVAL);

        packed = kmem_alloc(size, KM_SLEEP);

        if ((error = ddi_copyin((void *)(uintptr_t)nvl, packed, size,
            iflag)) != 0) {
                kmem_free(packed, size);
                return (error);
        }

        if ((error = nvlist_unpack(packed, size, &list, 0)) != 0) {
                kmem_free(packed, size);
                return (error);
        }

        kmem_free(packed, size);

        *nvp = list;
        return (0);
}

/*
 * Reduce the size of this nvlist until it can be serialized in 'max' bytes.
 * Entries will be removed from the end of the nvlist, and one int32 entry
 * named "N_MORE_ERRORS" will be added indicating how many entries were
 * removed.
 */
static int
nvlist_smush(nvlist_t *errors, size_t max)
{
        size_t size;

        size = fnvlist_size(errors);

        if (size > max) {
                nvpair_t *more_errors;
                int n = 0;

                if (max < 1024)
                        return (ENOMEM);

                fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, 0);
                more_errors = nvlist_prev_nvpair(errors, NULL);

                do {
                        nvpair_t *pair = nvlist_prev_nvpair(errors,
                            more_errors);
                        fnvlist_remove_nvpair(errors, pair);
                        n++;
                        size = fnvlist_size(errors);
```



```
                } while (size > max);

                fnvlist_remove_nvpair(errors, more_errors);
                fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, n);
                ASSERT3U(fnvlist_size(errors), <=, max);
        }

        return (0);
}

static int
put_nvlist(zfs_cmd_t *zc, nvlist_t *nvl)
{
        char *packed = NULL;
        int error = 0;
        size_t size;

        size = fnvlist_size(nvl);

        if (size > zc->zc_nvlist_dst_size) {
                error = ENOMEM;
        } else {
                packed = fnvlist_pack(nvl, &size);
                if (ddi_copyout(packed, (void *)(uintptr_t)zc->zc_nvlist_dst,
                    size, zc->zc_iflags) != 0)
                        error = EFAULT;
                fnvlist_pack_free(packed, size);
        }

        zc->zc_nvlist_dst_size = size;
        zc->zc_nvlist_dst_filled = B_TRUE;
        return (error);
}

static int
getzfsvfs(const char *dsname, zfsvfs_t **zfvp)
{
        objset_t *os;
        int error;

        error = dmu_objset_hold(dsname, FTAG, &os);
        if (error)
                return (error);
        if (dmu_objset_type(os) != DMU_OST_ZFS) {
                dmu_objset_rele(os, FTAG);
                return (EINVAL);
        }

        mutex_enter(&os->os_user_ptr_lock);
        *zfvp = dmu_objset_get_user(os);
        if (*zfvp) {
                VFS_HOLD((*zfvp)->z_vfs);
        } else {
                error = ESRCH;
        }
        mutex_exit(&os->os_user_ptr_lock);
        dmu_objset_rele(os, FTAG);
        return (error);
}

/*
 * Find a zfsvfs_t for a mounted filesystem, or create our own, in which
 * case its z_vfs will be NULL, and it will be opened as the owner.
 * If 'writer' is set, the z_teardown_lock will be held for RW_WRITER,
 * which prevents all vnode ops from running.
 */
```

```
1313 static int
1314 zfsvfs_hold(const char *name, void *tag, zfsvfs_t **zfvp, boolean_t writer)
1315 {
1316         int error = 0;

1318         if (getzfsvfs(name, zfvp) != 0)
1319                 error = zfsvfs_create(name, zfvp);
1320         if (error == 0) {
1321                 rrw_enter(&(*zfvp)->z_teardown_lock, (writer) ? RW_WRITER :
1322                     RW_READER, tag);
1323                 if ((*zfvp)->z_unmounted) {
1324                         /*
1325                          * XXX we could probably try again, since the unmounting
1326                          * thread should be just about to disassociate the
1327                          * objset from the zfsvfs.
1328                          */
1329                         rrw_exit(&(*zfvp)->z_teardown_lock, tag);
1330                         return (EBUSY);
1331                 }
1332         }
1333         return (error);
1334 }

1336 static void
1337 zfsvfs_rele(zfsvfs_t *zfsvfs, void *tag)
1338 {
1339         rrw_exit(&zfsvfs->z_teardown_lock, tag);

1341         if (zfsvfs->z_vfs) {
1342                 VFS_RELE(zfsvfs->z_vfs);
1343         } else {
1344                 dmu_objset_disown(zfsvfs->z_os, zfsvfs);
1345                 zfsvfs_free(zfsvfs);
1346         }
1347 }

1349 static int
1350 zfs_ioc_pool_create(zfs_cmd_t *zc)
1351 {
1352         int error;
1353         nvlist_t *config, *props = NULL;
1354         nvlist_t *rootprops = NULL;
1355         nvlist_t *zplprops = NULL;

1357         if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1358             zc->zc_iflags, &config))
1359                 return (error);

1361         if (zc->zc_nvlist_src_size != 0 && (error =
1362             get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1363             zc->zc_iflags, &props))) {
1364                 nvlist_free(config);
1365                 return (error);
1366         }

1368         if (props) {
1369                 nvlist_t *nvl = NULL;
1370                 uint64_t version = SPA_VERSION;

1372                 (void) nvlist_lookup_uint64(props,
1373                     zpool_prop_to_name(ZPOOL_PROP_VERSION), &version);
1374                 if (!SPA_VERSION_IS_SUPPORTED(version)) {
1375                         error = EINVAL;
1376                         goto pool_props_bad;
1377                 }
1378                 (void) nvlist_lookup_nvlist(props, ZPOOL_ROOTFS_PROPS, &nvl);
```



```
1379                 if (nvl) {
1380                         error = nvlist_dup(nvl, &rootprops, KM_SLEEP);
1381                         if (error != 0) {
1382                                 nvlist_free(config);
1383                                 nvlist_free(props);
1384                                 return (error);
1385                         }
1386                         (void) nvlist_remove_all(props, ZPOOL_ROOTFS_PROPS);
1387                 }
1388                 VERIFY(nvlist_alloc(&zplprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);
1389                 error = zfs_fill_zplprops_root(version, rootprops,
1390                     zplprops, NULL);
1391                 if (error)
1392                         goto pool_props_bad;
1393         }

1395         error = spa_create(zc->zc_name, config, props, zplprops);

1397         /*
1398          * Set the remaining root properties
1399          */
1400         if (!error && (error = zfs_set_prop_nvlist(zc->zc_name,
1401             ZPROP_SRC_LOCAL, rootprops, NULL)) != 0)
1402                 (void) spa_destroy(zc->zc_name);

1404 pool_props_bad:
1405         nvlist_free(rootprops);
1406         nvlist_free(zplprops);
1407         nvlist_free(config);
1408         nvlist_free(props);

1410         return (error);
1411 }

1413 static int
1414 zfs_ioc_pool_destroy(zfs_cmd_t *zc)
1415 {
1416         int error;
1417         zfs_log_history(zc);
1418         error = spa_destroy(zc->zc_name);
1419         if (error == 0)
1420                 zvol_remove_minors(zc->zc_name);
1421         return (error);
1422 }

1424 static int
1425 zfs_ioc_pool_import(zfs_cmd_t *zc)
1426 {
1427         nvlist_t *config, *props = NULL;
1428         uint64_t guid;
1429         int error;

1431         if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1432             zc->zc_iflags, &config)) != 0)
1433                 return (error);

1435         if (zc->zc_nvlist_src_size != 0 && (error =
1436             get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1437             zc->zc_iflags, &props))) {
1438                 nvlist_free(config);
1439                 return (error);
1440         }

1442         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
1443             guid != zc->zc_guid)
1444                 error = EINVAL;
```

```
1445         else
1446                 error = spa_import(zc->zc_name, config, props, zc->zc_cookie);
1447
1448         if (zc->zc_nvlist_dst != 0) {
1449                 int err;
1450
1451                 if ((err = put_nvlist(zc, config)) != 0)
1452                         error = err;
1453         }
1454
1455         nvlist_free(config);
1456
1457         if (props)
1458                 nvlist_free(props);
1459
1460         return (error);
1461 }
1462
1463 static int
1464 zfs_ioc_pool_export(zfs_cmd_t *zc)
1465 {
1466         int error;
1467         boolean_t force = (boolean_t)zc->zc_cookie;
1468         boolean_t hardforce = (boolean_t)zc->zc_guid;
1469
1470         zfs_log_history(zc);
1471         error = spa_export(zc->zc_name, NULL, force, hardforce);
1472         if (error == 0)
1473                 zvol_remove_minors(zc->zc_name);
1474         return (error);
1475 }
1476
1477 static int
1478 zfs_ioc_pool_configs(zfs_cmd_t *zc)
1479 {
1480         nvlist_t *configs;
1481         int error;
1482
1483         if ((configs = spa_all_configs(&zc->zc_cookie)) == NULL)
1484                 return (EEXIST);
1485
1486         error = put_nvlist(zc, configs);
1487
1488         nvlist_free(configs);
1489
1490         return (error);
1491 }
1492
1493 /*
1494  * inputs:
1495  * zc_name              name of the pool
1496  *
1497  * outputs:
1498  * zc_cookie            real errno
1499  * zc_nvlist_dst        config nvlist
1500  * zc_nvlist_dst_size   size of config nvlist
1501  */
1502 static int
1503 zfs_ioc_pool_stats(zfs_cmd_t *zc)
1504 {
1505         nvlist_t *config;
1506         int error;
1507         int ret = 0;
1508
1509         error = spa_get_stats(zc->zc_name, &config, zc->zc_value,
1510             sizeof (zc->zc_value));
```



```
1511
1512         if (config != NULL) {
1513                 ret = put_nvlist(zc, config);
1514                 nvlist_free(config);
1515
1516                 /*
1517                  * The config may be present even if 'error' is non-zero.
1518                  * In this case we return success, and preserve the real errno
1519                  * in 'zc_cookie'.
1520                  */
1521                 zc->zc_cookie = error;
1522         } else {
1523                 ret = error;
1524         }
1525
1526         return (ret);
1527 }
1528
1529 /*
1530  * Try to import the given pool, returning pool stats as appropriate so that
1531  * user land knows which devices are available and overall pool health.
1532  */
1533 static int
1534 zfs_ioc_pool_tryimport(zfs_cmd_t *zc)
1535 {
1536         nvlist_t *tryconfig, *config;
1537         int error;
1538
1539         if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1540             zc->zc_iflags, &tryconfig)) != 0)
1541                 return (error);
1542
1543         config = spa_tryimport(tryconfig);
1544
1545         nvlist_free(tryconfig);
1546
1547         if (config == NULL)
1548                 return (EINVAL);
1549
1550         error = put_nvlist(zc, config);
1551         nvlist_free(config);
1552
1553         return (error);
1554 }
1555
1556 /*
1557  * inputs:
1558  * zc_name              name of the pool
1559  * zc_cookie            scan func (pool_scan_func_t)
1560  */
1561 static int
1562 zfs_ioc_pool_scan(zfs_cmd_t *zc)
1563 {
1564         spa_t *spa;
1565         int error;
1566
1567         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1568                 return (error);
1569
1570         if (zc->zc_cookie == POOL_SCAN_NONE)
1571                 error = spa_scan_stop(spa);
1572         else
1573                 error = spa_scan(spa, zc->zc_cookie);
1574
1575         spa_close(spa, FTAG);
```

```
1577         return (error);
1578 }

1580 static int
1581 zfs_ioc_pool_freeze(zfs_cmd_t *zc)
1582 {
1583         spa_t *spa;
1584         int error;

1586         error = spa_open(zc->zc_name, &spa, FTAG);
1587         if (error == 0) {
1588                 spa_freeze(spa);
1589                 spa_close(spa, FTAG);
1590         }
1591         return (error);
1592 }

1594 static int
1595 zfs_ioc_pool_upgrade(zfs_cmd_t *zc)
1596 {
1597         spa_t *spa;
1598         int error;

1600         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1601                 return (error);

1603         if (zc->zc_cookie < spa_version(spa) ||
1604             !SPA_VERSION_IS_SUPPORTED(zc->zc_cookie)) {
1605                 spa_close(spa, FTAG);
1606                 return (EINVAL);
1607         }

1609         spa_upgrade(spa, zc->zc_cookie);
1610         spa_close(spa, FTAG);

1612         return (error);
1613 }

1615 static int
1616 zfs_ioc_pool_get_history(zfs_cmd_t *zc)
1617 {
1618         spa_t *spa;
1619         char *hist_buf;
1620         uint64_t size;
1621         int error;

1623         if ((size = zc->zc_history_len) == 0)
1624                 return (EINVAL);

1626         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1627                 return (error);

1629         if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
1630                 spa_close(spa, FTAG);
1631                 return (ENOTSUP);
1632         }

1634         hist_buf = kmem_alloc(size, KM_SLEEP);
1635         if ((error = spa_history_get(spa, &zc->zc_history_offset,
1636             &zc->zc_history_len, hist_buf)) == 0) {
1637                 error = ddi_copyout(hist_buf,
1638                     (void *)(uintptr_t)zc->zc_history,
1639                     zc->zc_history_len, zc->zc_iflags);
1640         }

1642         spa_close(spa, FTAG);
```



```
1643         kmem_free(hist_buf, size);
1644         return (error);
1645 }

1647 static int
1648 zfs_ioc_pool_reguid(zfs_cmd_t *zc)
1649 {
1650         spa_t *spa;
1651         int error;

1653         error = spa_open(zc->zc_name, &spa, FTAG);
1654         if (error == 0) {
1655                 error = spa_change_guid(spa);
1656                 spa_close(spa, FTAG);
1657         }
1658         return (error);
1659 }

1661 static int
1662 zfs_ioc_dsobj_to_dsname(zfs_cmd_t *zc)
1663 {
1664         int error;

1666         if (error = dsl_dsobj_to_dsname(zc->zc_name, zc->zc_obj, zc->zc_value))
1667                 return (error);

1669         return (0);
1670 }

1672 /*
1673  * inputs:
1674  * zc_name              name of filesystem
1675  * zc_obj               object to find
1676  *
1677  * outputs:
1678  * zc_value             name of object
1679  */
1680 static int
1681 zfs_ioc_obj_to_path(zfs_cmd_t *zc)
1682 {
1683         objset_t *os;
1684         int error;

1686         /* XXX reading from objset not owned */
1687         if ((error = dmu_objset_hold(zc->zc_name, FTAG, &os)) != 0)
1688                 return (error);
1689         if (dmu_objset_type(os) != DMU_OST_ZFS) {
1690                 dmu_objset_rele(os, FTAG);
1691                 return (EINVAL);
1692         }
1693         error = zfs_obj_to_path(os, zc->zc_obj, zc->zc_value,
1694             sizeof (zc->zc_value));
1695         dmu_objset_rele(os, FTAG);

1697         return (error);
1698 }

1700 /*
1701  * inputs:
1702  * zc_name              name of filesystem
1703  * zc_obj               object to find
1704  *
1705  * outputs:
1706  * zc_stat              stats on object
1707  * zc_value             path to object
1708  */
```

```
1709 static int
1710 zfs_ioc_obj_to_stats(zfs_cmd_t *zc)
1711 {
1712         objset_t *os;
1713         int error;

1715         /* XXX reading from objset not owned */
1716         if ((error = dmu_objset_hold(zc->zc_name, FTAG, &os)) != 0)
1717                 return (error);
1718         if (dmu_objset_type(os) != DMU_OST_ZFS) {
1719                 dmu_objset_rele(os, FTAG);
1720                 return (EINVAL);
1721         }
1722         error = zfs_obj_to_stats(os, zc->zc_obj, &zc->zc_stat, zc->zc_value,
1723             sizeof (zc->zc_value));
1724         dmu_objset_rele(os, FTAG);

1726         return (error);
1727 }

1729 static int
1730 zfs_ioc_vdev_add(zfs_cmd_t *zc)
1731 {
1732         spa_t *spa;
1733         int error;
1734         nvlist_t *config, **l2cache, **spares;
1735         uint_t nl2cache = 0, nspares = 0;

1737         error = spa_open(zc->zc_name, &spa, FTAG);
1738         if (error != 0)
1739                 return (error);

1741         error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1742             zc->zc_iflags, &config);
1743         (void) nvlist_lookup_nvlist_array(config, ZPOOL_CONFIG_L2CACHE,
1744             &l2cache, &nl2cache);

1746         (void) nvlist_lookup_nvlist_array(config, ZPOOL_CONFIG_SPARES,
1747             &spares, &nspares);

1749         /*
1750          * A root pool with concatenated devices is not supported.
1751          * Thus, can not add a device to a root pool.
1752          *
1753          * Intent log device can not be added to a rootpool because
1754          * during mountroot, zil is replayed, a seperated log device
1755          * can not be accessed during the mountroot time.
1756          *
1757          * l2cache and spare devices are ok to be added to a rootpool.
1758          */
1759         if (spa_bootfs(spa) != 0 && nl2cache == 0 && nspares == 0) {
1760                 nvlist_free(config);
1761                 spa_close(spa, FTAG);
1762                 return (EDOM);
1763         }

1765         if (error == 0) {
1766                 error = spa_vdev_add(spa, config);
1767                 nvlist_free(config);
1768         }
1769         spa_close(spa, FTAG);
1770         return (error);
1771 }

1773 /*
1774  * inputs:
```



```
1775  * zc_name              name of the pool
1776  * zc_nvlist_conf       nvlist of devices to remove
1777  * zc_cookie            to stop the remove?
1778  */
1779 static int
1780 zfs_ioc_vdev_remove(zfs_cmd_t *zc)
1781 {
1782         spa_t *spa;
1783         int error;

1785         error = spa_open(zc->zc_name, &spa, FTAG);
1786         if (error != 0)
1787                 return (error);
1788         error = spa_vdev_remove(spa, zc->zc_guid, B_FALSE);
1789         spa_close(spa, FTAG);
1790         return (error);
1791 }

1793 static int
1794 zfs_ioc_vdev_set_state(zfs_cmd_t *zc)
1795 {
1796         spa_t *spa;
1797         int error;
1798         vdev_state_t newstate = VDEV_STATE_UNKNOWN;

1800         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1801                 return (error);
1802         switch (zc->zc_cookie) {
1803         case VDEV_STATE_ONLINE:
1804                 error = vdev_online(spa, zc->zc_guid, zc->zc_obj, &newstate);
1805                 break;

1807         case VDEV_STATE_OFFLINE:
1808                 error = vdev_offline(spa, zc->zc_guid, zc->zc_obj);
1809                 break;

1811         case VDEV_STATE_FAULTED:
1812                 if (zc->zc_obj != VDEV_AUX_ERR_EXCEEDED &&
1813                     zc->zc_obj != VDEV_AUX_EXTERNAL)
1814                         zc->zc_obj = VDEV_AUX_ERR_EXCEEDED;

1816                 error = vdev_fault(spa, zc->zc_guid, zc->zc_obj);
1817                 break;

1819         case VDEV_STATE_DEGRADED:
1820                 if (zc->zc_obj != VDEV_AUX_ERR_EXCEEDED &&
1821                     zc->zc_obj != VDEV_AUX_EXTERNAL)
1822                         zc->zc_obj = VDEV_AUX_ERR_EXCEEDED;

1824                 error = vdev_degrade(spa, zc->zc_guid, zc->zc_obj);
1825                 break;

1827         default:
1828                 error = EINVAL;
1829         }
1830         zc->zc_cookie = newstate;
1831         spa_close(spa, FTAG);
1832         return (error);
1833 }

1835 static int
1836 zfs_ioc_vdev_attach(zfs_cmd_t *zc)
1837 {
1838         spa_t *spa;
1839         int replacing = zc->zc_cookie;
1840         nvlist_t *config;
```

```
1841         int error;

1843         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1844                 return (error);

1846         if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1847             zc->zc_iflags, &config)) == 0) {
1848                 error = spa_vdev_attach(spa, zc->zc_guid, config, replacing);
1849                 nvlist_free(config);
1850         }

1852         spa_close(spa, FTAG);
1853         return (error);
1854 }

1856 static int
1857 zfs_ioc_vdev_detach(zfs_cmd_t *zc)
1858 {
1859         spa_t *spa;
1860         int error;

1862         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1863                 return (error);

1865         error = spa_vdev_detach(spa, zc->zc_guid, 0, B_FALSE);

1867         spa_close(spa, FTAG);
1868         return (error);
1869 }

1871 static int
1872 zfs_ioc_vdev_split(zfs_cmd_t *zc)
1873 {
1874         spa_t *spa;
1875         nvlist_t *config, *props = NULL;
1876         int error;
1877         boolean_t exp = !!(zc->zc_cookie & ZPOOL_EXPORT_AFTER_SPLIT);

1879         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1880                 return (error);

1882         if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1883             zc->zc_iflags, &config)) {
1884                 spa_close(spa, FTAG);
1885                 return (error);
1886         }

1888         if (zc->zc_nvlist_src_size != 0 && (error =
1889             get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1890             zc->zc_iflags, &props))) {
1891                 spa_close(spa, FTAG);
1892                 nvlist_free(config);
1893                 return (error);
1894         }

1896         error = spa_vdev_split_mirror(spa, zc->zc_string, config, props, exp);

1898         spa_close(spa, FTAG);

1900         nvlist_free(config);
1901         nvlist_free(props);

