

```
new/usr/src/man/man1m/mount_smbfs.1m
```

1

```
*****  
12327 Sun Mar 4 06:09:08 2018  
new/usr/src/man/man1m/mount_smbfs.1m  
Convert mount_smbfs.1m to mdoc  
*****
```

```
1 .\" te  
2 .\" Copyright (c) 2008, Sun Microsystems, Inc. All Rights Reserved.  
3 .\" Portions Copyright 1994-2008 The FreeBSD Project. All rights reserved.  
4 .\" Redistribution and use in source and binary forms, with or without modification  
5 .\" the following disclaimer. 2. Redistributions in binary form must reproduce the  
6 .\" BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES  
7 .\" SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  
8 .\" OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF AD  
9 .\" Copyright 2012 Nexenta Systems, Inc. All rights reserved.  
10 .Dd March 4, 2018  
11 .Dt MOUNT_SMBFS 1M  
12 .Os  
13 .Sh NAME  
14 .Nm mount_smbfs ,  
15 .Nm umount_smbfs  
16 .Nd mount and unmount a shared resource from an SMB file server  
17 .Sh SYNOPSIS  
18 .Nm mount  
19 .Op Fl F Cm smbfs  
20 .Op Ar generic-options  
21 .Op Fl o Ar nameNs Oo =Ns Ar valueOc  
22 .Op Fl o  
23 .Ar resource  
24 .Nm mount  
25 .Op Fl F Cm smbfs  
26 .Op Ar generic-options  
27 .Op Fl o Ar nameNs Oo =Ns Ar valueOc  
28 .Op Fl o  
29 .Ar mount-point  
30 .Nm mount  
31 .Op Fl F Cm smbfs  
32 .Op Ar generic-options  
33 .Op Fl o Ar nameNs Oo =Ns Ar valueOc  
34 .Op Fl o  
35 .Ar resource mount-point  
36 .Nm umount  
37 .Op Fl F Cm smbfs  
38 .Op Ar generic-options  
39 .Ar mount-point  
40 .Sh DESCRIPTION  
41 .Th MOUNT_SMBFS 1M "Jan 2, 2012"  
42 .Sh NAME  
43 mount_smbfs, umount_smbfs \- mount and unmount a shared resource from a CIFS  
44 file server  
45 .Sh SYNOPSIS  
46 .LP  
47 .nf  
48 .FB/sbin/mount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] [\fB-o\fR \fIname\fR]  
49 .fi  
50 .LP  
51 .nf  
52 .FB/sbin/mount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] [\fB-o\fR \fIname\fR]  
53 .fi  
54 .LP  
55 .nf  
56 .FB/sbin/mount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] [\fB-o\fR \fIname\fR]  
57 .fb-O\fR] \fIresource\fR \fImount-point\fR  
58 .fi
```

```
new/usr/src/man/man1m/mount_smbfs.1m
```

2

```
31 .LP  
32 .nf  
33 .FB/sbin/umount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] \fImount-point\fR  
34 .fi  
35 .SH DESCRIPTION  
36 .sp  
37 .LP  
38 The \fBmount\fR utility attaches a named resource, \fIresource\fR, to the file  
39 system hierarchy at the path name location, \fImount-point\fR, which must  
40 already exist.  
41 .sp  
42 .LP  
43 The \fBmount\fR utility attaches a named resource, \fIresource\fR, to the file  
44 system hierarchy at the path name location, \fImount-point\fR, which must  
45 already exist.  
46 .sp  
47 .LP  
48 The  
49 .Nm mount  
50 utility attaches a named resource,  
51 .Ar resource ,  
52 to the file system hierarchy at the path name location,  
53 .Ar mount-point ,  
54 which must already exist.  
55 .Pp  
56 If  
57 .Ar mount-point  
58 has any contents prior to the  
59 .Nm mount  
60 operation, those contents remain hidden until the resource is unmounted.  
61 An authorized user with the  
62 .Dv SYS_MOUNT  
63 privilege can perform a  
64 .Nm mount  
65 operation.  
66 Also, a user can perform SMBFS mount operations on a directory the user owns.  
67 .Pp  
68 If the resource is listed in the  
69 .Pa /etc/vfstab  
70 file, you can specify either  
71 .Ar resource  
72 or  
73 .Ar mount-point  
74 as the  
75 .Nm mount  
76 command will consult the  
77 .Pa /etc/vfstab  
78 file for more information.  
79 If the  
80 .Fl F  
81 option is omitted,  
82 .Nm mount  
83 takes the file system type from the entry in the  
84 .Pa /etc/vfstab  
85 If \fImount-point\fR has any contents prior to the \fBmount\fR operation, those  
86 contents remain hidden until the resource is unmounted. An authorized user with  
87 the \fBSYS_MOUNT\fR privilege can perform a \fBmount\fR operation. Also, a user  
88 can perform SMBFS mount operations on a directory the user owns.  
89 .LP  
90 If the resource is listed in the \fB/etc/vfstab\fR file, you can specify either  
91 \fIresource\fR or \fImount-point\fR as the \fBmount\fR command will consult the  
92 \fB/etc/vfstab\fR file for more information. If the \fB-F\fR option is omitted,  
93 \fBmount\fR takes the file system type from the entry in the \fB/etc/vfstab\fR  
94 file.  
95 .Pp  
96 If the resource is not listed in the  
97 .Pa /etc/vfstab  
98 file, the command line must specify both
```

```

90 .Ar resource
91 and
92 .Ar mount-point .
93 .Pp
94 The
95 .Nm umount
96 utility detaches a mounted file system from the file system hierarchy.
97 An authorized user with the
98 .Dv SYS_MOUNT
99 privilege can perform a
100 .Nm umount
101 operation.
102 Also, a user can perform SMBFS umount operations on a directory the user owns.
103 .Pp
104 The
105 .Em network/smb/client
106 service must be enabled to successfully mount an SMB share.
107 This service is enabled, by default.
108 .Pp
109 To enable the service, enter the following
110 .Xr svcadm 1M
111 command:
112 .Bd -literal
113 # svcadm enable network/smb/client
114 .Ed
115 .Ss Operands
116 The
117 .Nm mount
118 command supports the following operands:
119 .Bl -tag -width Ds
120 .It Xo
121 .Ar resource
122 .No // Ns Oo Ar workgroup Ns \&; Ns Oc Ns
123 .Oo Ar user Ns Oo : Ns Ar password Ns Oc Ns @ Ns Oc Ns
124 .Ar server Ns / Ns Ar share
125 .Xc
126 The name of the resource to be mounted.
127 In addition to its name, you can specify the following information about the
128 resource:
129 .Bl -bullet
130 .It
131 .Ar password
132 is the password associated with
133 .Ar user .
134 If
135 .Ar password
136 is not specified, the mount first attempts to use the password stored by the
137 .Nm smbutil Cm login
138 command (if any).
139 If that password fails to authenticate, the
140 .Nm mount_smbfs
141 prompts you for a password.
142 .It
143 .Ar server
144 is the DNS or NetBIOS name of the remote computer.
145 .It
146 .Ar share
147 is the resource name on the remote server.
148 .It
149 .Ar user
150 is the remote user name.
151 If
152 .Ar user
153 is omitted, the logged in user ID is used.
154 .It
155 .Ar workgroup

```

```

156 is the name of the workgroup or the Windows domain in which the user name is
157 defined.
158 .Pp
159 .Sp
160 .Lp
161 If the resource is not listed in the \fB/etc/vfstab\fR file, the command line
162 must specify both \fIresource\fR and \fImount-point\fR.
163 .Sp
164 .Lp
165 The \fBumount\fR utility detaches a mounted file system from the file system
166 hierarchy. An authorized user with the \fBSYS_MOUNT\fR privilege can perform a
167 \fBumount\fR operation. Also, a user can perform SMBFS umount operations on a
168 directory the user owns.
169 .Sp
170 .Lp
171 To enable the service, enter the following \fBsvcadm\fR(1M) command:
172 .Sp
173 .In +2
174 .Nf
175 # \fBsvcadm enable network/smb/client\fR
176 .Fi
177 .In -2
178 .Sp
179 .Ss "Operands"
180 .Sp
181 .Lp
182 .Ss
183 The \fBmount\fR command supports the following operands:
184 .Sp
185 .Ne 2
186 .Na
187 \fB\fIresource\fR
188 //[\fIworkgroup\fR][\fIuser\fR[: \fIpassword\fR@]\fIserver\fR/\fIshare\fR\fR
189 .Ad
190 .Sp .6
191 .Rs 4n
192 .Sp
193 .Lp
194 The name of the resource to be mounted. In addition to its name, you can
195 specify the following information about the resource:
196 .Rs +4
197 .Tp
198 .Ie t \(\bu
199 .El o
200 \fIpassword\fR is the password associated with \fIuser\fR. If \fIpassword\fR is
201 not specified, the mount first attempts to use the password stored by the
202 \fBsmutil login\fR command (if any). If that password fails to authenticate,
203 the \fBmount_smbfs\fR prompts you for a password.
204 .Re
205 .Rs +4
206 .Tp
207 .Ie t \(\bu
208 .El o
209 \fIserver\fR is the DNS or NetBIOS name of the remote computer.
210 .Re
211 .Rs +4
212 .Tp
213 .Ie t \(\bu
214 .El o
215 \fIshare\fR is the resource name on the remote server.
216 .Re
217 .Rs +4

```

```

118 .TP
119 .ie t \(\bu
120 .el o
121 \fIuser\fR is the remote user name. If \fIuser\fR is omitted, the logged in
122 user ID is used.
123 .RE
124 .RS +4
125 .TP
126 .ie t \(\bu
127 .el o
128 \fIworkgroup\fR is the name of the workgroup or the Windows domain in which the
129 user name is defined.
130 .sp
159 If the resource includes a workgroup, you must escape the semicolon that
160 appears after the workgroup name to prevent it from being interpreted by the
161 command shell.
162 For instance, surround the entire resource name with double quotes:
163 .Bd -literal
164 mount -F smbfs "//SALES;george@RSERVER" /mnt
165 .Ed
166 .El
167 .It Ar mount-point
133 command shell. For instance, surround the entire resource name with double
134 quotes: \fBmount -F smbfs "//SALES;george@RSERVER" /mnt\fR.
135 .RE
136 .RE

138 .sp
139 .ne 2
140 .na
141 \fB\fImount-point\fR\fR
142 .ad
143 .sp .6
144 .RS 4n
168 The path to the location where the file system is to be mounted or unmounted.
169 The
170 .Nm mount
171 command maintains a table of mounted file systems in the
172 .Pa /etc/mnttab
173 file.
174 See the
175 .Xr mnttab 4
176 man page.
177 .El
178 .Sh OPTIONS
179 See the
180 .Xr mount 1M
181 man page for the list of supported
182 .Ar generic-options .
183 .Bl -tag -width Ds
184 .It Fl o Ar name Ns Oo = Ns Ar value Oc
185 Sets the file system-specific properties.
186 You can specify more than one name-value pair as a list of comma-separated
187 pairs.
188 No spaces are permitted in the list.
189 The properties are as follows:
190 .Bl -tag -width Ds
191 .It Cm acl Ns | Ns Cm noacl
146 The \fBmount\fR command maintains a table of mounted file systems in the
147 \fB/etc/mnttab\fR file. See the \fBmnttab\fR(4) man page.
148 .RE

150 .Sh OPTIONS
151 .sp
152 .LP
153 See the \fBmount\fR(1M) man page for the list of supported

```

```

154 \fIgeneric-options\fR.
155 .sp
156 .ne 2
157 .na
158 \fB\fB-o\fR \fIname\fR\fB=\fR\fIvalue\fR or\fR
159 .ad
160 .br
161 .na
162 \fB\fB-o\fR \fIname\fR\fR
163 .ad
164 .sp .6
165 .RS 4n
166 Sets the file system-specific properties. You can specify more than one
167 name-value pair as a list of comma-separated pairs. No spaces are permitted in
168 the list. The properties are as follows:
170 .sp
171 .ne 2
172 .na
173 \fB\fBacl\fR/\fBnoacl\fR\fR
174 .ad
175 .sp .6
176 .RS 4n
192 Enable (or disable) presentation of Access Control Lists (ACLs)
193 on files and directories under this
194 .Xr smbfs 7FS
195 mount.
196 The default behavior is
197 .Cm noacl ,
198 which presents files and directories as owned by the owner of the mount point
199 and having permissions based on
200 .Cm fileperms
201 or
202 .Cm dirperms .
203 With the
204 .Cm acl
205 mount option, files are presented with ACLs obtained from the SMB server.
206 Setting the
207 .Cm acl
208 mount option is not advised unless the system is joined to an Active Directory
209 domain and using
210 .Xr ldap 1
178 on files and directories under this \fBsmbfs\fR(7FS) mount.
179 The default behavior is \fBnoacl\fR, which presents files and
180 directories as owned by the owner of the mount point and having
181 permissions based on \fBfileperms\fR or \fBdirperms\fR.
182 With the \fBacl\fR mount option, files are presented with ACLs
183 obtained from the SMB server.
184 Setting the \fBacl\fR mount option is not advised unless the system
185 is joined to an Active Directory domain and using \fBldap\fR(1)
211 so it can correctly present ACL identities from the SMB server.
212 .It Cm dirperms Ns = Ns Ar octaltriplet
213 Specifies the permissions to be assigned to directories.
214 The value must be specified as an octal triplet, such as
215 .Ql 755 .
216 The default value for the directory mode is taken from the
217 .Cm fileperms
218 setting, with execute permission added where
219 .Cm fileperms
220 has read permission.
221 .Pp
222 Note that these permissions have no relation to the rights granted by the SMB
187 .RE

189 .sp
190 .ne 2

```

```

191 .na
192 \fB\fBdirperms=\fR\fIoctaltriplet\fR\fR
193 .ad
194 .sp .6
195 .RS 4n
196 Specifies the permissions to be assigned to directories. The value must be
197 specified as an octal triplet, such as \fB755\fR. The default value for the
198 directory mode is taken from the \fBfileperms\fR setting, with execute
199 permission added where \fBfileperms\fR has read permission.
200 .sp
201 Note that these permissions have no relation to the rights granted by the CIFS
223 server.
224 .It Cm fileperms Ns = Ns Ar octaltriplet
225 Specifies the permissions to be assigned to files.
226 The value must be specified as an octal triplet, such as
227 .Ql 644 .
228 The default value is
229 .Ql 700 .
230 .Pp
231 Note that these permissions have no relation to the rights granted by the SMB
203 .RE

205 .sp
206 .ne 2
207 .na
208 \fB\fBfileperms=\fR\fIoctaltriplet\fR\fR
209 .ad
210 .sp .6
211 .RS 4n
212 Specifies the permissions to be assigned to files. The value must be specified
213 as an octal triplet, such as \fB644\fR. The default value is \fB700\fR.
214 .sp
215 Note that these permissions have no relation to the rights granted by the CIFS
232 server.
233 .It Cm gid Ns = Ns Ar groupid
234 Assigns the specified group ID to files.
235 The default value is the group ID of the directory where the volume is mounted.
236 .It Cm intr Ns | Ns Cm nointr
237 Enable (or disable) cancellation of
238 .Xr smbfs 7FS
239 I/O operations when the user interrupts the calling thread (for example, by
240 hitting Ctrl-C while an operation is underway).
241 The default is
242 .Cm intr
243 (interruption enabled), so cancellation is normally allowed.
244 .It Cm noprompt
245 Suppresses the prompting for a password when mounting a share.
246 This property enables you to permit anonymous access to a share.
247 Anonymous access does not require a password.
248 .Pp
249 The
250 .Nm mount
251 operation fails if a password is required, the
252 .Cm noprompt
253 property is set, and no password is stored by the
254 .Nm smbutil Cm login
255 command.
256 .It Cm retry_count Ns = Ns Ar number
217 .RE

219 .sp
220 .ne 2
221 .na
222 \fB\fBgid=\fR\fIgroupid\fR\fR
223 .ad
224 .sp .6

```

```

225 .RS 4n
226 Assigns the specified group ID to files. The default value is the group ID of
227 the directory where the volume is mounted.
228 .RE

230 .sp
231 .ne 2
232 .na
233 \fB\fBintr\fR/\fBnointr\fR\fR
234 .ad
235 .sp .6
236 .RS 4n
237 Enable (or disable) cancellation of \fBsmbs\fR(7FS) I/O operations when the
238 user interrupts the calling thread (for example, by hitting Ctrl-C while an
239 operation is underway). The default is \fBintr\fR (interruption enabled), so
240 cancellation is normally allowed.
241 .RE

243 .sp
244 .ne 2
245 .na
246 \fB\fBnoprompt\fR\fR
247 .ad
248 .sp .6
249 .RS 4n
250 Suppresses the prompting for a password when mounting a share. This property
251 enables you to permit anonymous access to a share. Anonymous access does not
252 require a password.
253 .sp
254 The \fBmount\fR operation fails if a password is required, the \fBnoprompt\fR
255 property is set, and no password is stored by the \fBsmbutil login\fR command.
256 .RE

258 .sp
259 .ne 2
260 .na
261 \fB\fBretry_count=\fR\fInumber\fR\fR
262 .ad
263 .sp .6
264 .RS 4n
257 Specifies the number of SMBFS retries to attempt before the connection is
258 marked as broken.
259 By default, 4 attempts are made.
260 .Pp
261 The
262 .Cm retry_count
263 property value set by the
264 .Nm mount
265 command overrides the global value set in SMF or the value set in your
266 .Pa \&.nsmbrc
267 file.
268 .It Cm timeout Ns = Ns Ar seconds
269 Specifies the SMB request timeout.
270 By default, the timeout is 15 seconds.
271 .Pp
272 The
273 .Cm timeout
274 property value set by the
275 .Nm mount
276 command overrides the global value set in SMF or the value set in your
277 .Pa \&.nsmbrc
278 file.
279 .It Cm uid Ns = Ns Ar userid
280 Assigns the specified user ID files.
281 The default value is the owner ID of the directory where the volume is mounted.
282 .It Cm xattr Ns | Ns Cm noxattr

```

```

283 Enable (or disable) Extended Attributes in this mount point.
284 This option defaults to
285 .Cm xattr
286 (enabled Extended Attributes), but note: if the SMB server does not support SMB
287 "named streams",
288 .Xr smbfs 7FS
289 forces this option to
290 .Cm noxattr .
291 When a mount has the
292 .Cm noxattr
293 option, attempts to use Extended attributes fail with
294 .Er EINVAL .
295 .El
296 .It Fl O
297 Overlays mount.
298 Allow the file system to be mounted over an existing mount point, making the
299 underlying file system inaccessible.
300 If a mount is attempted on a pre-existing mount point without setting this flag,
301 the mount fails, producing the error "device busy."
302 .El
303 .Sh FILES
304 .Bl -tag -width Pa
305 .It Pa /etc/mnttab
306 Table of mounted file systems.
307 .It Pa /etc/dfs/fstypes
308 Default distributed file system type.
309 .It Pa /etc/vfstab
310 Table of automatically mounted resources.
311 .It Pa ${HOME}/.nsmbrc
312 User-settable mount point configuration file to store the description for each
313 connection.
314 .El
315 .Sh EXAMPLES
316 .Bl -tag -width Ds
317 .It Sy Example 1 No Mounting an SMBFS Share
318 The following example shows how to mount the
319 .Pa /tmp
320 share from the
321 .Em nano
322 server in the
323 .Em SALES
324 workgroup on the local
325 .Pa /mnt
326 mount point.
327 You must supply the password for the root user to successfully perform the mount
328 operation.
329 .Bd -literal
330 # mount -F smbfs "//SALES;root@nano.sfbay/tmp" /mnt
326 marked as broken. By default, 4 attempts are made.
327 .sp
328 The \fBretry_count\fR property value set by the \fBmount\fR command overrides
329 the global value set in SMF or the value set in your \fB&.nsmbrc\fR file.
330 .RE

322 .sp
323 .ne 2
324 .na
325 \fB\fBtimeout=\fR\fIseconds\fR\fR
326 .ad
327 .sp .6
328 .RS 4n
329 Specifies the CIFS request timeout. By default, the timeout is 15 seconds.
330 .sp
331 The \fBtimeout\fR property value set by the \fBmount\fR command overrides the
332 global value set in SMF or the value set in your \fB&.nsmbrc\fR file.
333 .RE

```

```

285 .sp
286 .ne 2
287 .na
288 \fB\fBuid=\fR\fIuserid\fR\fR
289 .ad
290 .sp .6
291 .RS 4n
292 Assigns the specified user ID files. The default value is the owner ID of the
293 directory where the volume is mounted.
294 .RE

296 .sp
297 .ne 2
298 .na
299 \fB\fBxattr\fR/\fBnoxattr\fR\fR
300 .ad
301 .sp .6
302 .RS 4n
303 Enable (or disable) Solaris Extended Attributes in this mount point. This
304 option defaults to \fBxattr\fR (enabled Extended Attributes), but note: if the
305 CIFS server does not support CIFS "named streams", \fBsmbfs\fR(7FS) forces this
306 option to \fBnoxattr\fR. When a mount has the \fBnoxattr\fR option, attempts to
307 use Solaris Extended attributes fail with EINVAL.
308 .RE

310 .RE

312 .sp
313 .ne 2
314 .na
315 \fB\fB-O\fR\fR
316 .ad
317 .sp .6
318 .RS 4n
319 Overlays mount. Allow the file system to be mounted over an existing mount
320 point, making the underlying file system inaccessible. If a mount is attempted
321 on a pre-existing mount point without setting this flag, the mount fails,
322 producing the error "device busy."
323 .RE

325 .Sh EXAMPLES
326 .LP
327 \fBExample 1\fR Mounting an SMBFS Share
328 .sp
329 .LP
330 The following example shows how to mount the \fB/tmp\fR share from the
331 \fBnano\fR server in the \fBSALES\fR workgroup on the local \fB/mnt\fR mount
332 point. You must supply the password for the \fBroot\fR user to successfully
333 perform the mount operation.

335 .sp
336 .in +2
337 .nf
338 # \fBmount -F smbfs "//SALES;root@nano.sfbay/tmp" /mnt\fR
339 Password:
340 .Ed
341 .It Sy Example 2 No Verifying That an SMBFS File System Is Mounted
342 The following example shows how to mount the
343 .Pa /tmp
344 share from the
345 .Em nano
346 server on the local
347 .Pa /mnt
348 mount point.
349 You must supply the password for the root user to successfully perform the mount

```

new/usr/src/man/man1m/mount_smbfs.1m

11

```

342 operation.
343 .Bd -literal
344 # mount -F smbfs //root@nano.sfbay/tmp /mnt
340 .fi
341 .in -2
342 .sp

344 .LP
345 fBExample 2 \fRVerifying That an SMBFS File System Is Mounted
346 .sp
347 .LP
348 The following example shows how to mount the \fB/tmp\fR share from the
349 \fBnano\fR server on the local \fB/mnt\fR mount point. You must supply the
350 password for the \fBroot\fR user to successfully perform the mount operation.

352 .sp
353 .in +2
354 .nf
355 # \fBmount -F smbfs //root@nano.sfbay/tmp /mnt\fR
345 Password:
346 .Ed
347 .Pp
357 .fi
358 .in -2
359 .sp

361 .sp
362 .LP
348 You can verify that the share is mounted in the following ways:
349 .Bl -bullet
350 .It
351 View the file system entry in the
352 .Pa /etc/mntonab
353 file.
354 .Bd -literal
355 # grep root /etc/mntonab
364 .RS +4
365 .TP
366 .ie t \(\bu
367 .el o
368 View the file system entry in the \fB/etc/mntonab\fR file.
369 .sp
370 .in +2
371 .nf
372 # \fBgrep root /etc/mntonab\fR
356 //root@nano.sfbay/tmp /mnt smbfs dev=4900000 1177097833
357 .Ed
358 .It
359 View the output of the
360 .Ql mount
361 command.
362 .Bd -literal
363 # mount | grep root
374 .fi
375 .in -2
376 .sp

378 .RE
379 .RS +4
380 .TP
381 .ie t \(\bu
382 .el o
383 View the output of the \fBmount\fR command.
384 .sp
385 .in +2
386 .nf

```

new/usr/src/man/man1m/mount_smbfs.1m

```

387 # /fBmount | grep root\fr
364 /mnt on //root@nano.sfbay/tmp read/write/setuid/devices/dev=4900000 on
365 Fri Apr 20 13:37:13 2007
366 .Ed
367 .It
368 View the output of the
369 .Ol df /mnt
370 command.
371 .Bd -literal
372 # df /mnt
390 .fi
391 .in -2
392 .sp

394 .RE
395 .RS +4
396 .TP
397 .ie t \(\bu
398 .el o
399 View the output of the \fBdf /mnt\fR command.
400 .sp
401 .in +2
402 .nf
403 # \fBdf /mnt\fR
373 /mnt          (//root@nano.sfbay/tmp): 3635872 blocks      -1 files
374 .Ed
375 .El
376 .Pp
377 Obtain information about the mounted share by viewing the output of the
378 .Ol df -k /mnt
379 command.
380 .Bd -literal
381 # df -k /mnt
405 .fi
406 .in -2
407 .sp

409 .RE
410 .sp
411 .LP
412 Obtain information about the mounted share by viewing the output of the \fBdf
413 -k /mnt\fR command.

415 .sp
416 .in +2
417 .nf
418 # \fBdf -k /mnt\fR
382 Filesystem      kbytes    used   avail capacity  Mounted on
383 //root@nano.sfbay/tmp      1882384   64448  1817936      4%    /mnt
384 .Ed
386 .It Sy Example 3 No Unmounting an SMB Share
387 This example assumes that an SMB share has been mounted on the
388 /mnt
389 mount point.
390 The following command line unmounts the share from the mount point.
391 .Bd -literal
392 # umount /mnt
393 .Ed
394 .El
395 .Sh INTERFACE STABILITY
396 .Sy Committed
397 .Sh SEE ALSO
398 .Xr ldap 1 ,
399 .Xr smbutil 1 ,
400 .Xr mount 1M .

```

```

401 .Xr mountall 1M ,
402 .Xr svcadm 1M ,
403 .Xr acl 2 ,
404 .Xr fcntl 2 ,
405 .Xr link 2 ,
406 .Xr mknod 2 ,
407 .Xr mount 2 ,
408 .Xr symlink 2 ,
409 .Xr umount 2 ,
410 .Xr mnttab 4 ,
411 .Xr nsmbrc 4 ,
412 .Xr vfstab 4 ,
413 .Xr attributes 5 ,
414 .Xr pcfs 7FS ,
415 .Xr smbfs 7FS
416 .Sh AUTHORS
417 This manual page contains material originally authored by
418 .An Boris Popov
419 .Aq Mt bp@butya.kz ,
420 .Aq Mt bp@FreeBSD.org .
421 .Sh NOTES
422 The SMB client always attempts to use
423 .Xr gethostbyname 3NSL
424 to resolve host names.
425 If the host name cannot be resolved, the SMB client uses NetBIOS name
426 resolution (NBNS).
427 By default, the SMB client permits the use of NBNS to enable SMB clients in
428 Windows environments to work without additional configuration.
429 .Pp
430 Since NBNS has been exploited in the past, you might want to disable it.
431 To disable NBNS, set the
432 .Em nbns-enabled
433 service management facility property to
434 .Cm false .
435 By default,
436 .Em nbns-enabled
437 is set to
438 .Cm true .
439 .Pp
422 .fi
423 .in -2
424 .sp

426 .LP
427 \fBExample 3\fR Unmounting a CIFS Share
428 .sp
429 .LP
430 This example assumes that a CIFS share has been mounted on the \fb/mnt\fR mount
431 point. The following command line unmounts the share from the mount point.

433 .sp
434 .in +2
435 .nf
436 # \fBumount /mnt\fR
437 .fi
438 .in -2
439 .sp

441 .Sh FILES
442 .sp
443 .ne 2
444 .na
445 \fB\fb/etc/mnttab\fR\fR
446 .ad
447 .sp .6
448 .RS 4n

```

```

449 Table of mounted file systems.
450 .RE
452 .sp
453 .ne 2
454 .na
455 \fB\fB/etc/dfs/fstypes\fR\fR
456 .ad
457 .sp .6
458 .RS 4n
459 Default distributed file system type.
460 .RE
462 .sp
463 .ne 2
464 .na
465 \fB\fB/etc/vfstab\fR\fR
466 .ad
467 .sp .6
468 .RS 4n
469 Table of automatically mounted resources.
470 .RE
472 .sp
473 .ne 2
474 .na
475 \fB\fB$HOME/.nsmbrc\fR\fR
476 .ad
477 .sp .6
478 .RS 4n
479 User-settable mount point configuration file to store the description for each
480 connection.
481 .RE
483 .Sh ATTRIBUTES
484 .sp
485 .LP
486 See the \fBattributes\fR(5) man page for descriptions of the following
487 attributes:
488 .sp
490 .sp
491 .TS
492 box;
493 c / c
494 l / l .
495 ATTRIBUTE TYPE ATTRIBUTE VALUE
496
497 Interface Stability Committed
498 .TE
500 .Sh SEE ALSO
501 .sp
502 .LP
503 \fBldap\fR(1), \fBsmbutil\fR(1),
504 \fBmount\fR(1M), \fBmountall\fR(1M), \fBsvcadm\fR(1M),
505 \fBacl\fR(2), \fBfcntl\fR(2), \fBlink\fR(2), \fBmknod\fR(2), \fBmount\fR(2),
506 \fBsymlink\fR(2), \fBumount\fR(2), \fBmnttab\fR(4), \fBnsmbrc\fR(4),
507 \fBvfstab\fR(4), \fBattributes\fR(5), \fBpcfs\fR(7FS), \fBsmbs\fR(7FS)
508 .Sh AUTHORS
509 .sp
510 .LP
511 This manual page contains material originally authored by Boris Popov,
512 \fBbp@butya.kz\fR, \fBbp@FreeBSD.org\fR.
513 .Sh NOTES
514 .sp

```

```
515 .LP
516 The Solaris CIFS client always attempts to use \fBgethostbyname()\fR to resolve
517 host names. If the host name cannot be resolved, the CIFS client uses NetBIOS
518 name resolution (NBNS). By default, the Solaris CIFS client permits the use of
519 NBNS to enable Solaris CIFS clients in Windows environments to work without
520 additional configuration.
521 .sp
522 .LP
523 Since NBNS has been exploited in the past, you might want to disable it. To
524 disable NBNS, set the \fBnbns-enabled\fR service management facility property
525 to \fBfalse\fR. By default, \fBnbns-enabled\fR is set to \fBtrue\fR.
526 .sp
527 .LP
440 If the directory on which a file system is to be mounted is a symbolic link,
441 the file system is mounted on the directory to which the symbolic link refers,
442 rather than being mounted on top of the symbolic link itself.
```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs.h

```
*****
6523 Sun Mar 4 06:09:09 2018
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs.h
5404 smbfs needs mmap support
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>
*****
1 /*
2 * Copyright (c) 2000-2001, Boris Popov
3 * All rights reserved.
4 *
5 * Redistribution and use in source and binary forms, with or without
6 * modification, are permitted provided that the following conditions
7 * are met:
8 * 1. Redistributions of source code must retain the above copyright
9 * notice, this list of conditions and the following disclaimer.
10 * 2. Redistributions in binary form must reproduce the above copyright
11 * notice, this list of conditions and the following disclaimer in the
12 * documentation and/or other materials provided with the distribution.
13 * 3. All advertising materials mentioning features or use of this software
14 * must display the following acknowledgement:
15 * This product includes software developed by Boris Popov.
16 * 4. Neither the name of the author nor the names of any co-contributors
17 * may be used to endorse or promote products derived from this software
18 * without specific prior written permission.
19 *
20 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
21 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
22 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
23 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
24 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
25 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
26 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
27 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
28 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
29 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
30 * SUCH DAMAGE.
31 *
32 * $Id: smbfs.h,v 1.30.100.1 2005/05/27 02:35:28 lindak Exp $
33 */

35 /*
36 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
37 */
38

39 #ifndef _SMBFS_SMBFS_H
40 #define _SMBFS_SMBFS_H
41
42 /*
43 * FS-specific VFS structures for smbfs.
44 * (per-mount stuff, etc.)
45 *
46 * This file used to have mount args stuff,
47 * but that's now in sys/fs/smbfs_mount.h
48 */
49
50 #include <sys/param.h>
51 #include <sys/fstyp.h>
52 #include <sys/avl.h>
53 #include <sys/list.h>
54 #include <sys/t_lock.h>
55 #include <sys/vfs.h>
56 #include <sys/vfs_opreg.h>
57 #include <sys/fs/smbfs_mount.h>
58 #include <sys/zone.h>
59
60 /*
```

1

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs.h

```
61 * Path component length
62 *
63 * The generic fs code uses MAXNAMELEN to represent
64 * what the largest component length is, but note:
65 * that length DOES include the terminating NULL.
66 * SMB_MAXNAMELEN does NOT include the NULL.
67 */
68 #define SMB_MAXNAMELEN          (MAXNAMELEN-1) /* 255 */

70 /*
71 * SM_MAX_STATFSTIME is the maximum time to cache statvfs data. Since this
72 * should be a fast call on the server, the time the data cached is short.
73 * That lets the cache handle bursts of statvfs() requests without generating
74 * lots of network traffic.
75 */
76 #define SM_MAX_STATFSTIME 2

77 /* Mask values for smbmount structure sm_status field */
78 #define SM_STATUS_STATFS_BUSY 0x00000001 /* statvfs is in progress */
79 #define SM_STATUS_STATFS_WANT 0x00000002 /* statvfs wakeup is wanted */
80 #define SM_STATUS_TIMEOUT 0x00000004 /* this mount is not responding */
81 #define SM_STATUS_DEAD 0x00000010 /* connection gone - umount this */

84 extern const struct fs_operation_def smbfs_vnodeops_template[];
85 extern struct vnodeops *smbfs_vnodeops;

87 struct smbnodes;
88 struct smb_share;

90 /*
91 * The values for smi_flags (from nfs_clnt.h)
92 */
93 #define SMI_INT          0x04          /* interrupts allowed */
94 #define SMI_NOAC         0x10          /* don't cache attributes */
95 #define SMI_LLOCK        0x80          /* local locking only */
96 #define SMI_ACL          0x2000        /* share supports ACLs */
97 #define SMI_DIRECTIO     0x40000       /* do direct I/O */
98 #define SMI_EXTATTR      0x80000       /* share supports ext. attrs */
99 #define SMI_DEAD         0x200000      /* mount has been terminated */

101 /*
102 * Stuff returned by smbfs_smb_qfsattr
103 * See [CIFS] SMB_QUERY_FS_ATTRIBUTE_INFO
104 */
105 typedef struct smb_fs_attr_info {
106     uint32_t    fsa_aflags;      /* Attr. flags [CIFS 4.1.6.6] */
107     uint32_t    fsa_maxname;    /* max. component length */
108     char        fsa_tname[FSTYPSZ]; /* type name, i.e. "NTFS" */
109 } smb_fs_attr_info_t;
109 /* unchanged_portion_omitted */
```

2

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c
```

```
*****
20424 Sun Mar  4 06:09:09 2018
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c
Kill flags arg in smbfs_purge_caches
Lots of comment cleanup
5404 smbfs needs mmap support
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>
*****
```

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
23  *
24  * Copyright (c) 1983,1984,1985,1986,1987,1988,1989 AT&T.
25  * All rights reserved.
26 */
27 */
28
29 #include <sys/param.h>
30 #include <sys/sysctl.h>
31 #include <sys/thread.h>
32 #include <sys/t_lock.h>
33 #include <sys/time.h>
34 #include <sys/vnode.h>
35 #include <sys/vfs.h>
36 #include <sys/errno.h>
37 #include <sys/buf.h>
38 #include <sys/stat.h>
39 #include <sys/cred.h>
40 #include <sys/kmem.h>
41 #include <sys/debug.h>
42 #include <sys/vmsystm.h>
43 #include <sys/flock.h>
44 #include <sys/share.h>
45 #include <sys/cmn_err.h>
46 #include <sys/tiuser.h>
47 #include <sys/sysmacros.h>
48 #include <sys/callb.h>
49 #include <sys/acl.h>
50 #include <sys/kstat.h>
51 #include <sys/signal.h>
52 #include <sys/list.h>
53 #include <sys/zone.h>
54
55 #include <netsmb/smb.h>
56 #include <netsmb/smb_conn.h>
57 #include <netsmb/smb_subr.h>
```

```
1
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c
```

```
59 #include <smbfs/smbfs.h>
60 #include <smbfs/smbfs_node.h>
61 #include <smbfs/smbfs_subr.h>
62
63 #include <vm/hat.h>
64 #include <vm/as.h>
65 #include <vm/page.h>
66 #include <vm/pvn.h>
67 #include <vm/seg.h>
68 #include <vm/seg_map.h>
69 #include <vm/seg_vn.h>
70
71 #define ATTRCACHE_VALID(vp) (gethrtime() < VTOSMB(vp)->r_attrtime)
72
73 static int smbfs_getattr_cache(vnode_t *, smbfattr_t *);
74 static void smbfattr_to_vattr(vnode_t *, smbfattr_t *, vattr_t *);
75 static void smbfattr_to_xvattr(smbfattr_t *, vattr_t *);
76 static int smbfs_getattr_otw(vnode_t *, struct smbfattr *, cred_t *);
77
78 /*
79  * The following code provide zone support in order to perform an action
80  * for each smbfs mount in a zone. This is also where we would add
81  * per-zone globals and kernel threads for the smbfs module (since
82  * they must be terminated by the shutdown callback).
83  */
84
85 struct smi_globals {
86     kmutex_t smg_lock; /* lock protecting smg_list */
87     list_t smg_list; /* list of SMBFS mounts in zone */
88     boolean_t smg_destructor_called;
89 };
90
91 typedef struct smi_globals smi_globals_t;
92
93 static zone_key_t smi_list_key;
94
95 /*
96  * Attributes caching:
97  *
98  * Attributes are cached in the smbnode in struct vattr form.
99  * There is a time associated with the cached attributes (r_attrtime)
100 * which tells whether the attributes are valid. The time is initialized
101 * to the difference between current time and the modify time of the vnode
102 * when new attributes are cached. This allows the attributes for
103 * files that have changed recently to be timed out sooner than for files
104 * that have not changed for a long time. There are minimum and maximum
105 * timeout values that can be set per mount point.
106 */
107
108 /*
109  * Helper for _validate_caches
110  * Validate caches by checking cached attributes. If they have timed out
111  * get the attributes from the server and compare mtimes. If mtimes are
112  * different purge all caches for this vnode.
113 */
114 int
115 smbfs_waitfor_purge_complete(vnode_t *vp)
116 {
117     smbnode_t *np;
118     k_sigset_t smask;
119
120     np = VTOSMB(vp);
121     if (np->r_serial != NULL && np->r_serial != curthread) {
122         mutex_enter(&np->r_statelock);
123         sigintr(&smask, VTOSMI(vp)->smi_flags & SMI_INT);
124         while (np->r_serial != NULL) {
```

```
2
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c

122             if (!cv_wait_sig(&np->r_cv, &np->r_statelock))
123                 sigunintr(&smask);
124                 mutex_exit(&np->r_statelock);
125             return (EINTR);
126         }
127     }
128     sigunintr(&smask);
129     mutex_exit(&np->r_statelock);
130 }
131 return (0);
132 }

134 /*
135  * Validate caches by checking cached attributes. If the cached
136  * attributes have timed out, then get new attributes from the server.
137  * As a side affect, this will do cache invalidation if the attributes
138  * have changed.
139 */
140 /* If the attributes have not timed out and if there is a cache
141 * invalidation being done by some other thread, then wait until that
142 * thread has completed the cache invalidation.
143 */
144 int
145 smbfs_validate_caches(
146     struct vnode *vp,
147     cred_t *cr)
148 {
149     struct smbattr fa;
150     int error;
151     struct vattr va;

152     if (ATTRCACHE_VALID(vp)) {
153         error = smbfs_waitfor_purge_complete(vp);
154         if (error)
155             return (error);
156         return (0);
157     }

158     return (smbfs_getattr_otw(vp, &fa, cr));
159     va.va_mask = AT_SIZE;
160     return (smbfsgetattr(vp, &va, cr));
161 }

162 /*
163  * Purge all of the various data caches.
164 */
165 /* Here NFS also had a flags arg to control what gets flushed.
166  * We only have the page cache, so no flags arg.
167 */
168 /* ARGSUSED */
169 /*ARGSUSED*/
170 void
171 smbfs_purge_caches(struct vnode *vp, cred_t *cr)
172 smbfs_purge_caches(struct vnode *vp)
173 {

174 #if 0 /* not yet: mmap support */
175 /*
176  * Here NFS has: Purge the DNLC for this vp,
177  * NFS: Purge the DNLC for this vp,
178  * Clear any readdir state bits,
179  * the readlink response cache, ...
180 */
181     smbnode_t *np = VTOSMB(vp);

182 #endif /* not yet: mmap support */
183 }
```

```
3 new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c
180         * Flush the page cache.
181         */
182     if (vn_has_cached_data(vp)) {
183         (void) VOP_PUTPAGE(vp, (u_offset_t)0, 0, B_INVAL, cr, NULL);
184     }
185
186     /*
187      * Here NFS has: Flush the readdir response cache.
188      * No readdir cache in smbfs.
189      */
190 } /* endif /* not yet */
191
192 /*
193  * Here NFS has:
194  * nfs_purge_raddir_cache()
195  * nfs3_cache_post_op_attr()
196  * nfs3_cache_post_op_vattr()
197  * nfs3_cache_wcc_data()
198 */
199
200 /*
201  * Check the attribute cache to see if the new attributes match
202  * those cached. If they do, the various 'data' caches are
203  * considered to be good. Otherwise, purge the cached data.
204 */
205 static void
206 smbfs_cache_check(
207     struct vnode *vp,
208     struct smbattr *fap,
209     cred_t *cr)
210 {
211     smbnode_t *np;
212     int purge_data = 0;
213     int purge_acl = 0;
214
215     np = VTOSMB(vp);
216     mutex_enter(&np->r_statelock);
217
218     /*
219      * Compare with NFS macro: CACHE_VALID
220      * If the mtime or size has changed,
221      * purge cached data.
222      */
223     if (np->r_attr.fa_mtime.tv_sec != fap->fa_mtime.tv_sec ||
224         np->r_attr.fa_mtime.tv_nsec != fap->fa_mtime.tv_nsec)
225         purge_data = 1;
226     if (np->r_attr.fa_size != fap->fa_size)
227         purge_data = 1;
228
229     if (np->r_attr.fa_ctime.tv_sec != fap->fa_ctime.tv_sec ||
230         np->r_attr.fa_ctime.tv_nsec != fap->fa_ctime.tv_nsec)
231         purge_acl = 1;
232
233     if (purge_acl) {
234         /* just invalidate r_sectattr (XXX: OK?) */
235         np->r_sectime = gethrtime();
236     }
237
238     mutex_exit(&np->r_statelock);
239
240     if (purge_data)
241         smbfs_purge_caches(vp, cr);
242     smbfs_purge_caches(vp);
243 }
```

```

241 }

243 /* Set attributes cache for given vnode using vnode attributes.
188 * From NFS: nfs_attrcache_va
189 */
190 */
191 #if 0 /* not yet (not sure if we need this) */
192 void
193 smbfs_attrcache_va(vnode_t *vp, struct vattr *vap)
194 {
195     smbattr_t fa;
196     smbnode_t *np;

198     vattr_to_fattr(vp, vap, &fa);
199     smbfs_attrcache_fa(vp, &fa);
200 }
201#endif /* not yet */

203 /*
244 * Set attributes cache for given vnode using SMB fattr
245 * and update the attribute cache timeout.
246 */
247 * Based on NFS: nfs_attrcache, nfs_attrcache_va
207 * From NFS: nfs_attrcache, nfs_attrcache_va
248 */
249 void
250 smbfs_attrcache_fa(vnode_t *vp, struct smbattr *fap)
251 {
252     smbnode_t *np;
253     smbmtinfo_t *smi;
254     hrtimetime_t delta, now;
255     u_offset_t newsize;
256     vtype_t vtype, oldvdt;
257     mode_t mode;

259     np = VTOSMB(vp);
260     smi = VTOSMI(vp);

262     /*
263      * We allow v_type to change, so set that here
264      * (and the mode, which depends on the type).
265      */
266     if (fap->fa_attr & SMB_FA_DIR) {
267         vtype = VDIR;
268         mode = smi->smi_dmode;
269     } else {
270         vtype = VREG;
271         mode = smi->smi_fmode;
272     }

274     mutex_enter(&np->r_statelock);
275     now = gethrtime();
276
277     /*
278      * Delta is the number of nanoseconds that we will
279      * cache the attributes of the file. It is based on
280      * the number of nanoseconds since the last time that
281      * we detected a change. The assumption is that files
282      * that changed recently are likely to change again.
283      * There is a minimum and a maximum for regular files
284      * and for directories which is enforced though.
285      *
286      * Using the time since last change was detected
287      * eliminates direct comparison or calculation
288      * using mixed client and server times. SMBFS
289      * does not make any assumptions regarding the

