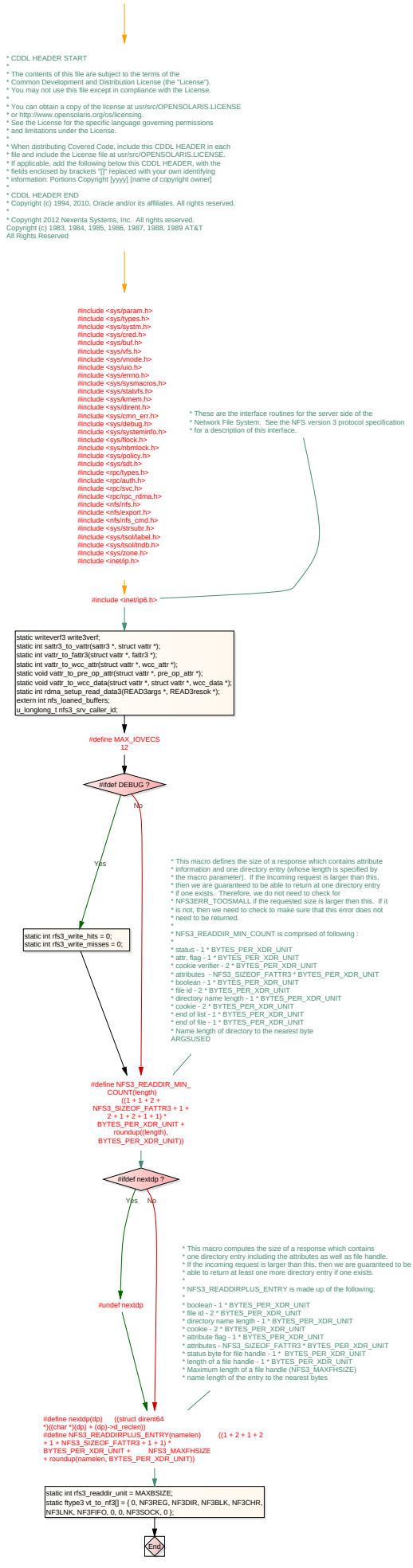
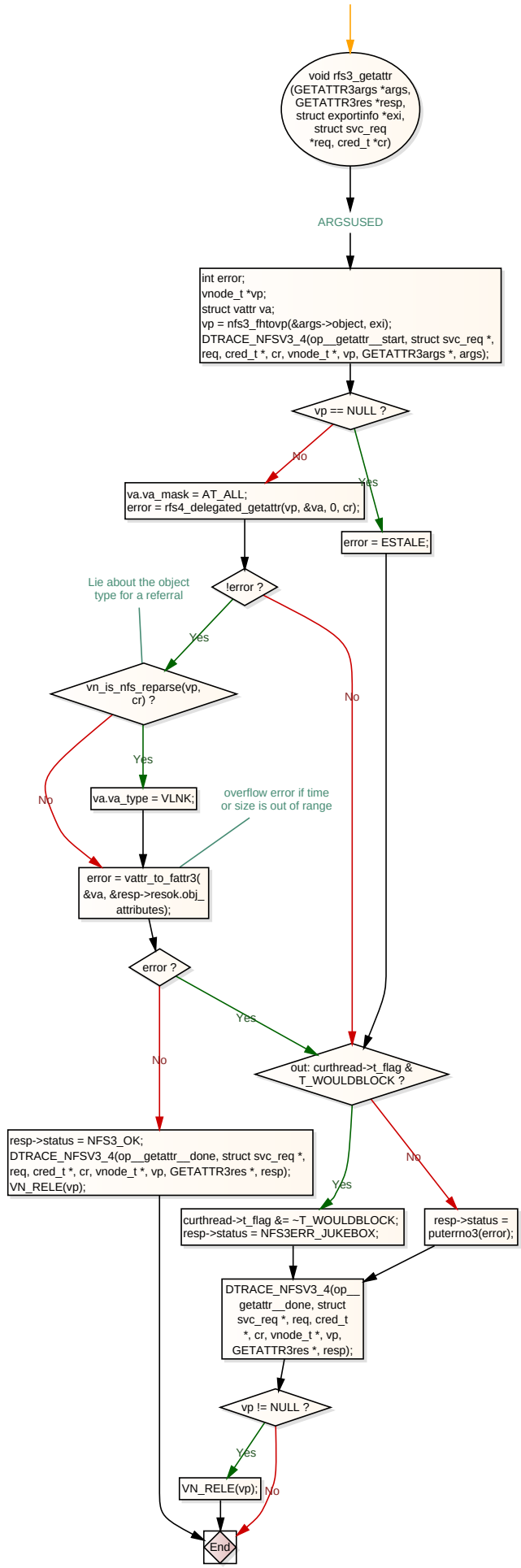
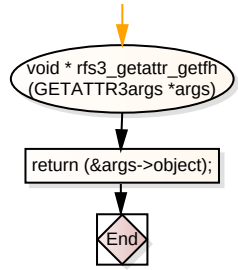
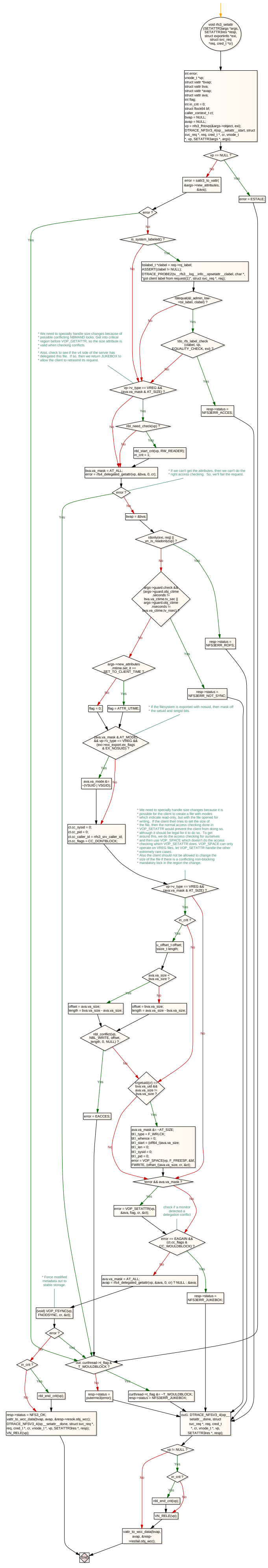
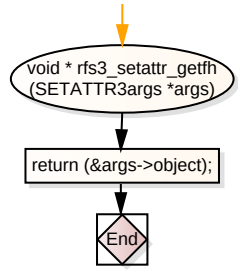
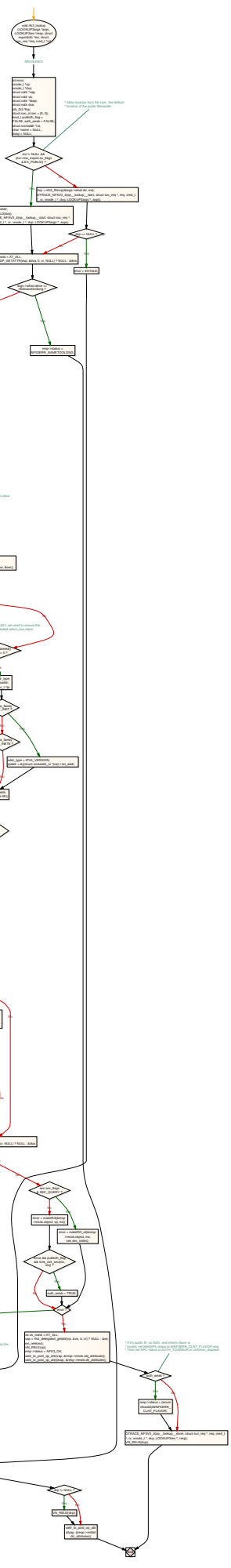

* CDDL HEADER START
*
* The contents of this file are subject to the terms of the
* Common Development and Distribution License (the "License").
* You may not use this file except in compliance with the License.
*
* You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
* or http://www.opensolaris.org/os/licensing.
* 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) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
*
* Copyright 2012 Nexenta Systems, Inc. All rights reserved.
Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T
All Rights Reserved
#include <sys/param.h>
#include <sys/types.h>
#include <sys/systm.h>
#include <sys/cred.h>
#include <sys/buf.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/uio.h>
#include <sys/errno.h>
#include <sys/sysmacros.h>
#include <sys/statvfs.h>
#include <sys/kmem.h>
#include <sys/dirent.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/systeminfo.h>
#include <sys/flock.h>
#include <sys/nbmlock.h>
#include <sys/policy.h>
#include <sys/sdt.h>
#include <rpc/types.h>
#include <rpc/auth.h>
#include <rpc/svc.h>
#include <rpc/rpc_rdma.h>
#include <nfs/nfs.h>
#include <nfs/export.h>
#include <nfs/nfs_cmd.h>
#include <sys/strsubr.h>
#include <sys/tsol/label.h>
#include <sys/tsol/tndb.h>
#include <sys/zone.h>
#include <inet/ip.h>
* These are the interface routines for the server side of the
* Network File System. See the NFS version 3 protocol specification
* for a description of this interface.
#include <inet/ip6.h>
static writeverf3 write3verf;
static int sattr3_to_vattr(sattr3 *, struct vattr *);
static int vattr_to_fattr3(struct vattr *, fattr3 *);
static int vattr_to_wcc_attr(struct vattr *, wcc_attr *);
static void vattr_to_pre_op_attr(struct vattr *, pre_op_attr *);
static void vattr_to_wcc_data(struct vattr *, struct vattr *, wcc_data *);
static int rdma_setup_read_data3(READ3args *, READ3resok *);
extern int nfs_loaned_buffers;
u_longlong_t nfs3_srv_caller_id;
#define MAX_IOVECS
12
#ifdef DEBUG ?
No
Yes
static int rfs3_write_hits = 0;
static int rfs3_write_misses = 0;
* This macro defines the size of a response which contains attribute
* information and one directory entry (whose length is specified by
* the macro parameter). If the incoming request is larger than this,
* then we are guaranteed to be able to return at one directory entry
* if one exists. Therefore, we do not need to check for
* NFS3ERR_TOOSMALL if the requested size is larger then this. If it
* is not, then we need to check to make sure that this error does not
* need to be returned.
*
* NFS3_READDIR_MIN_COUNT is comprised of following :
*
* status - 1 * BYTES_PER_XDR_UNIT
* attr. flag - 1 * BYTES_PER_XDR_UNIT
* cookie verifier - 2 * BYTES_PER_XDR_UNIT
* attributes - NFS3_SIZEOF_FATTR3 * BYTES_PER_XDR_UNIT
* boolean - 1 * BYTES_PER_XDR_UNIT
* file id - 2 * BYTES_PER_XDR_UNIT
* directory name length - 1 * BYTES_PER_XDR_UNIT
* cookie - 2 * BYTES_PER_XDR_UNIT
* end of list - 1 * BYTES_PER_XDR_UNIT
* end of file - 1 * BYTES_PER_XDR_UNIT
* Name length of directory to the nearest byte
ARGSUSED
#define NFS3_READDIR_MIN_
COUNT(length)
((1 + 1 + 2 +
NFS3_SIZEOF_FATTR3 + 1 +
2 + 1 + 2 + 1 + 1) *
BYTES_PER_XDR_UNIT +
roundup((length),
BYTES_PER_XDR_UNIT))
#ifdef nextdp ?
Yes
No
#undef nextdp
* This macro computes the size of a response which contains
* one directory entry including the attributes as well as file handle.
* If the incoming request is larger than this, then we are guaranteed to be
* able to return at least one more directory entry if one exists.
*
* NFS3_READDIRPLUS_ENTRY is made up of the following:
*
* boolean - 1 * BYTES_PER_XDR_UNIT
* file id - 2 * BYTES_PER_XDR_UNIT
* directory name length - 1 * BYTES_PER_XDR_UNIT
* cookie - 2 * BYTES_PER_XDR_UNIT
* attribute flag - 1 * BYTES_PER_XDR_UNIT
* attributes - NFS3_SIZEOF_FATTR3 * BYTES_PER_XDR_UNIT
* status byte for file handle - 1 * BYTES_PER_XDR_UNIT
* length of a file handle - 1 * BYTES_PER_XDR_UNIT
* Maximum length of a file handle (NFS3_MAXFHSIZE)
* name length of the entry to the nearest bytes
#define nextdp(dp)   ((struct dirent64
*)((char *)(dp) + (dp)->d_reclen))
#define NFS3_READDIRPLUS_ENTRY(namelen)   ((1 + 2 + 1 + 2
+ 1 + NFS3_SIZEOF_FATTR3 + 1 + 1) *
BYTES_PER_XDR_UNIT +   NFS3_MAXFHSIZE
+ roundup(namelen, BYTES_PER_XDR_UNIT))
static int rfs3_readdir_unit = MAXBSIZE;
static ftype3 vt_to_nf3[] = { 0, NF3REG, NF3DIR, NF3BLK, NF3CHR,
NF3LNK, NF3FIFO, 0, 0, NF3SOCK, 0 };
End

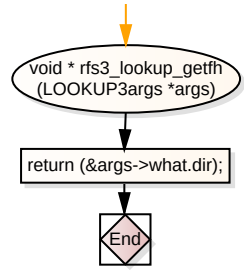

void rfs3_getattr (GETATTR3args *args, GETATTR3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
ARGSUSED
int error;
vnode_t *vp;
struct vattr va;
vp = nfs3_fhtovp(&args->object, exi);
DTRACE_NFSV3_4(op__getattr__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, GETATTR3args *, args);
vp == NULL ?
No
Yes
va.va_mask = AT_ALL;
error = rfs4_delegated_getattr(vp, &va, 0, cr);
error = ESTALE;
!error ?
Lie about the object type for a referral
Yes
No
vn_is_nfs_reparse(vp, cr) ?
Yes
No
va.va_type = VLNK;
overflow error if time or size is out of range
error = vattr_to_fattr3(&va, &resp->resok.obj_ attributes);
error ?
Yes
No
out: curthread->t_flag & T_WOULDBLOCK ?
Yes
No
resp->status = NFS3_OK;
DTRACE_NFSV3_4(op__getattr__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, GETATTR3res *, resp);
VN_RELE(vp);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
resp->status = puterrno3(error);
DTRACE_NFSV3_4(op__ getattr__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, GETATTR3res *, resp);
vp != NULL ?
Yes
VN_RELE(vp);
No
End

