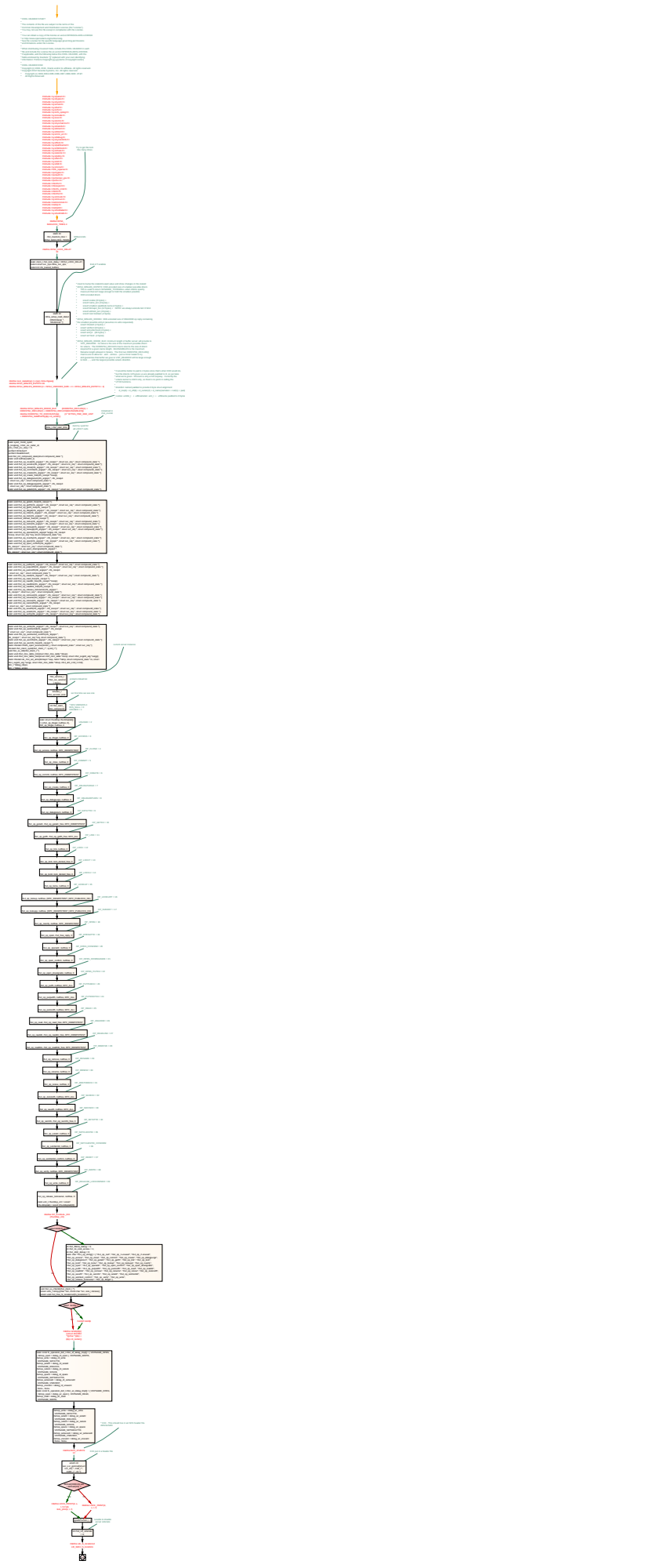
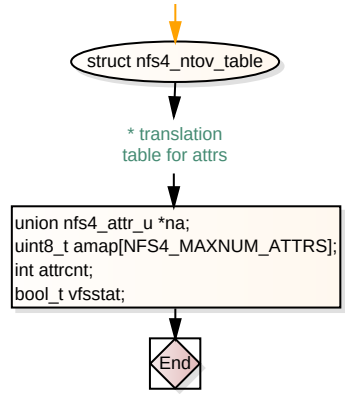
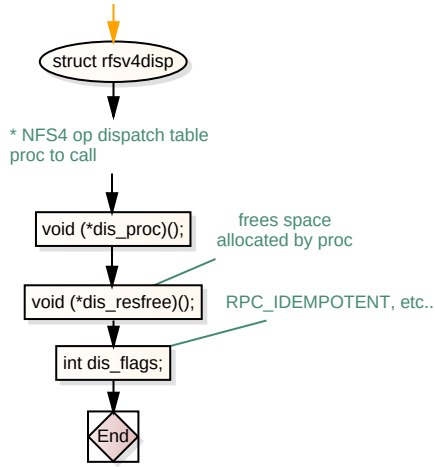
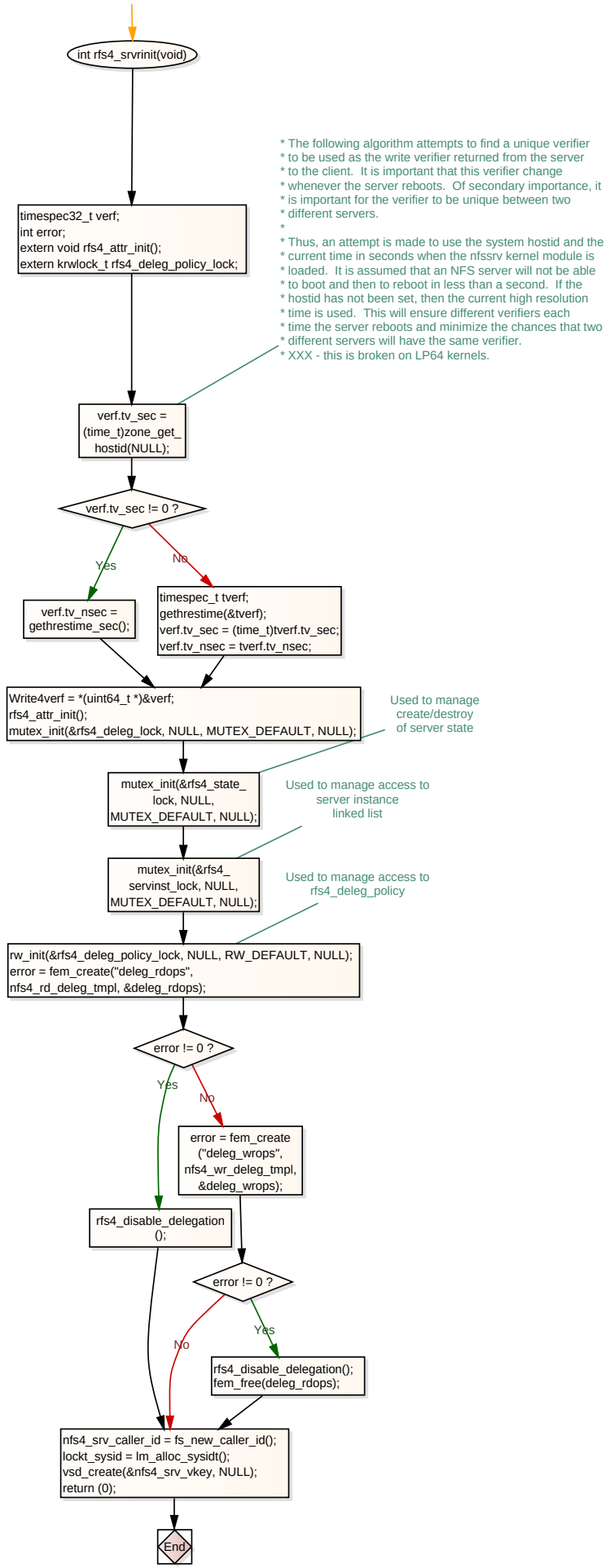
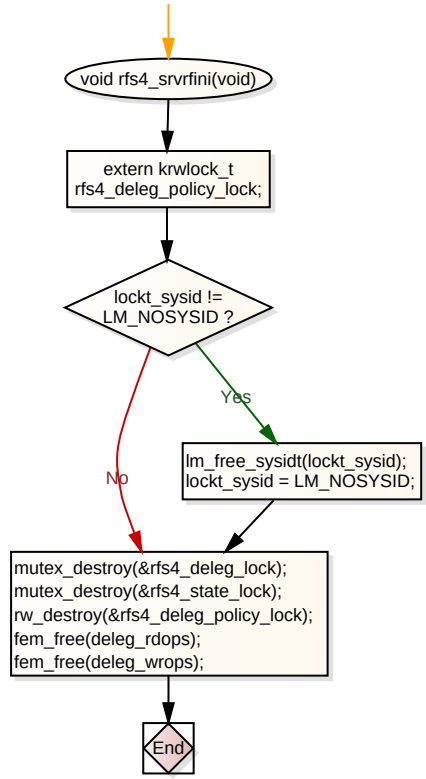
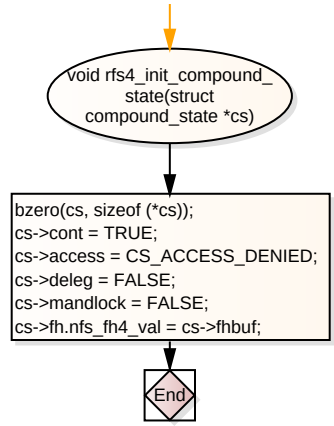


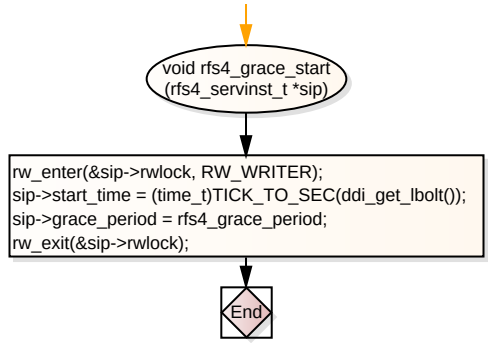

struct nfs4_ntov_table
* translation
table for attrs
union nfs4_attr_u *na;
uint8_t amap[NFS4_MAXNUM_ATTRS];
int attrcnt;
bool_t vfsstat;
End

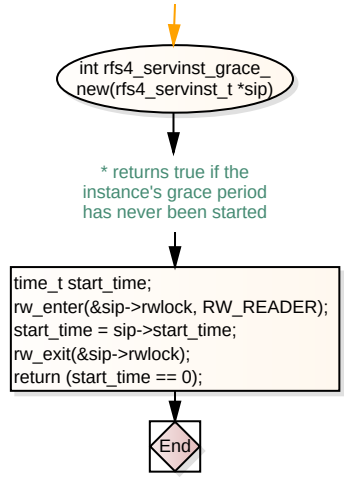

struct rfsv4disp
* NFS4 op dispatch table
proc to call
void (*dis_proc)();
frees space
allocated by proc
void (*dis_resfree)();
RPC_IDEMPOTENT, etc...
int dis_flags;
End

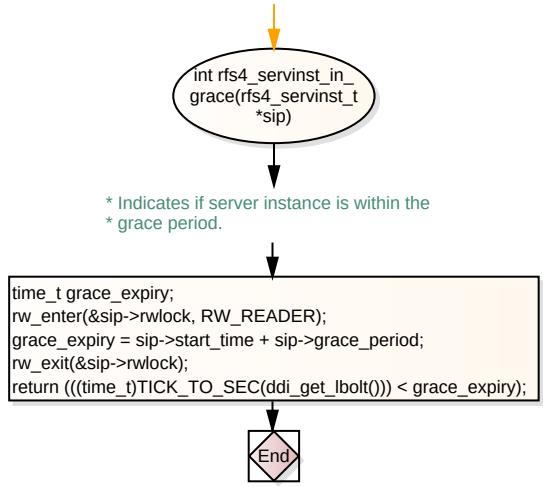

int rfs4_srvrinit(void)
timespec32_t verf;
int error;
extern void rfs4_attr_init();
extern krwlock_t rfs4_deleg_policy_lock;
* The following algorithm attempts to find a unique verifier
* to be used as the write verifier returned from the server
* to the client. It is important that this verifier change
* 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.
* XXX - this is broken on LP64 kernels.
verf.tv_sec = (time_t)zone_get_hostid(NULL);
verf.tv_sec != 0 ?
Yes
No
verf.tv_nsec = gethrestime_sec();
timespec_t tverf;
gethrestime(&tverf);
verf.tv_sec = (time_t)tverf.tv_sec;
verf.tv_nsec = tverf.tv_nsec;
Write4verf = *(uint64_t *)&verf;
rfs4_attr_init();
mutex_init(&rfs4_deleg_lock, NULL, MUTEX_DEFAULT, NULL);
Used to manage create/destroy of server state
mutex_init(&rfs4_state_lock, NULL, MUTEX_DEFAULT, NULL);
Used to manage access to server instance linked list
mutex_init(&rfs4_servinst_lock, NULL, MUTEX_DEFAULT, NULL);
Used to manage access to rfs4_deleg_policy
rw_init(&rfs4_deleg_policy_lock, NULL, RW_DEFAULT, NULL);
error = fem_create("deleg_rdops", nfs4_rd_deleg_tmpl, &deleg_rdops);
error != 0 ?
Yes
No
error = fem_create ("deleg_wrops", nfs4_wr_deleg_tmpl, &deleg_wrops);
rfs4_disable_delegation ();
error != 0 ?
No
Yes
rfs4_disable_delegation(); fem_free(deleg_rdops);
nfs4_srv_caller_id = fs_new_caller_id();
lockt_sysid = lm_alloc_sysidt();
vsd_create(&nfs4_srv_vkey, NULL);
return (0);
End

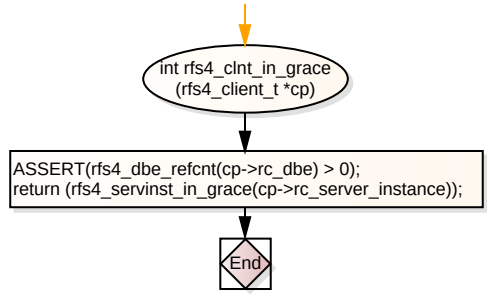

void rfs4_srvrfini(void)
extern krwlock_t
rfs4_deleg_policy_lock;
lockt_sysid !=
LM_NOSYSID ?
Yes
No
lm_free_sysidt(lockt_sysid);
lockt_sysid = LM_NOSYSID;
mutex_destroy(&rfs4_deleg_lock);
mutex_destroy(&rfs4_state_lock);
rw_destroy(&rfs4_deleg_policy_lock);
fem_free(deleg_rdops);
fem_free(deleg_wrops);
End

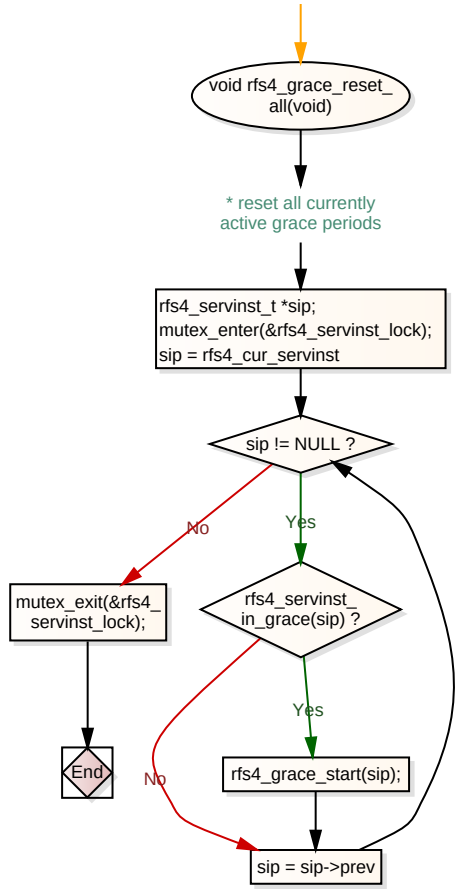

void rfs4_init_compound_
state(struct
compound_state *cs)
bzero(cs, sizeof (*cs));
cs->cont = TRUE;
cs->access = CS_ACCESS_DENIED;
cs->deleg = FALSE;
cs->mandlock = FALSE;
cs->fh.nfs_fh4_val = cs->fhbuf;
End

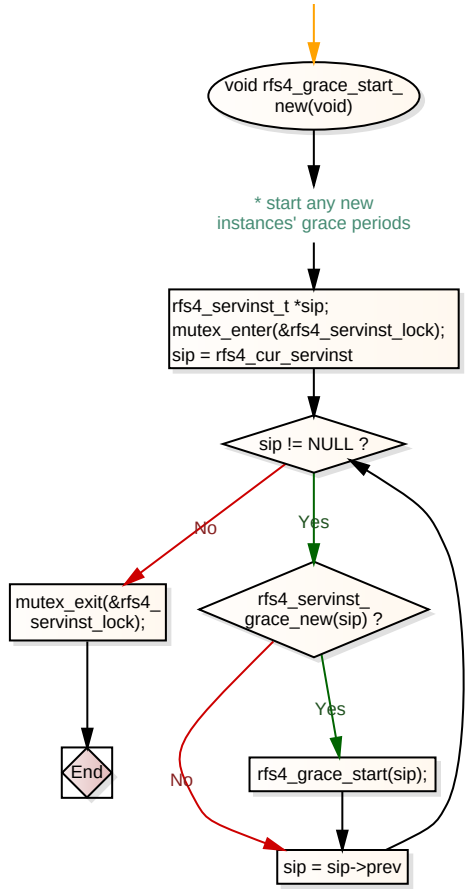

void rfs4_grace_start
(rfs4_servinst_t *sip)
rw_enter(&sip->rwlock, RW_WRITER);
sip->start_time = (time_t)TICK_TO_SEC(ddi_get_lbolt());
sip->grace_period = rfs4_grace_period;
rw_exit(&sip->rwlock);
End

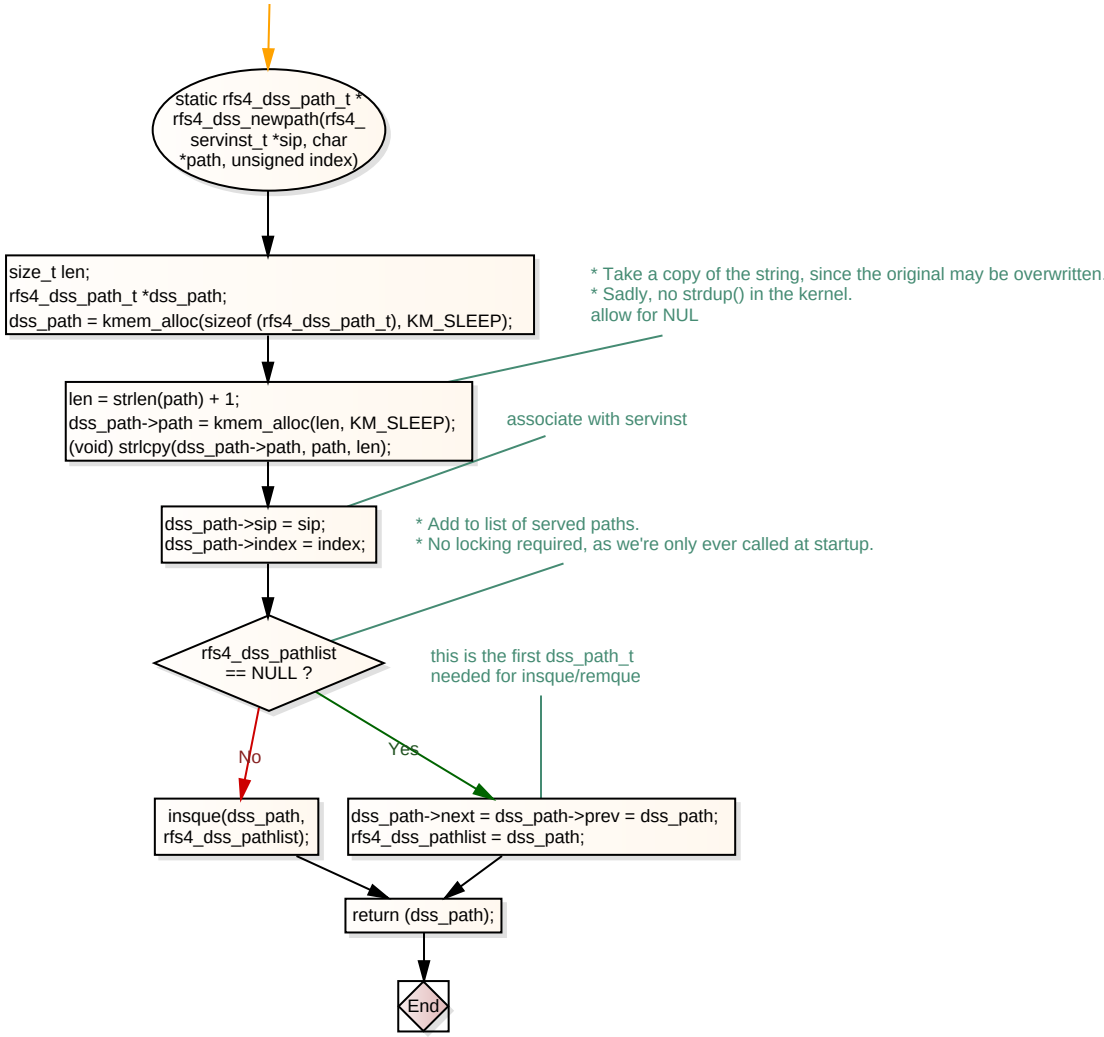

int rfs4_servinst_grace_
new(rfs4_servinst_t *sip)
* returns true if the
instance's grace period
has never been started
time_t start_time;
rw_enter(&sip->rwlock, RW_READER);
start_time = sip->start_time;
rw_exit(&sip->rwlock);
return (start_time == 0);
End

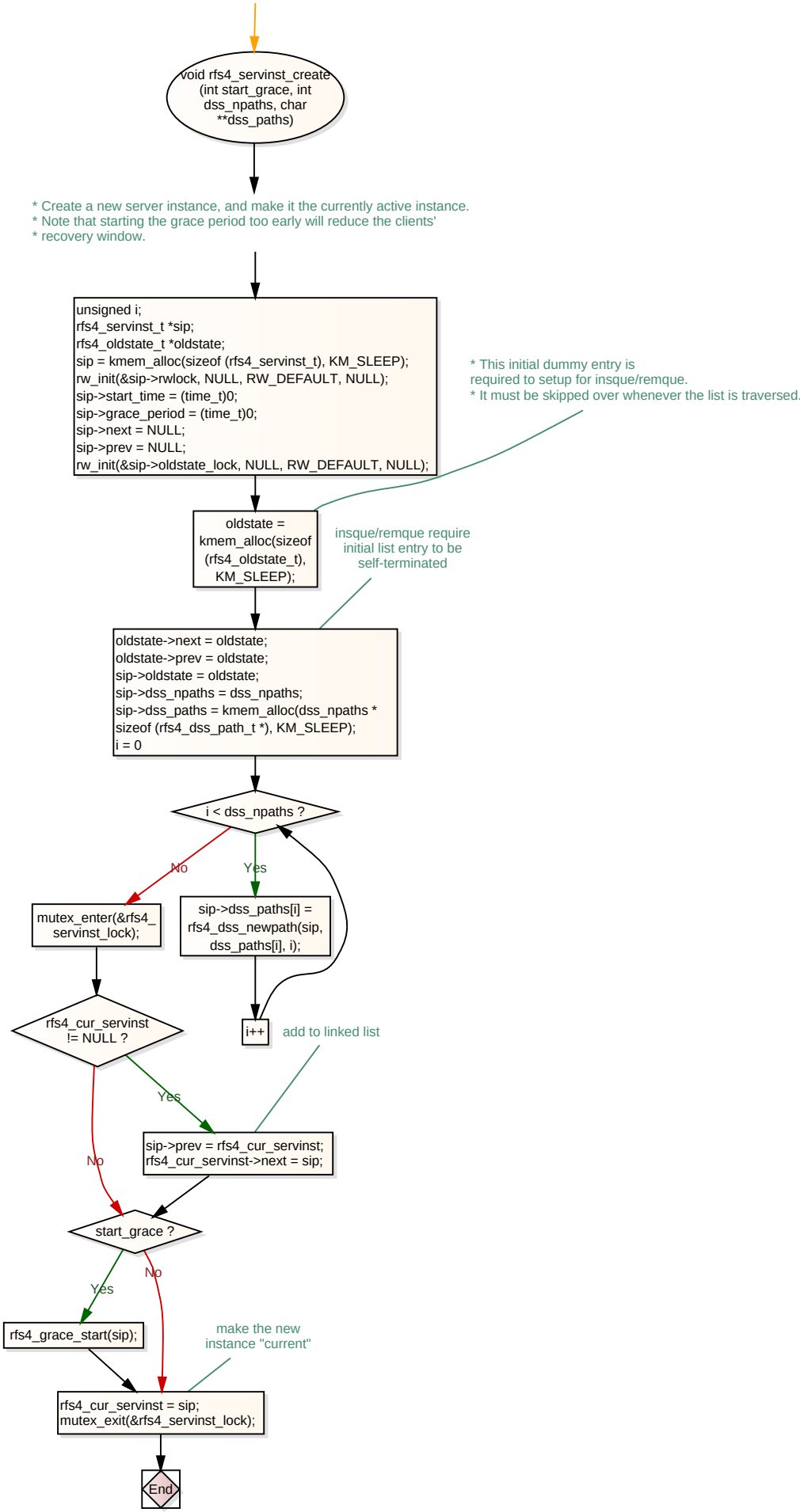

int rfs4_servinst_in_
grace(rfs4_servinst_t
*sip)
* Indicates if server instance is within the
* grace period.
time_t grace_expiry;
rw_enter(&sip->rwlock, RW_READER);
grace_expiry = sip->start_time + sip->grace_period;
rw_exit(&sip->rwlock);
return (((time_t)TICK_TO_SEC(ddi_get_lbolt())) < grace_expiry);
End

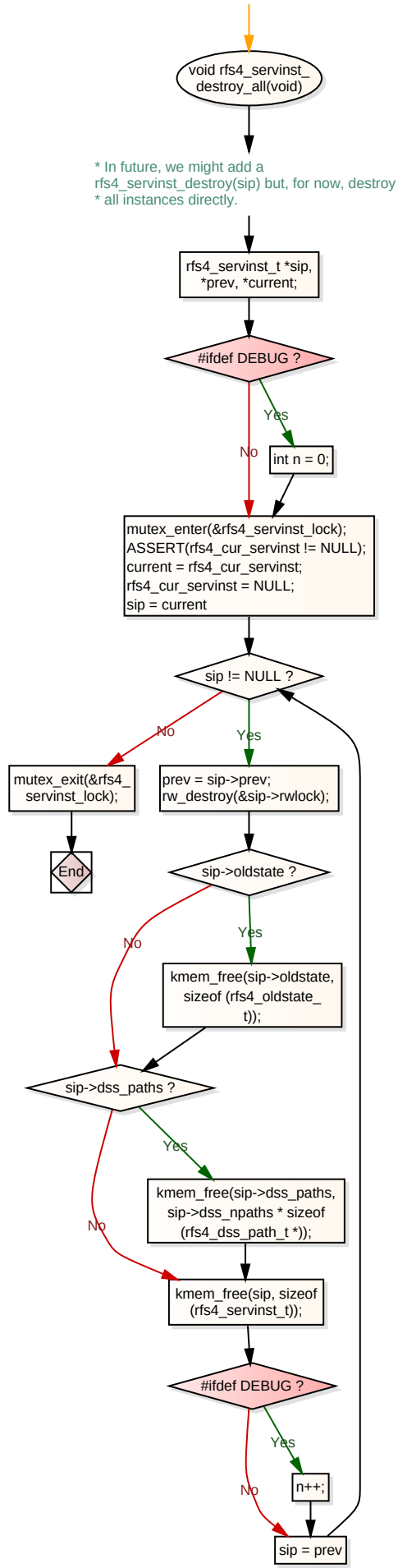

int rfs4_clnt_in_grace
(rfs4_client_t *cp)
ASSERT(rfs4_dbe_refcnt(cp->rc_dbe) > 0);
return (rfs4_servinst_in_grace(cp->rc_server_instance));
End

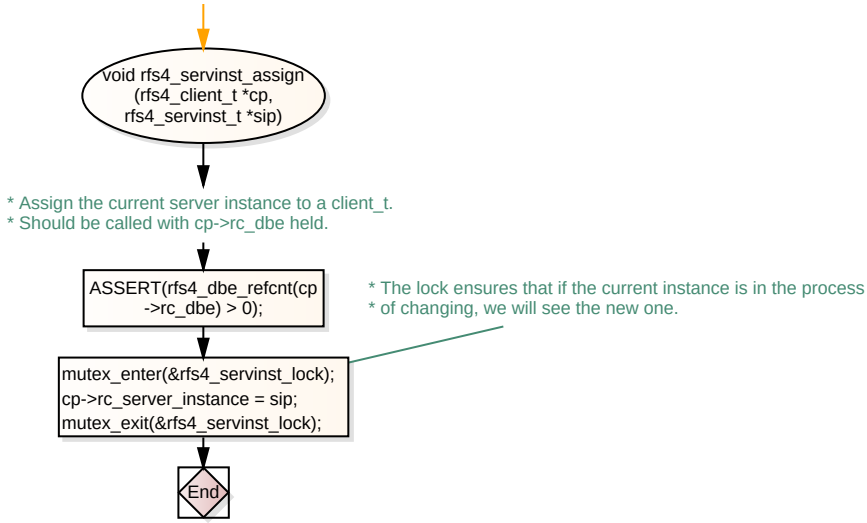

void rfs4_grace_reset_
all(void)
* reset all currently
active grace periods
rfs4_servinst_t *sip;
mutex_enter(&rfs4_servinst_lock);
sip = rfs4_cur_servinst
sip != NULL ?
No
Yes
mutex_exit(&rfs4_
servinst_lock);
rfs4_servinst_
in_grace(sip) ?
Yes
End
No
rfs4_grace_start(sip);
sip = sip->prev

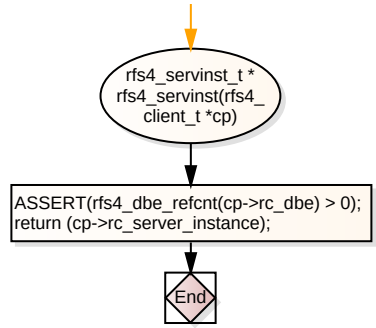

void rfs4_grace_start_
new(void)
* start any new
instances' grace periods
rfs4_servinst_t *sip;
mutex_enter(&rfs4_servinst_lock);
sip = rfs4_cur_servinst
sip != NULL ?
No
Yes
mutex_exit(&rfs4_
servinst_lock);
rfs4_servinst_
grace_new(sip) ?
End
Yes
No
rfs4_grace_start(sip);
sip = sip->prev

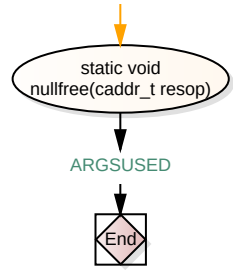

static rfs4_dss_path_t *
rfs4_dss_newpath(rfs4_
servinst_t *sip, char
*path, unsigned index)
size_t len;
rfs4_dss_path_t *dss_path;
dss_path = kmem_alloc(sizeof (rfs4_dss_path_t), KM_SLEEP);
* Take a copy of the string, since the original may be overwritten.
* Sadly, no strdup() in the kernel.
allow for NUL
len = strlen(path) + 1;
dss_path->path = kmem_alloc(len, KM_SLEEP);
(void) strlcpy(dss_path->path, path, len);
associate with servinst
dss_path->sip = sip;
dss_path->index = index;
* Add to list of served paths.
* No locking required, as we're only ever called at startup.
rfs4_dss_pathlist
== NULL ?
this is the first dss_path_t
needed for insque/remque
No
Yes
insque(dss_path,
rfs4_dss_pathlist);
dss_path->next = dss_path->prev = dss_path;
rfs4_dss_pathlist = dss_path;
return (dss_path);
End

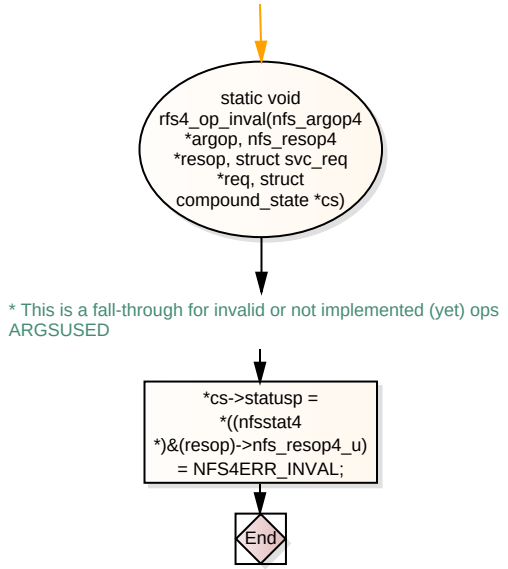

void rfs4_servinst_create (int start_grace, int dss_npaths, char **dss_paths)
* Create a new server instance, and make it the currently active instance.
* Note that starting the grace period too early will reduce the clients'
* recovery window.
unsigned i;
rfs4_servinst_t *sip;
rfs4_oldstate_t *oldstate;
sip = kmem_alloc(sizeof (rfs4_servinst_t), KM_SLEEP);
rw_init(&sip->rwlock, NULL, RW_DEFAULT, NULL);
sip->start_time = (time_t)0;
sip->grace_period = (time_t)0;
sip->next = NULL;
sip->prev = NULL;
rw_init(&sip->oldstate_lock, NULL, RW_DEFAULT, NULL);
* This initial dummy entry is
required to setup for insque/remque.
* It must be skipped over whenever the list is traversed.
oldstate = kmem_alloc(sizeof (rfs4_oldstate_t), KM_SLEEP);
insque/remque require initial list entry to be self-terminated
oldstate->next = oldstate;
oldstate->prev = oldstate;
sip->oldstate = oldstate;
sip->dss_npaths = dss_npaths;
sip->dss_paths = kmem_alloc(dss_npaths * sizeof (rfs4_dss_path_t *), KM_SLEEP);
i = 0
i < dss_npaths ?
No
Yes
mutex_enter(&rfs4_servinst_lock);
sip->dss_paths[i] = rfs4_dss_newpath(sip, dss_paths[i], i);
rfs4_cur_servinst != NULL ?
i++
add to linked list
Yes
No
sip->prev = rfs4_cur_servinst;
rfs4_cur_servinst->next = sip;
start_grace ?
Yes
No
rfs4_grace_start(sip);
make the new instance "current"
rfs4_cur_servinst = sip;
mutex_exit(&rfs4_servinst_lock);
End

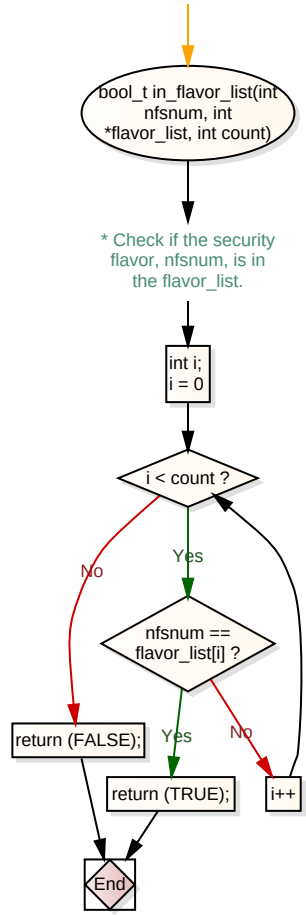

void rfs4_servinst_
destroy_all(void)
* In future, we might add a
rfs4_servinst_destroy(sip) but, for now, destroy
* all instances directly.
rfs4_servinst_t *sip,
*prev, *current;
#ifdef DEBUG ?
Yes
No
int n = 0;
mutex_enter(&rfs4_servinst_lock);
ASSERT(rfs4_cur_servinst != NULL);
current = rfs4_cur_servinst;
rfs4_cur_servinst = NULL;
sip = current
sip != NULL ?
No
Yes
mutex_exit(&rfs4_
servinst_lock);
prev = sip->prev;
rw_destroy(&sip->rwlock);
End
sip->oldstate ?
No
Yes
kmem_free(sip->oldstate,
sizeof (rfs4_oldstate_
t));
sip->dss_paths ?
Yes
No
kmem_free(sip->dss_paths,
sip->dss_npaths * sizeof
(rfs4_dss_path_t *));
kmem_free(sip, sizeof
(rfs4_servinst_t));
#ifdef DEBUG ?
Yes
No
n++;
sip = prev

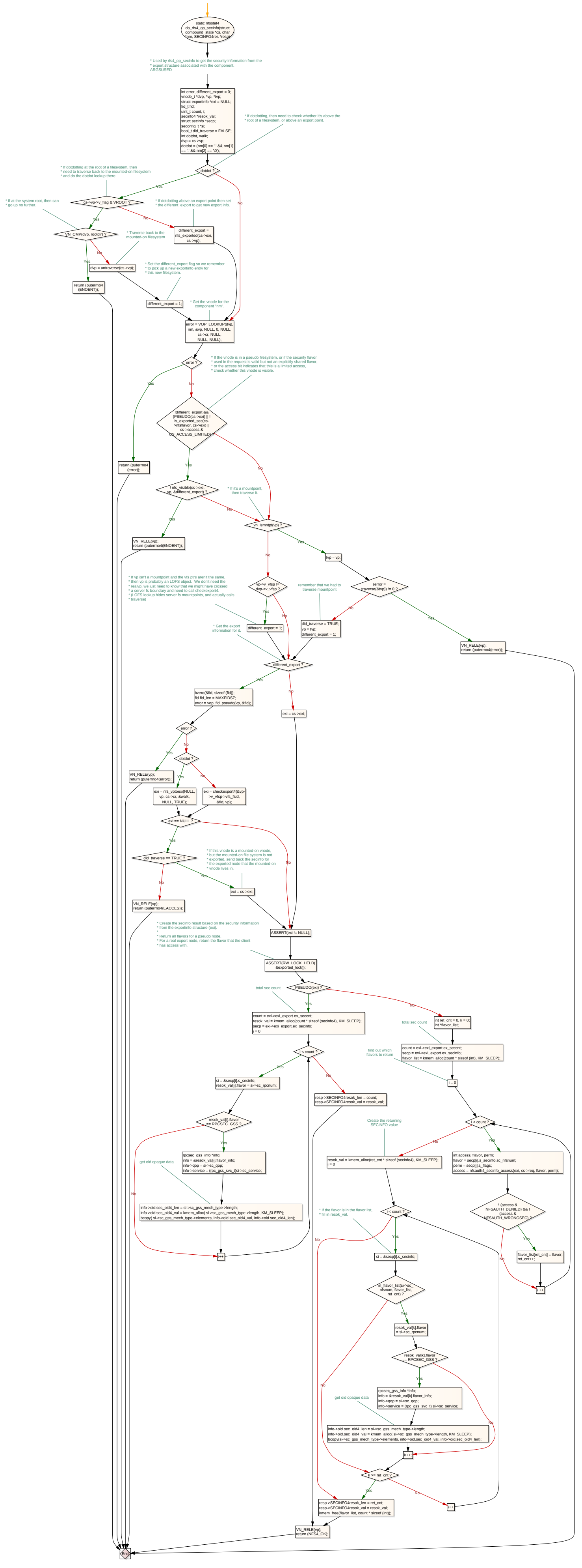

void rfs4_servinst_assign
(rfs4_client_t *cp,
rfs4_servinst_t *sip)
* Assign the current server instance to a client_t.
* Should be called with cp->rc_dbe held.
ASSERT(rfs4_dbe_refcnt(cp
->rc_dbe) > 0);
* The lock ensures that if the current instance is in the process
* of changing, we will see the new one.
mutex_enter(&rfs4_servinst_lock);
cp->rc_server_instance = sip;
mutex_exit(&rfs4_servinst_lock);
End