```

```

290             * client and server clocks being synchronized.
291             */
292             if (fap->fa_mtime.tv_sec != np->r_attr.fa_mtime.tv_sec ||
293                 fap->fa_mtime.tv_nsec != np->r_attr.fa_mtime.tv_nsec ||
294                 fap->fa_size != np->r_attr.fa_size)
295                 np->r_mtime = now;

297             if ((smi->smi_flags & SMI_NOAC) || (vp->v_flag & VNOCACHE))
298                 delta = 0;
299             else {
300                 delta = now - np->r_mtime;
301                 if (vtype == VDIR) {
302                     if (delta < smi->smi_acdirmn)
303                         delta = smi->smi_acdirmn;
304                     else if (delta > smi->smi_acdirmax)
305                         delta = smi->smi_acdirmax;
306                 } else {
307                     if (delta < smi->smi_acregmn)
308                         delta = smi->smi_acregmn;
309                     else if (delta > smi->smi_acregmax)
310                         delta = smi->smi_acregmax;
311                 }
312             }

314             np->r_attrtime = now + delta;
315             np->r_attr = *fap;
316             np->n_mode = mode;
317             oldvdt = vp->v_type;
318             vp->v_type = vtype;

320             /*
321              * Shall we update r_size? (local notion of size)
322              *
323              * The real criteria for updating r_size should be:
324              * if the file has grown on the server, or if
325              * the client has not modified the file.
326              *
327              * Also deal with the fact that SMB presents
328              * directories as having size=0. Doing that
329              * here and leaving fa_size as returned OTW
330              * avoids fixing the size lots of places.
331              */
332             newsize = fap->fa_size;
333             if (vtype == VDIR && newsize < DEV_BSIZE)
334                 newsize = DEV_BSIZE;

336             if (np->r_size != newsize &&
337                 (!vn_has_cached_data(vp) ||
338                  (!np->r_flags & RDIRTY) && np->r_count == 0))) {
339                 if (np->r_size != newsize) {
340                     /* if (!vn_has_cached_data(vp) || ...) */
341                     /* XXX: See NFS page cache code. */
342                 }
343
344                 /*
345                  * Here NFS has:
346                  * nfs_setswaplike(vp, va);
347                  * np->r_flags &= ~RWRITEATTR;
348                  * (not needed here)
349                  */
350
351             /* NFS: np->r_flags &= ~RWRITEATTR; */

```

```

350         np->n_flag &= ~NATTRCHANGED;
351         mutex_exit(&np->r_statelock);
353         if (oldvt != vtype) {
354             SMBVDEBUG("vtype change %d to %d\n", oldvt, vtype);
355         }
356     }


---


388     /*
389      * Get attributes over-the-wire and update attributes cache
390      * if no error occurred in the over-the-wire operation.
391      * Return 0 if successful, otherwise error.
392      * From NFS: nfs_getattr_otw
393      */
394 static int
395 int
396 smbfs_getattr_otw(vnode_t *vp, struct smbfattr *fap, cred_t *cr)
397 {
398     struct smbnode *np;
399     struct smb_cred scred;
400     int error;
401
402     np = VTOSMB(vp);
403
404     /*
405      * Here NFS uses the ACL RPC (if smi_flags & SMI_ACL)
406      * NFS uses the ACL rpc here (if smi_flags & SMI_ACL)
407      * With SMB, getting the ACL is a significantly more
408      * expensive operation, so we do that only when asked
409      * for the uid/gid. See smbfsgetattr().
410
411     /* Shared lock for (possible) n_fid use. */
412     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
413         return (EINVAL);
414     smb_credinit(&scred, cr);
415
416     bzero(fap, sizeof (*fap));
417     error = smbfs_smb_getfattr(np, fap, &scred);
418
419     smb_credrele(&scred);
420     smbfs_rw_exit(&np->r_lkserlock);
421
422     if (error) {
423         /*
424          * Here NFS has: PURGE_STALE_FH(error, vp, cr) */
425         /* NFS had: PURGE_STALE_FH(error, vp, cr) */
426         smbfs_attrcache_remove(np);
427         if (error == ENOENT || error == ENOTDIR) {
428             /*
429              * Getattr failed because the object was
430              * removed or renamed by another client.
431              * Remove any cached attributes under it.
432            */
433             smbfs_attrcache_prune(np);
434         }
435         return (error);
436     }
437
438     /*
439      * Here NFS has: nfs_cache_fattr(vap, fa, vap, t, cr);
440      * NFS: smbfs_cache_fattr(vap, fa, vap, t, cr);
441      * which did: fattr_to_vattr, nfs_attr_cache.
442      * We cache the fattr form, so just do the
443      * cache check and store the attributes.

```

```

440         */
441         smbfs_cache_check(vp, fap, cr);
442         smbfs_cache_check(vp, fap);
443         smbfs_attrcache_fa(vp, fap);
444
445     }


---


446     /*
447      * Return either cached or remote attributes. If we get remote attrs,
448      * Return either cached or remote attributes. If get remote attr
449      * use them to check and invalidate caches, then cache the new attributes.
450      */
451     /* From NFS: nfsggetattr()
452     */
453     int
454     smbfsgetattr(vnode_t *vp, struct vattr *vap, cred_t *cr)
455     {
456         struct smbfattr fa;
457         smbmtinfo_t *smi;
458         uint_t mask;
459         int error;
460
461         smi = VTOSMI(vp);
462
463         ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);
464
465         /*
466          * If asked for UID or GID, update n_uid, n_gid.
467          */
468         mask = AT_ALL;
469         if (vap->va_mask & (AT_UID | AT_GID)) {
470             if (smi->smi_flags & SMI_ACL)
471                 (void) smbfs_acl_getids(vp, cr);
472             /* else leave as set in make_smbnode */
473         } else {
474             mask &= ~(AT_UID | AT_GID);
475         }
476
477         /*
478          * If we've got cached attributes, just use them;
479          * otherwise go to the server to get attributes,
480          * which will update the cache in the process.
481          */
482         error = smbfs_getattr_cache(vp, &fa);
483         if (error)
484             error = smbfs_getattr_otw(vp, &fa, cr);
485         if (error)
486             return (error);
487         vap->va_mask |= mask;
488
489         /*
490          * Re. client's view of the file size, see:
491          * smbfs_attrcache_fa, smbfs_getattr_otw
492          */
493         smbfattr_to_vattr(vp, &fa, vap);
494         if (vap->va_mask & AT_XVATTR)
495             smbfattr_to_xvattr(&fa, vap);
496
497     }


---


498 }


---


499     /*
500      * Here NFS has:
501      * nfs_async... stuff

```

```

599 * which we're not using (no async I/O), and:
600 *      writerp(),
601 *      nfs_putpages()
602 *      nfs_invalidate_pages()
603 * which we have in smbfs_vnops.c, and
604 *      nfs_printfhandle()
605 *      nfs_write_error()
606 * not needed here.
607 */
608
609 /*
610 * Helper function for smbfs_sync
611 *
612 * Walk the per-zone list of smbfs mounts, calling smbfs_rflush
613 * on each one. This is a little tricky because we need to exit
614 * the list mutex before each _rflush call and then try to resume
615 * where we were in the list after re-entering the mutex.
616 */
617 void
618 smbfs_flushall(cred_t *cr)
619 {
620     smi_globals_t *smg;
621     smbmntinfo_t *tmp_smi, *cur_smi, *next_smi;
622
623     smg = zone_getspecific(smi_list_key, crgetzone(cr));
624     ASSERT(smg != NULL);
625
626     mutex_enter(&smg->smg_lock);
627     cur_smi = list_head(&smg->smg_list);
628     if (cur_smi == NULL) {
629         mutex_exit(&smg->smg_lock);
630         return;
631     }
632     VFS_HOLD(cur_smi->smi_vfsp);
633     mutex_exit(&smg->smg_lock);
634
635 flush:
636     smbfs_rflush(cur_smi->smi_vfsp, cr);
637
638     mutex_enter(&smg->smg_lock);
639     /*
640      * Resume after cur_smi if that's still on the list,
641      * otherwise restart at the head.
642      */
643     for (tmp_smi = list_head(&smg->smg_list);
644          tmp_smi != NULL;
645          tmp_smi = list_next(&smg->smg_list, tmp_smi))
646         if (tmp_smi == cur_smi)
647             break;
648         if (tmp_smi != NULL)
649             next_smi = list_next(&smg->smg_list, tmp_smi);
650         else
651             next_smi = list_head(&smg->smg_list);
652
653         if (next_smi != NULL)
654             VFS_HOLD(next_smi->smi_vfsp);
655         VFS_RELEASE(cur_smi->smi_vfsp);
656
657         mutex_exit(&smg->smg_lock);
658
659         if (next_smi != NULL) {
660             cur_smi = next_smi;
661             goto flush;
662         }
663 }

```

```

665 /*
666  * SMB Client initialization and cleanup.
667  * Much of it is per-zone now.
668 */
669
670 /* ARGSUSED */
671 static void *
672 smbfs_zone_init(zoneid_t zoneid)
673 {
674     smi_globals_t *smg;
675
676     smg = kmem_alloc(sizeof (*smg), KM_SLEEP);
677     mutex_init(&smg->smg_lock, NULL, MUTEX_DEFAULT, NULL);
678     list_create(&smg->smg_list, sizeof (smbmntinfo_t),
679                 offsetof(smbmntinfo_t, smi_zone_node));
680     smg->smg_destructor_called = B_FALSE;
681
682     return (smg);
683 }
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2689
2
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_node.c          1
*****
6280 Sun Mar  4 06:09:09 2018
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_node.c
Lots of comment cleanup
*****
_____unchanged_portion_omitted_____
204 /*
205  * smbfs_attrcache_enter, smbfs_attrcache_lookup replaced by
206  * code more closely resembling NFS.  See smbfs_client.c
207 */
208 void
209 smbfs_attr_touchdir(struct smbnode *dnp)
210 {
212     mutex_enter(&dnp->r_statelock);
214     /*
215      * Now that we keep the client's notion of mtime
216      * separately from the server, this is easy.
217      */
218     dnp->r_mtime = gethrtime();
220     /*
221      * Invalidate the cache, so that we go to the wire
222      * to check that the server doesn't have a better
223      * timestamp next time we care.
224      */
225     smbfs_attrcache_rm_locked(dnp);
226     mutex_exit(&dnp->r_statelock);
227 }
_____unchanged_portion_omitted_____

```

```
*****
11806 Sun Mar  4 06:09:09 2018
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_node.h
5404 smbfs needs mmap support
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>
*****
unchanged portion omitted

125 /*
126 * Below is the SMBFS-specific representation of a "node".
127 * This struct is a mixture of Sun NFS and Darwin code.
128 * Fields starting with "r_" came from NFS struct "rnode"
129 * and fields starting with "n_" came from Darwin, or
130 * were added during the Solaris port. We have avoided
131 * renaming fields so we would not cause excessive
132 * changes in the code using this struct.
133 *
134 * Now using an AVL tree instead of hash lists, but kept the
135 * "hash" in some member names and functions to reduce churn.
136 * One AVL tree per mount replaces the global hash buckets.
137 *
138 * Notes carried over from the NFS code:
139 *
140 * The smbnode is the "inode" for remote files. It contains all the
141 * information necessary to handle remote file on the client side.
142 *
143 * Note on file sizes: we keep two file sizes in the smbnode: the size
144 * according to the client (r_size) and the size according to the server
145 * (r_attr.fa_size). They can differ because we modify r_size during a
146 * write system call (smbfs_rdwr), before the write request goes over the
147 * wire (before the file is actually modified on the server). If an OTW
148 * request occurs before the cached data is written to the server the file
149 * size returned from the server (r_attr.fa_size) may not match r_size.
150 * r_size is the one we use, in general. r_attr.fa_size is only used to
151 * determine whether or not our cached data is valid.
152 *
153 * Each smbnode has 3 locks associated with it (not including the smbnode
154 * "hash" AVL tree and free list locks):
155 *
156 *     r_rwlock:    Serializes smbfs_write and smbfs_setattr requests
157 *                   and allows smbfs_read requests to proceed in parallel.
158 *                   Serializes reads/updates to directories.
159 *
160 *     r_lkserlock: Serializes lock requests with map, write, and
161 *                   readahead operations.
162 *
163 *     r_statelock: Protects all fields in the smbnode except for
164 *                   those listed below. This lock is intended
165 *                   to be held for relatively short periods of
166 *                   time (not across entire putpage operations,
167 *                   for example).
168 *
169 * The following members are protected by the mutex smbfreelist_lock:
170 *     r_freef
171 *     r_freeb
172 *
173 * The following members are protected by the AVL tree rwlock:
174 *     r_avl_node      (r_hdr.hdr_avl_node)
175 *
176 * Note: r_modaddr is only accessed when the r_statelock mutex is held.
177 * Its value is also controlled via r_rwlock. It is assumed that
178 * there will be only 1 writer active at a time, so it safe to
179 * set r_modaddr and release r_statelock as long as the r_rwlock
180 * writer lock is held.
181 *
182 * 64-bit offsets: the code formerly assumed that atomic reads of
```

```
183 * r_size were safe and reliable; on 32-bit architectures, this is
184 * not true since an intervening bus cycle from another processor
185 * could update half of the size field. The r_statelock must now
186 * be held whenever any kind of access of r_size is made.
187 *
188 * Lock ordering:
189 *     r_rwlock > r_lkserlock > r_statelock
190 */

192 typedef struct smbnode {
193     /* Our linkage in the node cache AVL tree (see above). */
194     smbfs_node_hdr_t r_hdr;
195
196     /* short-hand names for r_hdr members */
197 #define r_avl_node    r_hdr.hdr_avl_node
198 #define n_rpath      r_hdr.hdr_n_rpath
199 #define n_rplen     r_hdr.hdr_n_rplen
200
201     smbmntinfo_t   *n_mount;      /* VFS data */
202     vnode_t        *r_vnode;     /* associated vnode */
203
204     /*
205      * Linkage in smbfreelist, for reclaiming nodes.
206      * Lock for the free list is: smbfreelist_lock
207      */
208     struct smbnode *r_freef;     /* free list forward pointer */
209     struct smbnode *r_freeb;     /* free list back pointer */
210
211     smbfs_rwlock_t r_rwlock;    /* serialize write/setattr requests */
212     smbfs_rwlock_t r_lkserlock; /* serialize lock with other ops */
213     kmutex_t       r_statelock; /* protect (most) smbnode fields */
214
215     /*
216      * File handle, directory search handle,
217      * and reference counts for them, etc.
218      * Lock for these is: r_lkserlock
219      */
220     int             n_dirrefs;
221     struct smbfs_fctx *n_dirseq; /* ff context */
222     int             n_dirofs;    /* last ff offset */
223     int             n_fidrefs;
224     uint16_t        n_fid;       /* file handle */
225     enum vtype     n_ovtype;   /* vnode type opened */
226     uint32_t        n_rights;   /* granted rights */
227     int             n_vcgenid;  /* gereration no. (reconnect) */
228
229     /*
230      * Misc. bookkeeping
231      */
232     cred_t          *r_cred;    /* current credentials */
233     u_offset_t      r_nextr;   /* next read offset (read-ahead) */
234     long            r_mapcnt;  /* count of mmapped pages */
235     uint_t          r_inmap;   /* to serialize read/write and mmap */
236     uint_t          r_count;   /* # of refs not reflect in v_count */
237     uint_t          r_awcount; /* # of outstanding async write */
238     uint_t          r_gcount;  /* getatrrs waiting to flush pages */
239     uint_t          r_flags;   /* flags, see below */
240     uint32_t        n_flag;    /* N--- flags below */
241     uint32_t        n_flag;    /* XXXX flags below */
242     uint_t          r_error;   /* async write error */
243     kcondvar_t     r_cv;      /* condvar for blocked threads */
244     avl_tree_t     r_dir;     /* cache of readdir responses */
245     rddir_cache_t *r_direof; /* pointer to the EOF entry */
246     u_offset_t     r_modaddr; /* address for page in writenp */
247     kthread_t      *r_serial; /* id of purging thread */
248     list_t         r_indelmap; /* list of delmap callers */
```

```
249      /*
250       * Attributes: local, and as last seen on the server.
251       * See notes above re: r_size vs r_attr.fa_size, etc.
252       */
253     smbattr_t      r_attr;          /* attributes from the server */
254     hrtime_t       r_attrtime;    /* time attributes become invalid */
255     hrtime_t       r_mtime;        /* client time file last modified */
256     len_t          r_size;         /* client's view of file size */

258     /*
259      * Security attributes.
260      */
261     vsecattr_t     r_secattr;
262     hrtime_t       r_sectime;

264     /*
265      * Other attributes, not carried in smbattr_t
266      */
267     u_longlong_t   n_ino;
268     uid_t          n_uid;
269     gid_t          n_gid;
270     mode_t         n_mode;
271 } smbnode_t;


---

unchanged portion omitted
```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr.h

```
*****  
11294 Sun Mar 4 06:09:09 2018  
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr.h  
Implement ioctl _FIODIRECTIO  
Kill flags arg in smbfs_purge_caches  
5404 smbfs needs mmap support  
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>  
*****  
unchanged portion omitted
```

```
149 typedef struct smbfs_fctx smbfs_fctx_t;  
  
151 #define f_rq    f_urq.uf_rq  
152 #define f_t2    f_urq.uf_t2  
  
154 /*  
155 * smb level (smbfs_smb.c)  
156 */  
157 int  smbfs_smb_lock(struct smbnode *np, int op, caddr_t id,  
158          offset_t start, uint64_t len, int largelock,  
159          struct smb_cred *scrp, uint32_t timeout);  
160 int  smbfs_smb_qfsattr(struct smb_share *ssp, struct smb_fs_attr_info *,  
161          struct smb_cred *scrp);  
162 int  smbfs_smb_statfs(struct smb_share *ssp, statvfs64_t *sbp,  
163          struct smb_cred *scrp);  
  
165 int  smbfs_smb_setdisp(struct smbnode *np, uint16_t fid, uint8_t newdisp,  
166          struct smb_cred *scrp);  
167 int  smbfs_smb_setfsizze(struct smbnode *np, uint16_t fid, uint64_t newsize,  
168          struct smb_cred *scrp);  
  
170 int  smbfs_smb_getfattr(struct smbnode *np, struct smbattr *fap,  
171          struct smb_cred *scrp);  
  
173 int  smbfs_smb_setfattr(struct smbnode *np, int fid,  
174          uint32_t attr, struct timespec *mtime, struct timespec *atime,  
175          struct smb_cred *scrp);  
  
177 int  smbfs_smb_open(struct smbnode *np, const char *name, int nmllen,  
178          int xattr, uint32_t rights, struct smb_cred *scrp,  
179          uint16_t *fidp, uint32_t *rightsp, struct smbattr *fap);  
180 int  smbfs_smb_tmopen(struct smbnode *np, uint32_t rights,  
181          struct smb_cred *scrp, uint16_t *fidp);  
182 int  smbfs_smb_close(struct smb_share *ssp, uint16_t fid,  
183          struct timespec *mtime, struct smb_cred *scrp);  
184 int  smbfs_smb_tmclose(struct smbnode *ssp, uint16_t fid,  
185          struct smb_cred *scrp);  
186 int  smbfs_smb_create(struct smbnode *dnp, const char *name, int nmllen,  
187          int xattr, uint32_t rights, struct smb_cred *scrp, uint16_t *fidp);  
188 int  smbfs_smb_delete(struct smbnode *np, struct smb_cred *scrp,  
189          const char *name, int len, int xattr);  
190 int  smbfs_smb_rename(struct smbnode *src, struct smbnode *tdnp,  
191          const char *tname, int tnrlen, struct smb_cred *scrp);  
192 int  smbfs_smb_t2rename(struct smbnode *np, const char *tname, int tnrlen,  
193          struct smb_cred *scrp, uint16_t fid, int replace);  
194 int  smbfs_smb_move(struct smbnode *src, struct smbnode *tdnp,  
195          const char *tname, int tnrlen, uint16_t flags, struct smb_cred *scrp);  
196 int  smbfs_smb_mkdir(struct smbnode *dnp, const char *name, int len,  
197          struct smb_cred *scrp);  
198 int  smbfs_smb_rmdir(struct smbnode *np, struct smb_cred *scrp);  
199 int  smbfs_smb_findopen(struct smbnode *dnp, const char *wildcard, int wclen,  
200          int attr, struct smb_cred *scrp, struct smbfs_fctx **ctxpp);  
201 int  smbfs_smb_findnext(struct smbfs_fctx *ctx, int limit,  
202          struct smb_cred *scrp);  
203 int  smbfs_smb_findclose(struct smbfs_fctx *ctx, struct smb_cred *scrp);  
204 int  smbfs_fullpath(struct mbcchain *mbp, struct smb_vc *vcp,  
205          struct smbnode *dnp, const char *name, int nmllen, uint8_t sep);
```

1

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr.h

```
206 int  smbfs_smb_lookup(struct smbnode *dnp, const char **namep, int *nmllenp,  
207          struct smbattr *fap, struct smb_cred *scrp);  
208 int  smbfs_smb_hideit(struct smbnode *np, const char *name, int len,  
209          struct smb_cred *scrp);  
210 int  smbfs_smb_unhideit(struct smbnode *np, const char *name, int len,  
211          struct smb_cred *scrp);  
212 int  smbfs_smb_flush(struct smbnode *np, struct smb_cred *scrp);  
213 int  smbfs_Oextend(vnode_t *vp, uint16_t fid, len_t from, len_t to,  
214          struct smb_cred *scrdp, int timo);  
  
216 /* get/set security descriptor */  
217 int  smbfs_smb_getsec_m(struct smb_share *ssp, uint16_t fid,  
218          struct smb_cred *scrp, uint32_t selector,  
219          mblk_t **res, uint32_t *reslen);  
220 int  smbfs_smb_setsec_m(struct smb_share *ssp, uint16_t fid,  
221          struct smb_cred *scrp, uint32_t selector, mblk_t **mp);  
  
223 /*  
224 * VFS-level init, fini stuff  
225 */  
  
227 int  smbfs_vfsinit(void);  
228 void smbfs_vfsfini(void);  
229 int  smbfs_subinit(void);  
230 void smbfs_subfini(void);  
231 int  smbfs_clninit(void);  
232 void smbfs_clntfini(void);  
  
234 void smbfs_zonelist_add(smbmntinfo_t *smi);  
235 void smbfs_zonelist_remove(smbmntinfo_t *smi);  
  
237 int  smbfs_check_table(struct vfs *vfp, struct smbnode *srp);  
238 void smbfs_destroy_table(struct vfs *vfp);  
239 void smbfs_rflush(struct vfs *vfp, cred_t *cr);  
240 void smbfs_flushall(cred_t *cr);  
  
242 int  smbfs_directio(vnode_t *vp, int cmd, cred_t *cr);  
  
244 uint32_t smbfs_newnum(void);  
245 int  smbfs_newname(char *buf, size_t buflen);  
  
247 /*  
248 * Function definitions - those having to do with  
249 * smbfs nodes, vnodes, etc  
250 */  
  
252 void smbfs_attrcache_prune(struct smbnode *np);  
253 void smbfs_attrcache_remove(struct smbnode *np);  
254 void smbfs_attrcache_rm_locked(struct smbnode *np);  
255 #ifndef DEBUG  
256 #define smbfs_attrcache_rm_locked(np)  (np)->r_attrtime = gethrtime()  
257#endif  
258 void smbfs_attr_touchdir(struct smbnode *dnp);  
259 void smbfs_attrcache_fa(vnode_t *vp, struct smbattr *fap);  
260 void smbfs_cache_check(struct vnode *vp, struct smbattr *fap);  
  
261 int  smbfs_validate_caches(struct vnode *vp, cred_t *cr);  
262 void smbfs_purge_caches(struct vnode *vp, cred_t *cr);  
  
264 void smbfs_addfree(struct smbnode *sp);  
265 void smbfs_rhash(struct smbnode *);  
  
267 /* See avl_create in smbfs_vfsops.c */  
268 void smbfs_init_hash(avl_tree_t *);  
  
270 uint32_t smbfs_gethash(const char *rpath, int prlen);
```

2

```
271 uint32_t smbfs_getino(struct smbnode *dnp, const char *name, int nmllen);
273 extern struct smbattr smbfs_fattr;
274 smbnode_t *smbfs_node_findcreate(smbmntinfo_t *mi,
275     const char *dir, int dirlen,
276     const char *name, int nmllen,
277     char sep, struct smbattr *fap);
279 int smbfs_nget(vnode_t *dvp, const char *name, int nmllen,
280     struct smbattr *fap, vnode_t **vpp);
282 void smbfs_fname_tolocal(struct smbfs_fctx *ctx);
283 char *smbfs_name_alloc(const char *name, int nmllen);
284 void smbfs_name_free(const char *name, int nmllen);
286 int smbfs_read vnode(vnode_t *, uio_t *, cred_t *, struct vattr *);
287 int smbfs_write vnode(vnode_t *vp, uio_t *uiop, cred_t *cr,
288     int ioflag, int timo);
289 int smbfsgetat t(vnode_t *vp, struct vattr *vap, cred_t *cr);
291 /* nfs: writerp writenp */
292 /* nfs_putpages? */
293 void smbfs_invalidate_pages(vnode_t *vp, u_offset_t off, cred_t *cr);
295 /* smbfs ACL support */
296 int smbfs_acl_getids(vnode_t *, cred_t *);
297 int smbfs_acl_setids(vnode_t *, vattr_t *, cred_t *);
298 int smbfs_acl_getvsa(vnode_t *, vsecattr_t *, int, cred_t *);
299 int smbfs_acl_setvsa(vnode_t *, vsecattr_t *, int, cred_t *);
300 int smbfs_acl_iocget(vnode_t *, intptr_t, int, cred_t *);
301 int smbfs_acl_iocset(vnode_t *, intptr_t, int, cred_t *);
303 /* smbfs_xattr.c */
304 int smbfs_get_xattrdir(vnode_t *dvp, vnode_t **vpp, cred_t *cr, int);
305 int smbfs_xa_parent(vnode_t *vp, vnode_t **vpp);
306 int smbfs_xa_exists(vnode_t *vp, cred_t *cr);
307 int smbfs_xa_getfattr(struct smbnode *np, struct smbattr *fap,
308     struct smb_cred *scrp);
309 int smbfs_xa_fndopen(struct smbfs_fctx *ctx, struct smbnode *dnp,
310     const char *name, int nmllen);
311 int smbfs_xa_fndnext(struct smbfs_fctx *ctx, uint16_t limit);
312 int smbfs_xa_fndclose(struct smbfs_fctx *ctx);
314 /* For Solaris, interruptible rwlock */
315 int smbfs_rw_enter_sig(smbfs_rwlock_t *l, krw_t rw, int intr);
316 int smbfs_rw_tryenter(smbfs_rwlock_t *l, krw_t rw);
317 void smbfs_rw_exit(smbfs_rwlock_t *l);
318 int smbfs_rw_lock_held(smbfs_rwlock_t *l, krw_t rw);
319 void smbfs_rw_init(smbfs_rwlock_t *l, char *name, krw_type_t type, void *arg);
320 void smbfs_rw_destroy(smbfs_rwlock_t *l);
322 #endif /* !_FS_SMBFS_SMBFS_SUBR_H_ */
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.c
```

```
*****
```

```
32428 Sun Mar 4 06:09:09 2018
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.c
```

```
cstyle
```

```
Implement ioctl_FIODIRECTIO
```

```
Lots of comment cleanup
```

```
5404 smbfs needs mmap support
```

```
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>
```

```
*****
```

```
1 /*  
2  * CDDL HEADER START  
3  *  
4  * The contents of this file are subject to the terms of the  
5  * Common Development and Distribution License (the "License").  
6  * You may not use this file except in compliance with the License.  
7  *  
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE  
9  * or http://www.opensolaris.org/os/licensing.  
10 * See the License for the specific language governing permissions  
11 * and limitations under the License.  
12 *  
13 * When distributing Covered Code, include this CDDL HEADER in each  
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.  
15 * If applicable, add the following below this CDDL HEADER, with the  
16 * fields enclosed by brackets "[]" replaced with your own identifying  
17 * information: Portions Copyright [yyyy] [name of copyright owner]  
18 *  
19 * CDDL HEADER END  
20 */  
21 /*  
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.  
23 * Use is subject to license terms.  
24 *  
25 * Copyright (c) 1983,1984,1985,1986,1987,1988,1989 AT&T.  
26 * All rights reserved.  
27 */  
28 /*  
29 * Copyright (c) 2017 by Delphix. All rights reserved.  
30 */  
31 /*  
32 * Node hash implementation initially borrowed from NFS (nfs_subr.c)  
33 * but then heavily modified. It's no longer an array of hash lists,  
34 * but an AVL tree per mount point. More on this below.  
35 */  
36 /*  
37 #include <sys/param.h>  
38 #include <sys/sysm.h>  
39 #include <sys/time.h>  
40 #include <sys/vnode.h>  
41 #include <sys/bitmap.h>  
42 #include <sys/dnlc.h>  
43 #include <sys/kmem.h>  
44 #include <sys/sunddi.h>  
45 #include <sys/sysmacros.h>  
46 #include <sys/fcntl.h>  
47 #include <sys/fcntl.h>  
48 #include <netsmb/smb_osdep.h>  
49 #include <netsmb/smb.h>  
50 #include <netsmb/smb_conn.h>  
51 #include <netsmb/smb_subr.h>  
52 #include <netsmb/smb_rq.h>  
53 #include <smbfs/smbfs.h>  
54 #include <smbfs/smbfs_node.h>  
55 #include <smbfs/smbfs.h>  
56 #include <smbfs/smbfs_node.h>
```

```
1
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.c
```

```
58 #include <smbfs/smbfs_subr.h>  
59 /*  
60  * The AVL trees (now per-mount) allow finding an smbfs node by its  
61  * full remote path name. It also allows easy traversal of all nodes  
62  * below (path wise) any given node. A reader/writer lock for each  
63  * (per mount) AVL tree is used to control access and to synchronize  
64  * lookups, additions, and deletions from that AVL tree.  
65  *  
66  * Previously, this code use a global array of hash chains, each with  
67  * its own rwlock. A few struct members, functions, and comments may  
68  * still refer to a "hash", and those should all now be considered to  
69  * refer to the per-mount AVL tree that replaced the old hash chains.  
70  * (i.e. member smi_hash_lk, function sn_hashfind, etc.)  
71  *  
72  * The smbnodes freelist is organized as a doubly linked list with  
73  * a head pointer. Additions and deletions are synchronized via  
74  * a single mutex.  
75  *  
76  * In order to add an smbnodes to the free list, it must be linked into  
77  * the mount's AVL tree and the exclusive lock for the AVL must be held.  
78  * If an smbnodes is not linked into the AVL tree, then it is destroyed  
79  * because it represents no valuable information that can be reused  
80  * about the file. The exclusive lock for the AVL tree must be held  
81  * in order to prevent a lookup in the AVL tree from finding the  
82  * smbnodes and using it and assuming that the smbnodes is not on the  
83  * freelist. The lookup in the AVL tree will have the AVL tree lock  
84  * held, either exclusive or shared.  
85  *  
86  * The vnode reference count for each smbnodes is not allowed to drop  
87  * below 1. This prevents external entities, such as the VM  
88  * subsystem, from acquiring references to vnodes already on the  
89  * freelist and then trying to place them back on the freelist  
90  * when their reference is released. This means that when an  
91  * smbnodes is looked up in the AVL tree, then either the smbnodes  
92  * is removed from the freelist and that reference is transferred to  
93  * the new reference or the vnode reference count must be incremented  
94  * accordingly. The mutex for the freelist must be held in order to  
95  * accurately test to see if the smbnodes is on the freelist or not.  
96  * The AVL tree lock might be held shared and it is possible that  
97  * two different threads may race to remove the smbnodes from the  
98  * freelist. This race can be resolved by holding the mutex for the  
99  * freelist. Please note that the mutex for the freelist does not  
100 * need to be held if the smbnodes is not on the freelist. It can not be  
101 * placed on the freelist due to the requirement that the thread  
102 * putting the smbnodes on the freelist must hold the exclusive lock  
103 * for the AVL tree and the thread doing the lookup in the AVL tree  
104 * is holding either a shared or exclusive lock for the AVL tree.  
105 *  
106 * The lock ordering is:  
107 *  
108 *      AVL tree lock -> vnode lock  
109 *      AVL tree lock -> freelist lock  
110 */  
111 /*  
112 static kmutex_t smbfreelist_lock;  
113 static smbnodes_t *smbfreelist = NULL;  
114 static ulong_t smbnodenew = 0;  
115 long nsmbnode = 0;  
116  
117 static struct kmem_cache *smbnode_cache;  
118 static const vsecattr_t smbfs_vsa0 = { 0 } ;  
119  
120 /*  
121  * Mutex to protect the following variables:
```

```
2
```

```

124 *      smbfs_major
125 *      smbfs_minor
126 */
127 kmutex_t smbfs_minor_lock;
128 int smbfs_major;
129 int smbfs_minor;
130 /* See smbfs_node_findcreate() */
131 struct smbfattr smbfs_fattr0;
132
134 /*
135 * Local functions.
136 * SN for Smb Node
137 */
138 static void sn_rmtree(smbnode_t *);
139 static void sn_inactive(smbnode_t *);
140 static void sn_addhash_locked(smbnode_t *, avl_index_t);
141 static void sn_rmhlock(smbnode_t *);
142 static void sn_destroy_node(smbnode_t *);
143 void smbfs_kmem_reclaim(void *cdrarg);
144
145 static smbnode_t *
146 sn_hashfind(smbmntinfo_t *, const char *, int, avl_index_t *);
147
148 static smbnode_t *
149 make_smbnode(smbmntinfo_t *, const char *, int, int *);
150
151 /*
152 * Free the resources associated with an smbnode.
153 * Note: This is different from smbfs_inactive
154 */
155 /* From NFS: nfs_subr.c:rinactive
156 * NFS: nfs_subr.c:rinactive
157 */
158 static void
159 sn_inactive(smbnode_t *np)
160 {
161     vsecattr_t    ovsa;
162     cred_t        *oldcr;
163     char          *orpath;
164     int            orplen;
165     vnode_t       *vp;
166
167     /* Here NFS has:
168     * Flush and invalidate all pages (done by caller)
169     * Flush and invalidate all pages (todo)
170     * Free any held credentials and caches...
171     * etc. (See NFS code)
172     */
173     mutex_enter(&np->r_statelock);
174
175     ovsa = np->r_secattr;
176     np->r_secattr = smbfs_vsa0;
177     np->r_sectime = 0;
178
179     oldcr = np->r_cred;
180     np->r_cred = NULL;
181
182     orpath = np->n_rpath;
183     orplen = np->n_rplen;
184     np->n_rpath = NULL;
185     np->n_rplen = 0;
186
187     mutex_exit(&np->r_statelock);

```

```

188     vp = SMBTOV(np);
189     if (vn_has_cached_data(vp)) {
190         ASSERT3P(vp, ==, NULL);
191     }
192
193     if (ovsa.vsa_aclentp != NULL)
194         kmem_free(ovsa.vsa_aclentp, ovsa.vsa_aclentsz);
195
196     if (oldcr != NULL)
197         crfree(oldcr);
198
199     if (orpath != NULL)
200         kmem_free(orpath, orplen + 1);
201 }
202
203 /*
204 * Find and optionally create an smbnode for the passed
205 * mountinfo, directory, separator, and name. If the
206 * desired smbnode already exists, return a reference.
207 * If the file attributes pointer is non-null, the node
208 * is created if necessary and linked into the AVL tree.
209 *
210 * Callers that need a node created but don't have the
211 * real attributes pass smbfs_fattr0 to force creation.
212 *
213 * Note: make_smbnode() may upgrade the "hash" lock to exclusive.
214 */
215 /* Based on NFS: nfs_subr.c:makenfsnode
216 * NFS: nfs_subr.c:makenfsnode
217 */
218 smbnode_t *
219 smbfs_node_findcreate(
220     smbmntinfo_t *mi,
221     const char *dirnm,
222     int dirlen,
223     const char *name,
224     int nmlen,
225     char sep,
226     struct smbfattr *fap)
227 {
228     char tmpbuf[256];
229     size_t ralloc;
230     char *p, *rpath;
231     int rplen;
232     smbnode_t *np;
233     vnode_t *vp;
234     int newnode;
235
236     /*
237      * Build the search string, either in tmpbuf or
238      * in allocated memory if larger than tmpbuf.
239      */
240     rplen = dirlen;
241     if (sep != '\0')
242         rplen++;
243     rplen += nmlen;
244     if (rplen < sizeof (tmpbuf)) {
245         /* use tmpbuf */
246         ralloc = 0;
247         rpath = tmpbuf;
248     } else {
249         ralloc = rplen + 1;
250         rpath = kmem_alloc(ralloc, KM_SLEEP);
251     }
252     p = rpath;
253     bcopy(dirnm, p, dirlen);

```

```

253     p += dirlen;
254     if (sep != '\0')
255         *p++ = sep;
256     if (name != NULL) {
257         bcopy(name, p, nmlen);
258         p += nmlen;
259     }
260     ASSERT(p == rpath + rplen);

261     /*
262      * Find or create a node with this path.
263      */
264     rw_enter(&mi->smi_hash_lk, RW_READER);
265     if (fap == NULL)
266         np = sn_hashfind(mi, rpath, rplen, NULL);
267     else
268         np = make_smbnode(mi, rpath, rplen, &newnode);
269     rw_exit(&mi->smi_hash_lk);

270     if (rpalloc)
271         kmem_free(rpath, rpalloc);

272     if (fap == NULL) {
273         /*
274          * Caller is "just looking" (no create)
275          * so np may or may not be NULL here.
276          * Either way, we're done.
277          */
278         return (np);
279     }

280     /*
281      * We should have a node, possibly created.
282      * Do we have (real) attributes to apply?
283      */
284     ASSERT(np != NULL);
285     if (fap == &smbfs_fattr0)
286         return (np);

287     /*
288      * Apply the given attributes to this node,
289      * dealing with any cache impact, etc.
290      */
291     vp = SMBTOV(np);
292     if (!newnode) {
293         /*
294          * Found an existing node.
295          * Maybe purge caches...
296          */
297         smbfs_cache_check(vp, fap);
298     }
299     smbfs_attrcache_fa(vp, fap);

300     /*
301      * Note NFS sets vp->v_type here, assuming it
302      * can never change for the life of a node.
303      * We allow v_type to change, and set it in
304      * smbfs_attrcache(). Also: mode, uid, gid
305      */
306     return (np);
307 }

308 /**
309  * Here NFS has: nfs_subr.c:rtablehash
310  * NFS: nfs_subr.c:rtablehash
311  * We use smbfs_hash().