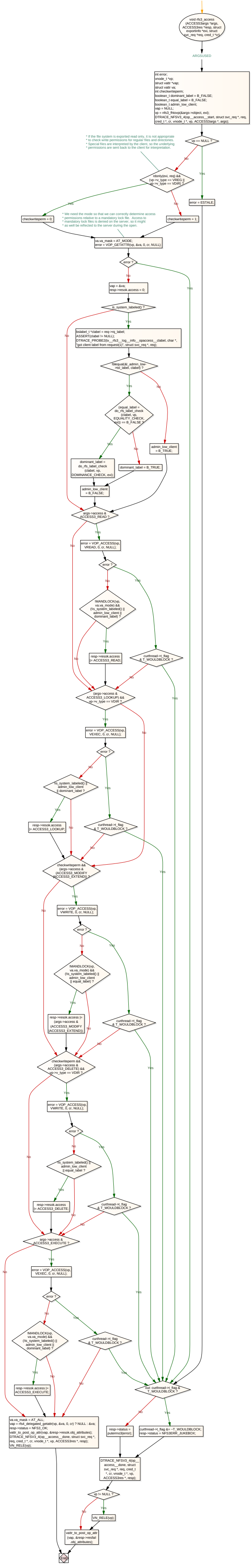

void * rfs3_getattr_getfh
(GETATTR3args *args)
return (&args->object);
End

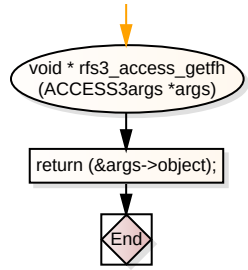

void rfs3_setattr (SETATTR3args *args, SETATTR3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp;
struct vattr *bvap;
struct vattr bva;
struct vattr *avap;
struct vattr ava;
int flag;
int in_crit = 0;
struct flock64 bf;
caller_context_t ct;
bvap = NULL;
avap = NULL;
vp = nfs3_fhtovp(&args->object, exi);
DTRACE_NFSV3_4(op__setattr__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, SETATTR3args *, args);
vp == NULL ?
error = sattr3_to_vattr(&args->new_attributes, &ava);
error = ESTALE;
error ?
is_system_labeled() ?
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opsetattr__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
!do_rfs_label_check (clabel, vp, EQUALITY_CHECK, exi) ?
* We need to specially handle size changes because of
* possible conflicting NBMAND locks. Get into critical
* region before VOP_GETATTR, so the size attribute is
* valid when checking conflicts.
*
* Also, check to see if the v4 side of the server has
* delegated this file. If so, then we return JUKEBOX to
* allow the client to retrasmit its request.
vp->v_type == VREG && (ava.va_mask & AT_SIZE) ?
resp->status = NFS3ERR_ACCES;
nbl_need_check(vp) ?
nbl_start_crit(vp, RW_READER);
in_crit = 1;
bva.va_mask = AT_ALL;
error = rfs4_delegated_getattr(vp, &bva, 0, cr);
* If we can't get the attributes, then we can't do the
* right access checking. So, we'll fail the request.
error ?
bvap = &bva;
rdonly(exi, req) || vn_is_readonly(vp) ?
args->guard.check && (args->guard.obj_ctime .seconds != bva.va_ctime.tv_sec || args->guard.obj_ctime .nseconds != bva.va_ctime.tv_nsec) ?
resp->status = NFS3ERR_ROFS;
args->new_attributes .mtime.set_it == SET_TO_CLIENT_TIME ?
resp->status = NFS3ERR_NOT_SYNC;
flag = 0;
flag = ATTR_UTIME;
* If the filesystem is exported with nosuid, then mask off
* the setuid and setgid bits.
(ava.va_mask & AT_MODE) && vp->v_type == VREG && (exi->exi_export.ex_flags & EX_NOSUID) ?
ava.va_mode &= ~(VSUID | VSGID);
ct.cc_sysid = 0;
ct.cc_pid = 0;
ct.cc_caller_id = nfs3_srv_caller_id;
ct.cc_flags = CC_DONTBLOCK;
* We need to specially handle size changes because it is
* possible for the client to create a file with modes
* which indicate read-only, but with the file opened for
* writing. If the client then tries to set the size of
* the file, then the normal access checking done in
* VOP_SETATTR would prevent the client from doing so,
* although it should be legal for it to do so. To get
* around this, we do the access checking for ourselves
* and then use VOP_SPACE which doesn't do the access
* checking which VOP_SETATTR does. VOP_SPACE can only
* operate on VREG files, let VOP_SETATTR handle the other
* extremely rare cases.
* Also the client should not be allowed to change the
* size of the file if there is a conflicting non-blocking
* mandatory lock in the region the change.
vp->v_type == VREG && (ava.va_mask & AT_SIZE) ?
in_crit ?
u_offset_t offset;
ssize_t length;
ava.va_size < bva.va_size ?
offset = ava.va_size;
length = bva.va_size - ava.va_size;
offset = bva.va_size;
length = ava.va_size - bva.va_size;
nbl_conflict(vp, NBL_WRITE, offset, length, 0, NULL) ?
crgetuid(cr) == bva.va_uid && ava.va_size != bva.va_size ?
error = EACCES;
ava.va_mask &= ~AT_SIZE;
bf.l_type = F_WRLCK;
bf.l_whence = 0;
bf.l_start = (off64_t)ava.va_size;
bf.l_len = 0;
bf.l_sysid = 0;
bf.l_pid = 0;
error = VOP_SPACE(vp, F_FREESP, &bf, FWRITE, (offset_t)ava.va_size, cr, &ct);
!error && ava.va_mask ?
error = VOP_SETATTR(vp, &ava, flag, cr, &ct);
check if a monitor detected a delegation conflict
error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK) ?
ava.va_mask = AT_ALL;
avap = rfs4_delegated_getattr(vp, &ava, 0, cr) ? NULL : &ava;
resp->status = NFS3ERR_JUKEBOX;
* Force modified metadata out to stable storage.
(void) VOP_FSYNC(vp, FNODSYNC, cr, &ct);
error ?
in_crit ?
out: curthread->t_flag & T_WOULDBLOCK ?
nbl_end_crit(vp);
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
resp->status = NFS3_OK;
vattr_to_wcc_data(bvap, avap, &resp->resok.obj_wcc);
DTRACE_NFSV3_4(op__setattr__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, SETATTR3res *, resp);
VN_RELE(vp);
out1: DTRACE_NFSV3_4(op__setattr__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, SETATTR3res *, resp);
vp != NULL ?
in_crit ?
nbl_end_crit(vp);
VN_RELE(vp);
vattr_to_wcc_data(bvap, avap, &resp->resfail.obj_wcc);
End
Yes
No

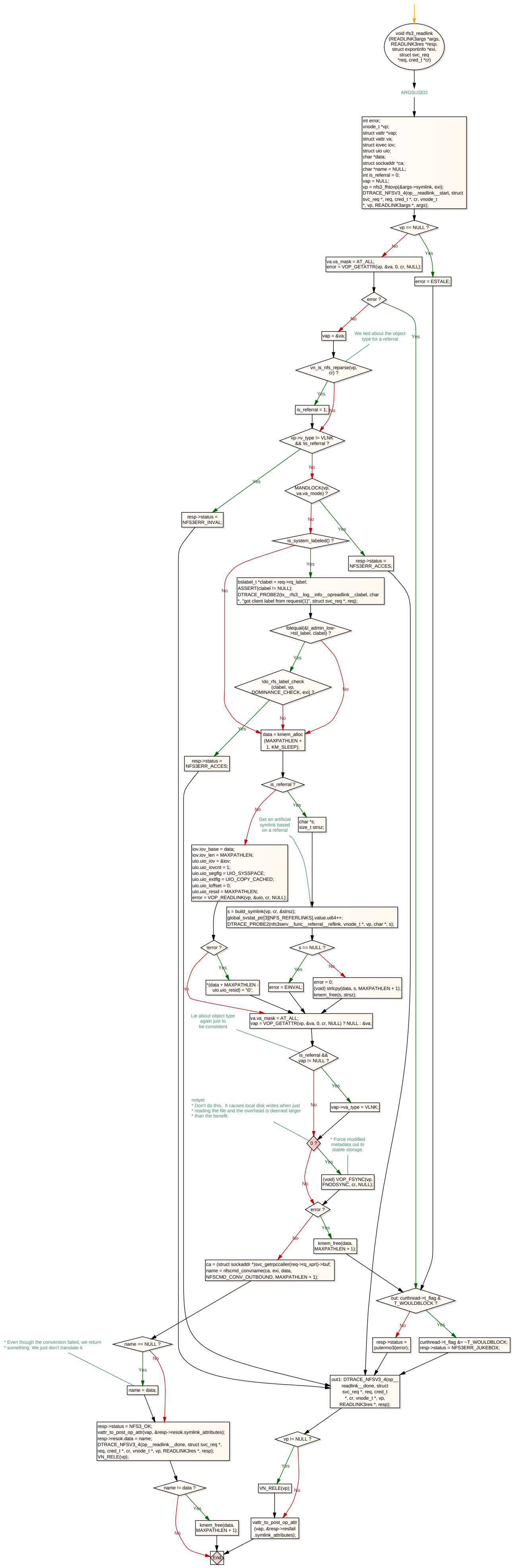

void * rfs3_setattr_getfh
(SETATTR3args *args)
return (&args->object);
End



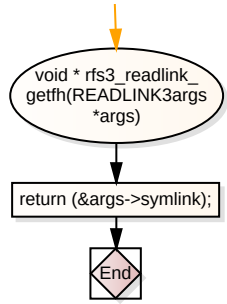

void * rfs3_lookup_getfh
(LOOKUP3args *args)
return (&args->what.dir);
End

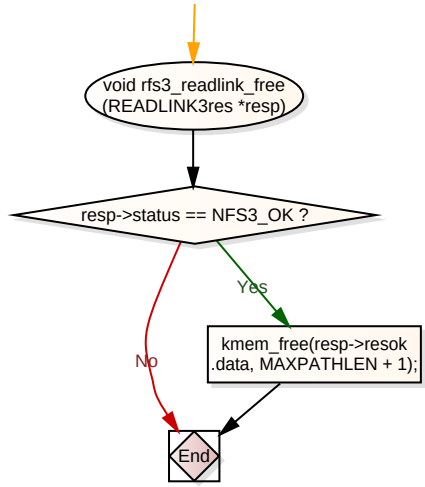

void rfs3_access (ACCESS3args *args, ACCESS3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
ARGSUSED
int error;
vnode_t *vp;
struct vattr *vap;
struct vattr va;
int checkwriteperm;
boolean_t dominant_label = B_FALSE;
boolean_t equal_label = B_FALSE;
boolean_t admin_low_client;
vap = NULL;
vp = nfs3_fhtovp(&args->object, exi);
DTRACE_NFSV3_4(op__access__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, ACCESS3args *, args);
vp == NULL ?
error = ESTALE;
* If the file system is exported read only, it is not appropriate
* to check write permissions for regular files and directories.
* Special files are interpreted by the client, so the underlying
* permissions are sent back to the client for interpretation.
rdonly(exi, req) && (vp->v_type == VREG || vp->v_type == VDIR) ?
checkwriteperm = 0;
checkwriteperm = 1;
* We need the mode so that we can correctly determine access
* permissions relative to a mandatory lock file. Access to
* mandatory lock files is denied on the server, so it might
* as well be reflected to the server during the open.
va.va_mask = AT_MODE;
error = VOP_GETATTR(vp, &va, 0, cr, NULL);
error ?
vap = &va;
resp->resok.access = 0;
is_system_labeled() ?
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opaccess__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
(equal_label = do_rfs_label_check (clabel, vp, EQUALITY_CHECK, exi)) == B_FALSE ?
admin_low_client = B_TRUE;
dominant_label = do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi);
dominant_label = B_TRUE;
admin_low_client = B_FALSE;
args->access & ACCESS3_READ ?
error = VOP_ACCESS(vp, VREAD, 0, cr, NULL);
error ?
!MANDLOCK(vp, va.va_mode) && (!is_system_labeled() || admin_low_client || dominant_label) ?
resp->resok.access |= ACCESS3_READ;
curthread->t_flag & T_WOULDBLOCK ?
(args->access & ACCESS3_LOOKUP) && vp->v_type == VDIR ?
error = VOP_ACCESS(vp, VEXEC, 0, cr, NULL);
error ?
!is_system_labeled() || admin_low_client || dominant_label ?
resp->resok.access |= ACCESS3_LOOKUP;
curthread->t_flag & T_WOULDBLOCK ?
checkwriteperm && (args->access & (ACCESS3_MODIFY |ACCESS3_EXTEND)) ?
error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL);
error ?
!MANDLOCK(vp, va.va_mode) && (!is_system_labeled() || admin_low_client || equal_label) ?
resp->resok.access |= (args->access & (ACCESS3_MODIFY |ACCESS3_EXTEND));
curthread->t_flag & T_WOULDBLOCK ?
checkwriteperm && (args->access & ACCESS3_DELETE) && vp->v_type == VDIR ?
error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL);
error ?
!is_system_labeled() || admin_low_client || equal_label ?
resp->resok.access |= ACCESS3_DELETE;
curthread->t_flag & T_WOULDBLOCK ?
args->access & ACCESS3_EXECUTE ?
error = VOP_ACCESS(vp, VEXEC, 0, cr, NULL);
error ?
!MANDLOCK(vp, va.va_mode) && (!is_system_labeled() || admin_low_client || dominant_label) ?
curthread->t_flag & T_WOULDBLOCK ?
resp->resok.access |= ACCESS3_EXECUTE;
out: curthread->t_flag & T_WOULDBLOCK ?
va.va_mask = AT_ALL;
vap = rfs4_delegated_getattr(vp, &va, 0, cr) ? NULL : &va;
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.obj_attributes);
DTRACE_NFSV3_4(op__access__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, ACCESS3res *, resp);
VN_RELE(vp);
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
DTRACE_NFSV3_4(op__access__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, ACCESS3res *, resp);
vp != NULL ?
VN_RELE(vp);
vattr_to_post_op_attr (vap, &resp->resfail .obj_attributes);
End

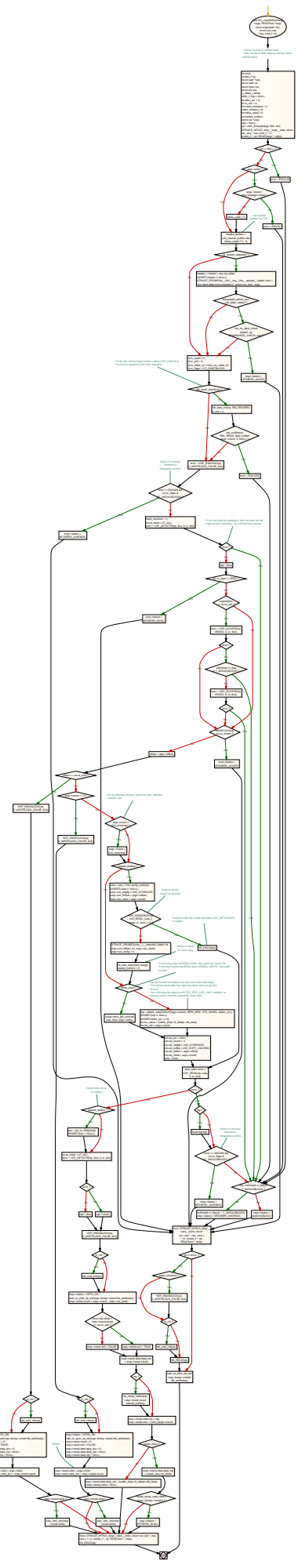

void * rfs3_access_getfh
(ACCESS3args *args)
return (&args->object);
End

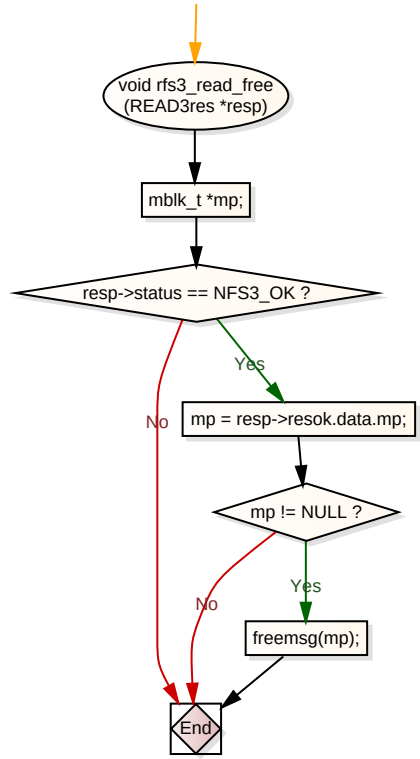


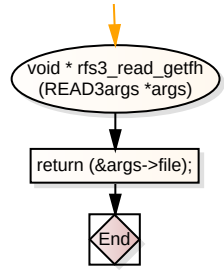
void nfs3_readlink (READLINK3args *args, READLINK3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
ARGSUSED
int error;
vnode_t *vp;
struct vattr *vap;
struct vattr va;
struct iovec iov;
struct uio uio;
char *data;
struct sockaddr *ca;
char *name = NULL;
int is_referral = 0;
vap = NULL;
vp = nfs3_fhtovp(&args->symlink, exi);
DTRACE_NFSV3_4(op__readlink__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, READLINK3args *, args);
vp == NULL ?
No
Yes
va.va_mask = AT_ALL;
error = VOP_GETATTR(vp, &va, 0, cr, NULL);
error = ESTALE;
error ?
No
Yes
vap = &va;
We lied about the object type for a referral
vn_is_nfs_reparse(vp, cr) ?
Yes
No
is_referral = 1;
vp->v_type != VLNK && !is_referral ?
Yes
No
resp->status = NFS3ERR_INVAL;
MANDLOCK(vp, va.va_mode) ?
No
Yes
is_system_labeled() ?
resp->status = NFS3ERR_ACCES;
Yes
No
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opreadlink__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi) ?
No
Yes
data = kmem_alloc (MAXPATHLEN + 1, KM_SLEEP);
resp->status = NFS3ERR_ACCES;
is_referral ?
No
Yes
Get an artificial symlink based on a referral
char *s;
size_t strsz;
iov.iov_base = data;
iov.iov_len = MAXPATHLEN;
uio.uio_iov = &iov;
uio.uio_iovcnt = 1;
uio.uio_segflg = UIO_SYSSPACE;
uio.uio_extflg = UIO_COPY_CACHED;
uio.uio_loffset = 0;
uio.uio_resid = MAXPATHLEN;
error = VOP_READLINK(vp, &uio, cr, NULL);
s = build_symlink(vp, cr, &strsz);
global_svstat_ptr[3][NFS_REFERLINKS].value.ui64++;
DTRACE_PROBE2(nfs3serv__func__referral__reflink, vnode_t *, vp, char *, s);
!error ?
Yes
No
*(data + MAXPATHLEN - uio.uio_resid) = '\0';
s == NULL ?
Yes
No
error = EINVAL;
error = 0;
(void) strlcpy(data, s, MAXPATHLEN + 1);
kmem_free(s, strsz);
Lie about object type again just to be consistent
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
is_referral && vap != NULL ?
Yes
No
vap->va_type = VLNK;
notyet
* Don't do this. It causes local disk writes when just
* reading the file and the overhead is deemed larger
* than the benefit.
0 ?
* Force modified metadata out to stable storage.
Yes
No
(void) VOP_FSYNC(vp, FNODSYNC, cr, NULL);
error ?
Yes
No
kmem_free(data, MAXPATHLEN + 1);
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, exi, data, NFSCMD_CONV_OUTBOUND, MAXPATHLEN + 1);
out: curthread->t_flag & T_WOULDBLOCK ?
No
Yes
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
* Even though the conversion failed, we return
* something. We just don't translate it.
name == NULL ?
Yes
No
name = data;
out1: DTRACE_NFSV3_4(op__readlink__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, READLINK3res *, resp);
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.symlink_attributes);
resp->resok.data = name;
DTRACE_NFSV3_4(op__readlink__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, READLINK3res *, resp);
VN_RELE(vp);
vp != NULL ?
Yes
No
VN_RELE(vp);
vattr_to_post_op_attr (vap, &resp->resfail .symlink_attributes);
name != data ?
Yes
No
kmem_free(data, MAXPATHLEN + 1);
End

