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

```
   1 .\"
   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 modificat
   5 .\" the following disclaimer. 2. Redistributions in binary form must reproduce t
   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 name Ns Oo = Ns Ar value Oc
  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 name Ns Oo = Ns Ar value Oc
  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 name Ns Oo = Ns Ar value Oc
  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
  10 .TH MOUNT_SMBFS 1M "Jan 2, 2012"
  11 .SH NAME
  12 mount_smbfs, umount_smbfs \- mount and unmount a shared resource from a CIFS
  13 file server
  14 .SH SYNOPSIS
  15 .LP
  16 .nf
  17 \fB/sbin/mount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] [\fB-o\fR \fIname\fR\
  18 .fi

  20 .LP
  21 .nf
  22 \fB/sbin/mount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] [\fB-o\fR \fIname\fR\
  23 .fi

  25 .LP
  26 .nf
  27 \fB/sbin/mount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] [\fB-o\fR \fIname\fR\
  28      [\fB-O\fR] \fIresource\fR \fImount-point\fR
  29 .fi
```



```
  31 .LP
  32 .nf
  33 \fB/sbin/umount\fR [\fB-F smbfs\fR] [\fIgeneric-options\fR] \fImount-point\fR
  34 .fi

  36 .SH DESCRIPTION
  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
  44 If \fImount-point\fR has any contents prior to the \fBmount\fR operation, those
  45 contents remain hidden until the resource is unmounted. An authorized user with
  46 the \fBSYS_MOUNT\fR privilege can perform a \fBmount\fR operation. Also, a user
  47 can perform SMBFS mount operations on a directory the user owns.
  48 .sp
  49 .LP
  50 If the resource is listed in the \fB/etc/vfstab\fR file, you can specify either
  51 \fIresource\fR or \fImount-point\fR as the \fBmount\fR command will consult the
  52 \fB/etc/vfstab\fR file for more information. If the \fB-F\fR option is omitted,
  53 \fBmount\fR takes the file system type from the entry in the \fB/etc/vfstab\fR
  85 file.
  86 .Pp
  87 If the resource is not listed in the
  88 .Pa /etc/vfstab
  89 file, the command line must specify both
```

```
.Ar resource
and
.Ar mount-point .
.Pp
The
.Nm umount
utility detaches a mounted file system from the file system hierarchy.
An authorized user with the
.Dv SYS_MOUNT
privilege can perform a
.Nm umount
operation.
Also, a user can perform SMBFS unmount operations on a directory the user owns.
.Pp
The
.Em network/smb/client
service must be enabled to successfully mount an SMB share.
This service is enabled, by default.
.Pp
To enable the service, enter the following
.Xr svcadm 1M
command:
.Bd -literal
# svcadm enable network/smb/client
.Ed
.Ss Operands
The
.Nm mount
command supports the following operands:
.Bl -tag -width Ds
.It Xo
.Ar resource
.No // Ns Oo Ar workgroup Ns \&; Ns Oc Ns
.Oo Ar user Ns Oo : Ns Ar password Ns Oc Ns @ Ns Oc Ns
.Ar server Ns / Ns Ar share
.Xc
The name of the resource to be mounted.
In addition to its name, you can specify the following information about the
resource:
.Bl -bullet
.It
.Ar password
is the password associated with
.Ar user .
If
.Ar password
is not specified, the mount first attempts to use the password stored by the
.Nm smbutil Cm login
command (if any).
If that password fails to authenticate, the
.Nm mount_smbfs
prompts you for a password.
.It
.Ar server
is the DNS or NetBIOS name of the remote computer.
.It
.Ar share
is the resource name on the remote server.
.It
.Ar user
is the remote user name.
If
.Ar user
is omitted, the logged in user ID is used.
.It
.Ar workgroup
```



```
is the name of the workgroup or the Windows domain in which the user name is
defined.
.Pp
.sp
.LP
If the resource is not listed in the \fB/etc/vfstab\fR file, the command line
must specify both \fIresource\fR and \fImount-point\fR.
.sp
.LP
The \fBumount\fR utility detaches a mounted file system from the file system
hierarchy. An authorized user with the \fBSYS_MOUNT\fR privilege can perform a
\fBumount\fR operation. Also, a user can perform SMBFS unmount operations on a
directory the user owns.
.sp
.LP
The \fBnetwork/smb/client\fR service must be enabled to successfully mount a
CIFS share. This service is enabled, by default.
.sp
.LP
To enable the service, enter the following \fBsvcadm\fR(1M) command:
.sp
.in +2
.nf
# \fBsvcadm enable network/smb/client\fR
.fi
.in -2
.sp

.SS "Operands"
.sp
.LP
The \fBmount\fR command supports the following operands:
.sp
.ne 2
.na
\fB\fIresource\fR
//[\fIworkgroup\fR;][\fIuser\fR[:\fIpassword\fR]@]\fIserver\fR/\fIshare\fR\fR
.ad
.sp .6
.RS 4n
.sp
.LP
The name of the resource to be mounted. In addition to its name, you can
specify the following information about the resource:
.RS +4
.TP
.ie t \(bu
.el o
\fIpassword\fR is the password associated with \fIuser\fR. If \fIpassword\fR is
not specified, the mount first attempts to use the password stored by the
\fBsmbutil login\fR command (if any). If that password fails to authenticate,
the \fBmount_smbfs\fR prompts you for a password.
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fIserver\fR is the DNS or NetBIOS name of the remote computer.
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fIshare\fR is the resource name on the remote server.
.RE
.RS +4
```

```
.TP
.ie t \(bu
.el o
\fIuser\fR is the remote user name. If \fIuser\fR is omitted, the logged in
user ID is used.
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fIworkgroup\fR is the name of the workgroup or the Windows domain in which the
user name is defined.
.sp
```

If the resource includes a workgroup, you must escape the semicolon that appears after the workgroup name to prevent it from being interpreted by the command shell.

For instance, surround the entire resource name with double quotes:

```
.Bd -literal
mount -F smbfs "//SALES;george@RSERVER" /mnt
.Ed
.El
.It Ar mount-point
command shell. For instance, surround the entire resource name with double
quotes: \fBmount -F smbfs "//SALES;george@RSERVER" /mnt\fR.
.RE
.RE

.sp
.ne 2
.na
\fB\fImount-point\fR\fR
.ad
.sp .6
.RS 4n
```

The path to the location where the file system is to be mounted or unmounted.

The .Nm mount command maintains a table of mounted file systems in the .Pa /etc/mnttab file. See the .Xr mnttab 4 man page. .El .Sh OPTIONS See the .Xr mount 1M man page for the list of supported .Ar generic-options . .Bl -tag -width Ds .It Fl o Ar name Ns Oo = Ns Ar value Oc Sets the file system-specific properties. You can specify more than one name-value pair as a list of comma-separated pairs. No spaces are permitted in the list. The properties are as follows: .Bl -tag -width Ds .It Cm acl Ns | Ns Cm noacl

```
The \fBmount\fR command maintains a table of mounted file systems in the
\fB/etc/mnttab\fR file. See the \fBmnttab\fR(4) man page.
.RE

.SH OPTIONS
.sp
.LP
See the \fBmount\fR(1M) man page for the list of supported
```



```
\fIgeneric-options\fR.
.sp
.ne 2
.na
\fB\fB-o\fR \fIname\fR\fB=\fR\fIvalue\fR or\fR
.ad
.br
.na
\fB\fB-o\fR \fIname\fR\fR
.ad
.sp .6
.RS 4n
Sets the file system-specific properties. You can specify more than one
name-value pair as a list of comma-separated pairs. No spaces are permitted in
the list. The properties are as follows:

.sp
.ne 2
.na
\fB\fBacl\fR|\fBnoacl\fR\fR
.ad
.sp .6
.RS 4n
```

Enable (or disable) presentation of Access Control Lists (ACLs)

on files and directories under this .Xr smbfs 7FS mount. The default behavior is .Cm noacl , which presents files and directories as owned by the owner of the mount point and having permissions based on .Cm fileperms or .Cm dirperms . With the .Cm acl mount option, files are presented with ACLs obtained from the SMB server. Setting the .Cm acl mount option is not advised unless the system is joined to an Active Directory domain and using .Xr ldap 1

```
on files and directories under this \fBsmbfs\fR(7FS) mount.
The default behavior is \fBnoacl\fR, which presents files and
directories as owned by the owner of the mount point and having
permissions based on \fBfileperms\fR or \fBdirperms\fR.
With the \fBacl\fR mount option, files are presented with ACLs
obtained from the SMB server.
Setting the \fBacl\fR mount option is not advised unless the system
is joined to an Active Directory domain and using \fBldap\fR(1)
```

so it can correctly present ACL identities from the SMB server.

.It Cm dirperms Ns = Ns Ar octaltriplet Specifies the permissions to be assigned to directories. The value must be specified as an octal triplet, such as .Ql 755 . The default value for the directory mode is taken from the .Cm fileperms setting, with execute permission added where .Cm fileperms has read permission. .Pp Note that these permissions have no relation to the rights granted by the SMB

```
.RE

.sp
.ne 2
```

```
.na
\fB\fBdirperms=\fR\fIoctaltriplet\fR\fR
.ad
.sp .6
.RS 4n
Specifies the permissions to be assigned to directories. The value must be
specified as an octal triplet, such as \fB755\fR. The default value for the
directory mode is taken from the \fBfileperms\fR setting, with execute
permission added where \fBfileperms\fR has read permission.
.sp
Note that these permissions have no relation to the rights granted by the CIFS
server.
.It Cm fileperms Ns = Ns Ar octaltriplet
Specifies the permissions to be assigned to files.
The value must be specified as an octal triplet, such as
.Ql 644 .
The default value is
.Ql 700 .
.Pp
Note that these permissions have no relation to the rights granted by the SMB
.RE

.sp
.ne 2
.na
\fB\fBfileperms=\fR\fIoctaltriplet\fR\fR
.ad
.sp .6
.RS 4n
Specifies the permissions to be assigned to files. The value must be specified
as an octal triplet, such as \fB644\fR. The default value is \fB700\fR.
.sp
Note that these permissions have no relation to the rights granted by the CIFS
server.
.It Cm gid Ns = Ns Ar groupid
Assigns the specified group ID to files.
The default value is the group ID of the directory where the volume is mounted.
.It Cm intr Ns | Ns Cm nointr
Enable (or disable) cancellation of
.Xr smbfs 7FS
I/O operations when the user interrupts the calling thread (for example, by
hitting Ctrl-C while an operation is underway).
The default is
.Cm intr
(interruption enabled), so cancellation is normally allowed.
.It Cm noprompt
Suppresses the prompting for a password when mounting a share.
This property enables you to permit anonymous access to a share.
Anonymous access does not require a password.
.Pp
The
.Nm mount
operation fails if a password is required, the
.Cm noprompt
property is set, and no password is stored by the
.Nm smbutil Cm login
command.
.It Cm retry_count Ns = Ns Ar number
.RE

.sp
.ne 2
.na
\fB\fBgid=\fR\fIgroupid\fR\fR
.ad
.sp .6
```



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

.sp
.ne 2
.na
\fB\fBintr\fR|\fBnointr\fR\fR
.ad
.sp .6
.RS 4n
Enable (or disable) cancellation of \fBsmbfs\fR(7FS) I/O operations when the
user interrupts the calling thread (for example, by hitting Ctrl-C while an
operation is underway). The default is \fBintr\fR (interruption enabled), so
cancellation is normally allowed.
.RE

.sp
.ne 2
.na
\fB\fBnoprompt\fR\fR
.ad
.sp .6
.RS 4n
Suppresses the prompting for a password when mounting a share. This property
enables you to permit anonymous access to a share. Anonymous access does not
require a password.
.sp
The \fBmount\fR operation fails if a password is required, the \fBnoprompt\fR
property is set, and no password is stored by the \fBsmbutil login\fR command.
.RE

.sp
.ne 2
.na
\fB\fBretry_count=\fR\fInumber\fR\fR
.ad
.sp .6
.RS 4n
Specifies the number of SMBFS retries to attempt before the connection is
marked as broken.
By default, 4 attempts are made.
.Pp
The
.Cm retry_count
property value set by the
.Nm mount
command overrides the global value set in SMF or the value set in your
.Pa \&.nsmbrc
file.
.It Cm timeout Ns = Ns Ar seconds
Specifies the SMB request timeout.
By default, the timeout is 15 seconds.
.Pp
The
.Cm timeout
property value set by the
.Nm mount
command overrides the global value set in SMF or the value set in your
.Pa \&.nsmbrc
file.
.It Cm uid Ns = Ns Ar userid
Assigns the specified user ID files.
The default value is the owner ID of the directory where the volume is mounted.
.It Cm xattr Ns | Ns Cm noxattr
```

```
Enable (or disable) Extended Attributes in this mount point.
This option defaults to
.Cm xattr
(enabled Extended Attributes), but note: if the SMB server does not support SMB
"named streams",
.Xr smbfs 7FS
forces this option to
.Cm noxattr .
When a mount has the
.Cm noxattr
option, attempts to use Extended attributes fail with
.Er EINVAL .
.El
.It Fl O
Overlays mount.
Allow the file system to be mounted over an existing mount point, making the
underlying file system inaccessible.
If a mount is attempted on a pre-existing mount point without setting this flag,
the mount fails, producing the error "device busy."
.El
.Sh FILES
.Bl -tag -width Pa
.It Pa /etc/mnttab
Table of mounted file systems.
.It Pa /etc/dfs/fstypes
Default distributed file system type.
.It Pa /etc/vfstab
Table of automatically mounted resources.
.It Pa $HOME/.nsmbrc
User-settable mount point configuration file to store the description for each
connection.
.El
.Sh EXAMPLES
.Bl -tag -width Ds
.It Sy Example 1 No Mounting an SMBFS Share
The following example shows how to mount the
.Pa /tmp
share from the
.Em nano
server in the
.Em SALES
workgroup on the local
.Pa /mnt
mount point.
You must supply the password for the root user to successfully perform the mount
operation.
.Bd -literal
# mount -F smbfs "//SALES;root@nano.sfbay/tmp" /mnt
marked as broken. By default, 4 attempts are made.
.sp
The \fBretry_count\fR property value set by the \fBmount\fR command overrides
the global value set in SMF or the value set in your \fB\&.nsmbrc\fR file.
.RE

.sp
.ne 2
.na
\fB\fBtimeout=\fR\fIseconds\fR\fR
.ad
.sp .6
.RS 4n
Specifies the CIFS request timeout. By default, the timeout is 15 seconds.
.sp
The \fBtimeout\fR property value set by the \fBmount\fR command overrides the
global value set in SMF or the value set in your \fB\&.nsmbrc\fR file.
.RE
```



```
.sp
.ne 2
.na
\fB\fBuid=\fR\fIuserid\fR\fR
.ad
.sp .6
.RS 4n
Assigns the specified user ID files. The default value is the owner ID of the
directory where the volume is mounted.
.RE

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

.RE

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

.SH EXAMPLES
.LP
\fBExample 1 \fRMounting an SMBFS Share
.sp
.LP
The following example shows how to mount the \fB/tmp\fR share from the
\fBnano\fR server in the \fBSALES\fR workgroup on the local \fB/mnt\fR mount
point. You must supply the password for the \fBroot\fR user to successfully
perform the mount operation.

.sp
.in +2
.nf
# \fBmount -F smbfs "//SALES;root@nano.sfbay/tmp" /mnt\fR
Password:
.Ed
.It Sy Example 2 No Verifying That an SMBFS File System Is Mounted
The following example shows how to mount the
.Pa /tmp
share from the
.Em nano
server on the local
.Pa /mnt
mount point.
You must supply the password for the root user to successfully perform the mount
```

```
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/mnttab
353 file.
354 .Bd -literal
355 # grep root /etc/mnttab
364 .RS +4
365 .TP
366 .ie t \(bu
367 .el o
368 View the file system entry in the \fB/etc/mnttab\fR file.
369 .sp
370 .in +2
371 .nf
372 # \fBgrep root /etc/mnttab\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
```



```
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 .Ql 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 .Ql 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
384                       1882384   64448 1817936     4%    /mnt
385 .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 .Pa /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 ,
```

```
.Xr mountall 1M ,
.Xr svcadm 1M ,
.Xr acl 2 ,
.Xr fcntl 2 ,
.Xr link 2 ,
.Xr mknod 2 ,
.Xr mount 2 ,
.Xr symlink 2 ,
.Xr umount 2 ,
.Xr mnttab 4 ,
.Xr nsmbrc 4 ,
.Xr vfstab 4 ,
.Xr attributes 5 ,
.Xr pcfs 7FS ,
.Xr smbfs 7FS
.Sh AUTHORS
This manual page contains material originally authored by
.An Boris Popov
.Aq Mt bp@butya.kz ,
.Aq Mt bp@FreeBSD.org .
.Sh NOTES
The SMB client always attempts to use
.Xr gethostbyname 3NSL
to resolve host names.
If the host name cannot be resolved, the SMB client uses NetBIOS name
resolution (NBNS).
By default, the SMB client permits the use of NBNS to enable SMB clients in
Windows environments to work without additional configuration.
.Pp
Since NBNS has been exploited in the past, you might want to disable it.
To disable NBNS, set the
.Em nbns-enabled
service management facility property to
.Cm false .
By default,
.Em nbns-enabled
is set to
.Cm true .
.Pp
.fi
.in -2
.sp

.LP
\fBExample 3 \fRUnmounting a CIFS Share
.sp
.LP
This example assumes that a CIFS share has been mounted on the \fB/mnt\fR mount
point. The following command line unmounts the share from the mount point.