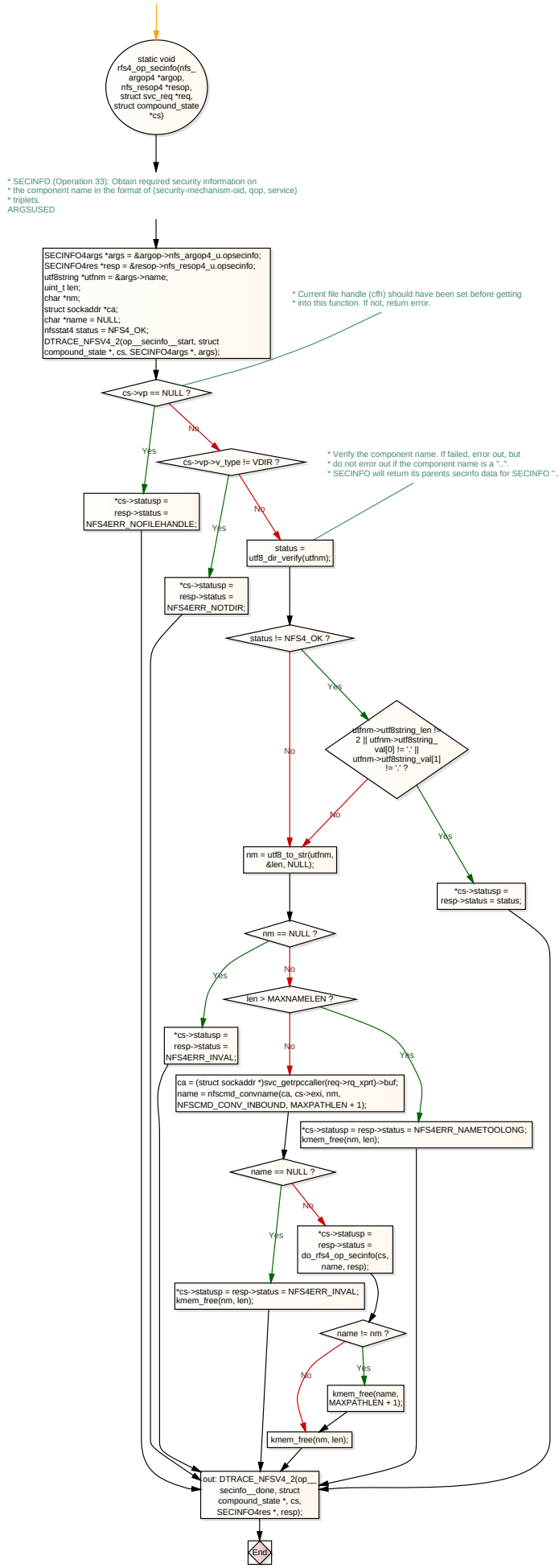

rfs4_servinst_t *
rfs4_servinst(rfs4_
client_t *cp)
ASSERT(rfs4_dbe_refcnt(cp->rc_dbe) > 0);
return (cp->rc_server_instance);
End

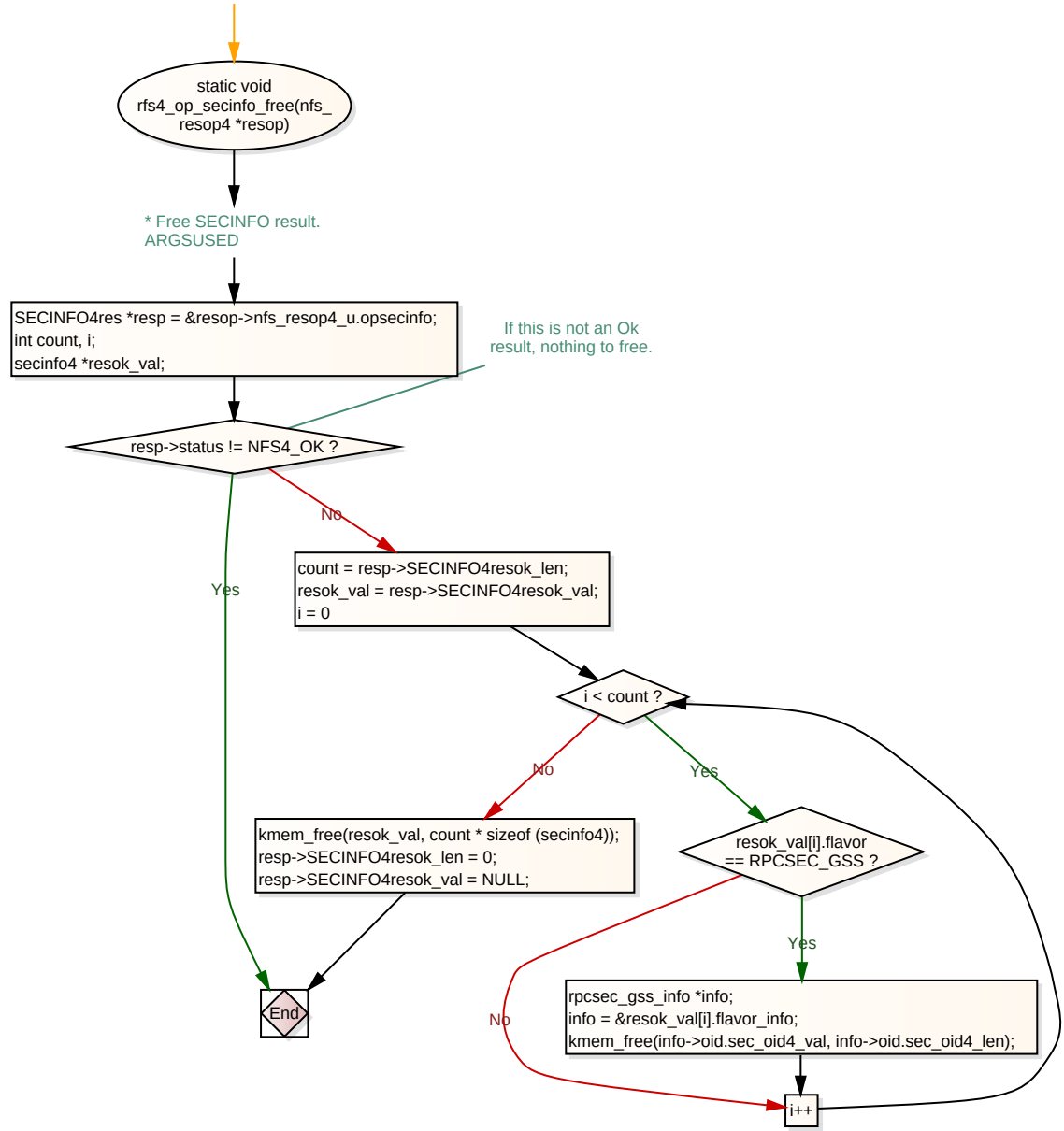

static void
nullfree(caddr_t resop)
ARGSUSED
End

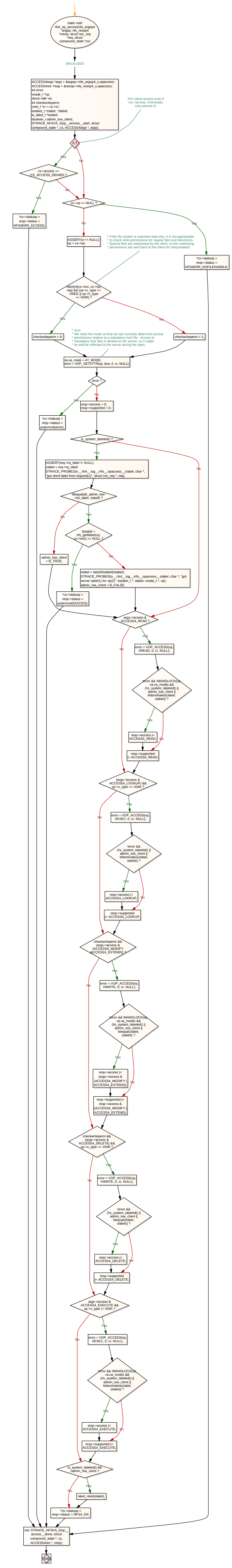

static void rfs4_op_inval(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* This is a fall-through for invalid or not implemented (yet) ops
ARGSUSED
*cs->statusp = *((nfsstat4 *)&(resop)->nfs_resop4_u) = NFS4ERR_INVAL;
End

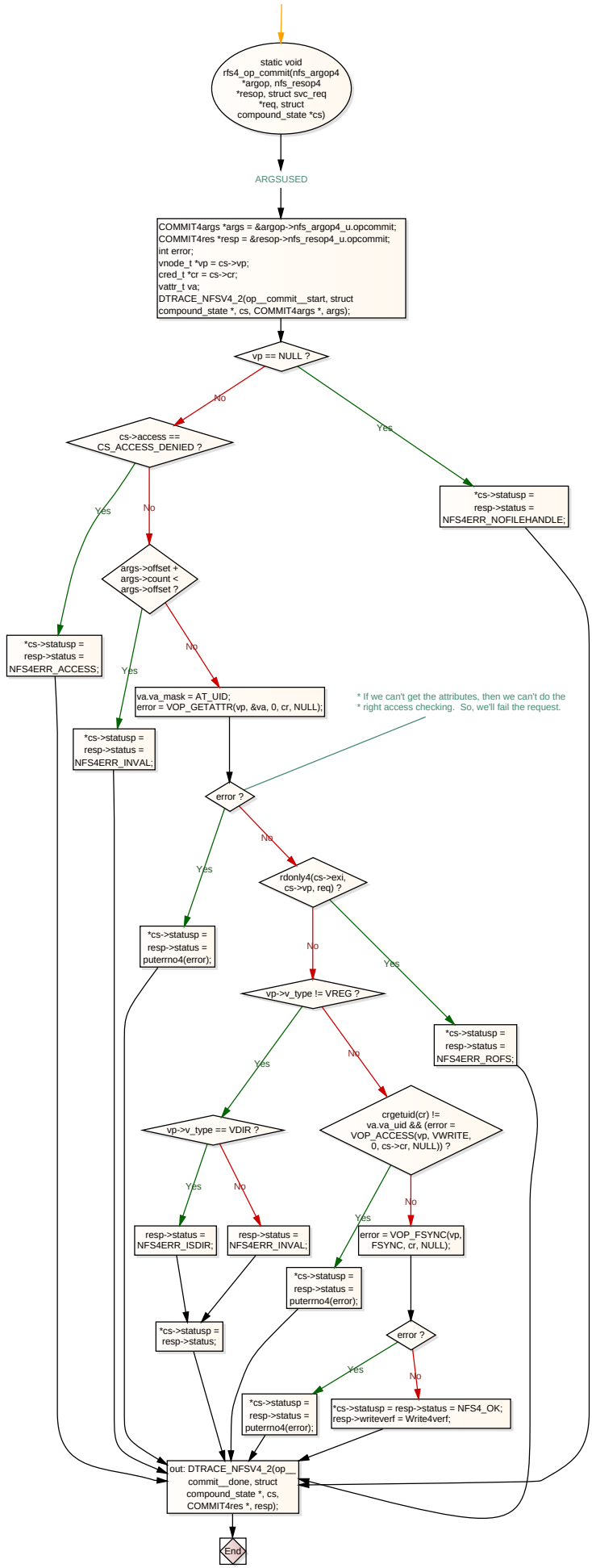

bool_t in_flavor_list(int nfsnum, int *flavor_list, int count)
* Check if the security flavor, nfsnum, is in the flavor_list.
int i;
i = 0
i < count ?
No
Yes
nfsnum == flavor_list[i] ?
return (FALSE);
Yes
No
return (TRUE);
i++
End

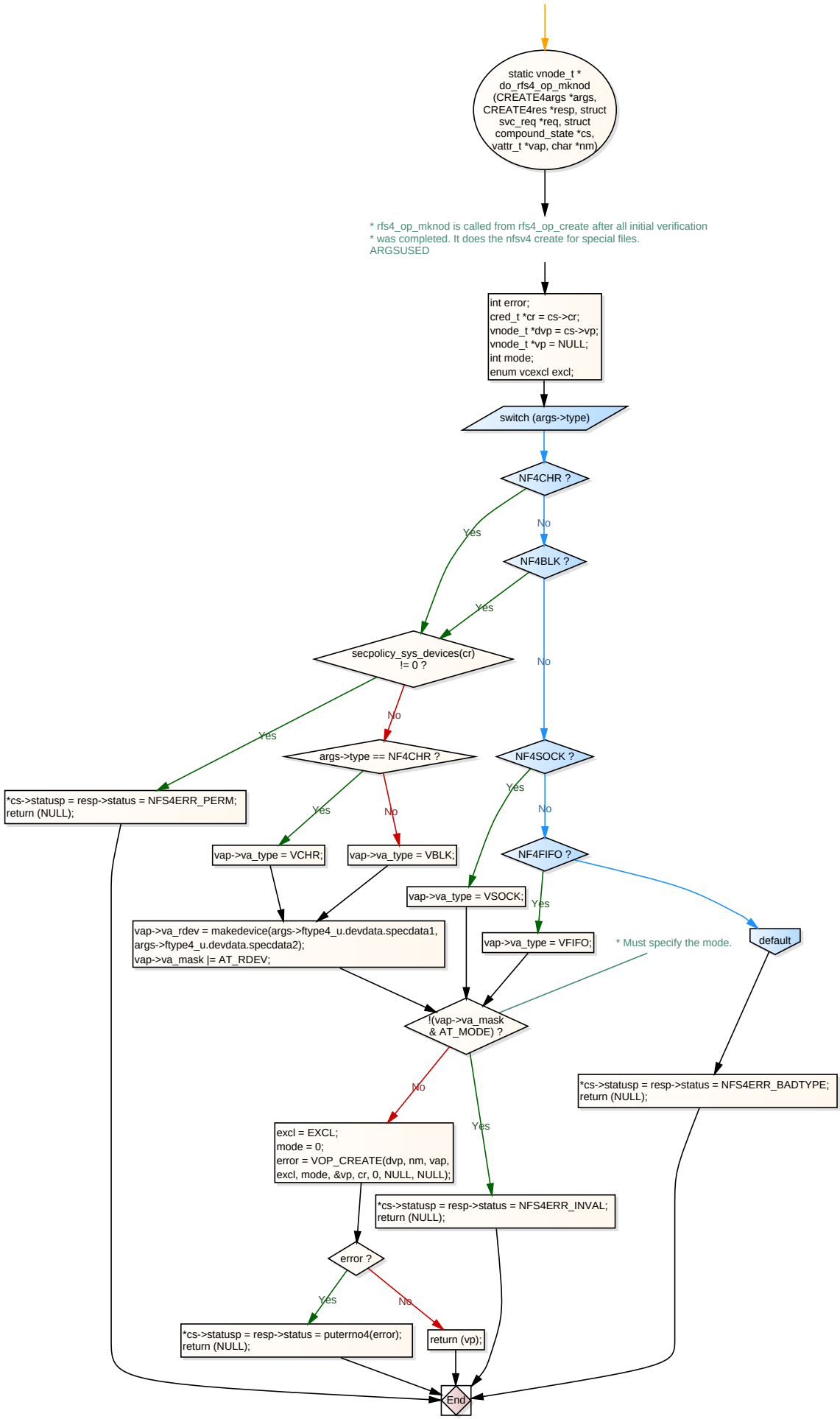

static nfsstat4 do_rfs4_op_secinfo(struct compound_state *cs, char *nm, SECINFO4res *resp)
* Used by rfs4_op_secinfo to get the security information from the
* export structure associated with the component.
ARGSUSED
int error, different_export = 0;
vnode_t *dvp, *vp, *tvp;
struct exportinfo *exi = NULL;
fid_t fid;
uint_t count, i;
secinfo4 *resok_val;
struct secinfo *secp;
seconfig_t *si;
bool_t did_traverse = FALSE;
int dotdot, walk;
dvp = cs->vp;
dotdot = (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0');
* If dotdotting, then need to check whether it's above the
* root of a filesystem, or above an export point.
dotdot ?
* If dotdotting at the root of a filesystem, then
* need to traverse back to the mounted-on filesystem
* and do the dotdot lookup there.
cs->vp->v_flag & VROOT ?
* If dotdotting above an export point then set
* the different_export to get new export info.
different_export = nfs_exported(cs->exi, cs->vp);
* If at the system root, then can
* go up no further.
VN_CMP(dvp, rootdir) ?
* Traverse back to the mounted-on filesystem
dvp = untraverse(cs->vp);
return (puterrno4 (ENOENT));
* Set the different_export flag so we remember
* to pick up a new exportinfo entry for
* this new filesystem.
different_export = 1;
* Get the vnode for the component "nm".
error = VOP_LOOKUP(dvp, nm, &vp, NULL, 0, NULL, cs->cr, NULL, NULL, NULL);
error ?
* If the vnode is in a pseudo filesystem, or if the security flavor
* used in the request is valid but not an explicitly shared flavor,
* or the access bit indicates that this is a limited access,
* check whether this vnode is visible.
return (puterrno4 (error));
!different_export && (PSEUDO(cs->exi) || !is_exported_sec(cs->nfsflavor, cs->exi) || cs->access & CS_ACCESS_LIMITED) ?
! nfs_visible(cs->exi, vp, &different_export) ?
* If it's a mountpoint, then traverse it.
VN_RELE(vp);
return (puterrno4(ENOENT));
vn_ismntpt(vp) ?
tvp = vp;
* If vp isn't a mountpoint and the vfs ptrs aren't the same,
* then vp is probably an LOFS object. We don't need the
* realvp, we just need to know that we might have crossed
* a server fs boundary and need to call checkexport4.
* (LOFS lookup hides server fs mountpoints, and actually calls
* traverse)
vp->v_vfsp != dvp->v_vfsp ?
remember that we had to traverse mountpoint
(error = traverse(&tvp)) != 0 ?
different_export = 1;
did_traverse = TRUE;
vp = tvp;
different_export = 1;
VN_RELE(vp);
return (puterrno4(error));
* Get the export information for it.
different_export ?
bzero(&fid, sizeof (fid));
fid.fid_len = MAXFIDSZ;
error = vop_fid_pseudo(vp, &fid);
exi = cs->exi;
error ?
dotdot ?
VN_RELE(vp);
return (puterrno4(error));
exi = nfs_vptoexi(NULL, vp, cs->cr, &walk, NULL, TRUE);
exi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);
exi == NULL ?
did_traverse == TRUE ?
* If this vnode is a mounted-on vnode,
* but the mounted-on file system is not
* exported, send back the secinfo for
* the exported node that the mounted-on
* vnode lives in.
exi = cs->exi;
VN_RELE(vp);
return (puterrno4(EACCES));
* Create the secinfo result based on the security information
* from the exportinfo structure (exi).
*
* Return all flavors for a pseudo node.
* For a real export node, return the flavor that the client
* has access with.
ASSERT(exi != NULL);
ASSERT(RW_LOCK_HELD(&exported_lock));
PSEUDO(exi) ?
total sec count
count = exi->exi_export.ex_seccnt;
resok_val = kmem_alloc(count * sizeof (secinfo4), KM_SLEEP);
secp = exi->exi_export.ex_secinfo;
i = 0
int ret_cnt = 0, k = 0;
int *flavor_list;
total sec count
count = exi->exi_export.ex_seccnt;
secp = exi->exi_export.ex_secinfo;
flavor_list = kmem_alloc(count * sizeof (int), KM_SLEEP);
find out which flavors to return
i = 0
i < count ?
si = &secp[i].s_secinfo;
resok_val[i].flavor = si->sc_rpcnum;
resp->SECINFO4resok_len = count;
resp->SECINFO4resok_val = resok_val;
resok_val[i].flavor == RPCSEC_GSS ?
rpcsec_gss_info *info;
info = &resok_val[i].flavor_info;
info->qop = si->sc_qop;
info->service = (rpc_gss_svc_t)si->sc_service;
get oid opaque data
info->oid.sec_oid4_len = si->sc_gss_mech_type->length;
info->oid.sec_oid4_val = kmem_alloc(si->sc_gss_mech_type->length, KM_SLEEP);
bcopy(si->sc_gss_mech_type->elements, info->oid.sec_oid4_val, info->oid.sec_oid4_len);
i++
i < count ?
Create the returning SECINFO value
resok_val = kmem_alloc(ret_cnt * sizeof (secinfo4), KM_SLEEP);
i = 0
int access, flavor, perm;
flavor = secp[i].s_secinfo.sc_nfsnum;
perm = secp[i].s_flags;
access = nfsauth4_secinfo_access(exi, cs->req, flavor, perm);
! (access & NFSAUTH_DENIED) && ! (access & NFSAUTH_WRONGSEC) ?
flavor_list[ret_cnt] = flavor;
ret_cnt++;
i ++
i < count ?
* If the flavor is in the flavor list,
* fill in resok_val.
si = &secp[i].s_secinfo;
in_flavor_list(si->sc_nfsnum, flavor_list, ret_cnt) ?
resok_val[k].flavor = si->sc_rpcnum;
resok_val[k].flavor == RPCSEC_GSS ?
rpcsec_gss_info *info;
info = &resok_val[k].flavor_info;
info->qop = si->sc_qop;
info->service = (rpc_gss_svc_t) si->sc_service;
get oid opaque data
info->oid.sec_oid4_len = si->sc_gss_mech_type->length;
info->oid.sec_oid4_val = kmem_alloc(si->sc_gss_mech_type->length, KM_SLEEP);
bcopy(si->sc_gss_mech_type->elements, info->oid.sec_oid4_val, info->oid.sec_oid4_len);
k++;
k >= ret_cnt ?
i++
resp->SECINFO4resok_len = ret_cnt;
resp->SECINFO4resok_val = resok_val;
kmem_free(flavor_list, count * sizeof (int));
VN_RELE(vp);
return (NFS4_OK);
End

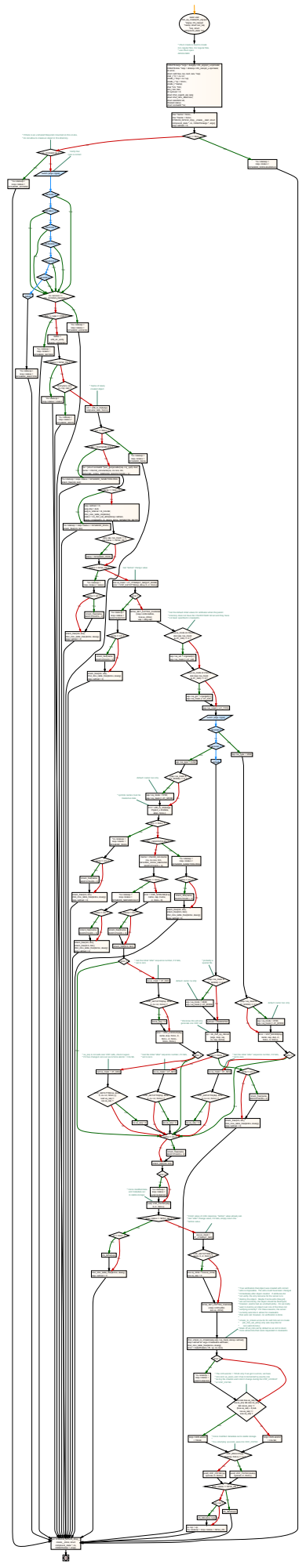

static void rfs4_op_secinfo(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* SECINFO (Operation 33): Obtain required security information on
* the component name in the format of (security-mechanism-oid, qop, service)
* triplets.
ARGSUSED
SECINFO4args *args = &argop->nfs_argop4_u.opsecinfo;
SECINFO4res *resp = &resop->nfs_resop4_u.opsecinfo;
utf8string *utfnm = &args->name;
uint_t len;
char *nm;
struct sockaddr *ca;
char *name = NULL;
nfsstat4 status = NFS4_OK;
DTRACE_NFSV4_2(op__secinfo__start, struct compound_state *, cs, SECINFO4args *, args);
* Current file handle (cfh) should have been set before getting
* into this function. If not, return error.
cs->vp == NULL ?
No
Yes
cs->vp->v_type != VDIR ?
* Verify the component name. If failed, error out, but
* do not error out if the component name is a "..".
* SECINFO will return its parents secinfo data for SECINFO "..".
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
No
Yes
status = utf8_dir_verify(utfnm);
*cs->statusp = resp->status = NFS4ERR_NOTDIR;
status != NFS4_OK ?
Yes
No
utfnm->utf8string_len != 2 || utfnm->utf8string_val[0] != '.' || utfnm->utf8string_val[1] != '.' ?
No
Yes
nm = utf8_to_str(utfnm, &len, NULL);
*cs->statusp = resp->status = status;
nm == NULL ?
Yes
No
len > MAXNAMELEN ?
*cs->statusp = resp->status = NFS4ERR_INVAL;
No
Yes
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
*cs->statusp = resp->status = NFS4ERR_NAMETOOLONG;
kmem_free(nm, len);
name == NULL ?
No
Yes
*cs->statusp = resp->status = do_rfs4_op_secinfo(cs, name, resp);
*cs->statusp = resp->status = NFS4ERR_INVAL;
kmem_free(nm, len);
name != nm ?
No
Yes
kmem_free(name, MAXPATHLEN + 1);
kmem_free(nm, len);
out: DTRACE_NFSV4_2(op__secinfo__done, struct compound_state *, cs, SECINFO4res *, resp);
End

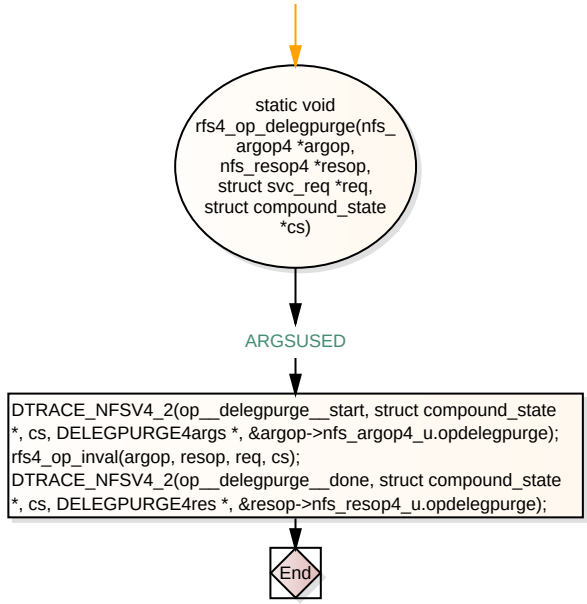

static void
rfs4_op_secinfo_free(nfs_
resop4 *resop)
* Free SECINFO result.
ARGSUSED
SECINFO4res *resp = &resop->nfs_resop4_u.opsecinfo;
int count, i;
secinfo4 *resok_val;
If this is not an Ok
result, nothing to free.
resp->status != NFS4_OK ?
No
Yes
count = resp->SECINFO4resok_len;
resok_val = resp->SECINFO4resok_val;
i = 0
i < count ?
No
Yes
kmem_free(resok_val, count * sizeof (secinfo4));
resp->SECINFO4resok_len = 0;
resp->SECINFO4resok_val = NULL;
resok_val[i].flavor
== RPCSEC_GSS ?
Yes
No
rpcsec_gss_info *info;
info = &resok_val[i].flavor_info;
kmem_free(info->oid.sec_oid4_val, info->oid.sec_oid4_len);
End
i++

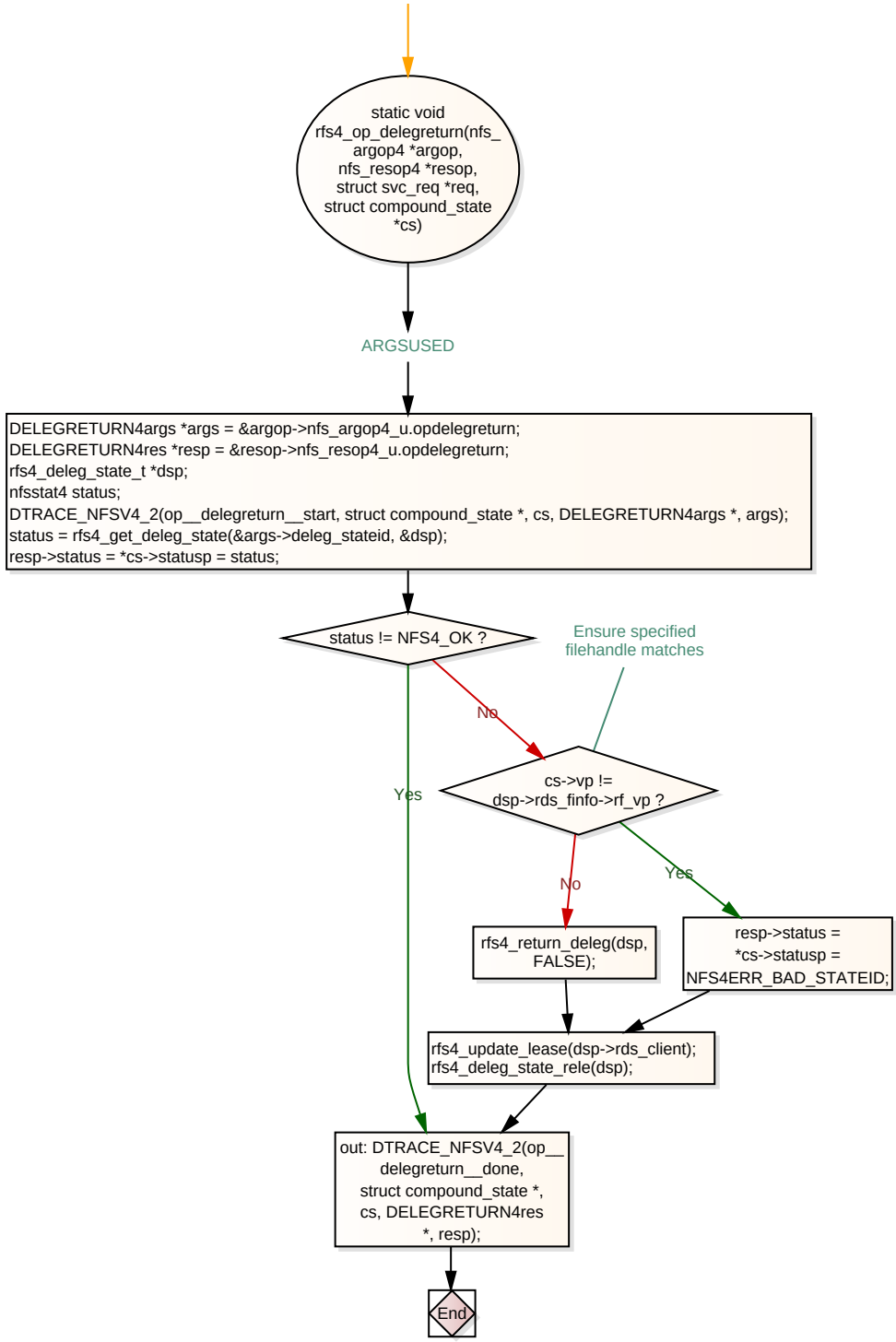

static void rfs4_op_access(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
ACCESS4args *args = &argop->nfs_argop4_u.opaccess;
ACCESS4res *resp = &resop->nfs_resop4_u.opaccess;
int error;
vnode_t *vp;
struct vattr va;
int checkwriteperm;
cred_t *cr = cs->cr;
bslabel_t *clabel, *slabel;
ts_label_t *tslabel;
boolean_t admin_low_client;
DTRACE_NFSV4_2(op__access__start, struct compound_state *, cs, ACCESS4args *, args);
XXX allow access even if !cs->access. Eventually only pseudo fs
0 ?
cs->access == CS_ACCESS_DENIED ?
*cs->statusp = resp->status = NFS4ERR_ACCESS;
cs->vp == NULL ?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
ASSERT(cr != NULL);
vp = cs->vp;
* 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.
rdonly4(cs->exi, cs->vp, req) && (vp->v_type == VREG || vp->v_type == VDIR) ?
checkwriteperm = 0;
checkwriteperm = 1;
* XXX
* 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 ?
*cs->statusp = resp->status = puterrno4(error);
resp->access = 0;
resp->supported = 0;
is_system_labeled() ?
ASSERT(req->rq_label != NULL);
clabel = req->rq_label;
DTRACE_PROBE2(tx__rfs4__log__info__opaccess__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
admin_low_client = B_TRUE;
(tslabel = nfs_getflabel(vp, cs->exi)) == NULL ?
slabel = label2bslabel(tslabel);
DTRACE_PROBE3(tx__rfs4__log__info__opaccess__slabel, char *, "got server label(1) for vp(2)", bslabel_t *, slabel, vnode_t *, vp);
admin_low_client = B_FALSE;
*cs->statusp = resp->status = puterrno4(EACCES);
args->access & ACCESS4_READ ?
error = VOP_ACCESS(vp, VREAD, 0, cr, NULL);
!error && !MANDLOCK(vp, va.va_mode) && (!is_system_labeled() || admin_low_client || bldominates(clabel, slabel)) ?
resp->access |= ACCESS4_READ;
resp->supported |= ACCESS4_READ;
(args->access & ACCESS4_LOOKUP) && vp->v_type == VDIR ?
error = VOP_ACCESS(vp, VEXEC, 0, cr, NULL);
!error && (!is_system_labeled() || admin_low_client || bldominates(clabel, slabel)) ?
resp->access |= ACCESS4_LOOKUP;
resp->supported |= ACCESS4_LOOKUP;
checkwriteperm && (args->access & (ACCESS4_MODIFY | ACCESS4_EXTEND)) ?
error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL);
!error && !MANDLOCK(vp, va.va_mode) && (!is_system_labeled() || admin_low_client || blequal(clabel, slabel)) ?
resp->access |= (args->access & (ACCESS4_MODIFY | ACCESS4_EXTEND));
resp->supported |= resp->access & (ACCESS4_MODIFY | ACCESS4_EXTEND);
checkwriteperm && (args->access & ACCESS4_DELETE) && vp->v_type == VDIR ?
error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL);
!error && (!is_system_labeled() || admin_low_client || blequal(clabel, slabel)) ?
resp->access |= ACCESS4_DELETE;
resp->supported |= ACCESS4_DELETE;
args->access & ACCESS4_EXECUTE && vp->v_type != VDIR ?
error = VOP_ACCESS(vp, VEXEC, 0, cr, NULL);
!error && !MANDLOCK(vp, va.va_mode) && (!is_system_labeled() || admin_low_client || bldominates(clabel, slabel)) ?
resp->access |= ACCESS4_EXECUTE;
resp->supported |= ACCESS4_EXECUTE;
is_system_labeled() && !admin_low_client ?
label_rele(tslabel);
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__access__done, struct compound_state *, cs, ACCESS4res *, resp);
End

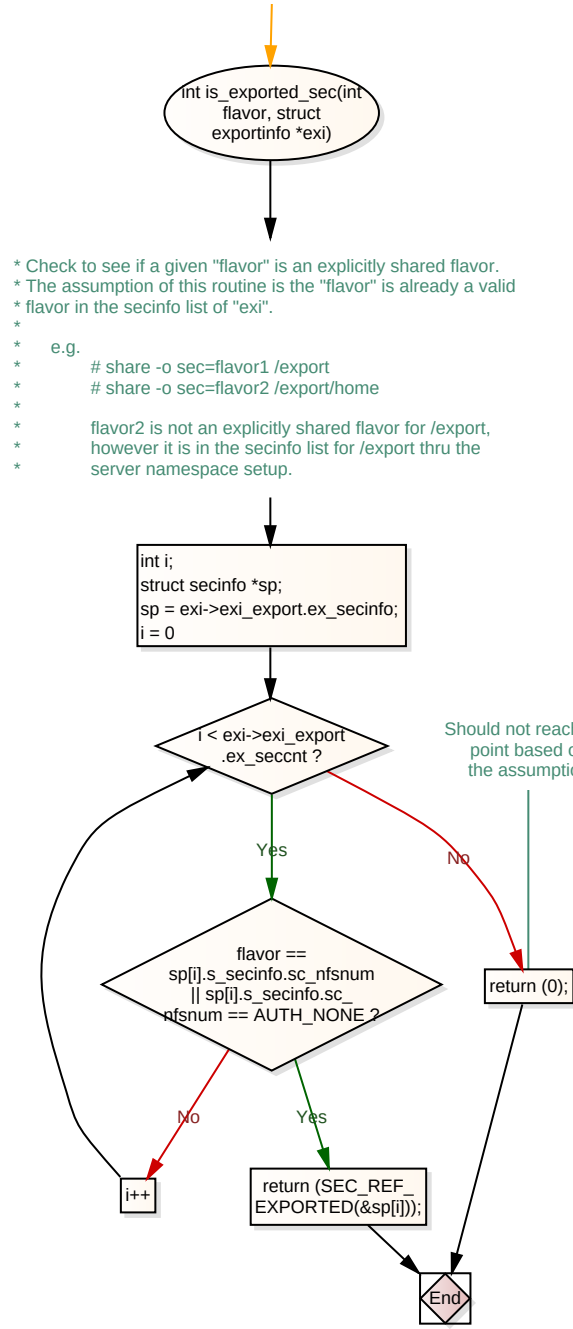

static void rfs4_op_commit(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
COMMIT4args *args = &argop->nfs_argop4_u.opcommit;
COMMIT4res *resp = &resop->nfs_resop4_u.opcommit;
int error;
vnode_t *vp = cs->vp;
cred_t *cr = cs->cr;
vattr_t va;
DTRACE_NFSV4_2(op__commit__start, struct compound_state *, cs, COMMIT4args *, args);
vp == NULL ?
No
Yes
cs->access == CS_ACCESS_DENIED ?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
Yes
No
args->offset + args->count < args->offset ?
*cs->statusp = resp->status = NFS4ERR_ACCESS;
No
Yes
va.va_mask = AT_UID;
error = VOP_GETATTR(vp, &va, 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.
*cs->statusp = resp->status = NFS4ERR_INVAL;
error ?
Yes
No
*cs->statusp = resp->status = puterrno4(error);
rdonly4(cs->exi, cs->vp, req) ?
No
Yes
vp->v_type != VREG ?
*cs->statusp = resp->status = NFS4ERR_ROFS;
Yes
No
vp->v_type == VDIR ?
crgetuid(cr) != va.va_uid && (error = VOP_ACCESS(vp, VWRITE, 0, cs->cr, NULL)) ?
Yes
No
Yes
No
resp->status = NFS4ERR_ISDIR;
resp->status = NFS4ERR_INVAL;
error = VOP_FSYNC(vp, FSYNC, cr, NULL);
*cs->statusp = resp->status = puterrno4(error);
*cs->statusp = resp->status;
error ?
Yes
No
*cs->statusp = resp->status = puterrno4(error);
*cs->statusp = resp->status = NFS4_OK; resp->writeverf = Write4verf;
out: DTRACE_NFSV4_2(op__commit__done, struct compound_state *, cs, COMMIT4res *, resp);
End