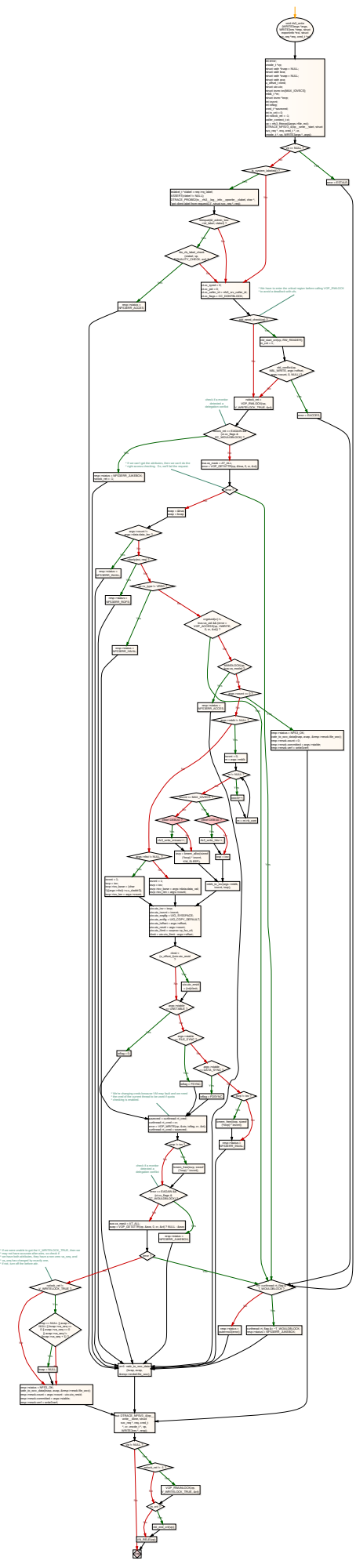

void * rfs3_readlink_
getfh(READLINK3args
*args)
return (&args->symlink);
End

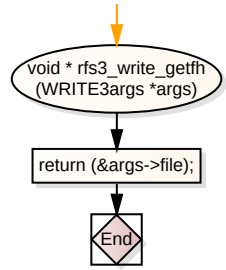

void rfs3_readlink_free
(READLINK3res *resp)
resp->status == NFS3_OK ?
Yes
No
kmem_free(resp->resok
.data, MAXPATHLEN + 1);
End

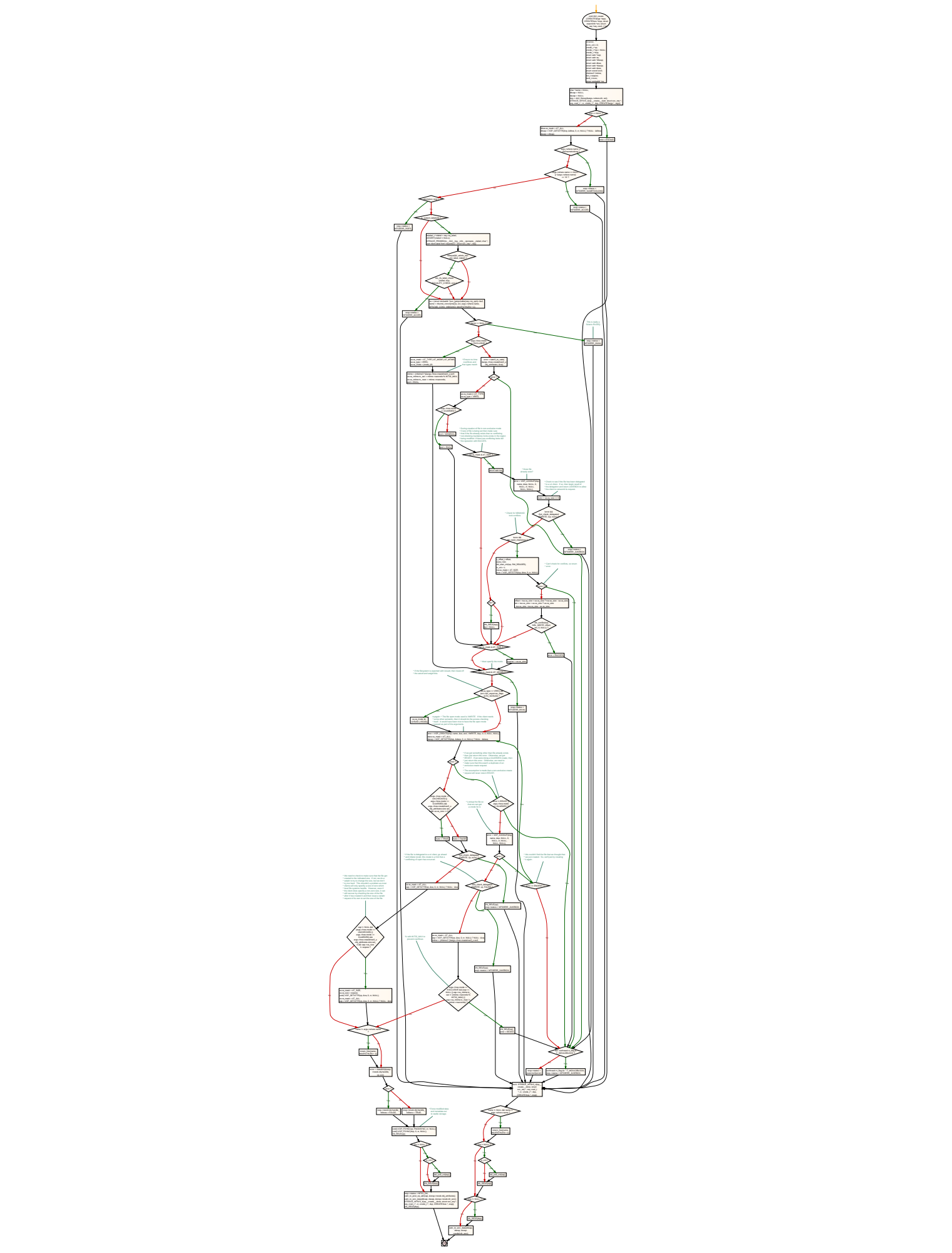


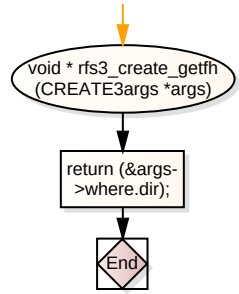

void rfs3_read_free
(READ3res *resp)
mblk_t *mp;
resp->status == NFS3_OK ?
Yes
No
mp = resp->resok.data.mp;
mp != NULL ?
Yes
No
freemsg(mp);
End

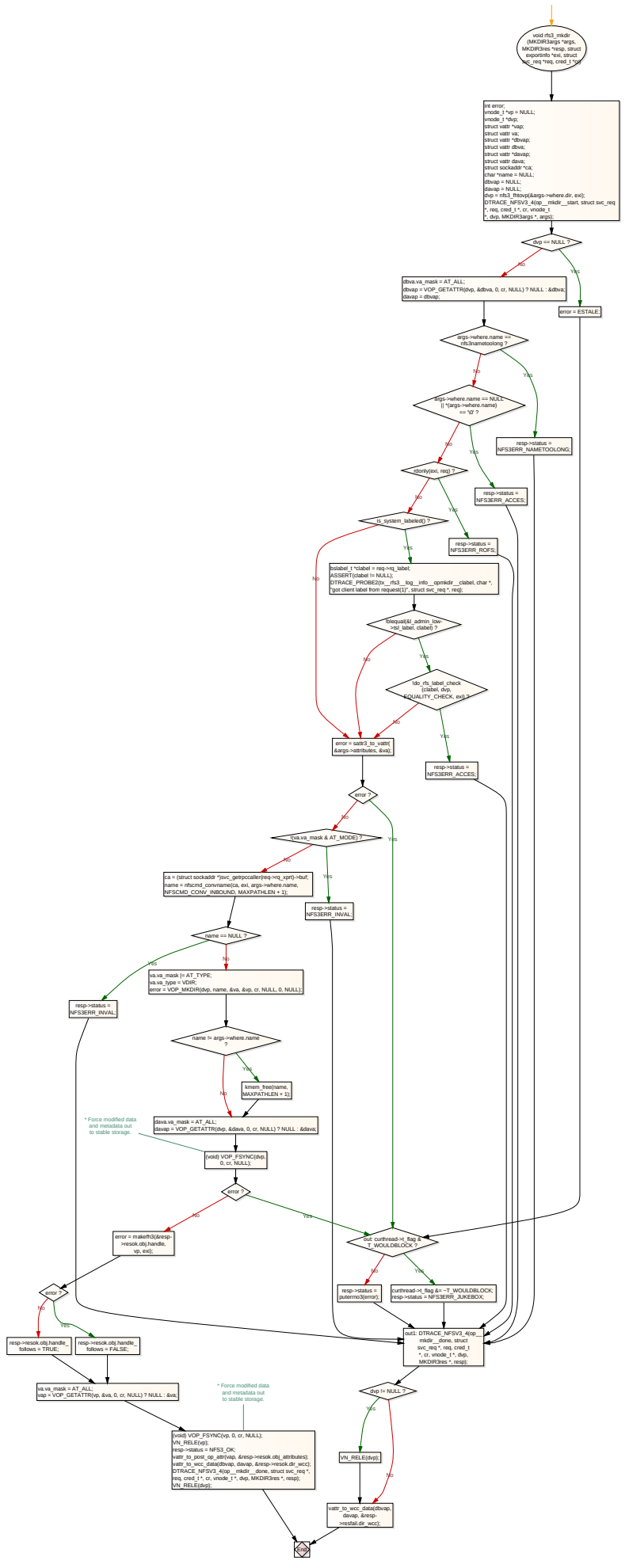

void * rfs3_read_getfh
(READ3args *args)
return (&args->file);
End

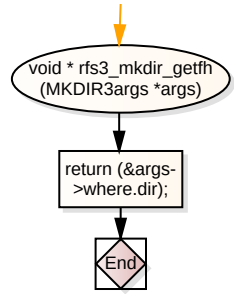


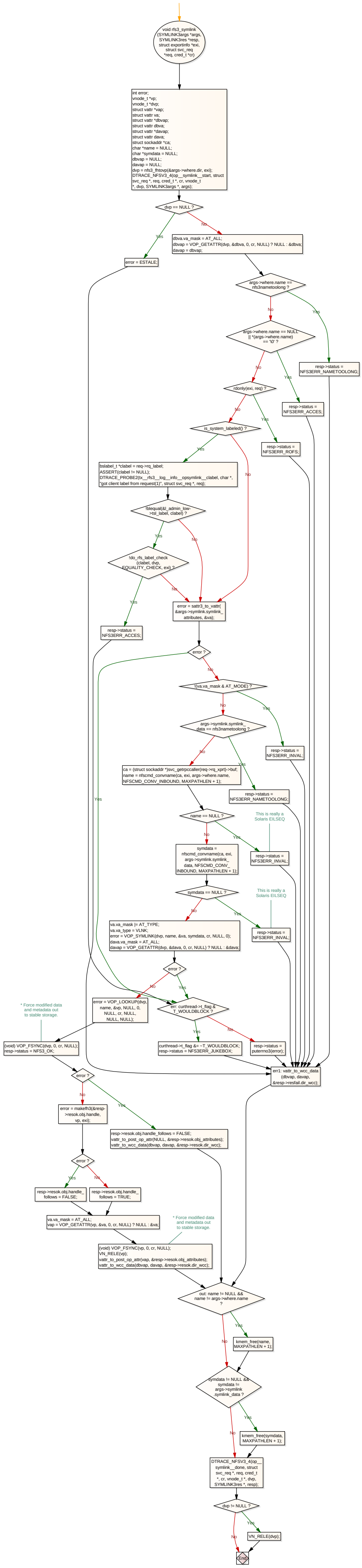

void * rfs3_write_getfh
(WRITE3args *args)
return (&args->file);
End



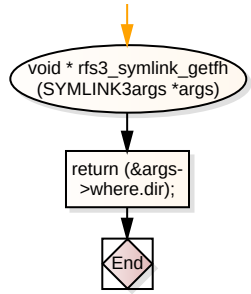

void * rfs3_create_getfh
(CREATE3args *args)
return (&args-
>where.dir);
End

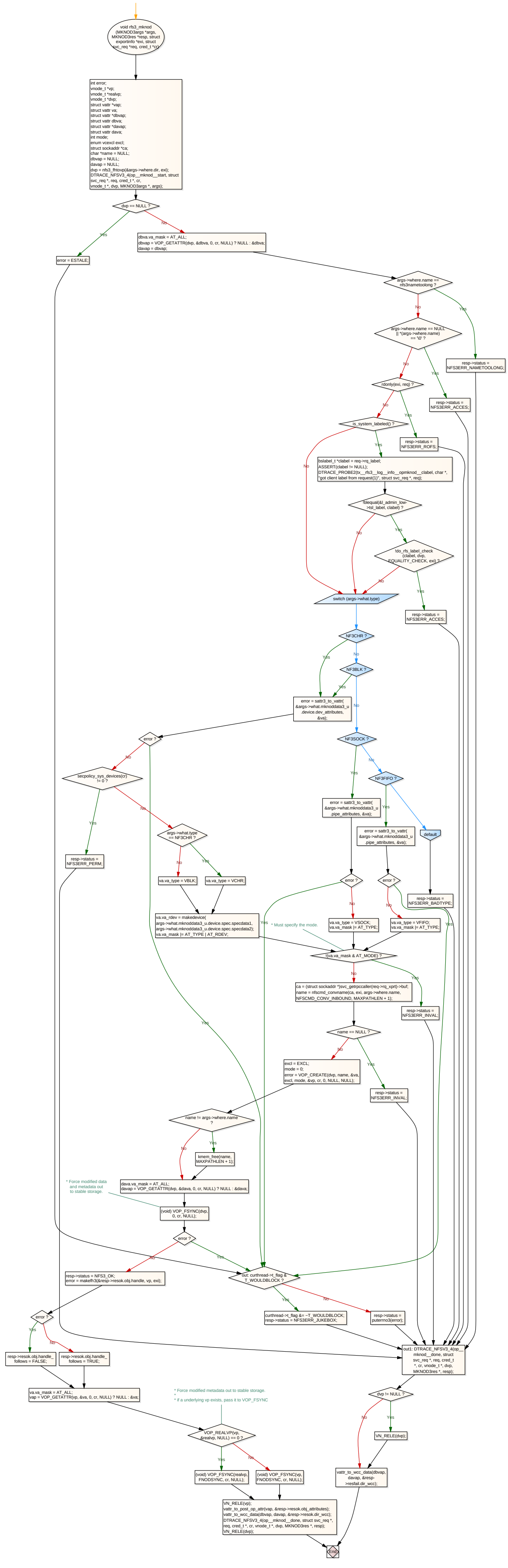

void rfs3_mkdir (MKDIR3args *args, MKDIR3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp = NULL;
vnode_t *dvp;
struct vattr *vap;
struct vattr va;
struct vattr *dbvap;
struct vattr dbva;
struct vattr *davap;
struct vattr dava;
struct sockaddr *ca;
char *name = NULL;
dbvap = NULL;
davap = NULL;
dvp = nfs3_fhtovp(&args->where.dir, exi);
DTRACE_NFSV3_4(op__mkdir__start, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, MKDIR3args *, args);
dvp == NULL ?
No
Yes
dbva.va_mask = AT_ALL;
dbvap = VOP_GETATTR(dvp, &dbva, 0, cr, NULL) ? NULL : &dbva;
davap = dbvap;
error = ESTALE;
args->where.name == nfs3nametoolong ?
No
Yes
*args->where.name == NULL || *(args->where.name) == '\0' ?
No
Yes
resp->status = NFS3ERR_NAMETOOLONG;
rdonly(exi, req) ?
No
Yes
resp->status = NFS3ERR_ACCES;
rs_system_labeled() ?
No
Yes
resp->status = NFS3ERR_ROFS;
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__nfs3__log__info__opmkdir__clabel, char *, "got client label from request(1)", struct svc_req *, req);
bslequal(&l_admin_low->sl_label, clabel) ?
No
Yes
!do_rfs_label_check (clabel, dvp, EQUALITY_CHECK, exi) ?
No
Yes
error = sattr3_to_vattr (&args->attributes, &va);
resp->status = NFS3ERR_ACCES;
error ?
No
Yes
!(va.va_mask & AT_MODE) ?
No
Yes
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, exi, args->where.name, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_INVAL;
name == NULL ?
Yes
No
resp->status = NFS3ERR_INVAL;
va.va_mask |= AT_TYPE;
va.va_type = VDIR;
error = VOP_MKDIR(dvp, name, &va, &vp, cr, NULL, 0, NULL);
name != args->where.name ?
Yes
No
kmem_free(name, MAXPATHLEN + 1);
* Force modified data and metadata out to stable storage.
davap.va_mask = AT_ALL;
davap = VOP_GETATTR(dvp, &dava, 0, cr, NULL) ? NULL : &dava;
(void) VOP_FSYNC(dvp, 0, cr, NULL);
error ?
No
Yes
error = makefh3(&resp->resok.obj.handle, vp, exi);
out: curthread->t_flag & T_WOULDBLOCK ?
No
Yes
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
error ?
No
Yes
resp->resok.obj.handle_follows = TRUE;
resp->resok.obj.handle_follows = FALSE;
out1: DTRACE_NFSV3_4(op__mkdir__done, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, MKDIR3res *, resp);
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(vp, 0, cr, NULL);
VN_RELE(vp);
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.obj_attributes);
vattr_to_wcc_data(dbvap, davap, &resp->resok.dir_wcc);
DTRACE_NFSV3_4(op__mkdir__done, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, MKDIR3res *, resp);
VN_RELE(dvp);
dvp != NULL ?
Yes
No
VN_RELE(dvp);
vattr_to_wcc_data(dbvap, davap, &resp->resfail.dir_wcc);
END