.sp
.in +2
.nf
# \fBumount /mnt\fR
.fi
.in -2
.sp

.SH FILES
.sp
.ne 2
.na
\fB\fB/etc/mnttab\fR\fR
.ad
.sp .6
.RS 4n
```



```
Table of mounted file systems.
.RE

.sp
.ne 2
.na
\fB\fB/etc/dfs/fstypes\fR\fR
.ad
.sp .6
.RS 4n
Default distributed file system type.
.RE

.sp
.ne 2
.na
\fB\fB/etc/vfstab\fR\fR
.ad
.sp .6
.RS 4n
Table of automatically mounted resources.
.RE

.sp
.ne 2
.na
\fB\fB$HOME/.nsmbrc\fR\fR
.ad
.sp .6
.RS 4n
User-settable mount point configuration file to store the description for each
connection.
.RE

.SH ATTRIBUTES
.sp
.LP
See the \fBattributes\fR(5) man page for descriptions of the following
attributes:
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Committed
.TE

.SH SEE ALSO
.sp
.LP
\fBldap\fR(1), \fBsmbutil\fR(1),
\fBmount\fR(1M), \fBmountall\fR(1M), \fBsvcadm\fR(1M),
\fBacl\fR(2), \fBfcntl\fR(2), \fBlink\fR(2), \fBmknod\fR(2), \fBmount\fR(2),
\fBsymlink\fR(2), \fBumount\fR(2), \fBmnttab\fR(4), \fBnsmbrc\fR(4),
\fBvfstab\fR(4), \fBattributes\fR(5), \fBpcfs\fR(7FS), \fBsmbfs\fR(7FS)
.SH AUTHORS
.sp
.LP
This manual page contains material originally authored by Boris Popov,
\fBbpATbutya.kz\fR, \fBbpATFreeBSD.org\fR.
.SH NOTES
.sp
```

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

```
**********************************************************
    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>
**********************************************************
```

```
/*
 * Copyright (c) 2000-2001, Boris Popov
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    This product includes software developed by Boris Popov.
 * 4. Neither the name of the author nor the names of any co-contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $Id: smbfs.h,v 1.30.100.1 2005/05/27 02:35:28 lindak Exp $
 */

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

#ifndef _SMBFS_SMBFS_H
#define _SMBFS_SMBFS_H

/*
 * FS-specific VFS structures for smbfs.
 * (per-mount stuff, etc.)
 *
 * This file used to have mount args stuff,
 * but that's now in sys/fs/smbfs_mount.h
 */

#include <sys/param.h>
#include <sys/fstyp.h>
#include <sys/avl.h>
#include <sys/list.h>
#include <sys/t_lock.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/fs/smbfs_mount.h>
#include <sys/zone.h>

/*
```



```
 * Path component length
 *
 * The generic fs code uses MAXNAMELEN to represent
 * what the largest component length is, but note:
 * that length DOES include the terminating NULL.
 * SMB_MAXFNAMELEN does NOT include the NULL.
 */
#define SMB_MAXFNAMELEN         (MAXNAMELEN-1)  /* 255 */

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

/* Mask values for smbmount structure sm_status field */
#define SM_STATUS_STATFS_BUSY 0x00000001 /* statvfs is in progress */
#define SM_STATUS_STATFS_WANT 0x00000002 /* statvfs wakeup is wanted */
#define SM_STATUS_TIMEO 0x00000004 /* this mount is not responding */
#define SM_STATUS_DEAD  0x00000010 /* connection gone - unmount this */

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

struct smbnode;
struct smb_share;

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

/*
 * Stuff returned by smbfs_smb_qfsattr
 * See [CIFS] SMB_QUERY_FS_ATTRIBUTE_INFO
 */
typedef struct smb_fs_attr_info {
        uint32_t        fsa_aflags;     /* Attr. flags [CIFS 4.1.6.6] */
        uint32_t        fsa_maxname;    /* max. component length */
        char            fsa_tname[FSTYPSZ]; /* type name, i.e. "NTFS" */
} smb_fs_attr_info_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   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>
**********************************************************
```

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

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 *      Copyright (c) 1983,1984,1985,1986,1987,1988,1989  AT&T.
 *      All rights reserved.
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/thread.h>
#include <sys/t_lock.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/errno.h>
#include <sys/buf.h>
#include <sys/stat.h>
#include <sys/cred.h>
#include <sys/kmem.h>
#include <sys/debug.h>
#include <sys/vmsystm.h>
#include <sys/flock.h>
#include <sys/share.h>
#include <sys/cmn_err.h>
#include <sys/tiuser.h>
#include <sys/sysmacros.h>
#include <sys/callb.h>
#include <sys/acl.h>
#include <sys/kstat.h>
#include <sys/signal.h>
#include <sys/list.h>
#include <sys/zone.h>

#include <netsmb/smb.h>
#include <netsmb/smb_conn.h>
#include <netsmb/smb_subr.h>
```



```
#include <smbfs/smbfs.h>
#include <smbfs/smbfs_node.h>
#include <smbfs/smbfs_subr.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/seg_vn.h>

#define ATTRCACHE_VALID(vp)     (gethrtime() < VTOSMB(vp)->r_attrtime)

static int smbfs_getattr_cache(vnode_t *, smbfattr_t *);
static void smbfattr_to_vattr(vnode_t *, smbfattr_t *, vattr_t *);
static void smbfattr_to_xvattr(smbfattr_t *, vattr_t *);
static int smbfs_getattr_otw(vnode_t *, struct smbfattr *, cred_t *);


/*
 * The following code provide zone support in order to perform an action
 * for each smbfs mount in a zone.  This is also where we would add
 * per-zone globals and kernel threads for the smbfs module (since
 * they must be terminated by the shutdown callback).
 */

struct smi_globals {
        kmutex_t        smg_lock;  /* lock protecting smg_list */
        list_t          smg_list;  /* list of SMBFS mounts in zone */
        boolean_t       smg_destructor_called;
};
typedef struct smi_globals smi_globals_t;

static zone_key_t smi_list_key;

/*
 * Attributes caching:
 *
 * Attributes are cached in the smbnode in struct vattr form.
 * There is a time associated with the cached attributes (r_attrtime)
 * which tells whether the attributes are valid. The time is initialized
 * to the difference between current time and the modify time of the vnode
 * when new attributes are cached. This allows the attributes for
 * files that have changed recently to be timed out sooner than for files
 * that have not changed for a long time. There are minimum and maximum
 * timeout values that can be set per mount point.
 */

/*
 * Helper for _validate_caches
 * Validate caches by checking cached attributes. If they have timed out
 * get the attributes from the server and compare mtimes. If mtimes are
 * different purge all caches for this vnode.
 */
int
smbfs_waitfor_purge_complete(vnode_t *vp)
{
        smbnode_t *np;
        k_sigset_t smask;

        np = VTOSMB(vp);
        if (np->r_serial != NULL && np->r_serial != curthread) {
                mutex_enter(&np->r_statelock);
                sigintr(&smask, VTOSMI(vp)->smi_flags & SMI_INT);
                while (np->r_serial != NULL) {
```

```
                        if (!cv_wait_sig(&np->r_cv, &np->r_statelock)) {
                                sigunintr(&smask);
                                mutex_exit(&np->r_statelock);
                                return (EINTR);
                        }
                }
                sigunintr(&smask);
                mutex_exit(&np->r_statelock);
        }
        return (0);
}

/*
 * Validate caches by checking cached attributes. If the cached
 * attributes have timed out, then get new attributes from the server.
 * As a side affect, this will do cache invalidation if the attributes
 * have changed.
 *
 * If the attributes have not timed out and if there is a cache
 * invalidation being done by some other thread, then wait until that
 * thread has completed the cache invalidation.
 */
int
smbfs_validate_caches(
        struct vnode *vp,
        cred_t *cr)
{
        struct smbfattr fa;
        int error;
        struct vattr va;

        if (ATTRCACHE_VALID(vp)) {
                error = smbfs_waitfor_purge_complete(vp);
                if (error)
                        return (error);
                return (0);
        }

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

/*
 * Purge all of the various data caches.
 *
 * Here NFS also had a flags arg to control what gets flushed.
 * We only have the page cache, so no flags arg.
 */
/* ARGSUSED */
/*ARGSUSED*/
void
smbfs_purge_caches(struct vnode *vp, cred_t *cr)
smbfs_purge_caches(struct vnode *vp)
{

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

        /*
```



```
         * Flush the page cache.
         */
        if (vn_has_cached_data(vp)) {
                (void) VOP_PUTPAGE(vp, (u_offset_t)0, 0, B_INVAL, cr, NULL);
        }

        /*
         * Here NFS has: Flush the readdir response cache.
         * No readdir cache in smbfs.
         */
#endif  /* not yet */
}

/*
 * Here NFS has:
 * nfs_purge_rddir_cache()
 * nfs3_cache_post_op_attr()
 * nfs3_cache_post_op_vattr()
 * nfs3_cache_wcc_data()
 */

/*
 * Check the attribute cache to see if the new attributes match
 * those cached.  If they do, the various 'data' caches are
 * considered to be good.  Otherwise, purge the cached data.
 */
static void
void
smbfs_cache_check(
        struct vnode *vp,
        struct smbfattr *fap,
        cred_t *cr)
        struct smbfattr *fap)
{
        smbnode_t *np;
        int purge_data = 0;
        int purge_acl = 0;

        np = VTOSMB(vp);
        mutex_enter(&np->r_statelock);

        /*
         * Compare with NFS macro: CACHE_VALID
         * If the mtime or size has changed,
         * purge cached data.
         */
        if (np->r_attr.fa_mtime.tv_sec != fap->fa_mtime.tv_sec ||
            np->r_attr.fa_mtime.tv_nsec != fap->fa_mtime.tv_nsec)
                purge_data = 1;
        if (np->r_attr.fa_size != fap->fa_size)
                purge_data = 1;

        if (np->r_attr.fa_ctime.tv_sec != fap->fa_ctime.tv_sec ||
            np->r_attr.fa_ctime.tv_nsec != fap->fa_ctime.tv_nsec)
                purge_acl = 1;

        if (purge_acl) {
                /* just invalidate r_secattr (XXX: OK?) */
                np->r_sectime = gethrtime();
        }

        mutex_exit(&np->r_statelock);

        if (purge_data)
                smbfs_purge_caches(vp, cr);
                smbfs_purge_caches(vp);
```

```
241 }

243 /*
188  * Set attributes cache for given vnode using vnode attributes.
189  * From NFS: nfs_attrcache_va
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         smbfattr_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 smbfattr *fap)
251 {
252         smbnode_t *np;
253         smbmntinfo_t *smi;
254         hrtime_t delta, now;
255         u_offset_t newsize;
256         vtype_t  vtype, oldvt;
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();

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_acdirmin)
303                                 delta = smi->smi_acdirmin;
304                         else if (delta > smi->smi_acdirmax)
305                                 delta = smi->smi_acdirmax;
306                 } else {
307                         if (delta < smi->smi_acregmin)
308                                 delta = smi->smi_acregmin;
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         oldvt = 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))) {
296         if (np->r_size != newsize) {
297 #if 0   /* not yet: mmap support */
298                 if (!vn_has_cached_data(vp) || ...)
299                         /* XXX: See NFS page cache code. */
300 #endif  /* not yet */
339                 /* OK to set the size. */
340                 np->r_size = newsize;
341         }

343         /*
344          * Here NFS has:
345          * nfs_setswaplike(vp, va);
346          * np->r_flags &= ~RWRITEATTR;
347          * (not needed here)
348          */

305         /* 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 }
_____unchanged_portion_omitted_

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
351 int
395 smbfs_getattr_otw(vnode_t *vp, struct smbfattr *fap, cred_t *cr)
396 {
397         struct smbnode *np;
398         struct smb_cred scred;
399         int error;

401         np = VTOSMB(vp);

403         /*
404          * Here NFS uses the ACL RPC (if smi_flags & SMI_ACL)
361          * NFS uses the ACL rpc here (if smi_flags & SMI_ACL)
405          * With SMB, getting the ACL is a significantly more
406          * expensive operation, so we do that only when asked
407          * for the uid/gid.  See smbfsgetattr().
408          */

410         /* Shared lock for (possible) n_fid use. */
411         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
412                 return (EINTR);
413         smb_credinit(&scred, cr);

415         bzero(fap, sizeof (*fap));
416         error = smbfs_smb_getfattr(np, fap, &scred);

418         smb_credrele(&scred);
419         smbfs_rw_exit(&np->r_lkserlock);

421         if (error) {
422                 /* Here NFS has: PURGE_STALE_FH(error, vp, cr) */
379                 /* NFS had: PURGE_STALE_FH(error, vp, cr) */
423                 smbfs_attrcache_remove(np);
424                 if (error == ENOENT || error == ENOTDIR) {
425                         /*
426                          * Getattr failed because the object was
427                          * removed or renamed by another client.
428                          * Remove any cached attributes under it.
429                          */
430                         smbfs_attrcache_prune(np);
431                 }
432                 return (error);
433         }

435         /*
436          * Here NFS has: nfs_cache_fattr(vap, fa, vap, t, cr);
393          * NFS: smbfs_cache_fattr(vap, fa, vap, t, cr);
437          * which did: fattr_to_vattr, nfs_attr_cache.
438          * We cache the fattr form, so just do the
439          * cache check and store the attributes.
```



```
440          */
441         smbfs_cache_check(vp, fap, cr);
398         smbfs_cache_check(vp, fap);
442         smbfs_attrcache_fa(vp, fap);

444         return (0);
445 }

447 /*
448  * Return either cached or remote attributes. If we get remote attrs,
405  * 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: nfsgetattr()
452  */
453 int
454 smbfsgetattr(vnode_t *vp, struct vattr *vap, cred_t *cr)
455 {
456         struct smbfattr fa;
457         smbmntinfo_t *smi;
458         uint_t mask;
459         int error;

461         smi = VTOSMI(vp);

463         ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);

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         }

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;

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);

497         return (0);
498 }
_____unchanged_portion_omitted_

596 /*
597  * Here NFS has:
598  *      nfs_async_... stuff
```

```
 * which we're not using (no async I/O), and:
 *      writerp(),
 *      nfs_putpages()
 *      nfs_invalidate_pages()
 * which we have in smbfs_vnops.c, and
 *      nfs_printfhandle()
 *      nfs_write_error()
 * not needed here.
 */

/*
 * Helper function for smbfs_sync
 *
 * Walk the per-zone list of smbfs mounts, calling smbfs_rflush
 * on each one.  This is a little tricky because we need to exit
 * the list mutex before each _rflush call and then try to resume
 * where we were in the list after re-entering the mutex.
 */
void
smbfs_flushall(cred_t *cr)
{
        smi_globals_t *smg;
        smbmntinfo_t *tmp_smi, *cur_smi, *next_smi;

        smg = zone_getspecific(smi_list_key, crgetzone(cr));
        ASSERT(smg != NULL);

        mutex_enter(&smg->smg_lock);
        cur_smi = list_head(&smg->smg_list);
        if (cur_smi == NULL) {
                mutex_exit(&smg->smg_lock);
                return;
        }
        VFS_HOLD(cur_smi->smi_vfsp);
        mutex_exit(&smg->smg_lock);

flush:
        smbfs_rflush(cur_smi->smi_vfsp, cr);

        mutex_enter(&smg->smg_lock);
        /*
         * Resume after cur_smi if that's still on the list,
         * otherwise restart at the head.
         */
        for (tmp_smi = list_head(&smg->smg_list);
            tmp_smi != NULL;
            tmp_smi = list_next(&smg->smg_list, tmp_smi))
                if (tmp_smi == cur_smi)
                        break;
        if (tmp_smi != NULL)
                next_smi = list_next(&smg->smg_list, tmp_smi);
        else
                next_smi = list_head(&smg->smg_list);

        if (next_smi != NULL)
                VFS_HOLD(next_smi->smi_vfsp);
        VFS_RELE(cur_smi->smi_vfsp);

        mutex_exit(&smg->smg_lock);

        if (next_smi != NULL) {
                cur_smi = next_smi;
                goto flush;
        }
}
```



```
/*
 * SMB Client initialization and cleanup.
 * Much of it is per-zone now.
 */


/* ARGSUSED */
static void *
smbfs_zone_init(zoneid_t zoneid)
{
        smi_globals_t *smg;

        smg = kmem_alloc(sizeof (*smg), KM_SLEEP);
        mutex_init(&smg->smg_lock, NULL, MUTEX_DEFAULT, NULL);
        list_create(&smg->smg_list, sizeof (smbmntinfo_t),
            offsetof(smbmntinfo_t, smi_zone_node));
        smg->smg_destructor_called = B_FALSE;
        return (smg);
}
_____unchanged_portion_omitted_
```