```

```

311 */

312 /*
313  * Find or create an smbnode.
314  * From NFS: nfs_subr.c:make_rnode
315  * NFS: nfs_subr.c:make_rnode
316  */
317 static smbnode_t *
318 make_smbnode(
319     smbmntinfo_t *mi,
320     const char *rpath,
321     int rplen,
322     int *newnode)
323 {
324     smbnode_t *np;
325     smbnode_t *tmp;
326     vnode_t *vp;
327     vfs_t *vfsp;
328     avl_index_t where;
329     char *new_rpath = NULL;

330     ASSERT(RW_READ_HELD(&mi->smi_hash_lk));
331     vfsp = mi->smi_vfsp;
332

333 start:
334     np = sn_hashfind(mi, rpath, rplen, NULL);
335     if (np != NULL) {
336         *newnode = 0;
337         return (np);
338     }

339     /*
340      * Note: will retake this lock below.
341      */
342     rw_exit(&mi->smi_hash_lk);

343     /*
344      * see if we can find something on the freelist
345      */
346     mutex_enter(&smbfreelist_lock);
347     if (smbfreelist != NULL && smbnodenew >= nsmbnode) {
348         np = smbfreelist;
349         sn_rmfree(np);
350         mutex_exit(&smbfreelist_lock);
351     }

352     vp = SMBTOV(np);

353     if (np->r_flags & RHASHED) {
354         smbmntinfo_t *tmp_mi = np->n_mount;
355         ASSERT(tmp_mi != NULL);
356         rw_enter(&tmp_mi->smi_hash_lk, RW_WRITER);
357         mutex_enter(&vp->v_lock);
358         if (vp->v_count > 1) {
359             VN_RELLE_LOCKED(vp);
360             mutex_exit(&vp->v_lock);
361             rw_exit(&tmp_mi->smi_hash_lk);
362             /* start over */
363             rw_enter(&mi->smi_hash_lk, RW_READER);
364             goto start;
365         }
366         mutex_exit(&vp->v_lock);
367         sn_rhash_locked(np);
368         rw_exit(&tmp_mi->smi_hash_lk);
369     }

370     sn_inactive(np);

371 }

372     mutex_enter(&vp->v_lock);

373 }

374 }

375 
```

```

376     if (vp->v_count > 1) {
377         VN_REL_E_LOCKED(vp);
378         mutex_exit(&vp->v_lock);
379         rw_enter(&mi->smi_hash_lk, RW_READER);
380         goto start;
381     }
382     mutex_exit(&vp->v_lock);
383     vn_invalidate(vp);
384     /*
385      * destroy old locks before bzero'ing and
386      * recreating the locks below.
387      */
388     smbfs_rw_destroy(&np->r_rwlock);
389     smbfs_rw_destroy(&np->r_lkserlock);
390     mutex_destroy(&np->r_statelock);
391     cv_destroy(&np->r_cv);
392     /*
393      * Make sure that if smbnode is recycled then
394      * VFS count is decremented properly before
395      * reuse.
396      */
397     VFS_REL_E(vp->v_vfsp);
398     vn_reinit(vp);
399 } else {
400     /*
401      * allocate and initialize a new smbnode
402      */
403     vnode_t *new_vp;

404     mutex_exit(&smbfreelist_lock);

405     np = kmem_cache_alloc(smbnode_cache, KM_SLEEP);
406     new_vp = vn_alloc(KM_SLEEP);

407     atomic_inc_ulong((ulong_t *)&smbnodenew);
408     vp = new_vp;
409 }

410 /*
411  * Allocate and copy the rpath we'll need below.
412  */
413 new_rpath = kmem_alloc(rplen + 1, KM_SLEEP);
414 bcopy(rpath, new_rpath, rplen);
415 new_rpath[rplen] = '\0';

416 /* Initialize smbnode_t */
417 bzero(np, sizeof (*np));

418 smbfs_rw_init(&np->r_rwlock, NULL, RW_DEFAULT, NULL);
419 smbfs_rw_init(&np->r_lkserlock, NULL, RW_DEFAULT, NULL);
420 mutex_init(&np->r_statelock, NULL, MUTEX_DEFAULT, NULL);
421 cv_init(&np->r_cv, NULL, CV_DEFAULT, NULL);
422 /* cv_init(&np->r_commit.c_cv, NULL, CV_DEFAULT, NULL); */

423 np->r_vnode = vp;
424 np->n_mount = mi;

425 np->n_fid = SMB_FID_UNUSED;
426 np->n_uid = mi->smi_uid;
427 np->n_gid = mi->smi_gid;
428 /* Leave attributes "stale." */

429 #if 0 /* XXX dircache */
430     /*
431      * Here NFS has avl_create(&np->r_dir, ...
432      * for the readdir cache (not used here).
433
434
435
436
437 #if 0 /* XXX dircache */
438     /*
439      * Here NFS has avl_create(&np->r_dir, ...
440      * for the readdir cache (not used here).

```

```

439     * We don't know if it's a directory yet.
440     * Let the caller do this? XXX
441     */
442     avl_create(&np->r_dir, compar, sizeof (raddir_cache),
443               offsetof(raddir_cache, tree));
444 #endif

445     /*
446      * Now fill in the vnode. */
447     vn_setops(vp, smbfs_vnodeops);
448     vp->v_data = (caddr_t)np;
449     VFS_HOLD(vfsp);
450     vp->v_vfsp = vfsp;
451     vp->v_type = VNON;

452     /*
453      * We entered with mi->smi_hash_lk held (reader).
454      * Retake it now, (as the writer).
455      * Will return with it held.
456      */
457     rw_enter(&mi->smi_hash_lk, RW_WRITER);

458     /*
459      * There is a race condition where someone else
460      * may alloc the smbnode while no locks are held,
461      * so check again and recover if found.
462      */
463     tnp = sn_hashfind(mi, rpath, rplen, &where);
464     if (tnp != NULL) {
465         /*
466          * Lost the race. Put the node we were building
467          * on the free list and return the one we found.
468          */
469     rw_exit(&mi->smi_hash_lk);
470     kmem_free(new_rpath, rplen + 1);
471     smbfs_addfree(np);
472     rw_enter(&mi->smi_hash_lk, RW_READER);
473     *newnode = 0;
474     return (tnp);
475 }

476     /*
477      * Hash search identifies nodes by the remote path
478      * (n_rpath) so fill that in now, before linking
479      * this node into the node cache (AVL tree).
480      */
481     np->n_rpath = new_rpath;
482     np->n_rplen = rplen;
483     np->n_ino = smbfs_gethash(new_rpath, rplen);

484     sn_addhash_locked(np, where);
485     *newnode = 1;
486     return (np);
487 }

488 */

489 /* smbfs_addfree
490  * Put an smbnode on the free list, or destroy it immediately
491  * if it offers no value were it to be reclaimed later. Also
492  * destroy immediately when we have too many smbnodes, etc.
493  *
494  * Normally called by smbfs_inactive, but also
495  * called in here during cleanup operations.
496  */
497 /* From NFS: nfs_subr.c:rp_addfree
498 */
499 /* NFS: nfs_subr.c:rp_addfree
500 */

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.c

```

501 void
502 smbefs_addfree(smbnode_t *np)
503 {
504     vnode_t *vp;
505     struct vfs *vfsp;
506     smbmntinfo_t *mi;
507
508     ASSERT(np->r_freef == NULL && np->r_freeb == NULL);
509
510     vp = SMBTOV(np);
511     ASSERT(vp->v_count >= 1);
512
513     vfsp = vp->v_vfsp;
514     mi = VFTOSMI(vfsp);
515
516     /*
517      * If there are no more references to this smbnode and:
518      * we have too many smbnodes allocated, or if the node
519      * is no longer accessible via the AVL tree (!RHASHED),
520      * or an i/o error occurred while writing to the file,
521      * or it's part of an unmounted FS, then try to destroy
522      * it instead of putting it on the smbnode freelist.
523      */
524     if (np->r_count == 0 && (
525         (np->r_flags & RHASHED) == 0 ||
526         (np->r_error != 0) ||
527         (vfsp->vfbs_flag & VFS_UNMOUNTED) ||
528         (smbnodefree > nsmbnode))) {
529
530         /* Try to destroy this node. */
531
532         if (np->r_flags & RHASHED) {
533             rw_enter(&mi->smi_hash_lk, RW_WRITER);
534             mutex_enter(&vp->v_lock);
535             if (vp->v_count > 1) {
536                 VN_RELSE_LOCKED(vp);
537                 mutex_exit(&vp->v_lock);
538                 rw_exit(&mi->smi_hash_lk);
539                 return;
540             }
541             /* Will get another call later,
542              * via smbfs_inactive.
543             */
544         }
545         mutex_exit(&vp->v_lock);
546         sn_rhash_locked(np);
547         rw_exit(&mi->smi_hash_lk);
548     }
549
550     sn_inactive(np);
551
552     /*
553      * Recheck the vnode reference count. We need to
554      * make sure that another reference has not been
555      * acquired while we were not holding v_lock. The
556      * smbnode is not in the smbnode "hash" AVL tree, so
557      * the only way for a reference to have been acquired
558      * is for a VOP_PUTPAGE because the smbnode was marked
559      * with RDIRTY or for a modified page. This vnode
560      * reference may have been acquired before our call
561      * to sn_inactive. The i/o may have been completed,
562      * thus allowing sn_inactive to complete, but the
563      * reference to the vnode may not have been released
564      * yet. In any case, the smbnode can not be destroyed
565      * until the other references to this vnode have been
566      * released. The other references will take care of

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.

```

567 * either destroying the smbnode or placing it on the
568 * smbnode freelist. If there are no other references,
569 * then the smbnode may be safely destroyed.
570 */
571 mutex_enter(&vp->v_lock);
572 if (vp->v_count > 1) {
573     VN_RELSE_LOCKED(vp);
574     mutex_exit(&vp->v_lock);
575     return;
576 }
577 mutex_exit(&vp->v_lock);

578 sn_destroy_node(np);
579 return;
580 }

581 }

582 /*
583 * Lock the AVL tree and then recheck the reference count
584 * to ensure that no other threads have acquired a reference
585 * to indicate that the smbnode should not be placed on the
586 * freelist. If another reference has been acquired, then
587 * just release this one and let the other thread complete
588 * the processing of adding this smbnode to the freelist.
589 */
590 rw_enter(&mi->smi_hash_lk, RW_WRITER);

591 mutex_enter(&vp->v_lock);
592 if (vp->v_count > 1) {
593     VN_RELSE_LOCKED(vp);
594     mutex_exit(&vp->v_lock);
595     rw_exit(&mi->smi_hash_lk);
596     return;
597 }
598 mutex_exit(&vp->v_lock);

599 }

600 mutex_enter(&vp->v_lock);

601 /*
602 * Put this node on the free list.
603 */
604 mutex_enter(&smbfreelist_lock);
605 if (smbfreelist == NULL) {
606     np->r_freef = np;
607     np->r_freeb = np;
608     smbfreelist = np;
609 } else {
610     np->r_freef = smbfreelist;
611     np->r_freeb = smbfreelist->r_freeb;
612     smbfreelist->r_freeb->r_freef = np;
613     smbfreelist->r_freeb = np;
614 }
615 mutex_exit(&smbfreelist_lock);

616 rw_exit(&mi->smi_hash_lk);
617 }

618 */

619 }

620 /*
621 * Remove an smbnode from the free list.
622 *
623 *
624 * The caller must be holding smbfreelist_lock and the smbnode
625 * must be on the freelist.
626 *
627 * From NFS: nfs_subr.c:rp_rmfree
628 * NFS: nfs_subr.c:rp_rmfree
629 */
630 static void
631 sn_rmfree(smbnode_t *np)
632 {

```

```

633     ASSERT(MUTEX_HELD(&smbfreelist_lock));
634     ASSERT(np->r_freef != NULL && np->r_freeb != NULL);
635
636     if (np == smbfreelist) {
637         smbfreelist = np->r_freef;
638         if (np == smbfreelist)
639             smbfreelist = NULL;
640     }
641
642     np->r_freeb->r_freef = np->r_freef;
643     np->r_freef->r_freeb = np->r_freeb;
644
645     np->r_freef = np->r_freeb = NULL;
646 }

648 /*
649 * Put an smbnode in the "hash" AVL tree.
650 *
651 * The caller must be hold the rwlock as writer.
652 */
653 /* From NFS: nfs_subr.c:rp_addhash
654 * NFS: nfs_subr.c:rp_addhash
655 */
656 static void
657 sn_addhash_locked(smbnode_t *np, avl_index_t where)
658 {
659     smbmntinfo_t *mi = np->n_mount;
660
661     ASSERT(RW_WRITE_HELD(&mi->smi_hash_lk));
662
663     mutex_enter(&np->r_statelock);
664     if ((np->r_flags & RHASHED) == 0) {
665         avl_insert(&mi->smi_hash_avl, np, where);
666         np->r_flags |= RHASHED;
667     }
668     mutex_exit(&np->r_statelock);

670 /*
671 * Remove an smbnode from the "hash" AVL tree.
672 *
673 * The caller must hold the rwlock as writer.
674 */
675 /* From NFS: nfs_subr.c:rp_rmhash_locked
676 * NFS: nfs_subr.c:rp_rmhash_locked
677 */
678 static void
679 sn_rmhash_locked(smbnode_t *np)
680 {
681     smbmntinfo_t *mi = np->n_mount;
682
683     ASSERT(RW_WRITE_HELD(&mi->smi_hash_lk));
684
685     mutex_enter(&np->r_statelock);
686     if ((np->r_flags & RHASHED) != 0) {
687         np->r_flags &= ~RHASHED;
688         avl_remove(&mi->smi_hash_avl, np);
689     }
690     mutex_exit(&np->r_statelock);

707 /*
708 * Lookup an smbnode by remote pathname
709 */

```

```

710     * The caller must be holding the AVL rwlock, either shared or exclusive.
711     *
712     * From NFS: nfs_subr.c:rfind
713     * NFS: nfs_subr.c:rfind
714     */
715     static smbnode_t *
716     sn_hashfind(
717         smbmntinfo_t *mi,
718         const char *rpath,
719         int rplen,
720         avl_index_t *pwhere) /* optional */
721     {
722         smbfss_node_hdr_t nhdr;
723         smbnode_t *np;
724         vnode_t *vp;

725         ASSERT(RW_LOCK_HELD(&mi->smi_hash_lk));
726
727         bzero(&nhdr, sizeof (nhdr));
728         nhdr.hdr_n_rpath = (char *)rpath;
729         nhdr.hdr_n_rplen = rplen;

730         /* See smbfss_node_cmp below. */
731         np = avl_find(&mi->smi_hash_avl, &nhdr, pwhere);
732
733         if (np == NULL)
734             return (NULL);

735         /*
736         * Found it in the "hash" AVL tree.
737         * Remove from free list, if necessary.
738         */
739         vp = SMBTOV(np);
740         if (np->r_freef != NULL) {
741             mutex_enter(&smbfreelist_lock);
742             /*
743             * If the smbnode is on the freelist,
744             * then remove it and use that reference
745             * as the new reference. Otherwise,
746             * need to increment the reference count.
747             */
748             if (np->r_freef != NULL) {
749                 sn_xmfree(np);
750                 mutex_exit(&smbfreelist_lock);
751             } else {
752                 mutex_exit(&smbfreelist_lock);
753                 VN_HOLD(vp);
754             }
755         } else
756             VN_HOLD(vp);

757         return (np);
758     }

759     unchanged_portion_omitted

851 #ifdef SMB_VNODE_DEBUG
852 int smbfss_check_table_debug = 1;
853 #else /* SMB_VNODE_DEBUG */
854 int smbfss_check_table_debug = 0;
855 #endif /* SMB_VNODE_DEBUG */

856 /*
857 * Return 1 if there is a active vnode belonging to this vfs in the
858 * smbnode cache.
859 */

```

```

862 * Several of these checks are done without holding the usual
863 * locks. This is safe because destroy_smbtable(), smbfs_addfree(),
864 * etc. will redo the necessary checks before actually destroying
865 * any smbnodes.
866 *
867 * From NFS: nfs_subr.c:check_rtable
868 * NFS: nfs_subr.c:check_rtable
869 * Debugging changes here relative to NFS.
870 * Relatively harmless, so left 'em in.
871 */
872 int
873 smbfs_check_table(struct vfs *vfsp, smbnode_t *rtnp)
874 {
875     smbmtinfo_t *mi;
876     smbnode_t *np;
877     vnode_t *vp;
878     int busycnt = 0;

880     mi = VFTOSMI(vfsp);
881     rw_enter(&mi->smi_hash_lk, RW_READER);
882     for (np = avl_first(&mi->smi_hash_avl); np != NULL;
883          np = avl_walk(&mi->smi_hash_avl, np, AVL_AFTER)) {

885         if (np == rtcp)
886             continue; /* skip the root */
887         vp = SMBTOV(np);

889         /* Now the 'busy' checks: */
890         /* Not on the free list? */
891         if (np->r_freef == NULL) {
892             SMBVDEBUG("!r_freef: node=0x%p, rpath=%s\n",
893                     (void *)np, np->n_rpath);
894             busycnt++;
895         }

897         /* Has dirty pages? */
898         if (vn_has_cached_data(vp) &&
899             (np->r_flags & RDIRTY)) {
900             SMBVDEBUG("is dirty: node=0x%p, rpath=%s\n",
901                     (void *)np, np->n_rpath);
902             busycnt++;
903         }

905         /* Other refs? (not reflected in v_count) */
906         if (np->r_count > 0) {
907             SMBVDEBUG("+r_count: node=0x%p, rpath=%s\n",
908                     (void *)np, np->n_rpath);
909             busycnt++;
910         }

912         if (busycnt && !smbfs_check_table_debug)
913             break;
914     }
915     rw_exit(&mi->smi_hash_lk);
916
917     return (busycnt);
918
919 }

921 */
922 * Destroy inactive vnodes from the AVL tree which belong to this
923 * vfs. It is essential that we destroy all inactive vnodes during a
924 * forced unmount as well as during a normal unmount.
925 *
926 * Based on NFS: nfs_subr.c:destroy_rtable

```

```

929 * NFS: nfs_subr.c:destroy_rtable
930 *
931 * In here, we're normally destroying all or most of the AVL tree,
932 * so the natural choice is to use avl_destroy_nodes. However,
933 * there may be a few busy nodes that should remain in the AVL
934 * tree when we're done. The solution: use a temporary tree to
935 * hold the busy nodes until we're done destroying the old tree,
936 * then copy the temporary tree over the (now empty) real tree.
937 */
938 void
939 smbfs_destroy_table(struct vfs *vfsp)
940 {
941     avl_tree_t tmp_avl;
942     smbmtinfo_t *mi;
943     smbnode_t *np;
944     smbnode_t *rlist;
945     void *v;

946     mi = VFTOSMI(vfsp);
947     rlist = NULL;
948     smbfs_init_hash_avl(&tmp_avl);

949     rw_enter(&mi->smi_hash_lk, RW_WRITER);
950     v = NULL;
951     while ((np = avl_destroy_nodes(&mi->smi_hash_avl, &v)) != NULL) {

952         mutex_enter(&smbfreelist_lock);
953         if (np->r_freef == NULL) {
954             /*
955              * Busy node (not on the free list).
956              * Will keep in the final AVL tree.
957              */
958             mutex_exit(&smbfreelist_lock);
959             avl_add(&tmp_avl, np);
960         } else {
961             /*
962              * It's on the free list. Remove and
963              * arrange for it to be destroyed.
964              */
965             sn_rmtree(np);
966             mutex_exit(&smbfreelist_lock);

967             /*
968              * Last part of sn_rhash_locked().
969              * NB: avl_destroy_nodes has already
970              * removed this from the "hash" AVL.
971              */
972             mutex_enter(&np->r_statelock);
973             np->r_flags |= ~RHASHED;
974             mutex_exit(&np->r_statelock);

975             /*
976              * Add to the list of nodes to destroy.
977              * Borrowing avl_child[0] for this list.
978              */
979             np->r_avl_node.avl_child[0] =
980                 (struct avl_node *)rlist;
981             rlist = np;
982         }
983     }
984     avl_destroy(&mi->smi_hash_avl);

985     /*
986      * Replace the (now destroyed) "hash" AVL with the
987      * temporary AVL, which restores the busy nodes.
988      */
989 
```

```

992     mi->smi_hash_avl = tmp_avl;
993     rw_exit(&mi->smi_hash_lk);

995     /*
996      * Now destroy the nodes on our temporary list (rlist).
997      * This call to smbfs_addfree will end up destroying the
998      * smbnodes, but in a safe way with the appropriate set
999      * of checks done.
1000     */
1001    while ((np = rlist) != NULL) {
1002        rlist = (smbnode_t *)np->r_avl_node.avl_child[0];
1003        smbfs_addfree(np);
1004    }
1005 }

1007 /**
1008  * This routine destroys all the resources associated with the smbnodes
1009  * and then the smbnodes itself. Note: sn_inactive has been called.
1010 */
1011 /* From NFS: nfs_subr.c:destroy_rnode
1012  * NFS: nfs_subr.c:destroy_rnode
1013 */
1014 static void
1015 sn_destroy_node(smbnode_t *np)
1016 {
1017     vnode_t *vp;
1018     vfs_t *vfsp;

1019     vp = SMBTOV(np);
1020     vfsp = vp->v_vfsp;

1022     ASSERT(vp->v_count == 1);
1023     ASSERT(np->r_count == 0);
1024     ASSERT(np->r_mapcnt == 0);
1025     ASSERT(np->r_secattr.vsa_aclentp == NULL);
1026     ASSERT(np->r_cred == NULL);
1027     ASSERT(np->r_path == NULL);
1028     ASSERT(!(np->r_flags & RHASHED));
1029     ASSERT(np->r_freef == NULL && np->r_freeb == NULL);
1030     atomic_dec_ulong((ulong_t *)&smbnodecache);
1031     vn_invalidate(vp);
1032     vn_free(vp);
1033     kmem_cache_free(smbnode_cache, np);
1034     VFS_RELEASE(vfsp);
1035 }

1037 /**
1038  * From NFS rflush()
1039  * Flush all vnodes in this (or every) vfs.
1040  * Used by smbfs_sync and by smbfs_unmount.
1041  */
1042 /*ARGSUSED*/
1043 void
1044 smbfs_rflush(struct vfs *vfsp, cred_t *cr)
1045 {
1046     smbmtinfo_t *mi;
1047     smbnodes_t *np;
1048     vnode_t *vp, **vplist;
1049     long num, cnt;

1051     mi = VFTOSMI(vfsp);

1053     /*
1054      * Check to see whether there is anything to do.
1055     */

```

```

1056     num = avl_numnodes(&mi->smi_hash_avl);
1057     if (num == 0)
1058         return;

1060     /*
1061      * Allocate a slot for all currently active rnodes on the
1062      * supposition that they all may need flushing.
1063      */
1064     vplist = kmalloc(num * sizeof (*vplist), KM_SLEEP);
1065     cnt = 0;

1067     /*
1068      * Walk the AVL tree looking for rnodes with page
1069      * lists associated with them. Make a list of these
1070      * files.
1071      */
1072     rw_enter(&mi->smi_hash_lk, RW_READER);
1073     for (np = avl_first(&mi->smi_hash_avl); np != NULL;
1074          np = avl_walk(&mi->smi_hash_avl, np, AVL_AFTER)) {
1075         vp = SMBTOV(np);
1076         /*
1077          * Don't bother sync'ing a vp if it
1078          * is part of virtual swap device or
1079          * if VFS is read-only
1080          */
1081         if (IS_SWAPVP(vp) || vn_is_readonly(vp))
1082             continue;
1083         /*
1084          * If the vnode has pages and is marked as either
1085          * dirty or mmap'd, hold and add this vnode to the
1086          * list of vnodes to flush.
1087          */
1088         if (vn_has_cached_data(vp) &&
1089             ((np->r_flags & RDIRTY) || np->r_mapcnt > 0)) {
1090             VN_HOLD(vp);
1091             vplist[cnt++] = vp;
1092             if (cnt == num)
1093                 break;
1094         }
1095     }
1096     rw_exit(&mi->smi_hash_lk);

1098     /*
1099      * Flush and release all of the files on the list.
1100      */
1101     while (cnt-- > 0) {
1102         vp = vplist[cnt];
1103         (void) VOP_PUTPAGE(vp, (u_offset_t)0, 0, B_ASYNC, cr, NULL);
1104         VN_RELEASE(vp);
1105     }

1107     kmalloc_free(vplist, num * sizeof (vnode_t *));
1108 }

1110 /* Here NFS has access cache stuff (nfs_subr.c) not used here */
1111 /* access cache (nfs_subr.c) not used here */

1112 /*
1113  * Set or Clear direct I/O flag
1114  * VOP_RWLOCK() is held for write access to prevent a race condition
1115  * which would occur if a process is in the middle of a write when
1116  * directio flag gets set. It is possible that all pages may not get flushed.
1117  * From nfs_common.c
1118 */

```

```

1120 /* ARGSUSED */
1121 int
1122 smbfs_directio(vnode_t *vp, int cmd, cred_t *cr)
1123 {
1124     int     error = 0;
1125     smbnode_t      *np;
1126
1127     np = VTOSMB(vp);
1128
1129     if (cmd == DIRECTIO_ON) {
1130         if (np->r_flags & RDIRECTIO)
1131             return (0);
1132
1133         /*
1134         * Flush the page cache.
1135         */
1136
1137         (void) VOP_RWLOCK(vp, V_WRITELOCK_TRUE, NULL);
1138
1139     if (np->r_flags & RDIRECTIO) {
1140         VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, NULL);
1141         return (0);
1142     }
1143
1144     /* Here NFS also checks ->r_awcount */
1145     if (vn_has_cached_data(vp) &&
1146         (np->r_flags & RDIRTY) != 0) {
1147         error = VOP_PUTPAGE(vp, (offset_t)0, (uint_t)0,
1148                             B_INVAL, cr, NULL);
1149
1150     if (error) {
1151         if (error == ENOSPC || error == EDQUOT) {
1152             mutex_enter(&np->r_statelock);
1153             if (!np->r_error)
1154                 np->r_error = error;
1155             mutex_exit(&np->r_statelock);
1156         }
1157         VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, NULL);
1158         return (error);
1159     }
1160
1161     mutex_enter(&np->r_statelock);
1162     np->r_flags |= RDIRECTIO;
1163     mutex_exit(&np->r_statelock);
1164     VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, NULL);
1165     return (0);
1166 }
1167
1168 if (cmd == DIRECTIO_OFF) {
1169     mutex_enter(&np->r_statelock);
1170     np->r_flags &= ~RDIRECTIO; /* disable direct mode */
1171     mutex_exit(&np->r_statelock);
1172     return (0);
1173 }
1174
1175     return (EINVAL);
1176 }
1177
1178 static kmutex_t smbfs_newnum_lock;
1179 static uint32_t smbfs_newnum_val = 0;
1180
1181 /*
1182 * Return a number 0..0xffffffff that's different from the last
1183 * 0xffffffff numbers this returned. Used for unlinked files.
1184 * From NFS nfs_subr.c newnum
1185 */

```

```

1186 */
1187 uint32_t
1188 smbfs_newnum(void)
1189 {
1190     uint32_t id;
1191
1192     mutex_enter(&smbfs_newnum_lock);
1193     if (smbfs_newnum_val == 0)
1194         smbfs_newnum_val = (uint32_t)gethrrestime_sec();
1195     id = smbfs_newnum_val++;
1196     mutex_exit(&smbfs_newnum_lock);
1197     return (id);
1198 }
1199
1200
1201
1202
1203
1204
1205 /*
1206 * initialize resources that are used by smbfs_subr.c
1207 * this is called from the _init() routine (by the way of smbfs_clntinit())
1208 *
1209 * From NFS: nfs_subr.c:nfs_subrinit
1210 * NFS: nfs_subr.c:nfs_subrinit
1211 */
1212 int
1213 smbfs_subrinit(void)
1214 {
1215     ulong_t nsmbnode_max;
1216
1217     /*
1218     * Allocate and initialize the smbnodes cache
1219     */
1220     if (nsmbnode <= 0)
1221         nsmbnode = ncsize; /* dnlc.h */
1222     nsmbnode_max = (ulong_t)((kmem_maxavail() >> 2) /
1223                             sizeof (struct smbnodes));
1224     if (nsmbnode > nsmbnode_max || (nsmbnode == 0 && ncsize == 0)) {
1225         zcmn_err(GLOBAL_ZONEID, CE_NOTE,
1226                  "setting nsmbnode to max value of %ld", nsmbnode_max);
1227         nsmbnode = nsmbnode_max;
1228     }
1229
1230     smbnodes_cache = kmem_cache_create("smbnodes_cache", sizeof (smbnodes_t),
1231                                       0, NULL, NULL, smbfs_kmem_reclaim, NULL, 0);
1232
1233     /*
1234     * Initialize the various mutexes and reader/writer locks
1235     */
1236     mutex_init(&smbfreelist_lock, NULL, MUTEX_DEFAULT, NULL);
1237     mutex_init(&smbfs_minor_lock, NULL, MUTEX_DEFAULT, NULL);
1238
1239     /*
1240     * Assign unique major number for all smbfs mounts
1241     */
1242     if ((smbfs_major = getudev()) == -1) {
1243         zcmn_err(GLOBAL_ZONEID, CE_WARN,
1244                  "smbfs: init: can't get unique device number");
1245         smbfs_major = 0;
1246     }
1247     smbfs_minor = 0;
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261 */
1262     /* free smbfs hash table, etc.

```

```
1263 * From NFS: nfs_subr.c:nfs_subrfini
1137 * NFS: nfs_subr.c:nfs_subrfini
1264 */
1265 void
1266 smbfs_subrfini(void)
1267 {
1268     /*
1269      * Destroy the smbnode cache
1270      */
1271     kmem_cache_destroy(smbnode_cache);
1272
1273     /*
1274      * Destroy the various mutexes and reader/writer locks
1275      */
1276     mutex_destroy(&smbfreelist_lock);
1277     mutex_destroy(&smbfs_minor_lock);
1278 }
____ unchanged_portion_omitted_
1338 /*
1339  * Here NFS has failover stuff and
1340  * nfs_rw_xxx - see smbfs_rwlock.c
1341  */
1212 /* nfs failover stuff */
1213 /* nfs_rw_xxx - see smbfs_rwlock.c */
```

```
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vfsops.c          1
*****
25639 Sun Mar  4 06:09:09 2018
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vfsops.c
5404 smbfs needs mmap support
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>
*****
_____ unchanged_portion_omitted_


884 static kmutex_t smbfs_syncbusy;

884 /*
885  * Flush dirty smbfs files for file system vfsp.
886  * If vfsp == NULL, all smbfs files are flushed.
887  */
888 /*ARGSUSED*/
889 static int
890 smbfs_sync(vfs_t *vfsp, short flag, cred_t *cr)
891 {
893     /*
894      * SYNC_ATTR is used by fsflush() to force old filesystems like UFS
895      * to sync metadata, which they would otherwise cache indefinitely.
896      * Semantically, the only requirement is that the sync be initiated.
897      * Assume the server-side takes care of attribute sync.
898      * Cross-zone calls are OK here, since this translates to a
899      * VOP_PUTPAGE(B_ASYNC), which gets picked up by the right zone.
900      */
901     if (flag & SYNC_ATTR)
902         return (0);
903
904     if (vfsp == NULL) {
905         /*
906          * Flush ALL smbfs mounts in this zone.
907          */
908         smbfs_flushall(cr);
909         return (0);
910     if (!!(flag & SYNC_ATTR) && mutex_tryenter(&smbfs_syncbusy) != 0) {
911         smbfs_rflush(vfsp, cr);
912         mutex_exit(&smbfs_syncbusy);
913     }
914
915     /*
916      * Initialization routine for VFS routines.  Should only be called once
917      */
918     int
919     smbfs_vfsinit(void)
920     {
921         mutex_init(&smbfs_syncbusy, NULL, MUTEX_DEFAULT, NULL);
922         return (0);
923     }
924
925     /*
926      * Shutdown routine for VFS routines.  Should only be called once
927      */
928     void
929     smbfs_vfsfini(void)
930     {
931         mutex_destroy(&smbfs_syncbusy);
932     }
_____ unchanged_portion_omitted_
```

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c          1
*****
123699 Sun Mar  4 06:09:10 2018
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c
cstyle
Fix missing logic for ESTALE and NFS_EOF
Implement ioctl _FIODIRECTIO
Kill flags arg in smbfs_purge_caches
Lots of comment cleanup
5404 smbfs needs mmap support
Portions contributed by: Gordon Ross <gordon.w.ross@gmail.com>
*****
1 /*
2  * Copyright (c) 2000-2001 Boris Popov
3  * All rights reserved.
4  *
5  * Redistribution and use in source and binary forms, with or without
6  * modification, are permitted provided that the following conditions
7  * are met:
8  * 1. Redistributions of source code must retain the above copyright
9  * notice, this list of conditions and the following disclaimer.
10 * 2. Redistributions in binary form must reproduce the above copyright
11 * notice, this list of conditions and the following disclaimer in the
12 * documentation and/or other materials provided with the distribution.
13 * 3. All advertising materials mentioning features or use of this software
14 * must display the following acknowledgement:
15 *   This product includes software developed by Boris Popov.
16 * 4. Neither the name of the author nor the names of any co-contributors
17 * may be used to endorse or promote products derived from this software
18 * without specific prior written permission.
19 *
20 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
21 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
22 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
23 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
24 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
25 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
26 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
27 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
28 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
29 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
30 * SUCH DAMAGE.
31 *
32 * $Id: smbfs_vnops.c,v 1.128.36.1 2005/05/27 02:35:28 lindak Exp $
33 */
34
35 /*
36 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
37 */
38
39 /*
40 * Vnode operations
41 *
42 * This file is similar to nfs3_vnops.c
43 */
44
45 #include <sys/param.h>
46 #include <sys/sysctl.h>
47 #include <sys/cred.h>
48 #include <sys/vnode.h>
49 #include <sys/vfs.h>
50 #include <sys/filio.h>
51 #include <sys/uio.h>
52 #include <sys/dirent.h>
53 #include <sys/errno.h>
54 #include <sys/sunddi.h>
55 #include <sys/sysmacros.h>

```

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c          2
*****
56 #include <sys/kmem.h>
57 #include <sys/cmn_err.h>
58 #include <sys/vfs_opreg.h>
59 #include <sys/policy.h>
60 #include <sys/sdt.h>
61 #include <sys/zone.h>
62 #include <sys/vmsystm.h>
63
64 #include <vm/hat.h>
65 #include <vm/as.h>
66 #include <vm/page.h>
67 #include <vm/pvn.h>
68 #include <vm/seg.h>
69 #include <vm/seg_map.h>
70 #include <vm/seg_kpm.h>
71 #include <vm/seg_vn.h>
72
73 #include <netsmb/smb_osdep.h>
74 #include <netsmb/smb.h>
75 #include <netsmb/smb_conn.h>
76 #include <netsmb/smb_subr.h>
77
78 #include <smbfs/smbfs.h>
79 #include <smbfs/smbfs_node.h>
80 #include <smbfs/smbfs_subr.h>
81
82 #include <sys/fs/smbfs_ioctl.h>
83 #include <fs/fs_subr.h>
84
85 /*
86 * We assign directory offsets like the NFS client, where the
87 * offset increments by _one_ after each directory entry.
88 * Further, the entries "." and ".." are always at offsets
89 * zero and one (respectively) and the "real" entries from
90 * the server appear at offsets starting with two. This
91 * macro is used to initialize the n_dirofs field after
92 * setting n_dirseq with a _findopen call.
93 */
94 #define FIRST_DIROFS      2
95
96 /*
97 * These characters are illegal in NTFS file names.
98 * ref: http://support.microsoft.com/kb/147438
99 */
100 * Careful! The check in the XATTR case skips the
101 * first character to allow colon in XATTR names.
102 */
103 static const char illegal_chars[] = {
104     ':', /* colon - keep this first! */
105     '\\', /* back slash */
106     '/', /* slash */
107     '*', /* asterisk */
108     '?', /* question mark */
109     '\"', /* double quote */
110     '<', /* less than sign */
111     '>', /* greater than sign */
112     '|', /* vertical bar */
113     0
114 };
115
116 /*
117 * Turning this on causes nodes to be created in the cache
118 * during directory listings, normally avoiding a second
119 * OtW attribute fetch just after a readdir.
120 */
121 int smbfs_fastlookup = 1;

```

```

123 struct vnodeops *smbfs_vnodeops = NULL;
125 /* local static function defines */
127 static int smbfslookup_cache(vnode_t *, char *, int, vnode_t **,
128     cred_t *);
129 static int smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
130     int cache_ok, caller_context_t *);
131 static int smbfsremove(vnode_t *dvp, vnode_t *vp, struct smb_cred *scred,
132     int flags);
133 static int smbfsrename(vnode_t *odvp, vnode_t *ovp, vnode_t *ndvp,
134     char *nm, struct smb_cred *scred, int flags);
135 static int smbfssetattr(vnode_t *, struct vattr *, int, cred_t *);
136 static int smbfs_accessx(void *, int, cred_t *);
137 static int smbfs_readdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
138     caller_context_t *);
139 static void smbfs_rele_fid(smbnode_t *, struct smb_cred *);
140 static uint32_t xvattr_to_dosattr(smbnode_t *, struct vattr *);

142 static int smbfs_rdwrlnb(vnode_t *, page_t *, u_offset_t, size_t, int,
143     cred_t *);
144 static int smbfs_bio(struct buf *, int, cred_t *);
145 static int smbfs_writenp(smbnode_t *np, caddr_t base, int tcount,
146     struct uio *uiop, int pgcreated);

148 static int smbfs_fsync(vnode_t *, int, cred_t *, caller_context_t *);
149 static int smbfs_putpage(vnode_t *, offset_t, size_t, int, cred_t *,
150     caller_context_t *);
151 static int smbfs_getapage(vnode_t *, u_offset_t, size_t, uint_t *,
152     page_t *[], size_t, struct seg *, caddr_t,
153     enum seg_rw, cred_t *);
154 static int smbfs_putapage(vnode_t *, page_t *, u_offset_t *, size_t *,
155     int, cred_t *);
156 static void smbfs_delmap_callback(struct as *, void *, uint_t);

158 /*
159  * Error flags used to pass information about certain special errors
160  * which need to be handled specially.
161  */
162 #define SMBFS_EOF -98
164 /* When implementing Otw locks, make this a real function. */
165 #define smbfs_lm_has_sleep(vp) 0

167 /*
168  * These are the vnode ops routines which implement the vnode interface to
169  * the networked file system. These routines just take their parameters,
170  * make them look networkish by putting the right info into interface structs,
171  * and then calling the appropriate remote routine(s) to do the work.
172  */
173 /* Note on directory name lookup cacheing: If we detect a stale fhandle,
174  * we purge the directory cache relative to that vnode. This way, the
175  * user won't get burned by the cache repeatedly. See <smbfs/smbnode.h> for
176  * more details on smbnode locking.
177 */

133 static int smbfs_open(vnode_t **, int, cred_t *, caller_context_t *);
134 static int smbfs_close(vnode_t *, int, int, offset_t, cred_t *,
135     caller_context_t *);
136 static int smbfs_read(vnode_t *, struct uio *, int, cred_t *,
137     caller_context_t *);
138 static int smbfs_write(vnode_t *, struct uio *, int, cred_t *,
139     caller_context_t *);
140 static int smbfs_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
141     caller_context_t *);