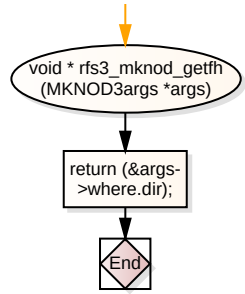

void * rfs3_mkdir_getfh
(MKDIR3args *args)
return (&args-
>where.dir);
End

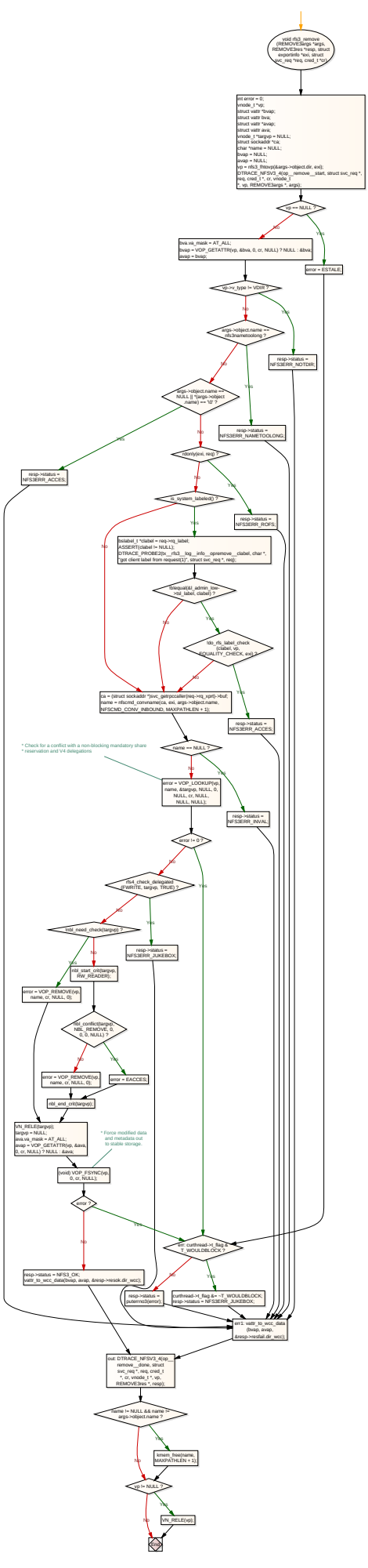


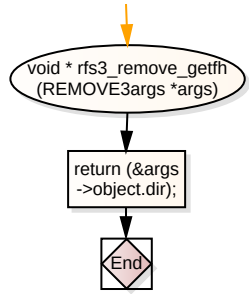
void rfs3_symlink (SYMLINK3args *args, SYMLINK3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error; vnode_t *vp; vnode_t *dvp; struct vattr *vap; struct vattr va; struct vattr *dbvap; struct vattr dbva; struct vattr *davap; struct vattr dava; struct sockaddr *ca; char *name = NULL; char *symdata = NULL; dbvap = NULL; davap = NULL; dvp = nfs3_fhtovp(&args->where.dir, exi); DTRACE_NFSV3_4(op__symlink__start, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, SYMLINK3args *, args);
dvp == NULL ?
error = ESTALE;
dbva.va_mask = AT_ALL; dbvap = VOP_GETATTR(dvp, &dbva, 0, cr, NULL) ? NULL : &dbva; davap = dbvap;
args->where.name == nfs3nametoolong ?
args->where.name == NULL || *(args->where.name) == '\0' ?
resp->status = NFS3ERR_NAMETOOLONG;
rdonly(exi, req) ?
resp->status = NFS3ERR_ACCES;
is_system_labeled() ?
resp->status = NFS3ERR_ROFS;
bslabel_t *clabel = req->rq_label; ASSERT(clabel != NULL); DTRACE_PROBE2(tx__rfs3__log__info__opsymlink__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
!do_rfs_label_check (clabel, dvp, EQUALITY_CHECK, exi) ?
error = sattr3_to_vattr(&args->symlink.symlink_attributes, &va);
resp->status = NFS3ERR_ACCES;
error ?
!(va.va_mask & AT_MODE) ?
args->symlink.symlink_data == nfs3nametoolong ?
resp->status = NFS3ERR_INVAL;
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf; name = nfscmd_convname(ca, exi, args->where.name, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_NAMETOOLONG;
name == NULL ?
This is really a Solaris EILSEQ
symdata = nfscmd_convname(ca, exi, args->symlink.symlink_data, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_INVAL;
symdata == NULL ?
This is really a Solaris EILSEQ
va.va_mask |= AT_TYPE; va.va_type = VLNK; error = VOP_SYMLINK(dvp, name, &va, symdata, cr, NULL, 0); dava.va_mask = AT_ALL; davap = VOP_GETATTR(dvp, &dava, 0, cr, NULL) ? NULL : &dava;
resp->status = NFS3ERR_INVAL;
error ?
* Force modified data and metadata out to stable storage.
error = VOP_LOOKUP(dvp, name, &vp, NULL, 0, NULL, cr, NULL, NULL, NULL);
err: curthread->t_flag & T_WOULDBLOCK ?
(void) VOP_FSYNC(dvp, 0, cr, NULL); resp->status = NFS3_OK;
curthread->t_flag &= ~T_WOULDBLOCK; resp->status = NFS3ERR_JUKEBOX;
resp->status = puterrno3(error);
err1: vattr_to_wcc_data (dbvap, davap, &resp->resfail.dir_wcc);
error ?
error = makefh3(&resp->resok.obj.handle, vp, exi);
resp->resok.obj.handle_follows = FALSE; vattr_to_post_op_attr(NULL, &resp->resok.obj_attributes); vattr_to_wcc_data(dbvap, davap, &resp->resok.dir_wcc);
error ?
resp->resok.obj.handle_follows = FALSE;
resp->resok.obj.handle_follows = TRUE;
va.va_mask = AT_ALL; vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(vp, 0, cr, NULL); VN_RELE(vp); vattr_to_post_op_attr(vap, &resp->resok.obj_attributes); vattr_to_wcc_data(dbvap, davap, &resp->resok.dir_wcc);
out: name != NULL && name != args->where.name ?
kmem_free(name, MAXPATHLEN + 1);
symdata != NULL && symdata != args->symlink.symlink_data ?
kmem_free(symdata, MAXPATHLEN + 1);
DTRACE_NFSV3_4(op__symlink__done, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, SYMLINK3res *, resp);
dvp != NULL ?
VN_RELE(dvp);
End

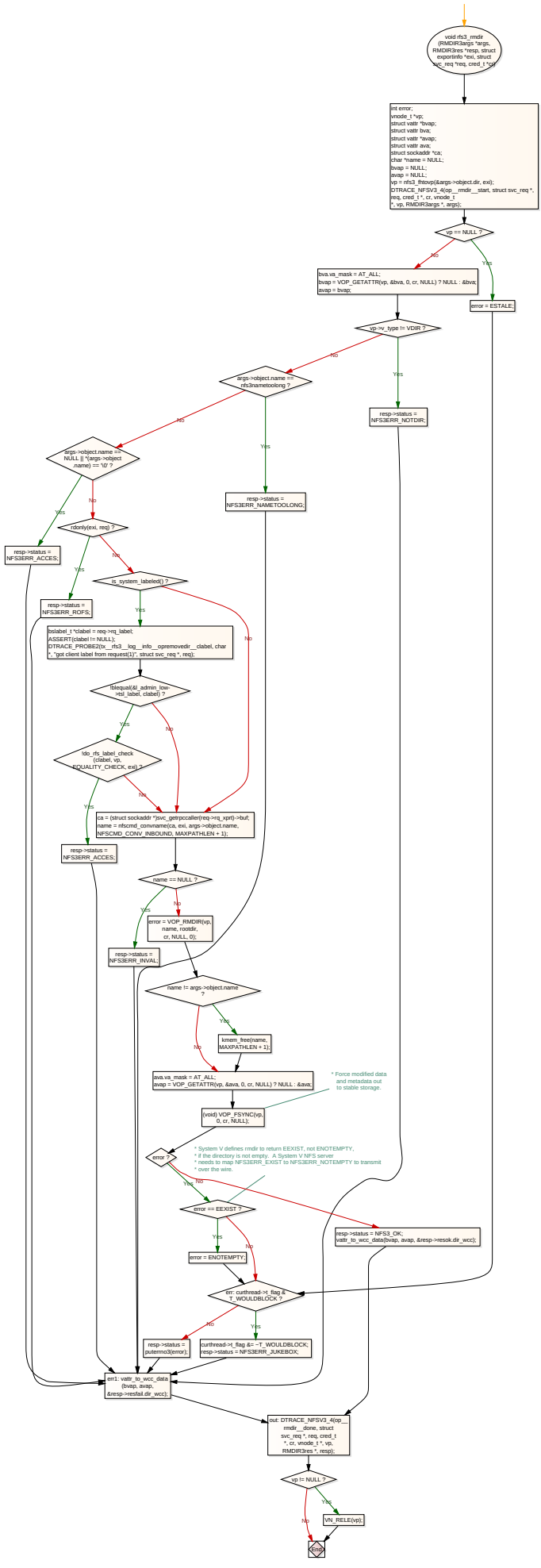

void * rfs3_symlink_getfh
(SYMLINK3args *args)
return (&args-
>where.dir);
End

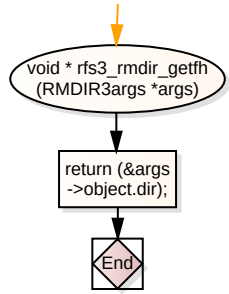

void rfs3_mknod (MKNOD3args *args, MKNOD3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp;
vnode_t *realvp;
vnode_t *dvp;
struct vattr *vap;
struct vattr va;
struct vattr *dbvap;
struct vattr dbva;
struct vattr *davap;
struct vattr dava;
int mode;
enum vcexcl excl;
struct sockaddr *ca;
char *name = NULL;
dbvap = NULL;
davap = NULL;
dvp = nfs3_fhtovp(&args->where.dir, exi);
DTRACE_NFSV3_4(op__mknod__start, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, MKNOD3args *, args);
dvp == NULL ?
error = ESTALE;
dbva.va_mask = AT_ALL;
dbvap = VOP_GETATTR(dvp, &dbva, 0, cr, NULL) ? NULL : &dbva;
davap = dbvap;
args->where.name == nfs3nametoolong ?
resp->status = NFS3ERR_NAMETOOLONG;
args->where.name == NULL || *(args->where.name) == '\0' ?
resp->status = NFS3ERR_ACCES;
rdonly(exi, req) ?
resp->status = NFS3ERR_ROFS;
is_system_labeled() ?
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opmknod__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
!do_rfs_label_check (clabel, dvp, EQUALITY_CHECK, exi) ?
resp->status = NFS3ERR_ACCES;
switch (args->what.type)
NF3CHR ?
NF3BLK ?
error = sattr3_to_vattr(&args->what.mknoddata3_u .device.dev_attributes, &va);
NF3SOCK ?
NF3FIFO ?
default
error ?
secpolicy_sys_devices(cr) != 0 ?
resp->status = NFS3ERR_PERM;
args->what.type == NF3CHR ?
va.va_type = VBLK;
va.va_type = VCHR;
va.va_rdev = makedevice(args->what.mknoddata3_u.device.spec.specdata1, args->what.mknoddata3_u.device.spec.specdata2);
va.va_mask |= AT_TYPE | AT_RDEV;
error = sattr3_to_vattr(&args->what.mknoddata3_u .pipe_attributes, &va);
error = sattr3_to_vattr(&args->what.mknoddata3_u .pipe_attributes, &va);
resp->status = NFS3ERR_BADTYPE;
error ?
error ?
va.va_type = VSOCK;
va.va_mask |= AT_TYPE;
va.va_type = VFIFO;
va.va_mask |= AT_TYPE;
* Must specify the mode.
!(va.va_mask & AT_MODE) ?
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, exi, args->where.name, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_INVAL;
name == NULL ?
excl = EXCL;
mode = 0;
error = VOP_CREATE(dvp, name, &va, excl, mode, &vp, cr, 0, NULL, NULL);
resp->status = NFS3ERR_INVAL;
name != args->where.name ?
kmem_free(name, MAXPATHLEN + 1);
* Force modified data and metadata out to stable storage.
dava.va_mask = AT_ALL;
davap = VOP_GETATTR(dvp, &dava, 0, cr, NULL) ? NULL : &dava;
(void) VOP_FSYNC(dvp, 0, cr, NULL);
error ?
resp->status = NFS3_OK;
error = makefh3(&resp->resok.obj.handle, vp, exi);
out: curthread->t_flag & T_WOULDBLOCK ?
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
resp->status = puterrno3(error);
error ?
resp->resok.obj.handle_ follows = FALSE;
resp->resok.obj.handle_ follows = TRUE;
out1: DTRACE_NFSV3_4(op__mknod__done, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, MKNOD3res *, resp);
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
* Force modified metadata out to stable storage.
*
* if a underlying vp exists, pass it to VOP_FSYNC
VOP_REALVP(vp, &realvp, NULL) == 0 ?
dvp != NULL ?
VN_RELE(dvp);
(void) VOP_FSYNC(realvp, FNODSYNC, cr, NULL);
(void) VOP_FSYNC(vp, FNODSYNC, cr, NULL);
vattr_to_wcc_data(dbvap, davap, &resp->resfail.dir_wcc);
VN_RELE(vp);
vattr_to_post_op_attr(vap, &resp->resok.obj_attributes);
vattr_to_wcc_data(dbvap, davap, &resp->resok.dir_wcc);
DTRACE_NFSV3_4(op__mknod__done, struct svc_req *, req, cred_t *, cr, vnode_t *, dvp, MKNOD3res *, resp);
VN_RELE(dvp);
End

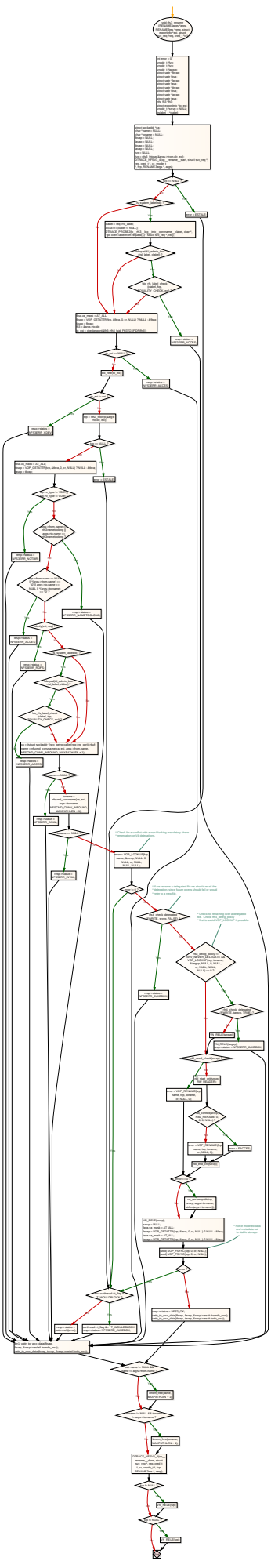

void * rfs3_mknod_getfh
(MKNOD3args *args)
return (&args-
>where.dir);
End

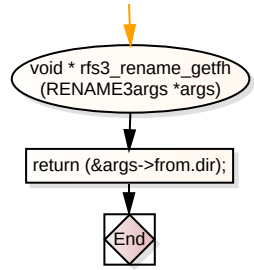

void rfs3_remove (REMOVE3args *args, REMOVE3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error = 0;
vnode_t *vp;
struct vattr *bvap;
struct vattr bva;
struct vattr *avap;
struct vattr ava;
vnode_t *targvp = NULL;
struct sockaddr *ca;
char *nama = NULL;
bvap = NULL;
avap = NULL;
vp = rfs3_fhtovp(&args->object.dir, exi);
DTRACE_NFSV3_4(cp__remove__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, REMOVE3args *, args);
vp == NULL ?
No
Yes
bva.va_mask = AT_ALL;
bvap = VOP_GETATTR(vp, &bva, 0, cr, NULL) ? NULL : &bva;
avap = bvap;
error = ESTALE;
vp->v_type != VDIR ?
No
Yes
args->object.name == rfs3nametoolong ?
No
Yes
resp->status = NFS3ERR_NOTDIR;
args->object.name == NULL || *(args->object.name) == '\0' ?
Yes
No
resp->status = NFS3ERR_NAMETOOLONG;
rdonly(exi, req) ?
No
Yes
resp->status = NFS3ERR_ACCES;
is_system_labeled() ?
Yes
No
resp->status = NFS3ERR_ROFS;
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opremove__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->sl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, vp, EQUALITY_CHECK, exi) ?
No
Yes
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, exi, args->object.name, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_ACCES;
name == NULL ?
No
Yes
* Check for a conflict with a non-blocking mandatory share
* reservation and V4 delegations
error = VOP_LOOKUP(vp, name, &targvp, NULL, 0, NULL, cr, NULL, NULL, NULL);
resp->status = NFS3ERR_INVAL;
error != 0 ?
No
Yes
rfs4_check_delegated (FWRITE, targvp, TRUE) ?
No
Yes
nbl_need_check(targvp) ?
No
Yes
resp->status = NFS3ERR_JUKEBOX;
nbl_start_crit(targvp, RW_READER);
error = VOP_REMOVE(vp, name, cr, NULL, 0);
nbl_conflict(targvp, NBL_REMOVE, 0, 0, 0, NULL) ?
No
Yes
error = VOP_REMOVE(vp, name, cr, NULL, 0);
error = EACCES;
nbl_end_crit(targvp);
VN_RELE(targvp);
targvp = NULL;
ava.va_mask = AT_ALL;
avap = VOP_GETATTR(vp, &ava, 0, cr, NULL) ? NULL : &ava;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(vp, 0, cr, NULL);
error ?
No
Yes
curthread->t_flag & T_WOULDBLOCK ?
No
Yes
resp->status = NFS3_OK;
vattr_to_wcc_data(bvap, avap, &resp->resok.dir_wcc);
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
err1: vattr_to_wcc_data (bvap, avap, &resp->resfail.dir_wcc);
out: DTRACE_NFSV3_4(cp__remove__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, REMOVE3res *, resp);
name != NULL && name != args->object.name ?
Yes
No
kmem_free(name, MAXPATHLEN + 1);
vp != NULL ?
Yes
No
VN_RELE(vp);
END