1903         return (error);
1904 }

1906 static int
```



```
1907 zfs_ioc_vdev_setpath(zfs_cmd_t *zc)
1908 {
1909         spa_t *spa;
1910         char *path = zc->zc_value;
1911         uint64_t guid = zc->zc_guid;
1912         int error;

1914         error = spa_open(zc->zc_name, &spa, FTAG);
1915         if (error != 0)
1916                 return (error);

1918         error = spa_vdev_setpath(spa, guid, path);
1919         spa_close(spa, FTAG);
1920         return (error);
1921 }

1923 static int
1924 zfs_ioc_vdev_setfru(zfs_cmd_t *zc)
1925 {
1926         spa_t *spa;
1927         char *fru = zc->zc_value;
1928         uint64_t guid = zc->zc_guid;
1929         int error;

1931         error = spa_open(zc->zc_name, &spa, FTAG);
1932         if (error != 0)
1933                 return (error);

1935         error = spa_vdev_setfru(spa, guid, fru);
1936         spa_close(spa, FTAG);
1937         return (error);
1938 }

1940 static int
1941 zfs_ioc_objset_stats_impl(zfs_cmd_t *zc, objset_t *os)
1942 {
1943         int error = 0;
1944         nvlist_t *nv;

1946         dmu_objset_fast_stat(os, &zc->zc_objset_stats);

1948         if (zc->zc_nvlist_dst != 0 &&
1949             (error = dsl_prop_get_all(os, &nv)) == 0) {
1950                 dmu_objset_stats(os, nv);
1951                 /*
1952                  * NB: zvol_get_stats() will read the objset contents,
1953                  * which we aren't supposed to do with a
1954                  * DS_MODE_USER hold, because it could be
1955                  * inconsistent.  So this is a bit of a workaround...
1956                  * XXX reading with out owning
1957                  */
1958                 if (!zc->zc_objset_stats.dds_inconsistent &&
1959                     dmu_objset_type(os) == DMU_OST_ZVOL) {
1960                         error = zvol_get_stats(os, nv);
1961                         if (error == EIO)
1962                                 return (error);
1963                         VERIFY0(error);
1964                 }
1965                 error = put_nvlist(zc, nv);
1966                 nvlist_free(nv);
1967         }

1969         return (error);
1970 }

1972 /*
```

```
1973  * inputs:
1974  * zc_name              name of filesystem
1975  * zc_nvlist_dst_size   size of buffer for property nvlist
1976  *
1977  * outputs:
1978  * zc_objset_stats      stats
1979  * zc_nvlist_dst        property nvlist
1980  * zc_nvlist_dst_size   size of property nvlist
1981  */
1982 static int
1983 zfs_ioc_objset_stats(zfs_cmd_t *zc)
1984 {
1985         objset_t *os = NULL;
1986         int error;

1988         if (error = dmu_objset_hold(zc->zc_name, FTAG, &os))
1989                 return (error);

1991         error = zfs_ioc_objset_stats_impl(zc, os);

1993         dmu_objset_rele(os, FTAG);

1995         return (error);
1996 }

1998 /*
1999  * inputs:
2000  * zc_name              name of filesystem
2001  * zc_nvlist_dst_size   size of buffer for property nvlist
2002  *
2003  * outputs:
2004  * zc_nvlist_dst        received property nvlist
2005  * zc_nvlist_dst_size   size of received property nvlist
2006  *
2007  * Gets received properties (distinct from local properties on or after
2008  * SPA_VERSION_RECVD_PROPS) for callers who want to differentiate received from
2009  * local property values.
2010  */
2011 static int
2012 zfs_ioc_objset_recvd_props(zfs_cmd_t *zc)
2013 {
2014         objset_t *os = NULL;
2015         int error;
2016         nvlist_t *nv;

2018         if (error = dmu_objset_hold(zc->zc_name, FTAG, &os))
2019                 return (error);

2021         /*
2022          * Without this check, we would return local property values if the
2023          * caller has not already received properties on or after
2024          * SPA_VERSION_RECVD_PROPS.
2025          */
2026         if (!dsl_prop_get_hasrecvd(os)) {
2027                 dmu_objset_rele(os, FTAG);
2028                 return (ENOTSUP);
2029         }

2031         if (zc->zc_nvlist_dst != 0 &&
2032             (error = dsl_prop_get_received(os, &nv)) == 0) {
2033                 error = put_nvlist(zc, nv);
2034                 nvlist_free(nv);
2035         }

2037         dmu_objset_rele(os, FTAG);
2038         return (error);
```



```
2039 }

2041 static int
2042 nvl_add_zplprop(objset_t *os, nvlist_t *props, zfs_prop_t prop)
2043 {
2044         uint64_t value;
2045         int error;

2047         /*
2048          * zfs_get_zplprop() will either find a value or give us
2049          * the default value (if there is one).
2050          */
2051         if ((error = zfs_get_zplprop(os, prop, &value)) != 0)
2052                 return (error);
2053         VERIFY(nvlist_add_uint64(props, zfs_prop_to_name(prop), value) == 0);
2054         return (0);
2055 }

2057 /*
2058  * inputs:
2059  * zc_name              name of filesystem
2060  * zc_nvlist_dst_size   size of buffer for zpl property nvlist
2061  *
2062  * outputs:
2063  * zc_nvlist_dst        zpl property nvlist
2064  * zc_nvlist_dst_size   size of zpl property nvlist
2065  */
2066 static int
2067 zfs_ioc_objset_zplprops(zfs_cmd_t *zc)
2068 {
2069         objset_t *os;
2070         int err;

2072         /* XXX reading without owning */
2073         if (err = dmu_objset_hold(zc->zc_name, FTAG, &os))
2074                 return (err);

2076         dmu_objset_fast_stat(os, &zc->zc_objset_stats);

2078         /*
2079          * NB: nvl_add_zplprop() will read the objset contents,
2080          * which we aren't supposed to do with a DS_MODE_USER
2081          * hold, because it could be inconsistent.
2082          */
2083         if (zc->zc_nvlist_dst != NULL &&
2084             !zc->zc_objset_stats.dds_inconsistent &&
2085             dmu_objset_type(os) == DMU_OST_ZFS) {
2086                 nvlist_t *nv;

2088                 VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2089                 if ((err = nvl_add_zplprop(os, nv, ZFS_PROP_VERSION)) == 0 &&
2090                     (err = nvl_add_zplprop(os, nv, ZFS_PROP_NORMALIZE)) == 0 &&
2091                     (err = nvl_add_zplprop(os, nv, ZFS_PROP_UTF8ONLY)) == 0 &&
2092                     (err = nvl_add_zplprop(os, nv, ZFS_PROP_CASE)) == 0)
2093                         err = put_nvlist(zc, nv);
2094                 nvlist_free(nv);
2095         } else {
2096                 err = ENOENT;
2097         }
2098         dmu_objset_rele(os, FTAG);
2099         return (err);
2100 }

2102 static boolean_t
2103 dataset_name_hidden(const char *name)
2104 {
```

```
2105         /*
2106          * Skip over datasets that are not visible in this zone,
2107          * internal datasets (which have a $ in their name), and
2108          * temporary datasets (which have a % in their name).
2109          */
2110         if (strchr(name, '$') != NULL)
2111                 return (B_TRUE);
2112         if (strchr(name, '%') != NULL)
2113                 return (B_TRUE);
2114         if (!INGLOBALZONE(curproc) && !zone_dataset_visible(name, NULL))
2115                 return (B_TRUE);
2116         return (B_FALSE);
2117 }

2119 /*
2120  * inputs:
2121  * zc_name              name of filesystem
2122  * zc_cookie            zap cursor
2123  * zc_nvlist_dst_size   size of buffer for property nvlist
2124  *
2125  * outputs:
2126  * zc_name              name of next filesystem
2127  * zc_cookie            zap cursor
2128  * zc_objset_stats      stats
2129  * zc_nvlist_dst        property nvlist
2130  * zc_nvlist_dst_size   size of property nvlist
2131  */
2132 static int
2133 zfs_ioc_dataset_list_next(zfs_cmd_t *zc)
2134 {
2135         objset_t *os;
2136         int error;
2137         char *p;
2138         size_t orig_len = strlen(zc->zc_name);

2140 top:
2141         if (error = dmu_objset_hold(zc->zc_name, FTAG, &os)) {
2142                 if (error == ENOENT)
2143                         error = ESRCH;
2144                 return (error);
2145         }

2147         p = strrchr(zc->zc_name, '/');
2148         if (p == NULL || p[1] != '\0')
2149                 (void) strlcat(zc->zc_name, "/", sizeof (zc->zc_name));
2150         p = zc->zc_name + strlen(zc->zc_name);

2152         /*
2153          * Pre-fetch the datasets.  dmu_objset_prefetch() always returns 0
2154          * but is not declared void because its called by dmu_objset_find().
2155          */
2156         if (zc->zc_cookie == 0) {
2157                 uint64_t cookie = 0;
2158                 int len = sizeof (zc->zc_name) - (p - zc->zc_name);

2160                 while (dmu_dir_list_next(os, len, p, NULL, &cookie) == 0) {
2161                         if (!dataset_name_hidden(zc->zc_name))
2162                                 (void) dmu_objset_prefetch(zc->zc_name, NULL);
2163                 }
2164         }

2166         do {
2167                 error = dmu_dir_list_next(os,
2168                     sizeof (zc->zc_name) - (p - zc->zc_name), p,
2169                     NULL, &zc->zc_cookie);
2170                 if (error == ENOENT)
```



```
2171                         error = ESRCH;
2172         } while (error == 0 && dataset_name_hidden(zc->zc_name));
2173         dmu_objset_rele(os, FTAG);

2175         /*
2176          * If it's an internal dataset (ie. with a '$' in its name),
2177          * don't try to get stats for it, otherwise we'll return ENOENT.
2178          */
2179         if (error == 0 && strchr(zc->zc_name, '$') == NULL) {
2180                 error = zfs_ioc_objset_stats(zc); /* fill in the stats */
2181                 if (error == ENOENT) {
2182                         /* We lost a race with destroy, get the next one. */
2183                         zc->zc_name[orig_len] = '\0';
2184                         goto top;
2185                 }
2186         }
2187         return (error);
2188 }

2190 /*
2191  * inputs:
2192  * zc_name              name of filesystem
2193  * zc_cookie            zap cursor
2194  * zc_nvlist_dst_size   size of buffer for property nvlist
2195  *
2196  * outputs:
2197  * zc_name              name of next snapshot
2198  * zc_objset_stats      stats
2199  * zc_nvlist_dst        property nvlist
2200  * zc_nvlist_dst_size   size of property nvlist
2201  */
2202 static int
2203 zfs_ioc_snapshot_list_next(zfs_cmd_t *zc)
2204 {
2205         objset_t *os;
2206         int error;

2208 top:
2209         if (zc->zc_cookie == 0)
2210                 (void) dmu_objset_find(zc->zc_name, dmu_objset_prefetch,
2211                     NULL, DS_FIND_SNAPSHOTS);

2213         error = dmu_objset_hold(zc->zc_name, FTAG, &os);
2214         if (error)
2215                 return (error == ENOENT ? ESRCH : error);

2217         /*
2218          * A dataset name of maximum length cannot have any snapshots,
2219          * so exit immediately.
2220          */
2221         if (strlcat(zc->zc_name, "@", sizeof (zc->zc_name)) >= MAXNAMELEN) {
2222                 dmu_objset_rele(os, FTAG);
2223                 return (ESRCH);
2224         }

2226         error = dmu_snapshot_list_next(os,
2227             sizeof (zc->zc_name) - strlen(zc->zc_name),
2228             zc->zc_name + strlen(zc->zc_name), &zc->zc_obj, &zc->zc_cookie,
2229             NULL);

2231         if (error == 0) {
2232                 dsl_dataset_t *ds;
2233                 dsl_pool_t *dp = os->os_dsl_dataset->ds_dir->dd_pool;

2235                 /*
2236                  * Since we probably don't have a hold on this snapshot,
```

```
2237                 * it's possible that the objsetid could have been destroyed
2238                 * and reused for a new objset. It's OK if this happens during
2239                 * a zfs send operation, since the new createtxg will be
2240                 * beyond the range we're interested in.
2241                 */
2242                rw_enter(&dp->dp_config_rwlock, RW_READER);
2243                error = dsl_dataset_hold_obj(dp, zc->zc_obj, FTAG, &ds);
2244                rw_exit(&dp->dp_config_rwlock);
2245                if (error) {
2246                        if (error == ENOENT) {
2247                                /* Racing with destroy, get the next one. */
2248                                *strchr(zc->zc_name, '@') = '\0';
2249                                dmu_objset_rele(os, FTAG);
2250                                goto top;
2251                        }
2252                } else {
2253                        objset_t *ossnap;

2255                        error = dmu_objset_from_ds(ds, &ossnap);
2256                        if (error == 0)
2257                                error = zfs_ioc_objset_stats_impl(zc, ossnap);
2258                        dsl_dataset_rele(ds, FTAG);
2259                }
2260        } else if (error == ENOENT) {
2261                error = ESRCH;
2262        }

2264        dmu_objset_rele(os, FTAG);
2265        /* if we failed, undo the @ that we tacked on to zc_name */
2266        if (error)
2267                *strchr(zc->zc_name, '@') = '\0';
2268        return (error);
2269 }

2271 static int
2272 zfs_prop_set_userquota(const char *dsname, nvpair_t *pair)
2273 {
2274        const char *propname = nvpair_name(pair);
2275        uint64_t *valary;
2276        unsigned int vallen;
2277        const char *domain;
2278        char *dash;
2279        zfs_userquota_prop_t type;
2280        uint64_t rid;
2281        uint64_t quota;
2282        zfsvfs_t *zfsvfs;
2283        int err;

2285        if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2286                nvlist_t *attrs;
2287                VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2288                if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2289                    &pair) != 0)
2290                        return (EINVAL);
2291        }

2293        /*
2294         * A correctly constructed propname is encoded as
2295         * userquota@<rid>-<domain>.
2296         */
2297        if ((dash = strchr(propname, '-')) == NULL ||
2298            nvpair_value_uint64_array(pair, &valary, &vallen) != 0 ||
2299            vallen != 3)
2300                return (EINVAL);

2302        domain = dash + 1;
```



```
2303        type = valary[0];
2304        rid = valary[1];
2305        quota = valary[2];

2307        err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_FALSE);
2308        if (err == 0) {
2309                err = zfs_set_userquota(zfsvfs, type, domain, rid, quota);
2310                zfsvfs_rele(zfsvfs, FTAG);
2311        }

2313        return (err);
2314 }

2316 /*
2317  * If the named property is one that has a special function to set its value,
2318  * return 0 on success and a positive error code on failure; otherwise if it is
2319  * not one of the special properties handled by this function, return -1.
2320  *
2321  * XXX: It would be better for callers of the property interface if we handled
2322  * these special cases in dsl_prop.c (in the dsl layer).
2323  */
2324 static int
2325 zfs_prop_set_special(const char *dsname, zprop_source_t source,
2326     nvpair_t *pair)
2327 {
2328        const char *propname = nvpair_name(pair);
2329        zfs_prop_t prop = zfs_name_to_prop(propname);
2330        uint64_t intval;
2331        int err;

2333        if (prop == ZPROP_INVAL) {
2334                if (zfs_prop_userquota(propname))
2335                        return (zfs_prop_set_userquota(dsname, pair));
2336                return (-1);
2337        }

2339        if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2340                nvlist_t *attrs;
2341                VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2342                VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2343                    &pair) == 0);
2344        }

2346        if (zfs_prop_get_type(prop) == PROP_TYPE_STRING)
2347                return (-1);

2349        VERIFY(0 == nvpair_value_uint64(pair, &intval));

2351        switch (prop) {
2352        case ZFS_PROP_QUOTA:
2353                err = dsl_dir_set_quota(dsname, source, intval);
2354                break;
2355        case ZFS_PROP_REFQUOTA:
2356                err = dsl_dataset_set_quota(dsname, source, intval);
2357                break;
2358        case ZFS_PROP_RESERVATION:
2359                err = dsl_dir_set_reservation(dsname, source, intval);
2360                break;
2361        case ZFS_PROP_REFRESERVATION:
2362                err = dsl_dataset_set_reservation(dsname, source, intval);
2363                break;
2364        case ZFS_PROP_VOLSIZE:
2365                err = zvol_set_volsize(dsname, ddi_driver_major(zfs_dip),
2366                    intval);
2367                break;
2368        case ZFS_PROP_VERSION:
```

```
2369         {
2370                 zfsvfs_t *zfsvfs;

2372                 if ((err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_TRUE)) != 0)
2373                         break;

2375                 err = zfs_set_version(zfsvfs, intval);
2376                 zfsvfs_rele(zfsvfs, FTAG);

2378                 if (err == 0 && intval >= ZPL_VERSION_USERSPACE) {
2379                         zfs_cmd_t *zc;

2381                         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);
2382                         (void) strcpy(zc->zc_name, dsname);
2383                         (void) zfs_ioc_userspace_upgrade(zc);
2384                         kmem_free(zc, sizeof (zfs_cmd_t));
2385                 }
2386                 break;
2387         }

2389         default:
2390                 err = -1;
2391         }

2393         return (err);
2394 }

2396 /*
2397  * This function is best effort. If it fails to set any of the given properties,
2398  * it continues to set as many as it can and returns the last error
2399  * encountered. If the caller provides a non-NULL errlist, it will be filled in
2400  * with the list of names of all the properties that failed along with the
2401  * corresponding error numbers.
2402  *
2403  * If every property is set successfully, zero is returned and errlist is not
2404  * modified.
2405  */
2406 int
2407 zfs_set_prop_nvlist(const char *dsname, zprop_source_t source, nvlist_t *nvl,
2408     nvlist_t *errlist)
2409 {
2410         nvpair_t *pair;
2411         nvpair_t *propval;
2412         int rv = 0;
2413         uint64_t intval;
2414         char *strval;
2415         nvlist_t *genericnvl = fnvlist_alloc();
2416         nvlist_t *retrynvl = fnvlist_alloc();

2418 retry:
2419         pair = NULL;
2420         while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2421                 const char *propname = nvpair_name(pair);
2422                 zfs_prop_t prop = zfs_name_to_prop(propname);
2423                 int err = 0;

2425                 /* decode the property value */
2426                 propval = pair;
2427                 if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2428                         nvlist_t *attrs;
2429                         attrs = fnvpair_value_nvlist(pair);
2430                         if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2431                             &propval) != 0)
2432                                 err = EINVAL;
2433                 }
```



```
2435                 /* Validate value type */
2436                 if (err == 0 && prop == ZPROP_INVAL) {
2437                         if (zfs_prop_user(propname)) {
2438                                 if (nvpair_type(propval) != DATA_TYPE_STRING)
2439                                         err = EINVAL;
2440                         } else if (zfs_prop_userquota(propname)) {
2441                                 if (nvpair_type(propval) !=
2442                                     DATA_TYPE_UINT64_ARRAY)
2443                                         err = EINVAL;
2444                         } else {
2445                                 err = EINVAL;
2446                         }
2447                 } else if (err == 0) {
2448                         if (nvpair_type(propval) == DATA_TYPE_STRING) {
2449                                 if (zfs_prop_get_type(prop) != PROP_TYPE_STRING)
2450                                         err = EINVAL;
2451                         } else if (nvpair_type(propval) == DATA_TYPE_UINT64) {
2452                                 const char *unused;

2454                                 intval = fnvpair_value_uint64(propval);

2456                                 switch (zfs_prop_get_type(prop)) {
2457                                 case PROP_TYPE_NUMBER:
2458                                         break;
2459                                 case PROP_TYPE_STRING:
2460                                         err = EINVAL;
2461                                         break;
2462                                 case PROP_TYPE_INDEX:
2463                                         if (zfs_prop_index_to_string(prop,
2464                                             intval, &unused) != 0)
2465                                                 err = EINVAL;
2466                                         break;
2467                                 default:
2468                                         cmn_err(CE_PANIC,
2469                                             "unknown property type");
2470                                 }
2471                         } else {
2472                                 err = EINVAL;
2473                         }
2474                 }

2476                 /* Validate permissions */
2477                 if (err == 0)
2478                         err = zfs_check_settable(dsname, pair, CRED());

2480                 if (err == 0) {
2481                         err = zfs_prop_set_special(dsname, source, pair);
2482                         if (err == -1) {
2483                                 /*
2484                                  * For better performance we build up a list of
2485                                  * properties to set in a single transaction.
2486                                  */
2487                                 err = nvlist_add_nvpair(genericnvl, pair);
2488                         } else if (err != 0 && nvl != retrynvl) {
2489                                 /*
2490                                  * This may be a spurious error caused by
2491                                  * receiving quota and reservation out of order.
2492                                  * Try again in a second pass.
2493                                  */
2494                                 err = nvlist_add_nvpair(retrynvl, pair);
2495                         }
2496                 }

2498                 if (err != 0) {
2499                         if (errlist != NULL)
2500                                 fnvlist_add_int32(errlist, propname, err);
```

```
2501                        rv = err;
2502                }
2503        }

2505        if (nvl != retrynvl && !nvlist_empty(retrynvl)) {
2506                nvl = retrynvl;
2507                goto retry;
2508        }