****
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  */

 209 /*
 205  * Update the local notion of the mtime of some directory.
 206  * See comments re. r_mtime in smbfs_node.h
 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_

/*
 * Below is the SMBFS-specific representation of a "node".
 * This struct is a mixture of Sun NFS and Darwin code.
 * Fields starting with "r_" came from NFS struct "rnode"
 * and fields starting with "n_" came from Darwin, or
 * were added during the Solaris port.  We have avoided
 * renaming fields so we would not cause excessive
 * changes in the code using this struct.
 *
 * Now using an AVL tree instead of hash lists, but kept the
 * "hash" in some member names and functions to reduce churn.
 * One AVL tree per mount replaces the global hash buckets.
 *
 * Notes carried over from the NFS code:
 *
 * The smbnode is the "inode" for remote files.  It contains all the
 * information necessary to handle remote file on the client side.
 *
 * Note on file sizes:  we keep two file sizes in the smbnode: the size
 * according to the client (r_size) and the size according to the server
 * (r_attr.fa_size).  They can differ because we modify r_size during a
 * write system call (smbfs_rdwr), before the write request goes over the
 * wire (before the file is actually modified on the server).  If an OTW
 * request occurs before the cached data is written to the server the file
 * size returned from the server (r_attr.fa_size) may not match r_size.
 * r_size is the one we use, in general.  r_attr.fa_size is only used to
 * determine whether or not our cached data is valid.
 *
 * Each smbnode has 3 locks associated with it (not including the smbnode
 * "hash" AVL tree and free list locks):
 *
 *      r_rwlock:       Serializes smbfs_write and smbfs_setattr requests
 *                      and allows smbfs_read requests to proceed in parallel.
 *                      Serializes reads/updates to directories.
 *
 *      r_lkserlock:    Serializes lock requests with map, write, and
 *                      readahead operations.
 *
 *      r_statelock:    Protects all fields in the smbnode except for
 *                      those listed below.  This lock is intented
 *                      to be held for relatively short periods of
 *                      time (not accross entire putpage operations,
 *                      for example).
 *
 * The following members are protected by the mutex smbfreelist_lock:
 *      r_freef
 *      r_freeb
 *
 * The following members are protected by the AVL tree rwlock:
 *      r_avl_node      (r__hdr.hdr_avl_node)
 *
 * Note: r_modaddr is only accessed when the r_statelock mutex is held.
 *      Its value is also controlled via r_rwlock.  It is assumed that
 *      there will be only 1 writer active at a time, so it safe to
 *      set r_modaddr and release r_statelock as long as the r_rwlock
 *      writer lock is held.
 *
 * 64-bit offsets: the code formerly assumed that atomic reads of
```



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

typedef struct smbnode {
        /* Our linkage in the node cache AVL tree (see above). */
        smbfs_node_hdr_t        r__hdr;

        /* short-hand names for r__hdr members */
#define r_avl_node      r__hdr.hdr_avl_node
#define n_rpath         r__hdr.hdr_n_rpath
#define n_rplen         r__hdr.hdr_n_rplen

        smbmntinfo_t    *n_mount;       /* VFS data */
        vnode_t         *r_vnode;       /* associated vnode */

        /*
         * Linkage in smbfreelist, for reclaiming nodes.
         * Lock for the free list is: smbfreelist_lock
         */
        struct smbnode  *r_freef;       /* free list forward pointer */
        struct smbnode  *r_freeb;       /* free list back pointer */

        smbfs_rwlock_t  r_rwlock;       /* serialize write/setattr requests */
        smbfs_rwlock_t  r_lkserlock;    /* serialize lock with other ops */
        kmutex_t        r_statelock;    /* protect (most) smbnode fields */

        /*
         * File handle, directory search handle,
         * and reference counts for them, etc.
         * Lock for these is: r_lkserlock
         */
        int             n_dirrefs;
        struct smbfs_fctx       *n_dirseq;      /* ff context */
        int             n_dirofs;       /* last ff offset */
        int             n_fidrefs;
        uint16_t        n_fid;          /* file handle */
        enum vtype      n_ovtype;       /* vnode type opened */
        uint32_t        n_rights;       /* granted rights */
        int             n_vcgenid;      /* gereration no. (reconnect) */

        /*
         * Misc. bookkeeping
         */
        cred_t          *r_cred;        /* current credentials */
        u_offset_t      r_nextr;        /* next read offset (read-ahead) */
        long            r_mapcnt;       /* count of mmapped pages */
        uint_t          r_inmap;        /* to serialize read/write and mmap */
        uint_t          r_count;        /* # of refs not reflect in v_count */
        uint_t          r_awcount;      /* # of outstanding async write */
        uint_t          r_gcount;       /* getattrs waiting to flush pages */
        uint_t          r_flags;        /* flags, see below */
        uint32_t        n_flag;         /* N--- flags below */
        uint32_t_t      n_flag;         /* NXXX flags below */
        uint_t          r_error;        /* async write error */
        kcondvar_t      r_cv;           /* condvar for blocked threads */
        avl_tree_t      r_dir;          /* cache of readdir responses */
        rddir_cache     *r_direof;      /* pointer to the EOF entry */
        u_offset_t      r_modaddr;      /* address for page in writenp */
        kthread_t       *r_serial;      /* id of purging thread */
        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         smbfattr_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 smbfattr_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_*

```
**********************************************************
   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_setfsize(struct smbnode *np, uint16_t fid, uint64_t newsize,
 168         struct smb_cred *scrp);

 170 int  smbfs_smb_getfattr(struct smbnode *np, struct smbfattr *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 nmlen,
 178         int xattr, uint32_t rights, struct smb_cred *scrp,
 179         uint16_t *fidp, uint32_t *rightsp, struct smbfattr *fap);
 180 int  smbfs_smb_tmpopen(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_tmpclose(struct smbnode *ssp, uint16_t fid,
 185         struct smb_cred *scrp);
 186 int  smbfs_smb_create(struct smbnode *dnp, const char *name, int nmlen,
 187         int xattr, uint32_t disp, 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 tnmlen, struct smb_cred *scrp);
 192 int  smbfs_smb_t2rename(struct smbnode *np, const char *tname, int tnmlen,
 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 tnmlen, 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 mbchain *mbp, struct smb_vc *vcp,
 205         struct smbnode *dnp, const char *name, int nmlen, uint8_t sep);
```



```
 206 int  smbfs_smb_lookup(struct smbnode *dnp, const char **namep, int *nmlenp,
 207         struct smbfattr *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_0extend(vnode_t *vp, uint16_t fid, len_t from, len_t to,
 214                 struct smb_cred *scredp, 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_subrinit(void);
 230 void smbfs_subrfini(void);
 231 int smbfs_clntinit(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 *vfsp, struct smbnode *srp);
 238 void smbfs_destroy_table(struct vfs *vfsp);
 239 void smbfs_rflush(struct vfs *vfsp, 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 smbfattr *fap);
 257 void smbfs_cache_check(struct vnode *vp, struct smbfattr *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_rmhash(struct smbnode *);

 267 /* See avl_create in smbfs_vfsops.c */
 268 void smbfs_init_hash_avl(avl_tree_t *);

 270 uint32_t smbfs_gethash(const char *rpath, int prlen);
```

```
271 uint32_t smbfs_getino(struct smbnode *dnp, const char *name, int nmlen);

273 extern struct smbfattr smbfs_fattr0;
274 smbnode_t *smbfs_node_findcreate(smbmntinfo_t *mi,
275     const char *dir, int dirlen,
276     const char *name, int nmlen,
277     char sep, struct smbfattr *fap);

279 int smbfs_nget(vnode_t *dvp, const char *name, int nmlen,
280         struct smbfattr *fap, vnode_t **vpp);

282 void smbfs_fname_tolocal(struct smbfs_fctx *ctx);
283 char    *smbfs_name_alloc(const char *name, int nmlen);
284 void    smbfs_name_free(const char *name, int nmlen);

286 int smbfs_readvnode(vnode_t *, uio_t *, cred_t *, struct vattr *);
287 int smbfs_writevnode(vnode_t *vp, uio_t *uiop, cred_t *cr,
288                      int ioflag, int timo);
289 int smbfsgetattr(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 smbfattr *fap,
308         struct smb_cred *scrp);
309 int smbfs_xa_findopen(struct smbfs_fctx *ctx, struct smbnode *dnp,
310         const char *name, int nmlen);
311 int smbfs_xa_findnext(struct smbfs_fctx *ctx, uint16_t limit);
312 int smbfs_xa_findclose(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_ */
```

```
**********************************************************
   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>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 *      Copyright (c) 1983,1984,1985,1986,1987,1988,1989  AT&T.
 *      All rights reserved.
 */
/*
 * Copyright (c) 2017 by Delphix. All rights reserved.
 */

/*
 * Node hash implementation initially borrowed from NFS (nfs_subr.c)
 * but then heavily modified. It's no longer an array of hash lists,
 * but an AVL tree per mount point.  More on this below.
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/bitmap.h>
#include <sys/dnlc.h>
#include <sys/kmem.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
#include <sys/fcntl.h>

#include <netsmb/smb_osdep.h>

#include <netsmb/smb.h>
#include <netsmb/smb_conn.h>
#include <netsmb/smb_subr.h>
#include <netsmb/smb_rq.h>

#include <smbfs/smbfs.h>
#include <smbfs/smbfs_node.h>
```



```
#include <smbfs/smbfs_subr.h>

/*
 * The AVL trees (now per-mount) allow finding an smbfs node by its
 * full remote path name.  It also allows easy traversal of all nodes
 * below (path wise) any given node.  A reader/writer lock for each
 * (per mount) AVL tree is used to control access and to synchronize
 * lookups, additions, and deletions from that AVL tree.
 *
 * Previously, this code use a global array of hash chains, each with
 * its own rwlock.  A few struct members, functions, and comments may
 * still refer to a "hash", and those should all now be considered to
 * refer to the per-mount AVL tree that replaced the old hash chains.
 * (i.e. member smi_hash_lk, function sn_hashfind, etc.)
 *
 * The smbnode freelist is organized as a doubly linked list with
 * a head pointer.  Additions and deletions are synchronized via
 * a single mutex.
 *
 * In order to add an smbnode to the free list, it must be linked into
 * the mount's AVL tree and the exclusive lock for the AVL must be held.
 * If an smbnode is not linked into the AVL tree, then it is destroyed
 * because it represents no valuable information that can be reused
 * about the file.  The exclusive lock for the AVL tree must be held
 * in order to prevent a lookup in the AVL tree from finding the
 * smbnode and using it and assuming that the smbnode is not on the
 * freelist.  The lookup in the AVL tree will have the AVL tree lock
 * held, either exclusive or shared.
 *
 * The vnode reference count for each smbnode is not allowed to drop
 * below 1.  This prevents external entities, such as the VM
 * subsystem, from acquiring references to vnodes already on the
 * freelist and then trying to place them back on the freelist
 * when their reference is released.  This means that the when an
 * smbnode is looked up in the AVL tree, then either the smbnode
 * is removed from the freelist and that reference is tranfered to
 * the new reference or the vnode reference count must be incremented
 * accordingly.  The mutex for the freelist must be held in order to
 * accurately test to see if the smbnode is on the freelist or not.
 * The AVL tree lock might be held shared and it is possible that
 * two different threads may race to remove the smbnode from the
 * freelist.  This race can be resolved by holding the mutex for the
 * freelist.  Please note that the mutex for the freelist does not
 * need to held if the smbnode is not on the freelist.  It can not be
 * placed on the freelist due to the requirement that the thread
 * putting the smbnode on the freelist must hold the exclusive lock
 * for the AVL tree and the thread doing the lookup in the AVL tree
 * is holding either a shared or exclusive lock for the AVL tree.
 *
 * The lock ordering is:
 *
 *      AVL tree lock -> vnode lock
 *      AVL tree lock -> freelist lock
 */

static kmutex_t smbfreelist_lock;
static smbnode_t *smbfreelist = NULL;
static ulong_t  smbnodenew = 0;
long    nsmbnode = 0;

static struct kmem_cache *smbnode_cache;

static const vsecattr_t smbfs_vsa0 = { 0 };

/*
 * Mutex to protect the following variables:
```

```
 *      smbfs_major
 *      smbfs_minor
 */
kmutex_t smbfs_minor_lock;
int smbfs_major;
int smbfs_minor;

/* See smbfs_node_findcreate() */
struct smbfattr smbfs_fattr0;

/*
 * Local functions.
 * SN for Smb Node
 */
static void sn_rmfree(smbnode_t *);
static void sn_inactive(smbnode_t *);
static void sn_addhash_locked(smbnode_t *, avl_index_t);
static void sn_rmhash_locked(smbnode_t *);
static void sn_destroy_node(smbnode_t *);
void smbfs_kmem_reclaim(void *cdrarg);

static smbnode_t *
sn_hashfind(smbmntinfo_t *, const char *, int, avl_index_t *);

static smbnode_t *
make_smbnode(smbmntinfo_t *, const char *, int, int *);

/*
 * Free the resources associated with an smbnode.
 * Note: This is different from smbfs_inactive
 *
 * From NFS: nfs_subr.c:rinactive
 * NFS: nfs_subr.c:rinactive
 */
static void
sn_inactive(smbnode_t *np)
{
        vsecattr_t      ovsa;
        cred_t          *oldcr;
        char            *orpath;
        int             orplen;
        vnode_t         *vp;

        /*
         * Here NFS has:
         * Flush and invalidate all pages (done by caller)
         * Flush and invalidate all pages (todo)
         * Free any held credentials and caches...
         * etc.  (See NFS code)
         */
        mutex_enter(&np->r_statelock);

        ovsa = np->r_secattr;
        np->r_secattr = smbfs_vsa0;
        np->r_sectime = 0;

        oldcr = np->r_cred;
        np->r_cred = NULL;

        orpath = np->n_rpath;
        orplen = np->n_rplen;
        np->n_rpath = NULL;
        np->n_rplen = 0;

        mutex_exit(&np->r_statelock);
```



```
        vp = SMBTOV(np);
        if (vn_has_cached_data(vp)) {
                ASSERT3P(vp,==,NULL);
        }

        if (ovsa.vsa_aclentp != NULL)
                kmem_free(ovsa.vsa_aclentp, ovsa.vsa_aclentsz);

        if (oldcr != NULL)
                crfree(oldcr);

        if (orpath != NULL)
                kmem_free(orpath, orplen + 1);
}

/*
 * Find and optionally create an smbnode for the passed
 * mountinfo, directory, separator, and name.  If the
 * desired smbnode already exists, return a reference.
 * If the file attributes pointer is non-null, the node
 * is created if necessary and linked into the AVL tree.
 *
 * Callers that need a node created but don't have the
 * real attributes pass smbfs_fattr0 to force creation.
 *
 * Note: make_smbnode() may upgrade the "hash" lock to exclusive.
 *
 * Based on NFS: nfs_subr.c:makenfsnode
 * NFS: nfs_subr.c:makenfsnode
 */
smbnode_t *
smbfs_node_findcreate(
        smbmntinfo_t *mi,
        const char *dirnm,
        int dirlen,
        const char *name,
        int nmlen,
        char sep,
        struct smbfattr *fap)
{
        char tmpbuf[256];
        size_t rpalloc;
        char *p, *rpath;
        int rplen;
        smbnode_t *np;
        vnode_t *vp;
        int newnode;

        /*
         * Build the search string, either in tmpbuf or
         * in allocated memory if larger than tmpbuf.
         */
        rplen = dirlen;
        if (sep != '\0')
                rplen++;
        rplen += nmlen;
        if (rplen < sizeof (tmpbuf)) {
                /* use tmpbuf */
                rpalloc = 0;
                rpath = tmpbuf;
        } else {
                rpalloc = rplen + 1;
                rpath = kmem_alloc(rpalloc, KM_SLEEP);
        }
        p = rpath;
        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);

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

272         if (rpalloc)
273                 kmem_free(rpath, rpalloc);

275         if (fap == NULL) {
276                 /*
277                  * Caller is "just looking" (no create)
278                  * so np may or may not be NULL here.
279                  * Either way, we're done.
280                  */
281                 return (np);
282         }

284         /*
285          * We should have a node, possibly created.
286          * Do we have (real) attributes to apply?
287          */
288         ASSERT(np != NULL);
289         if (fap == &smbfs_fattr0)
290                 return (np);

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

299         /*
300          * Note NFS sets vp->v_type here, assuming it
301          * can never change for the life of a node.
302          * We allow v_type to change, and set it in
303          * smbfs_attrcache().  Also: mode, uid, gid
304          */
305         return (np);
306 }