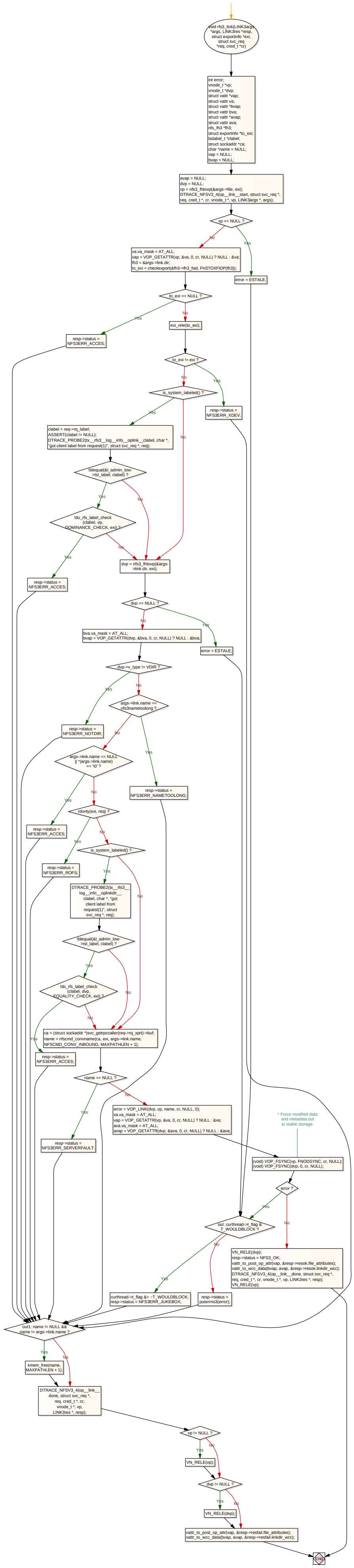

void * rfs3_remove_getfh
(REMOVE3args *args)
return (&args
->object.dir);
End


void rfs3_rmdir (RMDIR3args *args, RMDIR3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp;
struct vattr *bvap;
struct vattr bva;
struct vattr *avap;
struct vattr ava;
struct sockaddr *ca;
char *name = NULL;
bvap = NULL;
avap = NULL;
vp = nfs3_fhtovp(&args->object.dir, exi);
DTRACE_NFSV3_4(op__rmdir__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, RMDIR3args *, args);
vp == NULL ?
No
Yes
bva.va_mask = AT_ALL;
bvap = VOP_GETATTR(vp, &bva, 0, cr, NULL) ? NULL : &bva;
avap = bvap;
error = ESTALE;
vp->v_type != VDIR ?
No
Yes
args->object.name == NULL || *(args->object.name) == '\0' ?
No
Yes
resp->status = NFS3ERR_NOTDIR;
args->object.name == NFS3NAMETOOLONG ?
No
Yes
resp->status = NFS3ERR_ACCES;
rdonly(exi, req) ?
Yes
No
resp->status = NFS3ERR_NAMETOOLONG;
is_system_labeled() ?
Yes
No
resp->status = NFS3ERR_ROFS;
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opremovedir__clabel, char *, "got client label from request(1)", struct svc_req *, req);
blequal(&l_admin_low->tsl_label, clabel) ?
Yes
No
ldo_rfs_label_check (clabel, vp, EQUALITY_CHECK, exi) ?
Yes
No
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, exi, args->object.name, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_ACCES;
name == NULL ?
Yes
No
error = VOP_RMDIR(vp, name, rootdir, cr, NULL, 0);
resp->status = NFS3ERR_INVAL;
name != args->object.name ?
Yes
No
kmem_free(name, MAXPATHLEN + 1);
ava.va_mask = AT_ALL;
avap = VOP_GETATTR(vp, &ava, 0, cr, NULL) ? NULL : &ava;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(vp, 0, cr, NULL);
error ?
* System V defines rmdir to return EEXIST, not ENOTEMPTY,
* if the directory is not empty. A System V NFS server
* needs to map NFS3ERR_EXIST to NFS3ERR_NOTEMPTY to transmit
* over the wire.
Yes
No
error == EEXIST ?
Yes
No
resp->status = NFS3_OK;
vattr_to_wcc_data(bvap, avap, &resp->resok.dir_wcc);
error = ENOTEMPTY;
err: curthread->t_flag & T_WOULDBLOCK ?
No
Yes
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
err1: vattr_to_wcc_data(bvap, avap, &resp->resfail.dir_wcc);
out: DTRACE_NFSV3_4(op__rmdir__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, RMDIR3res *, resp);
vp != NULL ?
Yes
No
VN_RELE(vp);
End

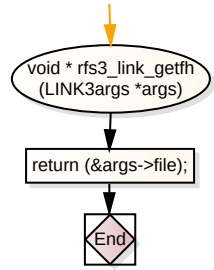

void * rfs3_rmdir_getfh
(RMDIR3args *args)
return (&args
->object.dir);
End

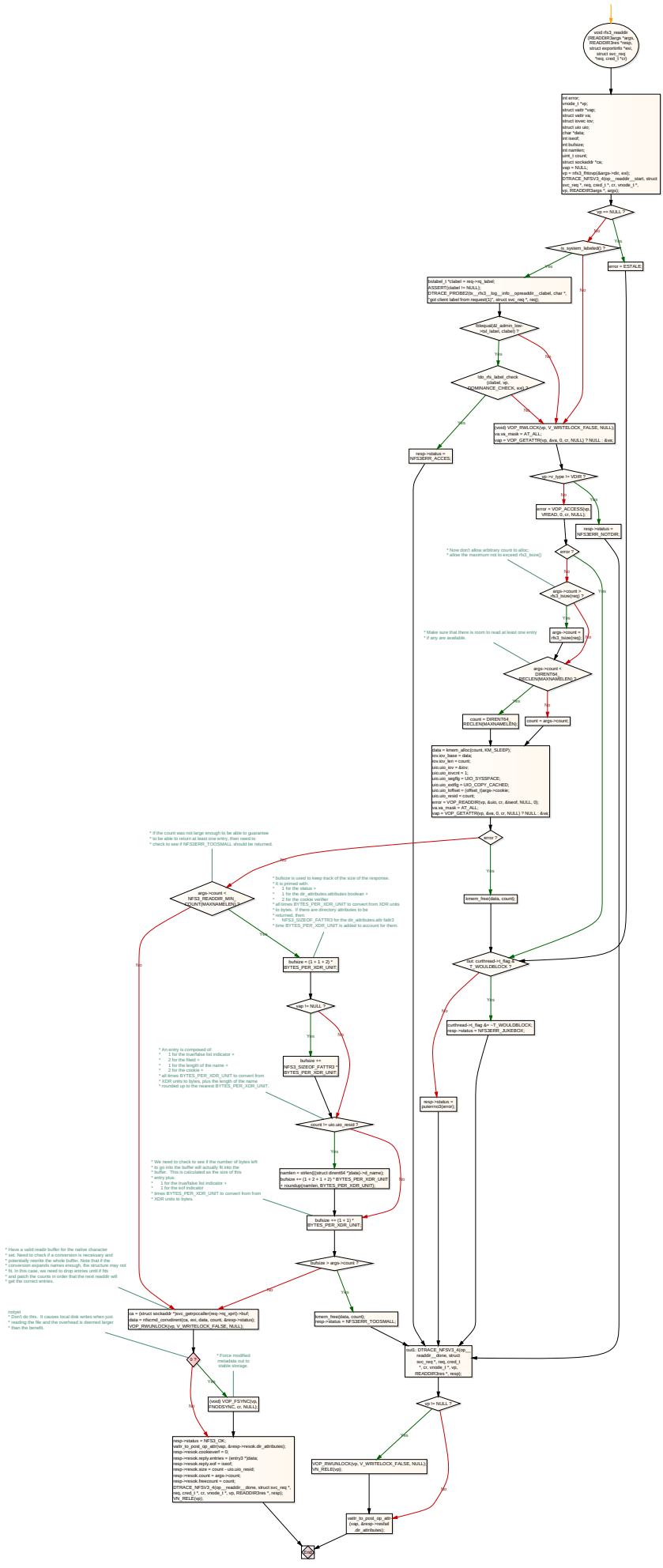


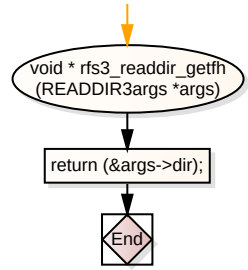

void * rfs3_rename_getfh
(RENAME3args *args)
return (&args->from.dir);
End

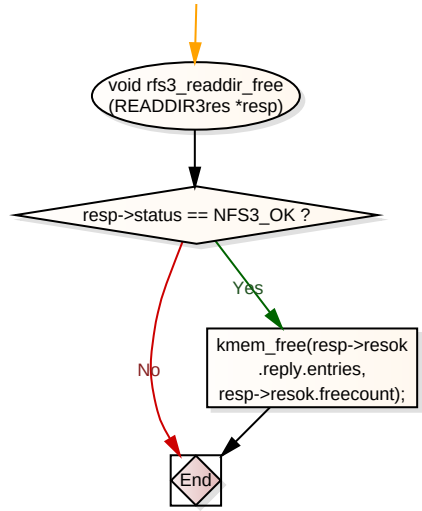


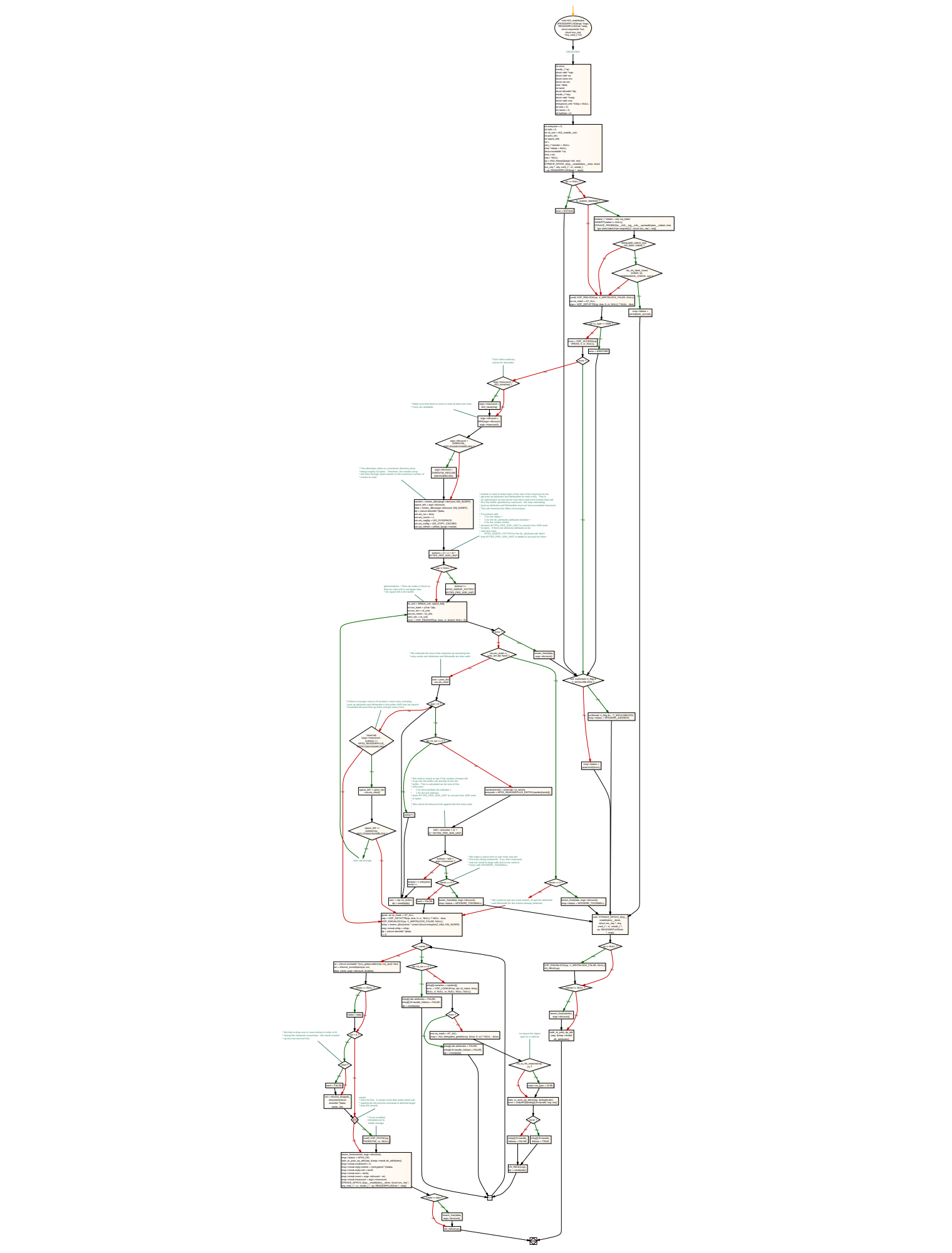
void nfs3_link(LINK3args *args, LINK3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp;
vnode_t *dvp;
struct vattr *vap;
struct vattr va;
struct vattr *bvap;
struct vattr bva;
struct vattr *avap;
struct vattr ava;
nfs_fh3 *fh3;
struct exportinfo *to_exi;
bslabel_t *clabel;
struct sockaddr *ca;
char *name = NULL;
vap = NULL;
bvap = NULL;
avap = NULL;
dvp = NULL;
vp = nfs3_fhtovp(&args->file, exi);
DTRACE_NFSV3_4(op__link__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, LINK3args *, args);
vp == NULL ?
No
Yes
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
fh3 = &args->link.dir;
to_exi = checkexport(&fh3->fh3_fsid, FH3TOXFIDP(fh3));
error = ESTALE;
to_exi == NULL ?
Yes
No
resp->status = NFS3ERR_ACCES;
exi_rele(to_exi);
to_exi != exi ?
No
Yes
is_system_labeled() ?
resp->status = NFS3ERR_XDEV;
Yes
No
clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__oplink__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi) ?
Yes
No
resp->status = NFS3ERR_ACCES;
dvp = nfs3_fhtovp(&args->link.dir, exi);
dvp == NULL ?
No
Yes
bva.va_mask = AT_ALL;
bvap = VOP_GETATTR(dvp, &bva, 0, cr, NULL) ? NULL : &bva;
error = ESTALE;
dvp->v_type != VDIR ?
Yes
No
args->link.name == nfs3nametoolong ?
resp->status = NFS3ERR_NOTDIR;
No
Yes
args->link.name == NULL || *(args->link.name) == '\0' ?
resp->status = NFS3ERR_NAMETOOLONG;
Yes
No
rdonly(exi, req) ?
resp->status = NFS3ERR_ACCES;
Yes
No
is_system_labeled() ?
resp->status = NFS3ERR_ROFS;
Yes
No
DTRACE_PROBE2(tx__rfs3__log__info__oplinkdir__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, dvp, EQUALITY_CHECK, exi) ?
Yes
No
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, exi, args->link.name, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
resp->status = NFS3ERR_ACCES;
name == NULL ?
No
Yes
error = VOP_LINK(dvp, vp, name, cr, NULL, 0);
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
ava.va_mask = AT_ALL;
avap = VOP_GETATTR(dvp, &ava, 0, cr, NULL) ? NULL : &ava;
resp->status = NFS3ERR_SERVERFAULT;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(vp, FNODSYNC, cr, NULL);
(void) VOP_FSYNC(dvp, 0, cr, NULL);
error ?
Yes
No
out: curthread->t_flag & T_WOULDBLOCK ?
No
Yes
VN_RELE(dvp);
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.file_attributes);
vattr_to_wcc_data(bvap, avap, &resp->resok.linkdir_wcc);
DTRACE_NFSV3_4(op__link__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, LINK3res *, resp);
VN_RELE(vp);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
resp->status = puterrno3(error);
out1: name != NULL && name != args->link.name ?
Yes
No
kmem_free(name, MAXPATHLEN + 1);
DTRACE_NFSV3_4(op__link__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, LINK3res *, resp);
vp != NULL ?
Yes
No
VN_RELE(vp);
dvp != NULL ?
Yes
No
VN_RELE(dvp);
vattr_to_post_op_attr(vap, &resp->resfail.file_attributes);
vattr_to_wcc_data(bvap, avap, &resp->resfail.linkdir_wcc);
End