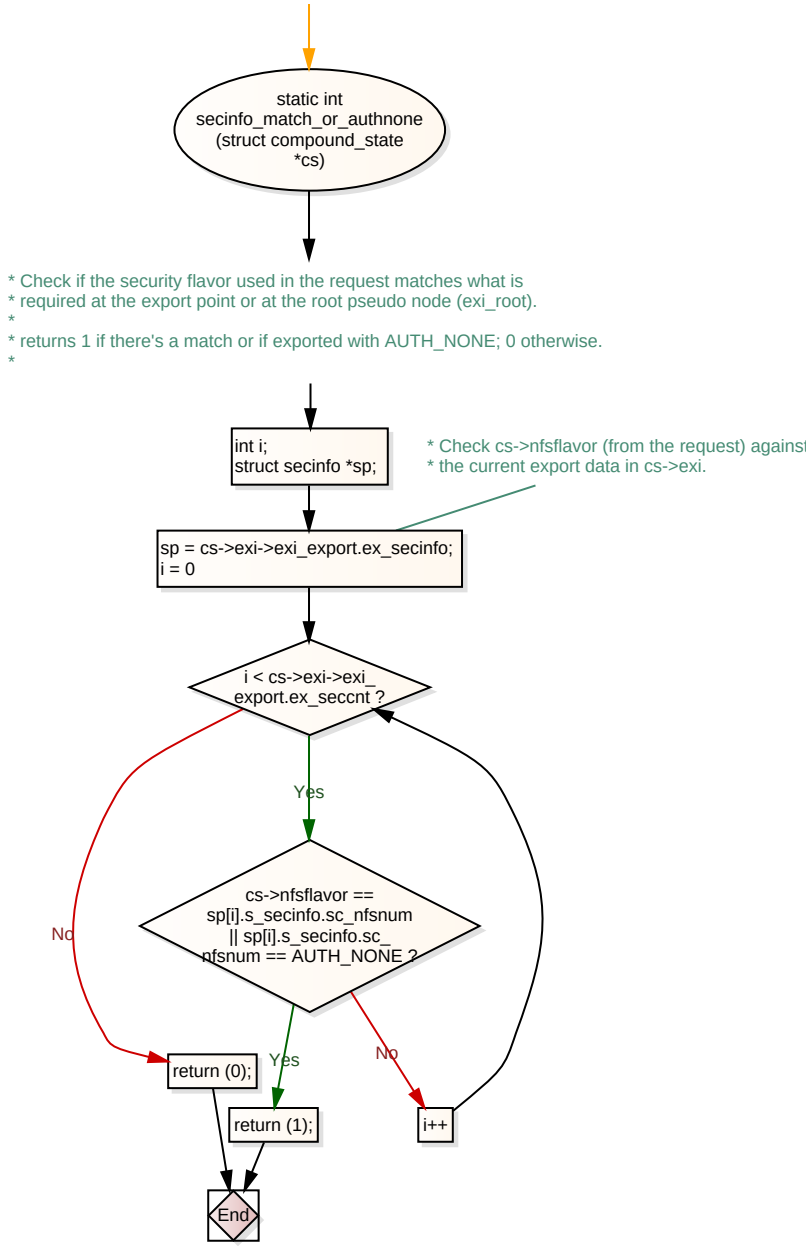

static vnode_t *
do_rfs4_op_mknod
(CREATE4args *args,
CREATE4res *resp, struct
svc_req *req, struct
compound_state *cs,
vattr_t *vap, char *nm)
* rfs4_op_mknod is called from rfs4_op_create after all initial verification
* was completed. It does the nfsv4 create for special files.
ARGSUSED
int error;
cred_t *cr = cs->cr;
vnode_t *dvp = cs->vp;
vnode_t *vp = NULL;
int mode;
enum vcexcl excl;
switch (args->type)
NF4CHR ?
Yes
No
NF4BLK ?
Yes
No
secpolicy_sys_devices(cr)
!= 0 ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_PERM;
return (NULL);
args->type == NF4CHR ?
Yes
No
vap->va_type = VCHR;
vap->va_type = VBLK;
vap->va_rdev = makedevice(args->ftype4_u.devdata.specdata1,
args->ftype4_u.devdata.specdata2);
vap->va_mask |= AT_RDEV;
NF4SOCK ?
Yes
No
vap->va_type = VSOCK;
NF4FIFO ?
Yes
vap->va_type = VFIFO;
default
* Must specify the mode.
!(vap->va_mask
& AT_MODE) ?
No
Yes
excl = EXCL;
mode = 0;
error = VOP_CREATE(dvp, nm, vap,
excl, mode, &vp, cr, 0, NULL, NULL);
*cs->statusp = resp->status = NFS4ERR_INVAL;
return (NULL);
*cs->statusp = resp->status = NFS4ERR_BADTYPE;
return (NULL);
error ?
Yes
No
*cs->statusp = resp->status = puterrno4(error);
return (NULL);
return (vp);
End

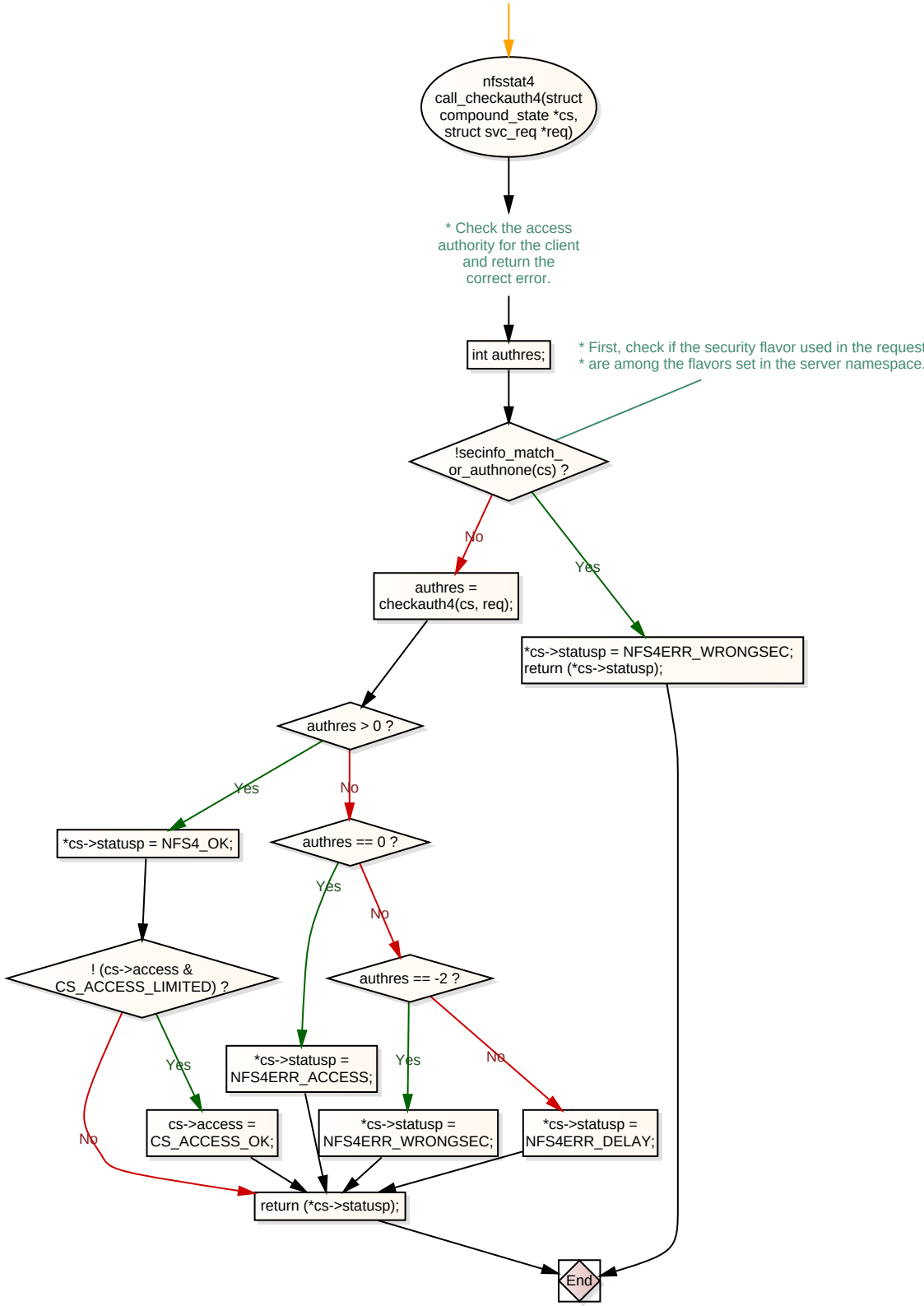


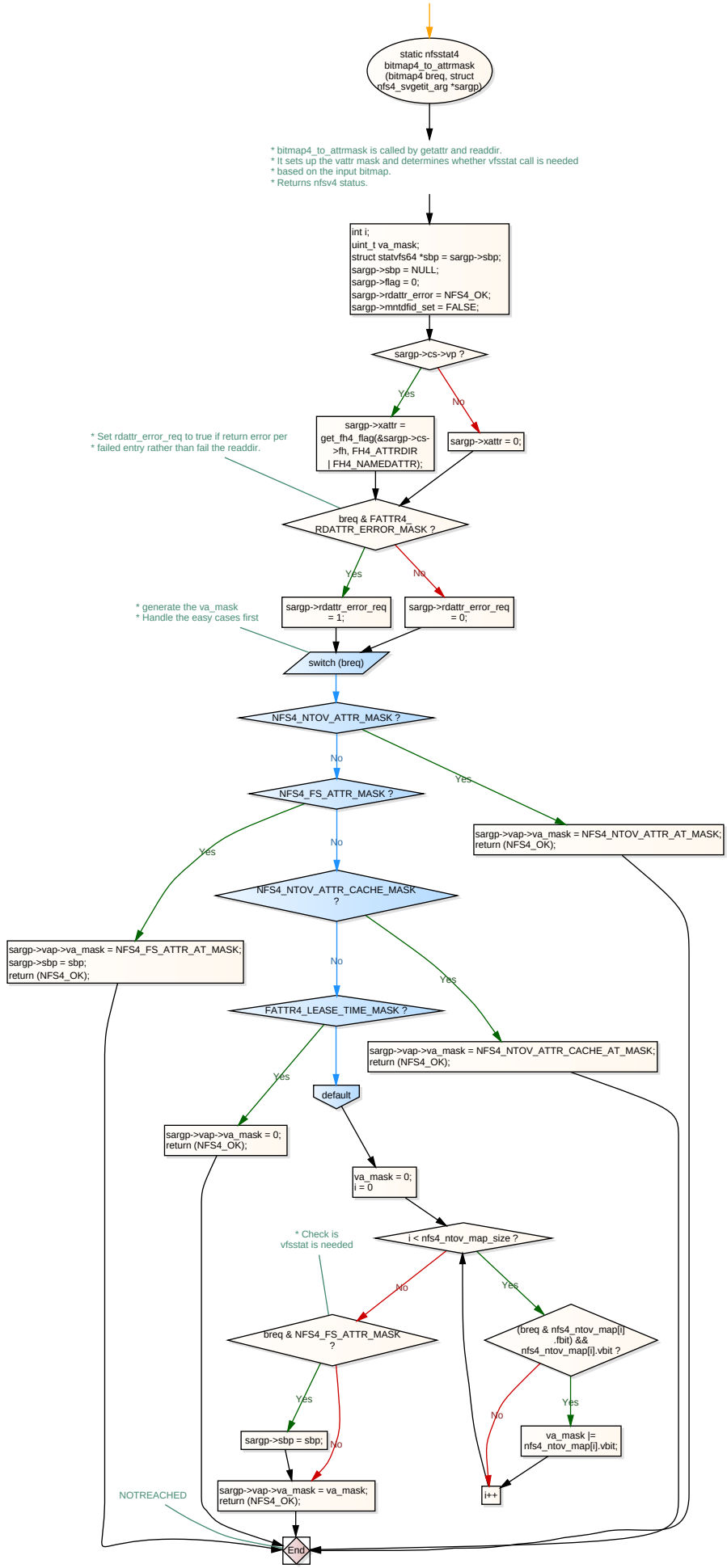

static void
rfs4_op_delegpurge(nfs_
argop4 *argop,
nfs_resop4 *resop,
struct svc_req *req,
struct compound_state
*cs)
ARGSUSED
DTRACE_NFSV4_2(op__delegpurge__start, struct compound_state
*, cs, DELEGPURGE4args *, &argop->nfs_argop4_u.opdelegpurge);
rfs4_op_inval(argop, resop, req, cs);
DTRACE_NFSV4_2(op__delegpurge__done, struct compound_state
*, cs, DELEGPURGE4res *, &resop->nfs_resop4_u.opdelegpurge);
End

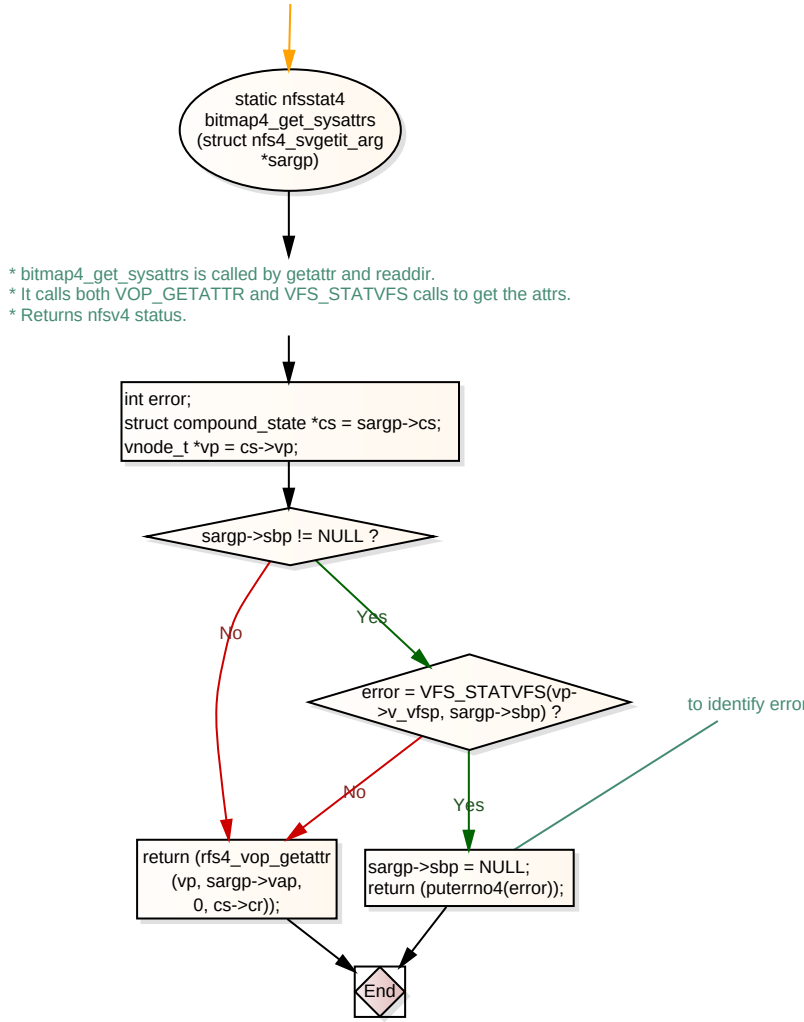

static void rfs4_op_delegreturn(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
DELEGRETURN4args *args = &argop->nfs_argop4_u.opdelegreturn;
DELEGRETURN4res *resp = &resop->nfs_resop4_u.opdelegreturn;
rfs4_deleg_state_t *dsp;
nfsstat4 status;
DTRACE_NFSV4_2(op__delegreturn__start, struct compound_state *, cs, DELEGRETURN4args *, args);
status = rfs4_get_deleg_state(&args->deleg_stateid, &dsp);
resp->status = *cs->statusp = status;
status != NFS4_OK ?
No
Yes
Ensure specified filehandle matches
cs->vp != dsp->rds_finfo->rf_vp ?
No
Yes
rfs4_return_deleg(dsp, FALSE);
resp->status = *cs->statusp = NFS4ERR_BAD_STATEID;
rfs4_update_lease(dsp->rds_client);
rfs4_deleg_state_rele(dsp);
out: DTRACE_NFSV4_2(op__delegreturn__done, struct compound_state *, cs, DELEGRETURN4res *, resp);
End

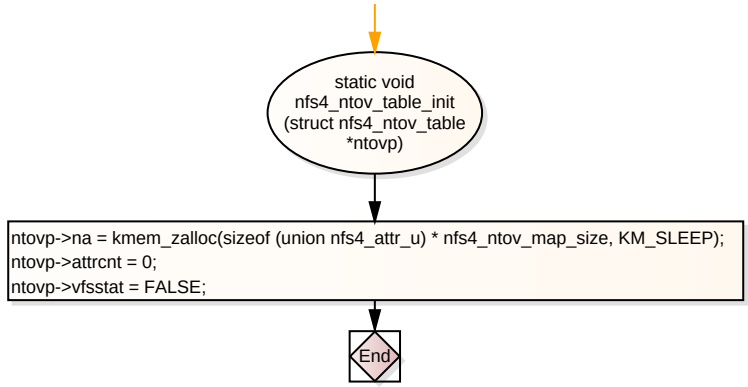

int is_exported_sec(int flavor, struct exportinfo *exi)
* Check to see if a given "flavor" is an explicitly shared flavor.
* The assumption of this routine is the "flavor" is already a valid
* flavor in the secinfo list of "exi".
*
* e.g.
* # share -o sec=flavor1 /export
* # share -o sec=flavor2 /export/home
*
* flavor2 is not an explicitly shared flavor for /export,
* however it is in the secinfo list for /export thru the
* server namespace setup.
int i;
struct secinfo *sp;
sp = exi->exi_export.ex_secinfo;
i = 0
i < exi->exi_export .ex_seccnt ?
Should not reach this point based on the assumption
Yes
No
flavor == sp[i].s_secinfo.sc_nfsnum || sp[i].s_secinfo.sc_ nfsnum == AUTH_NONE ?
return (0);
No
Yes
i++
return (SEC_REF_ EXPORTED(&sp[i]));
End

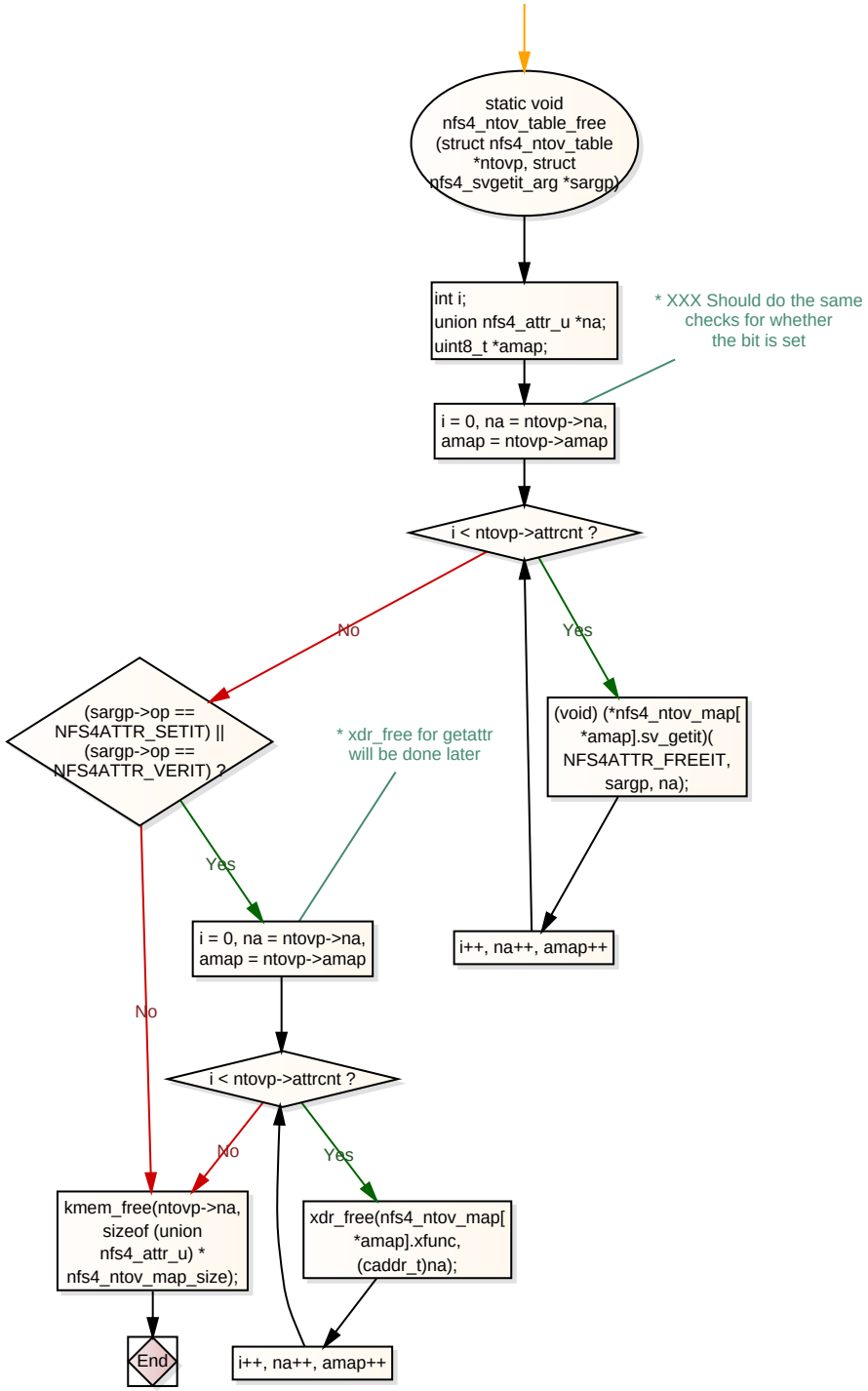

static int
secinfo_match_or_authnone
(struct compound_state
*cs)
* Check if the security flavor used in the request matches what is
* required at the export point or at the root pseudo node (exi_root).
*
* returns 1 if there's a match or if exported with AUTH_NONE; 0 otherwise.
*
int i;
struct secinfo *sp;
* Check cs->nfsflavor (from the request) against
* the current export data in cs->exi.
sp = cs->exi->exi_export.ex_secinfo;
i = 0
i < cs->exi->exi_
export.ex_seccnt ?
Yes
No
cs->nfsflavor ==
sp[i].s_secinfo.sc_nfsnum
|| sp[i].s_secinfo.sc_
nfsnum == AUTH_NONE ?
Yes
No
return (0);
return (1);
i++
End

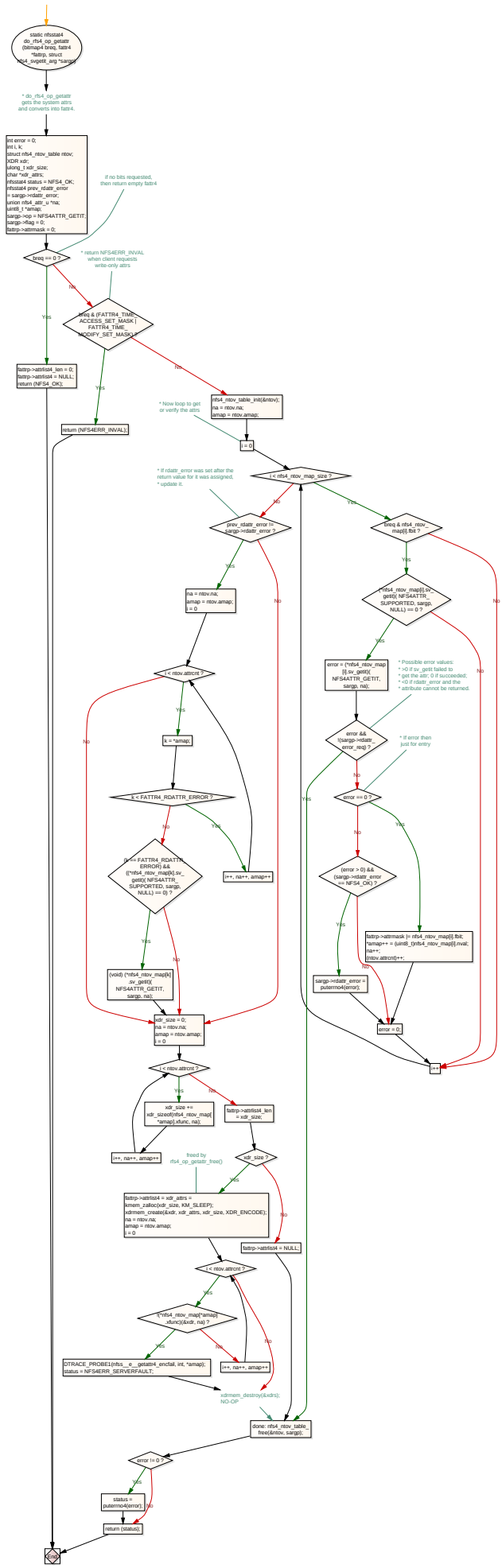

nfsstat4
call_checkauth4(struct
compound_state *cs,
struct svc_req *req)
* Check the access
authority for the client
and return the
correct error.
int authres;
* First, check if the security flavor used in the request
* are among the flavors set in the server namespace.
!secinfo_match_
or_authnone(cs) ?
No
Yes
authres =
checkauth4(cs, req);
*cs->statusp = NFS4ERR_WRONGSEC;
return (*cs->statusp);
authres > 0 ?
Yes
No
*cs->statusp = NFS4_OK;
authres == 0 ?
Yes
No
! (cs->access &
CS_ACCESS_LIMITED) ?
authres == -2 ?
Yes
No
*cs->statusp =
NFS4ERR_ACCESS;
Yes
No
cs->access =
CS_ACCESS_OK;
*cs->statusp =
NFS4ERR_WRONGSEC;
*cs->statusp =
NFS4ERR_DELAY;
No
return (*cs->statusp);
End

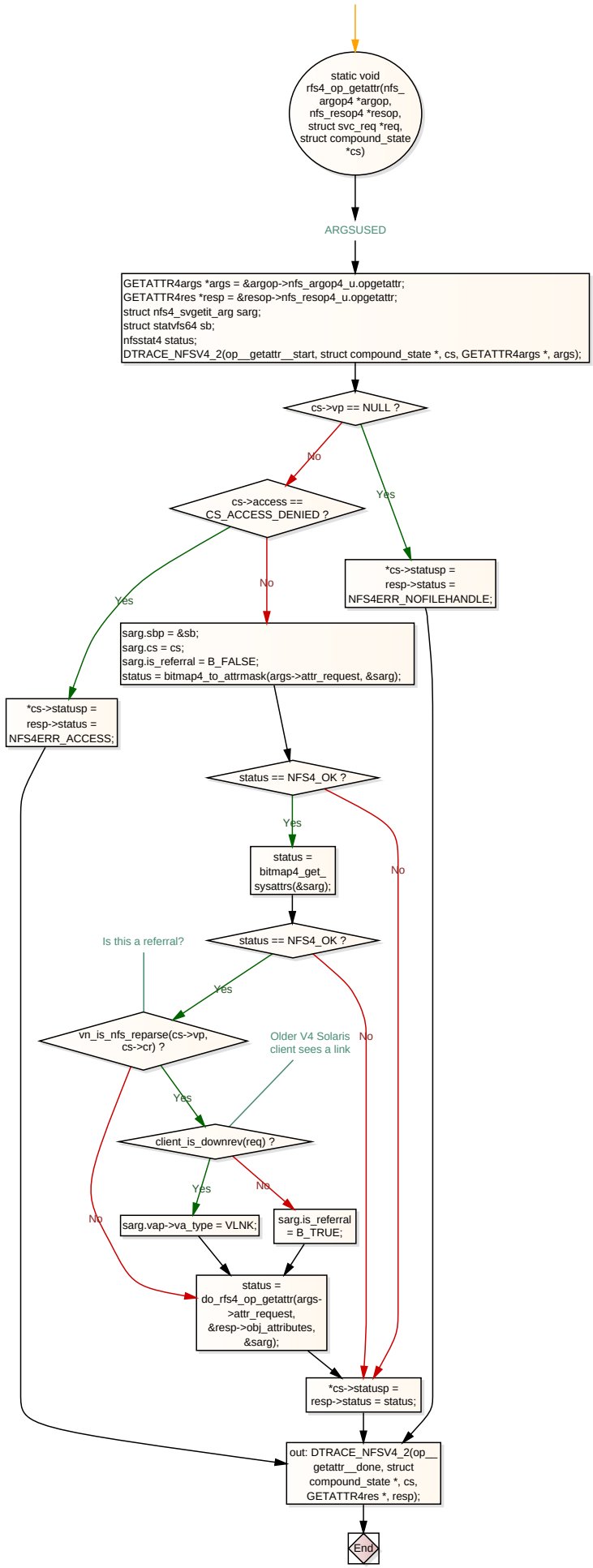

static nfsstat4 bitmap4_to_attrmask (bitmap4 breq, struct nfs4_svgetit_arg *sargp)
* bitmap4_to_attrmask is called by getattr and readdir.
* It sets up the vattr mask and determines whether vfsstat call is needed
* based on the input bitmap.
* Returns nfsv4 status.
int i;
uint_t va_mask;
struct statvfs64 *sbp = sargp->sbp;
sargp->sbp = NULL;
sargp->flag = 0;
sargp->rdattr_error = NFS4_OK;
sargp->mntdfid_set = FALSE;
sargp->cs->vp ?
Yes
No
sargp->xattr = get_fh4_flag(&sargp->cs->fh, FH4_ATTRDIR | FH4_NAMEDATTR);
sargp->xattr = 0;
* Set rdattr_error_req to true if return error per
* failed entry rather than fail the readdir.
breq & FATTR4_RDATTR_ERROR_MASK ?
Yes
No
sargp->rdattr_error_req = 1;
sargp->rdattr_error_req = 0;
* generate the va_mask
* Handle the easy cases first
switch (breq)
NFS4_NTOV_ATTR_MASK ?
No
Yes
NFS4_FS_ATTR_MASK ?
sargp->vap->va_mask = NFS4_NTOV_ATTR_AT_MASK;
return (NFS4_OK);
Yes
No
NFS4_NTOV_ATTR_CACHE_MASK ?
sargp->vap->va_mask = NFS4_FS_ATTR_AT_MASK;
sargp->sbp = sbp;
return (NFS4_OK);
No
Yes
FATTR4_LEASE_TIME_MASK ?
sargp->vap->va_mask = NFS4_NTOV_ATTR_CACHE_AT_MASK;
return (NFS4_OK);
Yes
default
sargp->vap->va_mask = 0;
return (NFS4_OK);
va_mask = 0;
i = 0
* Check is vfsstat is needed
i < nfs4_ntov_map_size ?
No
Yes
breq & NFS4_FS_ATTR_MASK ?
(breq & nfs4_ntov_map[i].fbit) && nfs4_ntov_map[i].vbit ?
Yes
No
Yes
No
sargp->sbp = sbp;
va_mask |= nfs4_ntov_map[i].vbit;
sargp->vap->va_mask = va_mask;
return (NFS4_OK);
i++
NOTREACHED
End

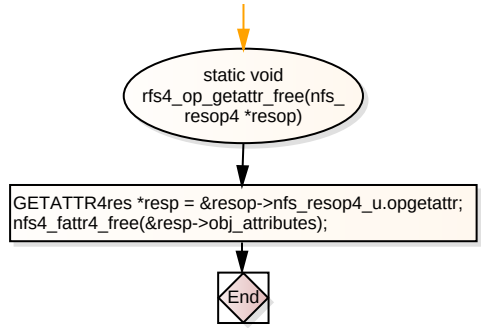

static nfsstat4 bitmap4_get_sysattrs (struct nfs4_svgetit_arg *sargp)
* bitmap4_get_sysattrs is called by getattr and readdir.
* It calls both VOP_GETATTR and VFS_STATVFS calls to get the attrs.
* Returns nfsv4 status.
int error;
struct compound_state *cs = sargp->cs;
vnode_t *vp = cs->vp;
sargp->sbp != NULL ?
No
Yes
error = VFS_STATVFS(vp->v_vfsp, sargp->sbp) ?
to identify error
No
Yes
return (rfs4_vop_getattr (vp, sargp->vap, 0, cs->cr));
sargp->sbp = NULL;
return (puterrno4(error));
End

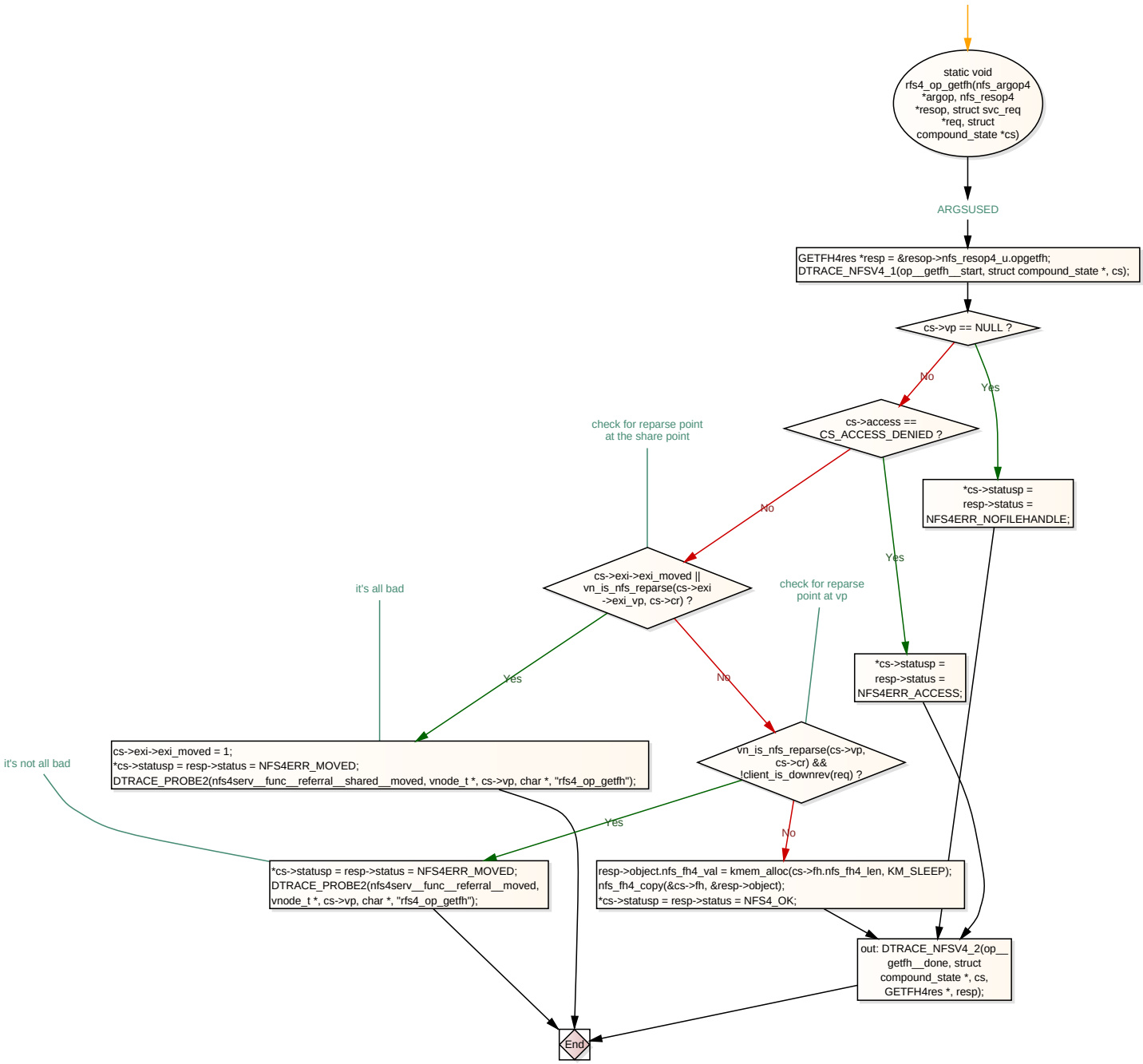

static void
nfs4_ntov_table_init
(struct nfs4_ntov_table
*ntovp)
ntovp->na = kmem_zalloc(sizeof (union nfs4_attr_u) * nfs4_ntov_map_size, KM_SLEEP);
ntovp->attrcnt = 0;
ntovp->vfsstat = FALSE;
End