```

```

142 static int smbfs_getattr(vnode_t *, struct vattr *, int, cred_t *,
143     caller_context_t *);
144 static int smbfs_setattr(vnode_t *, struct vattr *, int, cred_t *,
145     caller_context_t *);
146 static int smbfs_access(vnode_t *, int, int, cred_t *, caller_context_t *);
147 static int smbfs_fsync(vnode_t *, int, cred_t *, caller_context_t *);
148 static void smbfs_inactive(vnode_t *, cred_t *, caller_context_t *);
149 static int smbfs_lookup(vnode_t *, char *, vnode_t **, struct pathname *,
150     int, vnode_t *, cred_t *, caller_context_t *,
151     int *, pathname_t *);
152 static int smbfs_create(vnode_t *, char *, struct vattr *, enum vcexcl,
153     int, vnode_t **, cred_t *, int, caller_context_t *,
154     vsecattr_t *);
155 static int smbfs_remove(vnode_t *, char *, cred_t *, caller_context_t *,
156     int);
157 static int smbfs_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
158     caller_context_t *, int);
159 static int smbfs_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
160     cred_t *, caller_context_t *, int, vsecattr_t *);
161 static int smbfs_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
162     caller_context_t *, int);
163 static int smbfs_readdir(vnode_t *, struct uio *, cred_t *, int *,
164     caller_context_t *, int);
165 static int smbfs_rwlock(vnode_t *, int, caller_context_t *);
166 static void smbfs_rwunlock(vnode_t *, int, caller_context_t *);
167 static int smbfs_seek(vnode_t *, offset_t, offset_t, caller_context_t *);
168 static int smbfs_frllock(vnode_t *, int, struct flock64 *, int, offset_t,
169     struct flk_callback *, cred_t *, caller_context_t *);
170 static int smbfs_space(vnode_t *, int, struct flock64 *, int, offset_t,
171     cred_t *, caller_context_t *);
172 static int smbfs_pathconf(vnode_t *, int, ulong_t *, cred_t *,
173     caller_context_t *);
174 static int smbfs_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
175     caller_context_t *);
176 static int smbfs_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
177     caller_context_t *);
178 static int smbfs_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
179     caller_context_t *);

181 /* Dummy function to use until correct function is ported in */
182 int noop_vnodeop() {
183     return (0);
184 }

186 struct vnodeops *smbfs_vnodeops = NULL;

187 /*
188  * Most unimplemented ops will return ENOSYS because of fs_nosys().
189  * The only ops where that won't work are ACCESS (due to open(2)
190  * failures) and ... (anything else left?)
191  */
192 const fs_operation_def_t smbfs_vnodeops_template[] = {
193     { VOPNAME_OPEN, { .vop_open = smbfs_open } },
194     { VOPNAME_CLOSE, { .vop_close = smbfs_close } },
195     { VOPNAME_READ, { .vop_read = smbfs_read } },
196     { VOPNAME_WRITE, { .vop_write = smbfs_write } },
197     { VOPNAME_IOCTL, { .vop_ioctl = smbfs_ioctl } },
198     { VOPNAME_GETATTR, { .vop_getattr = smbfs_getattr } },
199     { VOPNAME_SETATTR, { .vop_setattr = smbfs_setattr } },
200     { VOPNAME_ACCESS, { .vop_access = smbfs_access } },
201     { VOPNAME_LOOKUP, { .vop_lookup = smbfs_lookup } },
202     { VOPNAME_CREATE, { .vop_create = smbfs_create } },
203     { VOPNAME_REMOVE, { .vop_remove = smbfs_remove } },
204     { VOPNAME_LINK, { .error = fs_nosys } }, /* smbfs_link, */
205     { VOPNAME_RENAME, { .vop_rename = smbfs_rename } },
206     { VOPNAME_MKDIR, { .vop_mkdir = smbfs_mkdir } },
207 }
```

```

208     { VOPNAME_RMDIR,           { .vop_rmdir = smbfs_rmdir } },
209     { VOPNAME_READDIR,        { .vop_readdir = smbfs_readdir } },
210     { VOPNAME_SYMLINK,        { .error = fs_nosys } }, /* smbfs_symlink, */
211     { VOPNAME_READLINK,        { .error = fs_nosys } }, /* smbfs_readlink, */
212     { VOPNAME_FSYNC,           { .vop_fsync = smbfs_fsync } },
213     { VOPNAME_INACTIVE,        { .vop_inactive = smbfs_inactive } },
214     { VOPNAME_FID,             { .error = fs_nosys } }, /* smbfs_fid, */
215     { VOPNAME_RWLOCK,          { .vop_rwlock = smbfs_rwlock } },
216     { VOPNAME_RWUNLOCK,        { .vop_rwunlock = smbfs_rwunlock } },
217     { VOPNAME_SEEK,            { .vop_seek = smbfs_seek } },
218     { VOPNAME_FRLOCK,          { .vop_frlock = smbfs_frlock } },
219     { VOPNAME_SPACE,           { .vop_space = smbfs_space } },
220     { VOPNAME_REALVP,          { .error = fs_nosys } }, /* smbfs_realvp, */
221     { VOPNAME_GETPAGE,          { .error = fs_nosys } }, /* smbfs_getpage, */
222     { VOPNAME_PUTPAGE,          { .error = fs_nosys } }, /* smbfs_putpage, */
223     { VOPNAME_MAP,              { .error = fs_nosys } }, /* smbfs_map, */
224     { VOPNAME_ADDMAP,           { .error = fs_nosys } }, /* smbfs_addmap, */
225     { VOPNAME_DELMAP,           { .error = fs_nosys } }, /* smbfs_delmap, */
226     { VOPNAME_DUMP,              { .error = fs_nosys } }, /* smbfs_dump, */
227     { VOPNAME_PATHCONF,         { .vop_pathconf = smbfs_pathconf } },
228     { VOPNAME_PAGEIO,           { .error = fs_nosys } }, /* smbfs_pageio, */
229     { VOPNAME_SETSECAATTR,       { .vop_setsecattr = smbfs_setsecattr } },
230     { VOPNAME_GETSECAATTR,       { .vop_getsecattr = smbfs_getsecattr } },
231     { VOPNAME_SHRLOCK,           { .vop_shrlock = smbfs_shrlock } },
232     { NULL, NULL }
233 };

235 /*
181 * XXX
182 * When new and relevant functionality is enabled, we should be
183 * calling vfs_set_feature() to inform callers that pieces of
184 * functionality are available, per PSARC 2007/227.
185 */
186 /* ARGSUSED */
187 static int
188 smbfs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
189 {
190     smbnode_t      *np;
191     vnode_t         *vp;
192     smbattr_t       fa;
193     u_int32_t       rights, rightsrcvd;
194     u_int16_t       fid, oldfid;
195     int             oldgenid;
196     struct smb_cred scred;
197     smbmtinfo_t    *smi;
198     smb_share_t    *ssp;
199     cred_t          *oldcr;
200     int             tmperror;
201     int             error = 0;

203     vp = *vpp;
204     np = VTOSMB(vp);
205     smi = VTOSMI(vp);
206     ssp = smi->smi_share;

208     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
209         return (EIO);

211     if (smi->smi_flags & SMI_DEAD || vp->vfsp->vfs_flag & VFS_UNMOUNTED)
212         return (EIO);

214     if (vp->v_type != VREG && vp->v_type != VDIR) { /* XXX VLNK? */
215         SMBVDEBUG("open eacces vtype=%d\n", vp->v_type);
216         return (EACCES);
217     }

```

```

219     /*
220      * Get exclusive access to n_fid and related stuff.
221      * No returns after this until out.
222      */
223     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
224         return (EINTR);
225     smb_credinit(&scred, cr);

227     /*
228      * Keep track of the vnode type at first open.
229      * It may change later, and we need close to do
230      * cleanup for the type we opened. Also deny
231      * open of new types until old type is closed.
232      * XXX: Per-open instance nodes whould help.
233      */
234     if (np->n_ovtype == VNON) {
235         ASSERT(np->n_dirrefs == 0);
236         ASSERT(np->n_fidrefs == 0);
237     } else if (np->n_ovtype != vp->v_type) {
238         SMBVDEBUG("open n_ovtype=%d v_type=%d\n",
239                  np->n_ovtype, vp->v_type);
240         error = EACCES;
241         goto out;
242     }

243     /*
244      * Directory open. See smbfs_readvdir()
245      */
246     if (vp->v_type == VDIR) {
247         if (np->n_dirseq == NULL) {
248             /* first open */
249             error = smbfs_smb_fndopen(np, "", 1,
250                                       SMB_FA_SYSTEM | SMB_FA_HIDDEN | SMB_FA_DIR,
251                                       &scred, &np->n_dirseq);
252             if (error != 0)
253                 goto out;
254         }
255         np->n_dirofs = FIRST_DIROFS;
256         np->n_dirrefs++;
257         goto have_fid;
258     }

260     /*
261      * If caller specified O_TRUNC/FTRUNC, then be sure to set
262      * FWRITE (to drive successful setattr(size=0) after open)
263      */
264     if (flag & FTRUNC)
265         flag |= FWRITE;

267     /*
268      * If we already have it open, and the FID is still valid,
269      * check whether the rights are sufficient for FID reuse.
270      */
271     if (np->n_fidrefs > 0 &&
272         np->n_vcgenid == ssp->ss_vcgenid) {
273         int upgrade = 0;

275         if ((flag & FWRITE) &&
276             !(np->n_rights & SA_RIGHT_FILE_WRITE_DATA))
277             upgrade = 1;
278         if ((flag & FREAD) &&
279             !(np->n_rights & SA_RIGHT_FILE_READ_DATA))
280             upgrade = 1;
281         if (!upgrade) {
282             /*
283              * the existing open is good enough

```

```

284             */
285             np->n_fidrefs++;
286             goto have_fid;
287         }
288     }
289     rights = np->n_fidrefs ? np->n_rights : 0;
290
291     /*
292      * we always ask for READ_CONTROL so we can always get the
293      * owner/group IDs to satisfy a stat. Ditto attributes.
294      */
295     rights |= (STD_RIGHT_READ_CONTROL_ACCESS |
296                 SA_RIGHT_FILE_READ_ATTRIBUTES);
297     if ((flag & FREAD)
298         rights |= SA_RIGHT_FILE_READ_DATA;
299     if ((flag & FWRITE))
300         rights |= SA_RIGHT_FILE_WRITE_DATA |
301                 SA_RIGHT_FILE_APPEND_DATA |
302                 SA_RIGHT_FILE_WRITE_ATTRIBUTES;
303
304     bzero(&fa, sizeof (fa));
305     error = smbfs_smb_open(np,
306                           NULL, 0, /* name nmllen xattr */
307                           rights, &scred,
308                           &fid, &rightsrvd, &fa);
309     if (error)
310         goto out;
311     smbfs_attrcache_fa(vp, &fa);
312
313     /*
314      * We have a new FID and access rights.
315      */
316     oldfid = np->n_fid;
317     oldgenid = np->n_vcgenid;
318     np->n_fid = fid;
319     np->n_vcgenid = ssp->ss_vcgenid;
320     np->n_rights = rightsrvd;
321     np->n_fidrefs++;
322     if (np->n_fidrefs > 1 &&
323         oldgenid == ssp->ss_vcgenid) {
324         /*
325          * We already had it open (presumably because
326          * it was open with insufficient rights.)
327          * Close old wire-open.
328          */
329         tmperror = smbfs_smb_close(ssp,
330                                   oldfid, NULL, &scred);
331         if (tmperror)
332             SMBVDEBUG("error %d closing %s\n",
333                     tmperror, np->n_rpath);
334     }
335
336     /*
337      * This thread did the open.
338      * Save our credentials too.
339      */
340     mutex_enter(&np->r_statelock);
341     oldcr = np->r_cred;
342     np->r_cred = cr;
343     crhold(cr);
344     if (oldcr)
345         crfree(oldcr);
346     mutex_exit(&np->r_statelock);
347
348 have_fid:
349     /*

```

```

350             * Keep track of the vnode type at first open.
351             * (see comments above)
352             */
353     if (np->n_ovtype == VNON)
354         np->n_ovtype = vp->v_type;
355
356     out:
357         smb_credrele(&scred);
358         smbfs_rw_exit(&np->r_lkserlock);
359         return (error);
360     }
361
362     /*ARGSUSED*/
363     static int
364     smbfs_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
365                 caller_context_t *ct)
366     {
367         smbnode_t           *np;
368         smbmtinfo_t         *smi;
369         struct smb_cred scred;
370         int error = 0;
371
372         np = VTOSMB(vp);
373         smi = VTOSMI(vp);
374
375         /*
376          * Don't "bail out" for VFS_UNMOUNTED here,
377          * as we want to do cleanup, etc.
378          */
379
380         /*
381          * zone_enter(2) prevents processes from changing zones with SMBFS files
382          * open; if we happen to get here from the wrong zone we can't do
383          * anything over the wire.
384          */
385     if (smi->smi_zone_ref.zref_zone != curproc->p_zone) {
386         /*
387          * We could attempt to clean up locks, except we're sure
388          * that the current process didn't acquire any locks on
389          * the file: any attempt to lock a file belong to another zone
390          * will fail, and one can't lock an SMBFS file and then change
391          * zones, as that fails too.
392          */
393
394         /*
395          * Returning an error here is the sane thing to do. A
396          * subsequent call to VN_RELE() which translates to a
397          * smbfs_inactive() will clean up state: if the zone of the
398          * vnode's origin is still alive and kicking, an async worker
399          * thread will handle the request (from the correct zone), and
400          * everything (minus the final smbfs_getattr_otw() call) should
401          * be OK. If the zone is going away smbfs_async_inactive() will
402          * throw away cached pages inline.
403         */
404         return (EIO);
405     }
406
407     /*
408      * If we are using local locking for this filesystem, then
409      * release all of the SYSV style record locks. Otherwise,
410      * we are doing network locking and we need to release all
411      * of the network locks. All of the locks held by this
412      * process on this file are released no matter what the
413      * incoming reference count is.
414      */
415     if (smi->smi_flags & SMI_LLOCK) {
416         pid_t pid = ddi_get_pid();
417         cleanlocks(vp, pid, 0);

```

```

416         cleanshares(vp, pid);
417     }
418     /*
419      * else doing OtW locking. SMB servers drop all locks
420      * on the file ID we close here, so no _lockrelease()
421      */
422
423     /*
424      * This (passed in) count is the ref. count from the
425      * user's file_t before the closef call (fio.c).
426      * The rest happens only on last close.
427      * We only care when the reference goes away.
428      */
429     if (count > 1)
430         return (0);
431
432     /* NFS has DNLC purge here. */
433
434     /*
435      * If the file was open for write and there are pages,
436      * then make sure dirty pages written back.
437      *
438      * NFS does this async when "close-to-open" is off
439      * (MI_NOCTO flag is set) to avoid blocking the caller.
440      * For now, always do this synchronously (no B_ASYNC).
441      */
442     if ((flag & FWRITE) && vn_has_cached_data(vp)) {
443         error = smbfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
444         if (error == EAGAIN)
445             error = 0;
446     }
447     if (error == 0) {
448         mutex_enter(&np->r_statelock);
449         np->r_flags &= ~RSTALE;
450         np->r_error = 0;
451         mutex_exit(&np->r_statelock);
452     }
453
454     /*
455      * Decrement the reference count for the FID
456      * and possibly do the OtW close.
457      *
458      * Exclusive lock for modifying n_fid stuff.
459      * Don't want this one ever interruptible.
460      */
461     (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
462     smb_credinit(&scred, cr);
463
464     smbfs_rele_fid(np, &scred);
465
466     smb_credrele(&scred);
467     smbfs_rw_exit(&np->r_lkserlock);
468
469 } unchanged_portion_omitted
470
471 /* ARGSUSED */
472 static int
473 smbfs_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
474             caller_context_t *ct)
475 {
476     struct smb_cred scred;
477     struct vattr va;
478     snode_t *np;
479     smbnode_t *smi;
480
481     /*
482      * Get the share information for the file
483      * and the offset and size for the read.
484      */
485     if (smbfs_getshare(vp, &scred, &smi, &np, &uiop->uiodata,
486                       &uiop->uiosize, &uiop->uioloffset, &uiop->uioreadsize))
487         return (EIO);
488
489     /*
490      * Set up the uio structure for the read.
491      */
492     uiop->uiodata = smi->smi_share;
493     uiop->uiosize = uiop->uioreadsize;
494     uiop->uioloffset = uiop->uioloffset;
495     uiop->uioreadsize = uiop->uioreadsize;
496
497     /*
498      * Set up the file attributes for the read.
499      */
500     va.va_size = uiop->uioreadsize;
501     va.va_mtime = uiop->uiotime;
502     va.va_nlink = uiop->uinlink;
503     va.va_type = uiop->uio_type;
504     va.va_mode = uiop->uio_mode;
505
506     /*
507      * Set up the file offset for the read.
508      */
509     if (uiop->uioloffset < 0)
510         uiop->uioloffset = 0;
511
512     /*
513      * Set up the file offset for the read.
514      */
515     if (uiop->uioloffset > uiop->uioreadsize)
516         uiop->uioloffset = uiop->uioreadsize;
517
518     /*
519      * Set up the file offset for the read.
520      */
521     if (uiop->uioloffset < 0)
522         uiop->uioloffset = 0;
523
524     /*
525      * Set up the file offset for the read.
526      */
527     if (uiop->uioloffset > uiop->uioreadsize)
528         uiop->uioloffset = uiop->uioreadsize;
529
530     /*
531      * Set up the file offset for the read.
532      */
533     if (uiop->uioloffset < 0)
534         uiop->uioloffset = 0;
535
536     /*
537      * Set up the file offset for the read.
538      */
539     if (uiop->uioloffset > uiop->uioreadsize)
540         uiop->uioloffset = uiop->uioreadsize;
541
542     /*
543      * Set up the file offset for the read.
544      */
545     if (uiop->uioloffset < 0)
546         uiop->uioloffset = 0;
547
548     /*
549      * Set up the file offset for the read.
550      */
551     if (uiop->uioloffset > uiop->uioreadsize)
552         uiop->uioloffset = uiop->uioreadsize;
553
554     /*
555      * Set up the file offset for the read.
556      */
557     if (uiop->uioloffset < 0)
558         uiop->uioloffset = 0;
559
560     /*
561      * Set up the file offset for the read.
562      */
563     if (uiop->uioloffset > uiop->uioreadsize)
564         uiop->uioloffset = uiop->uioreadsize;
565
566     /*
567      * Set up the file offset for the read.
568      */
569     if (uiop->uioloffset < 0)
570         uiop->uioloffset = 0;
571
572     /*
573      * Set up the file offset for the read.
574      */
575     if (uiop->uioloffset > uiop->uioreadsize)
576         uiop->uioloffset = uiop->uioreadsize;
577
578     /*
579      * Set up the file offset for the read.
580      */
581     if (uiop->uioloffset < 0)
582         uiop->uioloffset = 0;
583
584     /*
585      * Set up the file offset for the read.
586      */
587     if (uiop->uioloffset > uiop->uioreadsize)
588         uiop->uioloffset = uiop->uioreadsize;
589
590     /*
591      * Set up the file offset for the read.
592      */
593     if (uiop->uioloffset < 0)
594         uiop->uioloffset = 0;
595
596     /*
597      * Set up the file offset for the read.
598      */
599     if (uiop->uioloffset > uiop->uioreadsize)
600         uiop->uioloffset = uiop->uioreadsize;
601
602     /*
603      * Set up the file offset for the read.
604      */
605     if (uiop->uioloffset < 0)
606         uiop->uioloffset = 0;
607
608     /*
609      * Set up the file offset for the read.
610      */
611     if (uiop->uioloffset > uiop->uioreadsize)
612         uiop->uioloffset = uiop->uioreadsize;
613
614     /*
615      * Set up the file offset for the read.
616      */
617     if (uiop->uioloffset < 0)
618         uiop->uioloffset = 0;
619
620     /*
621      * Set up the file offset for the read.
622      */
623     if (uiop->uioloffset > uiop->uioreadsize)
624         uiop->uioloffset = uiop->uioreadsize;
625
626     /*
627      * Set up the file offset for the read.
628      */
629     if (uiop->uioloffset < 0)
630         uiop->uioloffset = 0;
631
632     /*
633      * Set up the file offset for the read.
634      */
635     if (uiop->uioloffset > uiop->uioreadsize)
636         uiop->uioloffset = uiop->uioreadsize;
637
638     /*
639      * Set up the file offset for the read.
640      */
641     if (uiop->uioloffset < 0)
642         uiop->uioloffset = 0;
643
644     /*
645      * Set up the file offset for the read.
646      */
647     if (uiop->uioloffset > uiop->uioreadsize)
648         uiop->uioloffset = uiop->uioreadsize;
649
650     /*
651      * Set up the file offset for the read.
652      */
653     if (uiop->uioloffset < 0)
654         uiop->uioloffset = 0;
655
656     /*
657      * Set up the file offset for the read.
658      */
659     if (uiop->uioloffset > uiop->uioreadsize)
660         uiop->uioloffset = uiop->uioreadsize;
661
662     /*
663      * Set up the file offset for the read.
664      */
665     if (uiop->uioloffset < 0)
666         uiop->uioloffset = 0;
667
668     /*
669      * Set up the file offset for the read.
670      */
671     if (uiop->uioloffset > uiop->uioreadsize)
672         uiop->uioloffset = uiop->uioreadsize;
673
674     /*
675      * Set up the file offset for the read.
676      */
677     if (uiop->uioloffset < 0)
678         uiop->uioloffset = 0;
679
680     /*
681      * Set up the file offset for the read.
682      */
683     if (uiop->uioloffset > uiop->uioreadsize)
684         uiop->uioloffset = uiop->uioreadsize;
685
686     /*
687      * Set up the file offset for the read.
688      */
689     if (uiop->uioloffset < 0)
690         uiop->uioloffset = 0;
691
692     /*
693      * Set up the file offset for the read.
694      */
695     if (uiop->uioloffset > uiop->uioreadsize)
696         uiop->uioloffset = uiop->uioreadsize;
697
698     /*
699      * Set up the file offset for the read.
700      */
701     if (uiop->uioloffset < 0)
702         uiop->uioloffset = 0;
703
704     /*
705      * Set up the file offset for the read.
706      */
707     if (uiop->uioloffset > uiop->uioreadsize)
708         uiop->uioloffset = uiop->uioreadsize;
709
710     /*
711      * Set up the file offset for the read.
712      */
713     if (uiop->uioloffset < 0)
714         uiop->uioloffset = 0;
715
716     /*
717      * Set up the file offset for the read.
718      */
719     if (uiop->uioloffset > uiop->uioreadsize)
720         uiop->uioloffset = uiop->uioreadsize;
721
722     /*
723      * Set up the file offset for the read.
724      */
725     if (uiop->uioloffset < 0)
726         uiop->uioloffset = 0;
727
728     /*
729      * Set up the file offset for the read.
730      */
731     if (uiop->uioloffset > uiop->uioreadsize)
732         uiop->uioloffset = uiop->uioreadsize;
733
734     /*
735      * Set up the file offset for the read.
736      */
737     if (uiop->uioloffset < 0)
738         uiop->uioloffset = 0;
739
740     /*
741      * Set up the file offset for the read.
742      */
743     if (uiop->uioloffset > uiop->uioreadsize)
744         uiop->uioloffset = uiop->uioreadsize;
745
746     /*
747      * Set up the file offset for the read.
748      */
749     if (uiop->uioloffset < 0)
750         uiop->uioloffset = 0;
751
752     /*
753      * Set up the file offset for the read.
754      */
755     if (uiop->uioloffset > uiop->uioreadsize)
756         uiop->uioloffset = uiop->uioreadsize;
757
758     /*
759      * Set up the file offset for the read.
760      */
761     if (uiop->uioloffset < 0)
762         uiop->uioloffset = 0;
763
764     /*
765      * Set up the file offset for the read.
766      */
767     if (uiop->uioloffset > uiop->uioreadsize)
768         uiop->uioloffset = uiop->uioreadsize;
769
770     /*
771      * Set up the file offset for the read.
772      */
773     if (uiop->uioloffset < 0)
774         uiop->uioloffset = 0;
775
776     /*
777      * Set up the file offset for the read.
778      */
779     if (uiop->uioloffset > uiop->uioreadsize)
780         uiop->uioloffset = uiop->uioreadsize;
781
782     /*
783      * Set up the file offset for the read.
784      */
785     if (uiop->uioloffset < 0)
786         uiop->uioloffset = 0;
787
788     /*
789      * Set up the file offset for the read.
790      */
791     if (uiop->uioloffset > uiop->uioreadsize)
792         uiop->uioloffset = uiop->uioreadsize;
793
794     /*
795      * Set up the file offset for the read.
796      */
797     if (uiop->uioloffset < 0)
798         uiop->uioloffset = 0;
799
799
800     /*
801      * Set up the file offset for the read.
802      */
803     if (uiop->uioloffset > uiop->uioreadsize)
804         uiop->uioloffset = uiop->uioreadsize;
805
806     /*
807      * Set up the file offset for the read.
808      */
809     if (uiop->uioloffset < 0)
810         uiop->uioloffset = 0;
811
812     /*
813      * Set up the file offset for the read.
814      */
815     if (uiop->uioloffset > uiop->uioreadsize)
816         uiop->uioloffset = uiop->uioreadsize;
817
818     /*
819      * Set up the file offset for the read.
820      */
821     if (uiop->uioloffset < 0)
822         uiop->uioloffset = 0;
823
824     /*
825      * Set up the file offset for the read.
826      */
827     if (uiop->uioloffset > uiop->uioreadsize)
828         uiop->uioloffset = uiop->uioreadsize;
829
830     /*
831      * Set up the file offset for the read.
832      */
833     if (uiop->uioloffset < 0)
834         uiop->uioloffset = 0;
835
836     /*
837      * Set up the file offset for the read.
838      */
839     if (uiop->uioloffset > uiop->uioreadsize)
840         uiop->uioloffset = uiop->uioreadsize;
841
842     /*
843      * Set up the file offset for the read.
844      */
845     if (uiop->uioloffset < 0)
846         uiop->uioloffset = 0;
847
848     /*
849      * Set up the file offset for the read.
850      */
851     if (uiop->uioloffset > uiop->uioreadsize)
852         uiop->uioloffset = uiop->uioreadsize;
853
854     /*
855      * Set up the file offset for the read.
856      */
857     if (uiop->uioloffset < 0)
858         uiop->uioloffset = 0;
859
860     /*
861      * Set up the file offset for the read.
862      */
863     if (uiop->uioloffset > uiop->uioreadsize)
864         uiop->uioloffset = uiop->uioreadsize;
865
866     /*
867      * Set up the file offset for the read.
868      */
869     if (uiop->uioloffset < 0)
870         uiop->uioloffset = 0;
871
872     /*
873      * Set up the file offset for the read.
874      */
875     if (uiop->uioloffset > uiop->uioreadsize)
876         uiop->uioloffset = uiop->uioreadsize;
877
878     /*
879      * Set up the file offset for the read.
880      */
881     if (uiop->uioloffset < 0)
882         uiop->uioloffset = 0;
883
884     /*
885      * Set up the file offset for the read.
886      */
887     if (uiop->uioloffset > uiop->uioreadsize)
888         uiop->uioloffset = uiop->uioreadsize;
889
890     /*
891      * Set up the file offset for the read.
892      */
893     if (uiop->uioloffset < 0)
894         uiop->uioloffset = 0;
895
896     /*
897      * Set up the file offset for the read.
898      */
899     if (uiop->uioloffset > uiop->uioreadsize)
900         uiop->uioloffset = uiop->uioreadsize;
901
902     /*
903      * Set up the file offset for the read.
904      */
905     if (uiop->uioloffset < 0)
906         uiop->uioloffset = 0;
907
908     /*
909      * Set up the file offset for the read.
910      */
911     if (uiop->uioloffset > uiop->uioreadsize)
912         uiop->uioloffset = uiop->uioreadsize;
913
914     /*
915      * Set up the file offset for the read.
916      */
917     if (uiop->uioloffset < 0)
918         uiop->uioloffset = 0;
919
920     /*
921      * Set up the file offset for the read.
922      */
923     if (uiop->uioloffset > uiop->uioreadsize)
924         uiop->uioloffset = uiop->uioreadsize;
925
926     /*
927      * Set up the file offset for the read.
928      */
929     if (uiop->uioloffset < 0)
930         uiop->uioloffset = 0;
931
932     /*
933      * Set up the file offset for the read.
934      */
935     if (uiop->uioloffset > uiop->uioreadsize)
936         uiop->uioloffset = uiop->uioreadsize;
937
938     /*
939      * Set up the file offset for the read.
940      */
940     if (uiop->uioloffset < 0)
941         uiop->uioloffset = 0;
942
943     /*
944      * Set up the file offset for the read.
945      */
946     if (uiop->uioloffset > uiop->uioreadsize)
947         uiop->uioloffset = uiop->uioreadsize;
948
949     /*
950      * Set up the file offset for the read.
951      */
951     if (uiop->uioloffset < 0)
952         uiop->uioloffset = 0;
953
954     /*
955      * Set up the file offset for the read.
956      */
956     if (uiop->uioloffset > uiop->uioreadsize)
957         uiop->uioloffset = uiop->uioreadsize;
958
959     /*
960      * Set up the file offset for the read.
961      */
961     if (uiop->uioloffset < 0)
962         uiop->uioloffset = 0;
963
964     /*
965      * Set up the file offset for the read.
966      */
966     if (uiop->uioloffset > uiop->uioreadsize)
967         uiop->uioloffset = uiop->uioreadsize;
968
969     /*
970      * Set up the file offset for the read.
971      */
971     if (uiop->uioloffset < 0)
972         uiop->uioloffset = 0;
973
974     /*
975      * Set up the file offset for the read.
976      */
976     if (uiop->uioloffset > uiop->uioreadsize)
977         uiop->uioloffset = uiop->uioreadsize;
978
979     /*
980      * Set up the file offset for the read.
981      */
981     if (uiop->uioloffset < 0)
982         uiop->uioloffset = 0;
983
984     /*
985      * Set up the file offset for the read.
986      */
986     if (uiop->uioloffset > uiop->uioreadsize)
987         uiop->uioloffset = uiop->uioreadsize;
988
989     /*
990      * Set up the file offset for the read.
991      */
991     if (uiop->uioloffset < 0)
992         uiop->uioloffset = 0;
993
994     /*
995      * Set up the file offset for the read.
996      */
996     if (uiop->uioloffset > uiop->uioreadsize)
997         uiop->uioloffset = uiop->uioreadsize;
998
999     /*
1000      * Set up the file offset for the read.
1001      */
1001     if (uiop->uioloffset < 0)
1002         uiop->uioloffset = 0;
1003
1004     /*
1005      * Set up the file offset for the read.
1006      */
1006     if (uiop->uioloffset > uiop->uioreadsize)
1007         uiop->uioloffset = uiop->uioreadsize;
1008
1009     /*
1010      * Set up the file offset for the read.
1011      */
1011     if (uiop->uioloffset < 0)
1012         uiop->uioloffset = 0;
1013
1014     /*
1015      * Set up the file offset for the read.
1016      */
1016     if (uiop->uioloffset > uiop->uioreadsize)
1017         uiop->uioloffset = uiop->uioreadsize;
1018
1019     /*
1020      * Set up the file offset for the read.
1021      */
1021     if (uiop->uioloffset < 0)
1022         uiop->uioloffset = 0;
1023
1024     /*
1025      * Set up the file offset for the read.
1026      */
1026     if (uiop->uioloffset > uiop->uioreadsize)
1027         uiop->uioloffset = uiop->uioreadsize;
1028
1029     /*
1030      * Set up the file offset for the read.
1031      */
1031     if (uiop->uioloffset < 0)
1032         uiop->uioloffset = 0;
1033
1034     /*
1035      * Set up the file offset for the read.
1036      */
1036     if (uiop->uioloffset > uiop->uioreadsize)
1037         uiop->uioloffset = uiop->uioreadsize;
1038
1039     /*
1040      * Set up the file offset for the read.
1041      */
1041     if (uiop->uioloffset < 0)
1042         uiop->uioloffset = 0;
1043
1044     /*
1045      * Set up the file offset for the read.
1046      */
1046     if (uiop->uioloffset > uiop->uioreadsize)
1047         uiop->uioloffset = uiop->uioreadsize;
1048
1049     /*
1050      * Set up the file offset for the read.
1051      */
1051     if (uiop->uioloffset < 0)
1052         uiop->uioloffset = 0;
1053
1054     /*
1055      * Set up the file offset for the read.
1056      */
1056     if (uiop->uioloffset > uiop->uioreadsize)
1057         uiop->uioloffset = uiop->uioreadsize;
1058
1059     /*
1060      * Set up the file offset for the read.
1061      */
1061     if (uiop->uioloffset < 0)
1062         uiop->uioloffset = 0;
1063
1064     /*
1065      * Set up the file offset for the read.
1066      */
1066     if (uiop->uioloffset > uiop->uioreadsize)
1067         uiop->uioloffset = uiop->uioreadsize;
1068
1069     /*
1070      * Set up the file offset for the read.
1071      */
1071     if (uiop->uioloffset < 0)
1072         uiop->uioloffset = 0;
1073
1074     /*
1075      * Set up the file offset for the read.
1076      */
1076     if (uiop->uioloffset > uiop->uioreadsize)
1077         uiop->uioloffset = uiop->uioreadsize;
1078
1079     /*
1080      * Set up the file offset for the read.
1081      */
1081     if (uiop->uioloffset < 0)
1082         uiop->uioloffset = 0;
1083
1084     /*
1085      * Set up the file offset for the read.
1086      */
1086     if (uiop->uioloffset > uiop->uioreadsize)
1087         uiop->uioloffset = uiop->uioreadsize;
1088
1089     /*
1090      * Set up the file offset for the read.
1091      */
1091     if (uiop->uioloffset < 0)
1092         uiop->uioloffset = 0;
1093
1094     /*
1095      * Set up the file offset for the read.
1096      */
1096     if (uiop->uioloffset > uiop->uioreadsize)
1097         uiop->uioloffset = uiop->uioreadsize;
1098
1099     /*
1100      * Set up the file offset for the read.
1101      */
1101     if (uiop->uioloffset < 0)
1102         uiop->uioloffset = 0;
1103
1104     /*
1105      * Set up the file offset for the read.
1106      */
1106     if (uiop->uioloffset > uiop->uioreadsize)
1107         uiop->uioloffset = uiop->uioreadsize;
1108
1109     /*
1110      * Set up the file offset for the read.
1111      */
1111     if (uiop->uioloffset < 0)
1112         uiop->uioloffset = 0;
1113
1114     /*
1115      * Set up the file offset for the read.
1116      */
1116     if (uiop->uioloffset > uiop->uioreadsize)
1117         uiop->uioloffset = uiop->uioreadsize;
1118
1119     /*
1120      * Set up the file offset for the read.
1121      */
1121     if (uiop->uioloffset < 0)
1122         uiop->uioloffset = 0;
1123
1124     /*
1125      * Set up the file offset for the read.
1126      */
1126     if (uiop->uioloffset > uiop->uioreadsize)
1127         uiop->uioloffset = uiop->uioreadsize;
1128
1129     /*
1130      * Set up the file offset for the read.
1131      */
1131     if (uiop->uioloffset < 0)
1132         uiop->uioloffset = 0;
1133
1134     /*
1135      * Set up the file offset for the read.
1136      */
1136     if (uiop->uioloffset > uiop->uioreadsize)
1137         uiop->uioloffset = uiop->uioreadsize;
1138
1139     /*
1140      * Set up the file offset for the read.
1141      */
1141     if (uiop->uioloffset < 0)
1142         uiop->uioloffset = 0;
1143
1144     /*
1145      * Set up the file offset for the read.
1146      */
1146     if (uiop->uioloffset > uiop->uioreadsize)
1147         uiop->uioloffset = uiop->uioreadsize;
1148
1149     /*
1150      * Set up the file offset for the read.
1151      */
1151     if (uiop->uioloffset < 0)
1152         uiop->uioloffset = 0;
1153
1154     /*
1155      * Set up the file offset for the read.
1156      */
1156     if (uiop->uioloffset > uiop->uioreadsize)
1157         uiop->uioloffset = uiop->uioreadsize;
1158
1159     /*
1160      * Set up the file offset for the read.
1161      */
1161     if (uiop->uioloffset < 0)
1162         uiop->uioloffset = 0;
1163
1164     /*
1165      * Set up the file offset for the read.
1166      */
1166     if (uiop->uioloffset > uiop->uioreadsize)
1167         uiop->uioloffset = uiop->uioreadsize;
1168
1169     /*
1170      * Set up the file offset for the read.
1171      */
1171     if (uiop->uioloffset < 0)
1172         uiop->uioloffset = 0;
1173
1174     /*
1175      * Set up the file offset for the read.
1176      */
1176     if (uiop->uioloffset > uiop->uioreadsize)
1177         uiop->uioloffset = uiop->uioreadsize;
1178
1179     /*
1180      * Set up the file offset for the read.
1181      */
1181     if (uiop->uioloffset < 0)
1182         uiop->uioloffset = 0;
1183
1184     /*
1185      * Set up the file offset for the read.
1186      */
1186     if (uiop->uioloffset > uiop->uioreadsize)
1187         uiop->uioloffset = uiop->uioreadsize;
1188
1189     /*
1190      * Set up the file offset for the read.
1191      */
1191     if (uiop->uioloffset < 0)
1192         uiop->uioloffset = 0;
1193
1194     /*
1195      * Set up the file offset for the read.
1196      */
1196     if (uiop->uioloffset > uiop->uioreadsize)
1197         uiop->uioloffset = uiop->uioreadsize;
1198
1199     /*
1200      * Set up the file offset for the read.
1201      */
1201     if (uiop->uioloffset < 0)
1202         uiop->uioloffset = 0;
1203
1204     /*
1205      * Set up the file offset for the read.
1206      */
1206     if (uiop->uioloffset > uiop->uioreadsize)
1207         uiop->uioloffset = uiop->uioreadsize;
1208
1209     /*
1210      * Set up the file offset for the read.
1211      */
1211     if (uiop->uioloffset < 0)
1212         uiop->uioloffset = 0;
1213
1214     /*
1215      * Set up the file offset for the read.
1216      */
1216     if (uiop->uioloffset > uiop->uioreadsize)
1217         uiop->uioloffset = uiop->uioreadsize;
1218
1219     /*
1220      * Set up the file offset for the read.
1221      */
1221     if (uiop->uioloffset < 0)
1222         uiop->uioloffset = 0;
1223
1224     /*
1225      * Set up the file offset for the read.
1226      */
1226     if (uiop->uioloffset > uiop->uioreadsize)
1227         uiop->uioloffset = uiop->uioreadsize;
1228
1229     /*
1230      * Set up the file offset for the read.
1231      */
1231     if (uiop->uioloffset < 0)
1232         uiop->uioloffset = 0;
1233
1234     /*
1235      * Set up the file offset for the read.
1236      */
1236     if (uiop->uioloffset > uiop->uioreadsize)
1237         uiop->uioloffset = uiop->uioreadsize;
1238
1239     /*
1240      * Set up the file offset for the read.
1241      */
1241     if (uiop->uioloffset < 0)
1242         uiop->uioloffset = 0;
1243
1244     /*
1245      * Set up the file offset for the read.
1246      */
1246     if (uiop->uioloffset > uiop->uioreadsize)
1247         uiop->uioloffset = uiop->uioreadsize;
1248
1249     /*
1250      * Set up the file offset for the read.
1251      */
1251     if (uiop->uioloffset < 0)
1252         uiop->uioloffset = 0;
1253
1254     /*
12
```

```

625     (((np->r_flags & RDIRECTIO) || (smi->smi_flags & SMI_DIRECTIO)) &&
626     np->r_mapcnt == 0 && np->r_inmap == 0 &&
627     !vn_has_cached_data(vp))) {
628
629     /* Shared lock for n_fid use in smb_rwui0 */
630     if (smbfs_rw_enter_sig(np->r_lkserlock, RW_READER, SMBINTR(vp)))
631         return (EINTR);
632     smb_credinit(&scred, cr);
633
634     /* After reconnect, n_fid is invalid */
635     if (np->n_vcgenid != ssp->ss_vcgenid)
636         error = ESTALE;
637     else
638         error = smb_rwui0(ssp, np->n_fid, UIO_READ,
639                           uiop, &scred, smb_timo_read);
640
641     smb_credrele(&scred);
642     smbfs_rw_exit(&np->r_lkserlock);
643
644     /* undo adjustment of resid */
645     uiop->uio_resid += past_eof;
646
647     return (error);
648 }
649
650 /* (else) Do I/O through segmap. */
651 do {
652     off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
653     on = uiop->uio_loffset & MAXOFFSET; /* Relative offset */
654     n = MIN(MAXBSIZE - on, uiop->uio_resid);
655
656     error = smbfs_validate_caches(vp, cr);
657     if (error)
658         break;
659
660     /* NFS waits for RINCACHEPURGE here. */
661
662     if (vpm_enable) {
663         /*
664          * Copy data.
665          */
666         error = vpm_data_copy(vp, off + on, n, uiop,
667                               1, NULL, 0, S_READ);
668     } else {
669         base = segmap_getmapf1t(segkmap, vp, off + on, n, 1,
670                                S_READ);
671
672         error = uiomove(base + on, n, UIO_READ, uiop);
673     }
674
675     if (!error) {
676         /*
677          * If read a whole block or read to eof,
678          * won't need this buffer again soon.
679          */
680         mutex_enter(&np->r_statelock);
681         if (n + on == MAXBSIZE ||
682             uiop->uio_loffset == np->r_size)
683             flags = SM_DONTNEED;
684         else
685             flags = 0;
686         mutex_exit(&np->r_statelock);
687         if (vpm_enable) {
688             error = vpm_sync_pages(vp, off, n, flags);
689         } else {
690             error = segmap_release(segkmap, base, flags);
691         }
692     }
693 }
694
695     if (vpm_enable) {
696         (void) vpm_sync_pages(vp, off, n, 0);
697     } else {
698         (void) segmap_release(segkmap, base, 0);
699     }
700 }
701
702 /* undo adjustment of resid */
703 uiop->uio_resid += past_eof;
704
705 }