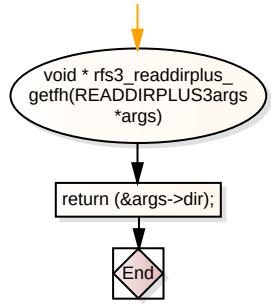

void * rfs3_link_getfh
(LINK3args *args)
return (&args->file);
End

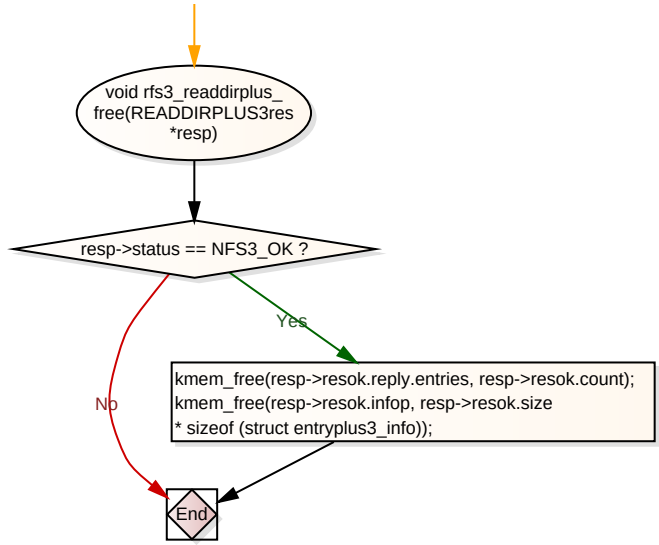

void rfs3_readdir(READDIR3args *args, READDIR3res *res, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp;
struct vattr *vap;
struct vattr va;
struct iovec iov;
struct uio uio;
char *data;
int iseof;
int bufsize;
int namlen;
uint_t count;
struct sockaddr *ca;
vap = NULL;
vp = rfs3_fhtovp(&args->dir, exi);
DTRACE_NFSV3_4(op__readdir__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, READDIR3args *, args);
vp == NULL ?
No
Yes
is_system_labeled() ?
Yes
No
error = ESTALE;
tsol_clabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opreaddir__clabel, char *, "get client label from request(1)", struct svc_req *, req);
blequal(&l_admin_low->tsl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi) ?
Yes
No
Yes
(void) VOP_RWLOCK(vp, V_WRITELOCK_FALSE, NULL);
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
resp->status = NFS3ERR_ACCES;
vp->v_type != VDIR ?
No
Yes
error = VOP_ACCESS(vp, VREAD, 0, cr, NULL);
resp->status = NFS3ERR_NOTDIR;
error ?
No
Yes
* Now don't allow arbitrary count to alloc;
* allow the maximum not to exceed rfs3_tsize()
args->count > rfs3_tsize(req) ?
Yes
No
args->count = rfs3_tsize(req);
* Make sure that there is room to read at least one entry
* if any are available.
args->count < DIRENT64_RECLEN(MAXNAMELEN) ?
Yes
No
count = DIRENT64_RECLEN(MAXNAMELEN);
count = args->count;
data = kmem_alloc(count, KM_SLEEP);
iov.iov_base = data;
iov.iov_len = count;
uio.uio_iov = &iov;
uio.uio_iovcnt = 1;
uio.uio_segflg = UIO_SYSSPACE;
uio.uio_extflg = UIO_COPY_CACHED;
uio.uio_loffset = (offset_t)args->cookie;
uio.uio_resid = count;
error = VOP_READDIR(vp, &uio, cr, &iseof, NULL, 0);
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
error ?
* If the count was not large enough to be able to guarantee
* to be able to return at least one entry, then need to
* check to see if NFS3ERR_TOOSMALL should be returned.
No
Yes
kmem_free(data, count);
* bufsize is used to keep track of the size of the response.
* It is primed with:
* 1 for the status +
* 1 for the dir_attributes.attributes boolean +
* 2 for the cookie verifier
* all times BYTES_PER_XDR_UNIT to convert from XDR units
* to bytes. If there are directory attributes to be
* returned, then:
* NFS3_SIZEOF_FATTR3 for the dir_attributes.attr fattr3
* time BYTES_PER_XDR_UNIT is added to account for them.
args->count < NFS3_READDIR_MIN_COUNT/MAXNAMELEN) ?
No
Yes
bufsize = (1 + 1 + 2) * BYTES_PER_XDR_UNIT;
* An entry is composed of:
* 1 for the truefalse list indicator +
* 2 for the fileid +
* 1 for the length of the name +
* 2 for the cookie +
* all times BYTES_PER_XDR_UNIT to convert from
* XDR units to bytes, plus the length of the name
* rounded up to the nearest BYTES_PER_XDR_UNIT.
vap != NULL ?
Yes
No
bufsize += NFS3_SIZEOF_FATTR3 * BYTES_PER_XDR_UNIT;
count != uio.uio_resid ?
Yes
* We need to check to see if the number of bytes left
* to go into the buffer will actually fit into the
* buffer. This is calculated as the size of this
* entry plus
* 1 for the truefalse list indicator +
* 1 for the eof indicator
* times BYTES_PER_XDR_UNIT to convert from from
* XDR units to bytes.
namlen = strlen((struct dirent64 *)data)->d_name);
bufsize += (1 + 2 + 1 + 2) * BYTES_PER_XDR_UNIT + roundup(namlen, BYTES_PER_XDR_UNIT);
No
bufsize += (1 + 1) * BYTES_PER_XDR_UNIT;
bufsize > args->count ?
No
Yes
* Have a valid readir buffer for the native character
* set. Need to check if a conversion is necessary and
* potentially reentre the whole buffer. Note that if the
* conversion expands names enough, the structure may not
* fit. In this case, we need to drop entries until it fits
* and patch the counts in order that the next readdir will
* get the correct entries.
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
data = nfscmd_convdirent(ca, exi, data, count, &resp->status);
VOP_RWUNLOCK(vp, V_WRITELOCK_FALSE, NULL);
kmem_free(data, count);
resp->status = NFS3ERR_TOOSMALL;
notyet
* Don't do this. It causes local disk writes when just
* reading the file and the overhead is deemed larger
* than the benefit.
* 0 ?
Yes
No
* Force modified metadata out to stable storage.
(void) VOP_FSYNC(vp, FNODSYNC, cr, NULL);
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.dir_attributes);
resp->resok.cookieverf = 0;
resp->resok.reply.entries = (entry3 *)data;
resp->resok.reply.eof = iseof;
resp->resok.size = count - uio.uio_resid;
resp->resok.count = args->count;
resp->resok.freecount = count;
DTRACE_NFSV3_4(op__readdir__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, READDIR3res *, resp);
VN_RELE(vp);
curthread->t_flag & T_WOULDBLOCK ?
No
Yes
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
resp->status = puterrno3(error);
out: DTRACE_NFSV3_4(op__readdir__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, READDIR3res *, resp);
vp != NULL ?
Yes
No
VOP_RWUNLOCK(vp, V_WRITELOCK_FALSE, NULL);
VN_RELE(vp);
vattr_to_post_op_attr(vap, &resp->resfail.dir_attributes);

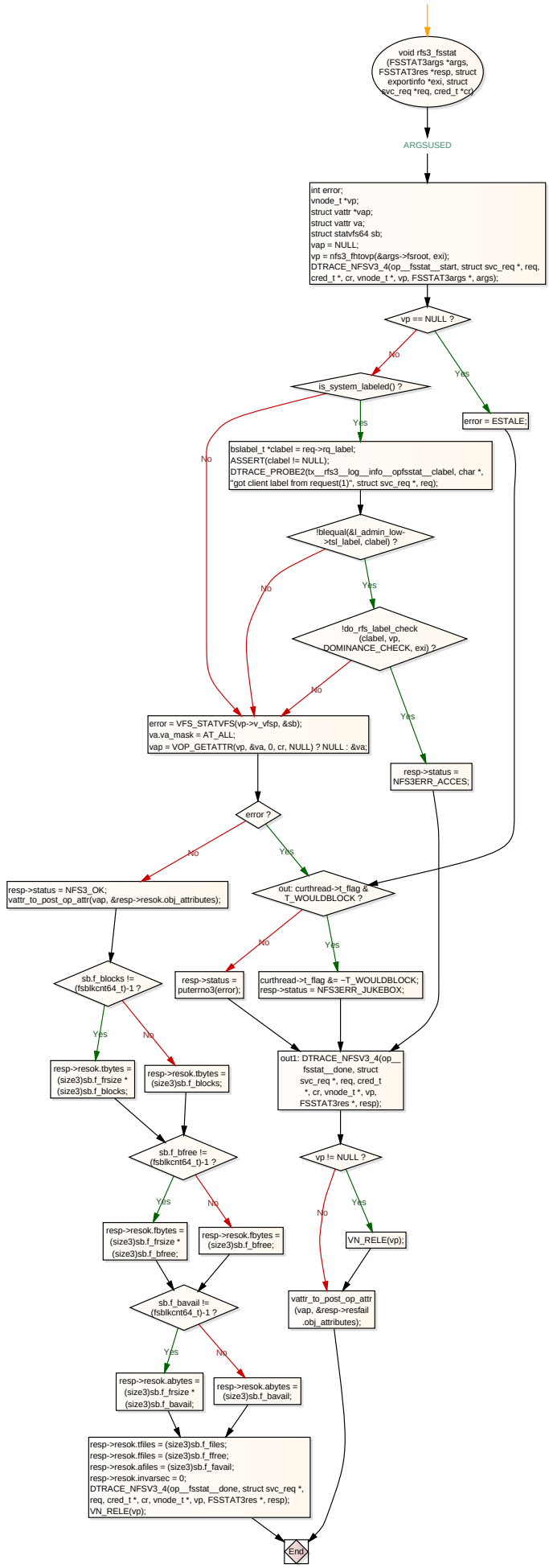

void * rfs3_readdir_getfh
(READDIR3args *args)
return (&args->dir);
End

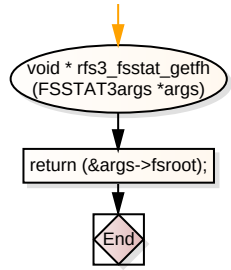

void rfs3_readdir_free
(READDIR3res *resp)
resp->status == NFS3_OK ?
Yes
No
kmem_free(resp->resok
.reply.entries,
resp->resok.freecount);
End

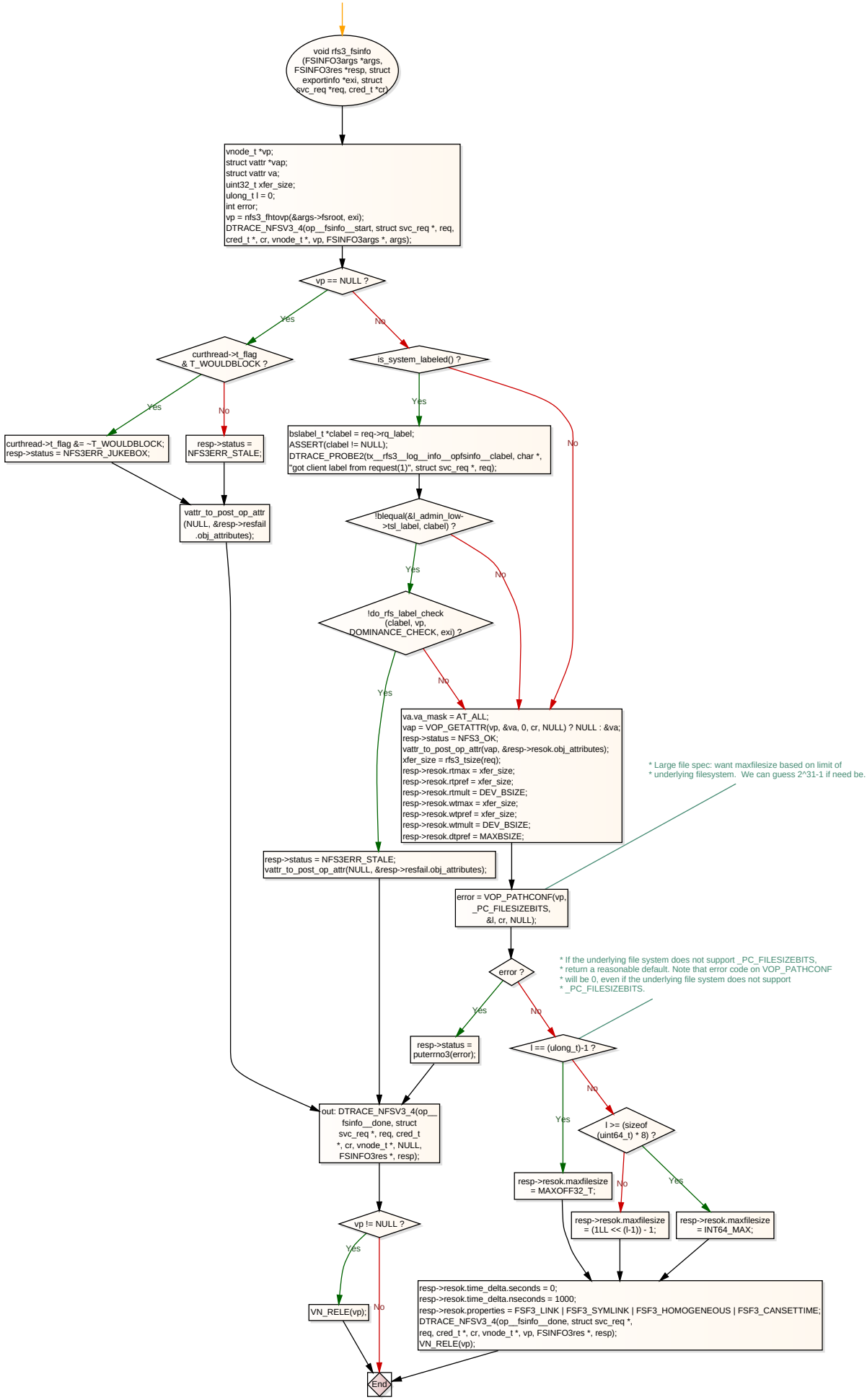


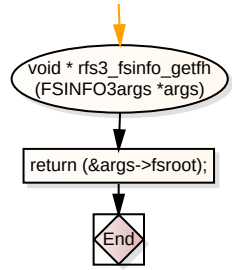

void * rfs3_readdirplus_
getfh(READDIRPLUS3args
*args)
return (&args->dir);
End

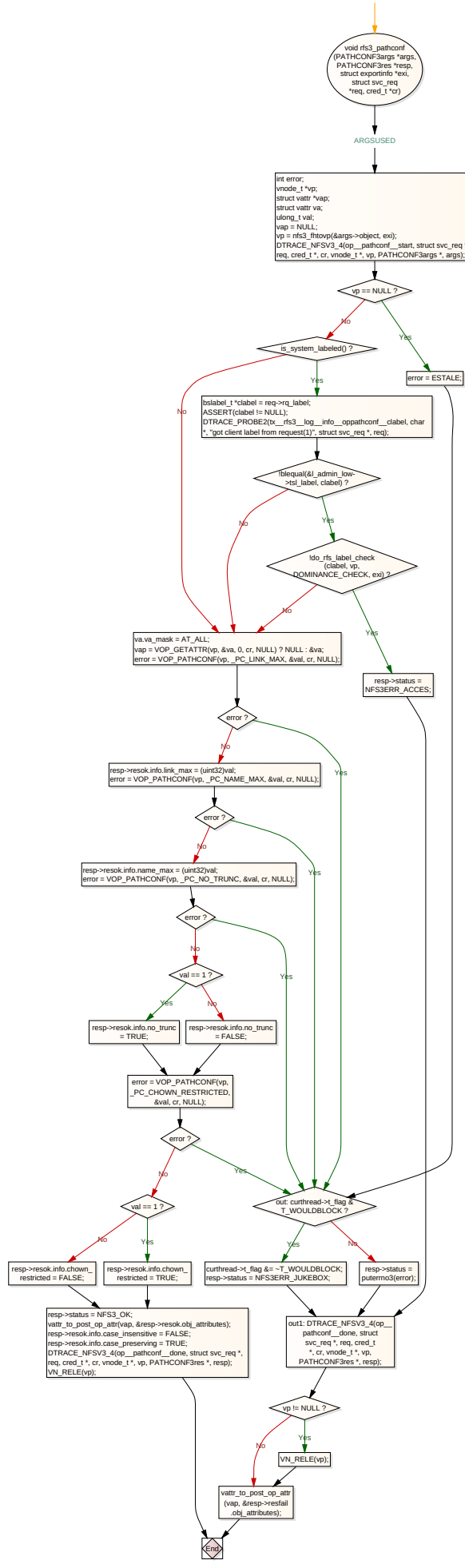

void rfs3_readdirplus_
free(READDIRPLUS3res
*resp)
resp->status == NFS3_OK ?
Yes
No
kmem_free(resp->resok.reply.entries, resp->resok.count);
kmem_free(resp->resok.infop, resp->resok.size
* sizeof (struct entryplus3_info));
End

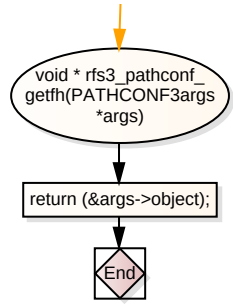

void rfs3_fsstat (FSSTAT3args *args, FSSTAT3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
ARGSUSED
int error;
vnode_t *vp;
struct vattr *vap;
struct vattr va;
struct statvfs64 sb;
vap = NULL;
vp = nfs3_fhtovp(&args->fsroot, exi);
DTRACE_NFSV3_4(op__fsstat__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, FSSTAT3args *, args);
vp == NULL ?
No
Yes
is_system_labeled() ?
No
Yes
error = ESTALE;
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opfsstat__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
No
Yes
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi) ?
No
Yes
error = VFS_STATVFS(vp->v_vfsp, &sb);
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
resp->status = NFS3ERR_ACCES;
error ?
No
Yes
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.obj_attributes);
out: curthread->t_flag & T_WOULDBLOCK ?
No
Yes
resp->status = puterrno3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
sb.f_blocks != (fsblkcnt64_t)-1 ?
Yes
No
resp->resok.tbytes = (size3)sb.f_frsize * (size3)sb.f_blocks;
resp->resok.tbytes = (size3)sb.f_blocks;
out1: DTRACE_NFSV3_4(op__fsstat__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, FSSTAT3res *, resp);
sb.f_bfree != (fsblkcnt64_t)-1 ?
Yes
No
resp->resok.fbytes = (size3)sb.f_frsize * (size3)sb.f_bfree;
resp->resok.fbytes = (size3)sb.f_bfree;
vp != NULL ?
No
Yes
VN_RELE(vp);
vattr_to_post_op_attr (vap, &resp->resfail .obj_attributes);
sb.f_bavail != (fsblkcnt64_t)-1 ?
Yes
No
resp->resok.abytes = (size3)sb.f_frsize * (size3)sb.f_bavail;
resp->resok.abytes = (size3)sb.f_bavail;
resp->resok.tfiles = (size3)sb.f_files;
resp->resok.ffiles = (size3)sb.f_ffree;
resp->resok.afiles = (size3)sb.f_favail;
resp->resok.invarsec = 0;
DTRACE_NFSV3_4(op__fsstat__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, FSSTAT3res *, resp);
VN_RELE(vp);
End