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



```
311  */

313 /*
314  * Find or create an smbnode.
315  * From NFS: nfs_subr.c:make_rnode
314  * 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 *tnp;
326         vnode_t *vp;
327         vfs_t *vfsp;
328         avl_index_t where;
329         char *new_rpath = NULL;

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

334 start:
335         np = sn_hashfind(mi, rpath, rplen, NULL);
336         if (np != NULL) {
337                 *newnode = 0;
338                 return (np);
339         }

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

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

353                 vp = SMBTOV(np);

355                 if (np->r_flags & RHASHED) {
356                         smbmntinfo_t *tmp_mi = np->n_mount;
357                         ASSERT(tmp_mi != NULL);
358                         rw_enter(&tmp_mi->smi_hash_lk, RW_WRITER);
359                         mutex_enter(&vp->v_lock);
360                         if (vp->v_count > 1) {
361                                 VN_RELE_LOCKED(vp);
362                                 mutex_exit(&vp->v_lock);
363                                 rw_exit(&tmp_mi->smi_hash_lk);
364                                 /* start over */
365                                 rw_enter(&mi->smi_hash_lk, RW_READER);
366                                 goto start;
367                         }
368                         mutex_exit(&vp->v_lock);
369                         sn_rmhash_locked(np);
370                         rw_exit(&tmp_mi->smi_hash_lk);
371                 }

373                 sn_inactive(np);

375                 mutex_enter(&vp->v_lock);
```

```
376                 if (vp->v_count > 1) {
377                         VN_RELE_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_invalid(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_RELE(vp->v_vfsp);
398                 vn_reinit(vp);
399         } else {
400                 /*
401                  * allocate and initialize a new smbnode
402                  */
403                 vnode_t *new_vp;

405                 mutex_exit(&smbfreelist_lock);

407                 np = kmem_cache_alloc(smbnode_cache, KM_SLEEP);
408                 new_vp = vn_alloc(KM_SLEEP);

410                 atomic_inc_ulong((ulong_t *)&smbnodenew);
411                 vp = new_vp;
412         }

414         /*
415          * Allocate and copy the rpath we'll need below.
416          */
417         new_rpath = kmem_alloc(rplen + 1, KM_SLEEP);
418         bcopy(rpath, new_rpath, rplen);
419         new_rpath[rplen] = '\0';

421         /* Initialize smbnode_t */
422         bzero(np, sizeof (*np));

424         smbfs_rw_init(&np->r_rwlock, NULL, RW_DEFAULT, NULL);
425         smbfs_rw_init(&np->r_lkserlock, NULL, RW_DEFAULT, NULL);
426         mutex_init(&np->r_statelock, NULL, MUTEX_DEFAULT, NULL);
427         cv_init(&np->r_cv, NULL, CV_DEFAULT, NULL);
428         /* cv_init(&np->r_commit.c_cv, NULL, CV_DEFAULT, NULL); */

430         np->r_vnode = vp;
431         np->n_mount = mi;

433         np->n_fid = SMB_FID_UNUSED;
434         np->n_uid = mi->smi_uid;
435         np->n_gid = mi->smi_gid;
436         /* Leave attributes "stale." */

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 (rddir_cache),
443             offsetof(rddir_cache, tree));
444 #endif

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

450         /*
451          * We entered with mi->smi_hash_lk held (reader).
452          * Retake it now, (as the writer).
453          * Will return with it held.
454          */
455         rw_enter(&mi->smi_hash_lk, RW_WRITER);

457         /*
458          * There is a race condition where someone else
459          * may alloc the smbnode while no locks are held,
460          * so check again and recover if found.
461          */
462         tnp = sn_hashfind(mi, rpath, rplen, &where);
463         if (tnp != NULL) {
464                 /*
465                  * Lost the race.  Put the node we were building
466                  * on the free list and return the one we found.
467                  */
468                 rw_exit(&mi->smi_hash_lk);
469                 kmem_free(new_rpath, rplen + 1);
470                 smbfs_addfree(np);
471                 rw_enter(&mi->smi_hash_lk, RW_READER);
472                 *newnode = 0;
473                 return (tnp);
474         }

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);

485         sn_addhash_locked(np, where);
486         *newnode = 1;
487         return (np);
488 }

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

```
501 void
502 smbfs_addfree(smbnode_t *np)
503 {
504         vnode_t *vp;
505         struct vfs *vfsp;
506         smbmntinfo_t *mi;

508         ASSERT(np->r_freef == NULL && np->r_freeb == NULL);

510         vp = SMBTOV(np);
511         ASSERT(vp->v_count >= 1);

513         vfsp = vp->v_vfsp;
514         mi = VFTOSMI(vfsp);

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->vfs_flag & VFS_UNMOUNTED) ||
528             (smbnodenew > nsmbnode))) {

530                 /* Try to destroy this node. */

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_RELE_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_rmhash_locked(np);
547                         rw_exit(&mi->smi_hash_lk);
548                 }

550                 sn_inactive(np);

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
```



```
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_RELE_LOCKED(vp);
574                         mutex_exit(&vp->v_lock);
575                         return;
576                 }
577                 mutex_exit(&vp->v_lock);

579                 sn_destroy_node(np);
580                 return;
581         }

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

593         mutex_enter(&vp->v_lock);
594         if (vp->v_count > 1) {
595                 VN_RELE_LOCKED(vp);
596                 mutex_exit(&vp->v_lock);
597                 rw_exit(&mi->smi_hash_lk);
598                 return;
599         }
600         mutex_exit(&vp->v_lock);

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

618         rw_exit(&mi->smi_hash_lk);
619 }

621 /*
622  * Remove an smbnode from the free list.
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
630  * NFS: nfs_subr.c:rp_rmfree
628  */
629 static void
630 sn_rmfree(smbnode_t *np)
631 {
```

```
633         ASSERT(MUTEX_HELD(&smbfreelist_lock));
634         ASSERT(np->r_freef != NULL && np->r_freeb != NULL);

636         if (np == smbfreelist) {
637                 smbfreelist = np->r_freef;
638                 if (np == smbfreelist)
639                         smbfreelist = NULL;
640         }

642         np->r_freeb->r_freef = np->r_freef;
643         np->r_freef->r_freeb = np->r_freeb;

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
656  * NFS: nfs_subr.c:rp_addhash
654  */
655 static void
656 sn_addhash_locked(smbnode_t *np, avl_index_t where)
657 {
658         smbmntinfo_t *mi = np->n_mount;

660         ASSERT(RW_WRITE_HELD(&mi->smi_hash_lk));

662         mutex_enter(&np->r_statelock);
663         if ((np->r_flags & RHASHED) == 0) {
664                 avl_insert(&mi->smi_hash_avl, np, where);
665                 np->r_flags |= RHASHED;
666         }
667         mutex_exit(&np->r_statelock);
668 }

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
678  * NFS: nfs_subr.c:rp_rmhash_locked
676  */
677 static void
678 sn_rmhash_locked(smbnode_t *np)
679 {
680         smbmntinfo_t *mi = np->n_mount;

682         ASSERT(RW_WRITE_HELD(&mi->smi_hash_lk));

684         mutex_enter(&np->r_statelock);
685         if ((np->r_flags & RHASHED) != 0) {
686                 np->r_flags &= ~RHASHED;
687                 avl_remove(&mi->smi_hash_avl, np);
688         }
689         mutex_exit(&np->r_statelock);
690 }
```

*unchanged_portion_omitted_*

```
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
715  * NFS: nfs_subr.c:rfind
713  */
714 static smbnode_t *
715 sn_hashfind(
716         smbmntinfo_t *mi,
717         const char *rpath,
718         int rplen,
719         avl_index_t *pwhere) /* optional */
720 {
721         smbfs_node_hdr_t nhdr;
722         smbnode_t *np;
723         vnode_t *vp;

725         ASSERT(RW_LOCK_HELD(&mi->smi_hash_lk));

727         bzero(&nhdr, sizeof (nhdr));
728         nhdr.hdr_n_rpath = (char *)rpath;
729         nhdr.hdr_n_rplen = rplen;

731         /* See smbfs_node_cmp below. */
732         np = avl_find(&mi->smi_hash_avl, &nhdr, pwhere);

734         if (np == NULL)
735                 return (NULL);

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

760         return (np);
761 }
```

*unchanged_portion_omitted_*

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


858 /*
859  * Return 1 if there is a active vnode belonging to this vfs in the
860  * smbnode cache.
861  *
```

```
 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
 870  * NFS: nfs_subr.c:check_rtable
 868  *
 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         smbmntinfo_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 == rtnp)
 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;

 915         }
 916         rw_exit(&mi->smi_hash_lk);

 918         return (busycnt);
 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
 927  *
 928  * In here, we're normally destrying all or most of the AVL tree,
 929  * so the natural choice is to use avl_destroy_nodes.  However,
 930  * there may be a few busy nodes that should remain in the AVL
 931  * tree when we're done.  The solution: use a temporary tree to
 932  * hold the busy nodes until we're done destroying the old tree,
 933  * then copy the temporary tree over the (now emtpy) real tree.
 934  */
 935 void
 936 smbfs_destroy_table(struct vfs *vfsp)
 937 {
 938         avl_tree_t tmp_avl;
 939         smbmntinfo_t *mi;
 940         smbnode_t *np;
 941         smbnode_t *rlist;
 942         void *v;

 944         mi = VFTOSMI(vfsp);
 945         rlist = NULL;
 946         smbfs_init_hash_avl(&tmp_avl);

 948         rw_enter(&mi->smi_hash_lk, RW_WRITER);
 949         v = NULL;
 950         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_rmfree(np);
 966                         mutex_exit(&smbfreelist_lock);

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

 977                         /*
 978                          * Add to the list of nodes to destroy.
 979                          * Borrowing avl_child[0] for this list.
 980                          */
 981                         np->r_avl_node.avl_child[0] =
 982                             (struct avl_node *)rlist;
 983                         rlist = np;
 984                 }
 985         }
 986         avl_destroy(&mi->smi_hash_avl);

 988         /*
 989          * Replace the (now destroyed) "hash" AVL with the
 990          * temporary AVL, which restores the busy nodes.
 991          */
```

```
 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          * smbnode, 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 smbnode
1009  * and then the smbnode itself.  Note: sn_inactive has been called.
1010  *
1011  * From NFS: nfs_subr.c:destroy_rnode
1014  * NFS: nfs_subr.c:destroy_rnode
1012  */
1013 static void
1014 sn_destroy_node(smbnode_t *np)
1015 {
1016         vnode_t *vp;
1017         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->n_rpath == NULL);
1028         ASSERT(!(np->r_flags & RHASHED));
1029         ASSERT(np->r_freef == NULL && np->r_freeb == NULL);
1030         atomic_dec_ulong((ulong_t *)&smbnodenew);
1031         vn_invalid(vp);
1032         vn_free(vp);
1033         kmem_cache_free(smbnode_cache, np);
1034         VFS_RELE(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.
1042  * Used by nfs_sync and by nfs_unmount.
1041  */
1042 /*ARGSUSED*/
1043 void
1044 smbfs_rflush(struct vfs *vfsp, cred_t *cr)
1045 {
1046         smbmntinfo_t *mi;
1047         smbnode_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 = kmem_alloc(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_RELE(vp);
1105         }

1107         kmem_free(vplist, num * sizeof (vnode_t *));
1048         /* Todo: mmap support. */
1108 }

1110 /* Here NFS has access cache stuff (nfs_subr.c) not used here */
1051 /* 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;

1127         np = VTOSMB(vp);

1129         if (cmd == DIRECTIO_ON) {

1131                 if (np->r_flags & RDIRECTIO)
1132                         return (0);

1134                 /*
1135                  * Flush the page cache.
1136                  */

1138                 (void) VOP_RWLOCK(vp, V_WRITELOCK_TRUE, NULL);

1140                 if (np->r_flags & RDIRECTIO) {
1141                         VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, NULL);
1142                         return (0);
1143                 }

1145                 /* Here NFS also checks ->r_awcount */
1146                 if (vn_has_cached_data(vp) &&
1147                     (np->r_flags & RDIRTY) != 0) {
1148                         error = VOP_PUTPAGE(vp, (offset_t)0, (uint_t)0,
1149                             B_INVAL, cr, NULL);
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                 }

1162                 mutex_enter(&np->r_statelock);
1163                 np->r_flags |= RDIRECTIO;
1164                 mutex_exit(&np->r_statelock);
1165                 VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, NULL);
1166                 return (0);
1167         }

1169         if (cmd == DIRECTIO_OFF) {
1170                 mutex_enter(&np->r_statelock);
1171                 np->r_flags &= ~RDIRECTIO;      /* disable direct mode */
1172                 mutex_exit(&np->r_statelock);
1173                 return (0);
1174         }

1176         return (EINVAL);
1177 }

1179 static kmutex_t smbfs_newnum_lock;
1180 static uint32_t smbfs_newnum_val = 0;

1182 /*
1183  * Return a number 0..0xffffffff that's different from the last
1184  * 0xffffffff numbers this returned.  Used for unlinked files.
1185  * From NFS nfs_subr.c newnum
```



```
1059  * (This too was copied from nfs_subr.c)
1186  */
1187 uint32_t
1188 smbfs_newnum(void)
1189 {
1190         uint32_t id;

1192         mutex_enter(&smbfs_newnum_lock);
1193         if (smbfs_newnum_val == 0)
1194                 smbfs_newnum_val = (uint32_t)gethrestime_sec();
1195         id = smbfs_newnum_val++;
1196         mutex_exit(&smbfs_newnum_lock);
1197         return (id);
1198 }
```

*unchanged_portion_omitted_*

```
1215 /*
1216  * initialize resources that are used by smbfs_subr.c
1217  * this is called from the _init() routine (by the way of smbfs_clntinit())
1218  *
1219  * From NFS: nfs_subr.c:nfs_subrinit
1093  * NFS: nfs_subr.c:nfs_subrinit
1220  */
1221 int
1222 smbfs_subrinit(void)
1223 {
1224         ulong_t nsmbnode_max;

1226         /*
1227          * Allocate and initialize the smbnode cache
1228          */
1229         if (nsmbnode <= 0)
1230                 nsmbnode = ncsize; /* dnlc.h */
1231         nsmbnode_max = (ulong_t)((kmem_maxavail() >> 2) /
1232             sizeof (struct smbnode));
1233         if (nsmbnode > nsmbnode_max || (nsmbnode == 0 && ncsize == 0)) {
1234                 zcmn_err(GLOBAL_ZONEID, CE_NOTE,
1235                     "setting nsmbnode to max value of %ld", nsmbnode_max);
1236                 nsmbnode = nsmbnode_max;
1237         }

1239         smbnode_cache = kmem_cache_create("smbnode_cache", sizeof (smbnode_t),
1240             0, NULL, NULL, smbfs_kmem_reclaim, NULL, NULL, 0);

1242         /*
1243          * Initialize the various mutexes and reader/writer locks
1244          */
1245         mutex_init(&smbfreelist_lock, NULL, MUTEX_DEFAULT, NULL);
1246         mutex_init(&smbfs_minor_lock, NULL, MUTEX_DEFAULT, NULL);

1248         /*
1249          * Assign unique major number for all smbfs mounts
1250          */
1251         if ((smbfs_major = getudev()) == -1) {
1252                 zcmn_err(GLOBAL_ZONEID, CE_WARN,
1253                     "smbfs: init: can't get unique device number");
1254                 smbfs_major = 0;
1255         }
1256         smbfs_minor = 0;

1258         return (0);
1259 }

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 {

1269         /*
1270          * Destroy the smbnode cache
1271          */
1272         kmem_cache_destroy(smbnode_cache);

1274         /*
1275          * Destroy the various mutexes and reader/writer locks
1276          */
1277         mutex_destroy(&smbfreelist_lock);
1278         mutex_destroy(&smbfs_minor_lock);
1279 }
```

_____***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 */
```

```
**********************************************************
   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.
895          * Cross-zone calls are OK here, since this translates to a
896          * VOP_PUTPAGE(B_ASYNC), which gets picked up by the right zone.
898          */
899         if (flag & SYNC_ATTR)
900                 return (0);

902         if (vfsp == NULL) {
903                 /*
904                  * Flush ALL smbfs mounts in this zone.
905                  */
906                 smbfs_flushall(cr);
907                 return (0);
898         if (!(flag & SYNC_ATTR) && mutex_tryenter(&smbfs_syncbusy) != 0) {
899                 smbfs_rflush(vfsp, cr);
900                 mutex_exit(&smbfs_syncbusy);
908         }

910         smbfs_rflush(vfsp, cr);

912         return (0);
913 }

915 /*
916  * Initialization routine for VFS routines.  Should only be called once
917  */
918 int
919 smbfs_vfsinit(void)
920 {
912         mutex_init(&smbfs_syncbusy, NULL, MUTEX_DEFAULT, NULL);
921         return (0);
922 }

924 /*
925  * Shutdown routine for VFS routines.  Should only be called once
926  */
927 void
928 smbfs_vfsfini(void)
929 {
922         mutex_destroy(&smbfs_syncbusy);
930 }
______unchanged_portion_omitted_
```

```
**********************************************************
  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>
**********************************************************
```