2510        if (!nvlist_empty(genericnvl) &&
2511            dsl_props_set(dsname, source, genericnvl) != 0) {
2512                /*
2513                 * If this fails, we still want to set as many properties as we
2514                 * can, so try setting them individually.
2515                 */
2516                pair = NULL;
2517                while ((pair = nvlist_next_nvpair(genericnvl, pair)) != NULL) {
2518                        const char *propname = nvpair_name(pair);
2519                        int err = 0;

2521                        propval = pair;
2522                        if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2523                                nvlist_t *attrs;
2524                                attrs = fnvpair_value_nvlist(pair);
2525                                propval = fnvlist_lookup_nvpair(attrs,
2526                                    ZPROP_VALUE);
2527                        }

2529                        if (nvpair_type(propval) == DATA_TYPE_STRING) {
2530                                strval = fnvpair_value_string(propval);
2531                                err = dsl_prop_set(dsname, propname, source, 1,
2532                                    strlen(strval) + 1, strval);
2533                        } else {
2534                                intval = fnvpair_value_uint64(propval);
2535                                err = dsl_prop_set(dsname, propname, source, 8,
2536                                    1, &intval);
2537                        }

2539                        if (err != 0) {
2540                                if (errlist != NULL) {
2541                                        fnvlist_add_int32(errlist, propname,
2542                                            err);
2543                                }
2544                                rv = err;
2545                        }
2546                }
2547        }
2548        nvlist_free(genericnvl);
2549        nvlist_free(retrynvl);

2551        return (rv);
2552 }

2554 /*
2555  * Check that all the properties are valid user properties.
2556  */
2557 static int
2558 zfs_check_userprops(const char *fsname, nvlist_t *nvl)
2559 {
2560        nvpair_t *pair = NULL;
2561        int error = 0;

2563        while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2564                const char *propname = nvpair_name(pair);
2565                char *valstr;
```



```
2567                if (!zfs_prop_user(propname) ||
2568                    nvpair_type(pair) != DATA_TYPE_STRING)
2569                        return (EINVAL);

2571                if (error = zfs_secpolicy_write_perms(fsname,
2572                    ZFS_DELEG_PERM_USERPROP, CRED()))
2573                        return (error);

2575                if (strlen(propname) >= ZAP_MAXNAMELEN)
2576                        return (ENAMETOOLONG);

2578                VERIFY(nvpair_value_string(pair, &valstr) == 0);
2579                if (strlen(valstr) >= ZAP_MAXVALUELEN)
2580                        return (E2BIG);
2581        }
2582        return (0);
2583 }

2585 static void
2586 props_skip(nvlist_t *props, nvlist_t *skipped, nvlist_t **newprops)
2587 {
2588        nvpair_t *pair;

2590        VERIFY(nvlist_alloc(newprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2592        pair = NULL;
2593        while ((pair = nvlist_next_nvpair(props, pair)) != NULL) {
2594                if (nvlist_exists(skipped, nvpair_name(pair)))
2595                        continue;

2597                VERIFY(nvlist_add_nvpair(*newprops, pair) == 0);
2598        }
2599 }

2601 static int
2602 clear_received_props(objset_t *os, const char *fs, nvlist_t *props,
2603     nvlist_t *skipped)
2604 {
2605        int err = 0;
2606        nvlist_t *cleared_props = NULL;
2607        props_skip(props, skipped, &cleared_props);
2608        if (!nvlist_empty(cleared_props)) {
2609                /*
2610                 * Acts on local properties until the dataset has received
2611                 * properties at least once on or after SPA_VERSION_RECVD_PROPS.
2612                 */
2613                zprop_source_t flags = (ZPROP_SRC_NONE |
2614                    (dsl_prop_get_hasrecvd(os) ? ZPROP_SRC_RECEIVED : 0));
2615                err = zfs_set_prop_nvlist(fs, flags, cleared_props, NULL);
2616        }
2617        nvlist_free(cleared_props);
2618        return (err);
2619 }

2621 /*
2622  * inputs:
2623  * zc_name              name of filesystem
2624  * zc_value             name of property to set
2625  * zc_nvlist_src{_size} nvlist of properties to apply
2626  * zc_cookie            received properties flag
2627  *
2628  * outputs:
2629  * zc_nvlist_dst{_size} error for each unapplied received property
2630  */
2631 static int
2632 zfs_ioc_set_prop(zfs_cmd_t *zc)
```

```
{
        nvlist_t *nvl;
        boolean_t received = zc->zc_cookie;
        zprop_source_t source = (received ? ZPROP_SRC_RECEIVED :
            ZPROP_SRC_LOCAL);
        nvlist_t *errors;
        int error;

        if ((error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
            zc->zc_iflags, &nvl)) != 0)
                return (error);

        if (received) {
                nvlist_t *origprops;
                objset_t *os;

                if (dmu_objset_hold(zc->zc_name, FTAG, &os) == 0) {
                        if (dsl_prop_get_received(os, &origprops) == 0) {
                                (void) clear_received_props(os,
                                    zc->zc_name, origprops, nvl);
                                nvlist_free(origprops);
                        }

                        dsl_prop_set_hasrecvd(os);
                        dmu_objset_rele(os, FTAG);
                }
        }

        errors = fnvlist_alloc();
        error = zfs_set_prop_nvlist(zc->zc_name, source, nvl, errors);

        if (zc->zc_nvlist_dst != NULL && errors != NULL) {
                (void) put_nvlist(zc, errors);
        }

        nvlist_free(errors);
        nvlist_free(nvl);
        return (error);
}

/*
 * inputs:
 * zc_name              name of filesystem
 * zc_value             name of property to inherit
 * zc_cookie            revert to received value if TRUE
 *
 * outputs:             none
 */
static int
zfs_ioc_inherit_prop(zfs_cmd_t *zc)
{
        const char *propname = zc->zc_value;
        zfs_prop_t prop = zfs_name_to_prop(propname);
        boolean_t received = zc->zc_cookie;
        zprop_source_t source = (received
            ? ZPROP_SRC_NONE            /* revert to received value, if any */
            : ZPROP_SRC_INHERITED);     /* explicitly inherit */

        if (received) {
                nvlist_t *dummy;
                nvpair_t *pair;
                zprop_type_t type;
                int err;

                /*
                 * zfs_prop_set_special() expects properties in the form of an
```



```
                 * nvpair with type info.
                 */
                if (prop == ZPROP_INVAL) {
                        if (!zfs_prop_user(propname))
                                return (EINVAL);

                        type = PROP_TYPE_STRING;
                } else if (prop == ZFS_PROP_VOLSIZE ||
                    prop == ZFS_PROP_VERSION) {
                        return (EINVAL);
                } else {
                        type = zfs_prop_get_type(prop);
                }

                VERIFY(nvlist_alloc(&dummy, NV_UNIQUE_NAME, KM_SLEEP) == 0);

                switch (type) {
                case PROP_TYPE_STRING:
                        VERIFY(0 == nvlist_add_string(dummy, propname, ""));
                        break;
                case PROP_TYPE_NUMBER:
                case PROP_TYPE_INDEX:
                        VERIFY(0 == nvlist_add_uint64(dummy, propname, 0));
                        break;
                default:
                        nvlist_free(dummy);
                        return (EINVAL);
                }

                pair = nvlist_next_nvpair(dummy, NULL);
                err = zfs_prop_set_special(zc->zc_name, source, pair);
                nvlist_free(dummy);
                if (err != -1)
                        return (err); /* special property already handled */
        } else {
                /*
                 * Only check this in the non-received case. We want to allow
                 * 'inherit -S' to revert non-inheritable properties like quota
                 * and reservation to the received or default values even though
                 * they are not considered inheritable.
                 */
                if (prop != ZPROP_INVAL && !zfs_prop_inheritable(prop))
                        return (EINVAL);
        }

        /* property name has been validated by zfs_secpolicy_inherit_prop() */
        return (dsl_prop_set(zc->zc_name, zc->zc_value, source, 0, 0, NULL));
}

static int
zfs_ioc_pool_set_props(zfs_cmd_t *zc)
{
        nvlist_t *props;
        spa_t *spa;
        int error;
        nvpair_t *pair;

        if (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
            zc->zc_iflags, &props))
                return (error);

        /*
         * If the only property is the configfile, then just do a spa_lookup()
         * to handle the faulted case.
         */
        pair = nvlist_next_nvpair(props, NULL);
```

```
2765         if (pair != NULL && strcmp(nvpair_name(pair),
2766             zpool_prop_to_name(ZPOOL_PROP_CACHEFILE)) == 0 &&
2767             nvlist_next_nvpair(props, pair) == NULL) {
2768                 mutex_enter(&spa_namespace_lock);
2769                 if ((spa = spa_lookup(zc->zc_name)) != NULL) {
2770                         spa_configfile_set(spa, props, B_FALSE);
2771                         spa_config_sync(spa, B_FALSE, B_TRUE);
2772                 }
2773                 mutex_exit(&spa_namespace_lock);
2774                 if (spa != NULL) {
2775                         nvlist_free(props);
2776                         return (0);
2777                 }
2778         }

2780         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0) {
2781                 nvlist_free(props);
2782                 return (error);
2783         }

2785         error = spa_prop_set(spa, props);

2787         nvlist_free(props);
2788         spa_close(spa, FTAG);

2790         return (error);
2791 }

2793 static int
2794 zfs_ioc_pool_get_props(zfs_cmd_t *zc)
2795 {
2796         spa_t *spa;
2797         int error;
2798         nvlist_t *nvp = NULL;

2800         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0) {
2801                 /*
2802                  * If the pool is faulted, there may be properties we can still
2803                  * get (such as altroot and cachefile), so attempt to get them
2804                  * anyway.
2805                  */
2806                 mutex_enter(&spa_namespace_lock);
2807                 if ((spa = spa_lookup(zc->zc_name)) != NULL)
2808                         error = spa_prop_get(spa, &nvp);
2809                 mutex_exit(&spa_namespace_lock);
2810         } else {
2811                 error = spa_prop_get(spa, &nvp);
2812                 spa_close(spa, FTAG);
2813         }

2815         if (error == 0 && zc->zc_nvlist_dst != NULL)
2816                 error = put_nvlist(zc, nvp);
2817         else
2818                 error = EFAULT;

2820         nvlist_free(nvp);
2821         return (error);
2822 }

2824 /*
2825  * inputs:
2826  * zc_name              name of filesystem
2827  * zc_nvlist_src{_size} nvlist of delegated permissions
2828  * zc_perm_action       allow/unallow flag
2829  *
2830  * outputs:             none
```



```
2831  */
2832 static int
2833 zfs_ioc_set_fsacl(zfs_cmd_t *zc)
2834 {
2835         int error;
2836         nvlist_t *fsaclnv = NULL;

2838         if ((error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2839             zc->zc_iflags, &fsaclnv)) != 0)
2840                 return (error);

2842         /*
2843          * Verify nvlist is constructed correctly
2844          */
2845         if ((error = zfs_deleg_verify_nvlist(fsaclnv)) != 0) {
2846                 nvlist_free(fsaclnv);
2847                 return (EINVAL);
2848         }

2850         /*
2851          * If we don't have PRIV_SYS_MOUNT, then validate
2852          * that user is allowed to hand out each permission in
2853          * the nvlist(s)
2854          */

2856         error = secpolicy_zfs(CRED());
2857         if (error) {
2858                 if (zc->zc_perm_action == B_FALSE) {
2859                         error = dsl_deleg_can_allow(zc->zc_name,
2860                             fsaclnv, CRED());
2861                 } else {
2862                         error = dsl_deleg_can_unallow(zc->zc_name,
2863                             fsaclnv, CRED());
2864                 }
2865         }

2867         if (error == 0)
2868                 error = dsl_deleg_set(zc->zc_name, fsaclnv, zc->zc_perm_action);

2870         nvlist_free(fsaclnv);
2871         return (error);
2872 }

2874 /*
2875  * inputs:
2876  * zc_name              name of filesystem
2877  *
2878  * outputs:
2879  * zc_nvlist_src{_size} nvlist of delegated permissions
2880  */
2881 static int
2882 zfs_ioc_get_fsacl(zfs_cmd_t *zc)
2883 {
2884         nvlist_t *nvp;
2885         int error;

2887         if ((error = dsl_deleg_get(zc->zc_name, &nvp)) == 0) {
2888                 error = put_nvlist(zc, nvp);
2889                 nvlist_free(nvp);
2890         }

2892         return (error);
2893 }

2895 /*
2896  * Search the vfs list for a specified resource.  Returns a pointer to it
```

```
 * or NULL if no suitable entry is found. The caller of this routine
 * is responsible for releasing the returned vfs pointer.
 */
static vfs_t *
zfs_get_vfs(const char *resource)
{
        struct vfs *vfsp;
        struct vfs *vfs_found = NULL;

        vfs_list_read_lock();
        vfsp = rootvfs;
        do {
                if (strcmp(refstr_value(vfsp->vfs_resource), resource) == 0) {
                        VFS_HOLD(vfsp);
                        vfs_found = vfsp;
                        break;
                }
                vfsp = vfsp->vfs_next;
        } while (vfsp != rootvfs);
        vfs_list_unlock();
        return (vfs_found);
}

/* ARGSUSED */
static void
zfs_create_cb(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx)
{
        zfs_creat_t *zct = arg;

        zfs_create_fs(os, cr, zct->zct_zplprops, tx);
}

#define ZFS_PROP_UNDEFINED      ((uint64_t)-1)

/*
 * inputs:
 * createprops          list of properties requested by creator
 * default_zplver       zpl version to use if unspecified in createprops
 * fuids_ok             fuids allowed in this version of the spa?
 * os                   parent objset pointer (NULL if root fs)
 *
 * outputs:
 * zplprops     values for the zplprops we attach to the master node object
 * is_ci        true if requested file system will be purely case-insensitive
 *
 * Determine the settings for utf8only, normalization and
 * casesensitivity.  Specific values may have been requested by the
 * creator and/or we can inherit values from the parent dataset.  If
 * the file system is of too early a vintage, a creator can not
 * request settings for these properties, even if the requested
 * setting is the default value.  We don't actually want to create dsl
 * properties for these, so remove them from the source nvlist after
 * processing.
 */
static int
zfs_fill_zplprops_impl(objset_t *os, uint64_t zplver,
    boolean_t fuids_ok, boolean_t sa_ok, nvlist_t *createprops,
    nvlist_t *zplprops, boolean_t *is_ci)
{
        uint64_t sense = ZFS_PROP_UNDEFINED;
        uint64_t norm = ZFS_PROP_UNDEFINED;
        uint64_t u8 = ZFS_PROP_UNDEFINED;

        ASSERT(zplprops != NULL);

        /*
```



```
         * Pull out creator prop choices, if any.
         */
        if (createprops) {
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_VERSION), &zplver);
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_NORMALIZE), &norm);
                (void) nvlist_remove_all(createprops,
                    zfs_prop_to_name(ZFS_PROP_NORMALIZE));
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_UTF8ONLY), &u8);
                (void) nvlist_remove_all(createprops,
                    zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_CASE), &sense);
                (void) nvlist_remove_all(createprops,
                    zfs_prop_to_name(ZFS_PROP_CASE));
        }

        /*
         * If the zpl version requested is whacky or the file system
         * or pool is version is too "young" to support normalization
         * and the creator tried to set a value for one of the props,
         * error out.
         */
        if ((zplver < ZPL_VERSION_INITIAL || zplver > ZPL_VERSION) ||
            (zplver >= ZPL_VERSION_FUID && !fuids_ok) ||
            (zplver >= ZPL_VERSION_SA && !sa_ok) ||
            (zplver < ZPL_VERSION_NORMALIZATION &&
            (norm != ZFS_PROP_UNDEFINED || u8 != ZFS_PROP_UNDEFINED ||
            sense != ZFS_PROP_UNDEFINED)))
                return (ENOTSUP);

        /*
         * Put the version in the zplprops
         */
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_VERSION), zplver) == 0);

        if (norm == ZFS_PROP_UNDEFINED)
                VERIFY(zfs_get_zplprop(os, ZFS_PROP_NORMALIZE, &norm) == 0);
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_NORMALIZE), norm) == 0);

        /*
         * If we're normalizing, names must always be valid UTF-8 strings.
         */
        if (norm)
                u8 = 1;
        if (u8 == ZFS_PROP_UNDEFINED)
                VERIFY(zfs_get_zplprop(os, ZFS_PROP_UTF8ONLY, &u8) == 0);
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_UTF8ONLY), u8) == 0);

        if (sense == ZFS_PROP_UNDEFINED)
                VERIFY(zfs_get_zplprop(os, ZFS_PROP_CASE, &sense) == 0);
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_CASE), sense) == 0);

        if (is_ci)
                *is_ci = (sense == ZFS_CASE_INSENSITIVE);

        return (0);
}

static int
```

```
3029 zfs_fill_zplprops(const char *dataset, nvlist_t *createprops,
3030     nvlist_t *zplprops, boolean_t *is_ci)
3031 {
3032         boolean_t fuids_ok, sa_ok;
3033         uint64_t zplver = ZPL_VERSION;
3034         objset_t *os = NULL;
3035         char parentname[MAXNAMELEN];
3036         char *cp;
3037         spa_t *spa;
3038         uint64_t spa_vers;
3039         int error;

3041         (void) strlcpy(parentname, dataset, sizeof (parentname));
3042         cp = strrchr(parentname, '/');
3043         ASSERT(cp != NULL);
3044         cp[0] = '\0';

3046         if ((error = spa_open(dataset, &spa, FTAG)) != 0)
3047                 return (error);

3049         spa_vers = spa_version(spa);
3050         spa_close(spa, FTAG);

3052         zplver = zfs_zpl_version_map(spa_vers);
3053         fuids_ok = (zplver >= ZPL_VERSION_FUID);
3054         sa_ok = (zplver >= ZPL_VERSION_SA);

3056         /*
3057          * Open parent object set so we can inherit zplprop values.
3058          */
3059         if ((error = dmu_objset_hold(parentname, FTAG, &os)) != 0)
3060                 return (error);

3062         error = zfs_fill_zplprops_impl(os, zplver, fuids_ok, sa_ok, createprops,
3063             zplprops, is_ci);
3064         dmu_objset_rele(os, FTAG);
3065         return (error);
3066 }

3068 static int
3069 zfs_fill_zplprops_root(uint64_t spa_vers, nvlist_t *createprops,
3070     nvlist_t *zplprops, boolean_t *is_ci)
3071 {
3072         boolean_t fuids_ok;
3073         boolean_t sa_ok;
3074         uint64_t zplver = ZPL_VERSION;
3075         int error;

3077         zplver = zfs_zpl_version_map(spa_vers);
3078         fuids_ok = (zplver >= ZPL_VERSION_FUID);
3079         sa_ok = (zplver >= ZPL_VERSION_SA);

3081         error = zfs_fill_zplprops_impl(NULL, zplver, fuids_ok, sa_ok,
3082             createprops, zplprops, is_ci);
3083         return (error);
3084 }

3086 /*
3087  * innvl: {
3088  *     "type" -> dmu_objset_type_t (int32)
3089  *     (optional) "props" -> { prop -> value }
3090  * }
3091  *
3092  * outnvl: propname -> error code (int32)
3093  */
3094 static int
```



```
3095 zfs_ioc_create(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3096 {
3097         int error = 0;
3098         zfs_creat_t zct = { 0 };
3099         nvlist_t *nvprops = NULL;
3100         void (*cbfunc)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx);
3101         int32_t type32;
3102         dmu_objset_type_t type;
3103         boolean_t is_insensitive = B_FALSE;

3105         if (nvlist_lookup_int32(innvl, "type", &type32) != 0)
3106                 return (EINVAL);
3107         type = type32;
3108         (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);

3110         switch (type) {
3111         case DMU_OST_ZFS:
3112                 cbfunc = zfs_create_cb;
3113                 break;

3115         case DMU_OST_ZVOL:
3116                 cbfunc = zvol_create_cb;
3117                 break;

3119         default:
3120                 cbfunc = NULL;
3121                 break;
3122         }
3123         if (strchr(fsname, '@') ||
3124             strchr(fsname, '%'))
3125                 return (EINVAL);

3127         zct.zct_props = nvprops;

3129         if (cbfunc == NULL)
3130                 return (EINVAL);

3132         if (type == DMU_OST_ZVOL) {
3133                 uint64_t volsize, volblocksize;

3135                 if (nvprops == NULL)
3136                         return (EINVAL);
3137                 if (nvlist_lookup_uint64(nvprops,
3138                     zfs_prop_to_name(ZFS_PROP_VOLSIZE), &volsize) != 0)
3139                         return (EINVAL);

3141                 if ((error = nvlist_lookup_uint64(nvprops,
3142                     zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
3143                     &volblocksize)) != 0 && error != ENOENT)
3144                         return (EINVAL);

3146                 if (error != 0)
3147                         volblocksize = zfs_prop_default_numeric(
3148                             ZFS_PROP_VOLBLOCKSIZE);

3150                 if ((error = zvol_check_volblocksize(
3151                     volblocksize)) != 0 ||
3152                     (error = zvol_check_volsize(volsize,
3153                     volblocksize)) != 0)
3154                         return (error);
3155         } else if (type == DMU_OST_ZFS) {
3156                 int error;

3158                 /*
3159                  * We have to have normalization and
3160                  * case-folding flags correct when we do the
```