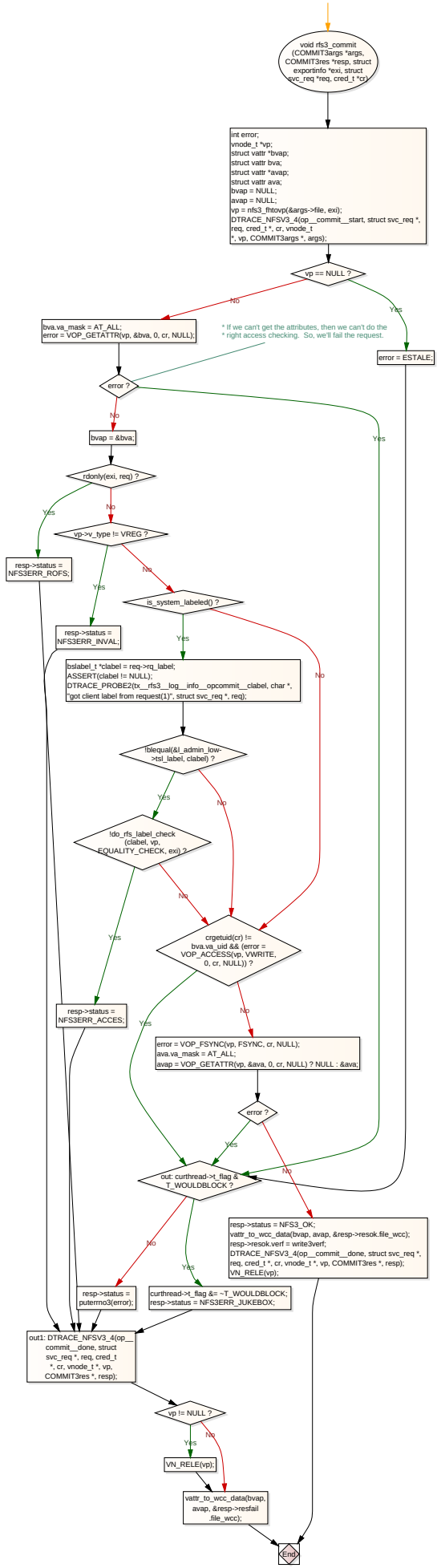

void * rfs3_fsstat_getfh
(FSSTAT3args *args)
return (&args->fsroot);
End

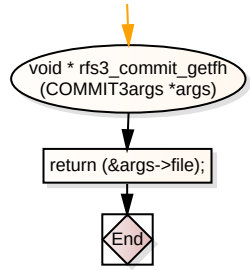

void rfs3_fsinfo (FSINFO3args *args, FSINFO3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
vnode_t *vp;
struct vattr *vap;
struct vattr va;
uint32_t xfer_size;
ulong_t l = 0;
int error;
vp = nfs3_fhtovp(&args->fsroot, exi);
DTRACE_NFSV3_4(op__fsinfo__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, FSINFO3args *, args);
vp == NULL ?
Yes
No
curthread->t_flag & T_WOULDBLOCK ?
is_system_labeled() ?
Yes
No
Yes
No
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
resp->status = NFS3ERR_STALE;
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opfsinfo__clabel, char *, "got client label from request(1)", struct svc_req *, req);
vattr_to_post_op_attr (NULL, &resp->resfail .obj_attributes);
!blequal(&l_admin_low- >tsl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi) ?
Yes
No
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.obj_attributes);
xfer_size = rfs3_tsize(req);
resp->resok.rtmax = xfer_size;
resp->resok.rtpref = xfer_size;
resp->resok.rtmult = DEV_BSIZE;
resp->resok.wtmax = xfer_size;
resp->resok.wtpref = xfer_size;
resp->resok.wtmult = DEV_BSIZE;
resp->resok.dtpref = MAXBSIZE;
* Large file spec: want maxfilesize based on limit of
* underlying filesystem. We can guess 2^31-1 if need be.
resp->status = NFS3ERR_STALE;
vattr_to_post_op_attr(NULL, &resp->resfail.obj_attributes);
error = VOP_PATHCONF(vp, _PC_FILESIZEBITS, &l, cr, NULL);
error ?
* If the underlying file system does not support _PC_FILESIZEBITS,
* return a reasonable default. Note that error code on VOP_PATHCONF
* will be 0, even if the underlying file system does not support
* _PC_FILESIZEBITS.
Yes
No
resp->status = puterrno3(error);
l == (ulong_t)-1 ?
No
Yes
l >= (sizeof (uint64_t) * 8) ?
resp->resok.maxfilesize = MAXOFF32_T;
No
Yes
resp->resok.maxfilesize = (1LL << (l-1)) - 1;
resp->resok.maxfilesize = INT64_MAX;
out: DTRACE_NFSV3_4(op__ fsinfo__done, struct svc_req *, req, cred_t *, cr, vnode_t *, NULL, FSINFO3res *, resp);
vp != NULL ?
Yes
No
VN_RELE(vp);
resp->resok.time_delta.seconds = 0;
resp->resok.time_delta.nseconds = 1000;
resp->resok.properties = FSF3_LINK | FSF3_SYMLINK | FSF3_HOMOGENEOUS | FSF3_CANSETTIME;
DTRACE_NFSV3_4(op__fsinfo__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, FSINFO3res *, resp);
VN_RELE(vp);
End

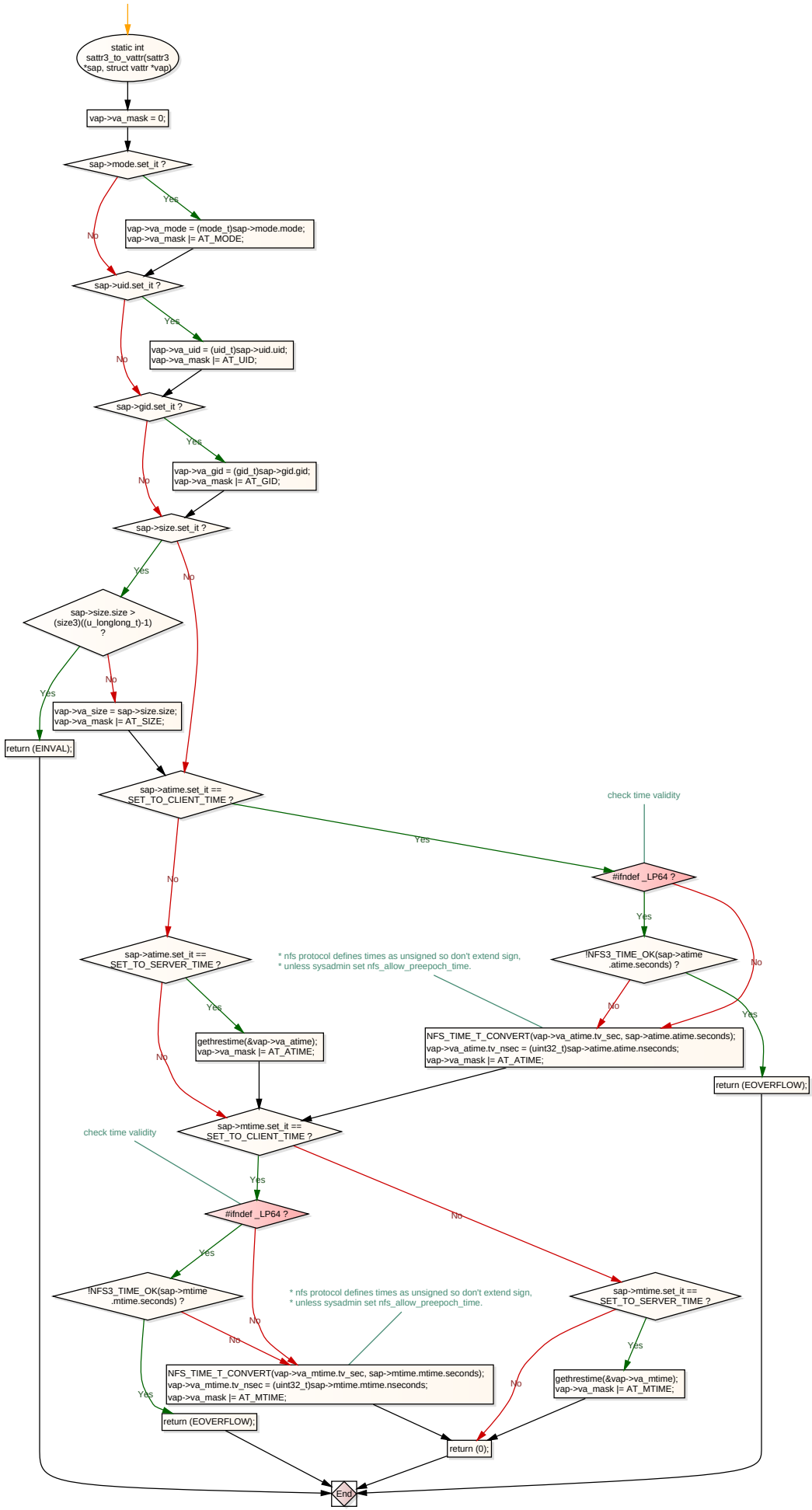

void * rfs3_fsinfo_getfh
(FSINFO3args *args)
return (&args->fsroot);
End

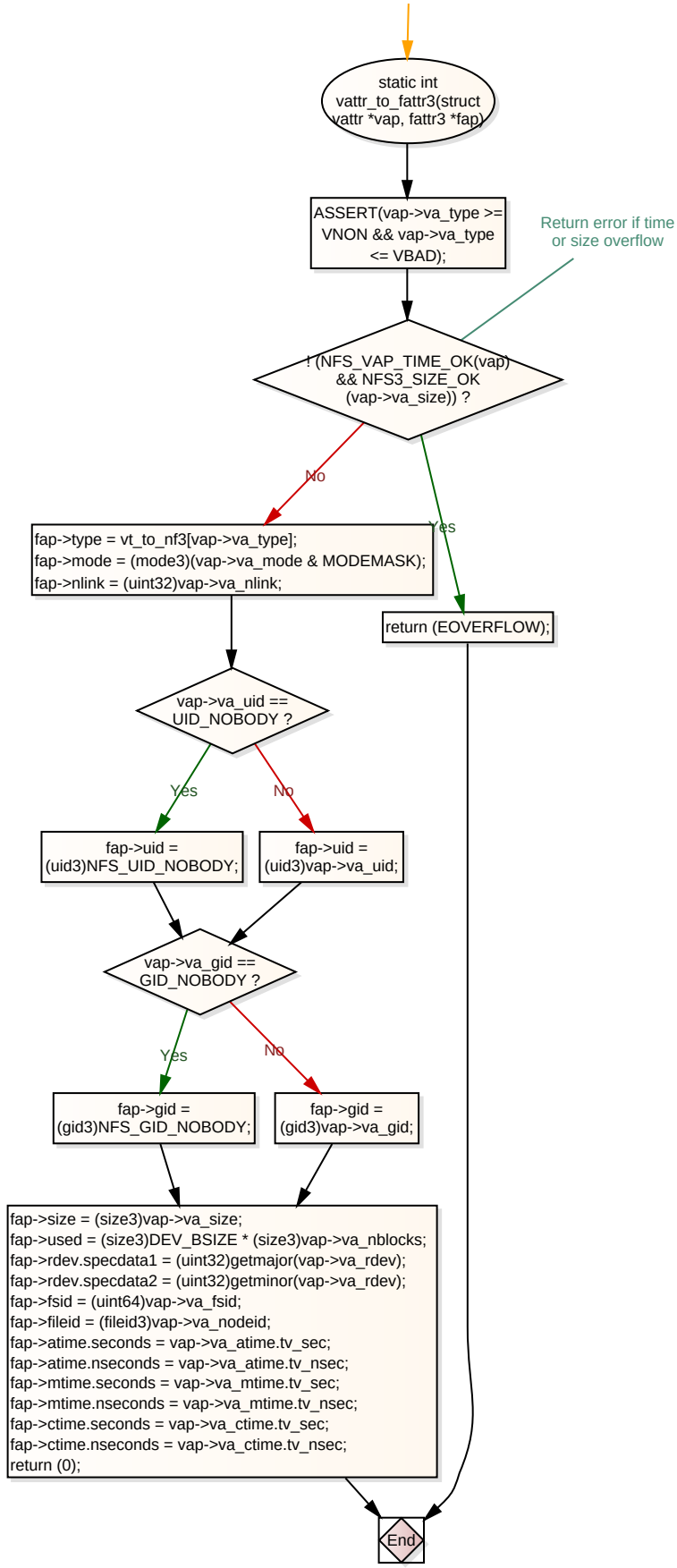

void rfs3_pathconf (PATHCONF3args *args, PATHCONF3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
ARGSUSED
int error;
vnode_t *vp;
struct vattr *vap;
struct vattr va;
ulong_t val;
vap = NULL;
vp = nfs3_fhtovp(&args->object, exi);
DTRACE_NFSV3_4(op__pathconf__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, PATHCONF3args *, args);
vp == NULL ?
No
Yes
is_system_labeled() ?
No
Yes
error = ESTALE;
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__oppathconf__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
No
Yes
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, exi) ?
No
Yes
va.va_mask = AT_ALL;
vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
error = VOP_PATHCONF(vp, _PC_LINK_MAX, &val, cr, NULL);
resp->status = NFS3ERR_ACCES;
error ?
No
Yes
resp->resok.info.link_max = (uint32)val;
error = VOP_PATHCONF(vp, _PC_NAME_MAX, &val, cr, NULL);
error ?
No
Yes
resp->resok.info.name_max = (uint32)val;
error = VOP_PATHCONF(vp, _PC_NO_TRUNC, &val, cr, NULL);
error ?
No
Yes
val == 1 ?
Yes
No
resp->resok.info.no_trunc = TRUE;
resp->resok.info.no_trunc = FALSE;
error = VOP_PATHCONF(vp, _PC_CHOWN_RESTRICTED, &val, cr, NULL);
error ?
No
Yes
val == 1 ?
No
Yes
out: curthread->t_flag & T_WOULDBLOCK ?
Yes
No
resp->resok.info.chown_ restricted = FALSE;
resp->resok.info.chown_ restricted = TRUE;
curthread->t_flag &= ~T_WOULDBLOCK; resp->status = NFS3ERR_JUKEBOX;
resp->status = puterrno3(error);
resp->status = NFS3_OK;
vattr_to_post_op_attr(vap, &resp->resok.obj_attributes);
resp->resok.info.case_insensitive = FALSE;
resp->resok.info.case_preserving = TRUE;
DTRACE_NFSV3_4(op__pathconf__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, PATHCONF3res *, resp);
VN_RELE(vp);
out1: DTRACE_NFSV3_4(op__ pathconf__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, PATHCONF3res *, resp);
vp != NULL ?
No
Yes
VN_RELE(vp);
vattr_to_post_op_attr (vap, &resp->resfail .obj_attributes);
End