```
/*
 * Copyright (c) 2000-2001 Boris Popov
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    This product includes software developed by Boris Popov.
 * 4. Neither the name of the author nor the names of any co-contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $Id: smbfs_vnops.c,v 1.128.36.1 2005/05/27 02:35:28 lindak Exp $
 */

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

/*
 * Vnode operations
 *
 * This file is similar to nfs3_vnops.c
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/cred.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/filio.h>
#include <sys/uio.h>
#include <sys/dirent.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
```



```
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vfs_opreg.h>
#include <sys/policy.h>
#include <sys/sdt.h>
#include <sys/zone.h>
#include <sys/vmsystm.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/seg_kpm.h>
#include <vm/seg_vn.h>

#include <netsmb/smb_osdep.h>
#include <netsmb/smb.h>
#include <netsmb/smb_conn.h>
#include <netsmb/smb_subr.h>

#include <smbfs/smbfs.h>
#include <smbfs/smbfs_node.h>
#include <smbfs/smbfs_subr.h>

#include <sys/fs/smbfs_ioctl.h>
#include <fs/fs_subr.h>

/*
 * We assign directory offsets like the NFS client, where the
 * offset increments by _one_ after each directory entry.
 * Further, the entries "." and ".." are always at offsets
 * zero and one (respectively) and the "real" entries from
 * the server appear at offsets starting with two.  This
 * macro is used to initialize the n_dirofs field after
 * setting n_dirseq with a _findopen call.
 */
#define FIRST_DIROFS    2

/*
 * These characters are illegal in NTFS file names.
 * ref: http://support.microsoft.com/kb/147438
 *
 * Careful!  The check in the XATTR case skips the
 * first character to allow colon in XATTR names.
 */
static const char illegal_chars[] = {
        ':',    /* colon - keep this first! */
        '\\',   /* back slash */
        '/',    /* slash */
        '*',    /* asterisk */
        '?',    /* question mark */
        '"',    /* double quote */
        '<',    /* less than sign */
        '>',    /* greater than sign */
        '|',    /* vertical bar */
        0
};

/*
 * Turning this on causes nodes to be created in the cache
 * during directory listings, normally avoiding a second
 * OtW attribute fetch just after a readdir.
 */
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 *nnm, 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_readvdir(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_rdwrlbn(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_frlock(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;

180 /*
189  * Most unimplemented ops will return ENOSYS because of fs_nosys().
190  * The only ops where that won't work are ACCESS (due to open(2)
191  * failures) and ... (anything else left?)
192  */
193 const fs_operation_def_t smbfs_vnodeops_template[] = {
194         { VOPNAME_OPEN,         { .vop_open = smbfs_open } },
195         { VOPNAME_CLOSE,        { .vop_close = smbfs_close } },
196         { VOPNAME_READ,         { .vop_read = smbfs_read } },
197         { VOPNAME_WRITE,        { .vop_write = smbfs_write } },
198         { VOPNAME_IOCTL,        { .vop_ioctl = smbfs_ioctl } },
199         { VOPNAME_GETATTR,      { .vop_getattr = smbfs_getattr } },
200         { VOPNAME_SETATTR,      { .vop_setattr = smbfs_setattr } },
201         { VOPNAME_ACCESS,       { .vop_access = smbfs_access } },
202         { VOPNAME_LOOKUP,       { .vop_lookup = smbfs_lookup } },
203         { VOPNAME_CREATE,       { .vop_create = smbfs_create } },
204         { VOPNAME_REMOVE,       { .vop_remove = smbfs_remove } },
205         { VOPNAME_LINK,         { .error = fs_nosys } }, /* smbfs_link, */
206         { VOPNAME_RENAME,       { .vop_rename = smbfs_rename } },
207         { VOPNAME_MKDIR,        { .vop_mkdir = smbfs_mkdir } },
```

```
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_SETSECATTR,   { .vop_setsecattr = smbfs_setsecattr } },
230         { VOPNAME_GETSECATTR,   { .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         smbfattr_t      fa;
193         u_int32_t       rights, rightsrcvd;
194         u_int16_t       fid, oldfid;
195         int             oldgenid;
196         struct smb_cred scred;
197         smbmntinfo_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->v_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.
287          * XXX: Per-open instance nodes whould help.
232          */
233         if (np->n_ovtype == VNON) {
234                 ASSERT(np->n_dirrefs == 0);
235                 ASSERT(np->n_fidrefs == 0);
236         } else if (np->n_ovtype != vp->v_type) {
237                 SMBVDEBUG("open n_ovtype=%d v_type=%d\n",
238                     np->n_ovtype, vp->v_type);
239                 error = EACCES;
240                 goto out;
241         }

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_findopen(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;

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;

304         bzero(&fa, sizeof (fa));
305         error = smbfs_smb_open(np,
306             NULL, 0, 0, /* name nmlen xattr */
307             rights, &scred,
308             &fid, &rightsrcvd, &fa);
309         if (error)
310                 goto out;
311         smbfs_attrcache_fa(vp, &fa);

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 = rightsrcvd;
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         }

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);

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;

356 out:
357         smb_credrele(&scred);
358         smbfs_rw_exit(&np->r_lkserlock);
359         return (error);
360 }

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         smbmntinfo_t    *smi;
369         struct smb_cred scred;
370         int error = 0;

372         np = VTOSMB(vp);
373         smi = VTOSMI(vp);

375         /*
376          * Don't "bail out" for VFS_UNMOUNTED here,
377          * as we want to do cleanup, etc.
378          */

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                  * Returning an error here is the sane thing to do.  A
394                  * subsequent call to VN_RELE() which translates to a
395                  * smbfs_inactive() will clean up state: if the zone of the
396                  * vnode's origin is still alive and kicking, an async worker
397                  * thread will handle the request (from the correct zone), and
398                  * everything (minus the final smbfs_getattr_otw() call) should
399                  * be OK. If the zone is going away smbfs_async_inactive() will
400                  * throw away cached pages inline.
401                  */
402                 return (EIO);
403         }

405         /*
406          * If we are using local locking for this filesystem, then
407          * release all of the SYSV style record locks.  Otherwise,
408          * we are doing network locking and we need to release all
409          * of the network locks.  All of the locks held by this
410          * process on this file are released no matter what the
411          * incoming reference count is.
412          */
413         if (smi->smi_flags & SMI_LLOCK) {
414                 pid_t pid = ddi_get_pid();
415                 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          */
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         /* NFS has DNLC purge here. */
433         /*
434          * If the file was open for write and there are pages,
435          * then make sure dirty pages written back.
436          *
437          * NFS does this async when "close-to-open" is off
438          * (MI_NOCTO flag is set) to avoid blocking the caller.
439          * For now, always do this synchronously (no B_ASYNC).
440          */
441         if ((flag & FWRITE) && vn_has_cached_data(vp)) {
442                 error = smbfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
443                 if (error == EAGAIN)
444                         error = 0;
445         }
446         if (error == 0) {
447                 mutex_enter(&np->r_statelock);
448                 np->r_flags &= ~RSTALE;
449                 np->r_error = 0;
450                 mutex_exit(&np->r_statelock);
451         }
453         /*
454          * Decrement the reference count for the FID
455          * and possibly do the OtW close.
456          *
457          * Exclusive lock for modifying n_fid stuff.
458          * Don't want this one ever interruptible.
459          */
460         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
461         smb_credinit(&scred, cr);
463         smbfs_rele_fid(np, &scred);
465         smb_credrele(&scred);
466         smbfs_rw_exit(&np->r_lkserlock);
468         return (0);
469 }
```

*unchanged_portion_omitted_*

```
550 /* ARGSUSED */
551 static int
552 smbfs_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
553         caller_context_t *ct)
554 {
555         struct smb_cred scred;
556         struct vattr    va;
557         smbnode_t       *np;
558         smbmntinfo_t    *smi;
```



```
559         smb_share_t     *ssp;
560         offset_t        endoff;
561         ssize_t         past_eof;
562         int             error;
564         caddr_t         base;
565         u_offset_t      off;
566         size_t          n;
567         int             on;
568         uint_t          flags;
570         np = VTOSMB(vp);
571         smi = VTOSMI(vp);
572         ssp = smi->smi_share;
574         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
575                 return (EIO);
577         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
578                 return (EIO);
580         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));
582         if (vp->v_type != VREG)
583                 return (EISDIR);
585         if (uiop->uio_resid == 0)
586                 return (0);
588         /*
589          * Like NFS3, just check for 63-bit overflow.
590          * Our SMB layer takes care to return EFBIG
591          * when it has to fallback to a 32-bit call.
592          */
593         endoff = uiop->uio_loffset + uiop->uio_resid;
594         if (uiop->uio_loffset < 0 || endoff < 0)
595                 return (EINVAL);
597         /* get vnode attributes from server */
598         va.va_mask = AT_SIZE | AT_MTIME;
599         if (error = smbfsgetattr(vp, &va, cr))
600                 return (error);
602         /* Update mtime with mtime from server here? */
604         /* if offset is beyond EOF, read nothing */
605         if (uiop->uio_loffset >= va.va_size)
606                 return (0);
608         /*
609          * Limit the read to the remaining file size.
610          * Do this by temporarily reducing uio_resid
611          * by the amount the lies beyoned the EOF.
612          */
613         if (endoff > va.va_size) {
614                 past_eof = (ssize_t)(endoff - va.va_size);
615                 uiop->uio_resid -= past_eof;
616         } else
617                 past_eof = 0;
619         /*
620          * Bypass VM if caching has been disabled (e.g., locking) or if
621          * using client-side direct I/O and the file is not mmap'd and
622          * there are no cached pages.
623          */
624         if ((vp->v_flag & VNOCACHE) ||
```

```
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))) {

629                 /* Shared lock for n_fid use in smb_rwuio */
630                 if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp))
631                         return (EINTR);
632                 smb_credinit(&scred, cr);

634                 /* After reconnect, n_fid is invalid */
635                 if (np->n_vcgenid != ssp->ss_vcgenid)
636                         error = ESTALE;
637                 else
638                         error = smb_rwuio(ssp, np->n_fid, UIO_READ,
639                             uiop, &scred, smb_timo_read);

641                 smb_credrele(&scred);
642                 smbfs_rw_exit(&np->r_lkserlock);

644                 /* undo adjustment of resid */
645                 uiop->uio_resid += past_eof;

647                 return (error);
648         }

650         /* (else) Do I/O through segmap. */
651         do {
652                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
653                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
654                 n = MIN(MAXBSIZE - on, uiop->uio_resid);

656                 error = smbfs_validate_caches(vp, cr);
657                 if (error)
658                         break;

660                 /* NFS waits for RINCACHEPURGE here. */

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_getmapflt(segkmap, vp, off + on, n, 1,
670                             S_READ);

672                         error = uiomove(base + on, n, UIO_READ, uiop);
673                 }

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                 } 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         } while (!error && uiop->uio_resid > 0);

701         /* undo adjustment of resid */
702         uiop->uio_resid += past_eof;

704         return (error);
705 }


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         smbmntinfo_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;

730         np = VTOSMB(vp);
731         smi = VTOSMI(vp);
732         ssp = smi->smi_share;

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

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

740         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_WRITER));

742         if (vp->v_type != VREG)
743                 return (EISDIR);

745         if (uiop->uio_resid == 0)
746                 return (0);

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);
702                         /* XXX: smbfs_vinvalbuf? */
754                 }
755         }
```

```
        if (ioflag & FAPPEND) {
                /*
                 * File size can be changed by another client
                 *
                 * Todo: Consider redesigning this to use a
                 * handle opened for append instead.
                 */
                va.va_mask = AT_SIZE;
                if (error = smbfsgetattr(vp, &va, cr))
                        return (error);
                uiop->uio_loffset = va.va_size;
        }

        /*
         * Like NFS3, just check for 63-bit overflow.
         */
        endoff = uiop->uio_loffset + uiop->uio_resid;
        if (uiop->uio_loffset < 0 || endoff < 0)
                return (EINVAL);

        /*
         * Check to make sure that the process will not exceed
         * its limit on file size.  It is okay to write up to
         * the limit, but not beyond.  Thus, the write which
         * reaches the limit will be short and the next write
         * will return an error.
         *
         * So if we're starting at or beyond the limit, EFBIG.
         * Otherwise, temporarily reduce resid to the amount
         * that is after the limit.
         * the falls after the limit.
         */
        limit = uiop->uio_llimit;
        if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
                limit = MAXOFFSET_T;
        if (uiop->uio_loffset >= limit) {
                proc_t *p = ttoproc(curthread);

                mutex_enter(&p->p_lock);
                (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE],
                    p->p_rctls, p, RCA_UNSAFE_SIGINFO);
                mutex_exit(&p->p_lock);
        if (uiop->uio_loffset >= limit)
                return (EFBIG);
        }
        if (endoff > limit) {
                past_limit = (ssize_t)(endoff - limit);
                uiop->uio_resid -= past_limit;
        } else
                past_limit = 0;

        /*
         * Bypass VM if caching has been disabled (e.g., locking) or if
         * using client-side direct I/O and the file is not mmap'd and
         * there are no cached pages.
         */
        if ((vp->v_flag & VNOCACHE) ||
            (((np->r_flags & RDIRECTIO) || (smi->smi_flags & SMI_DIRECTIO)) &&
            np->r_mapcnt == 0 && np->r_inmap == 0 &&
            !vn_has_cached_data(vp))) {

smbfs_fwrite:
                if (np->r_flags & RSTALE) {
                        last_resid = uiop->uio_resid;
                        last_off = uiop->uio_loffset;
                        error = np->r_error;
```



```
                        /*
                         * A close may have cleared r_error, if so,
                         * propagate ESTALE error return properly
                         */
                        if (error == 0)
                                error = ESTALE;
                        goto bottom;
                }

                /* Timeout: longer for append. */
                timo = smb_timo_write;
                if (endoff > np->r_size)
                        timo = smb_timo_append;

                /* Shared lock for n_fid use in smb_rwuio */
                if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp))
                        return (EINTR);
                smb_credinit(&scred, cr);

                /* After reconnect, n_fid is invalid */
                if (np->n_vcgenid != ssp->ss_vcgenid)
                        error = ESTALE;
                else
                        error = smb_rwuio(ssp, np->n_fid, UIO_WRITE,
                            uiop, &scred, timo);

                if (error == 0) {
                        mutex_enter(&np->r_statelock);
                        np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
                        if (uiop->uio_loffset > (offset_t)np->r_size)
                                np->r_size = (len_t)uiop->uio_loffset;
                        mutex_exit(&np->r_statelock);
                        if (ioflag & (FSYNC | FDSYNC)) {
                if (ioflag & (FSYNC|FDSYNC)) {
                                /* Don't error the I/O if this fails. */
                                (void) smbfs_smb_flush(np, &scred);
                        }
                }

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

                /* undo adjustment of resid */
                uiop->uio_resid += past_limit;

                return (error);
        }

        /* (else) Do I/O through segmap. */
        bsize = vp->v_vfsp->vfs_bsize;

        do {
                off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
                on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
                n = MIN(MAXBSIZE - on, uiop->uio_resid);

                last_resid = uiop->uio_resid;
                last_off = uiop->uio_loffset;

                if (np->r_flags & RSTALE) {
                        error = np->r_error;
                        /*
                         * A close may have cleared r_error, if so,
                         * propagate ESTALE error return properly
                         */
                        if (error == 0)
```

```
885                                     error = ESTALE;
886                             break;
887                     }

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);

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                     mutex_exit(&np->r_statelock);

916                     /*
917                      * Touch the page and fault it in if it is not in core
918                      * before segmap_getmapflt or vpm_data_copy can lock it.
919                      * This is to avoid the deadlock if the buffer is mapped
920                      * to the same file through mmap which we want to write.
921                      */
922                     uio_prefaultpages((long)n, uiop);

924                     if (vpm_enable) {
925                             /*
926                              * It will use kpm mappings, so no need to
927                              * pass an address.
928                              */
929                             error = smbfs_writenp(np, NULL, n, uiop, 0);
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;

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);

942                                     base = segmap_getmapflt(segkmap, vp, off + on,
943                                         pn, !pagecreate, S_WRITE);

945                                     error = smbfs_writenp(np, base + pon, n, uiop,
946                                         pagecreate);

948                             } else {
949                                     base = segmap_getmapflt(segkmap, vp, off + on,
950                                         n, 0, S_READ);
```



```
951                                     error = smbfs_writenp(np, base + on, n, uiop, 0)
952                             }
953                     }