```
3161                 * file system creation, so go figure them out
3162                 * now.
3163                 */
3164                VERIFY(nvlist_alloc(&zct.zct_zplprops,
3165                    NV_UNIQUE_NAME, KM_SLEEP) == 0);
3166                error = zfs_fill_zplprops(fsname, nvprops,
3167                    zct.zct_zplprops, &is_insensitive);
3168                if (error != 0) {
3169                        nvlist_free(zct.zct_zplprops);
3170                        return (error);
3171                }
3172        }

3174        error = dmu_objset_create(fsname, type,
3175            is_insensitive ? DS_FLAG_CI_DATASET : 0, cbfunc, &zct);
3176        nvlist_free(zct.zct_zplprops);

3178        /*
3179         * It would be nice to do this atomically.
3180         */
3181        if (error == 0) {
3182                error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3183                    nvprops, outnvl);
3184                if (error != 0)
3185                        (void) dmu_objset_destroy(fsname, B_FALSE);
3186        }
3187        return (error);
3188 }

3190 /*
3191  * innvl: {
3192  *     "origin" -> name of origin snapshot
3193  *     (optional) "props" -> { prop -> value }
3194  * }
3195  *
3196  * outnvl: propname -> error code (int32)
3197  */
3198 static int
3199 zfs_ioc_clone(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3200 {
3201        int error = 0;
3202        nvlist_t *nvprops = NULL;
3203        char *origin_name;
3204        dsl_dataset_t *origin;

3206        if (nvlist_lookup_string(innvl, "origin", &origin_name) != 0)
3207                return (EINVAL);
3208        (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);

3210        if (strchr(fsname, '@') ||
3211            strchr(fsname, '%'))
3212                return (EINVAL);

3214        if (dataset_namecheck(origin_name, NULL, NULL) != 0)
3215                return (EINVAL);

3217        error = dsl_dataset_hold(origin_name, FTAG, &origin);
3218        if (error)
3219                return (error);

3221        error = dmu_objset_clone(fsname, origin, 0);
3222        dsl_dataset_rele(origin, FTAG);
3223        if (error)
3224                return (error);

3226        /*
```



```
3227         * It would be nice to do this atomically.
3228         */
3229        if (error == 0) {
3230                error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3231                    nvprops, outnvl);
3232                if (error != 0)
3233                        (void) dmu_objset_destroy(fsname, B_FALSE);
3234        }
3235        return (error);
3236 }

3238 /*
3239  * innvl: {
3240  *     "snaps" -> { snapshot1, snapshot2 }
3241  *     (optional) "props" -> { prop -> value (string) }
3242  * }
3243  *
3244  * outnvl: snapshot -> error code (int32)
3245  *
3246  */
3247 static int
3248 zfs_ioc_snapshot(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3249 {
3250        nvlist_t *snaps;
3251        nvlist_t *props = NULL;
3252        int error, poollen;
3253        nvpair_t *pair;

3255        (void) nvlist_lookup_nvlist(innvl, "props", &props);
3256        if ((error = zfs_check_userprops(poolname, props)) != 0)
3257                return (error);

3259        if (!nvlist_empty(props) &&
3260            zfs_earlier_version(poolname, SPA_VERSION_SNAP_PROPS))
3261                return (ENOTSUP);

3263        if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3264                return (EINVAL);
3265        poollen = strlen(poolname);
3266        for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3267            pair = nvlist_next_nvpair(snaps, pair)) {
3268                const char *name = nvpair_name(pair);
3269                const char *cp = strchr(name, '@');

3271                /*
3272                 * The snap name must contain an @, and the part after it must
3273                 * contain only valid characters.
3274                 */
3275                if (cp == NULL || snapshot_namecheck(cp + 1, NULL, NULL) != 0)
3276                        return (EINVAL);

3278                /*
3279                 * The snap must be in the specified pool.
3280                 */
3281                if (strncmp(name, poolname, poollen) != 0 ||
3282                    (name[poollen] != '/' && name[poollen] != '@'))
3283                        return (EXDEV);

3285                /* This must be the only snap of this fs. */
3286                for (nvpair_t *pair2 = nvlist_next_nvpair(snaps, pair);
3287                    pair2 != NULL; pair2 = nvlist_next_nvpair(snaps, pair2)) {
3288                        if (strncmp(name, nvpair_name(pair2), cp - name + 1)
3289                            == 0) {
3290                                return (EXDEV);
3291                        }
3292                }
```

```
3293         }

3295         error = dmu_objset_snapshot(snaps, props, outnvl);
3296         return (error);
3297 }

3299 /*
3300  * innvl: "message" -> string
3301  */
3302 /* ARGSUSED */
3303 static int
3304 zfs_ioc_log_history(const char *unused, nvlist_t *innvl, nvlist_t *outnvl)
3305 {
3306         char *message;
3307         spa_t *spa;
3308         int error;
3309         char *poolname;

3311         /*
3312          * The poolname in the ioctl is not set, we get it from the TSD,
3313          * which was set at the end of the last successful ioctl that allows
3314          * logging.  The secpolicy func already checked that it is set.
3315          * Only one log ioctl is allowed after each successful ioctl, so
3316          * we clear the TSD here.
3317          */
3318         poolname = tsd_get(zfs_allow_log_key);
3319         (void) tsd_set(zfs_allow_log_key, NULL);
3320         error = spa_open(poolname, &spa, FTAG);
3321         strfree(poolname);
3322         if (error != 0)
3323                 return (error);

3325         if (nvlist_lookup_string(innvl, "message", &message) != 0)  {
3326                 spa_close(spa, FTAG);
3327                 return (EINVAL);
3328         }

3330         if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
3331                 spa_close(spa, FTAG);
3332                 return (ENOTSUP);
3333         }

3335         error = spa_history_log(spa, message);
3336         spa_close(spa, FTAG);
3337         return (error);
3338 }

3340 /* ARGSUSED */
3341 int
3342 zfs_unmount_snap(const char *name, void *arg)
3343 {
3344         vfs_t *vfsp;
3345         int err;

3347         if (strchr(name, '@') == NULL)
3348                 return (0);

3350         vfsp = zfs_get_vfs(name);
3351         if (vfsp == NULL)
3352                 return (0);

3354         if ((err = vn_vfswlock(vfsp->vfs_vnodecovered)) != 0) {
3355                 VFS_RELE(vfsp);
3356                 return (err);
3357         }
3358         VFS_RELE(vfsp);
```



```
3360         /*
3361          * Always force the unmount for snapshots.
3362          */
3363         return (dounmount(vfsp, MS_FORCE, kcred));
3364 }

3366 /*
3367  * innvl: {
3368  *     "snaps" -> { snapshot1, snapshot2 }
3369  *     (optional boolean) "defer"
3370  * }
3371  *
3372  * outnvl: snapshot -> error code (int32)
3373  *
3374  */
3375 static int
3376 zfs_ioc_destroy_snaps(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3377 {
3378         int poollen;
3379         nvlist_t *snaps;
3380         nvpair_t *pair;
3381         boolean_t defer;

3383         if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3384                 return (EINVAL);
3385         defer = nvlist_exists(innvl, "defer");

3387         poollen = strlen(poolname);
3388         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3389             pair = nvlist_next_nvpair(snaps, pair)) {
3390                 const char *name = nvpair_name(pair);

3392                 /*
3393                  * The snap must be in the specified pool.
3394                  */
3395                 if (strncmp(name, poolname, poollen) != 0 ||
3396                     (name[poollen] != '/' && name[poollen] != '@'))
3397                         return (EXDEV);

3399                 /*
3400                  * Ignore failures to unmount; dmu_snapshots_destroy_nvl()
3401                  * will deal with this gracefully (by filling in outnvl).
3402                  */
3403                 (void) zfs_unmount_snap(name, NULL);
3404         }

3406         return (dmu_snapshots_destroy_nvl(snaps, defer, outnvl));
3407 }

3409 /*
3410  * inputs:
3411  * zc_name              name of dataset to destroy
3412  * zc_objset_type       type of objset
3413  * zc_defer_destroy     mark for deferred destroy
3414  *
3415  * outputs:             none
3416  */
3417 static int
3418 zfs_ioc_destroy(zfs_cmd_t *zc)
3419 {
3420         int err;
3421         if (strchr(zc->zc_name, '@') && zc->zc_objset_type == DMU_OST_ZFS) {
3422                 err = zfs_unmount_snap(zc->zc_name, NULL);
3423                 if (err)
3424                         return (err);
```

```
3425         }

3427         err = dmu_objset_destroy(zc->zc_name, zc->zc_defer_destroy);
3428         if (zc->zc_objset_type == DMU_OST_ZVOL && err == 0)
3429                 (void) zvol_remove_minor(zc->zc_name);
3430         return (err);
3431 }

3433 /*
3434  * inputs:
3435  * zc_name      name of dataset to rollback (to most recent snapshot)
3436  *
3437  * outputs:     none
3438  */
3439 static int
3440 zfs_ioc_rollback(zfs_cmd_t *zc)
3441 {
3442         dsl_dataset_t *ds, *clone;
3443         int error;
3444         zfsvfs_t *zfsvfs;
3445         char *clone_name;

3447         error = dsl_dataset_hold(zc->zc_name, FTAG, &ds);
3448         if (error)
3449                 return (error);

3451         /* must not be a snapshot */
3452         if (dsl_dataset_is_snapshot(ds)) {
3453                 dsl_dataset_rele(ds, FTAG);
3454                 return (EINVAL);
3455         }

3457         /* must have a most recent snapshot */
3458         if (ds->ds_phys->ds_prev_snap_txg < TXG_INITIAL) {
3459                 dsl_dataset_rele(ds, FTAG);
3460                 return (EINVAL);
3461         }

3463         /*
3464          * Create clone of most recent snapshot.
3465          */
3466         clone_name = kmem_asprintf("%s/%%rollback", zc->zc_name);
3467         error = dmu_objset_clone(clone_name, ds->ds_prev, DS_FLAG_INCONSISTENT);
3468         if (error)
3469                 goto out;

3471         error = dsl_dataset_own(clone_name, B_TRUE, FTAG, &clone);
3472         if (error)
3473                 goto out;

3475         /*
3476          * Do clone swap.
3477          */
3478         if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
3479                 error = zfs_suspend_fs(zfsvfs);
3480                 if (error == 0) {
3481                         int resume_err;

3483                         if (dsl_dataset_tryown(ds, B_FALSE, FTAG)) {
3484                                 error = dsl_dataset_clone_swap(clone, ds,
3485                                     B_TRUE);
3486                                 dsl_dataset_disown(ds, FTAG);
3487                                 ds = NULL;
3488                         } else {
3489                                 error = EBUSY;
3490                         }
```



```
3491                         resume_err = zfs_resume_fs(zfsvfs, zc->zc_name);
3492                         error = error ? error : resume_err;
3493                 }
3494                 VFS_RELE(zfsvfs->z_vfs);
3495         } else {
3496                 if (dsl_dataset_tryown(ds, B_FALSE, FTAG)) {
3497                         error = dsl_dataset_clone_swap(clone, ds, B_TRUE);
3498                         dsl_dataset_disown(ds, FTAG);
3499                         ds = NULL;
3500                 } else {
3501                         error = EBUSY;
3502                 }
3503         }

3505         /*
3506          * Destroy clone (which also closes it).
3507          */
3508         (void) dsl_dataset_destroy(clone, FTAG, B_FALSE);

3510 out:
3511         strfree(clone_name);
3512         if (ds)
3513                 dsl_dataset_rele(ds, FTAG);
3514         return (error);
3515 }

3517 /*
3518  * inputs:
3519  * zc_name      old name of dataset
3520  * zc_value     new name of dataset
3521  * zc_cookie    recursive flag (only valid for snapshots)
3522  *
3523  * outputs:     none
3524  */
3525 static int
3526 zfs_ioc_rename(zfs_cmd_t *zc)
3527 {
3528         boolean_t recursive = zc->zc_cookie & 1;

3530         zc->zc_value[sizeof (zc->zc_value) - 1] = '\0';
3531         if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
3532             strchr(zc->zc_value, '%'))
3533                 return (EINVAL);

3535         /*
3536          * Unmount snapshot unless we're doing a recursive rename,
3537          * in which case the dataset code figures out which snapshots
3538          * to unmount.
3539          */
3540         if (!recursive && strchr(zc->zc_name, '@') != NULL &&
3541             zc->zc_objset_type == DMU_OST_ZFS) {
3542                 int err = zfs_unmount_snap(zc->zc_name, NULL);
3543                 if (err)
3544                         return (err);
3545         }
3546         if (zc->zc_objset_type == DMU_OST_ZVOL)
3547                 (void) zvol_remove_minor(zc->zc_name);
3548         return (dmu_objset_rename(zc->zc_name, zc->zc_value, recursive));
3549 }

3551 static int
3552 zfs_check_settable(const char *dsname, nvpair_t *pair, cred_t *cr)
3553 {
3554         const char *propname = nvpair_name(pair);
3555         boolean_t issnap = (strchr(dsname, '@') != NULL);
3556         zfs_prop_t prop = zfs_name_to_prop(propname);
```

```
3557         uint64_t intval;
3558         int err;
3559 
3560         if (prop == ZPROP_INVAL) {
3561                 if (zfs_prop_user(propname)) {
3562                         if (err = zfs_secpolicy_write_perms(dsname,
3563                             ZFS_DELEG_PERM_USERPROP, cr))
3564                                 return (err);
3565                         return (0);
3566                 }
3567 
3568                 if (!issnap && zfs_prop_userquota(propname)) {
3569                         const char *perm = NULL;
3570                         const char *uq_prefix =
3571                             zfs_userquota_prop_prefixes[ZFS_PROP_USERQUOTA];
3572                         const char *gq_prefix =
3573                             zfs_userquota_prop_prefixes[ZFS_PROP_GROUPQUOTA];
3574 
3575                         if (strncmp(propname, uq_prefix,
3576                             strlen(uq_prefix)) == 0) {
3577                                 perm = ZFS_DELEG_PERM_USERQUOTA;
3578                         } else if (strncmp(propname, gq_prefix,
3579                             strlen(gq_prefix)) == 0) {
3580                                 perm = ZFS_DELEG_PERM_GROUPQUOTA;
3581                         } else {
3582                                 /* USERUSED and GROUPUSED are read-only */
3583                                 return (EINVAL);
3584                         }
3585 
3586                         if (err = zfs_secpolicy_write_perms(dsname, perm, cr))
3587                                 return (err);
3588                         return (0);
3589                 }
3590 
3591                 return (EINVAL);
3592         }
3593 
3594         if (issnap)
3595                 return (EINVAL);
3596 
3597         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
3598                 /*
3599                  * dsl_prop_get_all_impl() returns properties in this
3600                  * format.
3601                  */
3602                 nvlist_t *attrs;
3603                 VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
3604                 VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3605                     &pair) == 0);
3606         }
3607 
3608         /*
3609          * Check that this value is valid for this pool version
3610          */
3611         switch (prop) {
3612         case ZFS_PROP_COMPRESSION:
3613                 /*
3614                  * If the user specified gzip compression, make sure
3615                  * the SPA supports it. We ignore any errors here since
3616                  * we'll catch them later.
3617                  */
3618                 if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3619                     nvpair_value_uint64(pair, &intval) == 0) {
3620                         if (intval >= ZIO_COMPRESS_GZIP_1 &&
3621                             intval <= ZIO_COMPRESS_GZIP_9 &&
3622                             zfs_earlier_version(dsname,
```



```
3623                                 SPA_VERSION_GZIP_COMPRESSION)) {
3624                                 return (ENOTSUP);
3625                         }
3626 
3627                         if (intval == ZIO_COMPRESS_ZLE &&
3628                             zfs_earlier_version(dsname,
3629                             SPA_VERSION_ZLE_COMPRESSION))
3630                                 return (ENOTSUP);
3631 
3632                         /*
3633                          * If this is a bootable dataset then
3634                          * verify that the compression algorithm
3635                          * is supported for booting. We must return
3636                          * something other than ENOTSUP since it
3637                          * implies a downrev pool version.
3638                          */
3639                         if (zfs_is_bootfs(dsname) &&
3640                             !BOOTFS_COMPRESS_VALID(intval)) {
3641                                 return (ERANGE);
3642                         }
3643                 }
3644                 break;
3645 
3646         case ZFS_PROP_COPIES:
3647                 if (zfs_earlier_version(dsname, SPA_VERSION_DITTO_BLOCKS))
3648                         return (ENOTSUP);
3649                 break;
3650 
3651         case ZFS_PROP_DEDUP:
3652                 if (zfs_earlier_version(dsname, SPA_VERSION_DEDUP))
3653                         return (ENOTSUP);
3654                 break;
3655 
3656         case ZFS_PROP_SHARESMB:
3657                 if (zpl_earlier_version(dsname, ZPL_VERSION_FUID))
3658                         return (ENOTSUP);
3659                 break;
3660 
3661         case ZFS_PROP_ACLINHERIT:
3662                 if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3663                     nvpair_value_uint64(pair, &intval) == 0) {
3664                         if (intval == ZFS_ACL_PASSTHROUGH_X &&
3665                             zfs_earlier_version(dsname,
3666                             SPA_VERSION_PASSTHROUGH_X))
3667                                 return (ENOTSUP);
3668                 }
3669                 break;
3670         }
3671 
3672         return (zfs_secpolicy_setprop(dsname, prop, pair, CRED()));
3673 }
3674 
3675 /*
3676  * Removes properties from the given props list that fail permission checks
3677  * needed to clear them and to restore them in case of a receive error. For each
3678  * property, make sure we have both set and inherit permissions.
3679  *
3680  * Returns the first error encountered if any permission checks fail. If the
3681  * caller provides a non-NULL errlist, it also gives the complete list of names
3682  * of all the properties that failed a permission check along with the
3683  * corresponding error numbers. The caller is responsible for freeing the
3684  * returned errlist.
3685  *
3686  * If every property checks out successfully, zero is returned and the list
3687  * pointed at by errlist is NULL.
3688  */
```

```
3689 static int
3690 zfs_check_clearable(char *dataset, nvlist_t *props, nvlist_t **errlist)
3691 {
3692         zfs_cmd_t *zc;
3693         nvpair_t *pair, *next_pair;
3694         nvlist_t *errors;
3695         int err, rv = 0;

3697         if (props == NULL)
3698                 return (0);

3700         VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

3702         zc = kmem_alloc(sizeof (zfs_cmd_t), KM_SLEEP);
3703         (void) strcpy(zc->zc_name, dataset);
3704         pair = nvlist_next_nvpair(props, NULL);
3705         while (pair != NULL) {
3706                 next_pair = nvlist_next_nvpair(props, pair);

3708                 (void) strcpy(zc->zc_value, nvpair_name(pair));
3709                 if ((err = zfs_check_settable(dataset, pair, CRED())) != 0 ||
3710                     (err = zfs_secpolicy_inherit_prop(zc, NULL, CRED())) != 0) {
3711                         VERIFY(nvlist_remove_nvpair(props, pair) == 0);
3712                         VERIFY(nvlist_add_int32(errors,
3713                             zc->zc_value, err) == 0);
3714                 }
3715                 pair = next_pair;
3716         }
3717         kmem_free(zc, sizeof (zfs_cmd_t));

3719         if ((pair = nvlist_next_nvpair(errors, NULL)) == NULL) {
3720                 nvlist_free(errors);
3721                 errors = NULL;
3722         } else {
3723                 VERIFY(nvpair_value_int32(pair, &rv) == 0);
3724         }

3726         if (errlist == NULL)
3727                 nvlist_free(errors);
3728         else
3729                 *errlist = errors;

3731         return (rv);
3732 }

3734 static boolean_t
3735 propval_equals(nvpair_t *p1, nvpair_t *p2)
3736 {
3737         if (nvpair_type(p1) == DATA_TYPE_NVLIST) {
3738                 /* dsl_prop_get_all_impl() format */
3739                 nvlist_t *attrs;
3740                 VERIFY(nvpair_value_nvlist(p1, &attrs) == 0);
3741                 VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3742                     &p1) == 0);
3743         }

3745         if (nvpair_type(p2) == DATA_TYPE_NVLIST) {
3746                 nvlist_t *attrs;
3747                 VERIFY(nvpair_value_nvlist(p2, &attrs) == 0);
3748                 VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3749                     &p2) == 0);
3750         }

3752         if (nvpair_type(p1) != nvpair_type(p2))
3753                 return (B_FALSE);
```



```
3755         if (nvpair_type(p1) == DATA_TYPE_STRING) {
3756                 char *valstr1, *valstr2;

3758                 VERIFY(nvpair_value_string(p1, (char **)&valstr1) == 0);
3759                 VERIFY(nvpair_value_string(p2, (char **)&valstr2) == 0);
3760                 return (strcmp(valstr1, valstr2) == 0);
3761         } else {
3762                 uint64_t intval1, intval2;

3764                 VERIFY(nvpair_value_uint64(p1, &intval1) == 0);
3765                 VERIFY(nvpair_value_uint64(p2, &intval2) == 0);
3766                 return (intval1 == intval2);
3767         }
3768 }

3770 /*
3771  * Remove properties from props if they are not going to change (as determined
3772  * by comparison with origprops). Remove them from origprops as well, since we
3773  * do not need to clear or restore properties that won't change.
3774  */
3775 static void
3776 props_reduce(nvlist_t *props, nvlist_t *origprops)
3777 {
3778         nvpair_t *pair, *next_pair;

3780         if (origprops == NULL)
3781                 return; /* all props need to be received */