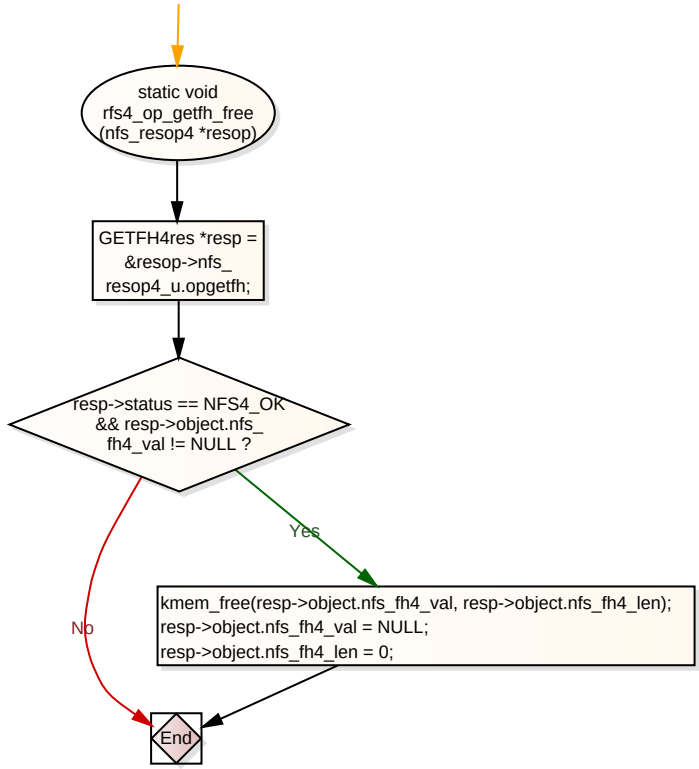

static void nfs4_ntov_table_free (struct nfs4_ntov_table *ntovp, struct nfs4_svgetit_arg *sargp)
int i;
union nfs4_attr_u *na;
uint8_t *amap;
* XXX Should do the same checks for whether the bit is set
i = 0, na = ntovp->na, amap = ntovp->amap
i < ntovp->attrcnt ?
No
Yes
(sargp->op == NFS4ATTR_SETIT) || (sargp->op == NFS4ATTR_VERIT) ?
* xdr_free for getattr will be done later
(void) (*nfs4_ntov_map[*amap].sv_getit)(NFS4ATTR_FREEIT, sargp, na);
Yes
i = 0, na = ntovp->na, amap = ntovp->amap
i++, na++, amap++
No
i < ntovp->attrcnt ?
No
Yes
kmem_free(ntovp->na, sizeof (union nfs4_attr_u) * nfs4_ntov_map_size);
xdr_free(nfs4_ntov_map[*amap].xfunc, (caddr_t)na);
End
i++, na++, amap++

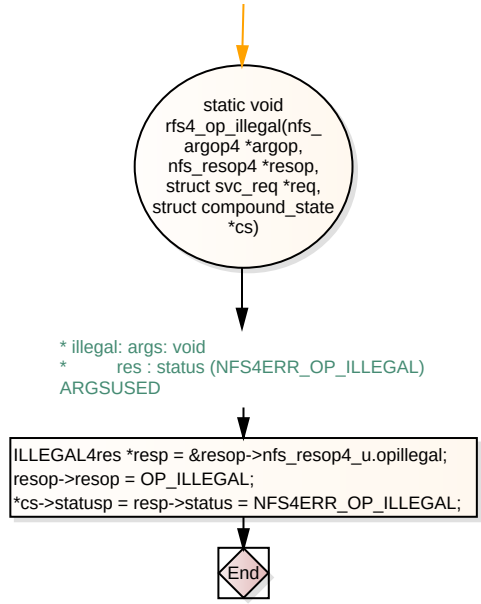

static nfsstat4 do_nfs4_op_getattr (bitmap4 breq, fattr4 *fattrp, struct nfs4_svgetit_arg *sargp)
* do_nfs4_op_getattr gets the system attrs and converts into fattr4.
int error = 0;
int i, k;
struct nfs4_ntov_table ntov;
XDR xdr;
ulong_t xdr_size;
char *xdr_attrs;
nfsstat4 status = NFS4_OK;
nfsstat4 prev_rdattr_error = sargp->rdattr_error;
union nfs4_attr_u *na;
uint8_t *amap;
sargp->op = NFS4ATTR_GETIT;
sargp->flag = 0;
fattrp->attrmask = 0;
if no bits requested, then return empty fattr4
breq == 0 ?
* return NFS4ERR_INVAL when client requests write-only attrs
breq & (FATTR4_TIME_ACCESS_SET_MASK | FATTR4_TIME_MODIFY_SET_MASK) ?
fattrp->attrlist4_len = 0; fattrp->attrlist4 = NULL; return (NFS4_OK);
return (NFS4ERR_INVAL);
* Now loop to get or verify the attrs
nfs4_ntov_table_init(&ntov); na = ntov.na; amap = ntov.amap;
i = 0
i < nfs4_ntov_map_size ?
* If rdattr_error was set after the return value for it was assigned, * update it.
prev_rdattr_error != sargp->rdattr_error ?
breq & nfs4_ntov_map[i].fbit ?
na = ntov.na; amap = ntov.amap; k = 0
(*nfs4_ntov_map[i].sv_getit)(NFS4ATTR_SUPPORTED, sargp, NULL) == 0 ?
k < ntov.attrcnt ?
error = (*nfs4_ntov_map[i].sv_getit)(NFS4ATTR_GETIT, sargp, na);
* Possible error values: * >0 if sv_getit failed to * get the attr; 0 if succeeded; * <0 if rdattr_error and the * attribute cannot be returned.
k = *amap;
error && !(sargp->rdattr_error_req) ?
* If error then just for entry
k < FATTR4_RDATTR_ERROR ?
error == 0 ?
i++, na++, amap++
(k >= FATTR4_RDATTR_ERROR) && ((*nfs4_ntov_map[k].sv_getit)(NFS4ATTR_SUPPORTED, sargp, NULL) == 0) ?
(error > 0) && (sargp->rdattr_error == NFS4_OK) ?
fattrp->attrmask |= nfs4_ntov_map[i].fbit; *amap++ = (uint8_t)nfs4_ntov_map[i].nval; na++; (ntov.attrcnt)++;
(void) (*nfs4_ntov_map[k].sv_getit)(NFS4ATTR_GETIT, sargp, na);
sargp->rdattr_error = puterrno4(error);
xdr_size = 0; na = ntov.na; amap = ntov.amap; i = 0
error = 0;
i++
i < ntov.attrcnt ?
xdr_size += xdr_sizeof(nfs4_ntov_map[*amap].sfunc, na);
fattrp->attrlist4_len = xdr_size;
i++, na++, amap++
freed by rfs4_op_getattr_free()
xdr_size ?
fattrp->attrlist4 = xdr_attrs = kmem_zalloc(xdr_size, KM_SLEEP); xdrmem_create(&xdr, xdr_attrs, xdr_size, XDR_ENCODE); na = ntov.na; amap = ntov.amap; i = 0
fattrp->attrlist4 = NULL;
i < ntov.attrcnt ?
!(*nfs4_ntov_map[*amap].sfunc)(&xdr, na) ?
DTRACE_PROBE1(nfss__e__getattr4_encfail, int, *amap); status = NFS4ERR_SERVERFAULT;
i++, na++, amap++
xdrmem_destroy(&xdrs); NO-OP
done: nfs4_ntov_table_free(&ntov, sargp);
error != 0 ?
status = puterrno4(error);
return (status);
End

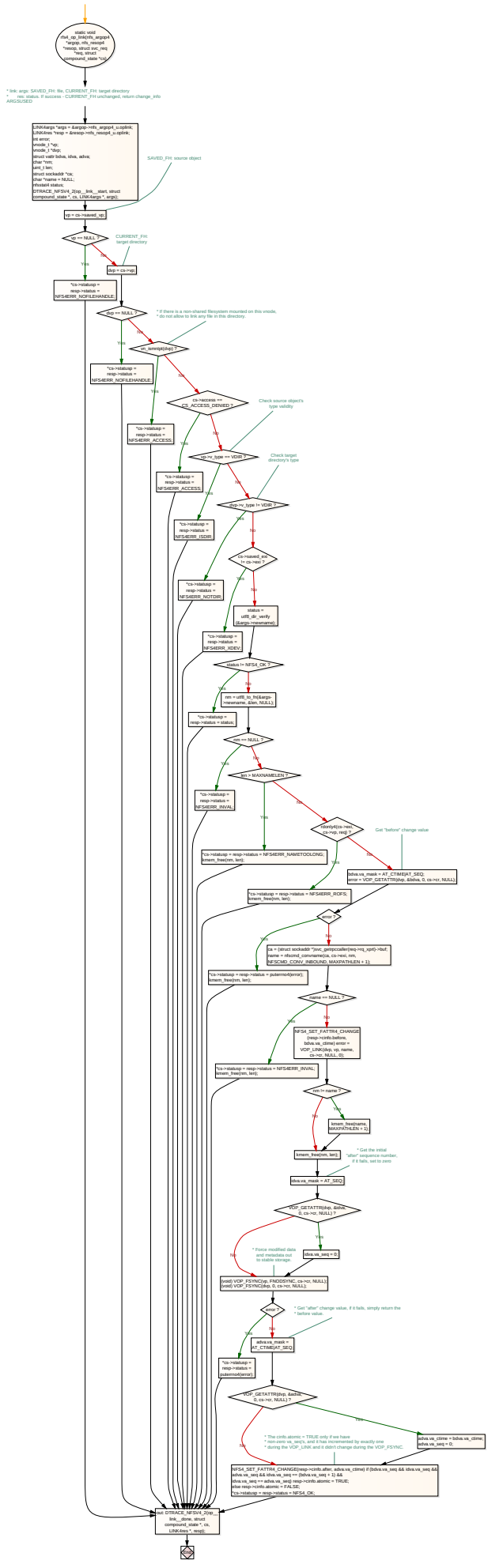

static void rfs4_op_getattr(nfs_ argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
GETATTR4args *args = &argop->nfs_argop4_u.opgetattr;
GETATTR4res *resp = &resop->nfs_resop4_u.opgetattr;
struct nfs4_svgetit_arg sarg;
struct statvfs64 sb;
nfsstat4 status;
DTRACE_NFSV4_2(op__getattr__start, struct compound_state *, cs, GETATTR4args *, args);
cs->vp == NULL ?
No
Yes
cs->access == CS_ACCESS_DENIED ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
sarg.sbp = &sb;
sarg.cs = cs;
sarg.is_referral = B_FALSE;
status = bitmap4_to_attrmask(args->attr_request, &sarg);
*cs->statusp = resp->status = NFS4ERR_ACCESS;
status == NFS4_OK ?
Yes
No
status = bitmap4_get_ sysattrs(&sarg);
Is this a referral?
status == NFS4_OK ?
Yes
No
vn_is_nfs_reparse(cs->vp, cs->cr) ?
Older V4 Solaris client sees a link
Yes
No
client_is_downrev(req) ?
Yes
No
No
sarg.vap->va_type = VLNK;
sarg.is_referral = B_TRUE;
status = do_rfs4_op_getattr(args->attr_request, &resp->obj_attributes, &sarg);
*cs->statusp = resp->status = status;
out: DTRACE_NFSV4_2(op__ getattr__done, struct compound_state *, cs, GETATTR4res *, resp);
End

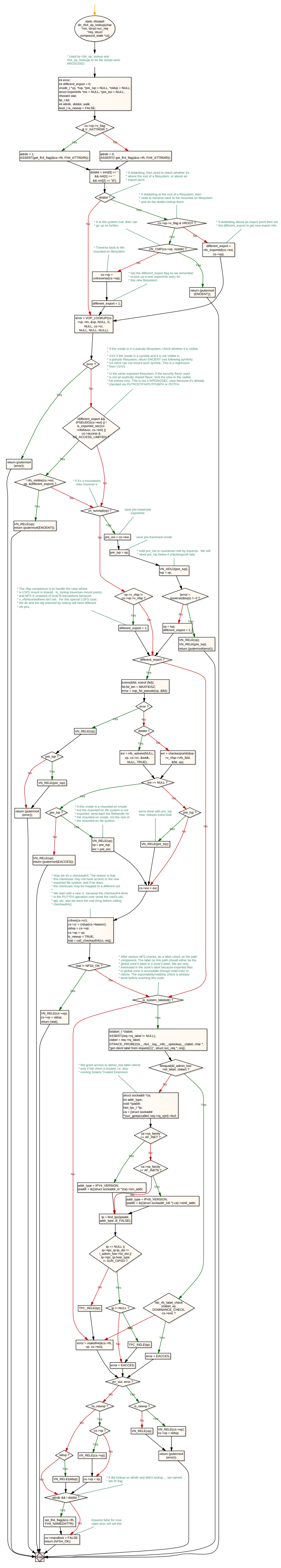

static void
rfs4_op_getattr_free(nfs_
resop4 *resop)
GETATTR4res *resp = &resop->nfs_resop4_u.opgetattr;
nfs4_fattr4_free(&resp->obj_attributes);
End


static void rfs4_op_getfh(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
GETFH4res *resp = &resop->nfs_resop4_u.opgetfh;
DTRACE_NFSV4_1(op__getfh__start, struct compound_state *, cs);
cs->vp == NULL ?
No
Yes
cs->access == CS_ACCESS_DENIED ?
check for reparse point at the share point
No
Yes
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
cs->exi->exi_moved || vn_is_nfs_reparse(cs->exi ->exi_vp, cs->cr) ?
it's all bad
check for reparse point at vp
Yes
No
*cs->statusp = resp->status = NFS4ERR_ACCESS;
it's not all bad
cs->exi->exi_moved = 1;
*cs->statusp = resp->status = NFS4ERR_MOVED;
DTRACE_PROBE2(nfs4serv__func__referral__shared__moved, vnode_t *, cs->vp, char *, "rfs4_op_getfh");
vn_is_nfs_reparse(cs->vp, cs->cr) && !client_is_downrev(req) ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_MOVED;
DTRACE_PROBE2(nfs4serv__func__referral__moved, vnode_t *, cs->vp, char *, "rfs4_op_getfh");
resp->object.nfs_fh4_val = kmem_alloc(cs->fh.nfs_fh4_len, KM_SLEEP);
nfs_fh4_copy(&cs->fh, &resp->object);
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__ getfh__done, struct compound_state *, cs, GETFH4res *, resp);
End

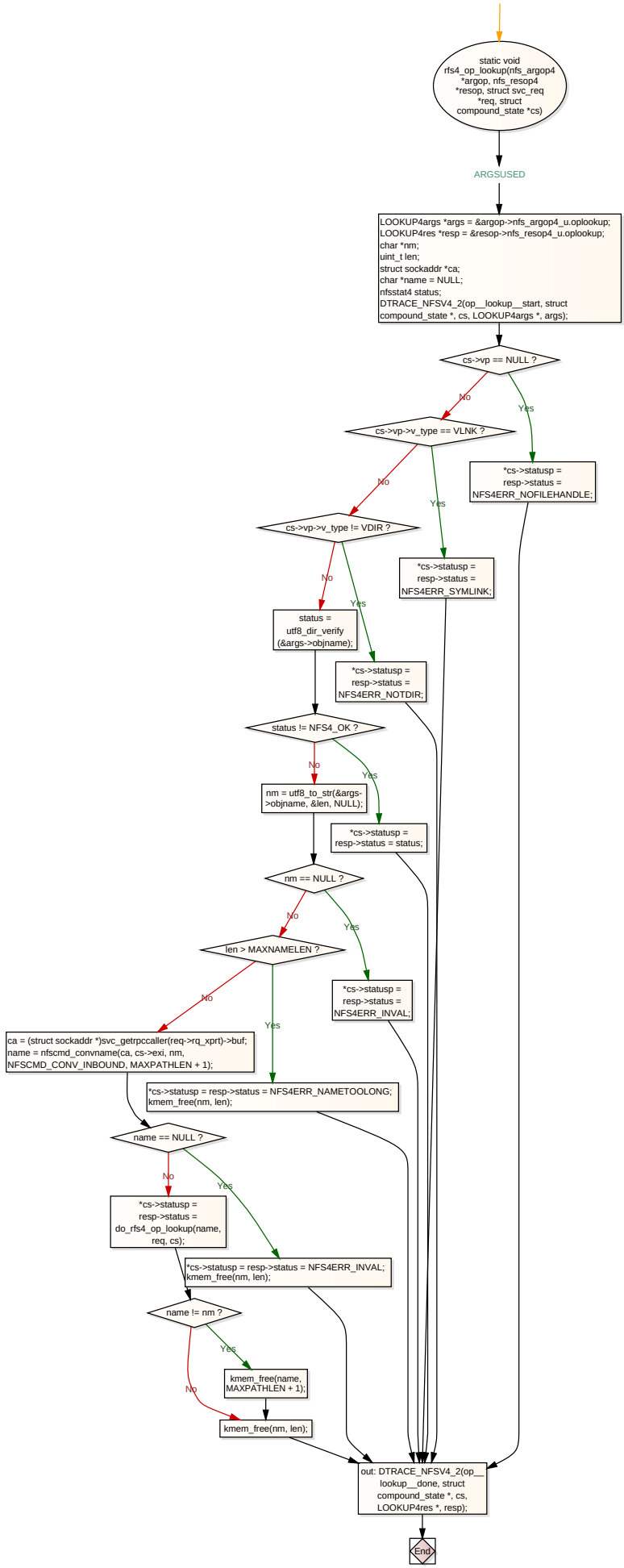

static void
rfs4_op_getfh_free
(nfs_resop4 *resop)
GETFH4res *resp =
&resop->nfs_
resop4_u.opgetfh;
resp->status == NFS4_OK
&& resp->object.nfs_
fh4_val != NULL ?
Yes
No
kmem_free(resp->object.nfs_fh4_val, resp->object.nfs_fh4_len);
resp->object.nfs_fh4_val = NULL;
resp->object.nfs_fh4_len = 0;
End

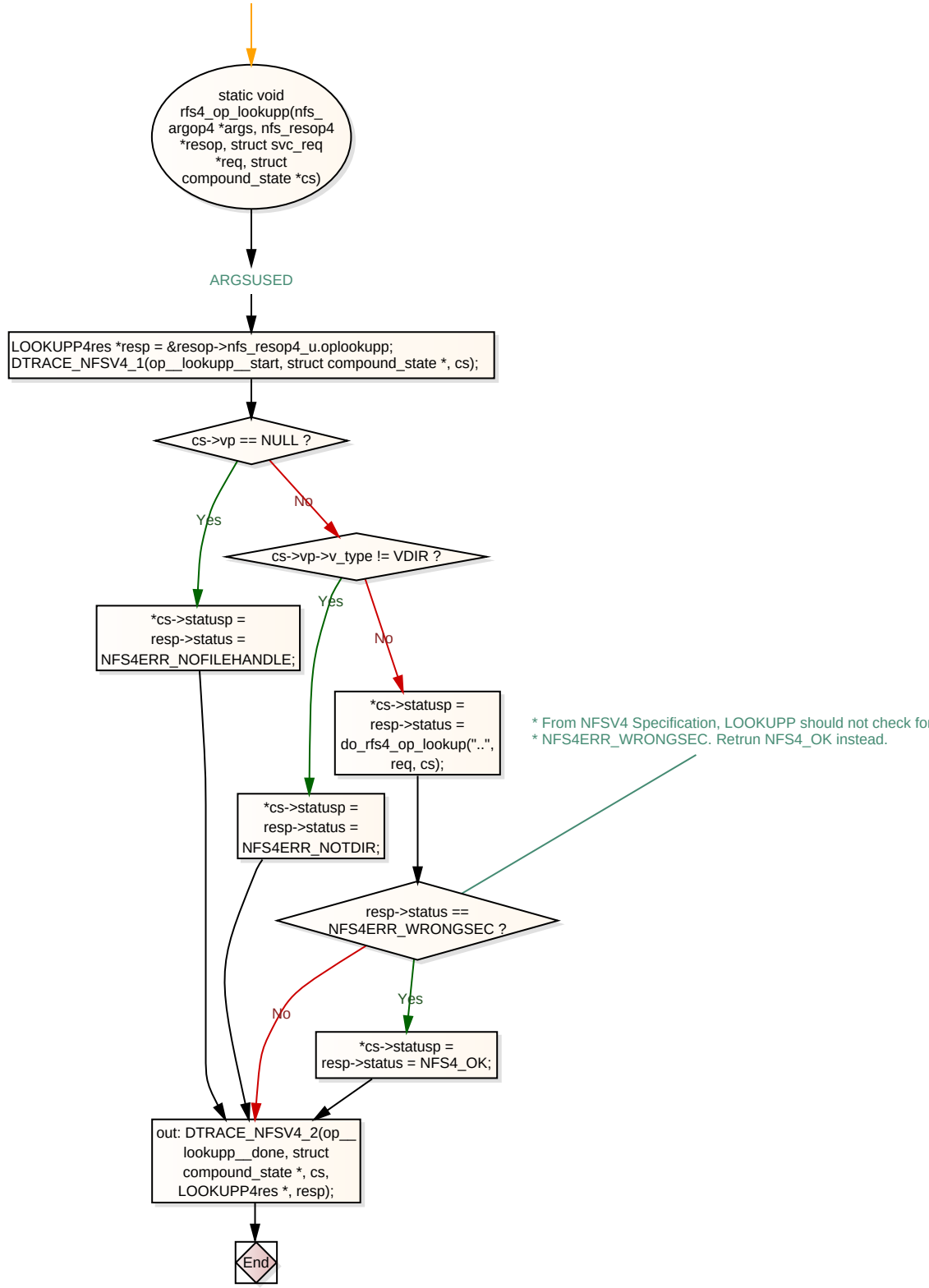

static void
rfs4_op_illegal(nfs_
argop4 *argop,
nfs_resop4 *resop,
struct svc_req *req,
struct compound_state
*cs)
* illegal: args: void
* res : status (NFS4ERR_OP_ILLEGAL)
ARGSUSED
ILLEGAL4res *resp = &resop->nfs_resop4_u.opillegal;
resop->resop = OP_ILLEGAL;
*cs->statusp = resp->status = NFS4ERR_OP_ILLEGAL;
End

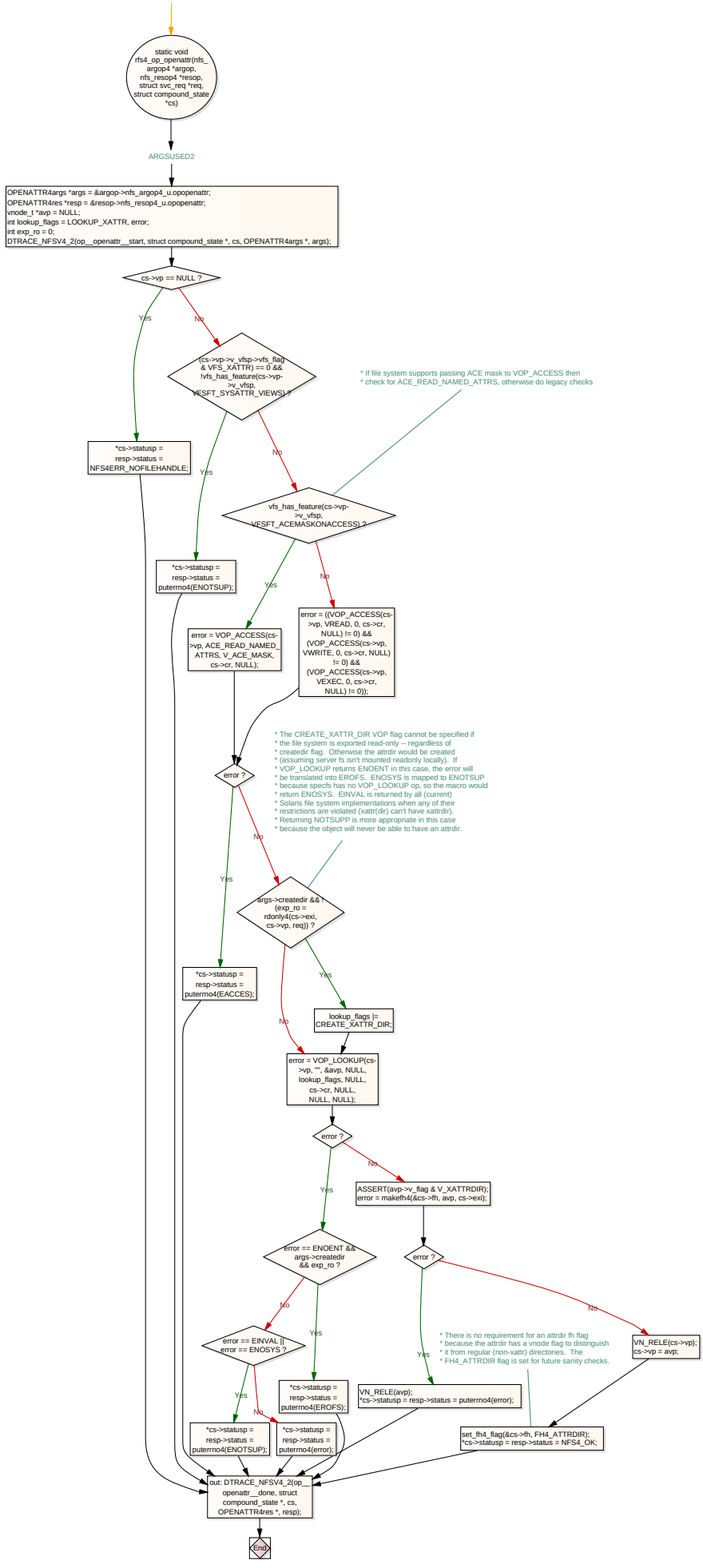

static void
nfs4_op_link(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* link: args: SAVED_FH: file, CURRENT_FH: target directory
* resc status. If success - CURRENT_FH unchanged, return change_info
ARGSUSED
LINK4args *args = &argop->nfs_argop4_u.oplink;
LINK4res *resp = &resop->nfs_resop4_u.oplink;
int error;
vnode_t *vp;
vnode_t *dvp;
struct vattr bdva, idva, adva;
char *nm;
uint_t len;
struct sockaddr *ca;
char *name = NULL;
nfsstat4 status;
DTRACE_NFSV4_2(op__link__start, struct compound_state *, cs, LINK4args *, args);
SAVED_FH: source object
vp = cs->saved_vp;
vp == NULL ?
CURRENT_FH: target directory
dvp = cs->vp;
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
dvp == NULL ?
* If there is a non-shared filesystem mounted on this vnode,
* do not allow to link any file in this directory.
vn_ismntpt(dvp) ?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE
cs->access == CS_ACCESS_DENIED ?
Check source object's type validity
*cs->statusp = resp->status = NFS4ERR_ACCESS;
vp->v_type == VDIR ?
Check target directory's type
*cs->statusp = resp->status = NFS4ERR_ACCESS;
dvp->v_type != VDIR ?
*cs->statusp = resp->status = NFS4ERR_ISDIR;
cs->saved_vp != cs->exi ?
*cs->statusp = resp->status = NFS4ERR_NOTDIR;
status = utf8_dir_verify (&args->newname);
*cs->statusp = resp->status = NFS4ERR_XDEV;
status != NFS4_OK ?
nm = utf8_to_fn(&args->newname, &len, NULL);
*cs->statusp = resp->status = status;
nm == NULL ?
len > MAXNAMELEN ?
*cs->statusp = resp->status = NFS4ERR_INVAL;
(dontyet(cs->exi, cs->vp, req) ?
Get "before" change value
*cs->statusp = resp->status = NFS4ERR_NAMETOOLONG; kmem_free(nm, len);
bdva.va_mask = AT_CTIME|AT_SEQ; error = VOP_GETATTR(dvp, &bdva, 0, cs->cr, NULL);
*cs->statusp = resp->status = NFS4ERR_ROFS; kmem_free(nm, len);
error ?
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf; name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
*cs->statusp = resp->status = puterrno4(error); kmem_free(nm, len);
name == NULL ?
NFS4_SET_FATTR4_CHANGE (resp->cinfo.before, bdva.va_ctime) error = VOP_LINK(dvp, vp, name, cs->cr, NULL, 0);
*cs->statusp = resp->status = NFS4ERR_INVAL; kmem_free(nm, len);
nm != name ?
kmem_free(name, MAXPATHLEN + 1);
kmem_free(nm, len);
* Get the initial "after" sequence number, if it fails, set to zero
adva.va_mask = AT_SEQ;
VOP_GETATTR(dvp, &adva, 0, cs->cr, NULL) ?
bdva.va_seq = 0;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(vp, FNODSYNC, cs->cr, NULL); (void) VOP_FSYNC(dvp, 0, cs->cr, NULL);
error ?
* Get "after" change value, if it fails, simply return the * before value.
adva.va_mask = AT_CTIME|AT_SEQ;
*cs->statusp = resp->status = puterrno4(error);
VOP_GETATTR(dvp, &adva, 0, cs->cr, NULL) ?
bdva.va_ctime = bdva.va_ctime; bdva.va_seq = 0;
* The cinfo.atomic = TRUE only if we have
* non-zero va_seq's, and it has incremented by exactly one
* during the VOP_LINK and it didn't change during the VOP_FSYNC.
NFS4_SET_FATTR4_CHANGE(resp->cinfo.after, adva.va_ctime) if (bdva.va_seq && adva.va_seq && bdva.va_seq + 1 == adva.va_seq && bdva.va_seq == (bdva.va_seq + 1) && adva.va_seq == adva.va_seq) resp->cinfo.atomic = TRUE; else resp->cinfo.atomic = FALSE; *cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__link__done, struct compound_state *, cs, LINK4res *, resp);

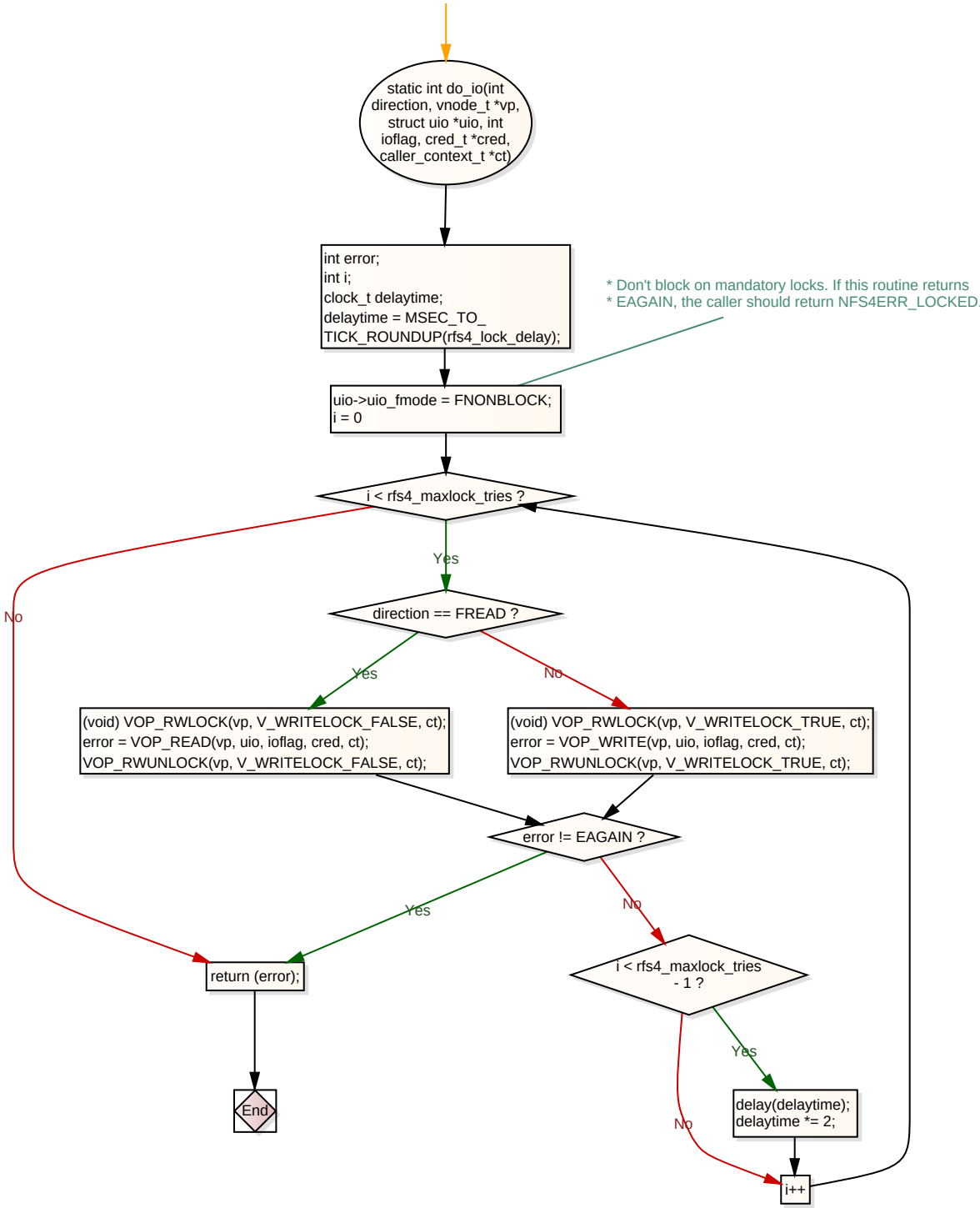


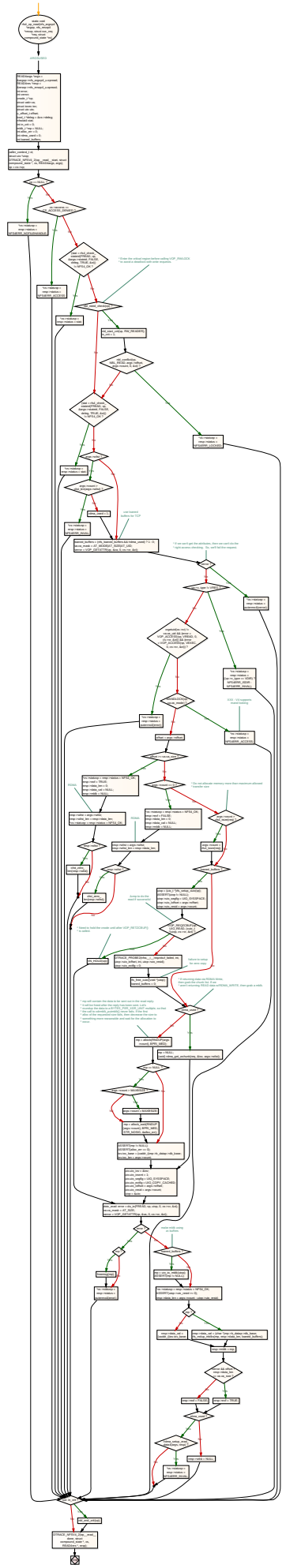
static nfsstat4 do_rfs4_op_lookup(char *nm, struct svc_req *req, struct compound_state *cs)
* Used by rfs4_op_lookup and rfs4_op_lookupp to do the actual work. ARGSUSED
int error;
int different_export = 0;
vnode_t *vp, *tvp, *pre_tvp = NULL, *oldvp = NULL;
struct exportinfo *exi = NULL, *pre_exi = NULL;
nfsstat4 stat;
fid_t fid;
int attrdir, dotdot, walk;
bool_t is_newvp = FALSE;
cs->vp->v_flag & V_XATTRDIR ?
attrdir = 1; ASSERT(get_fh4_flag(&cs->fh, FH4_ATTRDIR));
attrdir = 0; ASSERT(! get_fh4_flag(&cs->fh, FH4_ATTRDIR));
dotdot = (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0');
* If dotdotting, then need to check whether it's * above the root of a filesystem, or above an * export point.
dotdot ?
* If dotdotting at the root of a filesystem, then * need to traverse back to the mounted-on filesystem * and do the dotdot lookup there.
cs->vp->v_flag & VROOT ?
* If at the system root, then can * go up no further.
* If dotdotting above an export point then set * the different_export to get new export info.
VN_CMP(cs->vp, rootdir) ?
different_export = nfs_exported(cs->exi, cs->vp);
* Traverse back to the mounted-on filesystem
cs->vp = untraverse(cs->vp);
* Set the different_export flag so we remember * to pick up a new exportinfo entry for * this new filesystem.
return (puterrno4 (ENOENT));
different_export = 1;
error = VOP_LOOKUP(cs->vp, nm, &vp, NULL, 0, NULL, cs->cr, NULL, NULL, NULL);
* If the vnode is in a pseudo filesystem, check whether it is visible. * * XXX if the vnode is a symlink and it is not visible in * a pseudo filesystem, return ENOENT (not following symlink). * V4 client can not mount such symlink. This is a regression * from V2/V3. * * In the same exported filesystem, if the security flavor used * is not an explicitly shared flavor, limit the view to the visible * list entries only. This is not a WRONGSEC case because it's already * checked via PUTROOTFH/PUTPUBFH or PUTFH.
error ?
!different_export && (PSEUDO(cs->exi) || ! is_exported_sec(cs->nfsflavor, cs->exi) || cs->access & CS_ACCESS_LIMITED) ?
return (puterrno4 (error));
! nfs_visible(cs->exi, vp, &different_export) ?
* If it's a mountpoint, then traverse it.
vn_ismntpt(vp) ?
save pre-traversed exportinfo
VN_RELE(vp); return (puterrno4(ENOENT));
pre_exi = cs->exi;
save pre-traversed vnode
pre_tvp = vp;
* hold pre_tvp to counteract rele by traverse. We will * need pre_tvp below if checkexport4 fails
VN_HOLD(pre_tvp); tvp = vp;
* The vfsp comparison is to handle the case where * a LOFS mount is shared. lo_lookup traverses mount points, * and NFS is unaware of local fs transistions because * v_vfsmountedhere isn't set. For this special LOFS case, * the dir and the obj returned by lookup will have different * vfs ptrs.
vp->v_vfsp != cs->vp->v_vfsp ?
(error = traverse(&tvp)) != 0 ?
different_export = 1;
vp = tvp; different_export = 1;
VN_RELE(vp); VN_RELE(pre_tvp); return (puterrno4(error));
different_export ?
bzero(&fid, sizeof (fid)); fid.fid_len = MAXFIDSZ; error = vop_fid_pseudo(vp, &fid);
error ?
VN_RELE(vp);
dotdot ?
pre_tvp ?
exi = nfs_vptoexi(NULL, vp, cs->cr, &walk, NULL, TRUE);
exi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);
VN_RELE(pre_tvp);
exi == NULL ?
return (puterrno4 (error));
pre_tvp ?
* If this vnode is a mounted-on vnode, * but the mounted-on file system is not * exported, send back the filehandle for * the mounted-on vnode, not the root of * the mounted-on file system.
we're done with pre_tvp now. release extra hold
pre_tvp ?
VN_RELE(vp); vp = pre_tvp; exi = pre_exi;
VN_RELE(pre_tvp);
VN_RELE(vp); return (puterrno4(EACCES));
cs->exi = exi;
* Now we do a checkauth4. The reason is that * this client/user may not have access to the new * exported file system, and if he does, * the client/user may be mapped to a different uid. * * We start with a new cr, because the checkauth4 done * in the PUT*FH operation over wrote the cred's uid, * gid, etc, and we want the real thing before calling * checkauth4()
crfree(cs->cr);
cs->cr = crdup(cs->basecr);
oldvp = cs->vp;
cs->vp = vp;
is_newvp = TRUE;
stat = call_checkauth4(cs, req);
stat != NFS4_OK ?
* After various NFS checks, do a label check on the path * component. The label on this path should either be the * global zone's label or a zone's label. We are only * interested in the zone's label because exported files * in global zone is accessible (though read-only) to * clients. The exportability/visibility check is already * done before reaching this code.
is_system_labeled() ?
VN_RELE(cs->vp); cs->vp = oldvp; return (stat);
bslabel_t *clabel;
ASSERT(req->rq_label != NULL);
clabel = req->rq_label;
DTRACE_PROBE2(tx__rfs4__log__info__oplookup__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
* We grant access to admin_low label clients * only if the client is trusted, i.e. also * running Solaris Trusted Extension.
struct sockaddr *ca;
int addr_type;
void *ipaddr;
tsol_tpc_t *tp;
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
ca->sa_family == AF_INET ?
ca->sa_family == AF_INET6 ?
addr_type = IPV4_VERSION; ipaddr = &((struct sockaddr_in *)ca)->sin_addr;
addr_type = IPV6_VERSION; ipaddr = &((struct sockaddr_in6 *) ca)->sin6_addr;
tp = find_tpc(ipaddr, addr_type, B_FALSE);
tp == NULL || tp->tpc_tp.tp_doi != l_admin_low->tsl_doi || tp->tpc_tp.host_type != SUN_CIPSO ?
TPC_RELE(tp);
tp != NULL ?
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, cs->exi) ?
error = makefh4(&cs->fh, vp, cs->exi);
TPC_RELE(tp);
error = EACCES;
error = EACCES;
err_out: error ?
!is_newvp ?
is_newvp ?
cs->vp ?
VN_RELE(vp);
VN_RELE(cs->vp); cs->vp = oldvp;
oldvp ?
VN_RELE(cs->vp);
return (puterrno4 (error));
VN_RELE(oldvp);
cs->vp = vp;
* if did lookup on attrdir and didn't lookup .., set named * attr fh flag
attrdir && ! dotdot ?
set_fh4_flag(&cs->fh, FH4_NAMEDATTR);
Assume false for now, open proc will set this
cs->mandlock = FALSE; return (NFS4_OK);
End