955                     if (!error) {
956                             if (smi->smi_flags & SMI_NOAC)
957                                     flags = SM_WRITE;
958                             else if ((uiop->uio_loffset % bsize) == 0 ||
959                                 IS_SWAPVP(vp)) {
960                                     /*
961                                      * Have written a whole block.
962                                      * Start an asynchronous write
963                                      * and mark the buffer to
964                                      * indicate that it won't be
965                                      * needed again soon.
966                                      */
967                                     flags = SM_WRITE | SM_ASYNC | SM_DONTNEED;
968                             } else
969                                     flags = 0;
970                             if ((ioflag & (FSYNC|FDSYNC)) ||
971                                 (np->r_flags & ROUTOFSPACE)) {
972                                     flags &= ~SM_ASYNC;
973                                     flags |= SM_WRITE;
974                             }
975                             if (vpm_enable) {
976                                     error = vpm_sync_pages(vp, off, n, flags);
977                             } else {
978                                     error = segmap_release(segkmap, base, flags);
979                             }
980                     } else {
981                             if (vpm_enable) {
982                                     (void) vpm_sync_pages(vp, off, n, 0);
983                             } else {
984                                     (void) segmap_release(segkmap, base, 0);
985                             }
986                             /*
987                              * In the event that we got an access error while
988                              * faulting in a page for a write-only file just
989                              * force a write.
990                              */
991                             if (error == EACCES)
992                                     goto smbfs_fwrite;
993                     }
994             } while (!error && uiop->uio_resid > 0);

996     bottom:
997             /* undo adjustment of resid */
998             if (error) {
999                     uiop->uio_resid = last_resid + past_limit;
1000                    uiop->uio_loffset = last_off;
1001            } else {
1002                    uiop->uio_resid += past_limit;
1003            }

1005            return (error);
1006    }

1008    /*
1009     * Like nfs_client.c: writerp()
1010     *
1011     * Write by creating pages and uiomove data onto them.
1012     */

1014    int
1015    smbfs_writenp(smbnode_t *np, caddr_t base, int tcount, struct uio *uio,
1016        int pgcreated)
```

```
{
        int             pagecreate;
        int             n;
        int             saved_n;
        caddr_t         saved_base;
        u_offset_t      offset;
        int             error;
        int             sm_error;
        vnode_t         *vp = SMBTOV(np);

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

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

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

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

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



```
                         * interface change.
                         */
                        if (pgcreated == 0)
                                (void) segmap_pagecreate(segkmap, base,
                                    (uint_t)n, 1);
                        saved_base = base;
                        saved_n = n;
                }

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

                if (vpm_enable) {
                        /*
                         * Copy data. If new pages are created, part of
                         * the page that is not written will be initizliazed
                         * with zeros.
                         */
                        error = vpm_data_copy(vp, offset, n, uio,
                            !pagecreate, NULL, 0, S_WRITE);
                } else {
                        error = uiomove(base, n, UIO_WRITE, uio);
                }

                /*
                 * r_size is the maximum number of
                 * bytes known to be in the file.
                 * Make sure it is at least as high as the
                 * first unwritten byte pointed to by uio_loffset.
                 */
                mutex_enter(&np->r_statelock);
                if (np->r_size < uio->uio_loffset)
                        np->r_size = uio->uio_loffset;
                np->r_flags &= ~RMODINPROGRESS;
                np->r_flags |= RDIRTY;
                mutex_exit(&np->r_statelock);

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

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

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

1172        return (error);
1173 }

1175 /*
1176  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED}
1177  * Like nfs3_rdwrlbn()
1178  */
1179 static int
1180 smbfs_rdwrlbn(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
1181        int flags, cred_t *cr)
1182 {
1183        smbmntinfo_t    *smi = VTOSMI(vp);
1184        struct buf *bp;
1185        int error;
1186        int sync;

1188        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1189                return (EIO);

1191        if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1192                return (EIO);

1194        bp = pageio_setup(pp, len, vp, flags);
1195        ASSERT(bp != NULL);

1197        /*
1198         * pageio_setup should have set b_addr to 0.  This
1199         * is correct since we want to do I/O on a page
1200         * boundary.  bp_mapin will use this addr to calculate
1201         * an offset, and then set b_addr to the kernel virtual
1202         * address it allocated for us.
1203         */
1204        ASSERT(bp->b_un.b_addr == 0);

1206        bp->b_edev = 0;
1207        bp->b_dev = 0;
1208        bp->b_lblkno = lbtodb(off);
1209        bp->b_file = vp;
1210        bp->b_offset = (offset_t)off;
1211        bp_mapin(bp);

1213        /*
1214         * Calculate the desired level of stability to write data.
```



```
1215         */
1216        if ((flags & (B_WRITE|B_ASYNC)) == (B_WRITE|B_ASYNC) &&
1217            freemem > desfree) {
1218                sync = 0;
1219        } else {
1220                sync = 1;
1221        }

1223        error = smbfs_bio(bp, sync, cr);

1225        bp_mapout(bp);
1226        pageio_done(bp);

1228        return (error);
1229 }


1232 /*
1233  * Corresponds to nfs3_vnopc.c : nfs3_bio(), though the NFS code
1234  * uses nfs3read()/nfs3write() where we use smb_rwuio().  Also,
1235  * NFS has this later in the file.  Move it up here closer to
1236  * the one call site just above.
1237  */

1239 static int
1240 smbfs_bio(struct buf *bp, int sync, cred_t *cr)
1241 {
1242        struct iovec aiov[1];
1243        struct uio  auio;
1244        struct smb_cred scred;
1245        smbnode_t *np = VTOSMB(bp->b_vp);
1246        smbmntinfo_t *smi = np->n_mount;
1247        smb_share_t *ssp = smi->smi_share;
1248        offset_t offset;
1249        offset_t endoff;
1250        size_t count;
1251        size_t past_eof;
1252        int error;

1254        ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);

1256        offset = ldbtob(bp->b_lblkno);
1257        count = bp->b_bcount;
1258        endoff = offset + count;
1259        if (offset < 0 || endoff < 0)
1260                return (EINVAL);

1262        /*
1263         * Limit file I/O to the remaining file size, but see
1264         * the notes in smbfs_getpage about SMBFS_EOF.
1265         */
1266        mutex_enter(&np->r_statelock);
1267        if (offset >= np->r_size) {
1268                mutex_exit(&np->r_statelock);
1269                if (bp->b_flags & B_READ) {
1270                        return (SMBFS_EOF);
1271                } else {
1272                        return (EINVAL);
1273                }
1274        }
1275        if (endoff > np->r_size) {
1276                past_eof = (size_t)(endoff - np->r_size);
1277                count -= past_eof;
1278        } else
1279                past_eof = 0;
1280        mutex_exit(&np->r_statelock);
```

```
1281         ASSERT(count > 0);

1283         /* Caller did bpmapin().  Mapped address is... */
1284         aiov[0].iov_base = bp->b_un.b_addr;
1285         aiov[0].iov_len = count;
1286         auio.uio_iov = aiov;
1287         auio.uio_iovcnt = 1;
1288         auio.uio_loffset = offset;
1289         auio.uio_segflg = UIO_SYSSPACE;
1290         auio.uio_fmode = 0;
1291         auio.uio_resid = count;

1293         /* Shared lock for n_fid use in smb_rwuio */
1294         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER,
1295             smi->smi_flags & SMI_INT))
1296                 return (EINTR);
1297         smb_credinit(&scred, cr);

1299         DTRACE_IO1(start, struct buf *, bp);

1301         if (bp->b_flags & B_READ) {

1303                 /* After reconnect, n_fid is invalid */
1304                 if (np->n_vcgenid != ssp->ss_vcgenid)
1305                         error = ESTALE;
1306                 else
1307                         error = smb_rwuio(ssp, np->n_fid, UIO_READ,
1308                             &auio, &scred, smb_timo_read);

1310                 /* Like NFS, only set b_error here. */
1311                 bp->b_error = error;
1312                 bp->b_resid = auio.uio_resid;

1314                 if (!error && auio.uio_resid != 0)
1315                         error = EIO;
1316                 if (!error && past_eof != 0) {
1317                         /* Zero the memory beyond EOF. */
1318                         bzero(bp->b_un.b_addr + count, past_eof);
1319                 }
1320         } else {

1322                 /* After reconnect, n_fid is invalid */
1323                 if (np->n_vcgenid != ssp->ss_vcgenid)
1324                         error = ESTALE;
1325                 else
1326                         error = smb_rwuio(ssp, np->n_fid, UIO_WRITE,
1327                             &auio, &scred, smb_timo_write);

1329                 /* Like NFS, only set b_error here. */
1330                 bp->b_error = error;
1331                 bp->b_resid = auio.uio_resid;

1333                 if (!error && auio.uio_resid != 0)
1334                         error = EIO;
1335                 if (!error && sync) {
1336                         (void) smbfs_smb_flush(np, &scred);
1337                 }
1338         }

1340         /*
1341          * This comes from nfs3_commit()
1342          */
1343         if (error != 0) {
1344                 mutex_enter(&np->r_statelock);
1345                 if (error == ESTALE)
1346                         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         }

1353         DTRACE_IO1(done, struct buf *, bp);

1355         smb_credrele(&scred);
1356         smbfs_rw_exit(&np->r_lkserlock);

1358         if (error == ESTALE)
1359                 smbfs_attrcache_remove(np);

1361         return (error);
1362 }

1364 /*
1365  * Here NFS has: nfs3write, nfs3read
1366  * We use smb_rwuio instead.
1367  */

1369 /* ARGSUSED */
1370 static int
1371 smbfs_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag,
1372         cred_t *cr, int *rvalp, caller_context_t *ct)
1373 {
1374         int             error;
1375         smbmntinfo_t    *smi;

1377         smi = VTOSMI(vp);

1379         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1380                 return (EIO);

1382         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1383                 return (EIO);

1385         switch (cmd) {
 800                 /* First three from ZFS. XXX - need these? */

1387         case _FIOFFS:
1388                 error = smbfs_fsync(vp, 0, cr, ct);
1389                 break;

1391                 /*
1392                  * The following two ioctls are used by bfu.
1393                  * Silently ignore to avoid bfu errors.
1394                  */
1395         case _FIOGDIO:
1396         case _FIOSDIO:
1397                 error = 0;
1398                 break;

1400 #if 0   /* Todo - SMB ioctl query regions */
1401         case _FIO_SEEK_DATA:
1402         case _FIO_SEEK_HOLE:
1403 #endif

 815 #ifdef NOT_YET  /* XXX - from the NFS code. */
1405         case _FIODIRECTIO:
1406                 error = smbfs_directio(vp, (int)arg, cr);
1407                 break;
 818 #endif

1409                 /*
```

```
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;

1417        case SMBFSIO_SETSD:
1418                error = smbfs_acl_iocset(vp, arg, flag, cr);
1419                break;

1421        default:
1422                error = ENOTTY;
1423                break;
1424        }

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        smbmntinfo_t *smi;
1441        int error;

1443        smi = VTOSMI(vp);

1445        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1446                return (EIO);

1448        if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1449                return (EIO);

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 rdevid, 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         * Here NFS also checks for async writes (np->r_awcount)
1481         */
1482        if (vap->va_mask & AT_MTIME) {
1483                if (vn_has_cached_data(vp) &&
1484                    ((np->r_flags & RDIRTY) != 0)) {
1485                        mutex_enter(&np->r_statelock);
1486                        np->r_gcount++;
1487                        mutex_exit(&np->r_statelock);
1488                        error = smbfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
1489                        mutex_enter(&np->r_statelock);
1490                        if (error && (error == ENOSPC || error == EDQUOT)) {
1491                                if (!np->r_error)
1492                                        np->r_error = error;
1493                        }
1494                        if (--np->r_gcount == 0)
1495                                cv_broadcast(&np->r_cv);
1496                        mutex_exit(&np->r_statelock);
1497                }
1498        }

1500        return (smbfsgetattr(vp, vap, cr));
1501 }

1503 /* smbfsgetattr() in smbfs_client.c */

1505 /*ARGSUSED4*/
1506 static int
1507 smbfs_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1508                caller_context_t *ct)
1509 {
1510        vfs_t           *vfsp;
1511        smbmntinfo_t    *smi;
1512        int             error;
1513        uint_t          mask;
1514        struct vattr    oldva;

1516        vfsp = vp->v_vfsp;
1517        smi = VFTOSMI(vfsp);

1519        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1520                return (EIO);

1522        if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
1523                return (EIO);

1525        mask = vap->va_mask;
1526        if (mask & AT_NOSET)
1527                return (EINVAL);

1529        if (vfsp->vfs_flag & VFS_RDONLY)
1530                return (EROFS);

1532        /*
1533         * This is a _local_ access check so that only the owner of
1534         * this mount can set attributes.  With ACLs enabled, the
1535         * file owner can be different from the mount owner, and we
1536         * need to check the _mount_ owner here.  See _access_rwx
1537         */
1538        bzero(&oldva, sizeof (oldva));
1539        oldva.va_mask = AT_TYPE | AT_MODE;
1540        error = smbfsgetattr(vp, &oldva, cr);
1541        if (error)
```

```
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;

1547         error = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
1548             smbfs_accessx, vp);
1549         if (error)
1550                 return (error);

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 seting 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         }

1570         error = smbfssetattr(vp, vap, flags, cr);

1572 #ifdef  SMBFS_VNEVENT
1573         if (error == 0 && (vap->va_mask & AT_SIZE) && vap->va_size == 0)
1574                 vnevent_truncate(vp, ct);
1575 #endif

1577         return (error);
 956         return (smbfssetattr(vp, vap, flags, cr));
1578 }

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             cerror, modified = 0;
1594         unsigned short  fid;
1595         int have_fid = 0;
1596         uint32_t rights = 0;
1597         uint32_t dosattr = 0;

1599         ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);

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         }

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         }

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          */

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);

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);

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         }

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          */

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.
1055                  * I am commenting this code at present because the function
1056                  * smbfs_putapage() is not yet implemented.
1701                  */

1703                 /*
1704                  * Set the file size to vap->va_size.
1705                  */
1706                 ASSERT(have_fid);
1707                 error = smbfs_smb_setfsize(np, fid, vap->va_size, &scred);
1708                 if (error) {
1709                         SMBVDEBUG("setsize error %d file %s\n",
1710                             error, np->n_rpath);
1711                 } else {
1712                         /*
1713                          * Darwin had code here to zero-extend.
1714                          * Tests indicate the server will zero-fill,
1715                          * so looks like we don't need to do that.
1071                          * so looks like we don't need to do this.
1072                          * Good thing, as this could take forever.
1073                          *
1074                          * XXX: Reportedly, writing one byte of zero
1075                          * at the end offset avoids problems here.
1716                          */
1717                         mutex_enter(&np->r_statelock);
1718                         np->r_size = vap->va_size;
1719                         mutex_exit(&np->r_statelock);
1720                         modified = 1;
1721                 }
1722         }

1724         /*
1725          * Todo: Implement setting create_time (which is
1726          * different from ctime).
1085          * XXX: When Solaris has create_time, set that too.
1086          * Note: create_time is different from ctime.
1727          */
1728         mtime = ((mask & AT_MTIME) ? &vap->va_mtime : 0);
1729         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         }

1746 out:
1107         if (modified) {
1108                 /*
1109                  * Invalidate attribute cache in case the server
1110                  * doesn't set exactly the attributes we asked.
1111                  */
1112                 smbfs_attrcache_remove(np);
1113         }

1747         if (have_fid) {
1748                 cerror = smbfs_smb_tmpclose(np, fid, &scred);
1749                 if (cerror)
1750                         SMBVDEBUG("error %d closing %s\n",
1751                             cerror, np->n_rpath);
1752         }

1754         smb_credrele(&scred);
1755         smbfs_rw_exit(&np->r_lkserlock);

1757         if (modified) {
1758                 /*
1759                  * Invalidate attribute cache in case the server
1760                  * doesn't set exactly the attributes we asked.
1761                  */
1762                 smbfs_attrcache_remove(np);

1764                 /*
1765                  * If changing the size of the file, invalidate
1766                  * any local cached data which is no longer part
1767                  * of the file.  We also possibly invalidate the
1768                  * last page in the file.  We could use
1769                  * pvn_vpzero(), but this would mark the page as
1770                  * modified and require it to be written back to
1771                  * the server for no particularly good reason.
1772                  * This way, if we access it, then we bring it
1773                  * back in.  A read should be cheaper than a
1774                  * write.
1775                  */
1776                 if (mask & AT_SIZE) {
1777                         smbfs_invalidate_pages(vp,
1778                             (vap->va_size & PAGEMASK), cr);
1779                 }
1780         }

1782         return (error);
1783 }
_____unchanged_portion_omitted_

1843 /*
1844  * smbfs_access_rwx()
```

```
 * Common function for smbfs_access, etc.
 *
 * The security model implemented by the FS is unusual
 * due to the current "single user mounts" restriction:
 * All access under a given mount point uses the CIFS
 * credentials established by the owner of the mount.
 *
 * Most access checking is handled by the CIFS server,
 * but we need sufficient Unix access checks here to
 * prevent other local Unix users from having access
 * to objects under this mount that the uid/gid/mode
 * settings in the mount would not allow.
 *
 * With this model, there is a case where we need the
 * ability to do an access check before we have the
 * vnode for an object.  This function takes advantage
 * of the fact that the uid/gid/mode is per mount, and
 * avoids the need for a vnode.
 *
 * We still (sort of) need a vnode when we call
 * secpolicy_vnode_access, but that only uses
 * the vtype field, so we can use a pair of fake
 * vnodes that have only v_type filled in.
 *
 * XXX: Later, add a new secpolicy_vtype_access()
 * that takes the vtype instead of a vnode, and
 * get rid of the tmpl_vxxx fake vnodes below.
 */
static int
smbfs_access_rwx(vfs_t *vfsp, int vtype, int mode, cred_t *cr)
{
        /* See the secpolicy call below. */
        static const vnode_t tmpl_vdir = { .v_type = VDIR };
        static const vnode_t tmpl_vreg = { .v_type = VREG };
        vattr_t         va;
        vnode_t         *tvp;
        struct smbmntinfo *smi = VFTOSMI(vfsp);
        int shift = 0;

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

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

        /*
         * Disallow attempts to access mandatory lock files.
```



```
         * Similarly, expand MANDLOCK here.
         * XXX: not sure we need this.
         */
        if ((mode & (VWRITE | VREAD | VEXEC)) &&
            va.va_type == VREG && MANDMODE(va.va_mode))
                return (EACCES);

        /*
         * Access check is based on only
         * one of owner, group, public.
         * If not owner, then check group.
         * If not a member of the group,
         * then check public access.
         */
        if (crgetuid(cr) != va.va_uid) {
                shift += 3;
                if (!groupmember(va.va_gid, cr))
                        shift += 3;
        }

        /*
         * We need a vnode for secpolicy_vnode_access,
         * but the only thing it looks at is v_type,
         * so pass one of the templates above.
         */
        tvp = (va.va_type == VDIR) ?
            (vnode_t *)&tmpl_vdir :
            (vnode_t *)&tmpl_vreg;

        return (secpolicy_vnode_access2(cr, tvp, va.va_uid,
            va.va_mode << shift, mode));
}
```