3783         pair = nvlist_next_nvpair(props, NULL);
3784         while (pair != NULL) {
3785                 const char *propname = nvpair_name(pair);
3786                 nvpair_t *match;

3788                 next_pair = nvlist_next_nvpair(props, pair);

3790                 if ((nvlist_lookup_nvpair(origprops, propname,
3791                     &match) != 0) || !propval_equals(pair, match))
3792                         goto next; /* need to set received value */

3794                 /* don't clear the existing received value */
3795                 (void) nvlist_remove_nvpair(origprops, match);
3796                 /* don't bother receiving the property */
3797                 (void) nvlist_remove_nvpair(props, pair);
3798 next:
3799                 pair = next_pair;
3800         }
3801 }

3803 #ifdef  DEBUG
3804 static boolean_t zfs_ioc_recv_inject_err;
3805 #endif

3807 /*
3808  * inputs:
3809  * zc_name              name of containing filesystem
3810  * zc_nvlist_src{_size} nvlist of properties to apply
3811  * zc_value             name of snapshot to create
3812  * zc_string            name of clone origin (if DRR_FLAG_CLONE)
3813  * zc_cookie            file descriptor to recv from
3814  * zc_begin_record      the BEGIN record of the stream (not byteswapped)
3815  * zc_guid              force flag
3816  * zc_cleanup_fd        cleanup-on-exit file descriptor
3817  * zc_action_handle     handle for this guid/ds mapping (or zero on first call)
3818  *
3819  * outputs:
3820  * zc_cookie            number of bytes read
```

```
 * zc_nvlist_dst{_size} error for each unapplied received property
 * zc_obj               zprop_errflags_t
 * zc_action_handle     handle for this guid/ds mapping
 */
static int
zfs_ioc_recv(zfs_cmd_t *zc)
{
        file_t *fp;
        objset_t *os;
        dmu_recv_cookie_t drc;
        boolean_t force = (boolean_t)zc->zc_guid;
        int fd;
        int error = 0;
        int props_error = 0;
        nvlist_t *errors;
        offset_t off;
        nvlist_t *props = NULL; /* sent properties */
        nvlist_t *origprops = NULL; /* existing properties */
        objset_t *origin = NULL;
        char *tosnap;
        char tofs[ZFS_MAXNAMELEN];
        boolean_t first_recvd_props = B_FALSE;

        if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
            strchr(zc->zc_value, '@') == NULL ||
            strchr(zc->zc_value, '%'))
                return (EINVAL);

        (void) strcpy(tofs, zc->zc_value);
        tosnap = strchr(tofs, '@');
        *tosnap++ = '\0';

        if (zc->zc_nvlist_src != NULL &&
            (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
            zc->zc_iflags, &props)) != 0)
                return (error);

        fd = zc->zc_cookie;
        fp = getf(fd);
        if (fp == NULL) {
                nvlist_free(props);
                return (EBADF);
        }

        VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

        if (props && dmu_objset_hold(tofs, FTAG, &os) == 0) {
                if ((spa_version(os->os_spa) >= SPA_VERSION_RECVD_PROPS) &&
                    !dsl_prop_get_hasrecvd(os)) {
                        first_recvd_props = B_TRUE;
                }

                /*
                 * If new received properties are supplied, they are to
                 * completely replace the existing received properties, so stash
                 * away the existing ones.
                 */
                if (dsl_prop_get_received(os, &origprops) == 0) {
                        nvlist_t *errlist = NULL;
                        /*
                         * Don't bother writing a property if its value won't
                         * change (and avoid the unnecessary security checks).
                         *
                         * The first receive after SPA_VERSION_RECVD_PROPS is a
                         * special case where we blow away all local properties
                         * regardless.
```



```
                         */
                        if (!first_recvd_props)
                                props_reduce(props, origprops);
                        if (zfs_check_clearable(tofs, origprops,
                            &errlist) != 0)
                                (void) nvlist_merge(errors, errlist, 0);
                        nvlist_free(errlist);
                }

                dmu_objset_rele(os, FTAG);
        }

        if (zc->zc_string[0]) {
                error = dmu_objset_hold(zc->zc_string, FTAG, &origin);
                if (error)
                        goto out;
        }

        error = dmu_recv_begin(tofs, tosnap, zc->zc_top_ds,
            &zc->zc_begin_record, force, origin, &drc);
        if (origin)
                dmu_objset_rele(origin, FTAG);
        if (error)
                goto out;

        /*
         * Set properties before we receive the stream so that they are applied
         * to the new data. Note that we must call dmu_recv_stream() if
         * dmu_recv_begin() succeeds.
         */
        if (props) {
                if (dmu_objset_from_ds(drc.drc_logical_ds, &os) == 0) {
                        if (drc.drc_newfs) {
                                if (spa_version(os->os_spa) >=
                                    SPA_VERSION_RECVD_PROPS)
                                        first_recvd_props = B_TRUE;
                        } else if (origprops != NULL) {
                                if (clear_received_props(os, tofs, origprops,
                                    first_recvd_props ? NULL : props) != 0)
                                        zc->zc_obj |= ZPROP_ERR_NOCLEAR;
                        } else {
                                zc->zc_obj |= ZPROP_ERR_NOCLEAR;
                        }
                        dsl_prop_set_hasrecvd(os);
                } else if (!drc.drc_newfs) {
                        zc->zc_obj |= ZPROP_ERR_NOCLEAR;
                }

                (void) zfs_set_prop_nvlist(tofs, ZPROP_SRC_RECEIVED,
                    props, errors);
        }

        if (zc->zc_nvlist_dst_size != 0 &&
            (nvlist_smush(errors, zc->zc_nvlist_dst_size) != 0 ||
            put_nvlist(zc, errors) != 0)) {
                /*
                 * Caller made zc->zc_nvlist_dst less than the minimum expected
                 * size or supplied an invalid address.
                 */
                props_error = EINVAL;
        }

        off = fp->f_offset;
        error = dmu_recv_stream(&drc, fp->f_vnode, &off, zc->zc_cleanup_fd,
            &zc->zc_action_handle);
```

```
3953         if (error == 0) {
3954                 zfsvfs_t *zfsvfs = NULL;
3955
3956                 if (getzfsvfs(tofs, &zfsvfs) == 0) {
3957                         /* online recv */
3958                         int end_err;
3959
3960                         error = zfs_suspend_fs(zfsvfs);
3961                         /*
3962                          * If the suspend fails, then the recv_end will
3963                          * likely also fail, and clean up after itself.
3964                          */
3965                         end_err = dmu_recv_end(&drc);
3966                         if (error == 0)
3967                                 error = zfs_resume_fs(zfsvfs, tofs);
3968                         error = error ? error : end_err;
3969                         VFS_RELE(zfsvfs->z_vfs);
3970                 } else {
3971                         error = dmu_recv_end(&drc);
3972                 }
3973         }
3974
3975         zc->zc_cookie = off - fp->f_offset;
3976         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
3977                 fp->f_offset = off;
3978
3979 #ifdef  DEBUG
3980         if (zfs_ioc_recv_inject_err) {
3981                 zfs_ioc_recv_inject_err = B_FALSE;
3982                 error = 1;
3983         }
3984 #endif
3985         /*
3986          * On error, restore the original props.
3987          */
3988         if (error && props) {
3989                 if (dmu_objset_hold(tofs, FTAG, &os) == 0) {
3990                         if (clear_received_props(os, tofs, props, NULL) != 0) {
3991                                 /*
3992                                  * We failed to clear the received properties.
3993                                  * Since we may have left a $recvd value on the
3994                                  * system, we can't clear the $hasrecvd flag.
3995                                  */
3996                                 zc->zc_obj |= ZPROP_ERR_NORESTORE;
3997                         } else if (first_recvd_props) {
3998                                 dsl_prop_unset_hasrecvd(os);
3999                         }
4000                         dmu_objset_rele(os, FTAG);
4001                 } else if (!drc.drc_newfs) {
4002                         /* We failed to clear the received properties. */
4003                         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4004                 }
4005
4006                 if (origprops == NULL && !drc.drc_newfs) {
4007                         /* We failed to stash the original properties. */
4008                         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4009                 }
4010
4011                 /*
4012                  * dsl_props_set() will not convert RECEIVED to LOCAL on or
4013                  * after SPA_VERSION_RECVD_PROPS, so we need to specify LOCAL
4014                  * explictly if we're restoring local properties cleared in the
4015                  * first new-style receive.
4016                  */
4017                 if (origprops != NULL &&
4018                     zfs_set_prop_nvlist(tofs, (first_recvd_props ?
```



```
4019                         ZPROP_SRC_LOCAL : ZPROP_SRC_RECEIVED),
4020                         origprops, NULL) != 0) {
4021                         /*
4022                          * We stashed the original properties but failed to
4023                          * restore them.
4024                          */
4025                         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4026                 }
4027         }
4028 out:
4029         nvlist_free(props);
4030         nvlist_free(origprops);
4031         nvlist_free(errors);
4032         releasef(fd);
4033
4034         if (error == 0)
4035                 error = props_error;
4036
4037         return (error);
4038 }
4039
4040 /*
4041  * inputs:
4042  * zc_name      name of snapshot to send
4043  * zc_cookie    file descriptor to send stream to
4044  * zc_obj       fromorigin flag (mutually exclusive with zc_fromobj)
4045  * zc_sendobj   objsetid of snapshot to send
4046  * zc_fromobj   objsetid of incremental fromsnap (may be zero)
4047  * zc_guid      if set, estimate size of stream only.  zc_cookie is ignored.
4048  *              output size in zc_objset_type.
4049  *
4050  * outputs: none
4051  */
4052 static int
4053 zfs_ioc_send(zfs_cmd_t *zc)
4054 {
4055         objset_t *fromsnap = NULL;
4056         objset_t *tosnap;
4057         int error;
4058         offset_t off;
4059         dsl_dataset_t *ds;
4060         dsl_dataset_t *dsfrom = NULL;
4061         spa_t *spa;
4062         dsl_pool_t *dp;
4063         boolean_t estimate = (zc->zc_guid != 0);
4064
4065         error = spa_open(zc->zc_name, &spa, FTAG);
4066         if (error)
4067                 return (error);
4068
4069         dp = spa_get_dsl(spa);
4070         rw_enter(&dp->dp_config_rwlock, RW_READER);
4071         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
4072         rw_exit(&dp->dp_config_rwlock);
4073         spa_close(spa, FTAG);
4074         if (error)
4075                 return (error);
4076
4077         error = dmu_objset_from_ds(ds, &tosnap);
4078         if (error) {
4079                 dsl_dataset_rele(ds, FTAG);
4080                 return (error);
4081         }
4082
4083         if (zc->zc_fromobj != 0) {
4084                 rw_enter(&dp->dp_config_rwlock, RW_READER);
```

```
4085                error = dsl_dataset_hold_obj(dp, zc->zc_fromobj, FTAG, &dsfrom);
4086                rw_exit(&dp->dp_config_rwlock);
4087                if (error) {
4088                        dsl_dataset_rele(ds, FTAG);
4089                        return (error);
4090                }
4091                error = dmu_objset_from_ds(dsfrom, &fromsnap);
4092                if (error) {
4093                        dsl_dataset_rele(dsfrom, FTAG);
4094                        dsl_dataset_rele(ds, FTAG);
4095                        return (error);
4096                }
4097        }

4099        if (zc->zc_obj) {
4100                dsl_pool_t *dp = ds->ds_dir->dd_pool;

4102                if (fromsnap != NULL) {
4103                        dsl_dataset_rele(dsfrom, FTAG);
4104                        dsl_dataset_rele(ds, FTAG);
4105                        return (EINVAL);
4106                }

4108                if (dsl_dir_is_clone(ds->ds_dir)) {
4109                        rw_enter(&dp->dp_config_rwlock, RW_READER);
4110                        error = dsl_dataset_hold_obj(dp,
4111                            ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &dsfrom);
4112                        rw_exit(&dp->dp_config_rwlock);
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        }

4126        if (estimate) {
4127                error = dmu_send_estimate(tosnap, fromsnap,
4128                    &zc->zc_objset_type);
4129        } else {
4130                file_t *fp = getf(zc->zc_cookie);
4131                if (fp == NULL) {
4132                        dsl_dataset_rele(ds, FTAG);
4133                        if (dsfrom)
4134                                dsl_dataset_rele(dsfrom, FTAG);
4135                        return (EBADF);
4136                }

4138                off = fp->f_offset;
4139                error = dmu_send(tosnap, fromsnap,
4140                    zc->zc_cookie, fp->f_vnode, &off);

4142                if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4143                        fp->f_offset = off;
4144                releasef(zc->zc_cookie);
4145        }
4146        if (dsfrom)
4147                dsl_dataset_rele(dsfrom, FTAG);
4148        dsl_dataset_rele(ds, FTAG);
4149        return (error);
4150 }
```



```
4152 /*
4153  * inputs:
4154  * zc_name      name of snapshot on which to report progress
4155  * zc_cookie    file descriptor of send stream
4156  *
4157  * outputs:
4158  * zc_cookie    number of bytes written in send stream thus far
4159  */
4160 static int
4161 zfs_ioc_send_progress(zfs_cmd_t *zc)
4162 {
4163         dsl_dataset_t *ds;
4164         dmu_sendarg_t *dsp = NULL;
4165         int error;

4167         if ((error = dsl_dataset_hold(zc->zc_name, FTAG, &ds)) != 0)
4168                 return (error);

4170         mutex_enter(&ds->ds_sendstream_lock);

4172         /*
4173          * Iterate over all the send streams currently active on this dataset.
4174          * If there's one which matches the specified file descriptor _and_ the
4175          * stream was started by the current process, return the progress of
4176          * that stream.
4177          */
4178         for (dsp = list_head(&ds->ds_sendstreams); dsp != NULL;
4179             dsp = list_next(&ds->ds_sendstreams, dsp)) {
4180                 if (dsp->dsa_outfd == zc->zc_cookie &&
4181                     dsp->dsa_proc == curproc)
4182                         break;
4183         }

4185         if (dsp != NULL)
4186                 zc->zc_cookie = *(dsp->dsa_off);
4187         else
4188                 error = ENOENT;

4190         mutex_exit(&ds->ds_sendstream_lock);
4191         dsl_dataset_rele(ds, FTAG);
4192         return (error);
4193 }

4195 static int
4196 zfs_ioc_inject_fault(zfs_cmd_t *zc)
4197 {
4198         int id, error;

4200         error = zio_inject_fault(zc->zc_name, (int)zc->zc_guid, &id,
4201             &zc->zc_inject_record);

4203         if (error == 0)
4204                 zc->zc_guid = (uint64_t)id;

4206         return (error);
4207 }

4209 static int
4210 zfs_ioc_clear_fault(zfs_cmd_t *zc)
4211 {
4212         return (zio_clear_fault((int)zc->zc_guid));
4213 }

4215 static int
4216 zfs_ioc_inject_list_next(zfs_cmd_t *zc)
```

```
4217 {
4218         int id = (int)zc->zc_guid;
4219         int error;

4221         error = zio_inject_list_next(&id, zc->zc_name, sizeof (zc->zc_name),
4222             &zc->zc_inject_record);

4224         zc->zc_guid = id;

4226         return (error);
4227 }

4229 static int
4230 zfs_ioc_error_log(zfs_cmd_t *zc)
4231 {
4232         spa_t *spa;
4233         int error;
4234         size_t count = (size_t)zc->zc_nvlist_dst_size;

4236         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
4237                 return (error);

4239         error = spa_get_errlog(spa, (void *)(uintptr_t)zc->zc_nvlist_dst,
4240             &count);
4241         if (error == 0)
4242                 zc->zc_nvlist_dst_size = count;
4243         else
4244                 zc->zc_nvlist_dst_size = spa_get_errlog_size(spa);

4246         spa_close(spa, FTAG);

4248         return (error);
4249 }

4251 static int
4252 zfs_ioc_clear(zfs_cmd_t *zc)
4253 {
4254         spa_t *spa;
4255         vdev_t *vd;
4256         int error;

4258         /*
4259          * On zpool clear we also fix up missing slogs
4260          */
4261         mutex_enter(&spa_namespace_lock);
4262         spa = spa_lookup(zc->zc_name);
4263         if (spa == NULL) {
4264                 mutex_exit(&spa_namespace_lock);
4265                 return (EIO);
4266         }
4267         if (spa_get_log_state(spa) == SPA_LOG_MISSING) {
4268                 /* we need to let spa_open/spa_load clear the chains */
4269                 spa_set_log_state(spa, SPA_LOG_CLEAR);
4270         }
4271         spa->spa_last_open_failed = 0;
4272         mutex_exit(&spa_namespace_lock);

4274         if (zc->zc_cookie & ZPOOL_NO_REWIND) {
4275                 error = spa_open(zc->zc_name, &spa, FTAG);
4276         } else {
4277                 nvlist_t *policy;
4278                 nvlist_t *config = NULL;

4280                 if (zc->zc_nvlist_src == NULL)
4281                         return (EINVAL);
```



```
4283                 if ((error = get_nvlist(zc->zc_nvlist_src,
4284                     zc->zc_nvlist_src_size, zc->zc_iflags, &policy)) == 0) {
4285                         error = spa_open_rewind(zc->zc_name, &spa, FTAG,
4286                             policy, &config);
4287                         if (config != NULL) {
4288                                 int err;

4290                                 if ((err = put_nvlist(zc, config)) != 0)
4291                                         error = err;
4292                                 nvlist_free(config);
4293                         }
4294                         nvlist_free(policy);
4295                 }
4296         }

4298         if (error)
4299                 return (error);

4301         spa_vdev_state_enter(spa, SCL_NONE);

4303         if (zc->zc_guid == 0) {
4304                 vd = NULL;
4305         } else {
4306                 vd = spa_lookup_by_guid(spa, zc->zc_guid, B_TRUE);
4307                 if (vd == NULL) {
4308                         (void) spa_vdev_state_exit(spa, NULL, ENODEV);
4309                         spa_close(spa, FTAG);
4310                         return (ENODEV);
4311                 }
4312         }

4314         vdev_clear(spa, vd);

4316         (void) spa_vdev_state_exit(spa, NULL, 0);

4318         /*
4319          * Resume any suspended I/Os.
4320          */
4321         if (zio_resume(spa) != 0)
4322                 error = EIO;

4324         spa_close(spa, FTAG);

4326         return (error);
4327 }

4329 static int
4330 zfs_ioc_pool_reopen(zfs_cmd_t *zc)
4331 {
4332         spa_t *spa;
4333         int error;

4335         error = spa_open(zc->zc_name, &spa, FTAG);
4336         if (error)
4337                 return (error);

4339         spa_vdev_state_enter(spa, SCL_NONE);

4341         /*
4342          * If a resilver is already in progress then set the
4343          * spa_scrub_reopen flag to B_TRUE so that we don't restart
4344          * the scan as a side effect of the reopen. Otherwise, let
4345          * vdev_open() decided if a resilver is required.
4346          */
4347         spa->spa_scrub_reopen = dsl_scan_resilvering(spa->spa_dsl_pool);
4348         vdev_reopen(spa->spa_root_vdev);
```

```
4349         spa->spa_scrub_reopen = B_FALSE;

4351         (void) spa_vdev_state_exit(spa, NULL, 0);
4352         spa_close(spa, FTAG);
4353         return (0);
4354 }
4355 /*
4356  * inputs:
4357  * zc_name      name of filesystem
4358  * zc_value     name of origin snapshot
4359  *
4360  * outputs:
4361  * zc_string    name of conflicting snapshot, if there is one
4362  */
4363 static int
4364 zfs_ioc_promote(zfs_cmd_t *zc)
4365 {
4366         char *cp;

4368         /*
4369          * We don't need to unmount *all* the origin fs's snapshots, but
4370          * it's easier.
4371          */
4372         cp = strchr(zc->zc_value, '@');
4373         if (cp)
4374                 *cp = '\0';
4375         (void) dmu_objset_find(zc->zc_value,
4376             zfs_unmount_snap, NULL, DS_FIND_SNAPSHOTS);
4377         return (dsl_dataset_promote(zc->zc_name, zc->zc_string));
4378 }

4380 /*
4381  * Retrieve a single {user|group}{used|quota}@... property.
4382  *
4383  * inputs:
4384  * zc_name      name of filesystem
4385  * zc_objset_type zfs_userquota_prop_t
4386  * zc_value     domain name (eg. "S-1-234-567-89")
4387  * zc_guid      RID/UID/GID
4388  *
4389  * outputs:
4390  * zc_cookie    property value
4391  */
4392 static int
4393 zfs_ioc_userspace_one(zfs_cmd_t *zc)
4394 {
4395         zfsvfs_t *zfsvfs;
4396         int error;

4398         if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
4399                 return (EINVAL);

4401         error = zfsvfs_hold(zc->zc_name, FTAG, &zfsvfs, B_FALSE);
4402         if (error)
4403                 return (error);

4405         error = zfs_userspace_one(zfsvfs,
4406             zc->zc_objset_type, zc->zc_value, zc->zc_guid, &zc->zc_cookie);
4407         zfsvfs_rele(zfsvfs, FTAG);