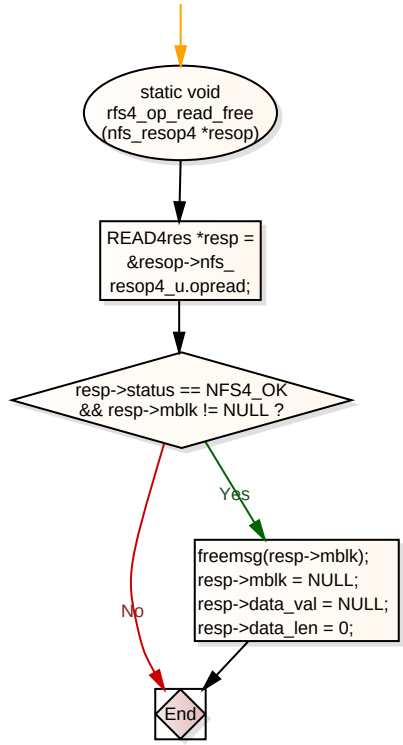

static void rfs4_op_lookup(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
LOOKUP4args *args = &argop->nfs_argop4_u.oplookup;
LOOKUP4res *resp = &resop->nfs_resop4_u.oplookup;
char *nm;
uint_t len;
struct sockaddr *ca;
char *name = NULL;
nfsstat4 status;
DTRACE_NFSV4_2(op__lookup__start, struct compound_state *, cs, LOOKUP4args *, args);
cs->vp == NULL ?
No
Yes
cs->vp->v_type == VLNK ?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
cs->vp->v_type != VDIR ?
*cs->statusp = resp->status = NFS4ERR_SYMLINK;
status = utf8_dir_verify (&args->objname);
*cs->statusp = resp->status = NFS4ERR_NOTDIR;
status != NFS4_OK ?
nm = utf8_to_str(&args->objname, &len, NULL);
*cs->statusp = resp->status = status;
nm == NULL ?
len > MAXNAMELEN ?
*cs->statusp = resp->status = NFS4ERR_INVAL;
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
*cs->statusp = resp->status = NFS4ERR_NAMETOOLONG; kmem_free(nm, len);
name == NULL ?
*cs->statusp = resp->status = do_rfs4_op_lookup(name, req, cs);
*cs->statusp = resp->status = NFS4ERR_INVAL; kmem_free(nm, len);
name != nm ?
kmem_free(name, MAXPATHLEN + 1);
kmem_free(nm, len);
out: DTRACE_NFSV4_2(op__lookup__done, struct compound_state *, cs, LOOKUP4res *, resp);
End

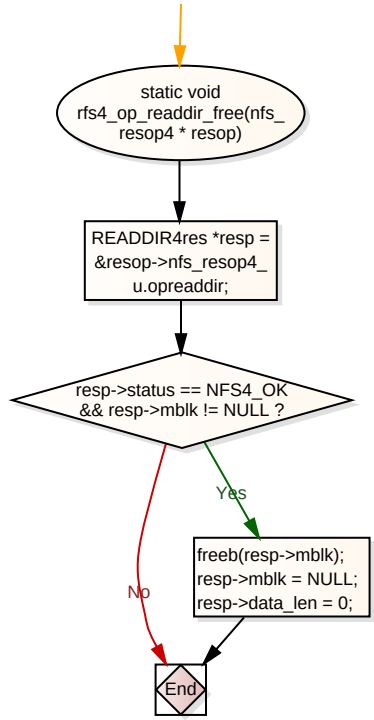

static void rfs4_op_lookupp(nfs_ argop4 *args, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
LOOKUPP4res *resp = &resop->nfs_resop4_u.oplookupp;
DTRACE_NFSV4_1(op__lookupp__start, struct compound_state *, cs);
cs->vp == NULL ?
Yes
No
cs->vp->v_type != VDIR ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
*cs->statusp = resp->status = do_rfs4_op_lookup("..", req, cs);
* From NFSV4 Specification, LOOKUPP should not check for
* NFS4ERR_WRONGSEC. Retrun NFS4_OK instead.
*cs->statusp = resp->status = NFS4ERR_NOTDIR;
resp->status == NFS4ERR_WRONGSEC ?
No
Yes
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__ lookupp__done, struct compound_state *, cs, LOOKUPP4res *, resp);
End

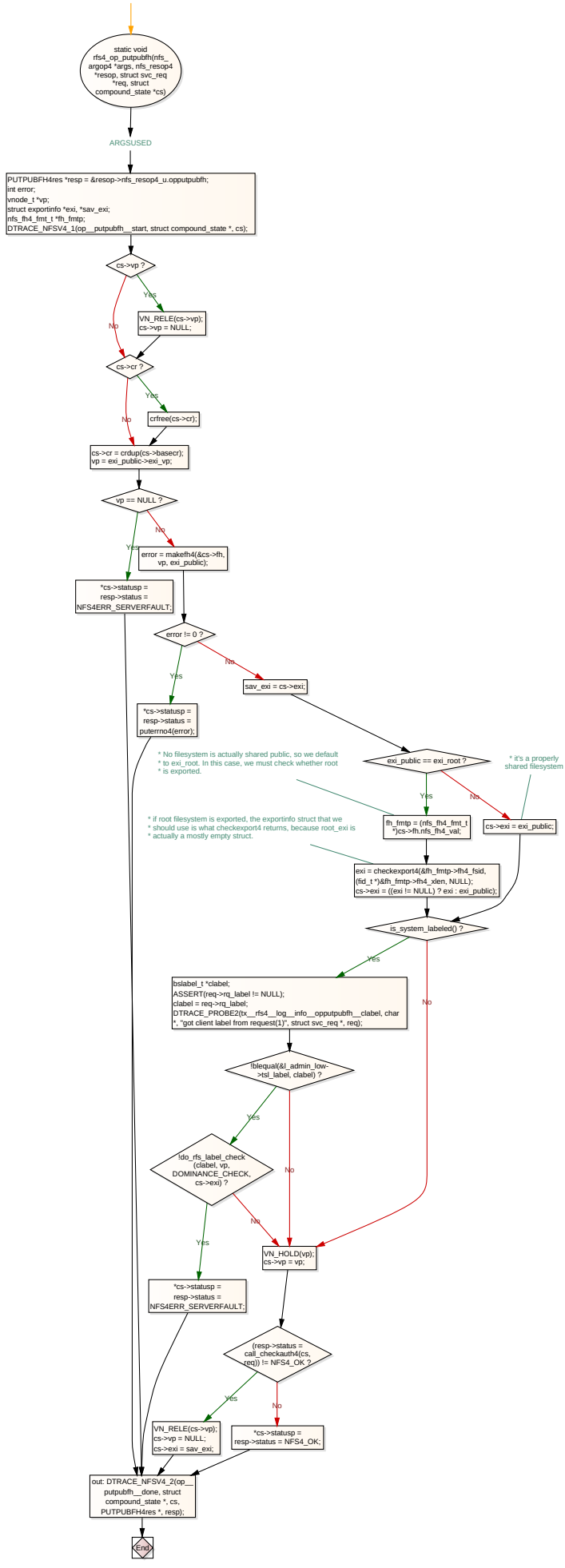

static void rfs4_op_openattr(nfs_ argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED2
OPENATTR4args *args = &argop->nfs_argop4_u.opopenattr;
OPENATTR4res *resp = &resop->nfs_resop4_u.opopenattr;
vnode_t *avp = NULL;
int lookup_flags = LOOKUP_XATTR, error;
int exp_ro = 0;
DTRACE_NFSV4_2(op__openattr__start, struct compound_state *, cs, OPENATTR4args *, args);
cs->vp == NULL ?
Yes
No
(cs->vp->v_vfsp->vfs_flag & VFS_XATTR) == 0 && !vfs_has_feature(cs->vp->v_vfsp, VFSFT_SYSATTR_VIEWS) ?
* If file system supports passing ACE mask to VOP_ACCESS then
* check for ACE_READ_NAMED_ATTRS, otherwise do legacy checks
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
vfs_has_feature(cs->vp->v_vfsp, VFSFT_ACEMASKONACCESS) ?
*cs->statusp = resp->status = puterrno4(ENOTSUP);
error = VOP_ACCESS(cs->vp, ACE_READ_NAMED_ATTRS, V_ACE_MASK, cs->cr, NULL);
error = ((VOP_ACCESS(cs->vp, VREAD, 0, cs->cr, NULL) != 0) && (VOP_ACCESS(cs->vp, VWRITE, 0, cs->cr, NULL) != 0) && (VOP_ACCESS(cs->vp, VEXEC, 0, cs->cr, NULL) != 0));
error ?
* The CREATE_XATTR_DIR VOP flag cannot be specified if
* the file system is exported read-only -- regardless of
* createdir flag. Otherwise the attrdir would be created
* (assuming server fs isn't mounted readonly locally). If
* VOP_LOOKUP returns ENOENT in this case, the error will
* be translated into EROFS. ENOSYS is mapped to ENOTSUP
* because specfs has no VOP_LOOKUP op, so the macro would
* return ENOSYS. EINVAL is returned by all (current)
* Solaris file system implementations when any of their
* restrictions are violated (xattr(dir) can't have xattrdir).
* Returning NOTSUPP is more appropriate in this case
* because the object will never be able to have an attrdir.
args->createdir && !(exp_ro = rdonly4(cs->exi, cs->vp, req)) ?
*cs->statusp = resp->status = puterrno4(EACCES);
lookup_flags |= CREATE_XATTR_DIR;
error = VOP_LOOKUP(cs->vp, "", &avp, NULL, lookup_flags, NULL, cs->cr, NULL, NULL, NULL);
error ?
ASSERT(avp->v_flag & V_XATTRDIR);
error = makefh4(&cs->fh, avp, cs->exi);
error == ENOENT && args->createdir && exp_ro ?
error ?
error == EINVAL || error == ENOSYS ?
*cs->statusp = resp->status = puterrno4(EROFS);
VN_RELE(avp);
*cs->statusp = resp->status = puterrno4(error);
* There is no requirement for an attrdir fh flag
* because the attrdir has a vnode flag to distinguish
* it from regular (non-xattr) directories. The
* FH4_ATTRDIR flag is set for future sanity checks.
VN_RELE(cs->vp);
cs->vp = avp;
*cs->statusp = resp->status = puterrno4(ENOTSUP);
*cs->statusp = resp->status = puterrno4(error);
set_fh4_flag(&cs->fh, FH4_ATTRDIR);
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__ openattr__done, struct compound_state *, cs, OPENATTR4res *, resp);
End

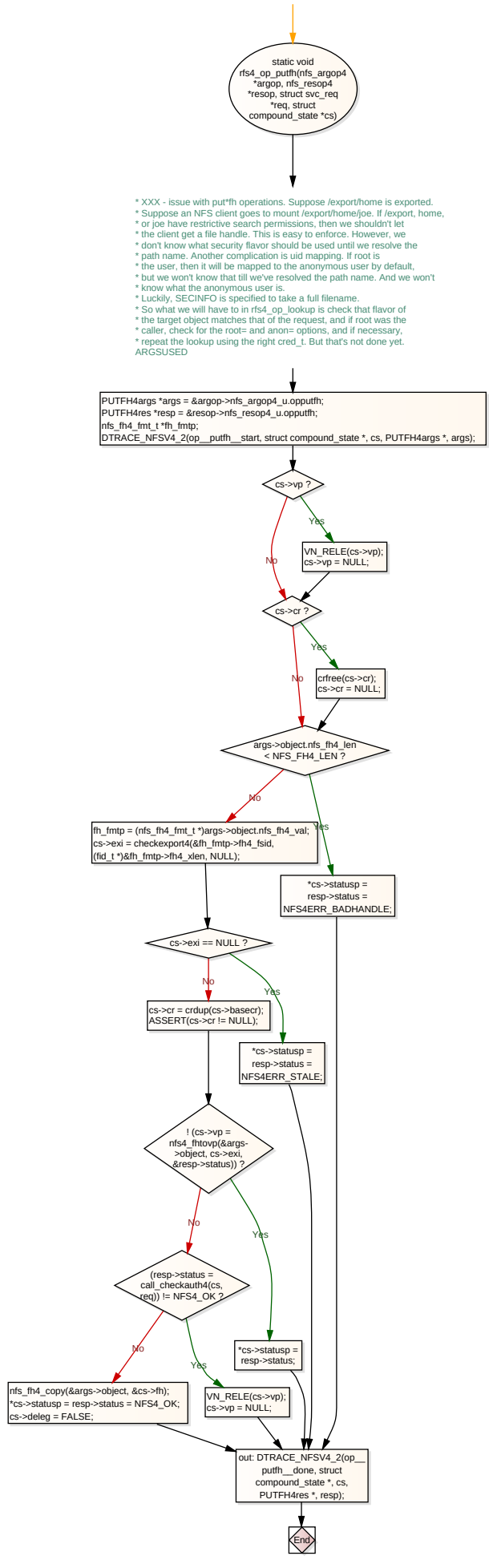

static int do_io(int direction, vnode_t *vp, struct uio *uio, int ioflag, cred_t *cred, caller_context_t *ct)
int error;
int i;
clock_t delaytime;
delaytime = MSEC_TO_TICK_ROUNDUP(rfs4_lock_delay);
* Don't block on mandatory locks. If this routine returns
* EAGAIN, the caller should return NFS4ERR_LOCKED.
uio->uio_fmode = FNONBLOCK;
i = 0
i < rfs4_maxlock_tries ?
Yes
No
direction == FREAD ?
Yes
No
(void) VOP_RWLOCK(vp, V_WRITELOCK_FALSE, ct);
error = VOP_READ(vp, uio, ioflag, cred, ct);
VOP_RWUNLOCK(vp, V_WRITELOCK_FALSE, ct);
(void) VOP_RWLOCK(vp, V_WRITELOCK_TRUE, ct);
error = VOP_WRITE(vp, uio, ioflag, cred, ct);
VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, ct);
error != EAGAIN ?
Yes
No
return (error);
i < rfs4_maxlock_tries - 1 ?
Yes
No
End
delay(delaytime);
delaytime *= 2;
i++

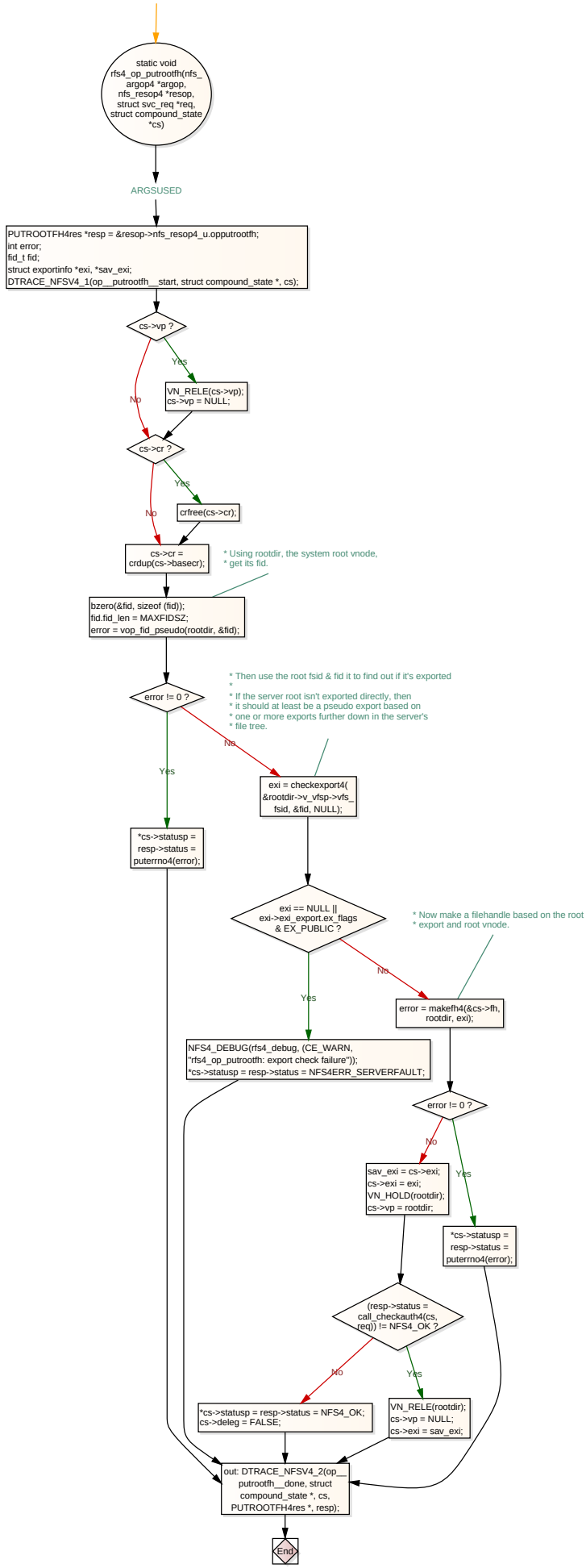


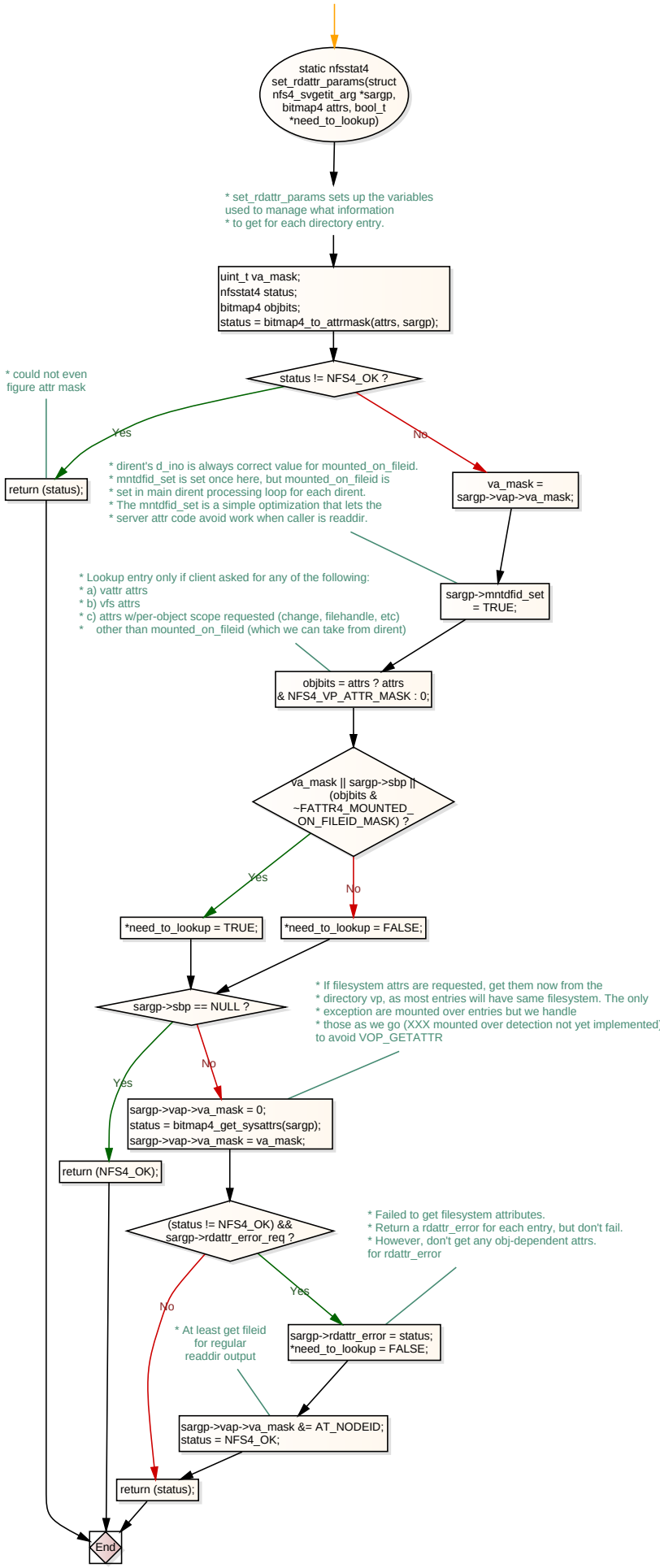

static void
rfs4_op_read_free
(nfs_resop4 *resop)
READ4res *resp =
&resop->nfs_
resop4_u.opread;
resp->status == NFS4_OK
&& resp->mblk != NULL ?
Yes
No
freemsg(resp->mblk);
resp->mblk = NULL;
resp->data_val = NULL;
resp->data_len = 0;
End

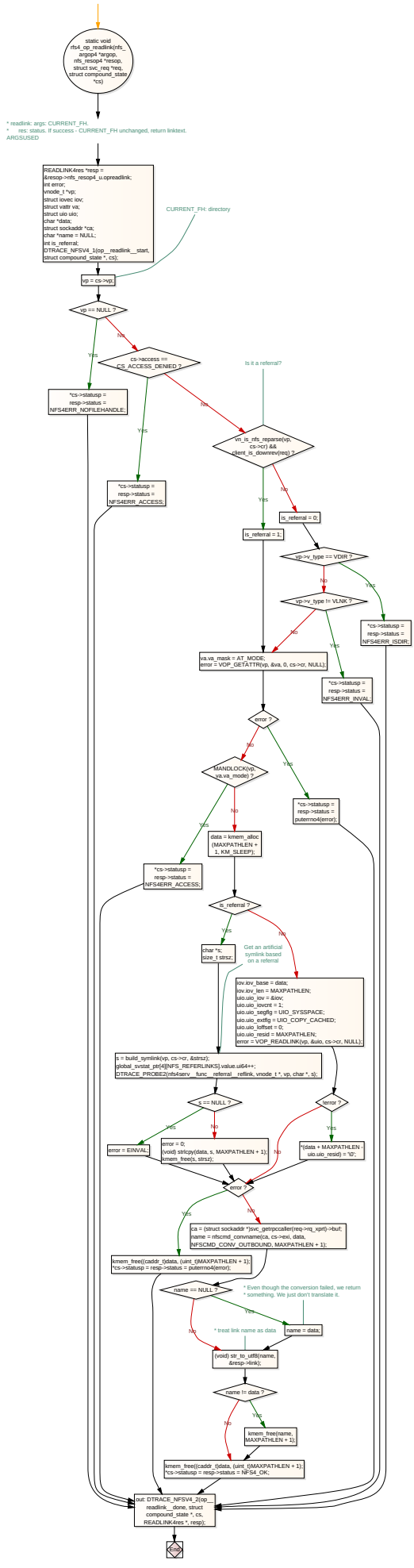

static void
rfs4_op_readdir_free(nfs_
resop4 * resop)
READDIR4res *resp =
&resop->nfs_resop4_
u.opreaddir;
resp->status == NFS4_OK
&& resp->mblk != NULL ?
Yes
No
freeb(resp->mblk);
resp->mblk = NULL;
resp->data_len = 0;
End

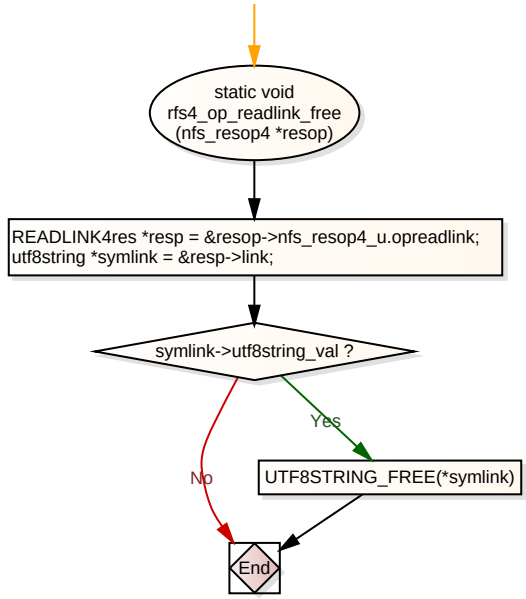

static void rfs4_op_putpubfh(nfs_ argop4 *args, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
PUTPUBFH4res *resp = &resop->nfs_resop4_u.opputpubfh;
int error;
vnode_t *vp;
struct exportinfo *exi, *sav_exi;
nfs_fh4_fmt_t *fh_fmtp;
DTRACE_NFSV4_1(op__putpubfh__start, struct compound_state *, cs);
cs->vp ?
Yes
VN_RELE(cs->vp);
cs->vp = NULL;
No
cs->cr ?
Yes
crfree(cs->cr);
No
cs->cr = crdup(cs->basecr);
vp = exi_public->exi_vp;
vp == NULL ?
No
Yes
error = makefh4(&cs->fh, vp, exi_public);
*cs->statusp = resp->status = NFS4ERR_SERVERFAULT;
error != 0 ?
Yes
No
*cs->statusp = resp->status = puterrno4(error);
sav_exi = cs->exi;
* No filesystem is actually shared public, so we default
* to exi_root. In this case, we must check whether root
* is exported.
exi_public == exi_root ?
* it's a properly shared filesystem
Yes
No
* if root filesystem is exported, the exportinfo struct that we
* should use is what checkexport4 returns, because root_exi is
* actually a mostly empty struct.
fh_fmtp = (nfs_fh4_fmt_t *)cs->fh.nfs_fh4_val;
cs->exi = exi_public;
exi = checkexport4(&fh_fmtp->fh4_fsid, (fid_t *)&fh_fmtp->fh4_xlen, NULL);
cs->exi = ((exi != NULL) ? exi : exi_public);
is_system_labeled() ?
Yes
No
bslabel_t *clabel;
ASSERT(req->rq_label != NULL);
clabel = req->rq_label;
DTRACE_PROBE2(tx__rfs4__log__info__opputpubfh__clabel, char *, "got client label from request(1)", struct svc_req *, req);
!blequal(&l_admin_low->tsl_label, clabel) ?
Yes
!do_rfs_label_check (clabel, vp, DOMINANCE_CHECK, cs->exi) ?
No
Yes
No
*cs->statusp = resp->status = NFS4ERR_SERVERFAULT;
VN_HOLD(vp);
cs->vp = vp;
(resp->status = call_checkauth4(cs, req)) != NFS4_OK ?
Yes
No
VN_RELE(cs->vp);
cs->vp = NULL;
cs->exi = sav_exi;
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__ putpubfh__done, struct compound_state *, cs, PUTPUBFH4res *, resp);
End

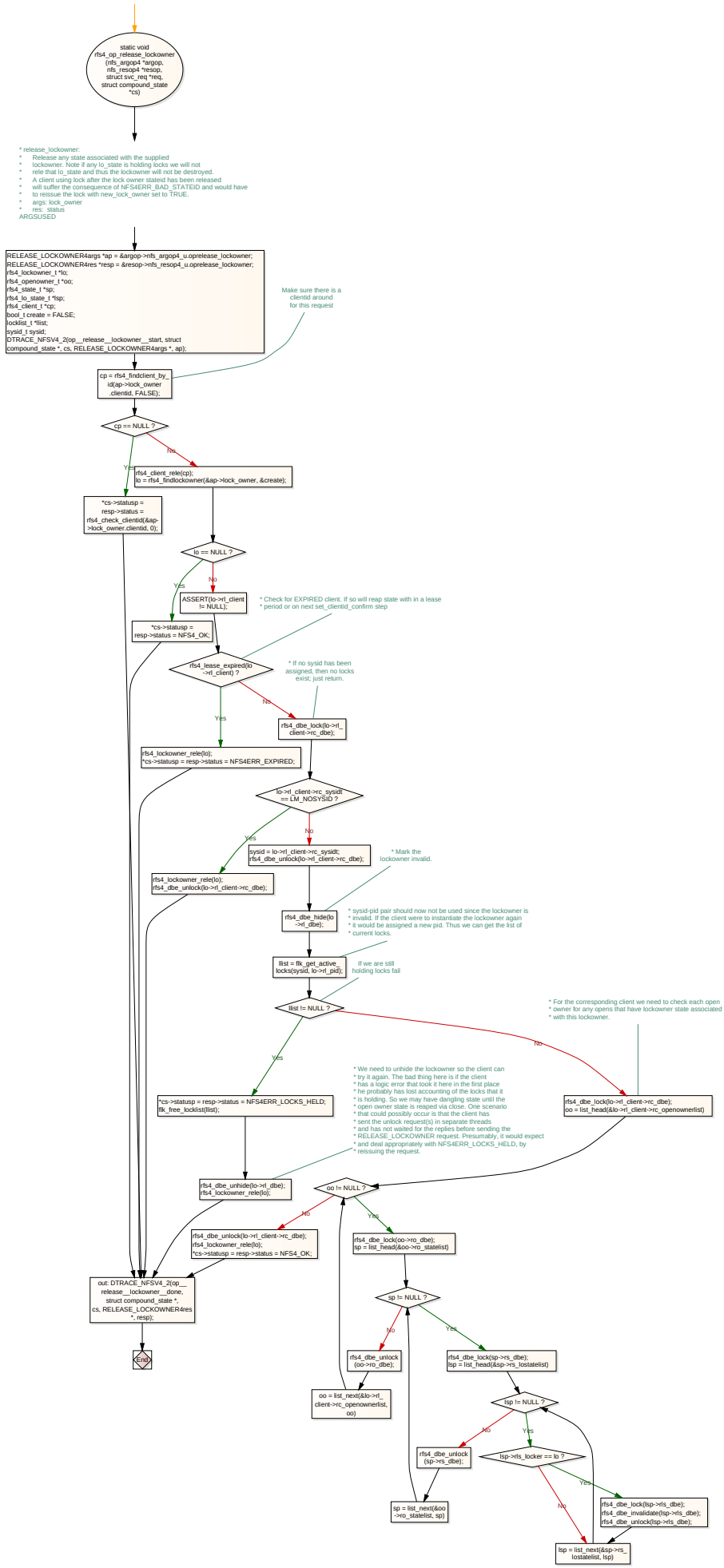

static void rfs4_op_putfh(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* XXX - issue with put*fh operations. Suppose /export/home is exported.
* Suppose an NFS client goes to mount /export/home/joe. If /export, home,
* or joe have restrictive search permissions, then we shouldn't let
* the client get a file handle. This is easy to enforce. However, we
* don't know what security flavor should be used until we resolve the
* path name. Another complication is uid mapping. If root is
* the user, then it will be mapped to the anonymous user by default,
* but we won't know that till we've resolved the path name. And we won't
* know what the anonymous user is.
* Luckily, SECINFO is specified to take a full filename.
* So what we will have to in rfs4_op_lookup is check that flavor of
* the target object matches that of the request, and if root was the
* caller, check for the root= and anon= options, and if necessary,
* repeat the lookup using the right cred_t. But that's not done yet.
ARGSUSED
PUTFH4args *args = &argop->nfs_argop4_u.opputfh;
PUTFH4res *resp = &resop->nfs_resop4_u.opputfh;
nfs_fh4_fmt_t *fh_fmtp;
DTRACE_NFSV4_2(op__putfh__start, struct compound_state *, cs, PUTFH4args *, args);
cs->vp ?
Yes
No
VN_RELE(cs->vp);
cs->vp = NULL;
cs->cr ?
Yes
No
crfree(cs->cr);
cs->cr = NULL;
args->object.nfs_fh4_len < NFS_FH4_LEN ?
No
Yes
fh_fmtp = (nfs_fh4_fmt_t *)args->object.nfs_fh4_val;
cs->exi = checkexport4(&fh_fmtp->fh4_fsid, (fid_t *)&fh_fmtp->fh4_xlen, NULL);
*cs->statusp = resp->status = NFS4ERR_BADHANDLE;
cs->exi == NULL ?
No
Yes
cs->cr = crdup(cs->basecr);
ASSERT(cs->cr != NULL);
*cs->statusp = resp->status = NFS4ERR_STALE;
! (cs->vp = nfs4_fhtovp(&args->object, cs->exi, &resp->status)) ?
No
Yes
(resp->status = call_checkauth4(cs, req)) != NFS4_OK ?
No
Yes
*cs->statusp = resp->status;
nfs_fh4_copy(&args->object, &cs->fh);
*cs->statusp = resp->status = NFS4_OK;
cs->deleg = FALSE;
VN_RELE(cs->vp);
cs->vp = NULL;
out: DTRACE_NFSV4_2(op__putfh__done, struct compound_state *, cs, PUTFH4res *, resp);
End

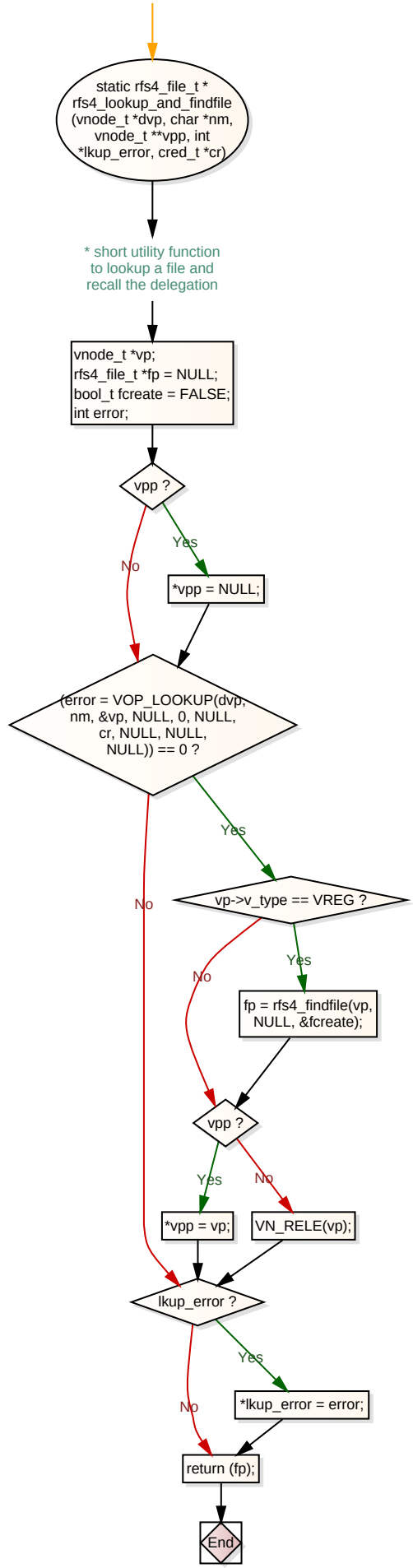

static void rfs4_op_putrootfh(nfs_ argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
PUTROOTFH4res *resp = &resop->nfs_resop4_u.opputrootfh;
int error;
fid_t fid;
struct exportinfo *exi, *sav_exi;
DTRACE_NFSV4_1(op__putrootfh__start, struct compound_state *, cs);
cs->vp ?
Yes
VN_RELE(cs->vp);
cs->vp = NULL;
No
cs->cr ?
Yes
crfree(cs->cr);
No
cs->cr = crdup(cs->basecr);
* Using rootdir, the system root vnode,
* get its fid.
bzero(&fid, sizeof (fid));
fid.fid_len = MAXFIDSZ;
error = vop_fid_pseudo(rootdir, &fid);
* Then use the root fsid & fid it to find out if it's exported
*
* If the server root isn't exported directly, then
* it should at least be a pseudo export based on
* one or more exports further down in the server's
* file tree.
error != 0 ?
Yes
No
exi = checkexport4(&rootdir->v_vfsp->vfs_ fsid, &fid, NULL);
*cs->statusp = resp->status = puterrno4(error);
exi == NULL || exi->exi_export.ex_flags & EX_PUBLIC ?
* Now make a filehandle based on the root
* export and root vnode.
Yes
No
error = makefh4(&cs->fh, rootdir, exi);
NFS4_DEBUG(rfs4_debug, (CE_WARN, "rfs4_op_putrootfh: export check failure"));
*cs->statusp = resp->status = NFS4ERR_SERVERFAULT;
error != 0 ?
No
Yes
sav_exi = cs->exi;
cs->exi = exi;
VN_HOLD(rootdir);
cs->vp = rootdir;
*cs->statusp = resp->status = puterrno4(error);
(resp->status = call_checkauth4(cs, req)) != NFS4_OK ?
No
Yes
*cs->statusp = resp->status = NFS4_OK;
cs->deleg = FALSE;
VN_RELE(rootdir);
cs->vp = NULL;
cs->exi = sav_exi;
out: DTRACE_NFSV4_2(op__ putrootfh__done, struct compound_state *, cs, PUTROOTFH4res *, resp);
End