*unchanged_portion_omitted_*

```
/* ARGSUSED */
static int
smbfs_readlink(vnode_t *vp, struct uio *uiop, cred_t *cr, caller_context_t *ct)
{
        /* Not yet... */
        return (ENOSYS);
}


/*
 * Flush local dirty pages to stable storage on the server.
 *
 * If FNODSYNC is specified, then there is nothing to do because
 * metadata changes are not cached on the client before being
 * sent to the server.
 */
/* ARGSUSED */
static int
smbfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
{
        int             error = 0;
        smbmntinfo_t    *smi;
        smbnode_t       *np;
        struct smb_cred scred;

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

        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
                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 {
1376         smbnode_t       *np;
2039         struct smb_cred scred;
2040         smbnode_t       *np = VTOSMB(vp);
2041         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          */

1385         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_RELE() 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          */

2057         /*
2058          * From NFS:rinactive()
2059          *
2060          * Before freeing anything, wait until all asynchronous
2061          * activity is done on this rnode.  This will allow all
2062          * asynchronous read ahead and write behind i/o's to
2063          * finish.
2064          */
2065         mutex_enter(&np->r_statelock);
2066         while (np->r_count > 0)
2067                 cv_wait(&np->r_cv, &np->r_statelock);
2068         mutex_exit(&np->r_statelock);
```



```
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);

2090         /*
2091          * Defend against the possibility that higher-level callers
2092          * might not correctly balance open and close calls.  If we
2093          * get here with open references remaining, it means there
2094          * was a missing VOP_CLOSE somewhere.  If that happens, do
2095          * the close here so we don't "leak" FIDs on the server.
2096          *
2097          * Exclusive lock for modifying n_fid stuff.
2098          * Don't want this one ever interruptible.
2099          */
2100         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
2101         smb_credinit(&scred, cr);

2103         switch (np->n_ovtype) {
2104         case VNON:
2105                 /* not open (OK) */
2106                 break;

2108         case VDIR:
2109                 if (np->n_dirrefs == 0)
2110                         break;
2111                 SMBVDEBUG("open dir: refs %d path %s\n",
2112                     np->n_dirrefs, np->n_rpath);
2113                 /* Force last close. */
2114                 np->n_dirrefs = 1;
2115                 smbfs_rele_fid(np, &scred);
2116                 break;

2118         case VREG:
2119                 if (np->n_fidrefs == 0)
2120                         break;
2121                 SMBVDEBUG("open file: refs %d id 0x%x path %s\n",
2122                     np->n_fidrefs, np->n_fid, np->n_rpath);
2123                 /* Force last close. */
2124                 np->n_fidrefs = 1;
2125                 smbfs_rele_fid(np, &scred);
2126                 break;

2128         default:
2129                 SMBVDEBUG("bad n_ovtype %d\n", np->n_ovtype);
2130                 np->n_ovtype = VNON;
2131                 break;
2132         }

2134         smb_credrele(&scred);
```

```
2135         smbfs_rw_exit(&np->r_lkserlock);

2137         /*
2138          * XATTR directories (and the files under them) have
2139          * little value for reclaim, so just remove them from
2140          * the "hash" (AVL) as soon as they go inactive.
2141          * Note that the node may already have been removed
2142          * from the hash by smbfsremove.
2143          */
2144         if ((np->n_flag & N_XATTR) != 0 &&
2145             (np->r_flags & RHASHED) != 0)
2146                 smbfs_rmhash(np);

2148         smbfs_addfree(np);
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         smbmntinfo_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;

2222         smi = VTOSMI(dvp);
2223         dnp = VTOSMB(dvp);

2225         ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);

2227 #ifdef NOT_YET
2228         vcp = SSTOVC(smi->smi_share);

2230         /* XXX: Should compute this once and store it in smbmntinfo_t */
2231         supplen = (SMB_DIALECT(vcp) >= SMB_DIALECT_LANMAN2_0) ? 255 : 12;
2232 #else
2233         supplen = 255;
2234 #endif

2236         /*
2237          * RWlock must be held, either reader or writer.
1543          * XXX: Can we check without looking directly
1544          * inside the struct smbfs_rwlock_t?
2238          */
2239         ASSERT(dnp->r_rwlock.count != 0);

2241         /*
2242          * If lookup is for "", just return dvp.
2243          * No need to perform any access checks.
2244          */
2245         if (nmlen == 0) {
2246                 VN_HOLD(dvp);
2247                 *vpp = dvp;
2248                 return (0);
2249         }
```



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

2257         /*
2258          * Need search permission in the directory.
2259          */
2260         error = smbfs_access(dvp, VEXEC, 0, cr, ct);
2261         if (error)
2262                 return (error);

2264         /*
2265          * If lookup is for ".", just return dvp.
2266          * Access check was done above.
2267          */
2268         if (nmlen == 1 && name[0] == '.') {
2269                 VN_HOLD(dvp);
2270                 *vpp = dvp;
2271                 return (0);
2272         }

2274         /*
2275          * Now some sanity checks on the name.
2276          * First check the length.
2277          */
2278         if (nmlen > supplen)
2279                 return (ENAMETOOLONG);

2281         /*
2282          * Avoid surprises with characters that are
2283          * illegal in Windows file names.
2284          * Todo: CATIA mappings?
1591          * Todo: CATIA mappings  XXX
2285          */
2286         ill = illegal_chars;
2287         if (dnp->n_flag & N_XATTR)
2288                 ill++; /* allow colon */
2289         if (strpbrk(nm, ill))
2290                 return (EINVAL);

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

```
2315                         */
2316                        VN_HOLD(dvp);
2317                        *vpp = dvp;
2318                        return (0);
2319                }

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                }

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;

2352                /* Success! */
2353                *vpp = vp;
2354                return (0);
2355        }

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        }

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;

2397        error = smbfs_nget(dvp, name, nmlen, &fa, &vp);
2398        if (error)
2399                goto out;

2401        /* Success! */
2402        *vpp = vp;

2404 out:
2405        /* smbfs_smb_lookup may have allocated name. */
2406        if (name != nm)
2407                smbfs_name_free(name, nmlen);

2409        return (error);
2410 }
```

_____***unchanged_portion_omitted_***

```
2506 /*
2507  * XXX
2508  * vsecattr_t is new to build 77, and we need to eventually support
2509  * it in order to create an ACL when an object is created.
2510  *
2511  * This op should support the new FIGNORECASE flag for case-insensitive
2512  * lookups, per PSARC 2007/244.
2513  */
2514 /* ARGSUSED */
2515 static int
2516 smbfs_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
2517         int mode, vnode_t **vpp, cred_t *cr, int lfaware, caller_context_t *ct,
2518         vsecattr_t *vsecp)
2519 {
2520         int             error;
2521         int             cerror;
2522         vfs_t           *vfsp;
2523         vnode_t         *vp;
1830 #ifdef NOT_YET
2524         smbnode_t       *np;
1832 #endif
2525         smbnode_t       *dnp;
2526         smbmntinfo_t    *smi;
2527         struct vattr    vattr;
2528         struct smbfattr fattr;
2529         struct smb_cred scred;
2530         const char *name = (const char *)nm;
2531         int             nmlen = strlen(nm);
2532         uint32_t        disp;
2533         uint16_t        fid;
2534         int             xattr;

2536         vfsp = dvp->v_vfsp;
```

```
2537         smi = VFTOSMI(vfsp);
2538         dnp = VTOSMB(dvp);
2539         vp = NULL;

2541         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2542                 return (EPERM);

2544         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2545                 return (EIO);

2547         /*
2548          * Note: this may break mknod(2) calls to create a directory,
2549          * but that's obscure use.  Some other filesystems do this.
2550          * Todo: redirect VDIR type here to _mkdir.
1858         * XXX: Later, redirect VDIR type here to _mkdir.
2551          */
2552         if (va->va_type != VREG)
2553                 return (EINVAL);

2555         /*
2556          * If the pathname is "", just use dvp, no checks.
2557          * Do this outside of the rwlock (like zfs).
2558          */
2559         if (nmlen == 0) {
2560                 VN_HOLD(dvp);
2561                 *vpp = dvp;
2562                 return (0);
2563         }

2565         /* Don't allow "." or ".." through here. */
2566         if ((nmlen == 1 && name[0] == '.') ||
2567             (nmlen == 2 && name[0] == '.' && name[1] == '.'))
2568                 return (EISDIR);

2570         /*
2571          * We make a copy of the attributes because the caller does not
2572          * expect us to change what va points to.
2573          */
2574         vattr = *va;

2576         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2577                 return (EINTR);
2578         smb_credinit(&scred, cr);

2580         /*
2581          * NFS needs to go over the wire, just to be sure whether the
2582          * file exists or not.  Using a cached result is dangerous in
2583          * this case when making a decision regarding existence.
2584          *
2585          * The SMB protocol does NOT really need to go OTW here
2586          * thanks to the expressive NTCREATE disposition values.
2587          * Unfortunately, to do Unix access checks correctly,
2588          * we need to know if the object already exists.
2589          * When the object does not exist, we need VWRITE on
2590          * the directory.  Note: smbfslookup() checks VEXEC.
2591          */
2592         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2593         if (error == 0) {
2594                 /*
2595                  * The file already exists.  Error?
2596                  * NB: have a hold from smbfslookup
2597                  */
2598                 if (exclusive == EXCL) {
2599                         error = EEXIST;
2600                         VN_RELE(vp);
2601                         goto out;
```



```
2602                 }
2603                 /*
2604                  * Verify requested access.
2605                  */
2606                 error = smbfs_access(vp, mode, 0, cr, ct);
2607                 if (error) {
2608                         VN_RELE(vp);
2609                         goto out;
2610                 }

2612                 /*
2613                  * Truncate (if requested).
2614                  */
2615                 if ((vattr.va_mask & AT_SIZE) && vp->v_type == VREG) {
2616                         np = VTOSMB(vp);
2617                         /*
2618                          * Check here for large file truncation by
2619                          * LF-unaware process, like ufs_create().
2620                          */
2621                         if (!(lfaware & FOFFMAX)) {
2622                                 mutex_enter(&np->r_statelock);
2623                                 if (np->r_size > MAXOFF32_T)
2624                                         error = EOVERFLOW;
2625                                 mutex_exit(&np->r_statelock);
2626                         }
2627                         if (error) {
2628                                 VN_RELE(vp);
2629                                 goto out;
2630                         }
1923                 if ((vattr.va_mask & AT_SIZE) && vattr.va_size == 0) {
2631                         vattr.va_mask = AT_SIZE;
2632                         error = smbfssetattr(vp, &vattr, 0, cr);
2633                         if (error) {
2634                                 VN_RELE(vp);
2635                                 goto out;
2636                         }
2637 #ifdef  SMBFS_VNEVENT
2638                         /* Existing file was truncated */
2639                         vnevent_create(vp, ct);
2640 #endif
2641                         /* invalidate pages done in smbfssetattr() */
2642                 }
2643                 /* Success! */
1932 #ifdef NOT_YET
1933                 vnevent_create(vp, ct);
1934 #endif
2644                 *vpp = vp;
2645                 goto out;
2646         }

2648         /*
2649          * The file did not exist.  Need VWRITE in the directory.
2650          */
2651         error = smbfs_access(dvp, VWRITE, 0, cr, ct);
2652         if (error)
2653                 goto out;

2655         /*
2656          * Now things get tricky.  We also need to check the
2657          * requested open mode against the file we may create.
2658          * See comments at smbfs_access_rwx
2659          */
2660         error = smbfs_access_rwx(vfsp, VREG, mode, cr);
2661         if (error)
2662                 goto out;
```

```
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         if (exclusive == EXCL)
2674                 disp = NTCREATEX_DISP_CREATE;
2675         else {
2676                 /* Truncate regular files if requested. */
2677                 if ((va->va_type == VREG) &&
2678                     (va->va_mask & AT_SIZE) &&
2679                     (va->va_size == 0))
2680                         disp = NTCREATEX_DISP_OVERWRITE_IF;
2681                 else
2682                         disp = NTCREATEX_DISP_OPEN_IF;
2683         }
2684         xattr = (dnp->n_flag & N_XATTR) ? 1 : 0;
2685         error = smbfs_smb_create(dnp,
2686             name, nmlen, xattr,
2687             disp, &scred, &fid);
2688         if (error)
2689                 goto out;

2691         /*
2983          * XXX: Missing some code here to deal with
2984          * the case where we opened an existing file,
2985          * it's size is larger than 32-bits, and we're
2986          * setting the size from a process that's not
2987          * aware of large file offsets.  i.e.
2988          * from the NFS3 code:
2989          */
2990 #if NOT_YET /* XXX */
2991         if ((vattr.va_mask & AT_SIZE) &&
2992             vp->v_type == VREG) {
2993                 np = VTOSMB(vp);
2994                 /*
2995                  * Check here for large file handled
2996                  * by LF-unaware process (as
2997                  * ufs_create() does)
2998                  */
2999                 if (!(lfaware & FOFFMAX)) {
3000                         mutex_enter(&np->r_statelock);
3001                         if (np->r_size > MAXOFF32_T)
3002                                 error = EOVERFLOW;
3003                         mutex_exit(&np->r_statelock);
3004                 }
3005                 if (!error) {
3006                         vattr.va_mask = AT_SIZE;
3007                         error = smbfssetattr(vp,
3008                             &vattr, 0, cr);
3009                 }
3010         }
3011 #endif /* XXX */
3012         /*
3692          * Should use the fid to get/set the size
3693          * while we have it opened here.  See above.
3694          */

2696         cerror = smbfs_smb_close(smi->smi_share, fid, NULL, &scred);
2697         if (cerror)
2698                 SMBVDEBUG("error %d closing %s\\%s\n",
2699                     cerror, dnp->n_rpath, name);
```



```
2701         /*
2702          * In the open case, the name may differ a little
2703          * from what we passed to create (case, etc.)
2704          * so call lookup to get the (opened) name.
2705          *
2706          * XXX: Could avoid this extra lookup if the
2707          * "createact" result from NT_CREATE says we
2708          * created the object.
2709          */
2710         error = smbfs_smb_lookup(dnp, &name, &nmlen, &fattr, &scred);
2711         if (error)
2712                 goto out;

2714         /* update attr and directory cache */
2715         smbfs_attr_touchdir(dnp);

2717         error = smbfs_nget(dvp, name, nmlen, &fattr, &vp);
2718         if (error)
2719                 goto out;

2042         /* XXX invalidate pages if we truncated? */

2721         /* Success! */
2722         *vpp = vp;
2723         error = 0;

2725 out:
2726         smb_credrele(&scred);
2727         smbfs_rw_exit(&dnp->r_rwlock);
2728         if (name != nm)
2729                 smbfs_name_free(name, nmlen);
2730         return (error);
2731 }
_____unchanged_portion_omitted_

2782 /*
2783  * smbfsremove does the real work of removing in SMBFS
2784  * Caller has done dir access checks etc.
2785  *
2786  * The normal way to delete a file over SMB is open it (with DELETE access),
2787  * set the "delete-on-close" flag, and close the file.  The problem for Unix
2788  * applications is that they expect the file name to be gone once the unlink
2789  * completes, and the SMB server does not actually delete the file until ALL
2790  * opens of that file are closed.  We can't assume our open handles are the
2791  * only open handles on a file we're deleting, so to be safe we'll try to
2792  * rename the file to a temporary name and then set delete-on-close.  If we
2793  * fail to set delete-on-close (i.e. because other opens prevent it) then
2794  * undo the changes we made and give up with EBUSY.  Note that we might have
2795  * permission to delete a file but lack permission to rename, so we want to
2796  * continue in cases where rename fails.  As an optimization, only do the
2797  * rename when we have the file open.
2798  *
2799  * This is similar to what NFS does when deleting a file that has local opens,
2800  * but thanks to SMB delete-on-close, we don't need to keep track of when the
2801  * last local open goes away and send a delete.  The server does that for us.
2802  */
2803 /* ARGSUSED */
2804 static int
2805 smbfsremove(vnode_t *dvp, vnode_t *vp, struct smb_cred *scred,
2806     int flags)
2807 {
2808         smbnode_t       *dnp = VTOSMB(dvp);
2809         smbnode_t       *np = VTOSMB(vp);
2810         char            *tmpname = NULL;
2811         int             tnlen;
```

```
2812        int             error;
2813        unsigned short  fid;
2814        boolean_t       have_fid = B_FALSE;
2815        boolean_t       renamed = B_FALSE;

2817        /*
2818         * The dvp RWlock must be held as writer.
2819         */
2820        ASSERT(dnp->r_rwlock.owner == curthread);

2822        /* Never allow link/unlink directories on SMB. */
2823        if (vp->v_type == VDIR)
2824                return (EPERM);

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        }

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);

2153        /* Force lookup to go OtW */
2154        smbfs_attrcache_remove(np);

2848        /*
2849         * Get a file handle with delete access.
2850         * Close this FID before return.
2851         */
2852        error = smbfs_smb_tmpopen(np, STD_RIGHT_DELETE_ACCESS,
2853            scred, &fid);
2854        if (error) {
2855                SMBVDEBUG("error %d opening %s\n",
2856                    error, np->n_rpath);
2857                goto out;
2858        }
2859        have_fid = B_TRUE;

2861        /*
2862         * If we have the file open, try to rename it to a temporary name.
2863         * If we can't rename, continue on and try setting DoC anyway.
2864         */
2865        if ((vp->v_count > 1) && (np->n_fidrefs > 0)) {
2866                tmpname = kmem_alloc(MAXNAMELEN, KM_SLEEP);
2867                tnlen = smbfs_newname(tmpname, MAXNAMELEN);
2868                error = smbfs_smb_t2rename(np, tmpname, tnlen, scred, fid, 0);
2869                if (error != 0) {
2870                        SMBVDEBUG("error %d renaming %s -> %s\n",
2871                            error, np->n_rpath, tmpname);
2872                        /* Keep going without the rename. */
2873                } else {
2874                        renamed = B_TRUE;
```



```
2875                }
2876        }

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;

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);

2910 #ifdef  SMBFS_VNEVENT
2911        vnevent_remove(vp, dvp, nm, ct);
2912 #endif

2914 out:
2915        if (tmpname != NULL)
2916                kmem_free(tmpname, MAXNAMELEN);

2918        if (have_fid)
2919                (void) smbfs_smb_tmpclose(np, fid, scred);
2920        smbfs_rw_exit(&np->r_lkserlock);

2922        if (error == 0) {
2923                /* Keep lookup from finding this node anymore. */
2924                smbfs_rmhash(np);
2925        }

2927        return (error);
2928 }


2931 /* ARGSUSED */
2932 static int
2933 smbfs_link(vnode_t *tdvp, vnode_t *svp, char *tnm, cred_t *cr,
2934        caller_context_t *ct, int flags)
2935 {
2936        /* Not yet... */
2937        return (ENOSYS);
2938 }
```

```
2941 /*
2942  * XXX
2943  * This op should support the new FIGNORECASE 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;

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);

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);

2967         if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
2968             strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0)
2969                 return (EINVAL);

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);

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         }

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);

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_RELE(ovp);
3028         }

3030         smb_credrele(&scred);
3031         smbfs_rw_exit(&odnp->r_rwlock);
3032         smbfs_rw_exit(&ndnp->r_rwlock);

3034         return (error);
3035 }

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;

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(odnp->r_rwlock.owner == curthread);
3057         ASSERT(ndnp->r_rwlock.owner == curthread);

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                        }
3077                } else {
3078                        if (nvp->v_type == VDIR) {
3079                                error = EISDIR;
3080                                goto out;
3081                        }
3082                }

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                }

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                }

3108                /*
3109                 * CIFS may give a SHARING_VIOLATION error when
3110                 * trying to rename onto an exising 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;

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_RELE(nvp);
3129                nvp = NULL;
3130        } /* nvp */

3132        smbfs_attrcache_remove(onp);

3133        error = smbfs_smb_rename(onp, ndnp, nnm, strlen(nnm), scred);

3135        /*
3136         * If the old name should no longer exist,
3137         * discard any cached attributes under it.
```



```
3138         */
3139        if (error == 0) {
2434        if (error == 0)
3140                smbfs_attrcache_prune(onp);
3141                /* SMBFS_VNEVENT... */
3142        }

3144 out:
3145        if (nvp) {
3146                if (nvp_locked)
3147                        vn_vfsunlock(nvp);
3148                VN_RELE(nvp);
3149        }

3151        return (error);
3152 }

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 /* ARGSUSED */
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 smbmntinfo *smi = VTOSMI(dvp);
3170        struct smb_cred scred;
3171        struct smbfattr fattr;
3172        const char              *name = (const char *) nm;
3173        int             nmlen = strlen(name);
3174        int             error, hiderr;

3176        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3177                return (EPERM);

3179        if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3180                return (EIO);

3182        if ((nmlen == 1 && name[0] == '.') ||
3183            (nmlen == 2 && name[0] == '.' && name[1] == '.'))
3184                return (EEXIST);

3186        /* Only plain files are allowed in V_XATTRDIR. */
3187        if (dvp->v_flag & V_XATTRDIR)
3188                return (EINVAL);

3190        if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
3191                return (EINTR);
3192        smb_credinit(&scred, cr);

3194        /*
2488         * XXX: Do we need r_lkserlock too?
2489         * No use of any shared fid or fctx...
2490         */

2492        /*
3195         * Require write access in the containing directory.
3196         */
3197        error = smbfs_access(dvp, VWRITE, 0, cr, ct);
```

```
3198         if (error)
3199                 goto out;

3201         error = smbfs_smb_mkdir(dnp, name, nmlen, &scred);
3202         if (error)
3203                 goto out;

3205         error = smbfs_smb_lookup(dnp, &name, &nmlen, &fattr, &scred);
3206         if (error)
3207                 goto out;

3209         smbfs_attr_touchdir(dnp);

3211         error = smbfs_nget(dvp, name, nmlen, &fattr, &vp);
3212         if (error)
3213                 goto out;

3215         if (name[0] == '.')
3216                 if ((hiderr = smbfs_smb_hideit(VTOSMB(vp), NULL, 0, &scred)))
3217                         SMBVDEBUG("hide failure %d\n", hiderr);

3219         /* Success! */
3220         *vpp = vp;
3221         error = 0;
3222 out:
3223         smb_credrele(&scred);
3224         smbfs_rw_exit(&dnp->r_rwlock);

3226         if (name != nm)
3227                 smbfs_name_free(name, nmlen);

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;

3354         smi = VTOSMI(vp);

3356         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3357                 return (EIO);

3359         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3360                 return (EIO);

3362         /*
3363          * Require read access in the directory.
3364          */
```



```
3365         error = smbfs_access(vp, VREAD, 0, cr, ct);
3366         if (error)
3367                 return (error);

3369         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));

3371         /*
3372          * Todo readdir cache here
2660          * XXX: Todo readdir cache here
2661          * Note: NFS code is just below this.
3373          *
3374          * I am serializing the entire readdir opreation
3375          * now since we have not yet implemented readdir
3376          * cache. This fix needs to be revisited once
3377          * we implement readdir cache.
3378          */
3379         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
3380                 return (EINTR);

3382         error = smbfs_readvdir(vp, uiop, cr, eofp, ct);

3384         smbfs_rw_exit(&np->r_lkserlock);

3386         return (error);
3387 }
```