4409         return (error);
4410 }

4412 /*
4413  * inputs:
4414  * zc_name              name of filesystem
```



```
4415  * zc_cookie            zap cursor
4416  * zc_objset_type       zfs_userquota_prop_t
4417  * zc_nvlist_dst[_size] buffer to fill (not really an nvlist)
4418  *
4419  * outputs:
4420  * zc_nvlist_dst[_size] data buffer (array of zfs_useracct_t)
4421  * zc_cookie    zap cursor
4422  */
4423 static int
4424 zfs_ioc_userspace_many(zfs_cmd_t *zc)
4425 {
4426         zfsvfs_t *zfsvfs;
4427         int bufsize = zc->zc_nvlist_dst_size;

4429         if (bufsize <= 0)
4430                 return (ENOMEM);

4432         int error = zfsvfs_hold(zc->zc_name, FTAG, &zfsvfs, B_FALSE);
4433         if (error)
4434                 return (error);

4436         void *buf = kmem_alloc(bufsize, KM_SLEEP);

4438         error = zfs_userspace_many(zfsvfs, zc->zc_objset_type, &zc->zc_cookie,
4439             buf, &zc->zc_nvlist_dst_size);

4441         if (error == 0) {
4442                 error = xcopyout(buf,
4443                     (void *)(uintptr_t)zc->zc_nvlist_dst,
4444                     zc->zc_nvlist_dst_size);
4445         }
4446         kmem_free(buf, bufsize);
4447         zfsvfs_rele(zfsvfs, FTAG);

4449         return (error);
4450 }

4452 /*
4453  * inputs:
4454  * zc_name              name of filesystem
4455  *
4456  * outputs:
4457  * none
4458  */
4459 static int
4460 zfs_ioc_userspace_upgrade(zfs_cmd_t *zc)
4461 {
4462         objset_t *os;
4463         int error = 0;
4464         zfsvfs_t *zfsvfs;

4466         if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
4467                 if (!dmu_objset_userused_enabled(zfsvfs->z_os)) {
4468                         /*
4469                          * If userused is not enabled, it may be because the
4470                          * objset needs to be closed & reopened (to grow the
4471                          * objset_phys_t).  Suspend/resume the fs will do that.
4472                          */
4473                         error = zfs_suspend_fs(zfsvfs);
4474                         if (error == 0)
4475                                 error = zfs_resume_fs(zfsvfs, zc->zc_name);
4476                 }
4477                 if (error == 0)
4478                         error = dmu_objset_userspace_upgrade(zfsvfs->z_os);
4479                 VFS_RELE(zfsvfs->z_vfs);
4480         } else {
```

```
4481                 /* XXX kind of reading contents without owning */
4482                 error = dmu_objset_hold(zc->zc_name, FTAG, &os);
4483                 if (error)
4484                         return (error);
4485
4486                 error = dmu_objset_userspace_upgrade(os);
4487                 dmu_objset_rele(os, FTAG);
4488         }
4489
4490         return (error);
4491 }
4492
4493 /*
4494  * We don't want to have a hard dependency
4495  * against some special symbols in sharefs
4496  * nfs, and smbsrv.  Determine them if needed when
4497  * the first file system is shared.
4498  * Neither sharefs, nfs or smbsrv are unloadable modules.
4499  */
4500 int (*znfsexport_fs)(void *arg);
4501 int (*zshare_fs)(enum sharefs_sys_op, share_t *, uint32_t);
4502 int (*zsmbexport_fs)(void *arg, boolean_t add_share);
4503
4504 int zfs_nfsshare_inited;
4505 int zfs_smbshare_inited;
4506
4507 ddi_modhandle_t nfs_mod;
4508 ddi_modhandle_t sharefs_mod;
4509 ddi_modhandle_t smbsrv_mod;
4510 kmutex_t zfs_share_lock;
4511
4512 static int
4513 zfs_init_sharefs()
4514 {
4515         int error;
4516
4517         ASSERT(MUTEX_HELD(&zfs_share_lock));
4518         /* Both NFS and SMB shares also require sharetab support. */
4519         if (sharefs_mod == NULL && ((sharefs_mod =
4520             ddi_modopen("fs/sharefs",
4521             KRTLD_MODE_FIRST, &error)) == NULL)) {
4522                 return (ENOSYS);
4523         }
4524         if (zshare_fs == NULL && ((zshare_fs =
4525             (int (*)(enum sharefs_sys_op, share_t *, uint32_t))
4526             ddi_modsym(sharefs_mod, "sharefs_impl", &error)) == NULL)) {
4527                 return (ENOSYS);
4528         }
4529         return (0);
4530 }
4531
4532 static int
4533 zfs_ioc_share(zfs_cmd_t *zc)
4534 {
4535         int error;
4536         int opcode;
4537
4538         switch (zc->zc_share.z_sharetype) {
4539         case ZFS_SHARE_NFS:
4540         case ZFS_UNSHARE_NFS:
4541                 if (zfs_nfsshare_inited == 0) {
4542                         mutex_enter(&zfs_share_lock);
4543                         if (nfs_mod == NULL && ((nfs_mod = ddi_modopen("fs/nfs",
4544                             KRTLD_MODE_FIRST, &error)) == NULL)) {
4545                                 mutex_exit(&zfs_share_lock);
4546                                 return (ENOSYS);
```



```
4547                         }
4548                         if (znfsexport_fs == NULL &&
4549                             ((znfsexport_fs = (int (*)(void *))
4550                             ddi_modsym(nfs_mod,
4551                             "nfs_export", &error)) == NULL)) {
4552                                 mutex_exit(&zfs_share_lock);
4553                                 return (ENOSYS);
4554                         }
4555                         error = zfs_init_sharefs();
4556                         if (error) {
4557                                 mutex_exit(&zfs_share_lock);
4558                                 return (ENOSYS);
4559                         }
4560                         zfs_nfsshare_inited = 1;
4561                         mutex_exit(&zfs_share_lock);
4562                 }
4563                 break;
4564         case ZFS_SHARE_SMB:
4565         case ZFS_UNSHARE_SMB:
4566                 if (zfs_smbshare_inited == 0) {
4567                         mutex_enter(&zfs_share_lock);
4568                         if (smbsrv_mod == NULL && ((smbsrv_mod =
4569                             ddi_modopen("drv/smbsrv",
4570                             KRTLD_MODE_FIRST, &error)) == NULL)) {
4571                                 mutex_exit(&zfs_share_lock);
4572                                 return (ENOSYS);
4573                         }
4574                         if (zsmbexport_fs == NULL && ((zsmbexport_fs =
4575                             (int (*)(void *, boolean_t))ddi_modsym(smbsrv_mod,
4576                             "smb_server_share", &error)) == NULL)) {
4577                                 mutex_exit(&zfs_share_lock);
4578                                 return (ENOSYS);
4579                         }
4580                         error = zfs_init_sharefs();
4581                         if (error) {
4582                                 mutex_exit(&zfs_share_lock);
4583                                 return (ENOSYS);
4584                         }
4585                         zfs_smbshare_inited = 1;
4586                         mutex_exit(&zfs_share_lock);
4587                 }
4588                 break;
4589         default:
4590                 return (EINVAL);
4591         }
4592
4593         switch (zc->zc_share.z_sharetype) {
4594         case ZFS_SHARE_NFS:
4595         case ZFS_UNSHARE_NFS:
4596                 if (error =
4597                     znfsexport_fs((void *)
4598                     (uintptr_t)zc->zc_share.z_exportdata))
4599                         return (error);
4600                 break;
4601         case ZFS_SHARE_SMB:
4602         case ZFS_UNSHARE_SMB:
4603                 if (error = zsmbexport_fs((void *)
4604                     (uintptr_t)zc->zc_share.z_exportdata,
4605                     zc->zc_share.z_sharetype == ZFS_SHARE_SMB ?
4606                     B_TRUE: B_FALSE)) {
4607                         return (error);
4608                 }
4609                 break;
4610         }
4611
4612         opcode = (zc->zc_share.z_sharetype == ZFS_SHARE_NFS ||
```

```
4613             zc->zc_share.z_sharetype == ZFS_SHARE_SMB) ?
4614             SHAREFS_ADD : SHAREFS_REMOVE;

4616         /*
4617          * Add or remove share from sharetab
4618          */
4619         error = zshare_fs(opcode,
4620             (void *)(uintptr_t)zc->zc_share.z_sharedata,
4621             zc->zc_share.z_sharemax);

4623         return (error);

4625 }

4627 ace_t full_access[] = {
4628         {(uid_t)-1, ACE_ALL_PERMS, ACE_EVERYONE, 0}
4629 };

4631 /*
4632  * inputs:
4633  * zc_name              name of containing filesystem
4634  * zc_obj               object # beyond which we want next in-use object #
4635  *
4636  * outputs:
4637  * zc_obj               next in-use object #
4638  */
4639 static int
4640 zfs_ioc_next_obj(zfs_cmd_t *zc)
4641 {
4642         objset_t *os = NULL;
4643         int error;

4645         error = dmu_objset_hold(zc->zc_name, FTAG, &os);
4646         if (error)
4647                 return (error);

4649         error = dmu_object_next(os, &zc->zc_obj, B_FALSE,
4650             os->os_dsl_dataset->ds_phys->ds_prev_snap_txg);

4652         dmu_objset_rele(os, FTAG);
4653         return (error);
4654 }

4656 /*
4657  * inputs:
4658  * zc_name              name of filesystem
4659  * zc_value             prefix name for snapshot
4660  * zc_cleanup_fd        cleanup-on-exit file descriptor for calling process
4661  *
4662  * outputs:
4663  * zc_value             short name of new snapshot
4664  */
4665 static int
4666 zfs_ioc_tmp_snapshot(zfs_cmd_t *zc)
4667 {
4668         char *snap_name;
4669         int error;

4671         snap_name = kmem_asprintf("%s@%s-%016llx", zc->zc_name, zc->zc_value,
4672             (u_longlong_t)ddi_get_lbolt64());

4674         if (strlen(snap_name) >= MAXPATHLEN) {
4675                 strfree(snap_name);
4676                 return (E2BIG);
4677         }
```



```
4679         error = dmu_objset_snapshot_tmp(snap_name, "%temp", zc->zc_cleanup_fd);
4680         if (error != 0) {
4681                 strfree(snap_name);
4682                 return (error);
4683         }

4685         (void) strcpy(zc->zc_value, strchr(snap_name, '@') + 1);
4686         strfree(snap_name);
4687         return (0);
4688 }

4690 /*
4691  * inputs:
4692  * zc_name              name of "to" snapshot
4693  * zc_value             name of "from" snapshot
4694  * zc_cookie            file descriptor to write diff data on
4695  *
4696  * outputs:
4697  * dmu_diff_record_t's to the file descriptor
4698  */
4699 static int
4700 zfs_ioc_diff(zfs_cmd_t *zc)
4701 {
4702         objset_t *fromsnap;
4703         objset_t *tosnap;
4704         file_t *fp;
4705         offset_t off;
4706         int error;

4708         error = dmu_objset_hold(zc->zc_name, FTAG, &tosnap);
4709         if (error)
4710                 return (error);

4712         error = dmu_objset_hold(zc->zc_value, FTAG, &fromsnap);
4713         if (error) {
4714                 dmu_objset_rele(tosnap, FTAG);
4715                 return (error);
4716         }

4718         fp = getf(zc->zc_cookie);
4719         if (fp == NULL) {
4720                 dmu_objset_rele(fromsnap, FTAG);
4721                 dmu_objset_rele(tosnap, FTAG);
4722                 return (EBADF);
4723         }

4725         off = fp->f_offset;

4727         error = dmu_diff(tosnap, fromsnap, fp->f_vnode, &off);

4729         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4730                 fp->f_offset = off;
4731         releasef(zc->zc_cookie);

4733         dmu_objset_rele(fromsnap, FTAG);
4734         dmu_objset_rele(tosnap, FTAG);
4735         return (error);
4736 }

4738 /*
4739  * Remove all ACL files in shares dir
4740  */
4741 static int
4742 zfs_smb_acl_purge(znode_t *dzp)
4743 {
4744         zap_cursor_t    zc;
```

```
4745         zap_attribute_t zap;
4746         zfsvfs_t *zfsvfs = dzp->z_zfsvfs;
4747         int error;

4749         for (zap_cursor_init(&zc, zfsvfs->z_os, dzp->z_id);
4750             (error = zap_cursor_retrieve(&zc, &zap)) == 0;
4751             zap_cursor_advance(&zc)) {
4752                 if ((error = VOP_REMOVE(ZTOV(dzp), zap.za_name, kcred,
4753                     NULL, 0)) != 0)
4754                         break;
4755         }
4756         zap_cursor_fini(&zc);
4757         return (error);
4758 }

4760 static int
4761 zfs_ioc_smb_acl(zfs_cmd_t *zc)
4762 {
4763         vnode_t *vp;
4764         znode_t *dzp;
4765         vnode_t *resourcevp = NULL;
4766         znode_t *sharedir;
4767         zfsvfs_t *zfsvfs;
4768         nvlist_t *nvlist;
4769         char *src, *target;
4770         vattr_t vattr;
4771         vsecattr_t vsec;
4772         int error = 0;

4774         if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
4775             NO_FOLLOW, NULL, &vp)) != 0)
4776                 return (error);

4778         /* Now make sure mntpnt and dataset are ZFS */

4780         if (vp->v_vfsp->vfs_fstype != zfsfstype ||
4781             (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
4782             zc->zc_name) != 0)) {
4783                 VN_RELE(vp);
4784                 return (EINVAL);
4785         }

4787         dzp = VTOZ(vp);
4788         zfsvfs = dzp->z_zfsvfs;
4789         ZFS_ENTER(zfsvfs);

4791         /*
4792          * Create share dir if its missing.
4793          */
4794         mutex_enter(&zfsvfs->z_lock);
4795         if (zfsvfs->z_shares_dir == 0) {
4796                 dmu_tx_t *tx;

4798                 tx = dmu_tx_create(zfsvfs->z_os);
4799                 dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, TRUE,
4800                     ZFS_SHARES_DIR);
4801                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
4802                 error = dmu_tx_assign(tx, TXG_WAIT);
4803                 if (error) {
4804                         dmu_tx_abort(tx);
4805                 } else {
4806                         error = zfs_create_share_dir(zfsvfs, tx);
4807                         dmu_tx_commit(tx);
4808                 }
4809                 if (error) {
4810                         mutex_exit(&zfsvfs->z_lock);
```



```
4811                         VN_RELE(vp);
4812                         ZFS_EXIT(zfsvfs);
4813                         return (error);
4814                 }
4815         }
4816         mutex_exit(&zfsvfs->z_lock);

4818         ASSERT(zfsvfs->z_shares_dir);
4819         if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &sharedir)) != 0) {
4820                 VN_RELE(vp);
4821                 ZFS_EXIT(zfsvfs);
4822                 return (error);
4823         }

4825         switch (zc->zc_cookie) {
4826         case ZFS_SMB_ACL_ADD:
4827                 vattr.va_mask = AT_MODE|AT_UID|AT_GID|AT_TYPE;
4828                 vattr.va_type = VREG;
4829                 vattr.va_mode = S_IFREG|0777;
4830                 vattr.va_uid = 0;
4831                 vattr.va_gid = 0;

4833                 vsec.vsa_mask = VSA_ACE;
4834                 vsec.vsa_aclentp = &full_access;
4835                 vsec.vsa_aclentsz = sizeof (full_access);
4836                 vsec.vsa_aclcnt = 1;

4838                 error = VOP_CREATE(ZTOV(sharedir), zc->zc_string,
4839                     &vattr, EXCL, 0, &resourcevp, kcred, 0, NULL, &vsec);
4840                 if (resourcevp)
4841                         VN_RELE(resourcevp);
4842                 break;

4844         case ZFS_SMB_ACL_REMOVE:
4845                 error = VOP_REMOVE(ZTOV(sharedir), zc->zc_string, kcred,
4846                     NULL, 0);
4847                 break;

4849         case ZFS_SMB_ACL_RENAME:
4850                 if ((error = get_nvlist(zc->zc_nvlist_src,
4851                     zc->zc_nvlist_src_size, zc->zc_iflags, &nvlist)) != 0) {
4852                         VN_RELE(vp);
4853                         ZFS_EXIT(zfsvfs);
4854                         return (error);
4855                 }
4856                 if (nvlist_lookup_string(nvlist, ZFS_SMB_ACL_SRC, &src) ||
4857                     nvlist_lookup_string(nvlist, ZFS_SMB_ACL_TARGET,
4858                     &target)) {
4859                         VN_RELE(vp);
4860                         VN_RELE(ZTOV(sharedir));
4861                         ZFS_EXIT(zfsvfs);
4862                         nvlist_free(nvlist);
4863                         return (error);
4864                 }
4865                 error = VOP_RENAME(ZTOV(sharedir), src, ZTOV(sharedir), target,
4866                     kcred, NULL, 0);
4867                 nvlist_free(nvlist);
4868                 break;

4870         case ZFS_SMB_ACL_PURGE:
4871                 error = zfs_smb_acl_purge(sharedir);
4872                 break;

4874         default:
4875                 error = EINVAL;
4876                 break;
```

```
4877         }

4879         VN_RELE(vp);
4880         VN_RELE(ZTOV(sharedir));

4882         ZFS_EXIT(zfsvfs);

4884         return (error);
4885 }

4887 /*
4888  * inputs:
4889  * zc_name              name of filesystem
4890  * zc_value             short name of snap
4891  * zc_string            user-supplied tag for this hold
4892  * zc_cookie            recursive flag
4893  * zc_temphold          set if hold is temporary
4894  * zc_cleanup_fd        cleanup-on-exit file descriptor for calling process
4895  * zc_sendobj           if non-zero, the objid for zc_name@zc_value
4896  * zc_createtxg         if zc_sendobj is non-zero, snap must have zc_createtxg
4897  *
4898  * outputs:             none
4899  */
4900 static int
4901 zfs_ioc_hold(zfs_cmd_t *zc)
4902 {
4903         boolean_t recursive = zc->zc_cookie;
4904         spa_t *spa;
4905         dsl_pool_t *dp;
4906         dsl_dataset_t *ds;
4907         int error;
4908         minor_t minor = 0;

4910         if (snapshot_namecheck(zc->zc_value, NULL, NULL) != 0)
4911                 return (EINVAL);

4913         if (zc->zc_sendobj == 0) {
4914                 return (dsl_dataset_user_hold(zc->zc_name, zc->zc_value,
4915                     zc->zc_string, recursive, zc->zc_temphold,
4916                     zc->zc_cleanup_fd));
4917         }

4919         if (recursive)
4920                 return (EINVAL);

4922         error = spa_open(zc->zc_name, &spa, FTAG);
4923         if (error)
4924                 return (error);

4926         dp = spa_get_dsl(spa);
4927         rw_enter(&dp->dp_config_rwlock, RW_READER);
4928         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
4929         rw_exit(&dp->dp_config_rwlock);
4930         spa_close(spa, FTAG);
4931         if (error)
4932                 return (error);

4934         /*
4935          * Until we have a hold on this snapshot, it's possible that
4936          * zc_sendobj could've been destroyed and reused as part
4937          * of a later txg.  Make sure we're looking at the right object.
4938          */
4939         if (zc->zc_createtxg != ds->ds_phys->ds_creation_txg) {
4940                 dsl_dataset_rele(ds, FTAG);
4941                 return (ENOENT);
4942         }
```



```
4944         if (zc->zc_cleanup_fd != -1 && zc->zc_temphold) {
4945                 error = zfs_onexit_fd_hold(zc->zc_cleanup_fd, &minor);
4946                 if (error) {
4947                         dsl_dataset_rele(ds, FTAG);
4948                         return (error);
4949                 }
4950         }

4952         error = dsl_dataset_user_hold_for_send(ds, zc->zc_string,
4953             zc->zc_temphold);
4954         if (minor != 0) {
4955                 if (error == 0) {
4956                         dsl_register_onexit_hold_cleanup(ds, zc->zc_string,
4957                             minor);
4958                 }
4959                 zfs_onexit_fd_rele(zc->zc_cleanup_fd);
4960         }
4961         dsl_dataset_rele(ds, FTAG);

4963         return (error);
4964 }

4966 /*
4967  * inputs:
4968  * zc_name      name of dataset from which we're releasing a user hold
4969  * zc_value     short name of snap
4970  * zc_string    user-supplied tag for this hold
4971  * zc_cookie    recursive flag
4972  *
4973  * outputs:     none
4974  */
4975 static int
4976 zfs_ioc_release(zfs_cmd_t *zc)
4977 {
4978         boolean_t recursive = zc->zc_cookie;

4980         if (snapshot_namecheck(zc->zc_value, NULL, NULL) != 0)
4981                 return (EINVAL);

4983         return (dsl_dataset_user_release(zc->zc_name, zc->zc_value,
4984             zc->zc_string, recursive));
4985 }

4987 /*
4988  * inputs:
4989  * zc_name              name of filesystem
4990  *
4991  * outputs:
4992  * zc_nvlist_src{_size} nvlist of snapshot holds
4993  */
4994 static int
4995 zfs_ioc_get_holds(zfs_cmd_t *zc)
4996 {
4997         nvlist_t *nvp;
4998         int error;

5000         if ((error = dsl_dataset_get_holds(zc->zc_name, &nvp)) == 0) {
5001                 error = put_nvlist(zc, nvp);
5002                 nvlist_free(nvp);
5003         }