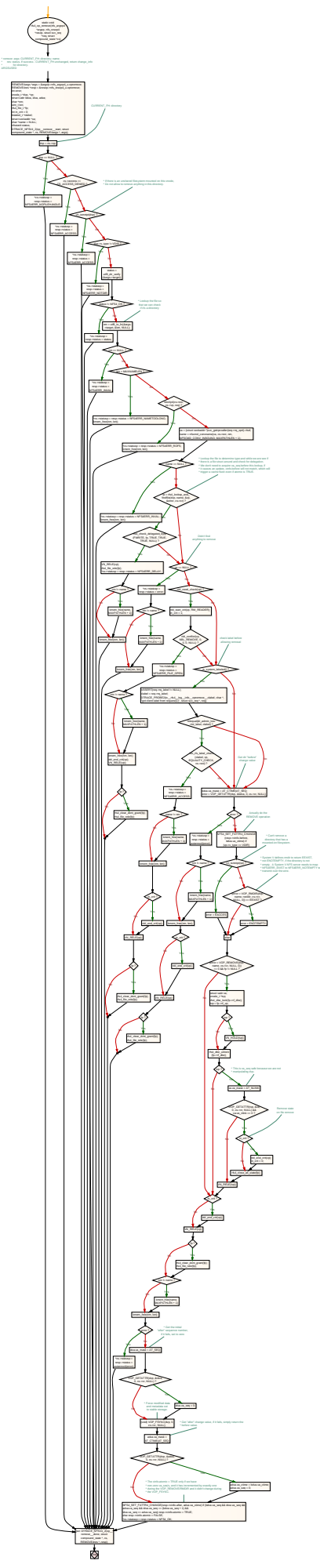

static nfsstat4 set_rdattr_params(struct nfs4_svgetit_arg *sargp, bitmap4 attrs, bool_t *need_to_lookup)
* set_rdattr_params sets up the variables used to manage what information * to get for each directory entry.
uint_t va_mask;
nfsstat4 status;
bitmap4 objbits;
status = bitmap4_to_attrmask(attrs, sargp);
status != NFS4_OK ?
* could not even figure attr mask
Yes
No
return (status);
* dirent's d_ino is always correct value for mounted_on_fileid. * mntdfid_set is set once here, but mounted_on_fileid is * set in main dirent processing loop for each dirent. * The mntdfid_set is a simple optimization that lets the * server attr code avoid work when caller is readdir.
va_mask = sargp->vap->va_mask;
* Lookup entry only if client asked for any of the following: * a) vattr attrs * b) vfs attrs * c) attrs w/per-object scope requested (change, filehandle, etc) * other than mounted_on_fileid (which we can take from dirent)
sargp->mntdfid_set = TRUE;
objbits = attrs ? attrs & NFS4_VP_ATTR_MASK : 0;
va_mask || sargp->sbp || (objbits & ~FATTR4_MOUNTED_ON_FILEID_MASK) ?
Yes
No
*need_to_lookup = TRUE;
*need_to_lookup = FALSE;
sargp->sbp == NULL ?
* If filesystem attrs are requested, get them now from the * directory vp, as most entries will have same filesystem. The only * exception are mounted over entries but we handle * those as we go (XXX mounted over detection not yet implemented). to avoid VOP_GETATTR
Yes
No
sargp->vap->va_mask = 0; status = bitmap4_get_sysattrs(sargp); sargp->vap->va_mask = va_mask;
return (NFS4_OK);
(status != NFS4_OK) && sargp->rdattr_error_req ?
* Failed to get filesystem attributes. * Return a rdattr_error for each entry, but don't fail. * However, don't get any obj-dependent attrs. for rdattr_error
Yes
No
* At least get fileid for regular readdir output
sargp->rdattr_error = status; *need_to_lookup = FALSE;
sargp->vap->va_mask &= AT_NODEID; status = NFS4_OK;
return (status);
End

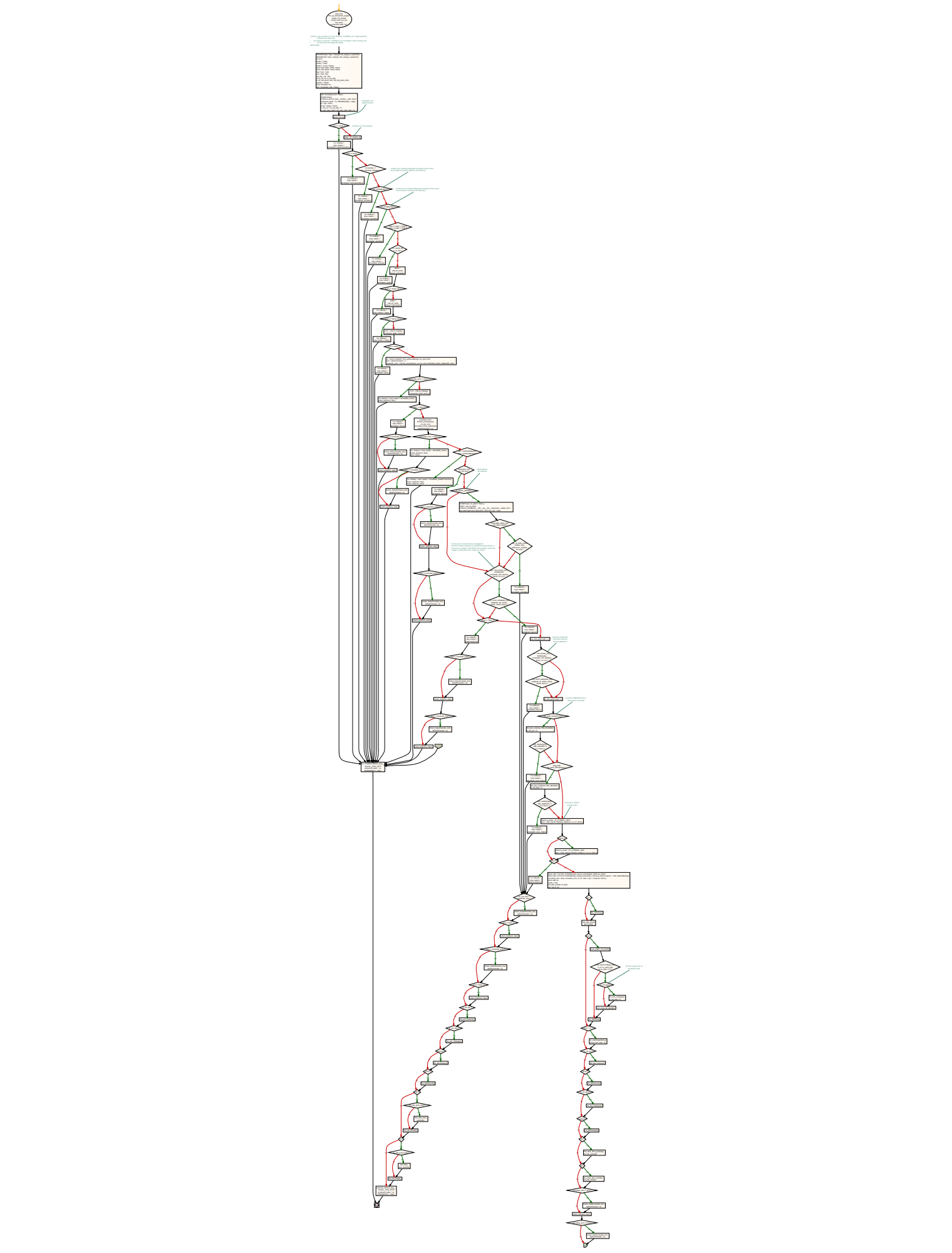

static void rfs4_op_readlink(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* readlink: args: CURRENT_FH.
* res: status. If success - CURRENT_FH unchanged, return linktext.
ARGSUSED
READLINK4res *resp = &resop->nfs_resop4_u.opreadlink;
int error;
vnode_t *vp;
struct iovec iov;
struct vattr va;
struct uio uio;
char *data;
struct sockaddr *ca;
char *name = NULL;
int is_referral;
DTRACE_NFSV4_1(op__readlink__start, struct compound_state *, cs);
CURRENT_FH: directory
vp = cs->vp;
vp == NULL ?
Yes
No
cs->access == CS_ACCESS_DENIED ?
Is it a referral?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
Yes
No
*cs->statusp = resp->status = NFS4ERR_ACCESS;
vn_is_nfs_reparse(vp, cs->cr) && client_is_downrev(req) ?
Yes
No
is_referral = 0;
is_referral = 1;
vp->v_type == VDIR ?
No
Yes
vp->v_type != VLNK ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_ISDIR;
va.va_mask = AT_MODE;
error = VOP_GETATTR(vp, &va, 0, cs->cr, NULL);
*cs->statusp = resp->status = NFS4ERR_INVAL;
error ?
No
Yes
MANDLOCK(vp, va.va_mode) ?
Yes
No
*cs->statusp = resp->status = puterrno4(error);
data = kmem_alloc (MAXPATHLEN + 1, KM_SLEEP);
*cs->statusp = resp->status = NFS4ERR_ACCESS;
is_referral ?
Yes
No
char *s;
size_t strsz;
Get an artificial symlink based on a referral
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, cs->cr, NULL);
s = build_symlink(vp, cs->cr, &strsz);
global_svstat_ptr[4][NFS_REFERLINKS].value.ui64++;
DTRACE_PROBE2(nfs4serv__func__referral__reflink, vnode_t *, vp, char *, s);
s == NULL ?
Yes
No
error = EINVAL;
error = 0;
(void) strlcpy(data, s, MAXPATHLEN + 1);
kmem_free(s, strsz);
error ?
Yes
No
*(data + MAXPATHLEN - uio.uio_resid) = '\0';
error ?
Yes
No
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, cs->exi, data, NFSCMD_CONV_OUTBOUND, MAXPATHLEN + 1);
kmem_free((caddr_t)data, (uint_t)MAXPATHLEN + 1);
*cs->statusp = resp->status = puterrno4(error);
name == NULL ?
* Even though the conversion failed, we return
* something. We just don't translate it.
Yes
No
name = data;
* treat link name as data
(void) str_to_utf8(name, &resp->link);
name != data ?
Yes
No
kmem_free(name, MAXPATHLEN + 1);
kmem_free((caddr_t)data, (uint_t)MAXPATHLEN + 1);
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__readlink__done, struct compound_state *, cs, READLINK4res *, resp);
End

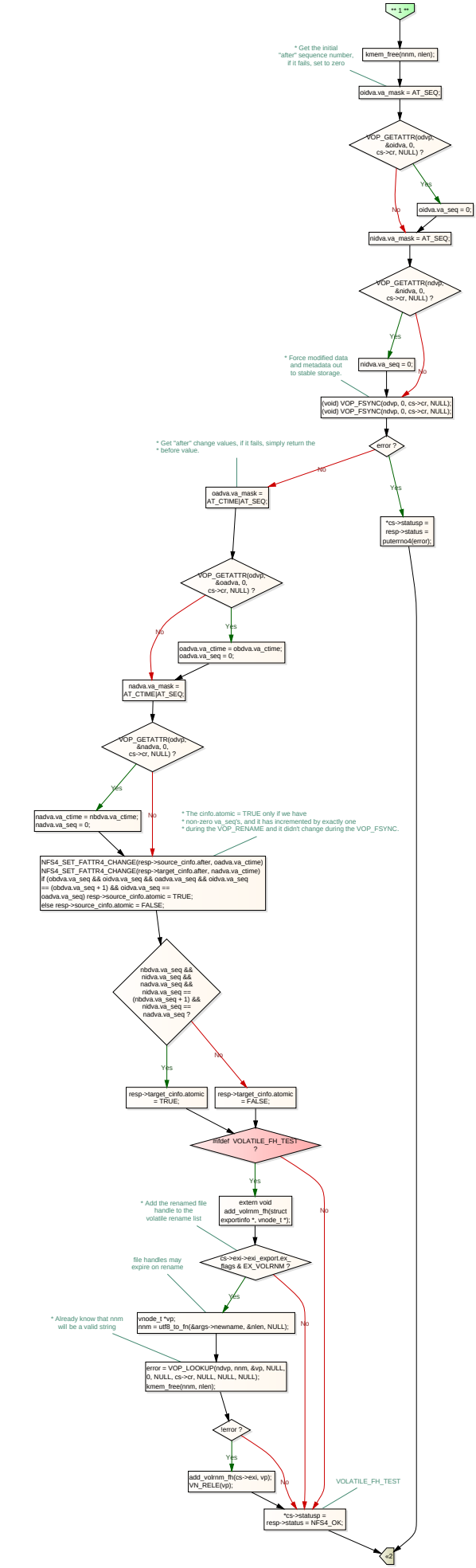

static void
rfs4_op_readlink_free
(nfs_resop4 *resop)
READLINK4res *resp = &resop->nfs_resop4_u.opreadlink;
utf8string *symlink = &resp->link;
symlink->utf8string_val ?
Yes
No
UTF8STRING_FREE(*symlink)
End

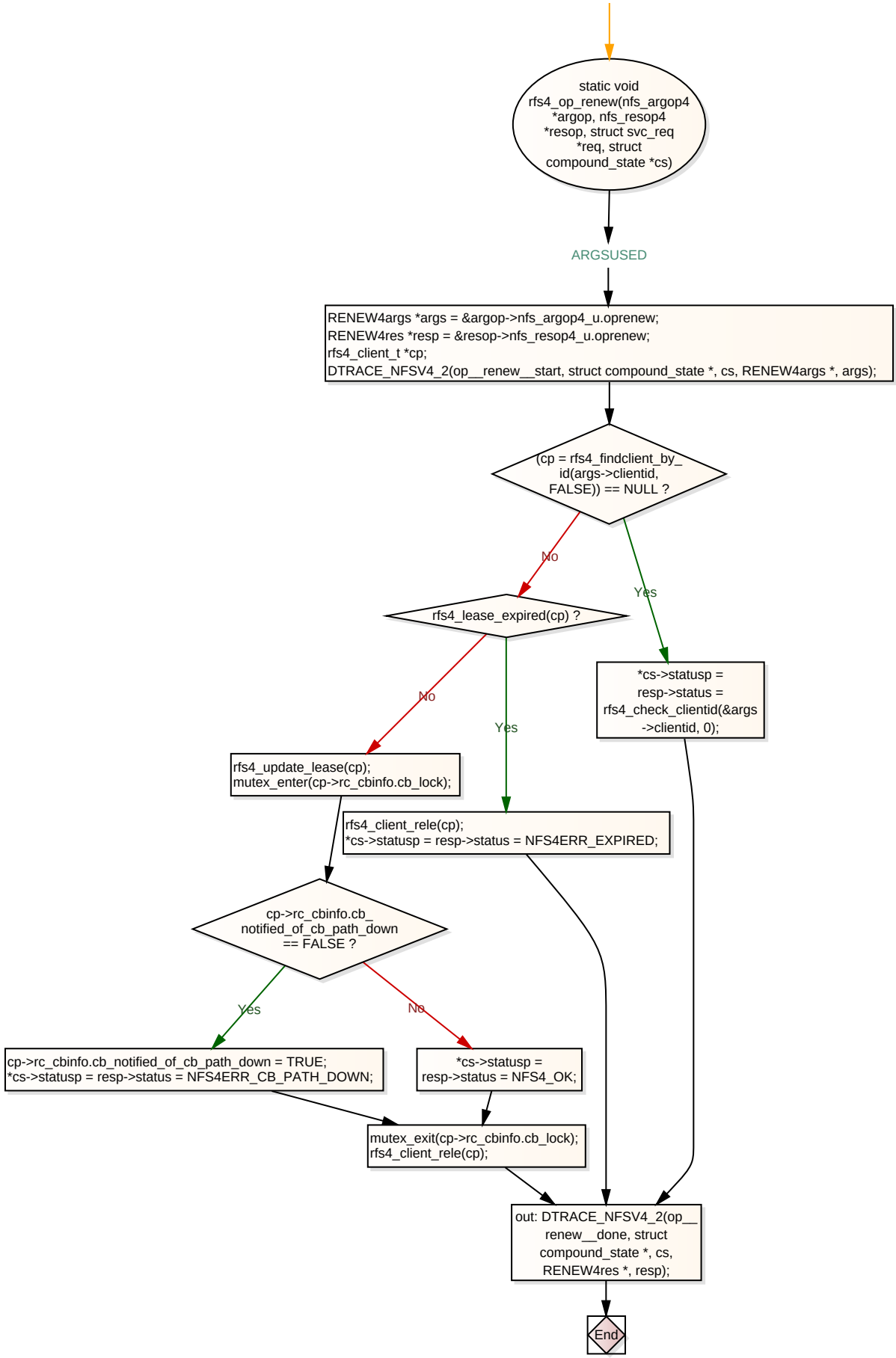

static void rfs4_op_release_lockowner (nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* release_lockowner:
* Release any state associated with the supplied
* lockowner. Note if any lo_state is holding locks we will not
* rele that lo_state and thus the lockowner will not be destroyed.
* A client using lock after the lock owner stateid has been released
* will suffer the consequence of NFS4ERR_BAD_STATEID and would have
* to reissue the lock with new_lock_owner set to TRUE.
* args: lock_owner
* res: status
ARGSUSED
RELEASE_LOCKOWNER4args *ap = &argop->nfs_argop4_u.oprelease_lockowner;
RELEASE_LOCKOWNER4res *resp = &resop->nfs_resop4_u.oprelease_lockowner;
rfs4_lockowner_t *lo;
rfs4_openowner_t *oo;
rfs4_state_t *sp;
rfs4_lo_state_t *lsp;
rfs4_client_t *cp;
bool_t create = FALSE;
locklist_t *llist;
sysid_t sysid;
DTRACE_NFSV4_2(op__release__lockowner__start, struct compound_state *, cs, RELEASE_LOCKOWNER4args *, ap);
Make sure there is a clientid around for this request
cp = rfs4_findclient_by_id(ap->lock_owner.clientid, FALSE);
cp == NULL ?
No
Yes
rfs4_client_rele(cp);
lo = rfs4_findlockowner(&ap->lock_owner, &create);
*cs->statusp = resp->status = rfs4_check_clientid(&ap->lock_owner.clientid, 0);
lo == NULL ?
Yes
No
ASSERT(lo->rl_client != NULL);
* Check for EXPIRED client. If so will reap state with in a lease
* period or on next set_clientid_confirm step
*cs->statusp = resp->status = NFS4_OK;
rfs4_lease_expired(lo->rl_client) ?
* If no sysid has been assigned, then no locks exist; just return.
Yes
No
rfs4_dbe_lock(lo->rl_client->rc_dbe);
rfs4_lockowner_rele(lo);
*cs->statusp = resp->status = NFS4ERR_EXPIRED;
lo->rl_client->rc_sysidt == LM_NOSYSID ?
Yes
No
sysid = lo->rl_client->rc_sysidt;
rfs4_dbe_unlock(lo->rl_client->rc_dbe);
* Mark the lockowner invalid.
rfs4_lockowner_rele(lo);
rfs4_dbe_unlock(lo->rl_client->rc_dbe);
rfs4_dbe_hide(lo->rl_dbe);
* sysid-pid pair should now not be used since the lockowner is
* invalid. If the client were to instantiate the lockowner again
* it would be assigned a new pid. Thus we can get the list of
* current locks.
llist = flk_get_active_locks(sysid, lo->rl_pid);
If we are still holding locks fail
llist != NULL ?
Yes
No
* For the corresponding client we need to check each open
* owner for any opens that have lockowner state associated
* with this lockowner.
*cs->statusp = resp->status = NFS4ERR_LOCKS_HELD;
flk_free_locklist(llist);
* We need to unhide the lockowner so the client can
* try it again. The bad thing here is if the client
* has a logic error that took it here in the first place
* he probably has lost accounting of the locks that it
* is holding. So we may have dangling state until the
* open owner state is reaped via close. One scenario
* that could possibly occur is that the client has
* sent the unlock request(s) in separate threads
* and has not waited for the replies before sending the
* RELEASE_LOCKOWNER request. Presumably, it would expect
* and deal appropriately with NFS4ERR_LOCKS_HELD, by
* reissuing the request.
rfs4_dbe_lock(lo->rl_client->rc_dbe);
oo = list_head(&lo->rl_client->rc_openownerlist)
rfs4_dbe_unhide(lo->rl_dbe);
rfs4_lockowner_rele(lo);
oo != NULL ?
No
Yes
rfs4_dbe_unlock(lo->rl_client->rc_dbe);
rfs4_lockowner_rele(lo);
*cs->statusp = resp->status = NFS4_OK;
rfs4_dbe_lock(oo->ro_dbe);
sp = list_head(&oo->ro_statelist)
out: DTRACE_NFSV4_2(op__release__lockowner__done, struct compound_state *, cs, RELEASE_LOCKOWNER4res *, resp);
sp != NULL ?
No
Yes
End
rfs4_dbe_unlock (oo->ro_dbe);
rfs4_dbe_lock(sp->rs_dbe);
lsp = list_head(&sp->rs_lostatelist)
oo = list_next(&lo->rl_client->rc_openownerlist, oo)
lsp != NULL ?
No
Yes
rfs4_dbe_unlock (sp->rs_dbe);
lsp->rls_locker == lo ?
Yes
No
sp = list_next(&oo->ro_statelist, sp)
rfs4_dbe_lock(lsp->rls_dbe);
rfs4_dbe_invalidate(lsp->rls_dbe);
rfs4_dbe_unlock(lsp->rls_dbe);
lsp = list_next(&sp->rs_lostatelist, lsp)

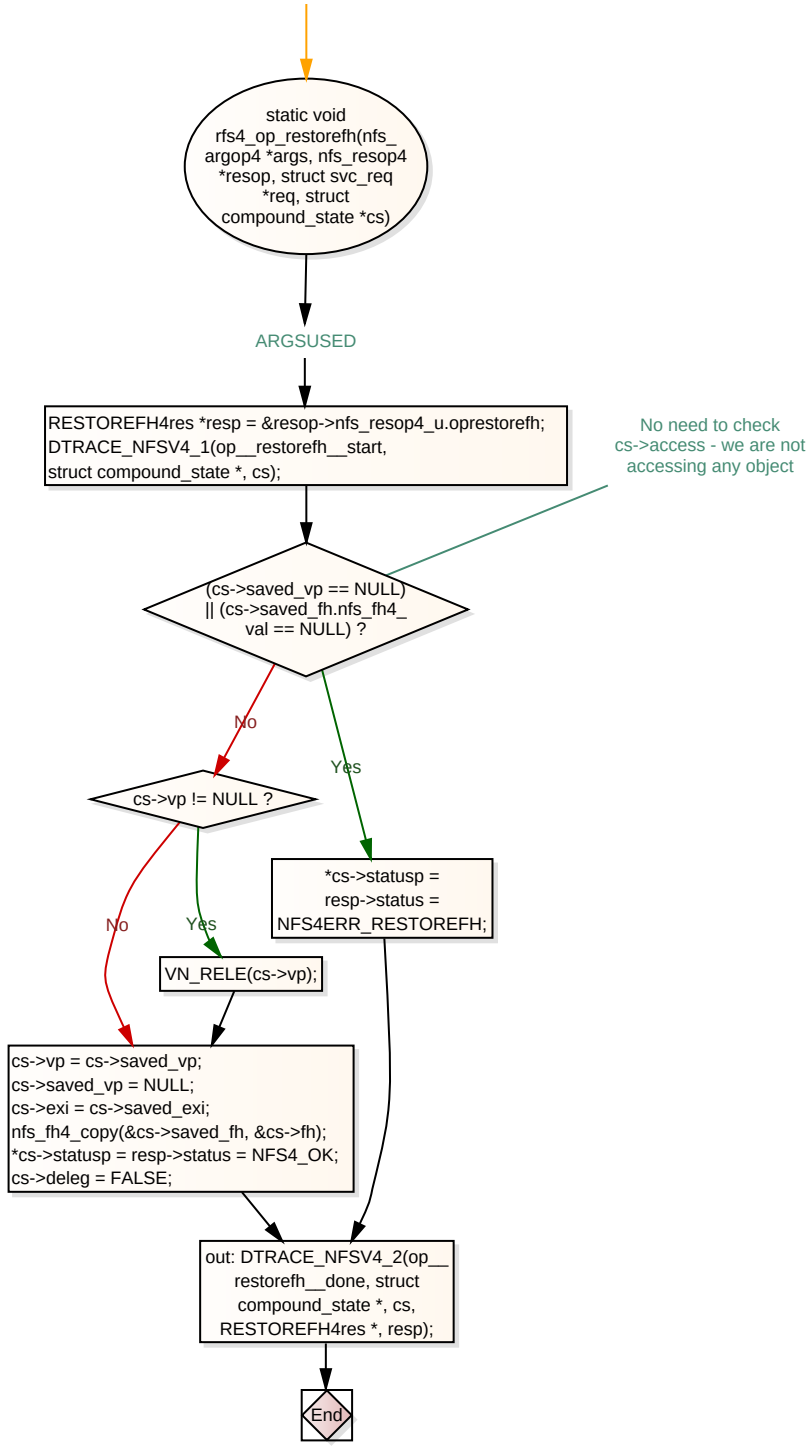

static rfs4_file_t *
rfs4_lookup_and_findfile
(vnode_t *dvp, char *nm,
vnode_t **vpp, int
*lkup_error, cred_t *cr)
* short utility function
to lookup a file and
recall the delegation
vnode_t *vp;
rfs4_file_t *fp = NULL;
bool_t fcreate = FALSE;
int error;
vpp ?
Yes
No
*vpp = NULL;
(error = VOP_LOOKUP(dvp,
nm, &vp, NULL, 0, NULL,
cr, NULL, NULL,
NULL)) == 0 ?
Yes
No
vp->v_type == VREG ?
Yes
No
fp = rfs4_findfile(vp,
NULL, &fcreate);
vpp ?
Yes
No
*vpp = vp;
VN_RELE(vp);
lkup_error ?
Yes
No
*lkup_error = error;
return (fp);
End

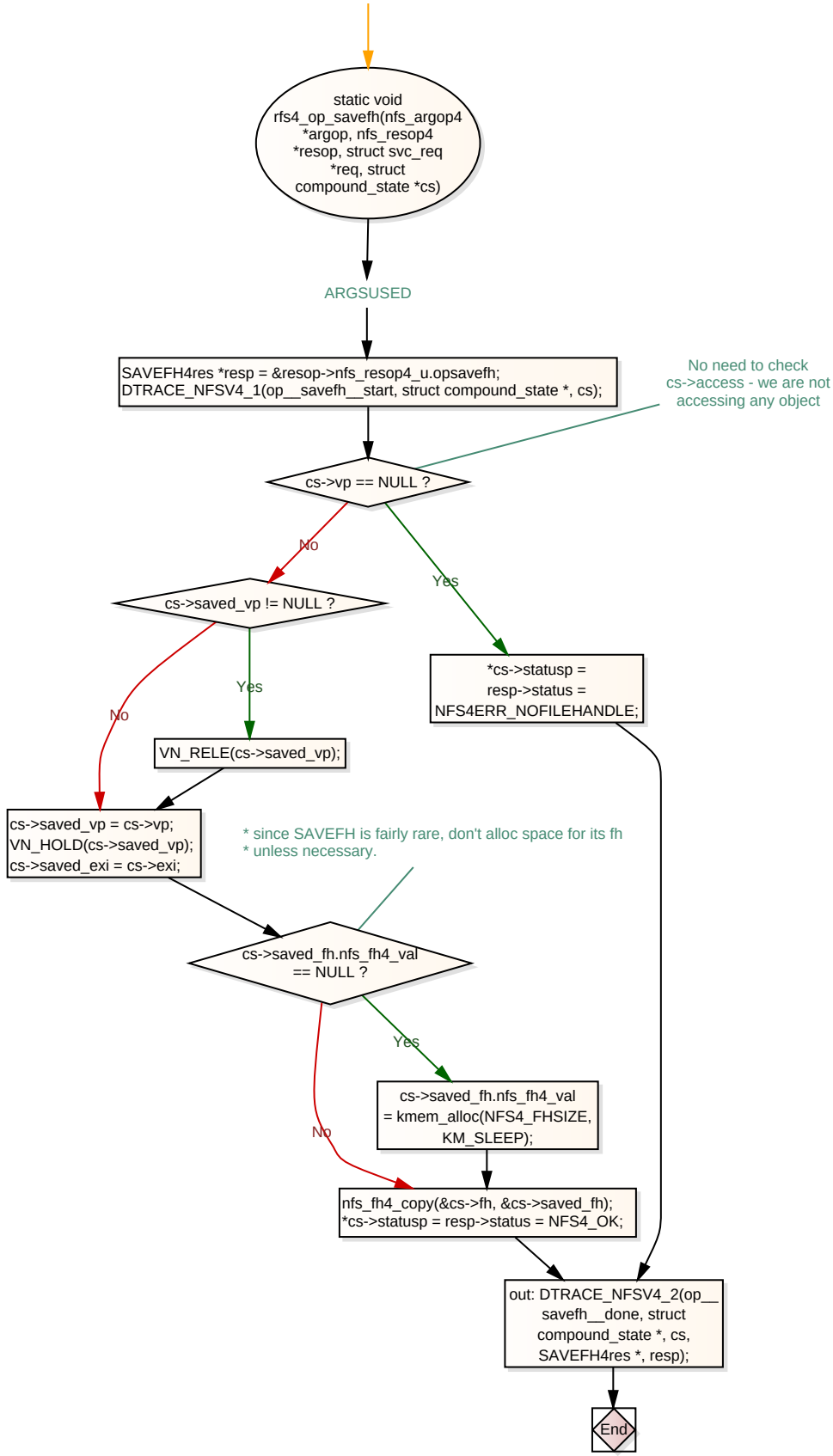


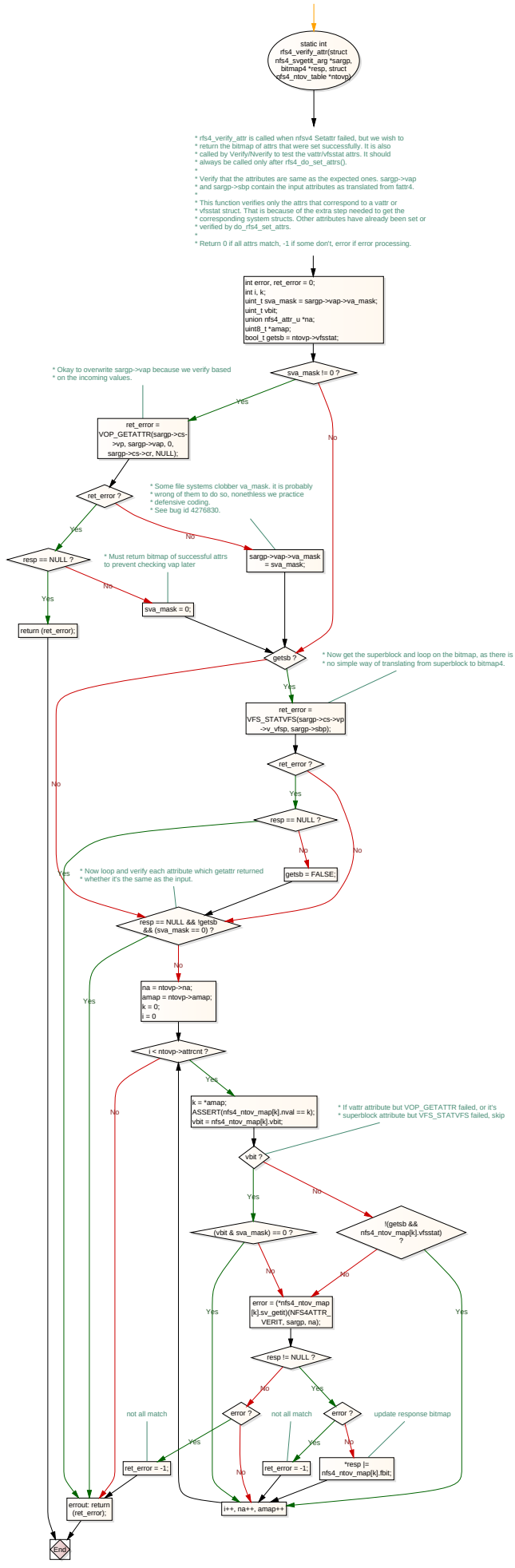


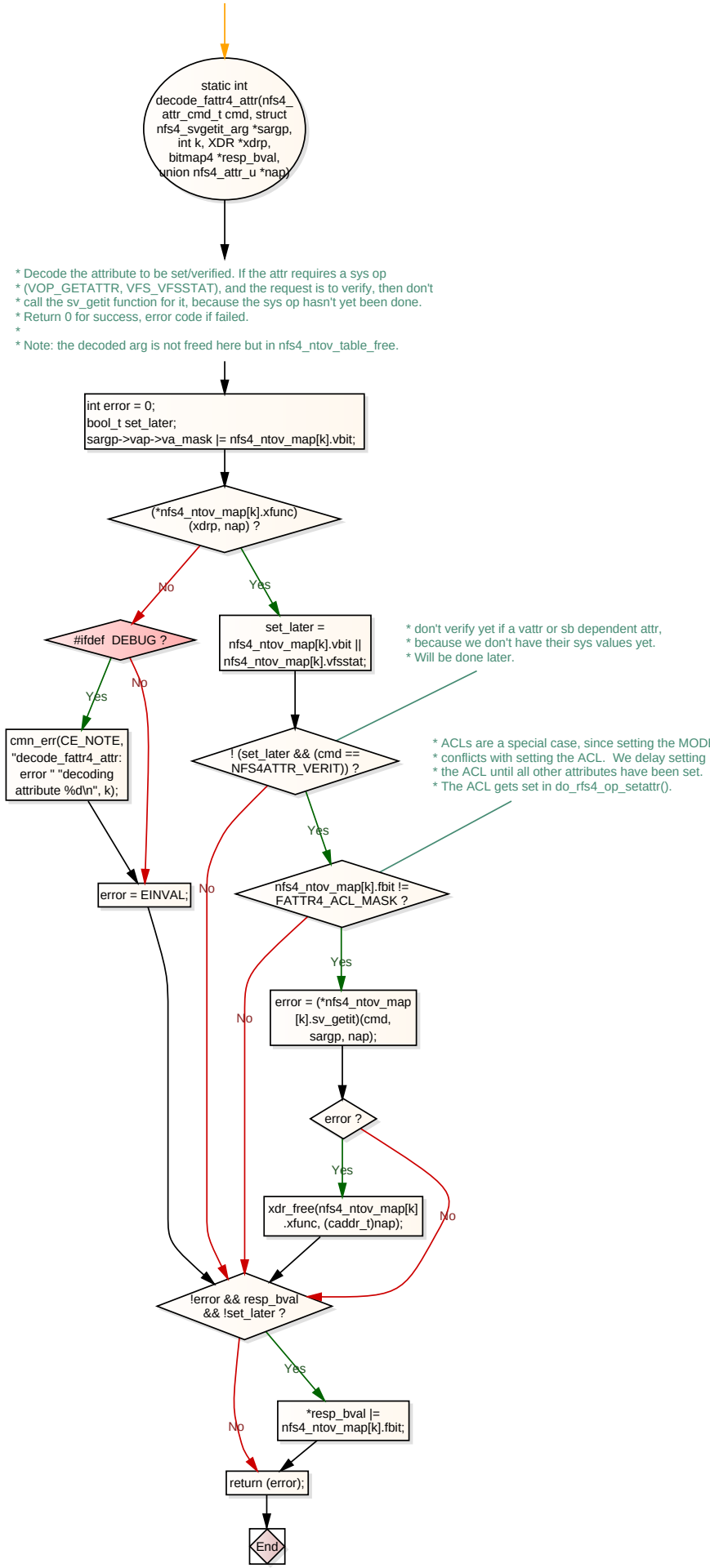

** 1 **
kmem_free(nnm, nlen);
* Get the initial "after" sequence number, if it fails, set to zero
oidva.va_mask = AT_SEQ;
VOP_GETATTR(odvp, &oidva, 0, cs->cr, NULL) ?
Yes
No
oidva.va_seq = 0;
nidva.va_mask = AT_SEQ;
VOP_GETATTR(ndvp, &nidva, 0, cs->cr, NULL) ?
Yes
No
nidva.va_seq = 0;
* Force modified data and metadata out to stable storage.
(void) VOP_FSYNC(odvp, 0, cs->cr, NULL);
(void) VOP_FSYNC(ndvp, 0, cs->cr, NULL);
error ?
No
Yes
* Get "after" change values, if it fails, simply return the * before value.
oadva.va_mask = AT_CTIME|AT_SEQ;
*cs->statusp = resp->status = puterrno4(error);
VOP_GETATTR(odvp, &oadva, 0, cs->cr, NULL) ?
No
Yes
oadva.va_ctime = obdva.va_ctime; oadva.va_seq = 0;
nadva.va_mask = AT_CTIME|AT_SEQ;
VOP_GETATTR(odvp, &nadva, 0, cs->cr, NULL) ?
Yes
No
nadva.va_ctime = nbdva.va_ctime; nadva.va_seq = 0;
* The cinfo.atomic = TRUE only if we have * non-zero va_seq's, and it has incremented by exactly one * during the VOP_RENAME and it didn't change during the VOP_FSYNC.
NFS4_SET_FATTR4_CHANGE(resp->source_cinfo.after, oadva.va_ctime)
NFS4_SET_FATTR4_CHANGE(resp->target_cinfo.after, nadva.va_ctime)
if (obdva.va_seq && oidva.va_seq && oadva.va_seq && oidva.va_seq == (obdva.va_seq + 1) && oidva.va_seq == oadva.va_seq) resp->source_cinfo.atomic = TRUE;
else resp->source_cinfo.atomic = FALSE;
nbdva.va_seq && nidva.va_seq && nadva.va_seq && nidva.va_seq == (nbdva.va_seq + 1) && nidva.va_seq == nadva.va_seq ?
Yes
No
resp->target_cinfo.atomic = TRUE;
resp->target_cinfo.atomic = FALSE;
ifdef VOLATILE_FH_TEST ?
Yes
No
* Add the renamed file handle to the volatile rename list
extern void add_volrnm_fh(struct exportinfo *, vnode_t *);
file handles may expire on rename
cs->exi->exi_export.ex_ flags & EX_VOLRNM ?
Yes
No
* Already know that nnm will be a valid string
vnode_t *vp;
nnm = utf8_to_fn(&args->newname, &nlen, NULL);
error = VOP_LOOKUP(ndvp, nnm, &vp, NULL, 0, NULL, cs->cr, NULL, NULL, NULL);
kmem_free(nnm, nlen);
!error ?
Yes
No
add_volrnm_fh(cs->exi, vp); VN_RELE(vp);
VOLATILE_FH_TEST
*cs->statusp = resp->status = NFS4_OK;
x2