```

```

691
692     } else {
693         if (vpm_enable) {
694             (void) vpm_sync_pages(vp, off, n, 0);
695         } else {
696             (void) segmap_release(segkmap, base, 0);
697         }
698     }
699 }
700
701 /* error & uiop->uio_resid > 0);
702
703 /* undo adjustment of resid */
704 uiop->uio_resid += past_eof;
705
706 }
707
708 /* ARGSUSED */
709 static int
710 smbfs_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
711              caller_context_t *ct)
712 {
713     struct smb_cred scred;
714     struct vattr va;
715     smbnode_t *np;
716     smbmtinfo_t *smi;
717     smb_share_t *ssp;
718     offset_t endoff, limit;
719     ssize_t past_limit;
720     int error, timo;
721     caddr_t base;
722     u_offset_t off;
723     size_t n;
724     int on;
725     uint_t flags;
726     u_offset_t last_off;
727     size_t last_resid;
728     uint_t bsize;
729
730     np = VTOSMB(vp);
731     smi = VTOSMI(vp);
732     ssp = smi->smi_share;
733
734     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
735         return (EIO);
736
737     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
738         return (EIO);
739
740     ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_WRITER));
741
742     if (vp->v_type != VREG)
743         return (EISDIR);
744
745     if (uiop->uio_resid == 0)
746         return (0);
747
748     /*
749      * Handle ioflag bits: (FAPPEND|FSYNC|FDSYNC)
750      */
751     if (ioflag & (FAPPEND | FSYNC)) {
752         if (np->n_flag & NMODIFIED) {
753             smbfs_attrcache_remove(np);
754             /* XXX: smbfs_vinvalbuf? */
755         }
756     }

```

[new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c](#)

13

[new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c](#)

14

```

820
821             /*
822             * A close may have cleared r_error, if so,
823             * propagate ESTALE error return properly
824             */
825             if (error == 0)
826                 error = ESTALE;
827             goto bottom;
828
829         /* Timeout: longer for append. */
830         timo = smb_timo_write;
831         if (endoff > np->r_size)
832             timo = smb_timo_append;
833
834         /* Shared lock for n_fid use in smb_rwui0 */
835         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
836             return (EINTR);
837         smb_creditinit(&scred, cr);
838
839         /* After reconnect, n_fid is invalid */
840         if (np->n_vcgenid != ssp->ss_vcgenid)
841             error = ESTALE;
842         else
843             error = smb_rwui0(ssp, np->n_fid, UIO_WRITE,
844                               uiop, &scred, timo);
845
846         if (error == 0) {
847             mutex_enter(&np->r_statelock);
848             np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
849             if (uiop->uio_loffset > (offset_t)np->r_size)
850                 np->r_size = (len_t)uiop->uio_loffset;
851             mutex_exit(&np->r_statelock);
852             if (ioflag & (FSYNC | FDSYNC)) {
853                 /* Don't error the I/O if this fails. */
854                 (void) smbfs_smb_flush(np, &scred);
855             }
856         }
857
858         smb_credrele(&scred);
859         smbfs_rw_exit(&np->r_lkserlock);
860
861         /* undo adjustment of resid */
862         uiop->uio_resid += past_limit;
863
864     return (error);
865 }
866
867 /* (else) Do I/O through segmap. */
868 bsize = vp->v_vfsp->vfs_bsiz;
869
870 do {
871     off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
872     on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
873     n = MIN(MAXBSIZE - on, uiop->uio_resid);
874
875     last_resid = uiop->uio_resid;
876     last_off = uiop->uio_loffset;
877
878     if (np->r_flags & RSTALE) {
879         error = np->r_error;
880         /*
881          * A close may have cleared r_error, if so,
882          * propagate ESTALE error return properly
883          */
884         if (error == 0)

```

```

885             error = ESTALE;
886         }
887     }
888
889     /*
890      * From NFS: Don't create dirty pages faster than they
891      * can be cleaned.
892      *
893      * Here NFS also checks for async writes (np->r_awcount)
894      */
895     mutex_enter(&np->r_statelock);
896     while (np->r_gcount > 0) {
897         if (SMBINTR(vp)) {
898             klwp_t *lwp = ttolwp(curthread);
899
900             if (lwp != NULL)
901                 lwp->lwp_nostop++;
902             if (!cv_wait_sig(&np->r_cv, &np->r_statelock)) {
903                 mutex_exit(&np->r_statelock);
904                 if (lwp != NULL)
905                     lwp->lwp_nostop--;
906                 error = EINTR;
907                 goto bottom;
908
909                 if (lwp != NULL)
910                     lwp->lwp_nostop--;
911             } else
912                 cv_wait(&np->r_cv, &np->r_statelock);
913         }
914     }
915     mutex_exit(&np->r_statelock);
916
917     /*
918      * Touch the page and fault it in if it is not in core
919      * before segmap_getmapflt or vpm_data_copy can lock it.
920      * This is to avoid the deadlock if the buffer is mapped
921      * to the same file through mmap which we want to write.
922      */
923     uio_prefaultpages((long)n, uiop);
924
925     if (vpm_enable) {
926         /*
927          * It will use kpm mappings, so no need to
928          * pass an address.
929          */
930     } else {
931         if (segmap_kpm) {
932             int pon = uiop->uio_loffset & PAGEOFFSET;
933             size_t pn = MIN(PAGESIZE - pon,
934                             uiop->uio_resid);
935             int pagecreate;
936
937             mutex_enter(&np->r_statelock);
938             pagecreate = (pon == 0) && (pn == PAGESIZE ||
939                           uiop->uio_loffset + pn >= np->r_size);
940             mutex_exit(&np->r_statelock);
941
942             base = segmap_getmapflt(segkmap, vp, off + on,
943                                     pn, !pagecreate, S_WRITE);
944
945             error = smbfs_writenp(np, base + pon, n, uiop,
946                                   pagecreate);
947
948         } else {
949             base = segmap_getmapflt(segkmap, vp, off + on,
950                                     n, 0, S_READ);
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
296
```

```

1017 {
1018     int      pagecreate;
1019     int      n;
1020     int      saved_n;
1021     caddr_t  saved_base;
1022     u_offset_t offset;
1023     int      error;
1024     int      sm_error;
1025     vnode_t   *vp = SMBTOV(np);

1027     ASSERT(tcount <= MAXBSIZE && tcount <= uio->uio_resid);
1028     ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_WRITER));
1029     if (!vpm_enable) {
1030         ASSERT(((uintptr_t)base & MAXBOFFSET) + tcount <= MAXBSIZE);
1031     }

1033 /*
1034  * Move bytes in at most PAGESIZE chunks. We must avoid
1035  * spanning pages in uiomove() because page faults may cause
1036  * the cache to be invalidated out from under us. The r_size is not
1037  * updated until after the uiomove. If we push the last page of a
1038  * file before r_size is correct, we will lose the data written past
1039  * the current (and invalid) r_size.
1040 */
1041 do {
1042     offset = uio->uio_loffset;
1043     pagecreate = 0;

1045 /*
1046  * n is the number of bytes required to satisfy the request
1047  * or the number of bytes to fill out the page.
1048 */
1049 n = (int)MIN((PAGESIZE - (offset & PAGEOFFSET)), tcount);

1051 /*
1052  * Check to see if we can skip reading in the page
1053  * and just allocate the memory. We can do this
1054  * if we are going to rewrite the entire mapping
1055  * or if we are going to write to or beyond the current
1056  * end of file from the beginning of the mapping.
1057 */
1058 /*
1059  * The read of r_size is now protected by r_statelock.
1060 */
1061 mutex_enter(&np->r_statelock);
1062 /*
1063  * When pgcreated is nonzero the caller has already done
1064  * a segmap_getmapflt with forcefault 0 and S_WRITE. With
1065  * segkpm this means we already have at least one page
1066  * created and mapped at base.
1067 */
1068 pagecreate = pgcreated ||
1069 ((offset & PAGEOFFSET) == 0 &&
1070 (n == PAGESIZE || ((offset + n) >= np->r_size)));

1071 mutex_exit(&np->r_statelock);
1072 if (!vpm_enable && pagecreate) {
1073     /*
1074      * The last argument tells segmap_pagecreate() to
1075      * always lock the page, as opposed to sometimes
1076      * returning with the page locked. This way we avoid a
1077      * fault on the ensuing uiomove(), but also
1078      * more importantly (to fix bug 1094402) we can
1079      * call segmap_fault() to unlock the page in all
1080      * cases. An alternative would be to modify
1081      * segmap_pagecreate() to tell us when it is
1082      * locking a page, but that's a fairly major

```

```

1083                                     * interface change.
1084                                     */
1085     if (pgcreated == 0)
1086         (void) segmap_pagecreate(segkmap, base,
1087                                  (uint_t)n, 1);
1088     saved_base = base;
1089     saved_n = n;
1090 }

1092 /**
1093  * The number of bytes of data in the last page can not
1094  * be accurately be determined while page is being
1095  * uiomove'd to and the size of the file being updated.
1096  * Thus, inform threads which need to know accurately
1097  * how much data is in the last page of the file. They
1098  * will not do the i/o immediately, but will arrange for
1099  * the i/o to happen later when this modify operation
1100  * will have finished.
1101 */
1102 ASSERT(!(np->r_flags & RMODINPROGRESS));
1103 mutex_enter(&np->r_statelock);
1104 np->r_flags |= RMODINPROGRESS;
1105 np->r_modaddr = (offset & MAXBMASK);
1106 mutex_exit(&np->r_statelock);

1108 if (vpm_enable) {
1109     /*
1110      * Copy data. If new pages are created, part of
1111      * the page that is not written will be initialized
1112      * with zeros.
1113 */
1114     error = vpm_data_copy(vp, offset, n, uio,
1115                           !pagecreate, NULL, 0, S_WRITE);
1116 } else {
1117     error = uiomove(base, n, UIO_WRITE, uio);
1118 }

1120 /**
1121  * r_size is the maximum number of
1122  * bytes known to be in the file.
1123  * Make sure it is at least as high as the
1124  * first unwritten byte pointed to by uio_loffset.
1125 */
1126 mutex_enter(&np->r_statelock);
1127 if (np->r_size < uio->uio_loffset)
1128     np->r_size = uio->uio_loffset;
1129 np->r_flags &= ~RMODINPROGRESS;
1130 np->r_flags |= RDIRTY;
1131 mutex_exit(&np->r_statelock);

1133 /* n = # of bytes written */
1134 n = (int)(uio->uio_loffset - offset);

1136 if (!vpm_enable) {
1137     base += n;
1138 }
1139 tcount -= n;
1140 /*
1141  * If we created pages w/o initializing them completely,
1142  * we need to zero the part that wasn't set up.
1143  * This happens on a most EOF write cases and if
1144  * we had some sort of error during the uiomove.
1145 */
1146 if (!vpm_enable && pagecreate) {
1147     if ((uio->uio_loffset & PAGEOFFSET) || n == 0)
1148         (void) kzero(base, PAGESIZE - n);

```

```

1150
1151     if (pgcreated) {
1152         /*
1153          * Caller is responsible for this page,
1154          * it was not created in this loop.
1155         */
1156         pgcreated = 0;
1157     } else {
1158         /*
1159          * For bug 1094402: segmap_pagecreate locks
1160          * page. Unlock it. This also unlocks the
1161          * pages allocated by page_create_va() in
1162          * segmap_pagecreate().
1163         */
1164         sm_error = segmap_fault(kas.a_hat, segkmap,
1165             saved_base, saved_n,
1166             F_SOFTUNLOCK, S_WRITE);
1167         if (error == 0)
1168             error = sm_error;
1169     }
1170 } while (tcount > 0 && error == 0);
1171
1172 return (error);
1173 }

1174 /*
1175  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED}
1176  * Like nfs3_rdwrln()
1177  */
1178 static int
1179 smbfs_rdwrln(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
1180   int flags, cred_t *cr)
1181 {
1182     smbmntinfo_t *smi = VTOSMI(vp);
1183     struct buf *bp;
1184     int error;
1185     int sync;
1186
1187     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1188         return (EINVAL);
1189
1190     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1191         return (EINVAL);
1192
1193     bp = pageio_setup(pp, len, vp, flags);
1194     ASSERT(bp != NULL);
1195
1196     /*
1197      * pageio_setup should have set b_addr to 0. This
1198      * is correct since we want to do I/O on a page
1199      * boundary. bp_mapin will use this addr to calculate
1200      * an offset, and then set b_addr to the kernel virtual
1201      * address it allocated for us.
1202     */
1203     ASSERT(bp->b_un.b_addr == 0);
1204
1205     bp->b_edev = 0;
1206     bp->b_dev = 0;
1207     bp->b_lblkno = lbtodb(off);
1208     bp->b_file = vp;
1209     bp->b_offset = (offset_t)off;
1210     bp_mapin(bp);
1211
1212     /*
1213      * Calculate the desired level of stability to write data.
1214 
```

```

1215
1216     /*
1217      * if ((flags & (B_WRITE|B_ASYNC)) == (B_WRITE|B_ASYNC) &&
1218      * freemem > desfree) {
1219      *     sync = 0;
1220      } else {
1221      *     sync = 1;
1222      }
1223
1224     error = smbfs_bio(bp, sync, cr);
1225
1226     bp_mapout(bp);
1227     pageio_done(bp);
1228
1229     return (error);
1230 }

1231 /*
1232  * Corresponds to nfs3_vnopc.c : nfs3_bio(), though the NFS code
1233  * uses nfs3read()/nfs3write() where we use smb_rwui0(). Also,
1234  * NFS has this later in the file. Move it up here closer to
1235  * the one call site just above.
1236 */
1237
1238 static int
1239 smbfs_bio(struct buf *bp, int sync, cred_t *cr)
1240 {
1241     struct iovec aiov[1];
1242     struct uio auio;
1243     struct smb_cred scred;
1244     smbnode_t *np = VTOSMB(bp->b_vp);
1245     smbmntinfo_t *smi = np->n_mount;
1246     smb_share_t *ssp = smi->smi_share;
1247     offset_t offset;
1248     offset_t endoff;
1249     size_t count;
1250     size_t past_eof;
1251     int error;
1252
1253     ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);
1254
1255     offset = ldtob(bp->b_lblkno);
1256     count = bp->b_bcount;
1257     endoff = offset + count;
1258     if (offset < 0 || endoff < 0)
1259         return (EINVAL);
1260
1261     /*
1262      * Limit file I/O to the remaining file size, but see
1263      * the notes in smbfs_getpage about SMBFS_EOF.
1264     */
1265     mutex_enter(&np->r_statelock);
1266     if (offset >= np->r_size) {
1267         mutex_exit(&np->r_statelock);
1268         if (bp->b_flags & B_READ) {
1269             return (SMBFS_EOF);
1270         } else {
1271             return (EINVAL);
1272         }
1273     }
1274     if (endoff > np->r_size) {
1275         past_eof = (size_t)(endoff - np->r_size);
1276         count -= past_eof;
1277     } else
1278         past_eof = 0;
1279     mutex_exit(&np->r_statelock);
1280 }
```

```

1281     ASSERT(count > 0);

1282     /* Caller did bpmapin(). Mapped address is... */
1283     aiov[0].iov_base = bp->b_un.b_addr;
1284     aiov[0].iov_len = count;
1285     aioio.uio_iov = aiov;
1286     aioio.uio_iocnt = 1;
1287     aioio.uio_loffset = offset;
1288     aioio.uio_segflg = UIO_SYSSPACE;
1289     aioio.uio_fmode = 0;
1290     aioio.uio_resid = count;

1291     /* Shared lock for n_fid use in smb_rwuio */
1292     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER,
1293                           smi->smi_flags & SMI_INT))
1294         return (EINTR);
1295     smb_creditinit(&scred, cr);

1296     DTRACE_IOL(start, struct buf *, bp);

1297     if (bp->b_flags & B_READ) {

1298         /* After reconnect, n_fid is invalid */
1299         if (np->n_vcgenid != ssp->ss_vcgenid)
1300             error = ESTALE;
1301         else
1302             error = smb_rwuio(ssp, np->n_fid, UIO_READ,
1303                               &aioio, &scred, smb_timo_read);

1304         /* Like NFS, only set b_error here. */
1305         bp->b_error = error;
1306         bp->b_resid = aioio.uio_resid;

1307         if (!error && aioio.uio_resid != 0)
1308             error = EIO;
1309         if (!error && past_eof != 0) {
1310             /* Zero the memory beyond EOF. */
1311             bzero(bp->b_un.b_addr + count, past_eof);
1312         }
1313     } else {

1314         /* After reconnect, n_fid is invalid */
1315         if (np->n_vcgenid != ssp->ss_vcgenid)
1316             error = ESTALE;
1317         else
1318             error = smb_rwuio(ssp, np->n_fid, UIO_WRITE,
1319                               &aioio, &scred, smb_timo_write);

1320         /* Like NFS, only set b_error here. */
1321         bp->b_error = error;
1322         bp->b_resid = aioio.uio_resid;

1323         if (!error && aioio.uio_resid != 0)
1324             error = EIO;
1325         if (!error && sync) {
1326             (void) smbfs_smb_flush(np, &scred);
1327         }
1328     }

1329     /*
1330      * This comes from nfs3_commit()
1331      */
1332     if (error != 0) {
1333         mutex_enter(&np->r_statelock);
1334         if (error == ESTALE)
1335             np->r_flags |= RSTALE;

```

```

1347             if (!np->r_error)
1348                 np->r_error = error;
1349             mutex_exit(&np->r_statelock);
1350             bp->b_flags |= B_ERROR;
1351         }

1352         DTRACE_IOL(done, struct buf *, bp);

1353         smb_credrele(&scred);
1354         smbfs_rw_exit(&np->r_lkserlock);

1355         if (error == ESTALE)
1356             smbfs_attrcache_remove(np);

1357         return (error);
1358     }

1359     /*
1360      * Here NFS has: nfs3write, nfs3read
1361      * We use smb_rwuio instead.
1362     */

1363     /*
1364      * Here NFS has: nfs3write, nfs3read
1365      * We use smb_rwuio instead.
1366     */

1367     /* ARGUSED */
1368     static int
1369     smbfs_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag,
1370                 cred_t *cr, int *rvalp, caller_context_t *ct)
1371     {
1372         int error;
1373         smbmntinfo_t *smi;
1374
1375         smi = VTOSMI(vp);
1376
1377         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1378             return (EIO);
1379
1380         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1381             return (EIO);
1382
1383         switch (cmd) {
1384             case _FIOREAD:
1385                 /* First three from ZFS. XXX - need these? */
1386
1387                 error = smbfs_fsync(vp, 0, cr, ct);
1388                 break;
1389
1390             /*
1391              * The following two ioctls are used by bfu.
1392              * Silently ignore to avoid bfu errors.
1393              */
1394             case _FIODIO:
1395             case _FIOSDIO:
1396                 error = 0;
1397                 break;
1398
1399             #if 0 /* Todo - SMB ioctl query regions */
1400             case _FIO_SEEK_DATA:
1401             case _FIO_SEEK_HOLE:
1402                 break;
1403             #endif
1404
1405             #ifdef NOT_YET /* XXX - from the NFS code. */
1406             case _FIODIRECTIO:
1407                 error = smbfs_directio(vp, (int)arg, cr);
1408                 break;
1409             #endif
1410
1411             /*
1412              * The following two ioctls are used by bfu.
1413              * Silently ignore to avoid bfu errors.
1414              */
1415             case _FIOSQDIO:
1416                 error = 0;
1417                 break;
1418             #endif
1419
1420             /*
1421              * The following two ioctls are used by bfu.
1422              * Silently ignore to avoid bfu errors.
1423              */
1424             case _FIOSQDIO:
1425                 error = 0;
1426                 break;
1427             #endif
1428
1429             /*
1430              * The following two ioctls are used by bfu.
1431              * Silently ignore to avoid bfu errors.
1432              */
1433             case _FIOSQDIO:
1434                 error = 0;
1435                 break;
1436             #endif
1437
1438             /*
1439              * The following two ioctls are used by bfu.
1440              * Silently ignore to avoid bfu errors.
1441              */
1442             case _FIOSQDIO:
1443                 error = 0;
1444                 break;
1445             #endif
1446
1447             /*
1448              * The following two ioctls are used by bfu.
1449              * Silently ignore to avoid bfu errors.
1450              */
1451             case _FIOSQDIO:
1452                 error = 0;
1453                 break;
1454             #endif
1455
1456             /*
1457              * The following two ioctls are used by bfu.
1458              * Silently ignore to avoid bfu errors.
1459              */
1460             case _FIOSQDIO:
1461                 error = 0;
1462                 break;
1463             #endif
1464
1465             /*
1466              * The following two ioctls are used by bfu.
1467              * Silently ignore to avoid bfu errors.
1468              */
1469             case _FIOSQDIO:
1470                 error = 0;
1471                 break;
1472             #endif
1473
1474             /*
1475              * The following two ioctls are used by bfu.
1476              * Silently ignore to avoid bfu errors.
1477              */
1478             case _FIOSQDIO:
1479                 error = 0;
1480                 break;
1481             #endif
1482
1483             /*
1484              * The following two ioctls are used by bfu.
1485              * Silently ignore to avoid bfu errors.
1486              */
1487             case _FIOSQDIO:
1488                 error = 0;
1489                 break;
1490             #endif
1491
1492             /*
1493              * The following two ioctls are used by bfu.
1494              * Silently ignore to avoid bfu errors.
1495              */
1496             case _FIOSQDIO:
1497                 error = 0;
1498                 break;
1499             #endif
1500
1501             /*
1502              * The following two ioctls are used by bfu.
1503              * Silently ignore to avoid bfu errors.
1504              */
1505             case _FIOSQDIO:
1506                 error = 0;
1507                 break;
1508             #endif
1509
1510             /*
1511              * The following two ioctls are used by bfu.
1512              * Silently ignore to avoid bfu errors.
1513              */
1514             case _FIOSQDIO:
1515                 error = 0;
1516                 break;
1517             #endif
1518
1519             /*
1520              * The following two ioctls are used by bfu.
1521              * Silently ignore to avoid bfu errors.
1522              */
1523             case _FIOSQDIO:
1524                 error = 0;
1525                 break;
1526             #endif
1527
1528             /*
1529              * The following two ioctls are used by bfu.
1530              * Silently ignore to avoid bfu errors.
1531              */
1532             case _FIOSQDIO:
1533                 error = 0;
1534                 break;
1535             #endif
1536
1537             /*
1538              * The following two ioctls are used by bfu.
1539              * Silently ignore to avoid bfu errors.
1540              */
1541             case _FIOSQDIO:
1542                 error = 0;
1543                 break;
1544             #endif
1545
1546             /*
1547              * The following two ioctls are used by bfu.
1548              * Silently ignore to avoid bfu errors.
1549              */
1550             case _FIOSQDIO:
1551                 error = 0;
1552                 break;
1553             #endif
1554
1555             /*
1556              * The following two ioctls are used by bfu.
1557              * Silently ignore to avoid bfu errors.
1558              */
1559             case _FIOSQDIO:
1560                 error = 0;
1561                 break;
1562             #endif
1563
1564             /*
1565              * The following two ioctls are used by bfu.
1566              * Silently ignore to avoid bfu errors.
1567              */
1568             case _FIOSQDIO:
1569                 error = 0;
1570                 break;
1571             #endif
1572
1573             /*
1574              * The following two ioctls are used by bfu.
1575              * Silently ignore to avoid bfu errors.
1576              */
1577             case _FIOSQDIO:
1578                 error = 0;
1579                 break;
1580             #endif
1581
1582             /*
1583              * The following two ioctls are used by bfu.
1584              * Silently ignore to avoid bfu errors.
1585              */
1586             case _FIOSQDIO:
1587                 error = 0;
1588                 break;
1589             #endif
1590
1591             /*
1592              * The following two ioctls are used by bfu.
1593              * Silently ignore to avoid bfu errors.
1594              */
1595             case _FIOSQDIO:
1596                 error = 0;
1597                 break;
1598             #endif
1599
1600             /*
1601              * The following two ioctls are used by bfu.
1602              * Silently ignore to avoid bfu errors.
1603              */
1604             case _FIOSQDIO:
1605                 error = 0;
1606                 break;
1607             #endif
1608
1609             /*
1610              * The following two ioctls are used by bfu.
1611              * Silently ignore to avoid bfu errors.
1612              */
1613             case _FIOSQDIO:
1614                 error = 0;
1615                 break;
1616             #endif
1617
1618             /*
1619              * The following two ioctls are used by bfu.
1620              * Silently ignore to avoid bfu errors.
1621              */
1622             case _FIOSQDIO:
1623                 error = 0;
1624                 break;
1625             #endif
1626
1627             /*
1628              * The following two ioctls are used by bfu.
1629              * Silently ignore to avoid bfu errors.
1630              */
1631             case _FIOSQDIO:
1632                 error = 0;
1633                 break;
1634             #endif
1635
1636             /*
1637              * The following two ioctls are used by bfu.
1638              * Silently ignore to avoid bfu errors.
1639              */
1640             case _FIOSQDIO:
1641                 error = 0;
1642                 break;
1643             #endif
1644
1645             /*
1646              * The following two ioctls are used by bfu.
1647              * Silently ignore to avoid bfu errors.
1648              */
1649             case _FIOSQDIO:
1650                 error = 0;
1651                 break;
1652             #endif
1653
1654             /*
1655              * The following two ioctls are used by bfu.
1656              * Silently ignore to avoid bfu errors.
1657              */
1658             case _FIOSQDIO:
1659                 error = 0;
1660                 break;
1661             #endif
1662
1663             /*
1664              * The following two ioctls are used by bfu.
1665              * Silently ignore to avoid bfu errors.
1666              */
1667             case _FIOSQDIO:
1668                 error = 0;
1669                 break;
1670             #endif
1671
1672             /*
1673              * The following two ioctls are used by bfu.
1674              * Silently ignore to avoid bfu errors.
1675              */
1676             case _FIOSQDIO:
1677                 error = 0;
1678                 break;
1679             #endif
1680
1681             /*
1682              * The following two ioctls are used by bfu.
1683              * Silently ignore to avoid bfu errors.
1684              */
1685             case _FIOSQDIO:
1686                 error = 0;
1687                 break;
1688             #endif
1689
1690             /*
1691              * The following two ioctls are used by bfu.
1692              * Silently ignore to avoid bfu errors.
1693              */
1694             case _FIOSQDIO:
1695                 error = 0;
1696                 break;
1697             #endif
1698
1699             /*
1700              * The following two ioctls are used by bfu.
1701              * Silently ignore to avoid bfu errors.
1702              */
1703             case _FIOSQDIO:
1704                 error = 0;
1705                 break;
1706             #endif
1707
1708             /*
1709              * The following two ioctls are used by bfu.
1710              * Silently ignore to avoid bfu errors.
1711              */
1712             case _FIOSQDIO:
1713                 error = 0;
1714                 break;
1715             #endif
1716
1717             /*
1718              * The following two ioctls are used by bfu.
1719              * Silently ignore to avoid bfu errors.
1720              */
1721             case _FIOSQDIO:
1722                 error = 0;
1723                 break;
1724             #endif
1725
1726             /*
1727              * The following two ioctls are used by bfu.
1728              * Silently ignore to avoid bfu errors.
1729              */
1730             case _FIOSQDIO:
1731                 error = 0;
1732                 break;
1733             #endif
1734
1735             /*
1736              * The following two ioctls are used by bfu.
1737              * Silently ignore to avoid bfu errors.
1738              */
1739             case _FIOSQDIO:
1740                 error = 0;
1741                 break;
1742             #endif
1743
1744             /*
1745              * The following two ioctls are used by bfu.
1746              * Silently ignore to avoid bfu errors.
1747              */
1748             case _FIOSQDIO:
1749                 error = 0;
1750                 break;
1751             #endif
1752
1753             /*
1754              * The following two ioctls are used by bfu.
1755              * Silently ignore to avoid bfu errors.
1756              */
1757             case _FIOSQDIO:
1758                 error = 0;
1759                 break;
1760             #endif
1761
1762             /*
1763              * The following two ioctls are used by bfu.
1764              * Silently ignore to avoid bfu errors.
1765              */
1766             case _FIOSQDIO:
1767                 error = 0;
1768                 break;
1769             #endif
1770
1771             /*
1772              * The following two ioctls are used by bfu.
1773              * Silently ignore to avoid bfu errors.
1774              */
1775             case _FIOSQDIO:
1776                 error = 0;
1777                 break;
1778             #endif
1779
1780             /*
1781              * The following two ioctls are used by bfu.
1782              * Silently ignore to avoid bfu errors.
1783              */
1784             case _FIOSQDIO:
1785                 error = 0;
1786                 break;
1787             #endif
1788
1789             /*
1790              * The following two ioctls are used by bfu.
1791              * Silently ignore to avoid bfu errors.
1792              */
1793             case _FIOSQDIO:
1794                 error = 0;
1795                 break;
1796             #endif
1797
1798             /*
1799              * The following two ioctls are used by bfu.
1800              * Silently ignore to avoid bfu errors.
1801              */
1802             case _FIOSQDIO:
1803                 error = 0;
1804                 break;
1805             #endif
1806
1807             /*
1808              * The following two ioctls are used by bfu.
1809              * Silently ignore to avoid bfu errors.
1810              */
1811             case _FIOSQDIO:
1812                 error = 0;
1813                 break;
1814             #endif
1815
1816             /*
1817              * The following two ioctls are used by bfu.
1818              * Silently ignore to avoid bfu errors.
1819              */
1820             case _FIOSQDIO:
1821                 error = 0;
1822                 break;
1823             #endif
1824
1825             /*
1826              * The following two ioctls are used by bfu.
1827              * Silently ignore to avoid bfu errors.
1828              */
1829             case _FIOSQDIO:
1830                 error = 0;
1831                 break;
1832             #endif
1833
1834             /*
1835              * The following two ioctls are used by bfu.
1836              * Silently ignore to avoid bfu errors.
1837              */
1838             case _FIOSQDIO:
1839                 error = 0;
1840                 break;
1841             #endif
1842
1843             /*
1844              * The following two ioctls are used by bfu.
1845              * Silently ignore to avoid bfu errors.
1846              */
1847             case _FIOSQDIO:
1848                 error = 0;
1849                 break;
1850             #endif
1851
1852             /*
1853              * The following two ioctls are used by bfu.
1854              * Silently ignore to avoid bfu errors.
1855              */
1856             case _FIOSQDIO:
1857                 error = 0;
1858                 break;
1859             #endif
1860
1861             /*
1862              * The following two ioctls are used by bfu.
1863              * Silently ignore to avoid bfu errors.
1864              */
1865             case _FIOSQDIO:
1866                 error = 0;
1867                 break;
1868             #endif
1869
1870             /*
1871              * The following two ioctls are used by bfu.
1872              * Silently ignore to avoid bfu errors.
1873              */
1874             case _FIOSQDIO:
1875                 error = 0;
1876                 break;
1877             #endif
1878
1879             /*
1880              * The following two ioctls are used by bfu.
1881              * Silently ignore to avoid bfu errors.
1882              */
1883             case _FIOSQDIO:
1884                 error = 0;
1885                 break;
1886             #endif
1887
1888             /*
1889              * The following two ioctls are used by bfu.
1890              * Silently ignore to avoid bfu errors.
1891              */
1892             case _FIOSQDIO:
1893                 error = 0;
1894                 break;
1895             #endif
1896
1897             /*
1898              * The following two ioctls are used by bfu.
1899              * Silently ignore to avoid bfu errors.
1900              */
1901             case _FIOSQDIO:
1902                 error = 0;
1903                 break;
1904             #endif
1905
1906             /*
1907              * The following two ioctls are used by bfu.
1908              * Silently ignore to avoid bfu errors.
1909              */
1910             case _FIOSQDIO:
1911                 error = 0;
1912                 break;
1913             #endif
1914
1915             /*
1916              * The following two ioctls are used by bfu.
1917              * Silently ignore to avoid bfu errors.
1918              */
1919             case _FIOSQDIO:
1920                 error = 0;
1921                 break;
1922             #endif
1923
1924             /*
1925              * The following two ioctls are used by bfu.
1926              * Silently ignore to avoid bfu errors.
1927              */
1928             case _FIOSQDIO:
1929                 error = 0;
1930                 break;
1931             #endif
1932
1933             /*
1934              * The following two ioctls are used by bfu.
1935              * Silently ignore to avoid bfu errors.
1936              */
1937             case _FIOSQDIO:
1938                 error = 0;
1939                 break;
1940             #endif
1941
1942             /*
1943              * The following two ioctls are used by bfu.
1944              * Silently ignore to avoid bfu errors.
1945              */
1946             case _FIOSQDIO:
1947                 error = 0;
1948                 break;
1949             #endif
1950
1951             /*
1952              * The following two ioctls are used by bfu.
1953              * Silently ignore to avoid bfu errors.
1954              */
1955             case _FIOSQDIO:
1956                 error = 0;
1957                 break;
1958             #endif
1959
1960             /*
1961              * The following two ioctls are used by bfu.
1962              * Silently ignore to avoid bfu errors.
1963              */
1964             case _FIOSQDIO:
1965                 error = 0;
1966                 break;
1967             #endif
1968
1969             /*
1970              * The following two ioctls are used by bfu.
1971              * Silently ignore to avoid bfu errors.
1972              */
1973             case _FIOSQDIO:
1974                 error = 0;
1975                 break;
1976             #endif
1977
1978             /*
1979              * The following two ioctls are used by bfu.
1980              * Silently ignore to avoid bfu errors.
1981              */
1982             case _FIOSQDIO:
1983                 error = 0;
1984                 break;
1985             #endif
1986
1987             /*
1988              * The following two ioctls are used by bfu.
1989              * Silently ignore to avoid bfu errors.
1990              */
1991             case _FIOSQDIO:
1992                 error = 0;
1993                 break;
1994             #endif
1995
1996             /*
1997              * The following two ioctls are used by bfu.
1998              * Silently ignore to avoid bfu errors.
1999              */
1999             case _FIOSQDIO:
2000                 error = 0;
2001                 break;
2002             #endif
2003
2004             /*
2005              * The following two ioctls are used by bfu.
2006              * Silently ignore to avoid bfu errors.
2007              */
2008             case _FIOSQDIO:
2009                 error = 0;
2010                 break;
2011             #endif
2012
2013             /*
2014              * The following two ioctls are used by bfu.
2015              * Silently ignore to avoid bfu errors.
2016              */
2017             case _FIOSQDIO:
2018                 error = 0;
2019                 break;
2020             #endif
2021
2022             /*
2023              * The following two ioctls are used by bfu.
2024              * Silently ignore to avoid bfu errors.
2025              */
2026             case _FIOSQDIO:
2027                 error = 0;
2028                 break;
2029             #endif
2030
2031             /*
2032              * The following two ioctls are used by bfu.
2033              * Silently ignore to avoid bfu errors.
2034              */
2035             case _FIOSQDIO:
2036                 error = 0;
2037                 break;
2038             #endif
2039
2040             /*
2041              * The following two ioctls are used by bfu.
2042              * Silently ignore to avoid bfu errors.
2043              */
2044             case _FIOSQDIO:
2045                 error = 0;
2046                 break;
2047             #endif
2048
2049             /*
2050              * The following two ioctls are used by bfu.
2051              * Silently ignore to avoid bfu errors.
2052              */
2053             case _FIOSQDIO:
2054                 error = 0;
2055                 break;
2056             #endif
2057
2058             /*
2059              * The following two ioctls are used by bfu.
2060              * Silently ignore to avoid bfu errors.
2061              */
2062             case _FIOSQDIO:
2063                 error = 0;
2064                 break;
2065             #endif
2066
2067             /*
2068              * The following two ioctls are used by bfu.
2069              * Silently ignore to avoid bfu errors.
2070              */
2071             case _FIOSQDIO:
2072                 error = 0;
2073                 break;
2074             #endif
2075
2076             /*
2077              * The following two ioctls are used by bfu.
2078              * Silently ignore to avoid bfu errors.
2079              */
2079             case _FIOSQDIO:
2080                 error = 0;
2081                 break;
2082             #endif
2083
2084             /*
2085              * The following two ioctls are used by bfu.
2086              * Silently ignore to avoid bfu errors.
2087              */
2087             case _FIOSQDIO:
2088                 error = 0;
2089                 break;
2090             #endif
2091
2092             /*
2093              * The following two ioctls are used by bfu.
2094              * Silently ignore to avoid bfu errors.
2095              */
2095             case _FIOSQDIO:
2096                 error = 0;
2097                 break;
2098             #endif
2099
2100             /*
2101              * The following two ioctls are used by bfu.
2102              * Silently ignore to avoid bfu errors.
2103              */
2103             case _FIOSQDIO:
2104                 error = 0;
2105                 break;
2106             #endif
2107
2108             /*
2109              * The following two ioctls are used by bfu.
2110              * Silently ignore to avoid bfu errors.
2111              */
2111             case _FIOSQDIO:
2112                 error = 0;
2113                 break;
2114             #endif
2115
2116             /*
2117              * The following two ioctls are used by bfu.
2118              * Silently ignore to avoid bfu errors.
2119              */
2119             case _FIOSQDIO:
2120                 error = 0;
2121                 break;
2122             #endif
2123
2124             /*
2125              * The following two ioctls are used by bfu.
2126              * Silently ignore to avoid bfu errors.
2127              */
2127             case _FIOSQDIO:
2128                 error = 0;
2129                 break;
2130             #endif
2131
2132             /*
2133              * The following two ioctls are used by bfu.
2134              * Silently ignore to avoid bfu errors.
2135              */
2135             case _FIOSQDIO:
2136                 error = 0;
2137                 break;
2138             #endif
2139
2140             /*
2141              * The following two ioctls are used by bfu.
2142              * Silently ignore to avoid bfu errors.
2143              */
2143             case _FIOSQDIO:
2144                 error = 0;
2145                 break;
2146             #endif
2147
2148             /*
2149              * The following two ioctls are used by bfu.
2150              * Silently ignore to avoid bfu errors.
2151              */
2151             case _FIOSQDIO:
2152                 error = 0;
2153                 break;
2154             #endif
2155
2156             /*
2157              * The following two ioctls are used by bfu.
2158              * Silently ignore to avoid bfu errors.
2159              */
2159             case _FIOSQDIO:
2160                 error = 0;
2161                 break;
2162             #endif
2163
2164             /*
2165              * The following two ioctls are used by bfu.
2166              * Silently ignore to avoid bfu errors.
2167              */
2167             case _FIOSQDIO:
2168                 error = 0;
2169                 break;
2170             #endif
2171
2172             /*
2173              * The following two ioctls are used by bfu.
2174              * Silently ignore to avoid bfu errors.
2175              */
2175             case _FIOSQDIO:
2176                 error = 0;
2177                 break;
2178             #endif
2179
2180             /*
2181              * The following two ioctls are used by bfu.
2182              * Silently ignore to avoid bfu errors.
2183              */
2183             case _FIOSQDIO:
2184                 error = 0;
2185                 break;
2186             #endif
2187
2188             /*
2189              * The following two ioctls are used by bfu.
2190              * Silently ignore to avoid bfu errors.
2191              */
2191             case _FIOSQDIO:
2192                 error = 0;
2193                 break;
2194             #endif
2195
2196             /*
2197              * The following two ioctls are used by bfu.
2198              * Silently ignore to avoid bfu errors.
2199              */
2199             case _FIOSQDIO:
2200                 error = 0;
2201                 break;
2202             #endif
2203
2204             /*
2205              * The following two ioctls are used by bfu.
2206              * Silently ignore to avoid bfu errors.
2207              */
2207             case _FIOSQDIO:
2208                 error = 0;
2209                 break;
2210             #endif
2211
2212             /*
2213              * The following two ioctls are used by bfu.
2214              * Silently ignore to avoid bfu errors.
2215              */
2215             case _FIOSQDIO:
2216                 error = 0;
2217                 break;
2218             #endif
2219
2220             /*
2221              * The following two ioctls are used by bfu.
2222              * Silently ignore to avoid bfu errors.
2223              */
2223             case _FIOSQDIO:
2224                 error = 0;
2225                 break;
2226             #endif
2227
2228             /*
2229              * The following two ioctls are used by bfu.
2230              * Silently ignore to avoid bfu errors.
2231              */
2231             case _FIOSQDIO:
2232                 error = 0;
2233                 break;
2234             #endif
2235
2236             /*
2237              * The following two ioctls are used by bfu.
2238              * Silently ignore to avoid bfu errors.
2239              */
2239             case _FIOSQDIO:
2240                 error = 0;
2241                 break;
2242             #endif
2243
2244             /*
2245              * The following two ioctls are used by bfu.
2246              * Silently ignore to avoid bfu errors.
2247              */
2247             case _FIOSQDIO:
2248                 error = 0;
2249                 break;
2250             #endif
2251
2252             /*
2253              * The following two ioctls are used by bfu.
2254              * Silently ignore to avoid bfu errors.
2255              */
2255             case _FIOSQDIO:
2256                 error = 0;
2257                 break;
2258             #endif
2259
2260             /*
2261              * The following two ioctls are used by bfu.
2262              * Silently ignore to avoid bfu errors.
2263              */
2263             case _FIOSQDIO:
2264                 error = 0;
2265                 break;
2266             #endif
2267
2268             /*
2269              * The following two ioctls are used by bfu.
2270              * Silently ignore to avoid bfu errors.
2271              */
2271             case _FIOSQDIO:
2272                 error = 0;
2273                 break;
2274             #endif
2275
2276             /*
2277              * The following two ioctls are used by bfu.
2278              * Silently ignore to avoid bfu errors.
2279              */
2279             case _FIOSQDIO:
2280                 error = 0;
2281                 break;
2282             #endif
2283
2284             /*
2285              * The following two ioctls are used by bfu.
2286              * Silently ignore to avoid bfu errors.
2287              */
2287             case _FIOSQDIO:
2288                 error = 0;
2289                 break;
2290             #endif
2291
2292             /*
2293              * The following two ioctls are used by bfu.
2294              * Silently ignore to avoid bfu errors.
2295              */
2295             case _FIOSQDIO:
2296                 error = 0;
2297                 break;
2298             #endif
2299
2300             /*
2301              * The following two ioctls are used by bfu.
2302              * Silently ignore to avoid bfu errors.
2303              */
2303             case _FIOSQDIO:
2304                 error = 0;
2305                 break;
2306             #endif
2307
2308             /*
2309              * The following two ioctls are used by bfu.
2310              * Silently ignore to avoid bfu errors.
2311              */
2311             case _FIOSQDIO:
2312                 error = 0;
2313                 break;
2314             #endif
2315
2316             /*
2317              * The following two ioctls are used by bfu.
2318              * Silently ignore to avoid bfu errors.
2319              */
2319             case _FIOSQDIO:
2320                 error = 0;
2321                 break;
2322             #endif
2323
2324             /*
2325              * The following two ioctls are used by bfu.
2326              * Silently ignore to avoid bfu errors.
2327              */
2327             case _FIOSQDIO:
2328                 error = 0;
2329                 break;
2330             #endif
2331
2332             /*
2333              * The following two ioctls are used by bfu.
2334              * Silently ignore to avoid bfu errors.
2335              */
2335             case _FIOSQDIO:
2336                 error = 0;
2337                 break;
2338             #endif
2339
2340             /*
2341              * The following two ioctls are used by bfu.
2342              * Silently ignore to avoid bfu errors.
2343              */
2343             case _FIOSQDIO:
2344                 error = 0;
2345                 break;
2346             #endif
2347
2348             /*
2349              * The following two ioctls are used by bfu.
2350              * Silently ignore to avoid bfu errors.
2351              */
2351            
```

```

1410             * Allow get/set with "raw" security descriptor (SD) data.
1411             * Useful for testing, diagnosing idmap problems, etc.
1412             */
1413     case SMBFSIO_GETSD:
1414         error = smbfs_acl_iocget(vp, arg, flag, cr);
1415         break;
1416
1417     case SMBFSIO_SETSD:
1418         error = smbfs_acl_iocset(vp, arg, flag, cr);
1419         break;
1420
1421     default:
1422         error = ENOTTY;
1423         break;
1424     }
1425
1426     return (error);
1427 }

1430 /*
1431  * Return either cached or remote attributes. If get remote attr
1432  * use them to check and invalidate caches, then cache the new attributes.
1433 */
1434 /* ARGSUSED */
1435 static int
1436 smbfs_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1437                 caller_context_t *ct)
1438 {
1439     smbnode_t *np;
1440     smbmtinfo_t *smi;
1441     int error;
1442
1443     smi = VTOSMI(vp);
1444
1445     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1446         return (EIO);
1447
1448     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1449         return (EIO);
1450
1451     /*
1452      * If it has been specified that the return value will
1453      * just be used as a hint, and we are only being asked
1454      * for size, fsid or rdev, then return the client's
1455      * notion of these values without checking to make sure
1456      * that the attribute cache is up to date.
1457      * The whole point is to avoid an over the wire GETATTR
1458      * call.
1459     */
1460     np = VTOSMB(vp);
1461     if (flags & ATTR_HINT) {
1462         if (vap->va_mask ==
1463             (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
1464             mutex_enter(&np->r_statelock);
1465             if (vap->va_mask | AT_SIZE)
1466                 vap->va_size = np->r_size;
1467             if (vap->va_mask | AT_FSID)
1468                 vap->va_fsid = vp->v_vfsp->vfs_dev;
1469             if (vap->va_mask | AT_RDEV)
1470                 vap->va_rdev = vp->v_rdev;
1471             mutex_exit(&np->r_statelock);
1472             return (0);
1473         }
1474     }