5005         return (error);
5006 }

5008 /*
```

```
5009  * inputs:
5010  * zc_name              name of new filesystem or snapshot
5011  * zc_value             full name of old snapshot
5012  *
5013  * outputs:
5014  * zc_cookie            space in bytes
5015  * zc_objset_type       compressed space in bytes
5016  * zc_perm_action       uncompressed space in bytes
5017  */
5018 static int
5019 zfs_ioc_space_written(zfs_cmd_t *zc)
5020 {
5021         int error;
5022         dsl_dataset_t *new, *old;

5024         error = dsl_dataset_hold(zc->zc_name, FTAG, &new);
5025         if (error != 0)
5026                 return (error);
5027         error = dsl_dataset_hold(zc->zc_value, FTAG, &old);
5028         if (error != 0) {
5029                 dsl_dataset_rele(new, FTAG);
5030                 return (error);
5031         }

5033         error = dsl_dataset_space_written(old, new, &zc->zc_cookie,
5034             &zc->zc_objset_type, &zc->zc_perm_action);
5035         dsl_dataset_rele(old, FTAG);
5036         dsl_dataset_rele(new, FTAG);
5037         return (error);
5038 }
5039 /*
5040  * innvl: {
5041  *     "firstsnap" -> snapshot name
5042  * }
5043  *
5044  * outnvl: {
5045  *     "used" -> space in bytes
5046  *     "compressed" -> compressed space in bytes
5047  *     "uncompressed" -> uncompressed space in bytes
5048  * }
5049  */
5050 static int
5051 zfs_ioc_space_snaps(const char *lastsnap, nvlist_t *innvl, nvlist_t *outnvl)
5052 {
5053         int error;
5054         dsl_dataset_t *new, *old;
5055         char *firstsnap;
5056         uint64_t used, comp, uncomp;

5058         if (nvlist_lookup_string(innvl, "firstsnap", &firstsnap) != 0)
5059                 return (EINVAL);

5061         error = dsl_dataset_hold(lastsnap, FTAG, &new);
5062         if (error != 0)
5063                 return (error);
5064         error = dsl_dataset_hold(firstsnap, FTAG, &old);
5065         if (error != 0) {
5066                 dsl_dataset_rele(new, FTAG);
5067                 return (error);
5068         }

5070         error = dsl_dataset_space_wouldfree(old, new, &used, &comp, &uncomp);
5071         dsl_dataset_rele(old, FTAG);
5072         dsl_dataset_rele(new, FTAG);
5073         fnvlist_add_uint64(outnvl, "used", used);
5074         fnvlist_add_uint64(outnvl, "compressed", comp);
```



```
5075         fnvlist_add_uint64(outnvl, "uncompressed", uncomp);
5076         return (error);
5077 }

5079 /*
5080  * innvl: {
5081  *     "fd" -> file descriptor to write stream to (int32)
5082  *     (optional) "fromsnap" -> full snap name to send an incremental from
5083  * }
5084  *
5085  * outnvl is unused
5086  */
5087 /* ARGSUSED */
5088 static int
5089 zfs_ioc_send_new(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5090 {
5091         objset_t *fromsnap = NULL;
5092         objset_t *tosnap;
5093         int error;
5094         offset_t off;
5095         char *fromname;
5096         int fd;

5098         error = nvlist_lookup_int32(innvl, "fd", &fd);
5099         if (error != 0)
5100                 return (EINVAL);

5102         error = dmu_objset_hold(snapname, FTAG, &tosnap);
5103         if (error)
5104                 return (error);

5106         error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
5107         if (error == 0) {
5108                 error = dmu_objset_hold(fromname, FTAG, &fromsnap);
5109                 if (error) {
5110                         dmu_objset_rele(tosnap, FTAG);
5111                         return (error);
5112                 }
5113         }

5115         file_t *fp = getf(fd);
5116         if (fp == NULL) {
5117                 dmu_objset_rele(tosnap, FTAG);
5118                 if (fromsnap != NULL)
5119                         dmu_objset_rele(fromsnap, FTAG);
5120                 return (EBADF);
5121         }

5123         off = fp->f_offset;
5124         error = dmu_send(tosnap, fromsnap, fd, fp->f_vnode, &off);

5126         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
5127                 fp->f_offset = off;
5128         releasef(fd);
5129         if (fromsnap != NULL)
5130                 dmu_objset_rele(fromsnap, FTAG);
5131         dmu_objset_rele(tosnap, FTAG);
5132         return (error);
5133 }

5135 /*
5136  * inputs:
5137  * zc_name      name of snapshot to send
5138  * zc_cookie    file descriptor to send stream to
5139  * zc_obj       fromorigin flag (mutually exclusive with zc_fromobj)
5140  * zc_sendobj   objsetid of snapshot to send
```

```
 * zc_fromobj   objsetid of incremental fromsnap (may be zero)
 * zc_guid      if set, estimate size of stream only.  zc_cookie is ignored.
 *              output size in zc_objset_type.
 *
 * outputs: none
 */
static int
zfs_ioc_fits_send(zfs_cmd_t *zc)
{
        objset_t *fromsnap = NULL;
        objset_t *tosnap;
        int error;
        offset_t off;
        dsl_dataset_t *ds;
        dsl_dataset_t *dsfrom = NULL;
        spa_t *spa;
        file_t *fp;
        dsl_pool_t *dp;
        boolean_t estimate = (zc->zc_guid != 0);

        error = spa_open(zc->zc_name, &spa, FTAG);
        if (error)
                return (error);

        dp = spa_get_dsl(spa);
        rw_enter(&dp->dp_config_rwlock, RW_READER);
        error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
        rw_exit(&dp->dp_config_rwlock);
        spa_close(spa, FTAG);
        if (error)
                return (error);

        error = dmu_objset_from_ds(ds, &tosnap);
        if (error) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        if (zc->zc_fromobj != 0) {
                rw_enter(&dp->dp_config_rwlock, RW_READER);
                error = dsl_dataset_hold_obj(dp, zc->zc_fromobj, FTAG, &dsfrom);
                rw_exit(&dp->dp_config_rwlock);
                if (error) {
                        dsl_dataset_rele(ds, FTAG);
                        return (error);
                }
                error = dmu_objset_from_ds(dsfrom, &fromsnap);
                if (error) {
                        dsl_dataset_rele(dsfrom, FTAG);
                        dsl_dataset_rele(ds, FTAG);
                        return (error);
                }
        }

        if (zc->zc_obj) {
                dsl_pool_t *dp = ds->ds_dir->dd_pool;

                if (fromsnap != NULL) {
                        dsl_dataset_rele(dsfrom, FTAG);
                        dsl_dataset_rele(ds, FTAG);
                        return (EINVAL);
                }

                if (dsl_dir_is_clone(ds->ds_dir)) {
                        rw_enter(&dp->dp_config_rwlock, RW_READER);
                        error = dsl_dataset_hold_obj(dp,
```



```
                            ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &dsfrom);
                        rw_exit(&dp->dp_config_rwlock);
                        if (error) {
                                dsl_dataset_rele(ds, FTAG);
                                return (error);
                        }
                        error = dmu_objset_from_ds(dsfrom, &fromsnap);
                        if (error) {
                                dsl_dataset_rele(dsfrom, FTAG);
                                dsl_dataset_rele(ds, FTAG);
                                return (error);
                        }
                }
        }

        fp = getf(zc->zc_cookie);
        if (fp == NULL) {
                dsl_dataset_rele(ds, FTAG);
                if (dsfrom)
                        dsl_dataset_rele(dsfrom, FTAG);
                return (EBADF);
        }

        off = fp->f_offset;
        error = fits_send(tosnap, fromsnap, zc->zc_cookie, fp->f_vnode, &off);

        if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
                fp->f_offset = off;
        releasef(zc->zc_cookie);

        if (dsfrom)
                dsl_dataset_rele(dsfrom, FTAG);
        dsl_dataset_rele(ds, FTAG);
        return (error);
}

/*
#endif /* ! codereview */
 * Determine approximately how large a zfs send stream will be -- the number
 * of bytes that will be written to the fd supplied to zfs_ioc_send_new().
 *
 * innvl: {
 *     (optional) "fromsnap" -> full snap name to send an incremental from
 * }
 *
 * outnvl: {
 *     "space" -> bytes of space (uint64)
 * }
 */
static int
zfs_ioc_send_space(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
{
        objset_t *fromsnap = NULL;
        objset_t *tosnap;
        int error;
        char *fromname;
        uint64_t space;

        error = dmu_objset_hold(snapname, FTAG, &tosnap);
        if (error)
                return (error);

        error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
        if (error == 0) {
                error = dmu_objset_hold(fromname, FTAG, &fromsnap);
                if (error) {
```

```
5273                        dmu_objset_rele(tosnap, FTAG);
5274                        return (error);
5275                }
5276        }

5278        error = dmu_send_estimate(tosnap, fromsnap, &space);
5279        fnvlist_add_uint64(outnvl, "space", space);

5281        if (fromsnap != NULL)
5282                dmu_objset_rele(fromsnap, FTAG);
5283        dmu_objset_rele(tosnap, FTAG);
5284        return (error);
5285 }


5288 static zfs_ioc_vec_t zfs_ioc_vec[ZFS_IOC_LAST - ZFS_IOC_FIRST];

5290 static void
5291 zfs_ioctl_register_legacy(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5292     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5293     boolean_t log_history, zfs_ioc_poolcheck_t pool_check)
5294 {
5295        zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];

5297        ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5298        ASSERT3U(ioc, <, ZFS_IOC_LAST);
5299        ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5300        ASSERT3P(vec->zvec_func, ==, NULL);

5302        vec->zvec_legacy_func = func;
5303        vec->zvec_secpolicy = secpolicy;
5304        vec->zvec_namecheck = namecheck;
5305        vec->zvec_allow_log = log_history;
5306        vec->zvec_pool_check = pool_check;
5307 }

5309 /*
5310  * See the block comment at the beginning of this file for details on
5311  * each argument to this function.
5312  */
5313 static void
5314 zfs_ioctl_register(const char *name, zfs_ioc_t ioc, zfs_ioc_func_t *func,
5315     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5316     zfs_ioc_poolcheck_t pool_check, boolean_t smush_outnvlist,
5317     boolean_t allow_log)
5318 {
5319        zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];

5321        ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5322        ASSERT3U(ioc, <, ZFS_IOC_LAST);
5323        ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5324        ASSERT3P(vec->zvec_func, ==, NULL);

5326        /* if we are logging, the name must be valid */
5327        ASSERT(!allow_log || namecheck != NO_NAME);

5329        vec->zvec_name = name;
5330        vec->zvec_func = func;
5331        vec->zvec_secpolicy = secpolicy;
5332        vec->zvec_namecheck = namecheck;
5333        vec->zvec_pool_check = pool_check;
5334        vec->zvec_smush_outnvlist = smush_outnvlist;
5335        vec->zvec_allow_log = allow_log;
5336 }

5338 static void
```



```
5339 zfs_ioctl_register_pool(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5340     zfs_secpolicy_func_t *secpolicy, boolean_t log_history,
5341     zfs_ioc_poolcheck_t pool_check)
5342 {
5343        zfs_ioctl_register_legacy(ioc, func, secpolicy,
5344            POOL_NAME, log_history, pool_check);
5345 }

5347 static void
5348 zfs_ioctl_register_dataset_nolog(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5349     zfs_secpolicy_func_t *secpolicy, zfs_ioc_poolcheck_t pool_check)
5350 {
5351        zfs_ioctl_register_legacy(ioc, func, secpolicy,
5352            DATASET_NAME, B_FALSE, pool_check);
5353 }

5355 static void
5356 zfs_ioctl_register_pool_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5357 {
5358        zfs_ioctl_register_legacy(ioc, func, zfs_secpolicy_config,
5359            POOL_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5360 }

5362 static void
5363 zfs_ioctl_register_pool_meta(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5364     zfs_secpolicy_func_t *secpolicy)
5365 {
5366        zfs_ioctl_register_legacy(ioc, func, secpolicy,
5367            NO_NAME, B_FALSE, POOL_CHECK_NONE);
5368 }

5370 static void
5371 zfs_ioctl_register_dataset_read_secpolicy(zfs_ioc_t ioc,
5372     zfs_ioc_legacy_func_t *func, zfs_secpolicy_func_t *secpolicy)
5373 {
5374        zfs_ioctl_register_legacy(ioc, func, secpolicy,
5375            DATASET_NAME, B_FALSE, POOL_CHECK_SUSPENDED);
5376 }

5378 static void
5379 zfs_ioctl_register_dataset_read(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5380 {
5381        zfs_ioctl_register_dataset_read_secpolicy(ioc, func,
5382            zfs_secpolicy_read);
5383 }

5385 static void
5386 zfs_ioctl_register_dataset_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5387        zfs_secpolicy_func_t *secpolicy)
5388 {
5389        zfs_ioctl_register_legacy(ioc, func, secpolicy,
5390            DATASET_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5391 }

5393 static void
5394 zfs_ioctl_init(void)
5395 {
5396        zfs_ioctl_register("snapshot", ZFS_IOC_SNAPSHOT,
5397            zfs_ioc_snapshot, zfs_secpolicy_snapshot, POOL_NAME,
5398            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5400        zfs_ioctl_register("log_history", ZFS_IOC_LOG_HISTORY,
5401            zfs_ioc_log_history, zfs_secpolicy_log_history, NO_NAME,
5402            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_FALSE);

5404        zfs_ioctl_register("space_snaps", ZFS_IOC_SPACE_SNAPS,
```

```
5405             zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME,
5406             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5408         zfs_ioctl_register("send", ZFS_IOC_SEND_NEW,
5409             zfs_ioc_send_new, zfs_secpolicy_send_new, DATASET_NAME,
5410             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5412         zfs_ioctl_register("send_space", ZFS_IOC_SEND_SPACE,
5413             zfs_ioc_send_space, zfs_secpolicy_read, DATASET_NAME,
5414             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5416         zfs_ioctl_register("create", ZFS_IOC_CREATE,
5417             zfs_ioc_create, zfs_secpolicy_create_clone, DATASET_NAME,
5418             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5420         zfs_ioctl_register("clone", ZFS_IOC_CLONE,
5421             zfs_ioc_clone, zfs_secpolicy_create_clone, DATASET_NAME,
5422             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5424         zfs_ioctl_register("destroy_snaps", ZFS_IOC_DESTROY_SNAPS,
5425             zfs_ioc_destroy_snaps, zfs_secpolicy_destroy_snaps, POOL_NAME,
5426             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5428         /* IOCTLS that use the legacy function signature */

5430         zfs_ioctl_register_legacy(ZFS_IOC_POOL_FREEZE, zfs_ioc_pool_freeze,
5431             zfs_secpolicy_config, NO_NAME, B_FALSE, POOL_CHECK_READONLY);

5433         zfs_ioctl_register_pool(ZFS_IOC_POOL_CREATE, zfs_ioc_pool_create,
5434             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5435         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SCAN,
5436             zfs_ioc_pool_scan);
5437         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_UPGRADE,
5438             zfs_ioc_pool_upgrade);
5439         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ADD,
5440             zfs_ioc_vdev_add);
5441         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_REMOVE,
5442             zfs_ioc_vdev_remove);
5443         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SET_STATE,
5444             zfs_ioc_vdev_set_state);
5445         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ATTACH,
5446             zfs_ioc_vdev_attach);
5447         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_DETACH,
5448             zfs_ioc_vdev_detach);
5449         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETPATH,
5450             zfs_ioc_vdev_setpath);
5451         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETFRU,
5452             zfs_ioc_vdev_setfru);
5453         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SET_PROPS,
5454             zfs_ioc_pool_set_props);
5455         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SPLIT,
5456             zfs_ioc_vdev_split);
5457         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_REGUID,
5458             zfs_ioc_pool_reguid);

5460         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_CONFIGS,
5461             zfs_ioc_pool_configs, zfs_secpolicy_none);
5462         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_TRYIMPORT,
5463             zfs_ioc_pool_tryimport, zfs_secpolicy_config);
5464         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_FAULT,
5465             zfs_ioc_inject_fault, zfs_secpolicy_inject);
5466         zfs_ioctl_register_pool_meta(ZFS_IOC_CLEAR_FAULT,
5467             zfs_ioc_clear_fault, zfs_secpolicy_inject);
5468         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_LIST_NEXT,
5469             zfs_ioc_inject_list_next, zfs_secpolicy_inject);
```



```
5471         /*
5472          * pool destroy, and export don't log the history as part of
5473          * zfsdev_ioctl, but rather zfs_ioc_pool_export
5474          * does the logging of those commands.
5475          */
5476         zfs_ioctl_register_pool(ZFS_IOC_POOL_DESTROY, zfs_ioc_pool_destroy,
5477             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
5478         zfs_ioctl_register_pool(ZFS_IOC_POOL_EXPORT, zfs_ioc_pool_export,
5479             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);

5481         zfs_ioctl_register_pool(ZFS_IOC_POOL_STATS, zfs_ioc_pool_stats,
5482             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
5483         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_PROPS, zfs_ioc_pool_get_props,
5484             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);

5486         zfs_ioctl_register_pool(ZFS_IOC_ERROR_LOG, zfs_ioc_error_log,
5487             zfs_secpolicy_inject, B_FALSE, POOL_CHECK_SUSPENDED);
5488         zfs_ioctl_register_pool(ZFS_IOC_DSOBJ_TO_DSNAME,
5489             zfs_ioc_dsobj_to_dsname,
5490             zfs_secpolicy_diff, B_FALSE, POOL_CHECK_SUSPENDED);
5491         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_HISTORY,
5492             zfs_ioc_pool_get_history,
5493             zfs_secpolicy_config, B_FALSE, POOL_CHECK_SUSPENDED);

5495         zfs_ioctl_register_pool(ZFS_IOC_POOL_IMPORT, zfs_ioc_pool_import,
5496             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);

5498         zfs_ioctl_register_pool(ZFS_IOC_CLEAR, zfs_ioc_clear,
5499             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
5500         zfs_ioctl_register_pool(ZFS_IOC_POOL_REOPEN, zfs_ioc_pool_reopen,
5501             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);

5503         zfs_ioctl_register_dataset_read(ZFS_IOC_SPACE_WRITTEN,
5504             zfs_ioc_space_written);
5505         zfs_ioctl_register_dataset_read(ZFS_IOC_GET_HOLDS,
5506             zfs_ioc_get_holds);
5507         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_RECVD_PROPS,
5508             zfs_ioc_objset_recvd_props);
5509         zfs_ioctl_register_dataset_read(ZFS_IOC_NEXT_OBJ,
5510             zfs_ioc_next_obj);
5511         zfs_ioctl_register_dataset_read(ZFS_IOC_GET_FSACL,
5512             zfs_ioc_get_fsacl);
5513         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_STATS,
5514             zfs_ioc_objset_stats);
5515         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_ZPLPROPS,
5516             zfs_ioc_objset_zplprops);
5517         zfs_ioctl_register_dataset_read(ZFS_IOC_DATASET_LIST_NEXT,
5518             zfs_ioc_dataset_list_next);
5519         zfs_ioctl_register_dataset_read(ZFS_IOC_SNAPSHOT_LIST_NEXT,
5520             zfs_ioc_snapshot_list_next);
5521         zfs_ioctl_register_dataset_read(ZFS_IOC_SEND_PROGRESS,
5522             zfs_ioc_send_progress);

5524         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_DIFF,
5525             zfs_ioc_diff, zfs_secpolicy_diff);
5526         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_STATS,
5527             zfs_ioc_obj_to_stats, zfs_secpolicy_diff);
5528         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_PATH,
5529             zfs_ioc_obj_to_path, zfs_secpolicy_diff);
5530         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_ONE,
5531             zfs_ioc_userspace_one, zfs_secpolicy_userspace_one);
5532         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_MANY,
5533             zfs_ioc_userspace_many, zfs_secpolicy_userspace_many);
5534         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_SEND,
5535             zfs_ioc_send, zfs_secpolicy_send);
5536         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_FITS_SEND,
```