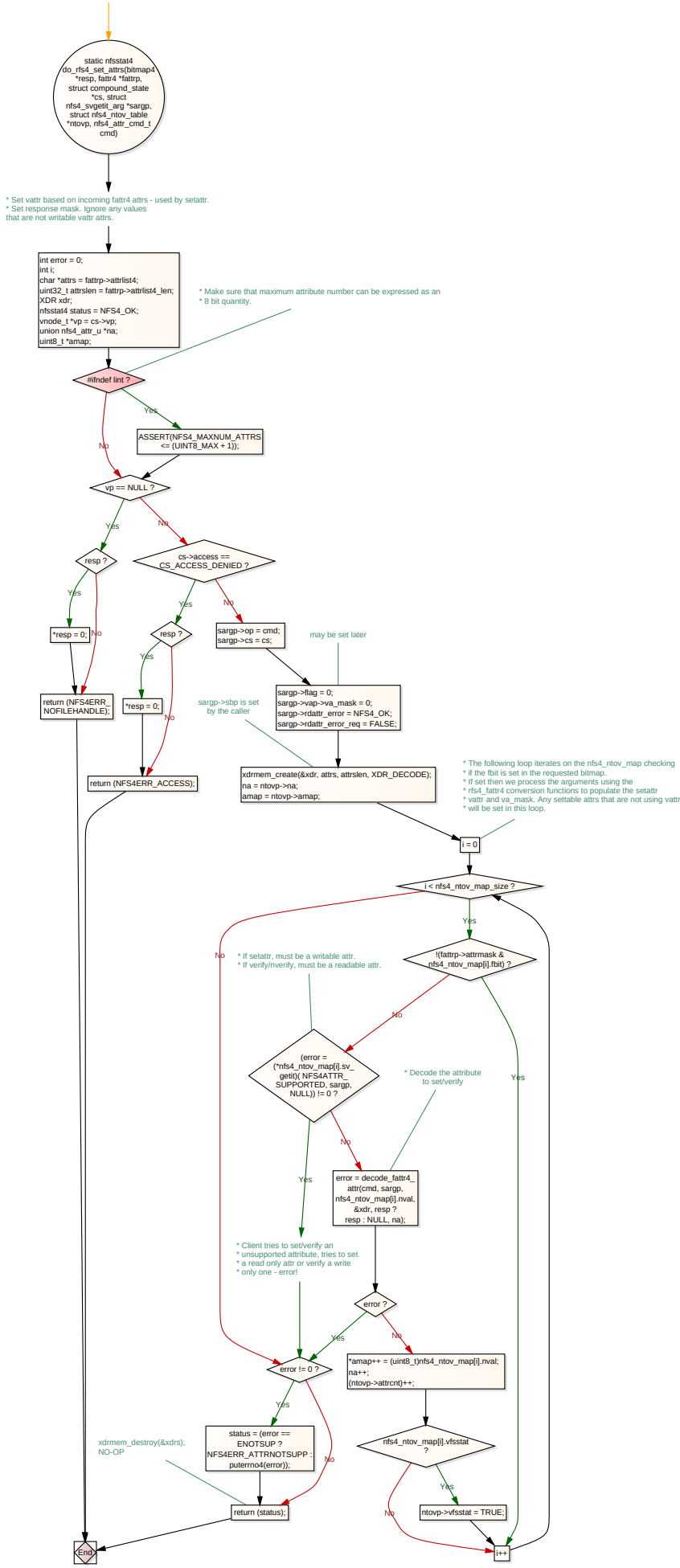

static void rfs4_op_renew(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
RENEW4args *args = &argop->nfs_argop4_u.oprenew;
RENEW4res *resp = &resop->nfs_resop4_u.oprenew;
rfs4_client_t *cp;
DTRACE_NFSV4_2(op__renew__start, struct compound_state *, cs, RENEW4args *, args);
(cp = rfs4_findclient_by_id(args->clientid, FALSE)) == NULL ?
No
Yes
rfs4_lease_expired(cp) ?
No
Yes
*cs->statusp = resp->status = rfs4_check_clientid(&args->clientid, 0);
rfs4_update_lease(cp);
mutex_enter(cp->rc_cbinfo.cb_lock);
rfs4_client_rele(cp);
*cs->statusp = resp->status = NFS4ERR_EXPIRED;
cp->rc_cbinfo.cb_notified_of_cb_path_down == FALSE ?
Yes
No
cp->rc_cbinfo.cb_notified_of_cb_path_down = TRUE;
*cs->statusp = resp->status = NFS4ERR_CB_PATH_DOWN;
*cs->statusp = resp->status = NFS4_OK;
mutex_exit(cp->rc_cbinfo.cb_lock);
rfs4_client_rele(cp);
out: DTRACE_NFSV4_2(op__renew__done, struct compound_state *, cs, RENEW4res *, resp);
End

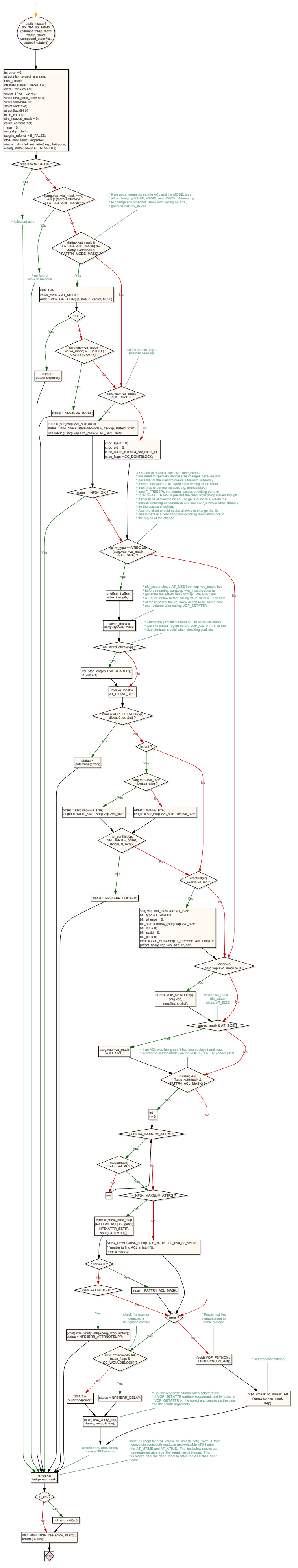

static void rfs4_op_restorefh(nfs_ argop4 *args, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
RESTOREFH4res *resp = &resop->nfs_resop4_u.oprestorefh;
DTRACE_NFSV4_1(op__restorefh__start, struct compound_state *, cs);
No need to check cs->access - we are not accessing any object
(cs->saved_vp == NULL) || (cs->saved_fh.nfs_fh4_ val == NULL) ?
No
Yes
cs->vp != NULL ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_RESTOREFH;
VN_RELE(cs->vp);
cs->vp = cs->saved_vp;
cs->saved_vp = NULL;
cs->exi = cs->saved_exi;
nfs_fh4_copy(&cs->saved_fh, &cs->fh);
*cs->statusp = resp->status = NFS4_OK;
cs->deleg = FALSE;
out: DTRACE_NFSV4_2(op__ restorefh__done, struct compound_state *, cs, RESTOREFH4res *, resp);
End


static void rfs4_op_savefh(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
SAVEFH4res *resp = &resop->nfs_resop4_u.opsavefh;
DTRACE_NFSV4_1(op__savefh__start, struct compound_state *, cs);
No need to check cs->access - we are not accessing any object
cs->vp == NULL ?
No
Yes
cs->saved_vp != NULL ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
VN_RELE(cs->saved_vp);
cs->saved_vp = cs->vp;
VN_HOLD(cs->saved_vp);
cs->saved_exi = cs->exi;
* since SAVEFH is fairly rare, don't alloc space for its fh
* unless necessary.
cs->saved_fh.nfs_fh4_val == NULL ?
Yes
No
cs->saved_fh.nfs_fh4_val = kmem_alloc(NFS4_FHSIZE, KM_SLEEP);
nfs_fh4_copy(&cs->fh, &cs->saved_fh);
*cs->statusp = resp->status = NFS4_OK;
out: DTRACE_NFSV4_2(op__savefh__done, struct compound_state *, cs, SAVEFH4res *, resp);
End

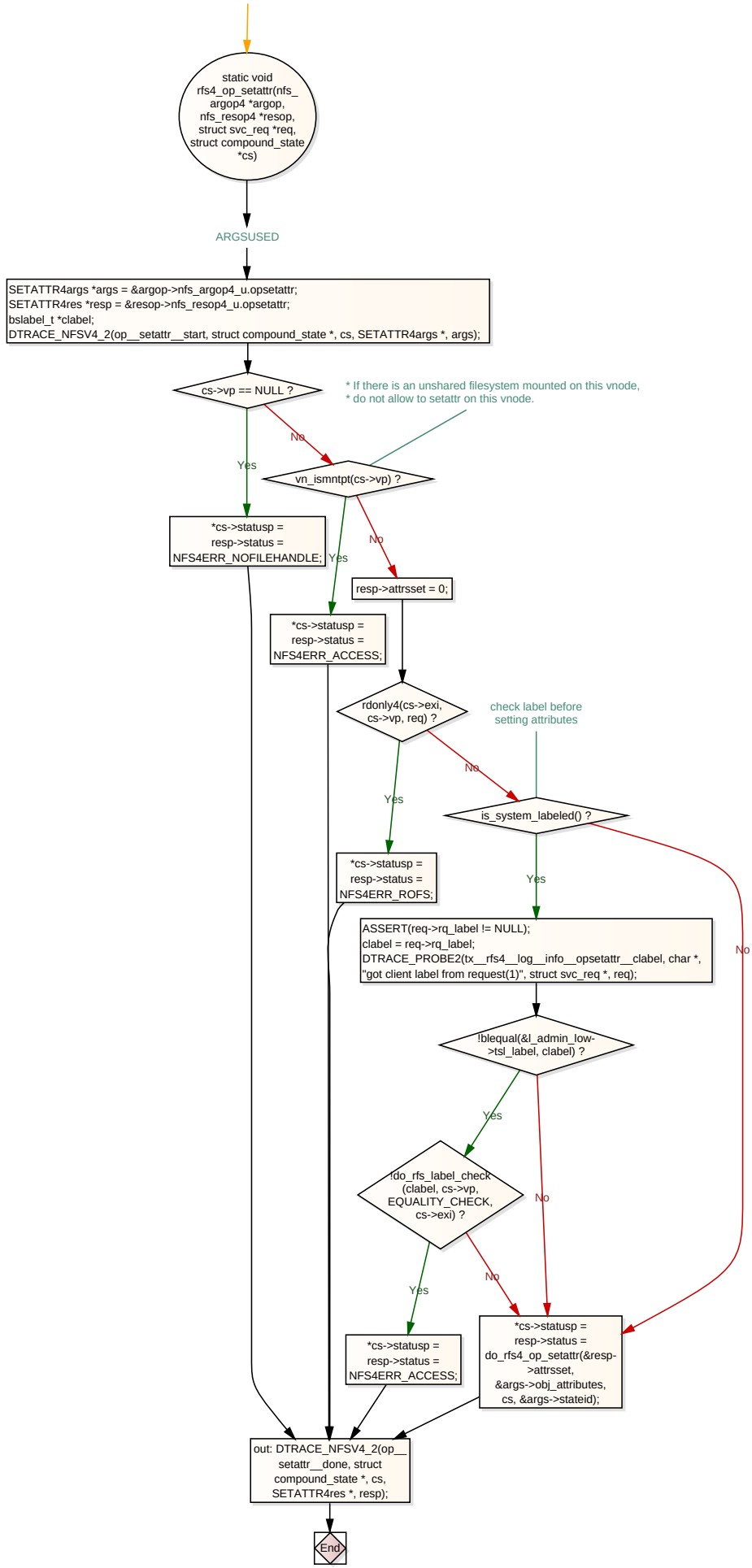

static int
rfs4_verify_attr(struct
nfs4_svgetit_arg *sargp,
bitmap4 *resp, struct
nfs4_ntov_table *ntovp)
* rfs4_verify_attr is called when nfsv4 Setattr failed, but we wish to
* return the bitmap of attrs that were set successfully. It is also
* called by Verify/Nverify to test the vattr/vfsstat attrs. It should
* always be called only after rfs4_do_set_attrs().
*
* Verify that the attributes are same as the expected ones. sargp->vap
* and sargp->sbp contain the input attributes as translated from fattr4.
*
* This function verifies only the attrs that correspond to a vattr or
* vfsstat struct. That is because of the extra step needed to get the
* corresponding system structs. Other attributes have already been set or
* verified by do_rfs4_set_attrs.
*
* Return 0 if all attrs match, -1 if some don't, error if error processing.
int error, ret_error = 0;
int i, k;
uint_t sva_mask = sargp->vap->va_mask;
uint_t vbit;
union nfs4_attr_u *na;
uint8_t *amap;
bool_t getsb = ntovp->vfsstat;
sva_mask != 0 ?
Yes
No
* Okay to overwrite sargp->vap because we verify based
* on the incoming values.
ret_error =
VOP_GETATTR(sargp->cs-
>vp, sargp->vap, 0,
sargp->cs->cr, NULL);
ret_error ?
Yes
No
* Some file systems clobber va_mask. it is probably
* wrong of them to do so, nonethless we practice
* defensive coding.
* See bug id 4276830.
resp == NULL ?
Yes
No
* Must return bitmap of successful attrs
to prevent checking vap later
sva_mask = 0;
sargp->vap->va_mask
= sva_mask;
return (ret_error);
getsb ?
Yes
No
* Now get the superblock and loop on the bitmap, as there is
* no simple way of translating from superblock to bitmap4.
ret_error =
VFS_STATVFS(sargp->cs->vp
->v_vfsp, sargp->sbp);
ret_error ?
Yes
No
resp == NULL ?
No
Yes
No
getsb = FALSE;
Yes
* Now loop and verify each attribute which getattr returned
* whether it's the same as the input.
resp == NULL && !getsb
&& (sva_mask == 0) ?
No
Yes
na = ntovp->na;
amap = ntovp->amap;
k = 0;
i = 0
i < ntovp->attrcnt ?
Yes
No
k = *amap;
ASSERT(nfs4_ntov_map[k].nval == k);
vbit = nfs4_ntov_map[k].vbit;
* If vattr attribute but VOP_GETATTR failed, or it's
* superblock attribute but VFS_STATVFS failed, skip
vbit ?
Yes
No
(vbit & sva_mask) == 0 ?
!(getsb &&
nfs4_ntov_map[k].vfsstat)
?
Yes
No
No
Yes
Yes
error = (*nfs4_ntov_map
[k].sv_getit)(NFS4ATTR_
VERIT, sargp, na);
resp != NULL ?
No
Yes
error ?
not all match
Yes
No
error ?
not all match
Yes
No
update response bitmap
ret_error = -1;
ret_error = -1;
*resp |=
nfs4_ntov_map[k].fbit;
errout: return
(ret_error);
i++, na++, amap++
End

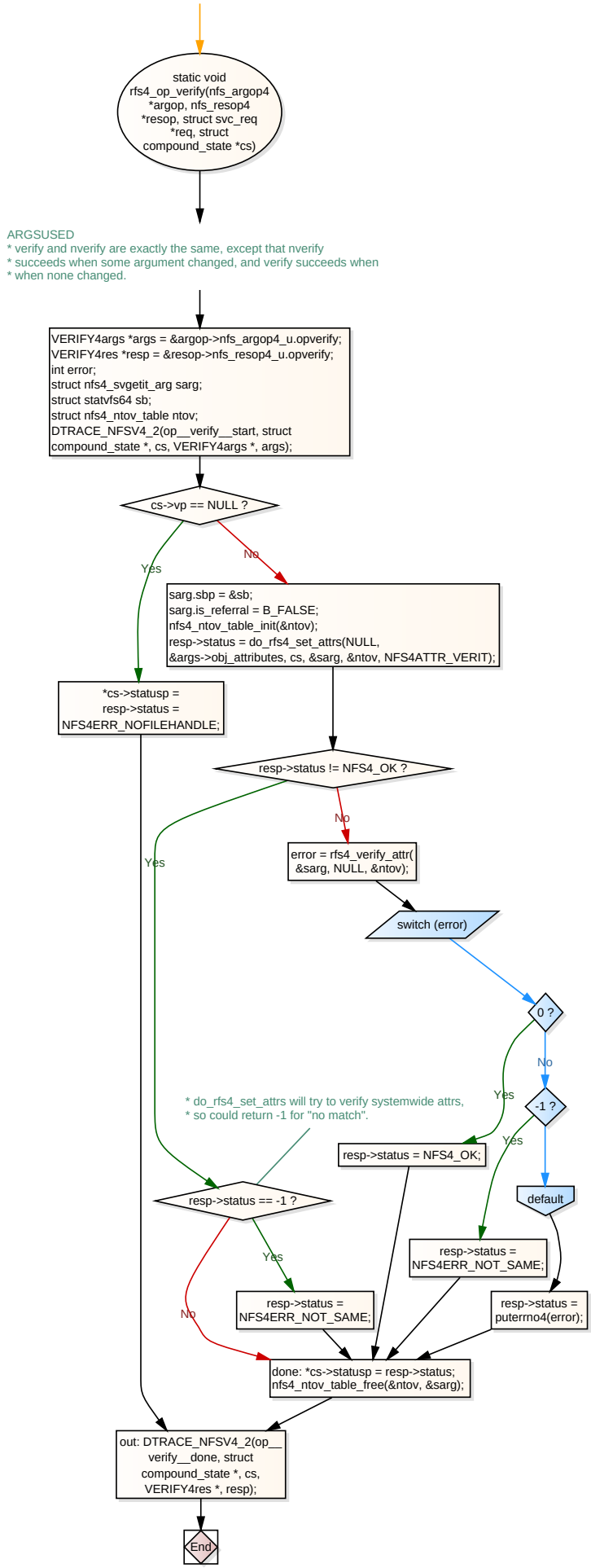

static int
decode_fattr4_attr(nfs4_
attr_cmd_t cmd, struct
nfs4_svgetit_arg *sargp,
int k, XDR *xdrp,
bitmap4 *resp_bval,
union nfs4_attr_u *nap)
* Decode the attribute to be set/verified. If the attr requires a sys op
* (VOP_GETATTR, VFS_VFSSTAT), and the request is to verify, then don't
* call the sv_getit function for it, because the sys op hasn't yet been done.
* Return 0 for success, error code if failed.
*
* Note: the decoded arg is not freed here but in nfs4_ntov_table_free.
int error = 0;
bool_t set_later;
sargp->vap->va_mask |= nfs4_ntov_map[k].vbit;
(*nfs4_ntov_map[k].xfunc)
(xdrp, nap) ?
No
Yes
#ifdef DEBUG ?
set_later =
nfs4_ntov_map[k].vbit ||
nfs4_ntov_map[k].vfsstat;
* don't verify yet if a vattr or sb dependent attr,
* because we don't have their sys values yet.
* Will be done later.
Yes
No
cmn_err(CE_NOTE,
"decode_fattr4_attr:
error " "decoding
attribute %d\n", k);
! (set_later && (cmd ==
NFS4ATTR_VERIT)) ?
* ACLs are a special case, since setting the MODE
* conflicts with setting the ACL. We delay setting
* the ACL until all other attributes have been set.
* The ACL gets set in do_rfs4_op_setattr().
Yes
error = EINVAL;
No
nfs4_ntov_map[k].fbit !=
FATTR4_ACL_MASK ?
Yes
No
error = (*nfs4_ntov_map
[k].sv_getit)(cmd,
sargp, nap);
error ?
Yes
No
xdr_free(nfs4_ntov_map[k]
.xfunc, (caddr_t)nap);
!error && resp_bval
&& !set_later ?
Yes
No
*resp_bval |=
nfs4_ntov_map[k].fbit;
return (error);
End

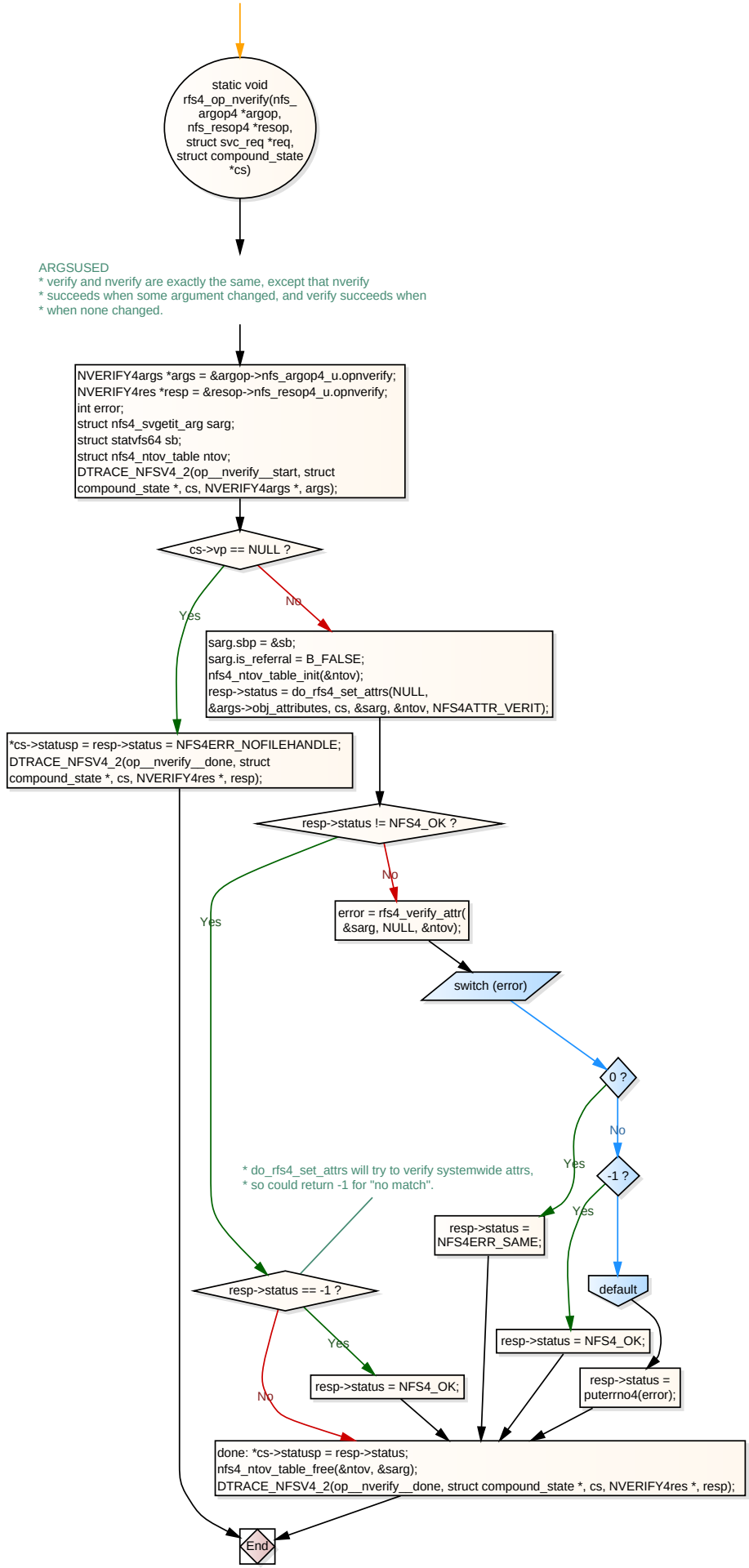

static nfsstat4 do_rfs4_set_attrs(bitmap4 *resp, fattr4 *fattrp, struct compound_state *cs, struct nfs4_svgetit_arg *sargp, struct nfs4_ntov_table *ntovp, nfs4_attr_cmd_t cmd)
* Set vattr based on incoming fattr4 attrs - used by setattr.
* Set response mask. Ignore any values
that are not writable vattr attrs.
int error = 0;
int i;
char *attrs = fattrp->attrlist4;
uint32_t attrslen = fattrp->attrlist4_len;
XDR xdr;
nfsstat4 status = NFS4_OK;
vnode_t *vp = cs->vp;
union nfs4_attr_u *na;
uint8_t *amap;
* Make sure that maximum attribute number can be expressed as an
* 8 bit quantity.
#ifndef lint ?
Yes
No
ASSERT(NFS4_MAXNUM_ATTRS <= (UINT8_MAX + 1));
vp == NULL ?
Yes
No
resp ?
cs->access == CS_ACCESS_DENIED ?
Yes
No
*resp = 0;
No
resp ?
sargp->op = cmd;
sargp->cs = cs;
may be set later
return (NFS4ERR_NOFILEHANDLE);
Yes
*resp = 0;
No
sargp->sbp is set by the caller
sargp->flag = 0;
sargp->vap->va_mask = 0;
sargp->rdattr_error = NFS4_OK;
sargp->rdattr_error_req = FALSE;
return (NFS4ERR_ACCESS);
xdrmem_create(&xdr, attrs, attrslen, XDR_DECODE);
na = ntovp->na;
amap = ntovp->amap;
* The following loop iterates on the nfs4_ntov_map checking
* if the fbit is set in the requested bitmap.
* If set then we process the arguments using the
* rfs4_fattr4 conversion functions to populate the setattr
* vattr and va_mask. Any settable attrs that are not using vattr
* will be set in this loop.
i = 0
i < nfs4_ntov_map_size ?
Yes
No
* If setattr, must be a writable attr.
* If verify/nverify, must be a readable attr.
!(fattrp->attrmask & nfs4_ntov_map[i].fbit) ?
No
Yes
(error = (*nfs4_ntov_map[i].sv_getit)(NFS4ATTR_SUPPORTED, sargp, NULL)) != 0 ?
* Decode the attribute to set/verify
No
Yes
error = decode_fattr4_attr(cmd, sargp, nfs4_ntov_map[i].nval, &xdr, resp ? resp : NULL, na);
Yes
* Client tries to set/verify an
* unsupported attribute, tries to set
* a read only attr or verify a write
* only one - error!
error ?
Yes
No
error != 0 ?
*amap++ = (uint8_t)nfs4_ntov_map[i].nval;
na++;
(ntovp->attrcnt)++;
Yes
No
status = (error == ENOTSUP ? NFS4ERR_ATTRNOTSUPP : puterrno4(error));
xdrmem_destroy(&xdrs);
NO-OP
nfs4_ntov_map[i].vfsstat ?
Yes
No
return (status);
ntovp->vfsstat = TRUE;
i++
End

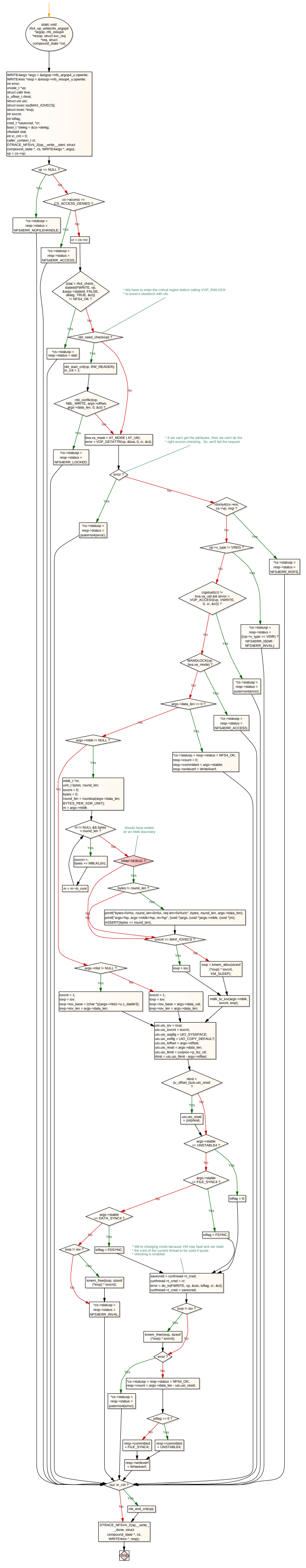


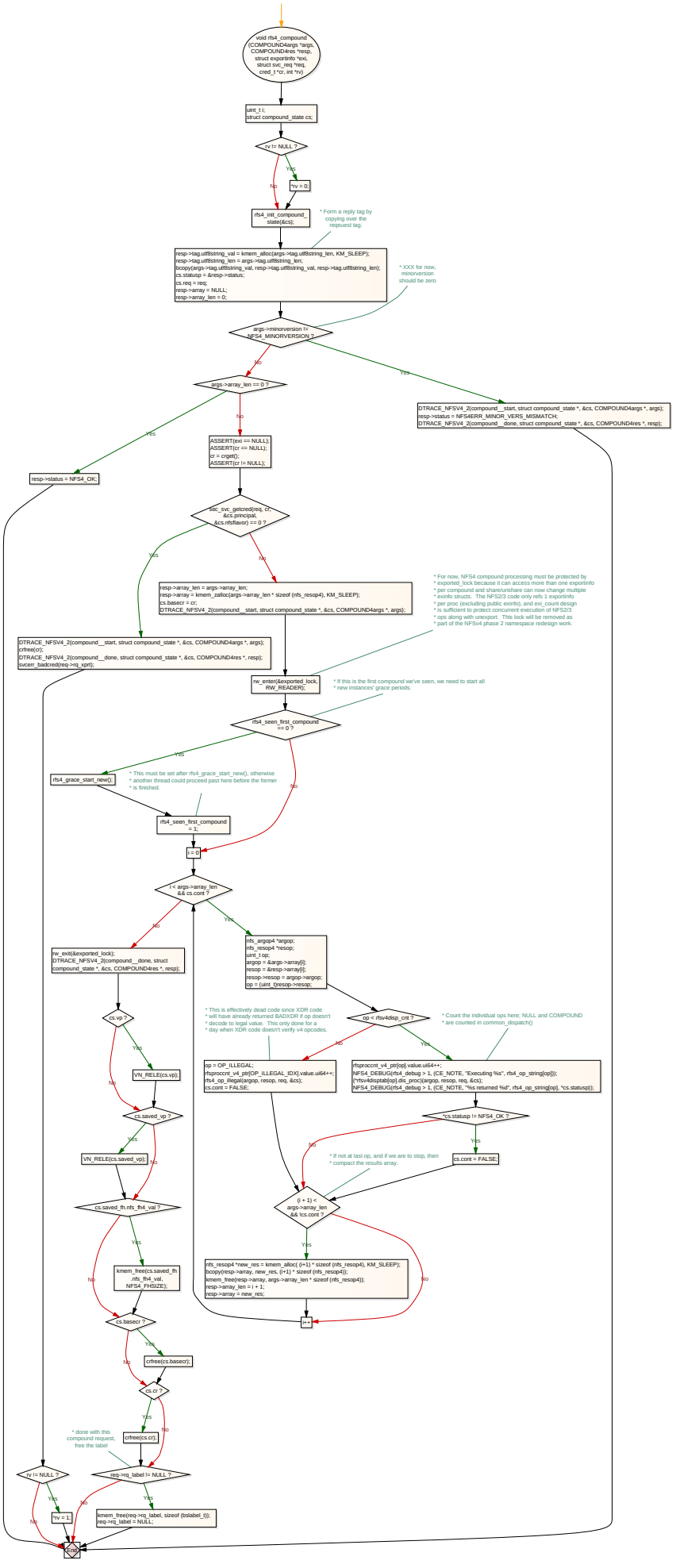
static nfsstat4 do_rfs4_op_setattr (bitmap4 *resp, fattr4 *fattrp, struct compound_state *cs, stateid4 *stateid)
int error = 0;
struct nfs4_svgetit_arg sarg;
bool_t trunc;
nfsstat4 status = NFS4_OK;
cred_t *cr = cs->cr;
vnode_t *vp = cs->vp;
struct nfs4_ntov_table ntov;
struct statvfs64 sb;
struct vattr bva;
struct flock64 bf;
int in_crit = 0;
uint_t saved_mask = 0;
caller_context_t ct;
*resp = 0;
sarg.sbp = &sb;
sarg.is_referral = B_FALSE;
nfs4_ntov_table_init(&ntov);
status = do_rfs4_set_attrs(resp, fattrp, cs, &sarg, &ntov, NFS4ATTR_SETIT);
status != NFS4_OK ?
Yes
No
(sarg.vap->va_mask == 0) && (! (fattrp->attrmask & FATTR4_ACL_MASK)) ?
* If we got a request to set the ACL and the MODE, only
* allow changing VSUID, VSGID, and VSVTX. Attempting
* to change any other bits, along with setting an ACL,
* gives NFS4ERR_INVAL.
* failed set attrs
(fattrp->attrmask & FATTR4_ACL_MASK) && (fattrp->attrmask & FATTR4_MODE_MASK) ?
* no further work to be done
vattr_t va;
va.va_mask = AT_MODE;
error = VOP_GETATTR(vp, &va, 0, cs->cr, NULL);
error ?
(sarg.vap->va_mode ^ va.va_mode) & ~(VSUID | VSGID | VSVTX) ?
Check stateid only if size has been set
status = puterrno4(error);
sarg.vap->va_mask & AT_SIZE ?
status = NFS4ERR_INVAL;
trunc = (sarg.vap->va_size == 0);
status = rfs4_check_stateid(FWRITE, cs->vp, stateid, trunc, &cs->deleg, sarg.vap->va_mask & AT_SIZE, &ct);
ct.cc_sysid = 0;
ct.cc_pid = 0;
ct.cc_caller_id = nfs4_srv_caller_id;
ct.cc_flags = CC_DONTBLOCK;
XXX start of possible race with delegations
* We need to specially handle size changes because it is
* possible for the client to create a file with read-only
* modes, but with the file opened for writing. If the client
* then tries to set the file size, e.g. ftruncate(3C),
* fcntl(F_FREESP), the normal access checking done in
* VOP_SETATTR would prevent the client from doing it even though
* it should be allowed to do so. To get around this, we do the
* access checking for ourselves and use VOP_SPACE which doesn't
* do the access checking.
* Also the client should not be allowed to change the file
* size if there is a conflicting non-blocking mandatory lock in
* the region of the change.
status != NFS4_OK ?
vp->v_type == VREG && (sarg.vap->va_mask & AT_SIZE) ?
u_offset_t offset;
ssize_t length;
* ufs_setattr clears AT_SIZE from vap->va_mask, but
* before returning, sarg.vap->va_mask is used to
* generate the setattr reply bitmap. We also clear
* AT_SIZE below before calling VOP_SPACE. For both
* of these cases, the va_mask needs to be saved here
* and restored after calling VOP_SETATTR.
saved_mask = sarg.vap->va_mask;
* Check any possible conflict due to NBMAND locks.
* Get into critical region before VOP_GETATTR, so the
* size attribute is valid when checking conflicts.
nbl_need_check(vp) ?
nbl_start_crit(vp, RW_READER);
in_crit = 1;
bva.va_mask = AT_UID|AT_SIZE;
error = VOP_GETATTR(vp, &bva, 0, cr, &ct) ?
in_crit ?
status = puterrno4(error);
sarg.vap->va_size < bva.va_size ?
offset = sarg.vap->va_size;
length = bva.va_size - sarg.vap->va_size;
offset = bva.va_size;
length = sarg.vap->va_size - bva.va_size;
nbl_conflict(vp, NBL_WRITE, offset, length, 0, &ct) ?
crgetuid(cr) == bva.va_uid ?
status = NFS4ERR_LOCKED;
sarg.vap->va_mask &= ~AT_SIZE;
bf.l_type = F_WRLCK;
bf.l_whence = 0;
bf.l_start = (off64_t)sarg.vap->va_size;
bf.l_len = 0;
bf.l_sysid = 0;
bf.l_pid = 0;
error = VOP_SPACE(vp, F_FREESP, &bf, FWRITE, (offset_t)sarg.vap->va_size, cr, &ct);
!error && sarg.vap->va_mask != 0 ?
error = VOP_SETATTR(vp, sarg.vap, sarg.flag, cr, &ct);
restore va_mask -- ufs_setattr clears AT_SIZE
saved_mask & AT_SIZE ?
sarg.vap->va_mask |= AT_SIZE;
* If an ACL was being set, it has been delayed until now,
* in order to set the mode (via the VOP_SETATTR() above) first.
(! error) && (fattrp->attrmask & FATTR4_ACL_MASK) ?
int i;
i = 0
i < NFS4_MAXNUM_ATTRS ?
ntov.amap[i] == FATTR4_ACL ?
i++
i < NFS4_MAXNUM_ATTRS ?
error = (*nfs4_ntov_map [FATTR4_ACL].sv_getit)(NFS4ATTR_SETIT, &sarg, &ntov.na[i]);
NFS4_DEBUG(rfs4_debug, (CE_NOTE, "do_rfs4_op_setattr: " "unable to find ACL in fattr4"));
error = EINVAL;
error == 0 ?
error == ENOTSUP ?
*resp |= FATTR4_ACL_MASK;
check if a monitor detected a delegation conflict
error ?
* Force modified metadata out to stable storage.
(void) rfs4_verify_attr(&sarg, resp, &ntov);
status = NFS4ERR_ATTRNOTSUPP;
error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK) ?
(void) VOP_FSYNC(vp, FNODSYNC, cr, &ct);
* Set response bitmap
status = puterrno4(error);
status = NFS4ERR_DELAY;
* Set the response bitmap when setattr failed.
* If VOP_SETATTR partially succeeded, test by doing a
* VOP_GETATTR on the object and comparing the data
* to the setattr arguments.
nfs4_vmask_to_nmask_set (sarg.vap->va_mask, resp);
(void) rfs4_verify_attr(&sarg, resp, &ntov);
Return early and already have a NFSv4 error
done: * Except for nfs4_vmask_to_nmask_set(), vattr --> fattr
* conversion sets both readable and writeable NFS4 attrs
* for AT_MTIME and AT_ATIME. The line below masks out
* unrequested attrs from the setattr result bitmap. This
* is placed after the done: label to catch the ATTRNOTSUP
* case.
*resp &= fattrp->attrmask;
in_crit ?
nbl_end_crit(vp);
nfs4_ntov_table_free(&ntov, &sarg);
return (status);
End