```

```

1476     /*
1477      * Only need to flush pages if asking for the mtime
1478      * and if there any dirty pages.
1479      */
1480     /*
1481      * Here NFS also checks for async writes (np->r_awcount)
1482      */
1483     if (vap->va_mask & AT_MTIME) {
1484         if (vn_has_cached_data(vp) &&
1485             ((np->r_flags & RDIRTY) != 0)) {
1486             mutex_enter(&np->r_statelock);
1487             np->r_gcount++;
1488             mutex_exit(&np->r_statelock);
1489             error = smbfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
1490             mutex_enter(&np->r_statelock);
1491             if (error && (error == ENOSPC || error == EDQUOT)) {
1492                 if (!np->r_error)
1493                     np->r_error = error;
1494             }
1495             if (--np->r_gcount == 0)
1496                 cv_broadcast(&np->r_cv);
1497         }
1498     }
1499
1500     return (smbfsgetattr(vp, vap, cr));
1501 }

1502 /* smbfsgetattr() in smbfs_client.c */

1503 /*ARGSUSED4*/
1504 static int
1505 smbfs_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1506                 caller_context_t *ct)
1507 {
1508     vfs_t           *vfsp;
1509     smbmtinfo_t    *smi;
1510     int              error;
1511     uint_t          mask;
1512     struct vattr   oldva;
1513
1514     vfsp = vp->v_vfsp;
1515     smi = VFTOSMI(vfsp);
1516
1517     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1518         return (EIO);
1519
1520     if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
1521         return (EIO);
1522
1523     mask = vap->va_mask;
1524     if (mask & AT_NOSET)
1525         return (EINVAL);
1526
1527     if (vfsp->vfs_flag & VFS_RDONLY)
1528         return (EROFS);
1529
1530     /*
1531      * This is a _local_ access check so that only the owner of
1532      * this mount can set attributes. With ACLs enabled, the
1533      * file owner can be different from the mount owner, and we
1534      * need to check the _mount_ owner here. See _access_rwx
1535      */
1536     bzero(&oldva, sizeof (oldva));
1537     oldva.va_mask = AT_TYPE | AT_MODE;
1538     error = smbfsgetattr(vp, &oldva, cr);
1539     if (error)
1540

```

```

1542         return (error);
1543     oldva.va_mask |= AT_UID | AT_GID;
1544     oldva.va_uid = smi->smi_uid;
1545     oldva.va_gid = smi->smi_gid;
1546
1547     error = secpolicy vnode_setattr(cr, vp, vap, &oldva, flags,
1548                                     smbfs_accessx, vp);
1549     if (error)
1550         return (error);
1551
1552     if (mask & (AT_UID | AT_GID)) {
1553         if (smi->smi_flags & SMI_ACL)
1554             error = smbfs_acl_setids(vp, vap, cr);
1555         else
1556             error = ENOSYS;
1557         if (error != 0) {
1558             SMBVDEBUG("error %d setting UID/GID on %s",
1559                       error, VTOSMB(vp)->n_rpath);
1560
1561         /* It might be more correct to return the
1562         * error here, but that causes complaints
1563         * when root extracts a cpio archive, etc.
1564         * So ignore this error, and go ahead with
1565         * the rest of the setattr work.
1566         */
1567     }
1568 }
1569
1570     error = smbfssetattr(vp, vap, flags, cr);
1571
1572 #ifdef SMBFS_VNEVENT
1573     if (error == 0 && (vap->va_mask & AT_SIZE) && vap->va_size == 0)
1574         vnevent_truncate(vp, ct);
1575 #endif
1576
1577     return (error);
1578     return (smbfssetattr(vp, vap, flags, cr));
1579 }
1580
1581 /* Mostly from Darwin smbfs_setattr()
1582 * but then modified a lot.
1583 */
1584 /* ARGSUSED */
1585 static int
1586 smbfssetattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr)
1587 {
1588     int             error = 0;
1589     smbnode_t       np = VTOSMB(vp);
1590     uint_t          mask = vap->va_mask;
1591     struct timespec *mtime, *atime;
1592     struct smb_cred scred;
1593     int             error, modified = 0;
1594     unsigned short  fid;
1595     int             have_fid = 0;
1596     uint32_t        rights = 0;
1597     uint32_t        dosattr = 0;
1598
1599     ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);
1600
1601     /*
1602     * There are no settable attributes on the XATTR dir,
1603     * so just silently ignore these. On XATTR files,
1604     * you can set the size but nothing else.
1605     */
1606     if ((vp->v_flag & V_XATTRDIR)

```

```

1607             return (0);
1608         if (np->n_flag & N_XATTR) {
1609             if (mask & AT_TIMES)
1610                 SMBVDEBUG("ignore set time on xattr\n");
1611             mask &= AT_SIZE;
1612         }
1613
1614         /*
1615         * Only need to flush pages if there are any pages and
1616         * if the file is marked as dirty in some fashion. The
1617         * file must be flushed so that we can accurately
1618         * determine the size of the file and the cached data
1619         * after the SETATTR returns. A file is considered to
1620         * be dirty if it is either marked with RDIRTY, has
1621         * outstanding i/o's active, or is mmap'd. In this
1622         * last case, we can't tell whether there are dirty
1623         * pages, so we flush just to be sure.
1624         */
1625         if (vn_has_cached_data(vp) &&
1626             ((np->r_flags & RDIRTY) ||
1627              np->r_count > 0 ||
1628              np->r_mapcnt > 0)) {
1629             ASSERT(vp->v_type != VCHR);
1630             error = smbfs_putpage(vp, (offset_t)0, 0, 0, cr, NULL);
1631             if (error && (error == ENOSPC || error == EDQUOT)) {
1632                 mutex_enter(&np->r_statelock);
1633                 if (!np->r_error)
1634                     np->r_error = error;
1635                 mutex_exit(&np->r_statelock);
1636             }
1637         }
1638
1639         /*
1640         * If our caller is trying to set multiple attributes, they
1641         * can make no assumption about what order they are done in.
1642         * Here we try to do them in order of decreasing likelihood
1643         * of failure, just to minimize the chance we'll wind up
1644         * with a partially complete request.
1645         */
1646
1647         /* Shared lock for (possible) n_fid use. */
1648         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
1649             return (EINTR);
1650         smb_credinit(&scred, cr);
1651
1652         /*
1653         * If the caller has provided extensible attributes,
1654         * map those into DOS attributes supported by SMB.
1655         * Note: zero means "no change".
1656         */
1657         if (mask & AT_XVATTR)
1658             dosattr = xvattr_to_dosattr(np, vap);
1659
1660         /*
1661         * Will we need an open handle for this setattr?
1662         * If so, what rights will we need?
1663         */
1664         if (dosattr || (mask & (AT_ATIME | AT_MTIME))) {
1665             rights |=
1666                         SA_RIGHT_FILE_WRITE_ATTRIBUTES;
1667         }
1668         if (mask & AT_SIZE) {
1669             rights |=
1670                         SA_RIGHT_FILE_WRITE_DATA |
1671                         SA_RIGHT_FILE_APPEND_DATA;
1672     }

```

```

1674     /*
1675      * Only SIZE really requires a handle, but it's
1676      * simpler and more reliable to set via a handle.
1677      * Some servers like NT4 won't set times by path.
1678      * Also, we're usually setting everything anyway.
1679      */
1680     if (rights != 0) {
1681         error = smbfs_smb_tmpopen(np, rights, &scred, &fid);
1682         if (error) {
1683             SMBVDEBUG("error %d opening %s\n",
1684                     error, np->n_rpath);
1685             goto out;
1686         }
1687         have_fid = 1;
1688     }
1689
1690     /*
1691      * If the server supports the UNIX extensions, right here is where
1692      * we'd support changes to uid, gid, mode, and possibly va_flags.
1693      * For now we claim to have made any such changes.
1694      */
1695
1696     if (mask & AT_SIZE) {
1697         /*
1698          * If the new file size is less than what the client sees as
1699          * the file size, then just change the size and invalidate
1700          * the pages.
1701          * I am commenting this code at present because the function
1702          * smbfs_putapage() is not yet implemented.
1703          */
1704
1705         /*
1706          * Set the file size to vap->va_size.
1707          */
1708         ASSERT(have_fid);
1709         error = smbfs_smb_setfilesize(np, fid, vap->va_size, &scred);
1710         if (error) {
1711             SMBVDEBUG("setsize error %d file %s\n",
1712                     error, np->n_rpath);
1713         } else {
1714             /*
1715              * Darwin had code here to zero-extend.
1716              * Tests indicate the server will zero-fill,
1717              * so looks like we don't need to do that.
1718              * so looks like we don't need to do this.
1719              * Good thing, as this could take forever.
1720              *
1721              * XXX: Reportedly, writing one byte of zero
1722              * at the end offset avoids problems here.
1723              */
1724             mutex_enter(&np->r_statelock);
1725             np->r_size = vap->va_size;
1726             mutex_exit(&np->r_statelock);
1727             modified = 1;
1728         }
1729     }
1730
1731     /*
1732      * Todo: Implement setting create_time (which is
1733      * different from ctime).
1734      * XXX: When Solaris has create_time, set that too.
1735      * Note: create_time is different from ctime.
1736      */
1737
1738     mtime = ((mask & AT_MTIME) ? &vap->va_mtime : 0);
1739     atime = ((mask & AT_ATIME) ? &vap->va_atime : 0);

```

```

1731     if (dosattr || mtime || atime) {
1732         /*
1733          * Always use the handle-based set attr call now.
1734          */
1735         ASSERT(have_fid);
1736         error = smbfs_smb_setfattr(np, fid,
1737                         dosattr, mtime, atime, &scred);
1738         if (error) {
1739             SMBVDEBUG("set times error %d file %s\n",
1740                     error, np->n_rpath);
1741         } else {
1742             modified = 1;
1743         }
1744     }
1745
1746     out:
1747     if (modified) {
1748         /*
1749          * Invalidate attribute cache in case the server
1750          * doesn't set exactly the attributes we asked.
1751          */
1752         smbfs_attrcache_remove(np);
1753
1754     if (have_fid) {
1755         error = smbfs_smb_tmpclose(np, fid, &scred);
1756         if (error)
1757             SMBVDEBUG("error %d closing %s\n",
1758                     error, np->n_rpath);
1759     }
1760
1761     smb_credrele(&scred);
1762     smbfs_rw_exit(&np->r_lkserlock);
1763
1764     if (modified) {
1765         /*
1766          * Invalidate attribute cache in case the server
1767          * doesn't set exactly the attributes we asked.
1768          */
1769         smbfs_attrcache_remove(np);
1770
1771         /*
1772          * If changing the size of the file, invalidate
1773          * any local cached data which is no longer part
1774          * of the file. We also possibly invalidate the
1775          * last page in the file. We could use
1776          * pvn_vpzero(), but this would mark the page as
1777          * modified and require it to be written back to
1778          * the server for no particularly good reason.
1779          * This way, if we access it, then we bring it
1780          * back in. A read should be cheaper than a
1781          * write.
1782          */
1783         if (mask & AT_SIZE) {
1784             smbfs_invalidate_pages(vp,
1785                         (vap->va_size & PAGEMASK), cr);
1786         }
1787
1788     return (error);
1789 }
1790
1791 unchanged_portion_omitted_
1792
1793
1794 /* */
1795 * smbfs_access_rwx()

```

```

1845 * Common function for smbfs_access, etc.
1846 *
1847 * The security model implemented by the FS is unusual
1848 * due to the current "single user mounts" restriction:
1849 * All access under a given mount point uses the CIFS
1850 * credentials established by the owner of the mount.
1851 *
1852 * Most access checking is handled by the CIFS server,
1853 * but we need sufficient Unix access checks here to
1854 * prevent other local Unix users from having access
1855 * to objects under this mount that the uid/gid/mode
1856 * settings in the mount would not allow.
1857 *
1858 * With this model, there is a case where we need the
1859 * ability to do an access check before we have the
1860 * vnode for an object. This function takes advantage
1861 * of the fact that the uid/gid/mode is per mount, and
1862 * avoids the need for a vnode.
1863 *
1864 * We still (sort of) need a vnode when we call
1865 * secpolicy vnode_access, but that only uses
1866 * the vtype field, so we can use a pair of fake
1867 * vnodes that have only v_type filled in.
1211 *
1212 * XXX: Later, add a new secpolicy_vtype_access()
1213 * that takes the vtype instead of a vnode, and
1214 * get rid of the tmp_vxxx fake vnodes below.
1868 */
1869 static int
1870 smbfs_access_rwx(vfs_t *vfsp, int vtype, int mode, cred_t *cr)
1871 {
    /* See the secpolicy call below. */
    static const vnode_t tmp_vdir = { .v_type = VDIR };
    static const vnode_t tmp_vreg = { .v_type = VREG };
1875     vattr_t        va;
1876     vnode_t       *tvp;
1877     struct smbmntinfo *smi = VFTOSMI(vfsp);
1878     int shift = 0;

1880     /*
1881      * Build our (fabricated) vnode attributes.
1882      * XXX: Could make these templates in the
1883      * per-mount struct and use them here.
1884      */
1885     bzero(&va, sizeof(va));
1886     va.va_mask = AT_TYPE | AT_MODE | AT_UID | AT_GID;
1887     va.va_type = vtype;
1888     va.va_mode = (vtype == VDIR) ?
1889         smi->smi_dmode : smi->smi_fmode;
1890     va.va_uid = smi->smi_uid;
1891     va.va_gid = smi->smi_gid;

1892     /*
1893      * Disallow write attempts on read-only file systems,
1894      * unless the file is a device or fifo node. Note:
1895      * Inline vn_is_readonly and IS_DEVVP here because
1896      * we may not have a vnode ptr. Original expr. was:
1897      * (mode & VWRITE) && vn_is_readonly(vp) && !IS_DEVVP(vp))
1898     */
1899     if ((mode & VWRITE) &&
1900         (vfsp->vfs_flag & VFS_RDONLY) &&
1901         !(vtype == VCHR || vtype == VBLK || vtype == VFIFO))
            return (EROFS);

1903     /*
1904      * Disallow attempts to access mandatory lock files.

```

```

1905             * Similarly, expand MANDLOCK here.
1255             */
1906             if ((mode & (VWRITE | VREAD | VEXEC)) &&
1907                 va.va_type == VREG && MANDMODE(va.va_mode))
1908                 return (EACCES);

1911             /*
1912              * Access check is based on only
1913              * one of owner, group, public.
1914              * If not owner, then check group.
1915              * If not a member of the group,
1916              * then check public access.
1917              */
1918             if (crgetuid(cr) != va.va_uid) {
1919                 shift += 3;
1920                 if (!groupmember(va.va_gid, cr))
1921                     shift += 3;
1922             }

1924             /*
1925              * We need a vnode for secpolicy vnode_access,
1926              * but the only thing it looks at is v_type,
1927              * so pass one of the templates above.
1928              */
1929             tvp = (va.va_type == VDIR) ?
1930                 (vnode_t *)&tmp_vdir :
1931                 (vnode_t *)&tmp_vreg;

1933             return (secpolicy vnode_access2(cr, tvp, va.va_uid,
1934                                             va.va_mode << shift, mode));
1935 }

1975 /* ARGSUSED */
1976 static int
1977 smbfs_readlink(vnode_t *vp, struct uio *uiop, cred_t *cr, caller_context_t *ct)
1978 {
1979     /* Not yet... */
1980     return (ENOSYS);
1981 }

1984 /*
1985  * Flush local dirty pages to stable storage on the server.
1986  *
1987  * If FNODSYNC is specified, then there is nothing to do because
1988  * metadata changes are not cached on the client before being
1989  * sent to the server.
1990  */
1991 /* ARGSUSED */
1992 static int
1993 smbfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
1994 {
1995     int          error = 0;
1996     smbmntinfo_t *smi;
1997     smbnode_t   *np;
1998     struct smb_cred scred;

2000     np = VTOSMB(vp);
2001     smi = VTOOSMI(vp);

2003     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2004         return (EIO);

```

```

2006     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2007         return (EIO);
2009     if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
2010         return (0);
2012     if ((syncflag & (FSYNC|FDSYNC)) == 0)
2013         return (0);
2015     error = smbfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
2016     if (error)
2017         return (error);
2019     /* Shared lock for n_fid use in _flush */
2020     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
2021         return (EINTR);
2022     smb_credinit(&scred, cr);
2024     error = smbfs_smb_flush(np, &scred);
2026     smb_credrele(&scred);
2027     smbfs_rw_exit(&np->r_lkserlock);
2029     return (error);
2030 }
2032 /*
2033 * Last reference to vnode went away.
2034 */
2035 /* ARGSUSED */
2036 static void
2037 smbfs_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
2038 {
2039     smbnode_t          *np;
2040     struct smb_cred scred;
2041     np = VTOSMB(vp);
2042     int error;
2043     /*
2044     * Don't "bail out" for VFS_UNMOUNTED here,
2045     * as we want to do cleanup, etc.
2046     * See also pcfs_inactive
2047     */
2048     np = VTOSMB(vp);
2049     /*
2050     * If this is coming from the wrong zone, we let someone in the right
2051     * zone take care of it asynchronously. We can get here due to
2052     * VN_RELSE() being called from pageout() or fsflush(). This call may
2053     * potentially turn into an expensive no-op if, for instance, v_count
2054     * gets incremented in the meantime, but it's still correct.
2055     */
2056     /*
2057     * From NFS:rinactive()
2058     *
2059     * Before freeing anything, wait until all asynchronous
2060     * activity is done on this rnode. This will allow all
2061     * asynchronous read ahead and write behind i/o's to
2062     * finish.
2063     */
2064     mutex_enter(&np->r_statelock);
2065     while (np->r_count > 0)
2066         cv_wait(&np->r_cv, &np->r_statelock);
2067     mutex_exit(&np->r_statelock);
2068 }
```

```

2070     /*
2071      * Flush and invalidate all pages associated with the vnode.
2072      */
2073     if (vn_has_cached_data(vp)) {
2074         if ((np->r_flags & RDIRTY) && !np->r_error) {
2075             error = smbfs_putpage(vp, (u_offset_t)0, 0, 0, cr, ct);
2076             if (error && (error == ENOSPC || error == EDQUOT)) {
2077                 mutex_enter(&np->r_statelock);
2078                 if (!np->r_error)
2079                     np->r_error = error;
2080                 mutex_exit(&np->r_statelock);
2081             }
2082         }
2083         smbfs_invalidate_pages(vp, (u_offset_t)0, cr);
2084     }
2085     /*
2086      * This vnode should have lost all cached data.
2087      */
2088     ASSERT(vn_has_cached_data(vp) == 0);
2089     /*
2090      * Defend against the possibility that higher-level callers
2091      * might not correctly balance open and close calls. If we
2092      * get here with open references remaining, it means there
2093      * was a missing VOP_CLOSE somewhere. If that happens, do
2094      * the close here so we don't "leak" FIDs on the server.
2095      *
2096      * Exclusive lock for modifying n_fid stuff.
2097      * Don't want this one ever interruptible.
2098      */
2099     (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
2100     smb_credinit(&scred, cr);
2101
2102     switch (np->n_ovtype) {
2103     case VNON:
2104         /* not open (OK) */
2105         break;
2106
2107     case VDIR:
2108         if (np->n_dirrefs == 0)
2109             break;
2110         SMBVDEBUG("open dir: refs %d path %s\n",
2111                  np->n_dirrefs, np->n_rpath);
2112         /* Force last close. */
2113         np->n_dirrefs = 1;
2114         smbfs_rele_fid(np, &scred);
2115         break;
2116
2117     case VREG:
2118         if (np->n_fidrefs == 0)
2119             break;
2120         SMBVDEBUG("open file: refs %d id 0x%llx path %s\n",
2121                  np->n_fidrefs, np->n_fid, np->n_rpath);
2122         /* Force last close. */
2123         np->n_fidrefs = 1;
2124         smbfs_rele_fid(np, &scred);
2125         break;
2126
2127     default:
2128         SMBVDEBUG("bad n_ovtype %d\n", np->n_ovtype);
2129         np->n_ovtype = VNON;
2130         break;
2131     }
2132
2133     smb_credrele(&scred);
2134 }
```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c

3

```

2135     smbfs_rw_exit(&np->r_lkserlock);
2137
2138     /*
2139      * XATTR directories (and the files under them) have
2140      * little value for reclaim, so just remove them from
2141      * the "hash" (AVL) as soon as they go inactive.
2142      * Note that the node may already have been removed
2143      * from the hash by smbfsremove.
2144     */
2145     if ((np->n_flag & N_XATTR) != 0 &&
2146         (np->r_flags & RHASHED) != 0)
2147         smbfs_rmhash(np);
2148
2149 }
unchanged portion omitted
2203 /* ARGSUSED */
2204 static int
2205 smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
2206               int cache_ok, caller_context_t *ct)
2207 {
2208     int             error;
2209     int             supplen; /* supported length */
2210     vnode_t        *vp;
2211     smbnode_t      *np;
2212     smbnode_t      *dnp;
2213     smbmntrinfo_t *smi;
2214     /* struct smb_vc           *vcp; */
2215     const char      *ill;
2216     const char      *name = (const char *)nm;
2217     int             nmlen = strlen(nm);
2218     int             rplen;
2219     struct smb_cred scred;
2220     struct smbfattr fa;
2221
2222     smi = VTOSMI(dvp);
2223     dnp = VTOSMB(dvp);
2224
2225     ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);
2226
2227 #ifdef NOT_YET
2228     vcp = SSTOVC(smi->smi_share);
2229
2230     /* XXX: Should compute this once and store it in smbmntrinfo */
2231     supplen = (SMB_DIALECT(vcp) >= SMB_DIALECT_LANMAN2_0) ? 256 :
2232 #else
2233     supplen = 255;
2234 #endif
2235
2236     /*
2237      * RWlock must be held, either reader or writer.
2238      * XXX: Can we check without looking directly
2239      * inside the struct smbfs_rwlock_t?
2240     */
2241     ASSERT(dnp->r_rwlock.count != 0);
2242
2243     /*
2244      * If lookup is for "", just return dvp.
2245      * No need to perform any access checks.
2246     */
2247     if (nmlen == 0) {
2248         VN_HOLD(dvp);
2249         *vpp = dvp;
2250         return (0);
2251     }

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.

```

2251      /*
2252      * Can't do lookups in non-directories.
2253      */
2254      if (dvp->v_type != VDIR)
2255          return (ENOTDIR);

2256      /*
2257      * Need search permission in the directory.
2258      */
2259      error = smbfs_access(dvp, VEXEC, 0, cr, ct);
2260      if (error)
2261          return (error);

2262      /*
2263      * If lookup is for ".", just return dvp.
2264      * Access check was done above.
2265      */
2266      if (nmllen == 1 && name[0] == '.')
2267          {
2268              VN_HOLD(dvp);
2269              *vpp = dvp;
2270              return (0);
2271          }

2272      /*
2273      * Now some sanity checks on the name.
2274      * First check the length.
2275      */
2276      if (nmllen > supplen)
2277          return (ENAMETOOLONG);

2278      /*
2279      * Avoid surprises with characters that are
2280      * illegal in Windows file names.
2281      * Todo: CATIA mappings?
2282      * Todo: CATIA mappings XXX
2283      */
2284      ill = illegal_chars;
2285      if (dnp->n_flag & N_XATTR)
2286          ill++; /* allow colon */
2287      if (strpbrk(nm, ill))
2288          return (EINVAL);

2289      /*
2290      * Special handling for lookup of "..."
2291      *
2292      * We keep full pathnames (as seen on the server)
2293      * so we can just trim off the last component to
2294      * get the full pathname of the parent. Note:
2295      * We don't actually copy and modify, but just
2296      * compute the trimmed length and pass that with
2297      * the current dir path (not null terminated).
2298      *
2299      * We don't go over-the-wire to get attributes
2300      * for "..." because we know it's a directory,
2301      * and we can just leave the rest "stale"
2302      * until someone does a getattr.
2303      */
2304      if (nmllen == 2 && name[0] == '.' && name[1] == '.')
2305          {
2306              if (dvp->v_flag & VROOT)
2307                  /*
2308                  * Already at the root. This can happen
2309                  * with directory listings at the root,
2310                  * which lookup ".." and "..." to get the
2311                  * inode numbers. Let "..." be the same
2312                  * as ".." in the FS root.
2313              }
2314          }

```

```

2315         */
2316         VN_HOLD(dvp);
2317         *vpp = dvp;
2318         return (0);
2319     }
2320
2321     /*
2322      * Special case for XATTR directory
2323      */
2324     if (dvp->v_flag & V_XATTRDIR) {
2325         error = smbfs_xa_parent(dvp, vpp);
2326         return (error);
2327     }
2328
2329     /*
2330      * Find the parent path length.
2331      */
2332     rplen = dnp->n_rplen;
2333     ASSERT(rplen > 0);
2334     while (--rplen >= 0) {
2335         if (dnp->n_rpath[rplen] == '\\\\')
2336             break;
2337     }
2338     if (rplen <= 0) {
2339         /* Found our way to the root. */
2340         vp = SMBTOV(smi->smi_root);
2341         VN_HOLD(vp);
2342         *vpp = vp;
2343         return (0);
2344     }
2345     np = smbfs_node_findcreate(smi,
2346         dnp->n_rpath, rplen, NULL, 0, 0,
2347         &smbfs_fattr0); /* force create */
2348     ASSERT(np != NULL);
2349     vp = SMBTOV(np);
2350     vp->v_type = VDIR;
2351
2352     /* Success! */
2353     *vpp = vp;
2354     return (0);
2355 }
2356
2357 /*
2358  * Normal lookup of a name under this directory.
2359  * Note we handled "", ".", ".." above.
2360  */
2361 if (cache_ok) {
2362     /*
2363      * The caller indicated that it's OK to use a
2364      * cached result for this lookup, so try to
2365      * reclaim a node from the smbfs node cache.
2366      */
2367     error = smbfslookup_cache(dvp, nm, nmlen, &vp, cr);
2368     if (error)
2369         return (error);
2370     if (vp != NULL) {
2371         /* hold taken in lookup_cache */
2372         *vpp = vp;
2373         return (0);
2374     }
2375 }
2376
2377 /*
2378  * OK, go over-the-wire to get the attributes,
2379  * then create the node.
2380 */

```

```

2381     smb_credinit(&scred, cr);
2382     /* Note: this can allocate a new "name" */
2383     error = smbfs_smb_lookup(dnp, &name, &nmlen, &fa, &scred);
2384     smb_credrele(&scred);
2385     if (error == ENOTDIR) {
2386         /*
2387          * Lookup failed because this directory was
2388          * removed or renamed by another client.
2389          * Remove any cached attributes under it.
2390          */
2391         smbfs_attrcache_remove(dnp);
2392         smbfs_attrcache_prune(dnp);
2393     }
2394     if (error)
2395         goto out;
2396
2397     error = smbfs_nget(dvp, name, nmlen, &fa, &vp);
2398     if (error)
2399         goto out;
2400
2401     /* Success! */
2402     *vpp = vp;
2403
2404 out:
2405     /* smbfs_smb_lookup may have allocated name. */
2406     if (name != nm)
2407         smbfs_name_free(name, nmlen);
2408
2409     return (error);
2410 }
2411
2412 unchanged_portion_omitted
2413
2414 /*
2415  * XXX
2416  * vsecattr_t is new to build 77, and we need to eventually support
2417  * it in order to create an ACL when an object is created.
2418  *
2419  * This op should support the new IGNORECASE flag for case-insensitive
2420  * lookups, per PSARC 2007/244.
2421  */
2422 /* ARGSUSED */
2423 static int
2424 smbfs_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
2425               int mode, vnode_t **vpp, cred_t *cr, int lfaware, caller_context_t *ct,
2426               vsecattr_t *vsecp)
2427 {
2428     int             error;
2429     int             cerror;
2430     vfs_t          *vfsp;
2431     vnode_t        *vp;
2432     smbnode_t      *np;
2433     #ifdef NOT_YET
2434     smbnode_t      *np;
2435     #endif
2436     smbnode_t      *dnp;
2437     smbmntinfo_t   *smi;
2438     struct vattr    vattr;
2439     struct smbattr  fattr;
2440     struct smb_cred scred;
2441     const char     *name = (const char *)nm;
2442     int             nmlen = strlen(nm);
2443     uint32_t        disp;
2444     uint16_t        fid;
2445     int             xattr;
2446
2447     vfsp = dvp->v_vfsp;

```