*unchanged_portion_omitted_*

```
3606 /*
3607  * Here NFS has: nfs3_bio
3608  * See smbfs_bio above.
3609  */

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);

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 }
```

*unchanged_portion_omitted_*

```
3692 /* mmap support ******************************************************** */

3694 #ifdef DEBUG
3695 static int smbfs_lostpage = 0;  /* number of times we lost original page */
3696 #endif

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 *protp,
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;

3712         np = VTOSMB(vp);
3713         smi = VTOSMI(vp);

3715         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3716                 return (EIO);

3718         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3719                 return (EIO);

3721         if (vp->v_flag & VNOMAP)
3722                 return (ENOSYS);

3724         if (protp != NULL)
3725                 *protp = PROT_ALL;

3727         /*
3728          * Now valididate that the caches are up to date.
3729          */
3730         error = smbfs_validate_caches(vp, cr);
3731         if (error)
3732                 return (error);

3734 retry:
3735         mutex_enter(&np->r_statelock);

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         }

3760         mutex_exit(&np->r_statelock);

3762         error = pvn_getpages(smbfs_getapage, vp, off, len, protp,
3763             pl, plsz, seg, addr, rw, cr);

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         }

3787         return (error);
3788 }

3790 /*
3791  * Called from pvn_getpages to get a particular page.
3792  * Like nfs3_getapage
3793  */
3794 /* ARGSUSED */
3795 static int
3796 smbfs_getapage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protp,
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;

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);

3891                         pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
3892                             &io_len, blkoff, blksize, 0);

3894                         /*
3895                          * Some other thread has entered the page,
3896                          * so just use it.
3897                          */
3898                         if (pp == NULL)
3899                                 goto again;

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(btopr(io_len));

3908                         bp = pageio_setup(pp, io_len, vp, B_READ);
3909                         ASSERT(bp != NULL);

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);

3920                         bp->b_edev = 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);

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                         }

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
3975        if (pl == NULL)
3976                return (error);
3977
3978        if (error) {
3979                if (pp != NULL)
3980                        pvn_read_done(pp, B_ERROR);
3981                return (error);
3982        }
3983
3984        if (pagefound) {
3985                se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);
3986
3987                /*
3988                 * Page exists in the cache, acquire the appropriate lock.
3989                 * If this fails, start all over again.
3990                 */
3991                if ((pp = page_lookup(vp, off, se)) == NULL) {
3992 #ifdef DEBUG
3993                        smbfs_lostpage++;
3994 #endif
3995                        goto reread;
3996                }
3997                pl[0] = pp;
3998                pl[1] = NULL;
3999                return (0);
4000        }
4001
4002        if (pp != NULL)
4003                pvn_plist_init(pp, pl, plsz, off, io_len, rw);
4004
4005        return (error);
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         smbmntinfo_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                 *
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;

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);

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;

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                        }

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

4158        return (error);
4159 }

4161 /*
4162  * Write out a single page, possibly klustering adjacent dirty pages.
4163  *
4164  * Like nfs3_putapage / nfs3_sync_putapage
4165  */
4166 static int
4167 smbfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *lenp,
4168        int flags, cred_t *cr)
4169 {
4170        smbnode_t *np;
4171        u_offset_t io_off;
4172        u_offset_t lbn_off;
4173        u_offset_t lbn;
4174        size_t io_len;
4175        uint_t bsize;
4176        int error;

4178        np = VTOSMB(vp);

4180        ASSERT(!vn_is_readonly(vp));

4182        bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
4183        lbn = pp->p_offset / bsize;
4184        lbn_off = lbn * bsize;

4186        /*
4187         * Find a kluster that fits in one block, or in
4188         * one page if pages are bigger than blocks.  If
4189         * there is less file space allocated than a whole
4190         * page, we'll shorten the i/o request below.
4191         */
4192        pp = pvn_write_kluster(vp, pp, &io_off, &io_len, lbn_off,
4193            roundup(bsize, PAGESIZE), flags);

4195        /*
4196         * pvn_write_kluster shouldn't have returned a page with offset
4197         * behind the original page we were given.  Verify that.
4198         */
4199        ASSERT((pp->p_offset / bsize) >= lbn);

4201        /*
4202         * Now pp will have the list of kept dirty pages marked for
4203         * write back.  It will also handle invalidation and freeing
4204         * of pages that are not dirty.  Check for page length rounding
4205         * problems.
4206         */
4207        if (io_off + io_len > lbn_off + bsize) {
4208                ASSERT((io_off + io_len) - (lbn_off + bsize) < PAGESIZE);
4209                io_len = lbn_off + bsize - io_off;
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         }

4270         /*
4271          * NFS handles (flags & B_ASYNC) here...
4272          * (See nfs_async_putapage())
4273          *
4274          * This code section from: nfs3_sync_putapage()
4275          */
```



```
4277         flags |= B_WRITE;

4279         error = smbfs_rdwrlbn(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                  * Don't do this if this is an async thread because
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         }

4323         /* Now more code from: nfs3_putapage */

4325         if (offp)
4326                 *offp = io_off;
4327         if (lenp)
4328                 *lenp = io_len;

4330         return (error);
4331 }

4333 /*
4334  * NFS has this in nfs_client.c (shared by v2,v3,...)
4335  * We have it here so smbfs_putapage can be file scope.
4336  */
4337 void
4338 smbfs_invalidate_pages(vnode_t *vp, u_offset_t off, cred_t *cr)
4339 {
4340         smbnode_t *np;

4342         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;

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);

4357         (void) pvn_vplist_dirty(vp, off, smbfs_putapage,
4358             B_INVAL | B_TRUNC, cr);

4360         mutex_enter(&np->r_statelock);
4361         np->r_flags &= ~RTRUNCATE;
4362         cv_broadcast(&np->r_cv);
4363         mutex_exit(&np->r_statelock);
4364 }

4366 /* Like nfs3_map */

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         smbmntinfo_t    *smi;
4378         int             error;

4380         np = VTOSMB(vp);
4381         smi = VTOSMI(vp);

4383         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
4384                 return (EIO);

4386         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
4387                 return (EIO);

4389         if (vp->v_flag & VNOMAP)
4390                 return (ENOSYS);

4392         if (off < 0 || off + (ssize_t)len < 0)
4393                 return (ENXIO);

4395         if (vp->v_type != VREG)
4396                 return (ENODEV);

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 = smbfsgetattr(vp, &va, cr)) != 0)
4404                 return (error);

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          * In this case the map request must be refused.  We use
4410          * rp->r_lkserlock to avoid a race with concurrent lock requests.
4411          */
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          */

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);

4431         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp))) {
4432                 atomic_dec_uint(&np->r_inmap);
4433                 return (EINTR);
4434         }

4436         if (vp->v_flag & VNOCACHE) {
4437                 error = EAGAIN;
4438                 goto done;
4439         }

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         }

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         }

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;

4469         error = as_map(as, *addrp, len, segvn_create, &vn_a);
4470         as_rangeunlock(as);

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 *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         smbnode_t *np = VTOSMB(vp);
4485         boolean_t inc_fidrefs = B_FALSE;

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);

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         }

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 *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         smbnode_t *np = VTOSMB(vp);
4529         smbfs_delmap_args_t     *dmapp;
4530         int error;

4532         dmapp = kmem_zalloc(sizeof (*dmapp), KM_SLEEP);

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;

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);

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);

4566                 if (dmapp->dec_fidrefs) {
4567                         struct smb_cred scred;

4569                         (void) smbfs_rw_enter_sig(&np->r_lkserlock,
4570                             RW_WRITER, 0);
4571                         smb_credinit(&scred, dmapp->cr);

4573                         smbfs_rele_fid(np, &scred);

4575                         smb_credrele(&scred);
4576                         smbfs_rw_exit(&np->r_lkserlock);
4577                 }
4578                 kmem_free(dmapp, sizeof (*dmapp));
4579         }

4581         return (0);
4582 }

4584 /*
4585  * Remove some pages from an mmap'd vnode.  Flush any
4586  * dirty pages in the unmapped range.
4587  */
4588 /* ARGSUSED */
4589 static void
4590 smbfs_delmap_callback(struct as *as, void *arg, uint_t event)
4591 {
4592         vnode_t                 *vp;
4593         smbnode_t               *np;
4594         smbmntinfo_t            *smi;
4595         smbfs_delmap_args_t     *dmapp = arg;

4597         vp = dmapp->vp;
4598         np = VTOSMB(vp);
4599         smi = VTOSMI(vp);

4601         /* Decremented r_mapcnt in smbfs_delmap */

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                 (void) 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. ************************************************** */
4654 /*
4655  * XXX
4656  * This op may need to support PSARC 2007/440, nbmand changes for CIFS Service.
4657  */
4658 static int
4659 smbfs_frlock(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_frlock(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_FREESP, 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_FREESP)
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 = smbfsgetattr(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_SETSECATTR,     { .vop_setsecattr = smbfs_setsecattr },
4962         VOPNAME_GETSECATTR,     { .vop_getsecattr = smbfs_getsecattr },
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 };
```