```
5537             zfs_ioc_fits_send, zfs_secpolicy_send);
5538 #endif /* ! codereview */
5539
5540         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_PROP, zfs_ioc_set_prop,
5541             zfs_secpolicy_none);
5542         zfs_ioctl_register_dataset_modify(ZFS_IOC_DESTROY, zfs_ioc_destroy,
5543             zfs_secpolicy_destroy);
5544         zfs_ioctl_register_dataset_modify(ZFS_IOC_ROLLBACK, zfs_ioc_rollback,
5545             zfs_secpolicy_rollback);
5546         zfs_ioctl_register_dataset_modify(ZFS_IOC_RENAME, zfs_ioc_rename,
5547             zfs_secpolicy_rename);
5548         zfs_ioctl_register_dataset_modify(ZFS_IOC_RECV, zfs_ioc_recv,
5549             zfs_secpolicy_recv);
5550         zfs_ioctl_register_dataset_modify(ZFS_IOC_PROMOTE, zfs_ioc_promote,
5551             zfs_secpolicy_promote);
5552         zfs_ioctl_register_dataset_modify(ZFS_IOC_HOLD, zfs_ioc_hold,
5553             zfs_secpolicy_hold);
5554         zfs_ioctl_register_dataset_modify(ZFS_IOC_RELEASE, zfs_ioc_release,
5555             zfs_secpolicy_release);
5556         zfs_ioctl_register_dataset_modify(ZFS_IOC_INHERIT_PROP,
5557             zfs_ioc_inherit_prop, zfs_secpolicy_inherit_prop);
5558         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_FSACL, zfs_ioc_set_fsacl,
5559             zfs_secpolicy_set_fsacl);
5560
5561         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SHARE, zfs_ioc_share,
5562             zfs_secpolicy_share, POOL_CHECK_NONE);
5563         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SMB_ACL, zfs_ioc_smb_acl,
5564             zfs_secpolicy_smb_acl, POOL_CHECK_NONE);
5565         zfs_ioctl_register_dataset_nolog(ZFS_IOC_USERSPACE_UPGRADE,
5566             zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
5567             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5568         zfs_ioctl_register_dataset_nolog(ZFS_IOC_TMP_SNAPSHOT,
5569             zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot,
5570             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5571 }
5572
5573 int
5574 pool_status_check(const char *name, zfs_ioc_namecheck_t type,
5575     zfs_ioc_poolcheck_t check)
5576 {
5577         spa_t *spa;
5578         int error;
5579
5580         ASSERT(type == POOL_NAME || type == DATASET_NAME);
5581
5582         if (check & POOL_CHECK_NONE)
5583                 return (0);
5584
5585         error = spa_open(name, &spa, FTAG);
5586         if (error == 0) {
5587                 if ((check & POOL_CHECK_SUSPENDED) && spa_suspended(spa))
5588                         error = EAGAIN;
5589                 else if ((check & POOL_CHECK_READONLY) && !spa_writeable(spa))
5590                         error = EROFS;
5591                 spa_close(spa, FTAG);
5592         }
5593         return (error);
5594 }
5595
5596 /*
5597  * Find a free minor number.
5598  */
5599 minor_t
5600 zfsdev_minor_alloc(void)
5601 {
5602         static minor_t last_minor;
```



```
5603         minor_t m;
5604
5605         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
5606
5607         for (m = last_minor + 1; m != last_minor; m++) {
5608                 if (m > ZFSDEV_MAX_MINOR)
5609                         m = 1;
5610                 if (ddi_get_soft_state(zfsdev_state, m) == NULL) {
5611                         last_minor = m;
5612                         return (m);
5613                 }
5614         }
5615
5616         return (0);
5617 }
5618
5619 static int
5620 zfs_ctldev_init(dev_t *devp)
5621 {
5622         minor_t minor;
5623         zfs_soft_state_t *zs;
5624
5625         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
5626         ASSERT(getminor(*devp) == 0);
5627
5628         minor = zfsdev_minor_alloc();
5629         if (minor == 0)
5630                 return (ENXIO);
5631
5632         if (ddi_soft_state_zalloc(zfsdev_state, minor) != DDI_SUCCESS)
5633                 return (EAGAIN);
5634
5635         *devp = makedevice(getemajor(*devp), minor);
5636
5637         zs = ddi_get_soft_state(zfsdev_state, minor);
5638         zs->zss_type = ZSST_CTLDEV;
5639         zfs_onexit_init((zfs_onexit_t **)&zs->zss_data);
5640
5641         return (0);
5642 }
5643
5644 static void
5645 zfs_ctldev_destroy(zfs_onexit_t *zo, minor_t minor)
5646 {
5647         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
5648
5649         zfs_onexit_destroy(zo);
5650         ddi_soft_state_free(zfsdev_state, minor);
5651 }
5652
5653 void *
5654 zfsdev_get_soft_state(minor_t minor, enum zfs_soft_state_type which)
5655 {
5656         zfs_soft_state_t *zp;
5657
5658         zp = ddi_get_soft_state(zfsdev_state, minor);
5659         if (zp == NULL || zp->zss_type != which)
5660                 return (NULL);
5661
5662         return (zp->zss_data);
5663 }
5664
5665 static int
5666 zfsdev_open(dev_t *devp, int flag, int otyp, cred_t *cr)
5667 {
5668         int error = 0;
```

```
5670         if (getminor(*devp) != 0)
5671                 return (zvol_open(devp, flag, otyp, cr));
5672
5673         /* This is the control device. Allocate a new minor if requested. */
5674         if (flag & FEXCL) {
5675                 mutex_enter(&zfsdev_state_lock);
5676                 error = zfs_ctldev_init(devp);
5677                 mutex_exit(&zfsdev_state_lock);
5678         }
5679
5680         return (error);
5681 }
5682
5683 static int
5684 zfsdev_close(dev_t dev, int flag, int otyp, cred_t *cr)
5685 {
5686         zfs_onexit_t *zo;
5687         minor_t minor = getminor(dev);
5688
5689         if (minor == 0)
5690                 return (0);
5691
5692         mutex_enter(&zfsdev_state_lock);
5693         zo = zfsdev_get_soft_state(minor, ZSST_CTLDEV);
5694         if (zo == NULL) {
5695                 mutex_exit(&zfsdev_state_lock);
5696                 return (zvol_close(dev, flag, otyp, cr));
5697         }
5698         zfs_ctldev_destroy(zo, minor);
5699         mutex_exit(&zfsdev_state_lock);
5700
5701         return (0);
5702 }
5703
5704 static int
5705 zfsdev_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp)
5706 {
5707         zfs_cmd_t *zc;
5708         uint_t vecnum;
5709         int error, rc, len;
5710         minor_t minor = getminor(dev);
5711         const zfs_ioc_vec_t *vec;
5712         char *saved_poolname = NULL;
5713         nvlist_t *innvl = NULL;
5714
5715         if (minor != 0 &&
5716             zfsdev_get_soft_state(minor, ZSST_CTLDEV) == NULL)
5717                 return (zvol_ioctl(dev, cmd, arg, flag, cr, rvalp));
5718
5719         vecnum = cmd - ZFS_IOC_FIRST;
5720         ASSERT3U(getmajor(dev), ==, ddi_driver_major(zfs_dip));
5721
5722         if (vecnum >= sizeof (zfs_ioc_vec) / sizeof (zfs_ioc_vec[0]))
5723                 return (EINVAL);
5724         vec = &zfs_ioc_vec[vecnum];
5725
5726         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);
5727
5728         error = ddi_copyin((void *)arg, zc, sizeof (zfs_cmd_t), flag);
5729         if (error != 0) {
5730                 error = EFAULT;
5731                 goto out;
5732         }
5733
5734         zc->zc_iflags = flag & FKIOCTL;
```



```
5735         if (zc->zc_nvlist_src_size != 0) {
5736                 error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
5737                     zc->zc_iflags, &innvl);
5738                 if (error != 0)
5739                         goto out;
5740         }
5741
5742         /*
5743          * Ensure that all pool/dataset names are valid before we pass down to
5744          * the lower layers.
5745          */
5746         zc->zc_name[sizeof (zc->zc_name) - 1] = '\0';
5747         switch (vec->zvec_namecheck) {
5748         case POOL_NAME:
5749                 if (pool_namecheck(zc->zc_name, NULL, NULL) != 0)
5750                         error = EINVAL;
5751                 else
5752                         error = pool_status_check(zc->zc_name,
5753                             vec->zvec_namecheck, vec->zvec_pool_check);
5754                 break;
5755
5756         case DATASET_NAME:
5757                 if (dataset_namecheck(zc->zc_name, NULL, NULL) != 0)
5758                         error = EINVAL;
5759                 else
5760                         error = pool_status_check(zc->zc_name,
5761                             vec->zvec_namecheck, vec->zvec_pool_check);
5762                 break;
5763
5764         case NO_NAME:
5765                 break;
5766         }
5767
5768
5769         if (error == 0 && !(flag & FKIOCTL))
5770                 error = vec->zvec_secpolicy(zc, innvl, cr);
5771
5772         if (error != 0)
5773                 goto out;
5774
5775         /* legacy ioctls can modify zc_name */
5776         len = strcspn(zc->zc_name, "/@") + 1;
5777         saved_poolname = kmem_alloc(len, KM_SLEEP);
5778         (void) strlcpy(saved_poolname, zc->zc_name, len);
5779
5780         if (vec->zvec_func != NULL) {
5781                 nvlist_t *outnvl;
5782                 int puterror = 0;
5783                 spa_t *spa;
5784                 nvlist_t *lognv = NULL;
5785
5786                 ASSERT(vec->zvec_legacy_func == NULL);
5787
5788                 /*
5789                  * Add the innvl to the lognv before calling the func,
5790                  * in case the func changes the innvl.
5791                  */
5792                 if (vec->zvec_allow_log) {
5793                         lognv = fnvlist_alloc();
5794                         fnvlist_add_string(lognv, ZPOOL_HIST_IOCTL,
5795                             vec->zvec_name);
5796                         if (!nvlist_empty(innvl)) {
5797                                 fnvlist_add_nvlist(lognv, ZPOOL_HIST_INPUT_NVL,
5798                                     innvl);
5799                         }
5800                 }
```

```
5802                 outnvl = fnvlist_alloc();
5803                 error = vec->zvec_func(zc->zc_name, innvl, outnvl);

5805                 if (error == 0 && vec->zvec_allow_log &&
5806                     spa_open(zc->zc_name, &spa, FTAG) == 0) {
5807                         if (!nvlist_empty(outnvl)) {
5808                                 fnvlist_add_nvlist(lognv, ZPOOL_HIST_OUTPUT_NVL,
5809                                     outnvl);
5810                         }
5811                         (void) spa_history_log_nvl(spa, lognv);
5812                         spa_close(spa, FTAG);
5813                 }
5814                 fnvlist_free(lognv);

5816                 if (!nvlist_empty(outnvl) || zc->zc_nvlist_dst_size != 0) {
5817                         int smusherror = 0;
5818                         if (vec->zvec_smush_outnvlist) {
5819                                 smusherror = nvlist_smush(outnvl,
5820                                     zc->zc_nvlist_dst_size);
5821                         }
5822                         if (smusherror == 0)
5823                                 puterror = put_nvlist(zc, outnvl);
5824                 }

5826                 if (puterror != 0)
5827                         error = puterror;

5829                 nvlist_free(outnvl);
5830         } else {
5831                 error = vec->zvec_legacy_func(zc);
5832         }

5834 out:
5835         nvlist_free(innvl);
5836         rc = ddi_copyout(zc, (void *)arg, sizeof (zfs_cmd_t), flag);
5837         if (error == 0 && rc != 0)
5838                 error = EFAULT;
5839         if (error == 0 && vec->zvec_allow_log) {
5840                 char *s = tsd_get(zfs_allow_log_key);
5841                 if (s != NULL)
5842                         strfree(s);
5843                 (void) tsd_set(zfs_allow_log_key, saved_poolname);
5844         } else {
5845                 if (saved_poolname != NULL)
5846                         strfree(saved_poolname);
5847         }

5849         kmem_free(zc, sizeof (zfs_cmd_t));
5850         return (error);
5851 }

5853 static int
5854 zfs_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
5855 {
5856         if (cmd != DDI_ATTACH)
5857                 return (DDI_FAILURE);

5859         if (ddi_create_minor_node(dip, "zfs", S_IFCHR, 0,
5860             DDI_PSEUDO, 0) == DDI_FAILURE)
5861                 return (DDI_FAILURE);

5863         zfs_dip = dip;

5865         ddi_report_dev(dip);
```



```
5867         return (DDI_SUCCESS);
5868 }

5870 static int
5871 zfs_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
5872 {
5873         if (spa_busy() || zfs_busy() || zvol_busy())
5874                 return (DDI_FAILURE);

5876         if (cmd != DDI_DETACH)
5877                 return (DDI_FAILURE);

5879         zfs_dip = NULL;

5881         ddi_prop_remove_all(dip);
5882         ddi_remove_minor_node(dip, NULL);

5884         return (DDI_SUCCESS);
5885 }

5887 /*ARGSUSED*/
5888 static int
5889 zfs_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
5890 {
5891         switch (infocmd) {
5892         case DDI_INFO_DEVT2DEVINFO:
5893                 *result = zfs_dip;
5894                 return (DDI_SUCCESS);

5896         case DDI_INFO_DEVT2INSTANCE:
5897                 *result = (void *)0;
5898                 return (DDI_SUCCESS);
5899         }

5901         return (DDI_FAILURE);
5902 }

5904 /*
5905  * OK, so this is a little weird.
5906  *
5907  * /dev/zfs is the control node, i.e. minor 0.
5908  * /dev/zvol/[r]dsk/pool/dataset are the zvols, minor > 0.
5909  *
5910  * /dev/zfs has basically nothing to do except serve up ioctls,
5911  * so most of the standard driver entry points are in zvol.c.
5912  */
5913 static struct cb_ops zfs_cb_ops = {
5914         zfsdev_open,    /* open */
5915         zfsdev_close,   /* close */
5916         zvol_strategy,  /* strategy */
5917         nodev,          /* print */
5918         zvol_dump,      /* dump */
5919         zvol_read,      /* read */
5920         zvol_write,     /* write */
5921         zfsdev_ioctl,   /* ioctl */
5922         nodev,          /* devmap */
5923         nodev,          /* mmap */
5924         nodev,          /* segmap */
5925         nochpoll,       /* poll */
5926         ddi_prop_op,    /* prop_op */
5927         NULL,           /* streamtab */
5928         D_NEW | D_MP | D_64BIT,         /* Driver compatibility flag */
5929         CB_REV,         /* version */
5930         nodev,          /* async read */
5931         nodev,          /* async write */
5932 };
```

```
5934 static struct dev_ops zfs_dev_ops = {
5935         DEVO_REV,       /* version */
5936         0,              /* refcnt */
5937         zfs_info,       /* info */
5938         nulldev,        /* identify */
5939         nulldev,        /* probe */
5940         zfs_attach,     /* attach */
5941         zfs_detach,     /* detach */
5942         nodev,          /* reset */
5943         &zfs_cb_ops,    /* driver operations */
5944         NULL,           /* no bus operations */
5945         NULL,           /* power */
5946         ddi_quiesce_not_needed, /* quiesce */
5947 };

5949 static struct modldrv zfs_modldrv = {
5950         &mod_driverops,
5951         "ZFS storage pool",
5952         &zfs_dev_ops
5953 };

5955 static struct modlinkage modlinkage = {
5956         MODREV_1,
5957         (void *)&zfs_modlfs,
5958         (void *)&zfs_modldrv,
5959         NULL
5960 };

5962 static void
5963 zfs_allow_log_destroy(void *arg)
5964 {
5965         char *poolname = arg;
5966         strfree(poolname);
5967 }

5969 int
5970 _init(void)
5971 {
5972         int error;

5974         spa_init(FREAD | FWRITE);
5975         zfs_init();
5976         zvol_init();
5977         zfs_ioctl_init();

5979         if ((error = mod_install(&modlinkage)) != 0) {
5980                 zvol_fini();
5981                 zfs_fini();
5982                 spa_fini();
5983                 return (error);
5984         }

5986         tsd_create(&zfs_fsyncer_key, NULL);
5987         tsd_create(&rrw_tsd_key, rrw_tsd_destroy);
5988         tsd_create(&zfs_allow_log_key, zfs_allow_log_destroy);

5990         error = ldi_ident_from_mod(&modlinkage, &zfs_li);
5991         ASSERT(error == 0);
5992         mutex_init(&zfs_share_lock, NULL, MUTEX_DEFAULT, NULL);

5994         return (0);
5995 }

5997 int
5998 _fini(void)
```



```
5999 {
6000         int error;

6002         if (spa_busy() || zfs_busy() || zvol_busy() || zio_injection_enabled)
6003                 return (EBUSY);

6005         if ((error = mod_remove(&modlinkage)) != 0)
6006                 return (error);

6008         zvol_fini();
6009         zfs_fini();
6010         spa_fini();
6011         if (zfs_nfsshare_inited)
6012                 (void) ddi_modclose(nfs_mod);
6013         if (zfs_smbshare_inited)
6014                 (void) ddi_modclose(smbsrv_mod);
6015         if (zfs_nfsshare_inited || zfs_smbshare_inited)
6016                 (void) ddi_modclose(sharefs_mod);

6018         tsd_destroy(&zfs_fsyncer_key);
6019         ldi_ident_release(zfs_li);
6020         zfs_li = NULL;
6021         mutex_destroy(&zfs_share_lock);

6023         return (error);
6024 }

6026 int
6027 _info(struct modinfo *modinfop)
6028 {
6029         return (mod_info(&modlinkage, modinfop));
6030 }
```

```
**********************************************************
   28824 Wed Oct 17 21:48:40 2012
new/usr/src/uts/common/sys/fs/zfs.h
FITS: generating send-streams in portable format
This commit adds the command 'zfs fits-send', analogous to zfs send. The
generated send stream is compatible with the stream generated with that
from 'btrfs send' and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

 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 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_POOL_REGUID,
 822         ZFS_IOC_POOL_REOPEN,
 823         ZFS_IOC_SEND_PROGRESS,
 824         ZFS_IOC_LOG_HISTORY,
 825         ZFS_IOC_SEND_NEW,
 826         ZFS_IOC_SEND_SPACE,
 827         ZFS_IOC_CLONE,
 828         ZFS_IOC_FITS_SEND,
 829 #endif /* ! codereview */
 830         ZFS_IOC_LAST
 831 } zfs_ioc_t;

 833 /*
 834  * Internal SPA load state.  Used by FMA diagnosis engine.
 835  */
 836 typedef enum {
 837         SPA_LOAD_NONE,          /* no load in progress  */
 838         SPA_LOAD_OPEN,          /* normal open          */
 839         SPA_LOAD_IMPORT,        /* import in progress   */
 840         SPA_LOAD_TRYIMPORT,     /* tryimport in progress */
 841         SPA_LOAD_RECOVER,       /* recovery requested   */
 842         SPA_LOAD_ERROR          /* load failed          */
 843 } spa_load_state_t;

 845 /*
 846  * Bookmark name values.
 847  */
 848 #define ZPOOL_ERR_LIST          "error list"
 849 #define ZPOOL_ERR_DATASET       "dataset"
 850 #define ZPOOL_ERR_OBJECT        "object"

 852 #define HIS_MAX_RECORD_LEN      (MAXPATHLEN + MAXPATHLEN + 1)

 854 /*
 855  * The following are names used in the nvlist describing
 856  * the pool's history log.
 857  */
 858 #define ZPOOL_HIST_RECORD       "history record"
 859 #define ZPOOL_HIST_TIME         "history time"
 860 #define ZPOOL_HIST_CMD          "history command"
 861 #define ZPOOL_HIST_WHO          "history who"
```

```
862 #define ZPOOL_HIST_ZONE         "history zone"
863 #define ZPOOL_HIST_HOST         "history hostname"
864 #define ZPOOL_HIST_TXG          "history txg"
865 #define ZPOOL_HIST_INT_EVENT    "history internal event"
866 #define ZPOOL_HIST_INT_STR      "history internal str"
867 #define ZPOOL_HIST_INT_NAME     "internal_name"
868 #define ZPOOL_HIST_IOCTL        "ioctl"
869 #define ZPOOL_HIST_INPUT_NVL    "in_nvl"
870 #define ZPOOL_HIST_OUTPUT_NVL   "out_nvl"
871 #define ZPOOL_HIST_DSNAME       "dsname"
872 #define ZPOOL_HIST_DSID         "dsid"

874 /*
875  * Flags for ZFS_IOC_VDEV_SET_STATE
876  */
877 #define ZFS_ONLINE_CHECKREMOVE  0x1
878 #define ZFS_ONLINE_UNSPARE      0x2
879 #define ZFS_ONLINE_FORCEFAULT   0x4
880 #define ZFS_ONLINE_EXPAND       0x8
881 #define ZFS_OFFLINE_TEMPORARY   0x1

883 /*
884  * Flags for ZFS_IOC_POOL_IMPORT
885  */
886 #define ZFS_IMPORT_NORMAL       0x0
887 #define ZFS_IMPORT_VERBATIM     0x1
888 #define ZFS_IMPORT_ANY_HOST     0x2
889 #define ZFS_IMPORT_MISSING_LOG  0x4
890 #define ZFS_IMPORT_ONLY         0x8

892 /*
893  * Sysevent payload members.  ZFS will generate the following sysevents with the
894  * given payloads:
895  *
896  *      ESC_ZFS_RESILVER_START
897  *      ESC_ZFS_RESILVER_END
898  *      ESC_ZFS_POOL_DESTROY
899  *      ESC_ZFS_POOL_REGUID
900  *
901  *              ZFS_EV_POOL_NAME        DATA_TYPE_STRING
902  *              ZFS_EV_POOL_GUID        DATA_TYPE_UINT64
903  *
904  *      ESC_ZFS_VDEV_REMOVE
905  *      ESC_ZFS_VDEV_CLEAR
906  *      ESC_ZFS_VDEV_CHECK
907  *
908  *              ZFS_EV_POOL_NAME        DATA_TYPE_STRING
909  *              ZFS_EV_POOL_GUID        DATA_TYPE_UINT64
910  *              ZFS_EV_VDEV_PATH        DATA_TYPE_STRING        (optional)
911  *              ZFS_EV_VDEV_GUID        DATA_TYPE_UINT64
912  */
913 #define ZFS_EV_POOL_NAME        "pool_name"
914 #define ZFS_EV_POOL_GUID        "pool_guid"
915 #define ZFS_EV_VDEV_PATH        "vdev_path"
916 #define ZFS_EV_VDEV_GUID        "vdev_guid"

918 #ifdef  __cplusplus
919 }
920 #endif

922 #endif  /* _SYS_FS_ZFS_H */
```