```

2537     smi = VFTOSMI(vfsp);
2538     dnp = VTOSMB(dvp);
2539     vp = NULL;
2541
2542     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2543         return (EPERM);
2544
2545     if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2546         return (EIO);
2547
2548     /*
2549      * Note: this may break mknod(2) calls to create a directory,
2550      * but that's obscure use. Some other filesystems do this.
2551      * Todo: redirect VDIR type here to _mkdir.
2552      * XXX: Later, redirect VDIR type here to _mkdir.
2553
2554     if (va->va_type != VREG)
2555         return (EINVAL);
2556
2557     /*
2558      * If the pathname is "", just use dvp, no checks.
2559      * Do this outside of the rwlock (like zfs).
2560
2561     if (nmllen == 0) {
2562         VN_HOLD(dvp);
2563         *vpp = dvp;
2564         return (0);
2565     }
2566
2567     /* Don't allow ".." or "..." through here. */
2568     if ((nmllen == 1 && name[0] == '.') ||
2569         (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2570         return (EISDIR);
2571
2572     /*
2573      * We make a copy of the attributes because the caller does not
2574      * expect us to change what va points to.
2575
2576     vatr = *va;
2577
2578     if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2579         return (EINTR);
2580     smb_credinit(&scred, cr);
2581
2582     /*
2583      * NFS needs to go over the wire, just to be sure whether the
2584      * file exists or not. Using a cached result is dangerous in
2585      * this case when making a decision regarding existence.
2586
2587      * The SMB protocol does NOT really need to go OTW here
2588      * thanks to the expressive NTCREATE disposition values.
2589      * Unfortunately, to do Unix access checks correctly,
2590      * we need to know if the object already exists.
2591      * When the object does not exist, we need VWRITE on
2592      * the directory. Note: smbfslookup() checks VEXEC.
2593
2594     error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2595     if (error == 0) {
2596         /*
2597          * The file already exists. Error?
2598          * NB: have a hold from smbfslookup
2599          */
2600         if (exclusive == EXCL) {
2601             error = EEXIST;
2602             VN_RELSE(vp);
2603             goto out;
2604
2605         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2606             return (EPERM);
2607
2608         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2609             return (EIO);
2610
2611         /*
2612          * Note: this may break mknod(2) calls to create a directory,
2613          * but that's obscure use. Some other filesystems do this.
2614          * Todo: redirect VDIR type here to _mkdir.
2615          * XXX: Later, redirect VDIR type here to _mkdir.
2616
2617         if (va->va_type != VREG)
2618             return (EINVAL);
2619
2620         /*
2621          * If the pathname is "", just use dvp, no checks.
2622
2623         if (nmllen == 0) {
2624             VN_HOLD(dvp);
2625             *vpp = dvp;
2626             return (0);
2627
2628         /*
2629          * Don't allow ".." or "..." through here. */
2630
2631         if ((nmllen == 1 && name[0] == '.') ||
2632             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2633             return (EISDIR);
2634
2635         /*
2636          * We make a copy of the attributes because the caller does not
2637          * expect us to change what va points to.
2638
2639         vatr = *va;
2640
2641         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2642             return (EINTR);
2643
2644         /*
2645          * NFS needs to go over the wire, just to be sure whether the
2646          * file exists or not. Using a cached result is dangerous in
2647          * this case when making a decision regarding existence.
2648
2649          * The SMB protocol does NOT really need to go OTW here
2650          * thanks to the expressive NTCREATE disposition values.
2651          * Unfortunately, to do Unix access checks correctly,
2652          * we need to know if the object already exists.
2653          * When the object does not exist, we need VWRITE on
2654          * the directory. Note: smbfslookup() checks VEXEC.
2655
2656         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2657
2658         if (error == 0) {
2659             /*
2660              * The file already exists. Error?
2661              * NB: have a hold from smbfslookup
2662              */
2663
2664             if (exclusive == EXCL) {
2665                 error = EEXIST;
2666                 VN_RELSE(vp);
2667                 goto out;
2668
2669             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2670                 return (EPERM);
2671
2672             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2673                 return (EIO);
2674
2675             /*
2676               * Note: this may break mknod(2) calls to create a directory,
2677               * but that's obscure use. Some other filesystems do this.
2678               * Todo: redirect VDIR type here to _mkdir.
2679               * XXX: Later, redirect VDIR type here to _mkdir.
2680
2681             if (va->va_type != VREG)
2682                 return (EINVAL);
2683
2684             /*
2685               * If the pathname is "", just use dvp, no checks.
2686
2687             if (nmllen == 0) {
2688                 VN_HOLD(dvp);
2689                 *vpp = dvp;
2690                 return (0);
2691
2692             /*
2693               * Don't allow ".." or "..." through here. */
2694
2695             if ((nmllen == 1 && name[0] == '.') ||
2696                 (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2697                 return (EISDIR);
2698
2699             /*
2700               * We make a copy of the attributes because the caller does not
2701               * expect us to change what va points to.
2702
2703             vatr = *va;
2704
2705             if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2706                 return (EINTR);
2707
2708             /*
2709               * NFS needs to go over the wire, just to be sure whether the
2710               * file exists or not. Using a cached result is dangerous in
2711               * this case when making a decision regarding existence.
2712
2713               * The SMB protocol does NOT really need to go OTW here
2714               * thanks to the expressive NTCREATE disposition values.
2715               * Unfortunately, to do Unix access checks correctly,
2716               * we need to know if the object already exists.
2717               * When the object does not exist, we need VWRITE on
2718               * the directory. Note: smbfslookup() checks VEXEC.
2719
2720             error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2721
2722             if (error == 0) {
2723                 /*
2724                   * The file already exists. Error?
2725                   * NB: have a hold from smbfslookup
2726                   */
2727
2728                 if (exclusive == EXCL) {
2729                     error = EEXIST;
2730                     VN_RELSE(vp);
2731                     goto out;
2732
2733                 if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2734                     return (EPERM);
2735
2736                 if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2737                     return (EIO);
2738
2739                 /*
2740                   * Note: this may break mknod(2) calls to create a directory,
2741                   * but that's obscure use. Some other filesystems do this.
2742                   * Todo: redirect VDIR type here to _mkdir.
2743                   * XXX: Later, redirect VDIR type here to _mkdir.
2744
2745                 if (va->va_type != VREG)
2746                     return (EINVAL);
2747
2748                 /*
2749                   * If the pathname is "", just use dvp, no checks.
2750
2751                 if (nmllen == 0) {
2752                     VN_HOLD(dvp);
2753                     *vpp = dvp;
2754                     return (0);
2755
2756                 /*
2757                   * Don't allow ".." or "..." through here. */
2758
2759                 if ((nmllen == 1 && name[0] == '.') ||
2760                     (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2761                     return (EISDIR);
2762
2763                 /*
2764                   * We make a copy of the attributes because the caller does not
2765                   * expect us to change what va points to.
2766
2767                 vatr = *va;
2768
2769                 if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2770                     return (EINTR);
2771
2772                 /*
2773                   * NFS needs to go over the wire, just to be sure whether the
2774                   * file exists or not. Using a cached result is dangerous in
2775                   * this case when making a decision regarding existence.
2776
2777                   * The SMB protocol does NOT really need to go OTW here
2778                   * thanks to the expressive NTCREATE disposition values.
2779                   * Unfortunately, to do Unix access checks correctly,
2780                   * we need to know if the object already exists.
2781                   * When the object does not exist, we need VWRITE on
2782                   * the directory. Note: smbfslookup() checks VEXEC.
2783
2784                 error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2785
2786                 if (error == 0) {
2787                     /*
2788                       * The file already exists. Error?
2789                       * NB: have a hold from smbfslookup
2790                       */
2791
2792                     if (exclusive == EXCL) {
2793                         error = EEXIST;
2794                         VN_RELSE(vp);
2795                         goto out;
2796
2797                     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2798                         return (EPERM);
2799
2800                     if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2801                         return (EIO);
2802
2803                     /*
2804                       * Note: this may break mknod(2) calls to create a directory,
2805                       * but that's obscure use. Some other filesystems do this.
2806                       * Todo: redirect VDIR type here to _mkdir.
2807                       * XXX: Later, redirect VDIR type here to _mkdir.
2808
2809                     if (va->va_type != VREG)
2810                         return (EINVAL);
2811
2812                     /*
2813                       * If the pathname is "", just use dvp, no checks.
2814
2815                     if (nmllen == 0) {
2816                         VN_HOLD(dvp);
2817                         *vpp = dvp;
2818                         return (0);
2819
2820                     /*
2821                       * Don't allow ".." or "..." through here. */
2822
2823                     if ((nmllen == 1 && name[0] == '.') ||
2824                         (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2825                         return (EISDIR);
2826
2827                     /*
2828                       * We make a copy of the attributes because the caller does not
2829                       * expect us to change what va points to.
2830
2831                     vatr = *va;
2832
2833                     if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2834                         return (EINTR);
2835
2836                     /*
2837                       * NFS needs to go over the wire, just to be sure whether the
2838                       * file exists or not. Using a cached result is dangerous in
2839                       * this case when making a decision regarding existence.
2840
2841                       * The SMB protocol does NOT really need to go OTW here
2842                       * thanks to the expressive NTCREATE disposition values.
2843                       * Unfortunately, to do Unix access checks correctly,
2844                       * we need to know if the object already exists.
2845                       * When the object does not exist, we need VWRITE on
2846                       * the directory. Note: smbfslookup() checks VEXEC.
2847
2848                     error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2849
2850                     if (error == 0) {
2851                         /*
2852                           * The file already exists. Error?
2853                           * NB: have a hold from smbfslookup
2854                           */
2855
2856                         if (exclusive == EXCL) {
2857                             error = EEXIST;
2858                             VN_RELSE(vp);
2859                             goto out;
2860
2861                         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2862                             return (EPERM);
2863
2864                         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2865                             return (EIO);
2866
2867                         /*
2868                           * Note: this may break mknod(2) calls to create a directory,
2869                           * but that's obscure use. Some other filesystems do this.
2870                           * Todo: redirect VDIR type here to _mkdir.
2871                           * XXX: Later, redirect VDIR type here to _mkdir.
2872
2873                         if (va->va_type != VREG)
2874                             return (EINVAL);
2875
2876                         /*
2877                           * If the pathname is "", just use dvp, no checks.
2878
2879                         if (nmllen == 0) {
2880                             VN_HOLD(dvp);
2881                             *vpp = dvp;
2882                             return (0);
2883
2884                         /*
2885                           * Don't allow ".." or "..." through here. */
2886
2887                         if ((nmllen == 1 && name[0] == '.') ||
2888                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2889                             return (EISDIR);
2890
2891                         /*
2892                           * We make a copy of the attributes because the caller does not
2893                           * expect us to change what va points to.
2894
2895                         vatr = *va;
2896
2897                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2898                             return (EINTR);
2899
2900                         /*
2901                           * NFS needs to go over the wire, just to be sure whether the
2902                           * file exists or not. Using a cached result is dangerous in
2903                           * this case when making a decision regarding existence.
2904
2905                           * The SMB protocol does NOT really need to go OTW here
2906                           * thanks to the expressive NTCREATE disposition values.
2907                           * Unfortunately, to do Unix access checks correctly,
2908                           * we need to know if the object already exists.
2909                           * When the object does not exist, we need VWRITE on
2910                           * the directory. Note: smbfslookup() checks VEXEC.
2911
2912                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2913
2914                         if (error == 0) {
2915                             /*
2916                               * The file already exists. Error?
2917                               * NB: have a hold from smbfslookup
2918                               */
2919
2920                             if (exclusive == EXCL) {
2921                                 error = EEXIST;
2922                                 VN_RELSE(vp);
2923                                 goto out;
2924
2925                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2926                                 return (EPERM);
2927
2928                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2929                                 return (EIO);
2930
2931                             /*
2932                               * Note: this may break mknod(2) calls to create a directory,
2933                               * but that's obscure use. Some other filesystems do this.
2934                               * Todo: redirect VDIR type here to _mkdir.
2935                               * XXX: Later, redirect VDIR type here to _mkdir.
2936
2937                             if (va->va_type != VREG)
2938                                 return (EINVAL);
2939
2940                             /*
2941                               * If the pathname is "", just use dvp, no checks.
2942
2943                             if (nmllen == 0) {
2944                                 VN_HOLD(dvp);
2945                                 *vpp = dvp;
2946                                 return (0);
2947
2948                             /*
2949                               * Don't allow ".." or "..." through here. */
2950
2951                             if ((nmllen == 1 && name[0] == '.') ||
2952                                 (nmllen == 2 && name[0] == '.' && name[1] == '.'))
2953                                 return (EISDIR);
2954
2955                             /*
2956                               * We make a copy of the attributes because the caller does not
2957                               * expect us to change what va points to.
2958
2959                             vatr = *va;
2960
2960                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2961                             return (EINTR);
2962
2963                         /*
2964                           * NFS needs to go over the wire, just to be sure whether the
2965                           * file exists or not. Using a cached result is dangerous in
2966                           * this case when making a decision regarding existence.
2967
2968                           * The SMB protocol does NOT really need to go OTW here
2969                           * thanks to the expressive NTCREATE disposition values.
2970                           * Unfortunately, to do Unix access checks correctly,
2971                           * we need to know if the object already exists.
2972                           * When the object does not exist, we need VWRITE on
2973                           * the directory. Note: smbfslookup() checks VEXEC.
2974
2975                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2976
2977                         if (error == 0) {
2978                             /*
2979                               * The file already exists. Error?
2980                               * NB: have a hold from smbfslookup
2981                               */
2982
2983                             if (exclusive == EXCL) {
2984                                 error = EEXIST;
2985                                 VN_RELSE(vp);
2986                                 goto out;
2987
2988                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2989                                 return (EPERM);
2990
2991                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2992                                 return (EIO);
2993
2994                             /*
2995                               * Note: this may break mknod(2) calls to create a directory,
2996                               * but that's obscure use. Some other filesystems do this.
2997                               * Todo: redirect VDIR type here to _mkdir.
2998                               * XXX: Later, redirect VDIR type here to _mkdir.
2999
2999                         if (va->va_type != VREG)
3000                             return (EINVAL);
3000
3001                         /*
3002                           * If the pathname is "", just use dvp, no checks.
3003
3004                         if (nmllen == 0) {
3005                             VN_HOLD(dvp);
3006                             *vpp = dvp;
3007                             return (0);
3008
3009                         /*
3010                           * Don't allow ".." or "..." through here. */
3011
3011                         if ((nmllen == 1 && name[0] == '.') ||
3012                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3013                             return (EISDIR);
3014
3015                         /*
3016                           * We make a copy of the attributes because the caller does not
3017                           * expect us to change what va points to.
3018
3019                         vatr = *va;
3020
3020                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3021                             return (EINTR);
3022
3023                         /*
3024                           * NFS needs to go over the wire, just to be sure whether the
3025                           * file exists or not. Using a cached result is dangerous in
3026                           * this case when making a decision regarding existence.
3027
3028                           * The SMB protocol does NOT really need to go OTW here
3029                           * thanks to the expressive NTCREATE disposition values.
3030                           * Unfortunately, to do Unix access checks correctly,
3031                           * we need to know if the object already exists.
3032                           * When the object does not exist, we need VWRITE on
3033                           * the directory. Note: smbfslookup() checks VEXEC.
3034
3035                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
3036
3037                         if (error == 0) {
3038                             /*
3039                               * The file already exists. Error?
3040                               * NB: have a hold from smbfslookup
3041                               */
3042
3043                             if (exclusive == EXCL) {
3044                                 error = EEXIST;
3045                                 VN_RELSE(vp);
3046                                 goto out;
3047
3048                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3049                                 return (EPERM);
3050
3051                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3052                                 return (EIO);
3053
3054                             /*
3055                               * Note: this may break mknod(2) calls to create a directory,
3056                               * but that's obscure use. Some other filesystems do this.
3057                               * Todo: redirect VDIR type here to _mkdir.
3058                               * XXX: Later, redirect VDIR type here to _mkdir.
3059
3059                         if (va->va_type != VREG)
3060                             return (EINVAL);
3060
3061                         /*
3062                           * If the pathname is "", just use dvp, no checks.
3063
3064                         if (nmllen == 0) {
3065                             VN_HOLD(dvp);
3066                             *vpp = dvp;
3067                             return (0);
3068
3069                         /*
3070                           * Don't allow ".." or "..." through here. */
3071
3071                         if ((nmllen == 1 && name[0] == '.') ||
3072                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3073                             return (EISDIR);
3074
3075                         /*
3076                           * We make a copy of the attributes because the caller does not
3077                           * expect us to change what va points to.
3078
3079                         vatr = *va;
3080
3080                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3081                             return (EINTR);
3082
3083                         /*
3084                           * NFS needs to go over the wire, just to be sure whether the
3085                           * file exists or not. Using a cached result is dangerous in
3086                           * this case when making a decision regarding existence.
3087
3088                           * The SMB protocol does NOT really need to go OTW here
3089                           * thanks to the expressive NTCREATE disposition values.
3090                           * Unfortunately, to do Unix access checks correctly,
3091                           * we need to know if the object already exists.
3092                           * When the object does not exist, we need VWRITE on
3093                           * the directory. Note: smbfslookup() checks VEXEC.
3094
3095                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
3096
3097                         if (error == 0) {
3098                             /*
3099                               * The file already exists. Error?
3100                               * NB: have a hold from smbfslookup
3101                               */
3102
3103                             if (exclusive == EXCL) {
3104                                 error = EEXIST;
3105                                 VN_RELSE(vp);
3106                                 goto out;
3107
3108                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3109                                 return (EPERM);
3110
3111                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3112                                 return (EIO);
3113
3114                             /*
3115                               * Note: this may break mknod(2) calls to create a directory,
3116                               * but that's obscure use. Some other filesystems do this.
3117                               * Todo: redirect VDIR type here to _mkdir.
3118                               * XXX: Later, redirect VDIR type here to _mkdir.
3119
3119                         if (va->va_type != VREG)
3120                             return (EINVAL);
3120
3121                         /*
3122                           * If the pathname is "", just use dvp, no checks.
3123
3124                         if (nmllen == 0) {
3125                             VN_HOLD(dvp);
3126                             *vpp = dvp;
3127                             return (0);
3128
3129                         /*
3130                           * Don't allow ".." or "..." through here. */
3131
3131                         if ((nmllen == 1 && name[0] == '.') ||
3132                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3133                             return (EISDIR);
3134
3135                         /*
3136                           * We make a copy of the attributes because the caller does not
3137                           * expect us to change what va points to.
3138
3139                         vatr = *va;
3140
3140                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3141                             return (EINTR);
3142
3143                         /*
3144                           * NFS needs to go over the wire, just to be sure whether the
3145                           * file exists or not. Using a cached result is dangerous in
3146                           * this case when making a decision regarding existence.
3147
3148                           * The SMB protocol does NOT really need to go OTW here
3149                           * thanks to the expressive NTCREATE disposition values.
3150                           * Unfortunately, to do Unix access checks correctly,
3151                           * we need to know if the object already exists.
3152                           * When the object does not exist, we need VWRITE on
3153                           * the directory. Note: smbfslookup() checks VEXEC.
3154
3155                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
3156
3157                         if (error == 0) {
3158                             /*
3159                               * The file already exists. Error?
3160                               * NB: have a hold from smbfslookup
3161                               */
3162
3163                             if (exclusive == EXCL) {
3164                                 error = EEXIST;
3165                                 VN_RELSE(vp);
3166                                 goto out;
3167
3168                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3169                                 return (EPERM);
3170
3171                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3172                                 return (EIO);
3173
3174                             /*
3175                               * Note: this may break mknod(2) calls to create a directory,
3176                               * but that's obscure use. Some other filesystems do this.
3177                               * Todo: redirect VDIR type here to _mkdir.
3178                               * XXX: Later, redirect VDIR type here to _mkdir.
3179
3179                         if (va->va_type != VREG)
3180                             return (EINVAL);
3180
3181                         /*
3182                           * If the pathname is "", just use dvp, no checks.
3183
3184                         if (nmllen == 0) {
3185                             VN_HOLD(dvp);
3186                             *vpp = dvp;
3187                             return (0);
3188
3189                         /*
3190                           * Don't allow ".." or "..." through here. */
3191
3191                         if ((nmllen == 1 && name[0] == '.') ||
3192                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3193                             return (EISDIR);
3194
3195                         /*
3196                           * We make a copy of the attributes because the caller does not
3197                           * expect us to change what va points to.
3198
3199                         vatr = *va;
3200
3200                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3201                             return (EINTR);
3202
3203                         /*
3204                           * NFS needs to go over the wire, just to be sure whether the
3205                           * file exists or not. Using a cached result is dangerous in
3206                           * this case when making a decision regarding existence.
3207
3208                           * The SMB protocol does NOT really need to go OTW here
3209                           * thanks to the expressive NTCREATE disposition values.
3210                           * Unfortunately, to do Unix access checks correctly,
3211                           * we need to know if the object already exists.
3212                           * When the object does not exist, we need VWRITE on
3213                           * the directory. Note: smbfslookup() checks VEXEC.
3214
3215                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
3216
3217                         if (error == 0) {
3218                             /*
3219                               * The file already exists. Error?
3220                               * NB: have a hold from smbfslookup
3221                               */
3222
3223                             if (exclusive == EXCL) {
3224                                 error = EEXIST;
3225                                 VN_RELSE(vp);
3226                                 goto out;
3227
3228                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3229                                 return (EPERM);
3230
3231                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3232                                 return (EIO);
3233
3234                             /*
3235                               * Note: this may break mknod(2) calls to create a directory,
3236                               * but that's obscure use. Some other filesystems do this.
3237                               * Todo: redirect VDIR type here to _mkdir.
3238                               * XXX: Later, redirect VDIR type here to _mkdir.
3239
3239                         if (va->va_type != VREG)
3240                             return (EINVAL);
3240
3241                         /*
3242                           * If the pathname is "", just use dvp, no checks.
3243
3244                         if (nmllen == 0) {
3245                             VN_HOLD(dvp);
3246                             *vpp = dvp;
3247                             return (0);
3248
3249                         /*
3250                           * Don't allow ".." or "..." through here. */
3251
3251                         if ((nmllen == 1 && name[0] == '.') ||
3252                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3253                             return (EISDIR);
3254
3255                         /*
3256                           * We make a copy of the attributes because the caller does not
3257                           * expect us to change what va points to.
3258
3259                         vatr = *va;
3260
3260                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3261                             return (EINTR);
3262
3263                         /*
3264                           * NFS needs to go over the wire, just to be sure whether the
3265                           * file exists or not. Using a cached result is dangerous in
3266                           * this case when making a decision regarding existence.
3267
3268                           * The SMB protocol does NOT really need to go OTW here
3269                           * thanks to the expressive NTCREATE disposition values.
3270                           * Unfortunately, to do Unix access checks correctly,
3271                           * we need to know if the object already exists.
3272                           * When the object does not exist, we need VWRITE on
3273                           * the directory. Note: smbfslookup() checks VEXEC.
3274
3275                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
3276
3277                         if (error == 0) {
3278                             /*
3279                               * The file already exists. Error?
3280                               * NB: have a hold from smbfslookup
3281                               */
3282
3283                             if (exclusive == EXCL) {
3284                                 error = EEXIST;
3285                                 VN_RELSE(vp);
3286                                 goto out;
3287
3288                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3289                                 return (EPERM);
3290
3291                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3292                                 return (EIO);
3293
3294                             /*
3295                               * Note: this may break mknod(2) calls to create a directory,
3296                               * but that's obscure use. Some other filesystems do this.
3297                               * Todo: redirect VDIR type here to _mkdir.
3298                               * XXX: Later, redirect VDIR type here to _mkdir.
3299
3299                         if (va->va_type != VREG)
3300                             return (EINVAL);
3300
3301                         /*
3302                           * If the pathname is "", just use dvp, no checks.
3303
3304                         if (nmllen == 0) {
3305                             VN_HOLD(dvp);
3306                             *vpp = dvp;
3307                             return (0);
3308
3309                         /*
3310                           * Don't allow ".." or "..." through here. */
3311
3311                         if ((nmllen == 1 && name[0] == '.') ||
3312                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3313                             return (EISDIR);
3314
3315                         /*
3316                           * We make a copy of the attributes because the caller does not
3317                           * expect us to change what va points to.
3318
3319                         vatr = *va;
3320
3320                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3321                             return (EINTR);
3322
3323                         /*
3324                           * NFS needs to go over the wire, just to be sure whether the
3325                           * file exists or not. Using a cached result is dangerous in
3326                           * this case when making a decision regarding existence.
3327
3328                           * The SMB protocol does NOT really need to go OTW here
3329                           * thanks to the expressive NTCREATE disposition values.
3330                           * Unfortunately, to do Unix access checks correctly,
3331                           * we need to know if the object already exists.
3332                           * When the object does not exist, we need VWRITE on
3333                           * the directory. Note: smbfslookup() checks VEXEC.
3334
3335                         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
3336
3337                         if (error == 0) {
3338                             /*
3339                               * The file already exists. Error?
3340                               * NB: have a hold from smbfslookup
3341                               */
3342
3343                             if (exclusive == EXCL) {
3344                                 error = EEXIST;
3345                                 VN_RELSE(vp);
3346                                 goto out;
3347
3348                             if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3349                                 return (EPERM);
3350
3351                             if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3352                                 return (EIO);
3353
3354                             /*
3355                               * Note: this may break mknod(2) calls to create a directory,
3356                               * but that's obscure use. Some other filesystems do this.
3357                               * Todo: redirect VDIR type here to _mkdir.
3358                               * XXX: Later, redirect VDIR type here to _mkdir.
3359
3359                         if (va->va_type != VREG)
3360                             return (EINVAL);
3360
3361                         /*
3362                           * If the pathname is "", just use dvp, no checks.
3363
3364                         if (nmllen == 0) {
3365                             VN_HOLD(dvp);
3366                             *vpp = dvp;
3367                             return (0);
3368
3369                         /*
3370                           * Don't allow ".." or "..." through here. */
3371
3371                         if ((nmllen == 1 && name[0] == '.') ||
3372                             (nmllen == 2 && name[0] == '.' && name[1] == '.'))
3373                             return (EISDIR);
3374
3375                         /*
3376                           * We make a copy of the attributes because the caller does not
3377                           * expect us to change what va points to.
3378
3379                         vatr = *va;
3380
3380                         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3381                             return (EINTR);
3382
3383                         /*
3384                           * NFS needs to go over the wire, just to be sure whether the
3385                           * file exists or not. Using a cached result is dangerous in
3386                           * this case when making a decision regarding existence.
3387
3388                           * The SMB protocol does NOT really need to go OTW here
3389                           * thanks
```

```

2664      /*
2665      * Now the code derived from Darwin,
2666      * but with greater use of NT_CREATE
2667      * disposition options. Much changed.
2668      *
2669      * Create (or open) a new child node.
2670      * Note we handled "." and ".." above.
2671      */
2673
2674     if (exclusive == EXCL)
2675         disp = NTCREATEX_DISP_CREATE;
2676     else {
2677         /* Truncate regular files if requested. */
2678         if ((va->va_type == VREG) &&
2679             (va->va_mask & AT_SIZE) &&
2680             (va->va_size == 0))
2681             disp = NTCREATEX_DISP_OVERWRITE_IF;
2682         else
2683             disp = NTCREATEX_DISP_OPEN_IF;
2684     }
2685     xattr = (dnp->n_flag & N_XATTR) ? 1 : 0;
2686     error = smbfs_smb_create(dnp,
2687         name, nmllen, xattr,
2688         disp, &scred, &fid);
2689     if (error)
2690         goto out;
2691
2692     /*
2693      * XXX: Missing some code here to deal with
2694      * the case where we opened an existing file,
2695      * it's size is larger than 32-bits, and we're
2696      * setting the size from a process that's not
2697      * aware of large file offsets. i.e.
2698      * from the NFS3 code:
2699      */
2700 #if NOT_YET /* XXX */
2701     if ((vatr.va_mask & AT_SIZE) &&
2702         vp->v_type == VREG) {
2703         np = VTOSMB(vp);
2704         /*
2705          * Check here for large file handled
2706          * by LF-unaware process (as
2707          * ufs_create() does)
2708          */
2709         if (!lffaware & FOFFMAX)) {
2710             mutex_enter(&np->r_statelock);
2711             if (np->r_size > MAXOFF32_T)
2712                 error = EOVERRLOW;
2713             mutex_exit(&np->r_statelock);
2714         }
2715         if (!error) {
2716             vatr.va_mask = AT_SIZE;
2717             error = smbfssetattr(vp,
2718                 &vatr, 0, cr);
2719         }
2720     }
2721 #endif /* XXX */
2722
2723     /*
2724      * Should use the fid to get/set the size
2725      * while we have it opened here. See above.
2726      */
2727
2728     cerror = smbfs_smb_close(smi->smi_share, fid, NULL, &scred);
2729     if (cerror)
2730         SMBVDEBUG("error %d closing %s\\%s\n",
2731             cerror, dnp->n_rpath, name);
2732

```

```

2733
2734     /*
2735      * In the open case, the name may differ a little
2736      * from what we passed to create (case, etc.)
2737      * so call lookup to get the (opened) name.
2738      *
2739      * XXX: Could avoid this extra lookup if the
2740      * "createact" result from NT_CREATE says we
2741      * created the object.
2742      */
2743     error = smbfs_smb_lookup(dnp, &name, nmllen, &fattr, &scred);
2744     if (error)
2745         goto out;
2746
2747     /* update attr and directory cache */
2748     smbfs_attr_touchdir(dnp);
2749
2750     error = smbfs_nget(dvp, name, nmllen, &fattr, &vp);
2751     if (error)
2752         goto out;
2753
2754     /* XXX invalidate pages if we truncated? */
2755
2756     /* Success! */
2757     *vpp = vp;
2758     error = 0;
2759
2760     out:
2761     smb_credrele(&scred);
2762     smbfs_rw_exit(&dnp->r_rwlock);
2763     if (name != nm)
2764         smbfs_name_free(name, nmllen);
2765     return (error);
2766 }
2767
2768 unchanged_portion_omitted
2769
2770 /*
2771  * smbfsremove does the real work of removing in SMBFS
2772  * Caller has done dir access checks etc.
2773  *
2774  * The normal way to delete a file over SMB is open it (with DELETE access),
2775  * set the "delete-on-close" flag, and close the file. The problem for Unix
2776  * applications is that they expect the file name to be gone once the unlink
2777  * completes, and the SMB server does not actually delete the file until ALL
2778  * opens of that file are closed. We can't assume our open handles are the
2779  * only open handles on a file we're deleting, so to be safe we'll try to
2780  * rename the file to a temporary name and then set delete-on-close. If we
2781  * fail to set delete-on-close (i.e. because other opens prevent it) then
2782  * undo the changes we made and give up with EBUSY. Note that we might have
2783  * permission to delete a file but lack permission to rename, so we want to
2784  * continue in cases where rename fails. As an optimization, only do the
2785  * rename when we have the file open.
2786  *
2787  * This is similar to what NFS does when deleting a file that has local opens,
2788  * but thanks to SMB delete-on-close, we don't need to keep track of when the
2789  * last local open goes away and send a delete. The server does that for us.
2790  */
2791
2792 /* ARGSUSED */
2793 static int
2794 smbfsremove(vnode_t *dvp, vnode_t *vp, struct smb_cred *scred,
2795             int flags)
2796 {
2797     smbnodes_t           *dnp = VTOSMB(dvp);
2798     smbnodes_t           *np = VTOSMB(vp);
2799     char                *tmpname = NULL;
2800     int                  tnlen;

```

```

2812     int          error;
2813     unsigned short fid;
2814     boolean_t    have_fid = B_FALSE;
2815     boolean_t    renamed = B_FALSE;
2816
2817     /*
2818      * The dvp RWlock must be held as writer.
2819      */
2820     ASSERT(dnp->r_rwlock.owner == curthread);
2821
2822     /* Never allow link/unlink directories on SMB. */
2823     if (vp->v_type == VDIR)
2824         return (EPERM);
2825
2826     /*
2827      * We need to flush any dirty pages which happen to
2828      * be hanging around before removing the file. This
2829      * shouldn't happen very often and mostly on file
2830      * systems mounted "nocto".
2831      */
2832     if (vn_has_cached_data(vp) &&
2833         ((np->r_flags & RDIRTY) || np->r_count > 0)) {
2834         error = smbfs_putpage(vp, (offset_t)0, 0, 0,
2835                               scred->scr_cred, NULL);
2836         if (error && (error == ENOSPC || error == EDQUOT)) {
2837             mutex_enter(&np->r_statelock);
2838             if (!np->r_error)
2839                 np->r_error = error;
2840             mutex_exit(&np->r_statelock);
2841         }
2842     }
2843
2844     /* Shared lock for n_fid use in smbfs_smb_setdisp etc. */
2845     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
2846         return (EINTR);
2847
2848     /* Force lookup to go OTW */
2849     smbfs_attrcache_remove(np);
2850
2851     /*
2852      * Get a file handle with delete access.
2853      * Close this FID before return.
2854      */
2855     error = smbfs_smb_tmpopen(np, STD_RIGHT_DELETE_ACCESS,
2856                               scred, &fid);
2857     if (error) {
2858         SMBVDEBUG("error %d opening %s\n",
2859                  error, np->n_rpath);
2860         goto out;
2861     }
2862     have_fid = B_TRUE;
2863
2864     /*
2865      * If we have the file open, try to rename it to a temporary name.
2866      * If we can't rename, continue on and try setting DoC anyway.
2867      */
2868     if ((vp->v_count > 1) && (np->n_fidrefs > 0)) {
2869         tmpname = kmem_alloc(MAXNAMELEN, KM_SLEEP);
2870         tnlen = smbfs_newname(tmpname, MAXNAMELEN);
2871         error = smbfs_smb_t2rename(np, tmpname, tnlen, scred, fid, 0);
2872         if (error != 0) {
2873             SMBVDEBUG("error %d renaming %s -> %s\n",
2874                      error, np->n_rpath, tmpname);
2875             /* Keep going without the rename. */
2876         } else {
2877             renamed = B_TRUE;

```

```

2875         }
2876     }
2877
2878     /*
2879      * Mark the file as delete-on-close. If we can't,
2880      * undo what we did and err out.
2881      */
2882     error = smbfs_smb_setdisp(np, fid, 1, scred);
2883     if (error != 0) {
2884         SMBVDEBUG("error %d setting DoC on %s\n",
2885                  error, np->n_rpath);
2886         /*
2887          * Failed to set DoC. If we renamed, undo that.
2888          * Need np->n_rpath relative to parent (dnp).
2889          * Use parent path name length plus one for
2890          * the separator ('/' or ':')
2891          */
2892     if (renamed) {
2893         char *oldname;
2894         int oldnlen;
2895         int err2;
2896
2897         oldname = np->n_rpath + (dnp->n_rplen + 1);
2898         oldnlen = np->n_rplen - (dnp->n_rplen + 1);
2899         err2 = smbfs_smb_t2rename(np, oldname, oldnlen,
2900                                   scred, fid, 0);
2901         SMBVDEBUG("error %d un-renaming %s -> %s\n",
2902                  err2, tmpname, np->n_rpath);
2903     }
2904     error = EBUSY;
2905     goto out;
2906 }
2907 /* Done! */
2908 smbfs_attrcache_prune(np);
2909
2910 #ifdef SMBFS_VNEVENT
2911 vnevent_remove(vp, dvp, nm, ct);
2912#endif
2913
2914 out:
2915     if (tmpname != NULL)
2916         kmem_free(tmpname, MAXNAMELEN);
2917
2918     if (have_fid)
2919         (void) smbfs_smb_tmpclose(np, fid, scred);
2920     smbfs_rw_exit(&np->r_lkserlock);
2921
2922     if (error == 0) {
2923         /* Keep lookup from finding this node anymore. */
2924         smbfs_rmhash(np);
2925     }
2926
2927     return (error);
2928 }
2929
2930 /* ARGSUSED */
2931 static int
2932 smbfs_link(vnode_t *tdvp, vnode_t *svp, char *tnm, cred_t *cr,
2933             caller_context_t *ct, int flags)
2934 {
2935     /* Not yet... */
2936     return (ENOSYS);
2937 }
2938 }
```

```

2941 /*
2942  * XXX
2943  * This op should support the new IGNORECASE flag for case-insensitive
2944  * lookups, per PSARC 2007/244.
2945 */
2946 /* ARGSUSED */
2947 static int
2948 smbfs_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2949   caller_context_t *ct, int flags)
2950 {
2951   struct smb_cred scred;
2952   smbnode_t      *odnp = VTOSMB(odvp);
2953   smbnode_t      *ndnp = VTOSMB(ndvp);
2954   vnode_t        *ovp;
2955   int error;
2956
2957   if (curproc->p_zone != VTOSMI(odvp)->smi_zone_ref.zref_zone ||
2958       curproc->p_zone != VTOSMI(ndvp)->smi_zone_ref.zref_zone)
2959     return (EPERM);
2960
2961   if (VTOSMI(odvp)->smi_flags & SMI_DEAD ||
2962       VTOSMI(ndvp)->smi_flags & SMI_DEAD ||
2963       odvp->v_vfsp->vfs_flag & VFS_UNMOUNTED ||
2964       ndvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2965     return (EIO);
2966
2967   if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
2968       strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0)
2969     return (EINVAL);
2970
2971 /*
2972  * Check that everything is on the same filesystem.
2973  * vn_rename checks the fsid's, but in case we don't
2974  * fill those in correctly, check here too.
2975 */
2976   if (odvp->v_vfsp != ndvp->v_vfsp)
2977     return (EXDEV);
2978
2979 /*
2980  * Need write access on source and target.
2981  * Server takes care of most checks.
2982 */
2983   error = smbfs_access(odvp, VWRITE|VEXEC, 0, cr, ct);
2984   if (error)
2985     return (error);
2986   if (odvp != ndvp) {
2987     error = smbfs_access(ndvp, VWRITE, 0, cr, ct);
2988     if (error)
2989       return (error);
2990   }
2991
2992 /*
2993  * Need to lock both old/new dirs as writer.
2994  *
2995  * Avoid deadlock here on old vs new directory nodes
2996  * by always taking the locks in order of address.
2997  * The order is arbitrary, but must be consistent.
2998 */
2999   if (odnp < ndnp) {
3000     if (smbfs_rw_enter_sig(&odnp->r_rwlock, RW_WRITER,
3001                           SMBINTR(odvp)))
3002       return (EINTR);
3003     if (smbfs_rw_enter_sig(&ndnp->r_rwlock, RW_WRITER,
3004                           SMBINTR(ndvp))) {
3005       smbfs_rw_exit(&odnp->r_rwlock);
3006       return (EINTR);

```

```

3007           }
3008     } else {
3009       if (smbfs_rw_enter_sig(&ndnp->r_rwlock, RW_WRITER,
3010                           SMBINTR(ndvp)))
3011         return (EINTR);
3012       if (smbfs_rw_enter_sig(&odnp->r_rwlock, RW_WRITER,
3013                           SMBINTR(odvp))) {
3014         smbfs_rw_exit(&ndnp->r_rwlock);
3015         return (EINTR);
3016       }
3017     }
3018   smb_credinit(&scred, cr);
3019
3020   /* Lookup the "old" name */
3021   error = smbfslookup(odvp, onm, &ovp, cr, 0, ct);
3022   if (error == 0) {
3023     /*
3024      * Do the real rename work
3025      */
3026     error = smbfsrename(odvp, ovp, ndvp, nnm, &scred, flags);
3027     VN_RELSE(ovp);
3028   }
3029
3030   smb_credrele(&scred);
3031   smbfs_rw_exit(&odnp->r_rwlock);
3032   smbfs_rw_exit(&ndnp->r_rwlock);
3033
3034   return (error);
3035 }
3036
3037 /*
3038  * smbfsrename does the real work of renaming in SMBFS
3039  * Caller has done dir access checks etc.
3040 */
3041 /* ARGSUSED */
3042 static int
3043 smbfsrename(vnode_t *odvp, vnode_t *ovp, vnode_t *ndvp, char *nnm,
3044   struct smb_cred *scred, int flags)
3045 {
3046   smbnode_t      *odnp = VTOSMB(odvp);
3047   smbnode_t      *onp = VTOSMB(ovp);
3048   smbnode_t      *ndnp = VTOSMB(ndvp);
3049   vnode_t        *nvp = NULL;
3050   int             error;
3051   int             nvp_locked = 0;
3052
3053   /* Things our caller should have checked. */
3054   ASSERT(curproc->p_zone == VTOSMI(odvp)->smi_zone_ref.zref_zone);
3055   ASSERT(odvp->v_vfsp == ndvp->v_vfsp);
3056   ASSERT(ndnp->r_rwlock.owner == curthread);
3057   ASSERT(ndnp->r_rwlock.owner == curthread);
3058
3059   /*
3060    * Lookup the target file. If it exists, it needs to be
3061    * checked to see whether it is a mount point and whether
3062    * it is active (open).
3063    */
3064   error = smbfslookup(ndvp, nnm, &nvp, scred->scr_cred, 0, NULL);
3065   if (!error) {
3066     /*
3067      * Target (nvp) already exists. Check that it
3068      * has the same type as the source. The server
3069      * will check this also, (and more reliably) but
3070      * this lets us return the correct error codes.
3071      */
3072   if (ovp->v_type == VDIR) {

```

```

3073         if (nvp->v_type != VDIR) {
3074             error = ENOTDIR;
3075             goto out;
3076         } else {
3077             if (nvp->v_type == VDIR) {
3078                 error = EISDIR;
3079                 goto out;
3080             }
3081         }
3082     }
3083
3084     /*
3085      * POSIX dictates that when the source and target
3086      * entries refer to the same file object, rename
3087      * must do nothing and exit without error.
3088      */
3089     if (ovp == nvp) {
3090         error = 0;
3091         goto out;
3092     }
3093
3094     /*
3095      * Also must ensure the target is not a mount point,
3096      * and keep mount/umount away until we're done.
3097      */
3098     if (vn_vfsrlock(nvp)) {
3099         error = EBUSY;
3100         goto out;
3101     }
3102     nvp_locked = 1;
3103     if (vn_mountedvfs(nvp) != NULL) {
3104         error = EBUSY;
3105         goto out;
3106     }
3107
3108     /*
3109      * CIFS may give a SHARING_VIOLATION error when
3110      * trying to rename onto an existing object,
3111      * so try to remove the target first.
3112      * (Only for files, not directories.)
3113      */
3114     if (nvp->v_type == VDIR) {
3115         error = EEXIST;
3116         goto out;
3117     }
3118     error = smbfsremove(ndvp, nvp, scred, flags);
3119     if (error != 0)
3120         goto out;
3121
3122     /*
3123      * OK, removed the target file. Continue as if
3124      * lookup target had failed (nvp == NULL).
3125      */
3126     vn_vfsunlock(nvp);
3127     nvp_locked = 0;
3128     VN_RELSE(nvp);
3129     nvp = NULL;
3130 } /* nvp */
3131
3132     smbfs_attrcache_remove(onp);
3133
3134     error = smbfs_smb_rename(onp, ndnp, nnm, strlen(nnm), scred);
3135
3136     /*
3137      * If the old name should no longer exist,
3138      * discard any cached attributes under it.
3139

```

```

3138         */
3139         if (error == 0) {
3140             if (error == 0)
3141                 smbfs_attrcache_prune(onp);
3142             /* SMBFS_VNEVENT... */
3143         }
3144     out:
3145         if (nvp) {
3146             if (nvp_locked)
3147                 vn_vfsunlock(nvp);
3148             VN_RELSE(nvp);
3149         }
3150         return (error);
3152     }
3153
3154     /*
3155      * XXX
3156      * vsecattr_t is new to build 77, and we need to eventually support
3157      * it in order to create an ACL when an object is created.
3158      *
3159      * This op should support the new FIGNORECASE flag for case-insensitive
3160      * lookups, per PSARC 2007/244.
3161      */
3162     /* ARGUSED */
3163     static int
3164     smbfs_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp,
3165                 cred_t *cr, caller_context_t *ct, int flags, vsecattr_t *vsecp)
3166     {
3167         vnode_t          *vp;
3168         struct smbnode   *dnp = VTOSMB(dvp);
3169         struct smbmtinfo *smi = VTOSMI(dvp);
3170         struct smb_cred  scred;
3171         struct smbattr   fattr;
3172         const char        *name = (const char *) nm;
3173         int               nmlen = strlen(name);
3174         int               error, hiderr;
3175
3176         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3177             return (EPERM);
3178
3179         if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3180             return (EIO);
3181
3182         if ((nmlen == 1 && name[0] == '.') ||
3183             (nmlen == 2 && name[0] == '.' && name[1] == '.'))
3184             return (EEXIST);
3185
3186         /* Only plain files are allowed in V_XATTRDIR. */
3187         if (dvp->v_flag & V_XATTRDIR)
3188             return (EINVAL);
3189
3190         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3191             return (EINTR);
3192         smb_credinit(&scred, cr);
3193
3194         /*
3195          * XXX: Do we need r_lkserlock too?
3196          * No use of any shared fid or fctx...
3197          */
3198
3199         /*
3200          * Require write access in the containing directory.
3201          */
3202         error = smbfs_access(dvp, VWRITE, 0, cr, ct);
3203

```

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c          47
3198     if (error)
3199         goto out;
3200
3201     error = smbfs_smb_mkdir(dnp, name, nmllen, &scred);
3202     if (error)
3203         goto out;
3204
3205     error = smbfs_smb_lookup(dnp, &name, &nmllen, &fattr, &scred);
3206     if (error)
3207         goto out;
3208
3209     smbfs_attr_touchdir(dnp);
3210
3211     error = smbfs_nget(dvp, name, nmllen, &fattr, &vp);
3212     if (error)
3213         goto out;
3214
3215     if (name[0] == '.')
3216         if ((hiderr = smbfs_smb_hideit(VTOSMB(vp), NULL, 0, &scred)))
3217             SMBVDEBUG("hide failure %d\n", hiderr);
3218
3219     /* Success! */
3220     *vpp = vp;
3221     error = 0;
3222 out:
3223     smb_credrele(&scred);
3224     smbfs_rw_exit(&dnp->r_rwlock);
3225
3226     if (name != nm)
3227         smbfs_name_free(name, nmllen);
3228
3229     return (error);
3230 }
unchanged_portion_omitted

3335 /* ARGSUSED */
3336 static int
3337 smbfs_symlink(vnode_t *dvp, char *lnm, struct vattr *tva, char *tnm, cred_t *cr,
3338                 caller_context_t *ct, int flags)
3339 {
3340     /* Not yet... */
3341     return (ENOSYS);
3342 }

3345 /* ARGSUSED */
3346 static int
3347 smbfs_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
3348                 caller_context_t *ct, int flags)
3349 {
3350     struct smbnode    *np = VTOSMB(vp);
3351     int                error = 0;
3352     smbmntinfo_t      *smi;
3353
3354     smi = VTOSMI(vp);
3355
3356     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3357         return (EIO);
3358
3359     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3360         return (EIO);
3361
3362 /*
3363 * Require read access in the directory.
3364 */

```

```

new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c          48
3365     error = smbfs_access(vp, VREAD, 0, cr, ct);
3366     if (error)
3367         return (error);
3368
3369     ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));
3370
3371 /*
3372 * Todo readdir cache here
3373 * XXX: Todo readdir cache here
3374 * Note: NFS code is just below this.
3375 *
3376 * I am serializing the entire readdir operation
3377 * now since we have not yet implemented readdir
3378 * cache. This fix needs to be revisited once
3379 * we implement readdir cache.
3380 */
3381 if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
3382     return (EINTR);
3383
3384 error = smbfs_readvmdir(vp, uiop, cr, eofp, ct);
3385
3386 smbfs_rw_exit(&np->r_lkserlock);
3387 }
unchanged_portion_omitted

3606 /*
3607 * Here NFS has: nfs3_bio
3608 * See smbfs_bio above.
3609 */
3610
3611 /* ARGSUSED */
3612 static int
3613 smbfs_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
3614 {
3615     return (ENOSYS);
3616 }

3619 /*
3620 * The pair of functions VOP_RWLOCK, VOP_RWUNLOCK
3621 * are optional functions that are called by:
3622 *     getdents, before/after VOP_READDIR
3623 *     pread, before/after ... VOP_READ
3624 *     pwrite, before/after ... VOP_WRITE
3625 *     (other places)
3626 *
3627 * Careful here: None of the above check for any
3628 * error returns from VOP_RWLOCK / VOP_RWUNLOCK!
3629 * In fact, the return value from _rwlock is NOT
3630 * an error code, but V_WRITELOCK_TRUE / _FALSE.
3631 *
3632 * Therefore, it's up to _this_ code to make sure
3633 * the lock state remains balanced, which means
3634 * we can't "bail out" on interrupts, etc.
3635 */

3637 /* ARGSUSED2 */
3638 static int
3639 smbfs_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
3640 {
3641     smbnode_t        *np = VTOSMB(vp);
3642
3643     if (!write_lock) {
3644         (void) smbfs_rw_enter_sig(&np->r_rwlock, RW_READER, FALSE);