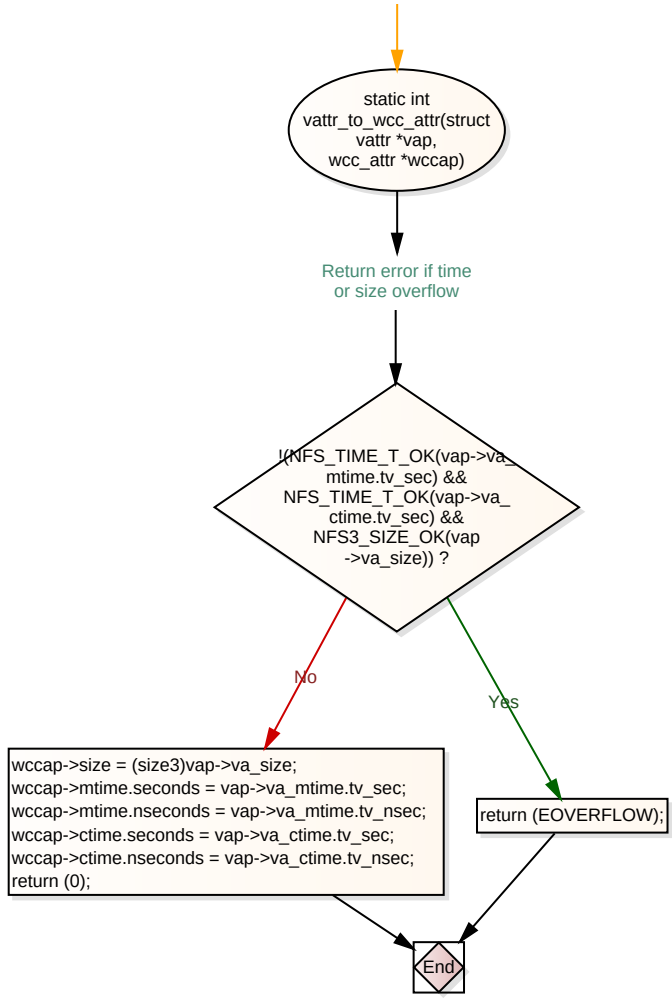

void * rfs3_pathconf_
getfh(PATHCONF3args
*args)
return (&args->object);
End

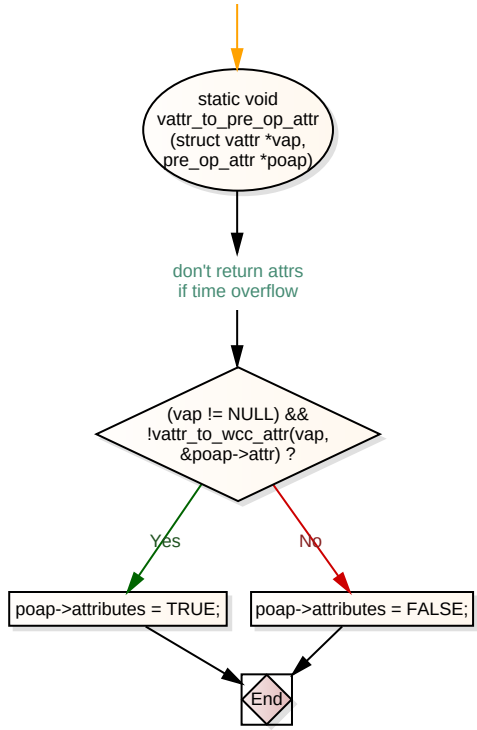

void rfs3_commit (COMMIT3args *args, COMMIT3res *resp, struct exportinfo *exi, struct svc_req *req, cred_t *cr)
int error;
vnode_t *vp;
struct vattr *bvap;
struct vattr bva;
struct vattr *avap;
struct vattr ava;
bvap = NULL;
avap = NULL;
vp = nfs3_fhtovp(&args->file, exi);
DTRACE_NFSV3_4(op__commit__start, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, COMMIT3args *, args);
vp == NULL ?
No
Yes
bva.va_mask = AT_ALL;
error = VOP_GETATTR(vp, &bva, 0, cr, NULL);
* If we can't get the attributes, then we can't do the
* right access checking. So, we'll fail the request.
error = ESTALE;
error ?
No
Yes
bvap = &bva;
rdonly(exi, req) ?
Yes
No
vp->v_type != VREG ?
resp->status = NFS3ERR_ROFS;
Yes
No
resp->status = NFS3ERR_INVAL;
Is_system_labeled() ?
Yes
No
bslabel_t *clabel = req->rq_label;
ASSERT(clabel != NULL);
DTRACE_PROBE2(tx__rfs3__log__info__opcommit__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
Yes
No
!do_rfs_label_check (clabel, vp, EQUALITY_CHECK, exi) ?
Yes
No
crgetuid(cr) != bva.va_uid && (error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL)) ?
Yes
No
resp->status = NFS3ERR_ACCES;
error = VOP_FSYNC(vp, FSYNC, cr, NULL);
ava.va_mask = AT_ALL;
avap = VOP_GETATTR(vp, &ava, 0, cr, NULL) ? NULL : &ava;
error ?
Yes
No
out: curthread->t_flag & T_WOULDBLOCK ?
No
Yes
resp->status = NFS3_OK;
vattr_to_wcc_data(bvap, avap, &resp->resok.file_wcc);
resp->resok.verf = write3verf;
DTRACE_NFSV3_4(op__commit__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, COMMIT3res *, resp);
VN_RELE(vp);
resp->status = putermo3(error);
curthread->t_flag &= ~T_WOULDBLOCK;
resp->status = NFS3ERR_JUKEBOX;
out1: DTRACE_NFSV3_4(op__commit__done, struct svc_req *, req, cred_t *, cr, vnode_t *, vp, COMMIT3res *, resp);
vp != NULL ?
Yes
No
VN_RELE(vp);
vattr_to_wcc_data(bvap, avap, &resp->resfail.file_wcc);
End

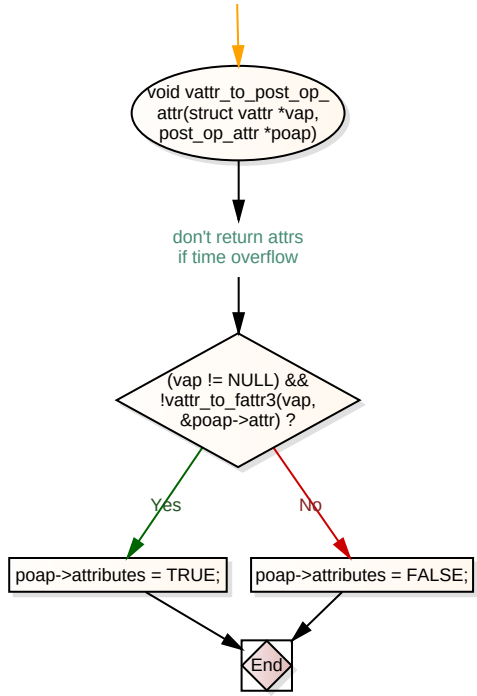

void * rfs3_commit_getfh
(COMMIT3args *args)
return (&args->file);
End

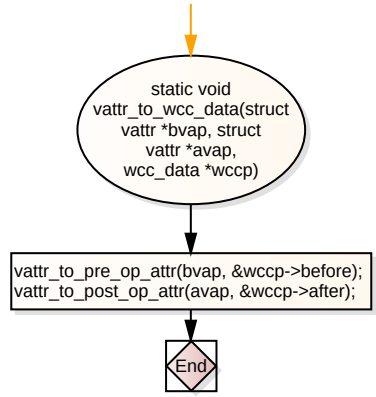

static int
sattr3_to_vattr(sattr3
*sap, struct vattr *vap)
vap->va_mask = 0;
sap->mode.set_it ?
Yes
No
vap->va_mode = (mode_t)sap->mode.mode;
vap->va_mask |= AT_MODE;
sap->uid.set_it ?
Yes
No
vap->va_uid = (uid_t)sap->uid.uid;
vap->va_mask |= AT_UID;
sap->gid.set_it ?
Yes
No
vap->va_gid = (gid_t)sap->gid.gid;
vap->va_mask |= AT_GID;
sap->size.set_it ?
Yes
No
sap->size.size >
(size3)((u_longlong_t)-1)
?
Yes
No
vap->va_size = sap->size.size;
vap->va_mask |= AT_SIZE;
return (EINVAL);
sap->atime.set_it ==
SET_TO_CLIENT_TIME ?
Yes
No
check time validity
#ifndef _LP64 ?
Yes
No
!NFS3_TIME_OK(sap->atime
.atime.seconds) ?
No
Yes
* nfs protocol defines times as unsigned so don't extend sign,
* unless sysadmin set nfs_allow_preepoch_time.
NFS_TIME_T_CONVERT(vap->va_atime.tv_sec, sap->atime.atime.seconds);
vap->va_atime.tv_nsec = (uint32_t)sap->atime.atime.nseconds;
vap->va_mask |= AT_ATIME;
return (EOVERFLOW);
sap->atime.set_it ==
SET_TO_SERVER_TIME ?
Yes
No
gethrestime(&vap->va_atime);
vap->va_mask |= AT_ATIME;
sap->mtime.set_it ==
SET_TO_CLIENT_TIME ?
check time validity
Yes
No
#ifndef _LP64 ?
Yes
No
!NFS3_TIME_OK(sap->mtime
.mtime.seconds) ?
No
Yes
* nfs protocol defines times as unsigned so don't extend sign,
* unless sysadmin set nfs_allow_preepoch_time.
NFS_TIME_T_CONVERT(vap->va_mtime.tv_sec, sap->mtime.mtime.seconds);
vap->va_mtime.tv_nsec = (uint32_t)sap->mtime.mtime.nseconds;
vap->va_mask |= AT_MTIME;
return (EOVERFLOW);
sap->mtime.set_it ==
SET_TO_SERVER_TIME ?
Yes
No
gethrestime(&vap->va_mtime);
vap->va_mask |= AT_MTIME;
return (0);
End

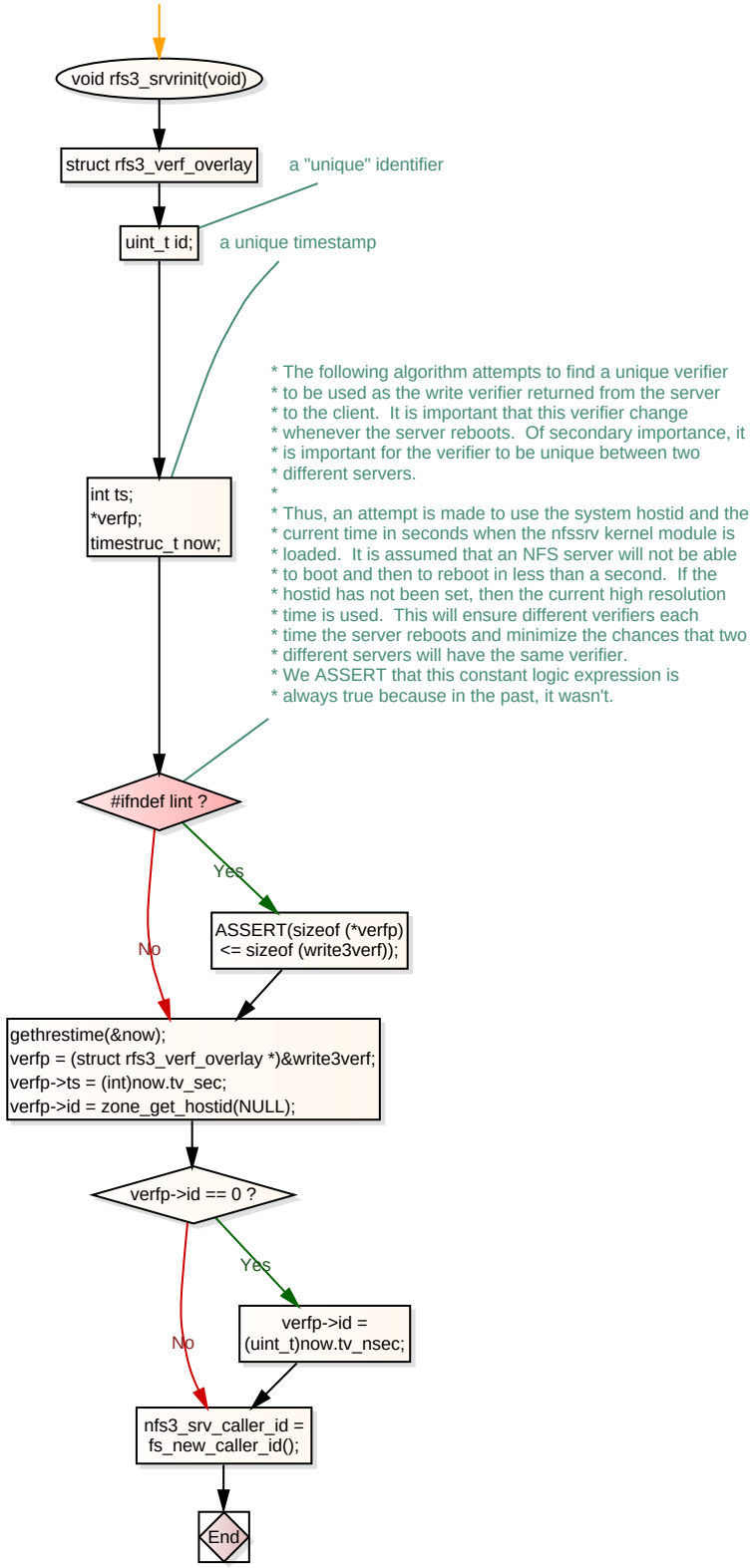

static int vattr_to_fattr3(struct vattr *vap, fattr3 *fap)
ASSERT(vap->va_type >= VNON && vap->va_type <= VBAD);
Return error if time or size overflow
! (NFS_VAP_TIME_OK(vap) && NFS3_SIZE_OK (vap->va_size)) ?
No
Yes
fap->type = vt_to_nf3[vap->va_type];
fap->mode = (mode3)(vap->va_mode & MODEMASK);
fap->nlink = (uint32)vap->va_nlink;
return (EOVERFLOW);
vap->va_uid == UID_NOBODY ?
Yes
No
fap->uid = (uid3)NFS_UID_NOBODY;
fap->uid = (uid3)vap->va_uid;
vap->va_gid == GID_NOBODY ?
Yes
No
fap->gid = (gid3)NFS_GID_NOBODY;
fap->gid = (gid3)vap->va_gid;
fap->size = (size3)vap->va_size;
fap->used = (size3)DEV_BSIZE * (size3)vap->va_nblocks;
fap->rdev.specdata1 = (uint32)getmajor(vap->va_rdev);
fap->rdev.specdata2 = (uint32)getminor(vap->va_rdev);
fap->fsid = (uint64)vap->va_fsid;
fap->fileid = (fileid3)vap->va_nodeid;
fap->atime.seconds = vap->va_atime.tv_sec;
fap->atime.nseconds = vap->va_atime.tv_nsec;
fap->mtime.seconds = vap->va_mtime.tv_sec;
fap->mtime.nseconds = vap->va_mtime.tv_nsec;
fap->ctime.seconds = vap->va_ctime.tv_sec;
fap->ctime.nseconds = vap->va_ctime.tv_nsec;
return (0);
End

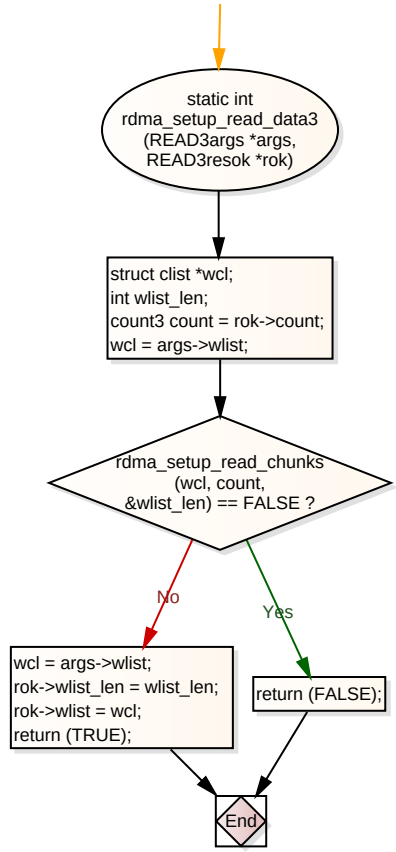

static int
vattr_to_wcc_attr(struct
vattr *vap,
wcc_attr *wccap)
Return error if time
or size overflow
!(NFS_TIME_T_OK(vap->va_
mtime.tv_sec) &&
NFS_TIME_T_OK(vap->va_
ctime.tv_sec) &&
NFS3_SIZE_OK(vap
->va_size)) ?
No
Yes
wccap->size = (size3)vap->va_size;
wccap->mtime.seconds = vap->va_mtime.tv_sec;
wccap->mtime.nseconds = vap->va_mtime.tv_nsec;
wccap->ctime.seconds = vap->va_ctime.tv_sec;
wccap->ctime.nseconds = vap->va_ctime.tv_nsec;
return (0);
return (EOVERFLOW);
End

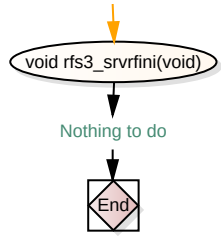

static void
vattr_to_pre_op_attr
(struct vattr *vap,
pre_op_attr *poap)
don't return attrs
if time overflow
(vap != NULL) &&
!vattr_to_wcc_attr(vap,
&poap->attr) ?
Yes
No
poap->attributes = TRUE;
poap->attributes = FALSE;
End


void vattr_to_post_op_
attr(struct vattr *vap,
post_op_attr *poap)
don't return attrs
if time overflow
(vap != NULL) &&
!vattr_to_fattr3(vap,
&poap->attr) ?
Yes
No
poap->attributes = TRUE;
poap->attributes = FALSE;
End


static void
vattr_to_wcc_data(struct
vattr *bvap, struct
vattr *avap,
wcc_data *wccp)
vattr_to_pre_op_attr(bvap, &wccp->before);
vattr_to_post_op_attr(avap, &wccp->after);
End


void rfs3_srvrinit(void)
struct rfs3_verf_overlay
a "unique" identifier
uint_t id;
a unique timestamp
* The following algorithm attempts to find a unique verifier
* to be used as the write verifier returned from the server
* to the client. It is important that this verifier change
* whenever the server reboots. Of secondary importance, it
* is important for the verifier to be unique between two
* different servers.
*
* Thus, an attempt is made to use the system hostid and the
* current time in seconds when the nfssrv kernel module is
* loaded. It is assumed that an NFS server will not be able
* to boot and then to reboot in less than a second. If the
* hostid has not been set, then the current high resolution
* time is used. This will ensure different verifiers each
* time the server reboots and minimize the chances that two
* different servers will have the same verifier.
* We ASSERT that this constant logic expression is
* always true because in the past, it wasn't.
int ts;
*verfp;
timestruc_t now;
#ifndef lint ?
Yes
No
ASSERT(sizeof (*verfp)
<= sizeof (write3verf));
gethrestime(&now);
verfp = (struct rfs3_verf_overlay *)&write3verf;
verfp->ts = (int)now.tv_sec;
verfp->id = zone_get_hostid(NULL);
verfp->id == 0 ?
Yes
No
verfp->id =
(uint_t)now.tv_nsec;
nfs3_srv_caller_id =
fs_new_caller_id();
End


static int
rdma_setup_read_data3
(READ3args *args,
READ3resok *rok)
struct clist *wcl;
int wlist_len;
count3 count = rok->count;
wcl = args->wlist;
rdma_setup_read_chunks
(wcl, count,
&wlist_len) == FALSE ?
No
Yes
wcl = args->wlist;
rok->wlist_len = wlist_len;
rok->wlist = wcl;
return (TRUE);
return (FALSE);
End


void rfs3_srvrfini(void)
Nothing to do
End