```

```

3645         return (V_WRITELOCK_FALSE);
3646     }

3649     (void) smbfs_rw_enter_sig(&np->r_rwlock, RW_WRITER, FALSE);
3650     return (V_WRITELOCK_TRUE);
3651 }


---


3692 /* mmap support ****
3693
3694 #ifdef DEBUG
3695 static int smbfs_lostpage = 0; /* number of times we lost original page */
3696 #endif
3697
3698 /*
3699 * Return all the pages from [off..off+len) in file
3700 * Like nfs3_getpage
3701 */
3702 /* ARGSUSED */
3703 static int
3704 smbfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protpp,
3705                 page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3706                 enum seg_rw rw, cred_t *cr, caller_context_t *ct)
3707 {
3708     smbnode_t      *np;
3709     smbmntinfo_t   *smi;
3710     int             error;
3711
3712     np = VTOSMB(vp);
3713     smi = VTOSMI(vp);
3714
3715     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3716         return (EIO);
3717
3718     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3719         return (EIO);
3720
3721     if (vp->v_flag & VNOMAP)
3722         return (ENOSYS);
3723
3724     if (protpp != NULL)
3725         *protpp = PROT_ALL;
3726
3727     /*
3728     * Now validate that the caches are up to date.
3729     */
3730     error = smbfs_validate_caches(vp, cr);
3731     if (error)
3732         return (error);
3733
3734 retry:
3735     mutex_enter(&np->r_statelock);
3736
3737     /*
3738     * Don't create dirty pages faster than they
3739     * can be cleaned ... (etc. see nfs)
3740     *
3741     * Here NFS also tests:
3742     * (mi->mi_max_threads != 0 &&
3743     * rp->r_awcount > 2 * mi->mi_max_threads)
3744     */
3745     if (rw == S_CREATE) {
3746         while (np->r_gcount > 0)
3747             cv_wait(&np->r_cv, &np->r_statelock);
3748     }

```

```

3750 */
3751     * If we are getting called as a side effect of a write
3752     * operation the local file size might not be extended yet.
3753     * In this case we want to be able to return pages of zeroes.
3754     */
3755     if (off + len > np->r_size + PAGEOFFSET && seg != segkmap) {
3756         mutex_exit(&np->r_statelock);
3757         return (EFAULT); /* beyond EOF */
3758     }
3759
3760     mutex_exit(&np->r_statelock);
3761
3762     error = pvn_getpages(smbfs_getapage, vp, off, len, protpp,
3763                          pl, plsz, seg, addr, rw, cr);
3764
3765     switch (error) {
3766     case SMBFS_EOF:
3767         smbfs_purge_caches(vp, cr);
3768         goto retry;
3769     case ESTALE:
3770         /*
3771         * Here NFS has: PURGE_STALE_FH(error, vp, cr);
3772         * In-line here as we only use it once.
3773         */
3774         mutex_enter(&np->r_statelock);
3775         np->r_flags |= RSTALE;
3776         if (!np->r_error)
3777             np->r_error = (error);
3778         mutex_exit(&np->r_statelock);
3779         if (vn_has_cached_data(vp))
3780             smbfs_invalidate_pages(vp, (u_offset_t)0, cr);
3781         smbfs_purge_caches(vp, cr);
3782         break;
3783     default:
3784         break;
3785     }
3786
3787     return (error);
3788 }
3789
3790 /*
3791 * Called from pvn_getpages to get a particular page.
3792 * Like nfs3_getpage
3793 */
3794 /* ARGSUSED */
3795 static int
3796 smbfs_getpage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protpp,
3797                 page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3798                 enum seg_rw rw, cred_t *cr)
3799 {
3800     smbnode_t      *np;
3801     smbmntinfo_t   *smi;
3802
3803     uint_t          bsize;
3804     struct buf      *bp;
3805     page_t          *pp;
3806     u_offset_t       lbn;
3807     u_offset_t       io_off;
3808     u_offset_t       blkoff;
3809     size_t          io_len;
3810     uint_t          blksize;
3811     int              error;
3812     /* int readahead; */
3813     int              readahead_issued = 0;
3814     /* int ra_window; * readahead window */

```

```

3815     page_t *pagefound;
3817     np = VTOSMB(vp);
3818     smi = VTOSMI(vp);
3820     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3821         return (EIO);
3823     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3824         return (EIO);
3826     bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

3828 reread:
3829     bp = NULL;
3830     pp = NULL;
3831     pagefound = NULL;
3833     if (pl != NULL)
3834         pl[0] = NULL;

3836     error = 0;
3837     lbn = off / bsize;
3838     blkoff = lbn * bsize;

3840     /*
3841      * NFS queues up readahead work here.
3842     */

3844 again:
3845     if ((pagefound = page_exists(vp, off)) == NULL) {
3846         if (pl == NULL) {
3847             (void) 0; /* Todo: smbfs_async_readahead(); */
3848         } else if (rw == S_CREATE) {
3849             /*
3850              * Block for this page is not allocated, or the offset
3851              * is beyond the current allocation size, or we're
3852              * allocating a swap slot and the page was not found,
3853              * so allocate it and return a zero page.
3854             */
3855             if ((pp = page_create_va(vp, off,
3856                                       PAGESIZE, PG_WAIT, seg, addr)) == NULL)
3857                 cmn_err(CE_PANIC, "smbfs_getapage: page_create");
3858             io_len = PAGESIZE;
3859             mutex_enter(&np->r_statelock);
3860             np->r_nextr = off + PAGESIZE;
3861             mutex_exit(&np->r_statelock);
3862         } else {
3863             /*
3864              * Need to go to server to get a BLOCK, exception to
3865              * that being while reading at offset = 0 or doing
3866              * random i/o, in that case read only a PAGE.
3867             */
3868             mutex_enter(&np->r_statelock);
3869             if (blkoff < np->r_size &&
3870                 blkoff + bsize >= np->r_size) {
3871                 /*
3872                  * If only a block or less is left in
3873                  * the file, read all that is remaining.
3874                 */
3875                 if (np->r_size <= off) {
3876                     /*
3877                      * Trying to access beyond EOF,
3878                      * set up to get at least one page.
3879                     */
3880                 blksize = off + PAGESIZE - blkoff;

```

```

3881             } else
3882                 blksize = np->r_size - blkoff;
3883             } else if ((off == 0) ||
3884             (off != np->r_nextr && !readahead_issued)) {
3885                 blksize = PAGESIZE;
3886                 blkoff = off; /* block = page here */
3887             } else
3888                 blksize = bsize;
3889             mutex_exit(&np->r_statelock);
3890             pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
3891             &io_len, blkoff, blksize, 0);
3892
3894             /*
3895              * Some other thread has entered the page,
3896              * so just use it.
3897             */
3898             if (pp == NULL)
3899                 goto again;
3900
3901             /*
3902              * Now round the request size up to page boundaries.
3903              * This ensures that the entire page will be
3904              * initialized to zeroes if EOF is encountered.
3905             */
3906             io_len = ptob(btocr(io_len));
3907             bp = pageio_setup(pp, io_len, vp, B_READ);
3908             ASSERT(bp != NULL);
3909
3911             /*
3912              * pageio_setup should have set b_addr to 0. This
3913              * is correct since we want to do I/O on a page
3914              * boundary. bp_mapin will use this addr to calculate
3915              * an offset, and then set b_addr to the kernel virtual
3916              * address it allocated for us.
3917             */
3918             ASSERT(bp->b_un.b_addr == 0);
3919
3920             bp->b_eaddr = 0;
3921             bp->b_dev = 0;
3922             bp->b_lblkno = lbtodb(io_off);
3923             bp->b_file = vp;
3924             bp->b_offset = (offset_t)off;
3925             bp_mapin(bp);
3926
3927             /*
3928              * If doing a write beyond what we believe is EOF,
3929              * don't bother trying to read the pages from the
3930              * server, we'll just zero the pages here. We
3931              * don't check that the rw flag is S_WRITE here
3932              * because some implementations may attempt a
3933              * read access to the buffer before copying data.
3934             */
3935             mutex_enter(&np->r_statelock);
3936             if (io_off >= np->r_size && seg == segkmap) {
3937                 mutex_exit(&np->r_statelock);
3938                 bzero(bp->b_un.b_addr, io_len);
3939             } else {
3940                 mutex_exit(&np->r_statelock);
3941                 error = smbfs_bio(bp, 0, cr);
3942             }
3943
3944             /*
3945              * Unmap the buffer before freeing it.
3946             */

```

```

3947         bp_mapout(bp);
3948         pageio_done(bp);
3949
3950         /* Here NFS3 updates all pp->p_fsdata */
3951
3952         if (error == SMBFS_EOF) {
3953             /*
3954             * If doing a write system call just return
3955             * zeroed pages, else user tried to get pages
3956             * beyond EOF, return error. We don't check
3957             * that the rw flag is S_WRITE here because
3958             * some implementations may attempt a read
3959             * access to the buffer before copying data.
3960             */
3961             if (seg == segkmap)
3962                 error = 0;
3963             else
3964                 error = EFAULT;
3965         }
3966
3967         if (!readahead_issued && !error) {
3968             mutex_enter(&np->r_statelock);
3969             np->r_nextr = io_off + io_len;
3970             mutex_exit(&np->r_statelock);
3971         }
3972     }
3973
3974     if (pl == NULL)
3975         return (error);
3976
3977     if (error) {
3978         if (pp != NULL)
3979             pvn_read_done(pp, B_ERROR);
3980         return (error);
3981     }
3982
3983     if (pagefound) {
3984         se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);
3985
3986         /*
3987         * Page exists in the cache, acquire the appropriate lock.
3988         * If this fails, start all over again.
3989         */
3990         if ((pp = page_lookup(vp, off, se)) == NULL) {
3991 #ifdef DEBUG
3992             smbfs_lostpage++;
3993 #endif
3994         goto reread;
3995     }
3996     pl[0] = pp;
3997     pl[1] = NULL;
3998     return (0);
3999 }
4000
4001 if (pp != NULL)
4002     pvn plist_init(pp, pl, plsz, off, io_len, rw);
4003
4004 return (error);
4005
4006 }
4007
4008 */
4009 /* Here NFS has: nfs3_readahead
4010 * No read-ahead in smbfs yet.
4011 */

```

```

4013 /*
4014  * Flags are composed of {B_INVAL, B_FREE, B_DONTNEED, B_FORCE}
4015  * If len == 0, do from off to EOF.
4016  *
4017  * The normal cases should be len == 0 && off == 0 (entire vp list),
4018  * len == MAXBSIZE (from segmap_release actions), and len == PAGESIZE
4019  * (from pageout).
4020  *
4021  * Like nfs3_putpage + nfs_putpages
4022  */
4023 /* ARGSUSED */
4024 static int
4025 smbfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags, cred_t *cr,
4026 caller_context_t *ct)
4027 {
4028     smbnode_t *np;
4029     smbmtinfo_t *smi;
4030     page_t *pp;
4031     u_offset_t eoff;
4032     u_offset_t io_off;
4033     size_t io_len;
4034     int error;
4035     int rdirty;
4036     int err;
4037
4038     np = VTOSMB(vp);
4039     smi = VTOSMI(vp);
4040
4041     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
4042         return (EIO);
4043
4044     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
4045         return (EIO);
4046
4047     if (vp->v_flag & VNOMAP)
4048         return (ENOSYS);
4049
4050     /* Here NFS does rp->r_count (++--) stuff. */
4051
4052     /* Beginning of code from nfs_putpages. */
4053
4054     if (!vn_has_cached_data(vp))
4055         return (0);
4056
4057     /*
4058     * If ROUTOFSpace is set, then all writes turn into B_INVAL
4059     * writes. B_FORCE is set to force the VM system to actually
4060     * invalidate the pages, even if the i/o failed. The pages
4061     * need to get invalidated because they can't be written out
4062     * because there isn't any space left on either the server's
4063     * file system or in the user's disk quota. The B_FREE bit
4064     * is cleared to avoid confusion as to whether this is a
4065     * request to place the page on the freelist or to destroy
4066     * it.
4067     */
4068     if ((np->r_flags & ROUTOFSpace) ||
4069         (vp->v_vfsp->vfs_flag & VFS_UNMOUNTED))
4070         flags = (flags & ~B_FREE) | B_INVAL | B_FORCE;
4071
4072     if (len == 0) {
4073         /*
4074         * If doing a full file synchronous operation, then clear
4075         * the RDIRTY bit. If a page gets dirtied while the flush
4076         * is happening, then RDIRTY will get set again. The
4077         * RDIRTY bit must get cleared before the flush so that
4078         * we don't lose this information.
4079     }

```

```

4079         *
4080         * NFS has B_ASYNC vs sync stuff here.
4081         */
4082     if (off == (u_offset_t)0 &&
4083         (np->r_flags & RDIRTY)) {
4084         mutex_enter(&np->r_statelock);
4085         rdirty = (np->r_flags & RDIRTY);
4086         np->r_flags &= ~RDIRTY;
4087         mutex_exit(&np->r_statelock);
4088     } else
4089         rdirty = 0;
4090
4091     /*
4092     * Search the entire vp list for pages >= off, and flush
4093     * the dirty pages.
4094     */
4095     error = pvn_vplist_dirty(vp, off, smbfs_putapage,
4096                             flags, cr);
4097
4098     /*
4099     * If an error occurred and the file was marked as dirty
4100     * before and we aren't forcibly invalidating pages, then
4101     * reset the RDIRTY flag.
4102     */
4103     if (error && rdirty &&
4104         (flags & (B_INVAL | B_FORCE)) != (B_INVAL | B_FORCE)) {
4105         mutex_enter(&np->r_statelock);
4106         np->r_flags |= RDIRTY;
4107         mutex_exit(&np->r_statelock);
4108     }
4109 } else {
4110     /*
4111     * Do a range from [off...off + len) looking for pages
4112     * to deal with.
4113     */
4114     error = 0;
4115     io_len = 1; /* quiet warnings */
4116     eoff = off + len;
4117
4118     for (io_off = off; io_off < eoff; io_off += io_len) {
4119         mutex_enter(&np->r_statelock);
4120         if (io_off >= np->r_size) {
4121             mutex_exit(&np->r_statelock);
4122             break;
4123         }
4124         mutex_exit(&np->r_statelock);
4125         /*
4126         * If we are not invalidating, synchronously
4127         * freeing or writing pages use the routine
4128         * page_lookup_nowait() to prevent reclaiming
4129         * them from the free list.
4130         */
4131     if ((flags & B_INVAL) || !(flags & B_ASYNC)) {
4132         pp = page_lookup(vp, io_off,
4133                           (flags & (B_INVAL | B_FREE)) ?
4134                           SE_EXCL : SE_SHARED);
4135     } else {
4136         pp = page_lookup_nowait(vp, io_off,
4137                               (flags & B_FREE) ? SE_EXCL : SE_SHARED);
4138     }
4139
4140     if (pp == NULL || !pvn_getdirty(pp, flags))
4141         io_len = PAGESIZE;
4142     else {
4143         err = smbfs_putapage(vp, pp, &io_off,
4144                             &io_len, flags, cr);
4145
4146     }
4147
4148     if (!error)
4149         error = err;
4150
4151     /*
4152     * "io_off" and "io_len" are returned as
4153     * the range of pages we actually wrote.
4154     * This allows us to skip ahead more quickly
4155     * since several pages may've been dealt
4156     * with by this iteration of the loop.
4157     */
4158 }
4159 }
```

```

4145     if (!error)
4146         error = err;
4147
4148     /*
4149     * "io_off" and "io_len" are returned as
4150     * the range of pages we actually wrote.
4151     * This allows us to skip ahead more quickly
4152     * since several pages may've been dealt
4153     * with by this iteration of the loop.
4154     */
4155 }
4156 }
```

4157 }

4158 return (error);

4159 }

4160 /*

4161 * Write out a single page, possibly klustering adjacent dirty pages.

4162 *

4163 * Like nfs3_putapage / nfs3_sync_putapage

4164 */

4165 static int

4166 smbfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *lenp,

4167 int flags, cred_t *cr)

4168 {

4169 smbnodes_t *np;

4170 u_offset_t io_off;

4171 u_offset_t lbn_off;

4172 u_offset_t lbn;

4173 size_t io_len;

4174 uint_t bsize;

4175 int error;

4176

4177 np = VTOSMB(vp);

4178

4179 ASSERT(!vn_is_readonly(vp));

4180

4181 bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

4182 lbn = pp->p_offset / bsize;

4183 lbn_off = lbn * bsize;

4184

4185 /*
4186 * Find a kluster that fits in one block, or in
4187 * one page if pages are bigger than blocks. If
4188 * there is less file space allocated than a whole
4189 * page, we'll shorten the i/o request below.
4190 */

4191 pp = pvn_write_kluster(vp, pp, &io_off, &io_len, lbn_off,

4192 roundup(bsize, PAGESIZE), flags);

4193

4194 /*
4195 * pvn_write_kluster shouldn't have returned a page with offset
4196 * behind the original page we were given. Verify that.
4197 */

4198 ASSERT((pp->p_offset / bsize) >= lbn);

4199

4200 /*
4201 * Now pp will have the list of kept dirty pages marked for
4202 * write back. It will also handle invalidation and freeing
4203 * of pages that are not dirty. Check for page length rounding
4204 * problems.
4205 */

4206 if (io_off + io_len > lbn_off + bsize) {
4207 ASSERT((io_off + io_len) - (lbn_off + bsize) < PAGESIZE);
4208 io_len = lbn_off + bsize - io_off;
4209 }
4210 }

```

4211     /*
4212      * The RMODINPROGRESS flag makes sure that smbfs_bio() sees a
4213      * consistent value of r_size. RMODINPROGRESS is set in writerp().
4214      * When RMODINPROGRESS is set it indicates that a uiomove() is in
4215      * progress and the r_size has not been made consistent with the
4216      * new size of the file. When the uiomove() completes the r_size is
4217      * updated and the RMODINPROGRESS flag is cleared.
4218      *
4219      * The RMODINPROGRESS flag makes sure that smbfs_bio() sees a
4220      * consistent value of r_size. Without this handshaking, it is
4221      * possible that smbfs_bio() picks up the old value of r_size
4222      * before the uiomove() in writerp() completes. This will result
4223      * in the write through smbfs_bio() being dropped.
4224      *
4225      * More precisely, there is a window between the time the uiomove()
4226      * completes and the time the r_size is updated. If a VOP_PUTPAGE()
4227      * operation intervenes in this window, the page will be picked up,
4228      * because it is dirty (it will be unlocked, unless it was
4229      * pagecreate'd). When the page is picked up as dirty, the dirty
4230      * bit is reset (pvn_getdirty()). In smbfs_write(), r_size is
4231      * checked. This will still be the old size. Therefore the page will
4232      * not be written out. When segmap_release() calls VOP_PUTPAGE(),
4233      * the page will be found to be clean and the write will be dropped.
4234      */
4235     if (np->r_flags & RMODINPROGRESS) {
4236         mutex_enter(&np->r_statelock);
4237         if ((np->r_flags & RMODINPROGRESS) &&
4238             np->r_modaddr + MAXBSIZE > io_off &&
4239             np->r_modaddr < io_off + io_len) {
4240             page_t *plist;
4241             /*
4242              * A write is in progress for this region of the file.
4243              * If we did not detect RMODINPROGRESS here then this
4244              * path through smbfs_putapage() would eventually go to
4245              * smbfs_bio() and may not write out all of the data
4246              * in the pages. We end up losing data. So we decide
4247              * to set the modified bit on each page in the page
4248              * list and mark the rnode with RDIRTY. This write
4249              * will be restarted at some later time.
4250              */
4251             plist = pp;
4252             while (plist != NULL) {
4253                 pp = plist;
4254                 page_sub(&plist, pp);
4255                 hat_setmod(pp);
4256                 page_io_unlock(pp);
4257                 page_unlock(pp);
4258             }
4259             np->r_flags |= RDIRTY;
4260             mutex_exit(&np->r_statelock);
4261             if (offp)
4262                 *offp = io_off;
4263             if (lenp)
4264                 *lenp = io_len;
4265             return (0);
4266         }
4267         mutex_exit(&np->r_statelock);
4268     }
4269     /*
4270      * NFS handles (flags & B_ASYNC) here...
4271      * (See nfs_async_putapage())
4272      *
4273      * This code section from: nfs3_sync_putapage()
4274      */

```

```

4277     flags |= B_WRITE;
4279     error = smbfs_rdwr1bn(vp, pp, io_off, io_len, flags, cr);
4281     if ((error == ENOSPC || error == EDQUOT || error == EFBIG |||
4282          error == EACCES) &&
4283          (flags & (B_INVAL|B_FORCE)) != (B_INVAL|B_FORCE)) {
4284         if (!(np->r_flags & ROUTOFSpace)) {
4285             mutex_enter(&np->r_statelock);
4286             np->r_flags |= ROUTOFSpace;
4287             mutex_exit(&np->r_statelock);
4288         }
4289         flags |= B_ERROR;
4290         pvn_write_done(pp, flags);
4291         /*
4292          * If this was not an async thread, then try again to
4293          * write out the pages, but this time, also destroy
4294          * them whether or not the write is successful. This
4295          * will prevent memory from filling up with these
4296          * pages and destroying them is the only alternative
4297          * if they can't be written out.
4298          */
4299         if (!!(flags & B_ASYNC)) {
4300             /* when the pages are unlocked in pvn_write_done,
4301             * some other thread could have come along, locked
4302             * them, and queued for an async thread. It would be
4303             * possible for all of the async threads to be tied
4304             * up waiting to lock the pages again and they would
4305             * all already be locked and waiting for an async
4306             * thread to handle them. Deadlock.
4307             */
4308             if (!(flags & B_ASYNC)) {
4309                 error = smbfs_putpage(vp, io_off, io_len,
4310                               B_INVAL | B_FORCE, cr, NULL);
4311             }
4312         } else {
4313             if (error)
4314                 flags |= B_ERROR;
4315             else if (np->r_flags & ROUTOFSpace) {
4316                 mutex_enter(&np->r_statelock);
4317                 np->r_flags &= ~ROUTOFSpace;
4318                 mutex_exit(&np->r_statelock);
4319             }
4320             pvn_write_done(pp, flags);
4321         }
4322         /*
4323          * Now more code from: nfs3_putapage */
4325         if (offp)
4326             *offp = io_off;
4327         if (lenp)
4328             *lenp = io_len;
4329         return (error);
4330     }
4331     /*
4332      * NFS has this in nfs_client.c (shared by v2,v3,...)
4333      * We have it here so smbfs_putapage can be file scope.
4334      */
4335     void
4336     smbfs_invalidate_pages(vnode_t *vp, u_offset_t off, cred_t *cr)
4337     {
4338         smbnode_t *np;
4339         np = VTOSMB(vp);

```

```

4344     mutex_enter(&np->r_statelock);
4345     while ((np->r_flags & RTRUNCATE)
4346             cv_wait(&np->r_cv, &np->r_statelock));
4347     np->r_flags |= RTRUNCATE;
4348
4349     if (off == (u_offset_t)0) {
4350         np->r_flags &= ~RDIRTY;
4351         if (!(np->r_flags & RSTALE))
4352             np->r_error = 0;
4353     }
4354     /* Here NFSv3 has np->r_truncaddr = off; */
4355     mutex_exit(&np->r_statelock);
4356
4357     (void) pvn_vplist_dirty(vp, off, smbfs_putapage,
4358      B_INVAL | B_TRUNC, cr);
4359
4360     mutex_enter(&np->r_statelock);
4361     np->r_flags &= ~RTRUNCATE;
4362     cv_broadcast(&np->r_cv);
4363     mutex_exit(&np->r_statelock);
4364 }
4365
4366 /* Like nfs3_map */
4367
4368 /* ARGSUSED */
4369 static int
4370 smbfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
4371             size_t len, uchar_t prot, uchar_t maxprot, uint_t flags,
4372             cred_t *cr, caller_context_t *ct)
4373 {
4374     segvn_crargs_t vn_a;
4375     struct vattr va;
4376     smbnode_t *np;
4377     smbmtinfo_t *smi;
4378     int error;
4379
4380     np = VTOSMB(vp);
4381     smi = VTOSMI(vp);
4382
4383     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
4384         return (EIO);
4385
4386     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
4387         return (EIO);
4388
4389     if (vp->v_flag & VNOMAP)
4390         return (ENOSYS);
4391
4392     if (off < 0 || off + (ssize_t)len < 0)
4393         return (ENXIO);
4394
4395     if (vp->v_type != VREG)
4396         return (ENODEV);
4397
4398     /*
4399      * NFS does close-to-open consistency stuff here.
4400      * Just get (possibly cached) attributes.
4401      */
4402     va.va_mask = AT_ALL;
4403     if ((error = smbfsgetattp(vp, &va, cr)) != 0)
4404         return (error);
4405
4406     /*
4407      * Check to see if the vnode is currently marked as not cachable.
4408      * This means portions of the file are locked (through VOP_FRLOCK).
4409

```

```

4410             * In this case the map request must be refused. We use
4411             * rp->r_lkserlock to avoid a race with concurrent lock requests.
4412             */
4413             /* Atomically increment r_inmap after acquiring r_rwlock. The
4414             * idea here is to acquire r_rwlock to block read/write and
4415             * not to protect r_inmap. r_inmap will inform smbfs_read/write()
4416             * that we are in smbfs_map(). Now, r_rwlock is acquired in order
4417             * and we can prevent the deadlock that would have occurred
4418             * when smbfs_addmap() would have acquired it out of order.
4419             *
4420             * Since we are not protecting r_inmap by any lock, we do not
4421             * hold any lock when we decrement it. We atomically decrement
4422             * r_inmap after we release r_lkserlock. Note that rwlock is
4423             * re-entered as writer in smbfs_addmap (called via as_map).
4424             */
4425
4426     if (smbfs_rw_enter_sig(&np->r_rwlock, RW_WRITER, SMBINTR(vp)))
4427         return (EINTR);
4428     atomic_inc_uint(&np->r_inmap);
4429     smbfs_rw_exit(&np->r_rwlock);
4430
4431     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp))) {
4432         atomic_dec_uint(&np->r_inmap);
4433         return (EINTR);
4434     }
4435
4436     if (vp->v_flag & VNOCACHE) {
4437         error = EAGAIN;
4438         goto done;
4439     }
4440
4441     /*
4442      * Don't allow concurrent locks and mapping if mandatory locking is
4443      * enabled.
4444      */
4445     if ((flk_has_remote_locks(vp) || smbfs_lm_has_sleep(vp)) &&
4446         MANDLOCK(vp, va.va_mode)) {
4447         error = EAGAIN;
4448         goto done;
4449     }
4450
4451     as_rangelock(as);
4452     error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
4453     if (error != 0) {
4454         as_rangeunlock(as);
4455         goto done;
4456     }
4457
4458     vn_a.vp = vp;
4459     vn_a.offset = off;
4460     vn_a.type = (flags & MAP_TYPE);
4461     vn_a.prot = (uchar_t)prot;
4462     vn_a.maxprot = (uchar_t)maxprot;
4463     vn_a.flags = (flags & ~MAP_TYPE);
4464     vn_a.cred = cr;
4465     vn_a.amp = NULL;
4466     vn_a.szc = 0;
4467     vn_a.lgrp_mem_policy_flags = 0;
4468
4469     error = as_map(as, *addrp, len, segvn_create, &vn_a);
4470     as_rangeunlock(as);
4471
4472 done:
4473     smbfs_rw_exit(&np->r_lkserlock);
4474     atomic_dec_uint(&np->r_inmap);

```

```

4475     return (error);
4476 }

4478 /* ARGSUSED */
4479 static int
4480 smbfs_addmap(vnode_t *vp, offset_t off, struct as, caddr_t addr,
4481     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags,
4482     cred_t *cr, caller_context_t *ct)
4483 {
4484     smbnodes_t *np = VTOSMB(vp);
4485     boolean_t inc_fidrefs = B_FALSE;
4486
4487     /*
4488      * When r_mapcnt goes from zero to non-zero,
4489      * increment n_fidrefs
4490      */
4491     mutex_enter(&np->r_statelock);
4492     if (np->r_mapcnt == 0)
4493         inc_fidrefs = B_TRUE;
4494     np->r_mapcnt += btopr(len);
4495     mutex_exit(&np->r_statelock);
4496
4497     if (inc_fidrefs) {
4498         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
4499         np->n_fidrefs++;
4500         smbfs_rw_exit(&np->r_lkserlock);
4501     }
4502
4503     return (0);
4504 }

4506 /*
4507  * Use an address space callback to flush pages dirty pages after unmap,
4508  * which we can't do directly in smbfs_delmap due to locking issues.
4509 */
4510 typedef struct smbfs_delmap_args {
4511     vnode_t             *vp;
4512     cred_t              *cr;
4513     offset_t             off;
4514     caddr_t              addr;
4515     size_t               len;
4516     uint_t               prot;
4517     uint_t               maxprot;
4518     uint_t               flags;
4519     boolean_t            dec_fidrefs;
4520 } smbfs_delmap_args_t;

4522 /* ARGSUSED */
4523 static int
4524 smbfs_delmap(vnode_t *vp, offset_t off, struct as, caddr_t addr,
4525     size_t len, uint_t prot, uint_t maxprot, uint_t flags,
4526     cred_t *cr, caller_context_t *ct)
4527 {
4528     smbnodes_t *np = VTOSMB(vp);
4529     smbfs_delmap_args_t *dmapp;
4530     int error;
4531
4532     dmapp = kmem_zalloc(sizeof (*dmapp), KM_SLEEP);
4533
4534     dmapp->vp = vp;
4535     dmapp->off = off;
4536     dmapp->addr = addr;
4537     dmapp->len = len;
4538     dmapp->prot = prot;
4539     dmapp->maxprot = maxprot;
4540     dmapp->flags = flags;

```

```

4541     dmapp->cr = cr;
4542     dmapp->dec_fidrefs = B_FALSE;
4543
4544     /*
4545      * When r_mapcnt returns to zero, arrange for the
4546      * callback to decrement n_fidrefs
4547      */
4548     mutex_enter(&np->r_statelock);
4549     np->r_mapcnt -= btopr(len);
4550     ASSERT(np->r_mapcnt >= 0);
4551     if (np->r_mapcnt == 0)
4552         dmapp->dec_fidrefs = B_TRUE;
4553     mutex_exit(&np->r_statelock);
4554
4555     error = as_add_callback(as, smbfs_delmap_callback, dmapp,
4556                             AS_UNMAP_EVENT, addr, len, KM_SLEEP);
4557     if (error != 0) {
4558         /*
4559          * So sad, no callback is coming. Can't flush pages
4560          * in delmap (as locks). Just handle n_fidrefs.
4561          */
4562         cmn_err(CE_NOTE, "smbfs_delmap(%p) "
4563                 "as_add_callback err=%d",
4564                 (void *)vp, error);
4565
4566     if (dmapp->dec_fidrefs) {
4567         struct smb_cred scred;
4568
4569         (void) smbfs_rw_enter_sig(&np->r_lkserlock,
4570                                   RW_WRITER, 0);
4571         smb_credinit(&scred, dmapp->cr);
4572
4573         smbfs_rele_fid(np, &scred);
4574         smb_credrele(&scred);
4575         smbfs_rw_exit(&np->r_lkserlock);
4576     }
4577     kmem_free(dmapp, sizeof (*dmapp));
4578 }
4579
4580     return (0);
4581 }
4582
4583 /*
4584  * Remove some pages from an mmap'd vnode. Flush any
4585  * dirty pages in the unmapped range.
4586 */
4587
4588 /* ARGSUSED */
4589 static void
4590 smbfs_delmap_callback(struct as *as, void *arg, uint_t event)
4591 {
4592     vnode_t             *vp;
4593     smbnodes_t           *np;
4594     smbmtinfo_t          *smi;
4595     smbfs_delmap_args_t  *dmapp = arg;
4596
4597     vp = dmapp->vp;
4598     np = VTOSMB(vp);
4599     smi = VTOSMI(vp);
4600
4601     /* Decremented r_mapcnt in smbfs_delmap */
4602
4603     /*
4604      * Initiate a page flush and potential commit if there are
4605      * pages, the file system was not mounted readonly, the segment
4606      * was mapped shared, and the pages themselves were writeable.

```

```

4607      *
4608      * mark RDIRTY here, will be used to check if a file is dirty when
4609      * unmount smbfs
4610      */
4611  if (vn_has_cached_data(vp) && !vn_is_READONLY(vp) &&
4612      dmapp->flags == MAP_SHARED && (dmapp->maxprot & PROT_WRITE)) {
4613      mutex_enter(&np->r_statelock);
4614      np->r_flags |= RDIRTY;
4615      mutex_exit(&np->r_statelock);

4617      /*
4618      * Need to finish the putpage before we
4619      * close the OTW FID needed for I/O.
4620      */
4621      (void) smbfs_putpage(vp, dmapp->off, dmapp->len, 0,
4622                           dmapp->cr, NULL);
4623  }

4625  if ((np->r_flags & RDIRECTIO) || (smi->smi_flags & SMI_DIRECTIO))
4626      (void) smbfs_putpage(vp, dmapp->off, dmapp->len,
4627                            B_INVAL, dmapp->cr, NULL);

4629  /*
4630  * If r_mapcnt went to zero, drop our FID ref now.
4631  * On the last fidref, this does an OtW close.
4632  */
4633  if (dmapp->dec_fidrefs) {
4634      struct smb_cred scred;

4636  smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
4637  smb_credinit(&scred, dmapp->cr);

4639  smbfs_rele_fid(np, &scred);

4641  smb_credrele(&scred);
4642  smbfs_rw_exit(&np->r_lkserlock);
4643  }

4645  (void) as_delete_callback(as, arg);
4646  kmem_free(dmapp, sizeof(*dmapp));
4647 }

4649 /* No smbfs_pageio() or smbfs_dispose() ops. */

4651 /* misc. *****/
4652 */

4654 */
4655 * XXX
4656 * This op may need to support PSARC 2007/440, nbmand changes for CIFS Service.
4657 */
4658 static int
4659 smbfs_frllock(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
4660                 offset_t offset, struct flk_callback *flk_cbp, cred_t *cr,
4661                 caller_context_t *ct)
4662 {
4663     if (curproc->p_zone != VTOSMI(vp)->smi_zone_ref.zref_zone)
4664         return (EIO);

4666     if (VTOSMI(vp)->smi_flags & SMI_LLOCK)
4667         return (fs_frllock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
4668     else
4669         return (ENOSYS);
4670 }

4672 */

```

```

4673  * Free storage space associated with the specified vnode. The portion
4674  * to be freed is specified by bfp->l_start and bfp->l_len (already
4675  * normalized to a "whence" of 0).
4676  *
4677  * Called by fcntl(fd, F_FREEESP, lkp) for libc:ftruncate, etc.
4678  */
4679 /* ARGSUSED */
4680 static int
4681 smbfs_space(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
4682              offset_t offset, cred_t *cr, caller_context_t *ct)
4683 {
4684     int             error;
4685     smbmntinfo_t   *smi;

4687     smi = VTOSMI(vp);

4689     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
4690         return (EIO);

4692     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
4693         return (EIO);

4695     /* Caller (fcntl) has checked v_type */
4696     ASSERT(vp->v_type == VREG);
4697     if (cmd != F_FREEESP)
4698         return (EINVAL);

4700     /*
4701     * Like NFS3, no 32-bit offset checks here.
4702     * Our SMB layer takes care to return EFBIG
4703     * when it has to fallback to a 32-bit call.
4704     */

4706     error = convoff(vp, bfp, 0, offset);
4707     if (!error) {
4708         ASSERT(bfp->l_start >= 0);
4709         if (bfp->l_len == 0) {
4710             struct vattr va;

4712             /*
4713             * ftruncate should not change the ctime and
4714             * mtime if we truncate the file to its
4715             * previous size.
4716             */
4717             va.va_mask = AT_SIZE;
4718             error = smbfssetattr(vp, &va, cr);
4719             if (error || va.va_size == bfp->l_start)
4720                 return (error);
4721             va.va_mask = AT_SIZE;
4722             va.va_size = bfp->l_start;
4723             error = smbfssetattr(vp, &va, 0, cr);
4724             /* SMBFS_VNEVENT... */
4725         } else
4726             error = EINVAL;
4727     }
4729     return (error);
4730 }

4733 /* ARGSUSED */
4734 static int
4735 smbfs_realvp(vnode_t *vp, vnode_t **vpp, caller_context_t *ct)
4736 {

4738     return (ENOSYS);

```

```

4739 }

4742 /* ARGSUSED */
4743 static int
4744 smbfs_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
4745     caller_context_t *ct)
4746 {
4747     vfs_t *vfs;
4748     smbmntinfo_t *smi;
4749     struct smb_share *ssp;

4751     vfs = vp->v_vfsp;
4752     smi = VFTOSMI(vfs);

4754     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
4755         return (EIO);

4757     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
4758         return (EIO);

4760     switch (cmd) {
4761     case _PC_FILESIZEBITS:
4762         ssp = smi->smi_share;
4763         if (SSTOVC(ssp)->vc_sopt.sv_caps & SMB_CAP_LARGE_FILES)
4764             *valp = 64;
4765         else
4766             *valp = 32;
4767         break;

4769     case _PC_LINK_MAX:
4770         /* We only ever report one link to an object */
4771         *valp = 1;
4772         break;

4774     case _PC_ACL_ENABLED:
4775         /*
4776          * Always indicate that ACLs are enabled and
4777          * that we support ACE_T format, otherwise
4778          * libsec will ask for ACLENT_T format data
4779          * which we don't support.
4780         */
4781         *valp = _ACL_ACE_ENABLED;
4782         break;

4784     case _PC_SYMLINK_MAX: /* No symlinks until we do Unix extensions */
4785         *valp = 0;
4786         break;

4788     case _PC_XATTR_EXISTS:
4789         if (vfs->vfs_flag & VFS_XATTR) {
4790             *valp = smbfs_xa_exists(vp, cr);
4791             break;
4792         }
4793         return (EINVAL);

4795     case _PC_SATTR_ENABLED:
4796     case _PC_SATTR_EXISTS:
4797         *valp = 1;
4798         break;

4800     case _PC_TIMESTAMP_RESOLUTION:
4801         /*
4802          * Windows times are tenths of microseconds
4803          * (multiples of 100 nanoseconds).
4804         */

```

```

4805             *valp = 100L;
4806             break;

4808         default:
4809             return (fs_pathconf(vp, cmd, valp, cr, ct));
4810         }
4811         return (0);
4812     } unchanged_portion_omitted

4920 /*
4921  * Most unimplemented ops will return ENOSYS because of fs_nosys().
4922  * The only ops where that won't work are ACCESS (due to open(2)
4923  * failures) and ... (anything else left?)
4924 */
4925 const fs_operation_def_t smbfs_vnodeops_template[] = {
4926     VOPNAME_OPEN, { .vop_open = smbfs_open },
4927     VOPNAME_CLOSE, { .vop_close = smbfs_close },
4928     VOPNAME_READ, { .vop_read = smbfs_read },
4929     VOPNAME_WRITE, { .vop_write = smbfs_write },
4930     VOPNAME_IOCTL, { .vop_ioctl = smbfs_ioctl },
4931     VOPNAME_GETATTR, { .vop_getattr = smbfs_getattr },
4932     VOPNAME_SETATTR, { .vop_setattr = smbfs_setattr },
4933     VOPNAME_ACCESS, { .vop_access = smbfs_access },
4934     VOPNAME_LOOKUP, { .vop_lookup = smbfs_lookup },
4935     VOPNAME_CREATE, { .vop_create = smbfs_create },
4936     VOPNAME_REMOVE, { .vop_remove = smbfs_remove },
4937     VOPNAME_LINK, { .vop_link = smbfs_link },
4938     VOPNAME_RENAME, { .vop_rename = smbfs_rename },
4939     VOPNAME_MKDIR, { .vop_mkdir = smbfs_mkdir },
4940     VOPNAME_RMDIR, { .vop_rmdir = smbfs_rmdir },
4941     VOPNAME_READDIR, { .vop_readdir = smbfs_readdir },
4942     VOPNAME_SYMLINK, { .vop_symlink = smbfs_symlink },
4943     VOPNAME_READLINK, { .vop_readlink = smbfs_readlink },
4944     VOPNAME_FSYNC, { .vop_fsync = smbfs_fsync },
4945     VOPNAME_INACTIVE, { .vop_inactive = smbfs_inactive },
4946     VOPNAME_FID, { .vop_fid = smbfs_fid },
4947     VOPNAME_RWLOCK, { .vop_rwlock = smbfs_rwlock },
4948     VOPNAME_RWUNLOCK, { .vop_rwunlock = smbfs_rwunlock },
4949     VOPNAME_SEEK, { .vop_seek = smbfs_seek },
4950     VOPNAME_FRLOCK, { .vop_frlock = smbfs_frlock },
4951     VOPNAME_SPACE, { .vop_space = smbfs_space },
4952     VOPNAME_REALVP, { .vop_realvp = smbfs_realvp },
4953     VOPNAME_GETPAGE, { .vop_getpage = smbfs_getpage },
4954     VOPNAME_PUTPAGE, { .vop_putpage = smbfs_putpage },
4955     VOPNAME_MAP, { .vop_map = smbfs_map },
4956     VOPNAME_ADDMAP, { .vop_addmap = smbfs_addmap },
4957     VOPNAME_DELMAP, { .vop_delmap = smbfs_delmap },
4958     VOPNAME_DUMP, { .error = fs_nosys }, /* smbfs_dump, */
4959     VOPNAME_PATHCONF, { .vop_pathconf = smbfs_pathconf },
4960     VOPNAME_PAGEIO, { .error = fs_nosys }, /* smbfs_pageio, */
4961     VOPNAME_SETSECAATTR, { .vop_setsecatr = smbfs_setsecatr },
4962     VOPNAME_GETSECAATTR, { .vop_getsecatr = smbfs_getsecatr },
4963     VOPNAME_SHRLOCK, { .vop_shrlock = smbfs_shrlock },
4964 #ifdef SMBFS_VNEVENT
4965     VOPNAME_VNEVENT, { .vop_vnevent = fs_vnevent_support },
4966 #endif
4967     { NULL, NULL }
4968 };

```