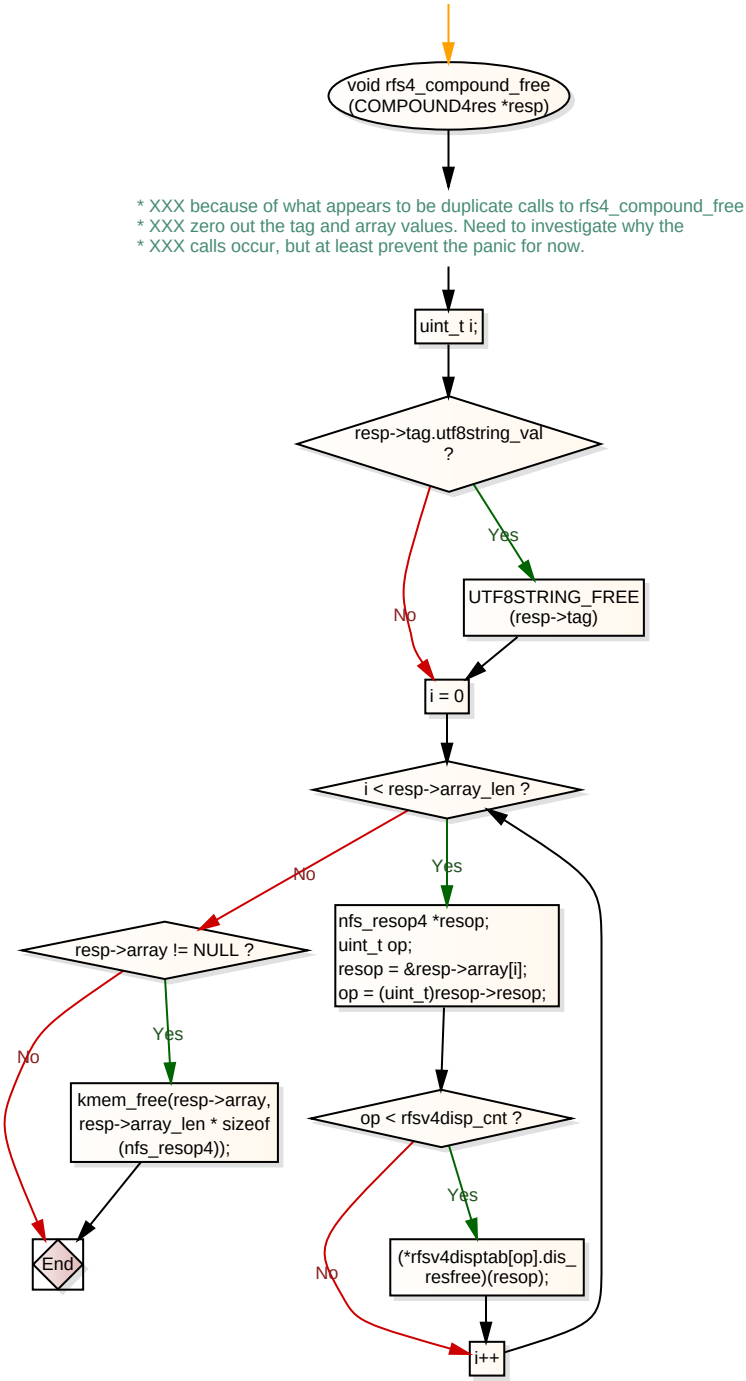

static void rfs4_op_setattr(nfs_ argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
SETATTR4args *args = &argop->nfs_argop4_u.opsetattr;
SETATTR4res *resp = &resop->nfs_resop4_u.opsetattr;
bslabel_t *clabel;
DTRACE_NFSV4_2(op__setattr__start, struct compound_state *, cs, SETATTR4args *, args);
cs->vp == NULL ?
No
Yes
* If there is an unshared filesystem mounted on this vnode,
* do not allow to setattr on this vnode.
vn_ismntpt(cs->vp) ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
resp->attrsset = 0;
*cs->statusp = resp->status = NFS4ERR_ACCESS;
rdonly4(cs->exi, cs->vp, req) ?
No
Yes
check label before setting attributes
is_system_labeled() ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_ROFS;
ASSERT(req->rq_label != NULL);
clabel = req->rq_label;
DTRACE_PROBE2(tx__rfs4__log__info__opsetattr__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, cs->vp, EQUALITY_CHECK, cs->exi) ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_ACCESS;
*cs->statusp = resp->status = do_rfs4_op_setattr(&resp->attrsset, &args->obj_attributes, cs, &args->stateid);
out: DTRACE_NFSV4_2(op__setattr__done, struct compound_state *, cs, SETATTR4res *, resp);
End

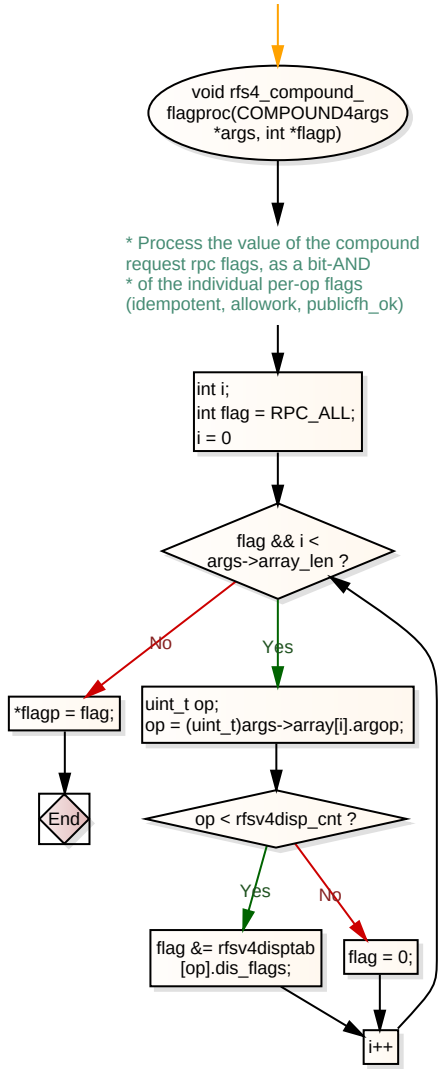

static void rfs4_op_verify(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
* verify and nverify are exactly the same, except that nverify
* succeeds when some argument changed, and verify succeeds when
* when none changed.
VERIFY4args *args = &argop->nfs_argop4_u.opverify;
VERIFY4res *resp = &resop->nfs_resop4_u.opverify;
int error;
struct nfs4_svgetit_arg sarg;
struct statvfs64 sb;
struct nfs4_ntov_table ntov;
DTRACE_NFSV4_2(op__verify__start, struct compound_state *, cs, VERIFY4args *, args);
cs->vp == NULL ?
Yes
No
sarg.sbp = &sb;
sarg.is_referral = B_FALSE;
nfs4_ntov_table_init(&ntov);
resp->status = do_rfs4_set_attrs(NULL, &args->obj_attributes, cs, &sarg, &ntov, NFS4ATTR_VERIT);
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
resp->status != NFS4_OK ?
Yes
No
error = rfs4_verify_attr(&sarg, NULL, &ntov);
switch (error)
0 ?
No
-1 ?
Yes
Yes
default
* do_rfs4_set_attrs will try to verify systemwide attrs,
* so could return -1 for "no match".
resp->status = NFS4_OK;
resp->status == -1 ?
Yes
No
resp->status = NFS4ERR_NOT_SAME;
resp->status = NFS4ERR_NOT_SAME;
resp->status = puterrno4(error);
done: *cs->statusp = resp->status; nfs4_ntov_table_free(&ntov, &sarg);
out: DTRACE_NFSV4_2(op__verify__done, struct compound_state *, cs, VERIFY4res *, resp);
End

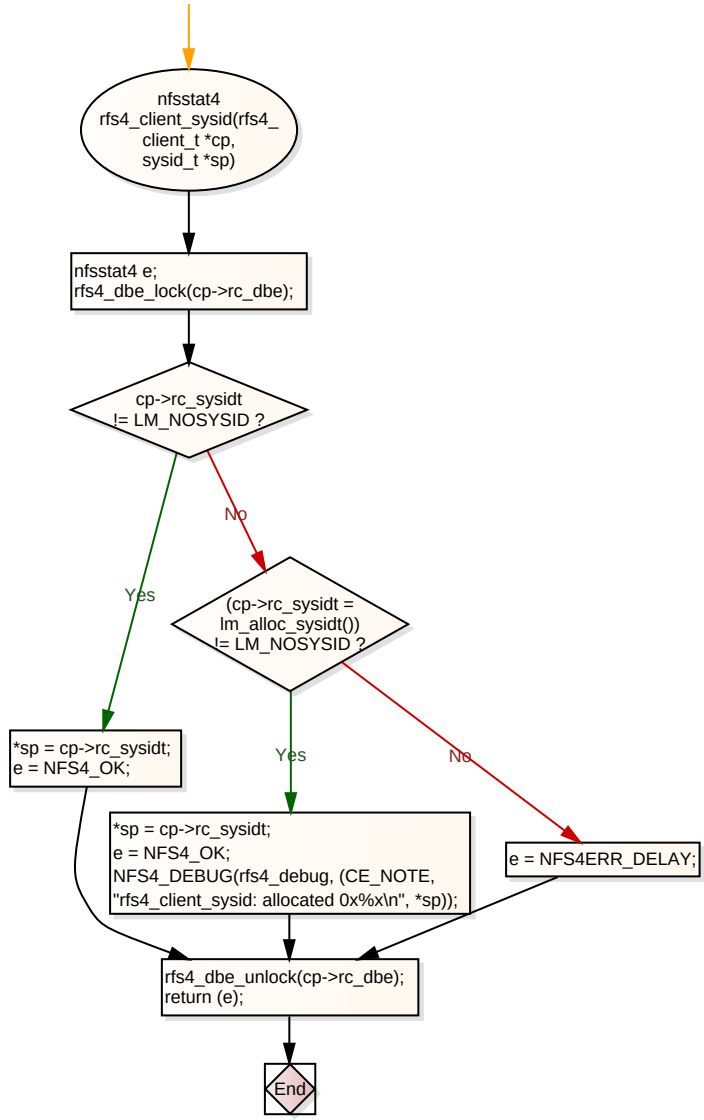

static void rfs4_op_nverify(nfs_ argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
* verify and nverify are exactly the same, except that nverify
* succeeds when some argument changed, and verify succeeds when
* when none changed.
NVERIFY4args *args = &argop->nfs_argop4_u.opnverify;
NVERIFY4res *resp = &resop->nfs_resop4_u.opnverify;
int error;
struct nfs4_svgetit_arg sarg;
struct statvfs64 sb;
struct nfs4_ntov_table ntov;
DTRACE_NFSV4_2(op__nverify__start, struct compound_state *, cs, NVERIFY4args *, args);
cs->vp == NULL ?
Yes
No
sarg.sbp = &sb;
sarg.is_referral = B_FALSE;
nfs4_ntov_table_init(&ntov);
resp->status = do_rfs4_set_attrs(NULL, &args->obj_attributes, cs, &sarg, &ntov, NFS4ATTR_VERIT);
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
DTRACE_NFSV4_2(op__nverify__done, struct compound_state *, cs, NVERIFY4res *, resp);
resp->status != NFS4_OK ?
Yes
No
error = rfs4_verify_attr(&sarg, NULL, &ntov);
switch (error)
0 ?
No
Yes
-1 ?
Yes
default
* do_rfs4_set_attrs will try to verify systemwide attrs,
* so could return -1 for "no match".
resp->status = NFS4ERR_SAME;
resp->status == -1 ?
Yes
No
resp->status = NFS4_OK;
resp->status = NFS4_OK;
resp->status = puterrno4(error);
done: *cs->statusp = resp->status;
nfs4_ntov_table_free(&ntov, &sarg);
DTRACE_NFSV4_2(op__nverify__done, struct compound_state *, cs, NVERIFY4res *, resp);
End

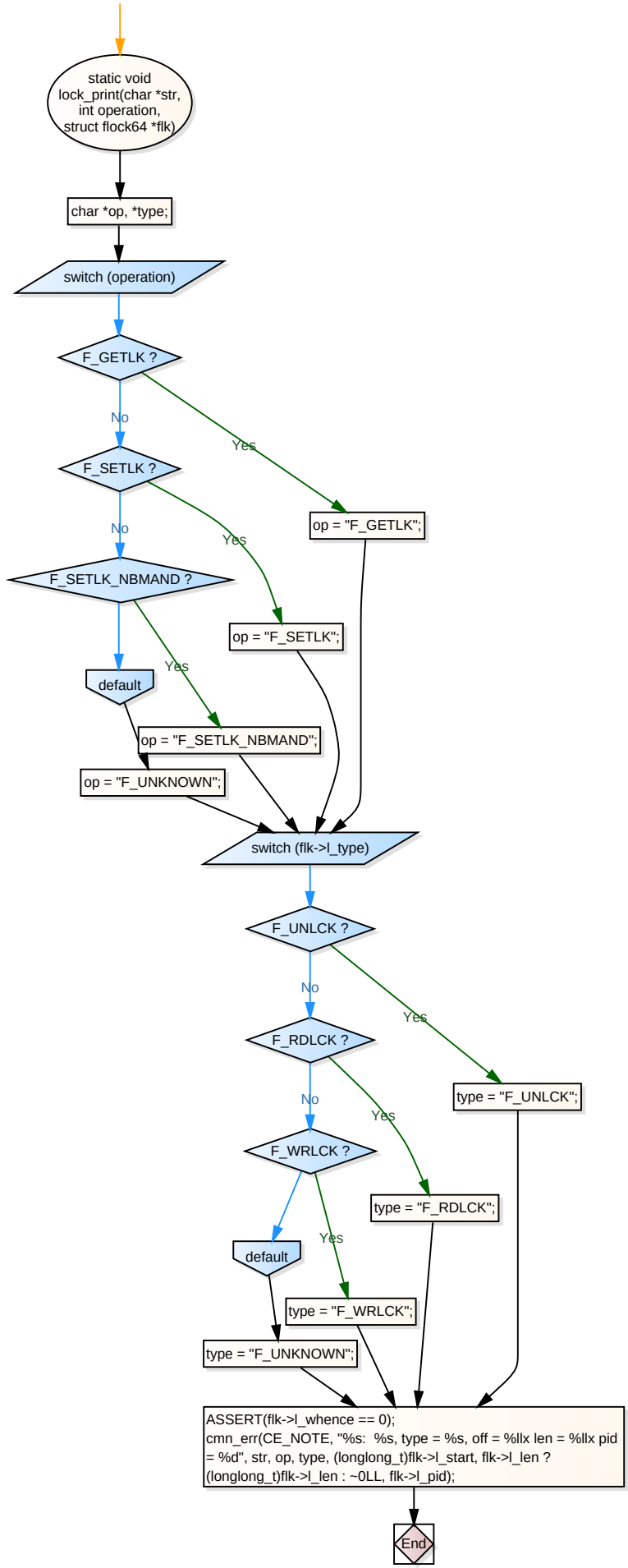

static void rfs4_op_write(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
WRITE4args *args = &argop->nfs_argop4_u.opwrite; WRITE4res *resp = &resop->nfs_resop4_u.opwrite; int error; vnode_t *vp; struct vattr bva; u_offset_t rlimit; struct uio uio; struct iovec iov[MAX_IOVECS]; struct iovec *iovp; int iovcnt; int ioflag; cred_t *savecred, *cr; bool_t *deleg = &cs->deleg; nfsstat4 stat; int in_crit = 0; caller_context_t ct; DTRACE_NFSV4_2(op__write__start, struct compound_state *, cs, WRITE4args *, args); vp = cs->vp;
vp == NULL ?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
cs->access == CS_ACCESS_DENIED ?
*cs->statusp = resp->status = NFS4ERR_ACCESS;
cr = cs->cr;
(stat = rfs4_check_stateid(FWRITE, vp, &args->stateid, FALSE, deleg, TRUE, &ct)) != NFS4_OK ?
* We have to enter the critical region before calling VOP_RWLOCK
* to avoid a deadlock with ufs.
nbl_need_check(vp) ?
*cs->statusp = resp->status = stat;
nbl_start_crit(vp, RW_READER); in_crit = 1;
nbl_conflict(vp, NBL_WRITE, args->offset, args->data_len, 0, &ct) ?
*cs->statusp = resp->status = NFS4ERR_LOCKED;
bva.va_mask = AT_MODE | AT_UID; error = VOP_GETATTR(vp, &bva, 0, cr, &ct);
* If we can't get the attributes, then we can't do the
* right access checking. So, we'll fail the request.
error ?
*cs->statusp = resp->status = puterrno4(error);
rdonly4(cs->exi, cs->vp, req) ?
*cs->statusp = resp->status = NFS4ERR_ROFS;
vp->v_type != VREG ?
*cs->statusp = resp->status = ((vp->v_type == VDIR) ? NFS4ERR_ISDIR : NFS4ERR_INVAL);
crgetuid(cr) != bva.va_uid && (error = VOP_ACCESS(vp, VWRITE, 0, cr, &ct)) ?
*cs->statusp = resp->status = puterrno4(error);
MANDLOCK(vp, bva.va_mode) ?
*cs->statusp = resp->status = NFS4ERR_ACCESS;
args->data_len == 0 ?
*cs->statusp = resp->status = NFS4_OK; resp->count = 0; resp->committed = args->stable; resp->writeverf = Write4verf;
args->mblk != NULL ?
mblk_t *m; uint_t bytes, round_len; iovcnt = 0; bytes = 0; round_len = roundup(args->data_len, BYTES_PER_XDR_UNIT); m = args->mblk
m != NULL && bytes < round_len ?
should have ended on an mblk boundary
iovcnt++; bytes += MBLKL(m);
m = m->b_cont
#ifdef DEBUG ?
bytes != round_len ?
printf("bytes=0x%x, round_len=0x%x, req len=0x%x\n", bytes, round_len, args->data_len); printf("args=%p, args->mblk=%p, m=%p", (void *)args, (void *)args->mblk, (void *)m); ASSERT(bytes == round_len);
iovcnt <= MAX_IOVECS ?
iovp = iov;
iovp = kmem_alloc(sizeof (*iovp) * iovcnt, KM_SLEEP);
mblk_to_iov(args->mblk, iovcnt, iovp);
args->rlist != NULL ?
iovcnt = 1; iovp = iov; iovp->iov_base = (char *)((args->rlist)->u.c_daddr3); iovp->iov_len = args->data_len;
iovcnt = 1; iovp = iov; iovp->iov_base = args->data_val; iovp->iov_len = args->data_len;
uio.uio_iov = iovp; uio.uio_iovcnt = iovcnt; uio.uio_segflg = UIO_SYSSPACE; uio.uio_extflg = UIO_COPY_DEFAULT; uio.uio_loffset = args->offset; uio.uio_resid = args->data_len; uio.uio_llimit = curproc->p_fsz_ctl; rlimit = uio.uio_llimit - args->offset;
rlimit < (u_offset_t)uio.uio_resid ?
uio.uio_resid = (int)rlimit;
args->stable == UNSTABLE4 ?
ioflag = 0;
args->stable == FILE_SYNC4 ?
ioflag = FSYNC;
args->stable == DATA_SYNC4 ?
ioflag = FDSYNC;
iovp != iov ?
kmem_free(iovp, sizeof (*iovp) * iovcnt);
*cs->statusp = resp->status = NFS4ERR_INVAL;
* We're changing creds because VM may fault and we need
* the cred of the current thread to be used if quota
* checking is enabled.
savecred = curthread->t_cred; curthread->t_cred = cr; error = do_io(FWRITE, vp, &uio, ioflag, cr, &ct); curthread->t_cred = savecred;
iovp != iov ?
kmem_free(iovp, sizeof (*iovp) * iovcnt);
error ?
*cs->statusp = resp->status = puterrno4(error);
*cs->statusp = resp->status = NFS4_OK; resp->count = args->data_len - uio.uio_resid;
ioflag == 0 ?
resp->committed = FILE_SYNC4;
resp->committed = UNSTABLE4;
resp->writeverf = Write4verf;
out: in_crit ?
nbl_end_crit(vp);
DTRACE_NFSV4_2(op__write__done, struct compound_state *, cs, WRITE4res *, resp);
End

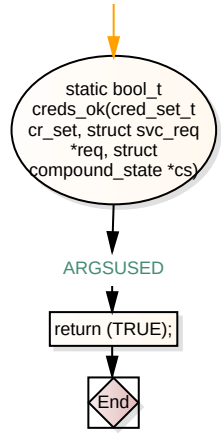

void nfs4_compound(COMPOUND4args *args, COMPOUND4res *resp, struct exportinfo *exi, struct svc_req *rtnq, cred_t *cr, int *rv)
uint_t i; struct compound_state cs;
rv != NULL ?
Yes
No
*rv = 0;
rfs4_init_compound_state(&cs);
* Form a reply tag by copying over the request tag.
resp->tag.utf8string_val = kmem_alloc(args->tag.utf8string_len, KM_SLEEP);
resp->tag.utf8string_len = args->tag.utf8string_len;
bcopy(args->tag.utf8string_val, resp->tag.utf8string_val, resp->tag.utf8string_len);
cs.statusp = &resp->status;
cs.req = req;
resp->array = NULL;
resp->array_len = 0;
* XXX for now, minorversion should be zero
args->minorversion != NFS4_MINORVERSION ?
No
Yes
args->array_len == 0 ?
Yes
No
DTRACE_NFSV4_2(compound__start, struct compound_state *, &cs, COMPOUND4args *, args);
resp->status = NFS4ERR_MINOR_VERS_MISMATCH;
DTRACE_NFSV4_2(compound__done, struct compound_state *, &cs, COMPOUND4res *, resp);
resp->status = NFS4_OK;
ASSERT(exi == NULL);
ASSERT(cr == NULL);
cr = crget();
ASSERT(cr != NULL);
sec_svc_getcred(req, cr, &cs.principal, &cs.nsflavor) == 0 ?
Yes
No
resp->array_len = args->array_len;
resp->array = kmem_zalloc(args->array_len * sizeof (nfs_resop4), KM_SLEEP);
cs.basecr = cr;
DTRACE_NFSV4_2(compound__start, struct compound_state *, &cs, COMPOUND4args *, args);
* For now, NFS4 compound processing must be protected by
* exported_lock because it can access more than one exportinfo
* per compound and share/unshare can now change multiple
* exinfo structs. The NFSv2/3 code only refs 1 exportinfo
* per proc (excluding public exinfo), and exi_count design
* is sufficient to protect concurrent execution of NFSv2/3
* ops along with unexport. This lock will be removed as
* part of the NFSv4 phase 2 namespace redesign work.
DTRACE_NFSV4_2(compound__start, struct compound_state *, &cs, COMPOUND4args *, args);
crfree(cr);
DTRACE_NFSV4_2(compound__done, struct compound_state *, &cs, COMPOUND4res *, resp);
sucerr_badcred(req->rq_xprt);
rw_enter(&exported_lock, RW_READER);
* If this is the first compound we've seen, we need to start all
* new instances' grace periods.
rfs4_seen_first_compound == 0 ?
Yes
No
rfs4_grace_start_new();
* This must be set after rfs4_grace_start_new(), otherwise
* another thread could proceed past here before the former
* is finished.
rfs4_seen_first_compound = 1;
i = 0;
i < args->array_len && cs.cont ?
No
Yes
rw_exit(&exported_lock);
DTRACE_NFSV4_2(compound__done, struct compound_state *, &cs, COMPOUND4res *, resp);
nfs_argop4 *argop;
nfs_resop4 *resop;
uint_t op;
argop = &args->array[i];
resop = &resp->array[i];
resop->resop = argop->argop;
op = (uint_t)resop->resop;
cs.vp ?
Yes
No
VN_RELE(cs.vp);
cs.saved_vp ?
Yes
No
VN_RELE(cs.saved_vp);
cs.saved_fh.nfs_fh4_val ?
Yes
No
kmem_free(cs.saved_fh.nfs_fh4_val, NFS4_FHSIZE);
cs.basecr ?
Yes
No
crfree(cs.basecr);
cs.cr ?
Yes
No
crfree(cs.cr);
* done with this compound request, free the label
req->rq_label != NULL ?
Yes
No
kmem_free(req->rq_label, sizeof (bslabel_t));
req->rq_label = NULL;
rv != NULL ?
Yes
No
*rv = 1;
End
* This is effectively dead code since XDR code
* will have already returned BADXDR if op doesn't
* decode to legal value. This only done for a
* day when XDR code doesn't verify v4 opcodes.
op < rfsv4disp_cnt ?
No
Yes
* Count the individual ops here; NULL and COMPOUND
* are counted in common_dispatch()
op = OP_ILLEGAL;
rfsproccnt_v4_ptr[OP_ILLEGAL_IDX].value.ui64++;
rfs4_op_illegal(argop, resop, req, &cs);
cs.cont = FALSE;
rfsproccnt_v4_ptr[op].value.ui64++;
NFS4_DEBUG(rfs4_debug > 1, (CE_NOTE, "Executing %s", rfs4_op_string[op]));
(*rfsv4disptab[op].dis_proc)(argop, resop, req, &cs);
NFS4_DEBUG(rfs4_debug > 1, (CE_NOTE, "%s returned %d", rfs4_op_string[op], *cs.statusp));
*cs.statusp != NFS4_OK ?
No
Yes
cs.cont = FALSE;
* If not at last op, and if we are to stop, then
* compact the results array.
(i + 1) < args->array_len && !cs.cont ?
Yes
No
nfs_resop4 *new_res = kmem_alloc((i+1) * sizeof (nfs_resop4), KM_SLEEP);
bcopy(resp->array, new_res, (i+1) * sizeof (nfs_resop4));
kmem_free(resp->array, args->array_len * sizeof (nfs_resop4));
resp->array_len = i + 1;
resp->array = new_res;
i++

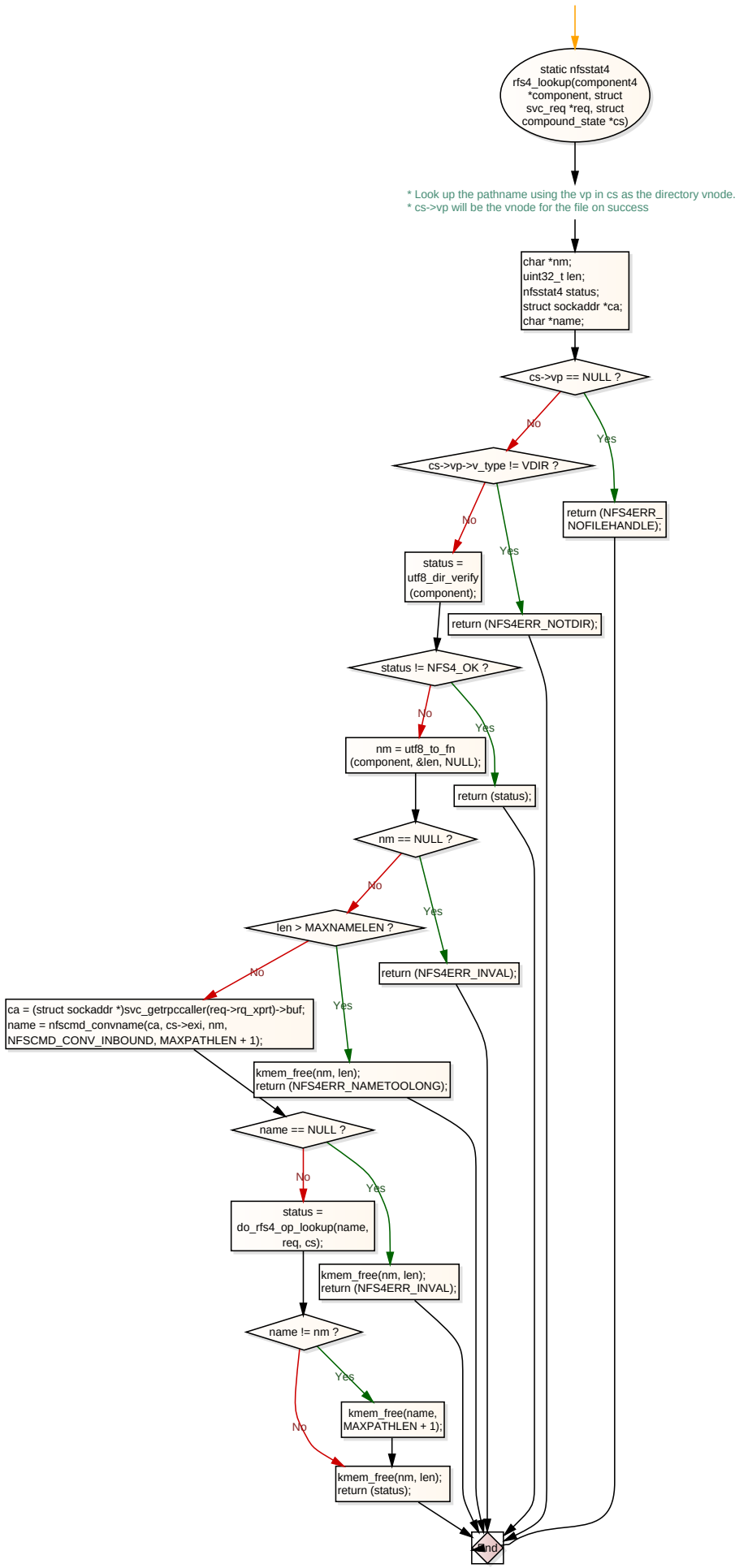

void rfs4_compound_free
(COMPOUND4res *resp)
* XXX because of what appears to be duplicate calls to rfs4_compound_free
* XXX zero out the tag and array values. Need to investigate why the
* XXX calls occur, but at least prevent the panic for now.
uint_t i;
resp->tag.utf8string_val
?
Yes
No
UTF8STRING_FREE
(resp->tag)
i = 0
i < resp->array_len ?
No
Yes
resp->array != NULL ?
nfs_resop4 *resop;
uint_t op;
resop = &resp->array[i];
op = (uint_t)resop->resop;
No
Yes
kmem_free(resp->array,
resp->array_len * sizeof
(nfs_resop4));
op < rfsv4disp_cnt ?
End
Yes
No
(*rfsv4disptab[op].dis_
resfree)(resop);
i++

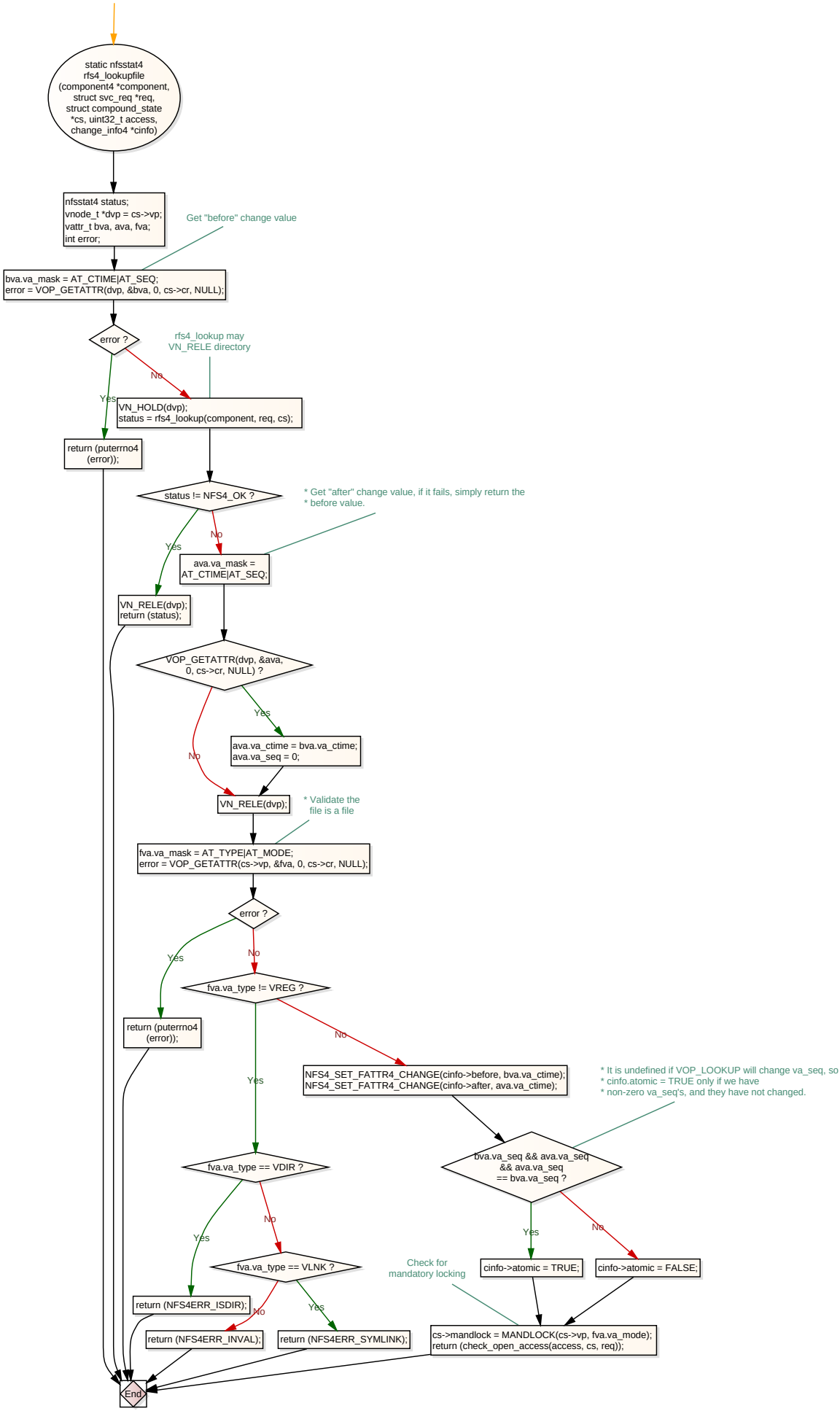

void rfs4_compound_
flagproc(COMPOUND4args
*args, int *flagp)
* Process the value of the compound
request rpc flags, as a bit-AND
* of the individual per-op flags
(idempotent, allowork, publicfh_ok)
int i;
int flag = RPC_ALL;
i = 0
flag && i <
args->array_len ?
No
Yes
*flagp = flag;
uint_t op;
op = (uint_t)args->array[i].argop;
End
op < rfsv4disp_cnt ?
Yes
No
flag &= rfsv4disptab
[op].dis_flags;
flag = 0;
i++

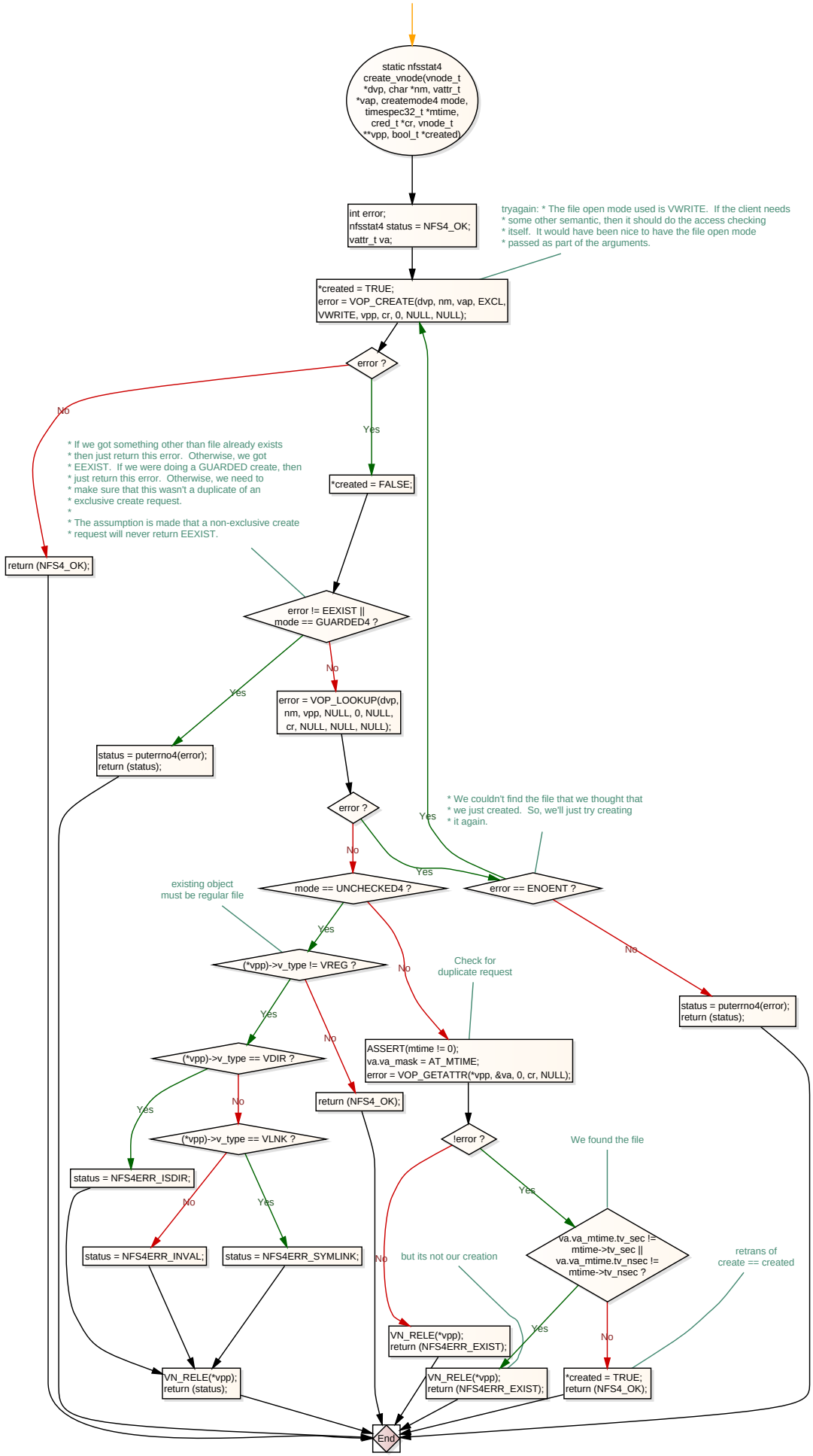

nfsstat4
rfs4_client_sysid(rfs4_
client_t *cp,
sysid_t *sp)
nfsstat4 e;
rfs4_dbe_lock(cp->rc_dbe);
cp->rc_sysidt
!= LM_NOSYSID ?
No
Yes
(cp->rc_sysidt =
lm_alloc_sysidt())
!= LM_NOSYSID ?
*sp = cp->rc_sysidt;
e = NFS4_OK;
Yes
No
*sp = cp->rc_sysidt;
e = NFS4_OK;
NFS4_DEBUG(rfs4_debug, (CE_NOTE,
"rfs4_client_sysid: allocated 0x%x\n", *sp));
e = NFS4ERR_DELAY;
rfs4_dbe_unlock(cp->rc_dbe);
return (e);
End

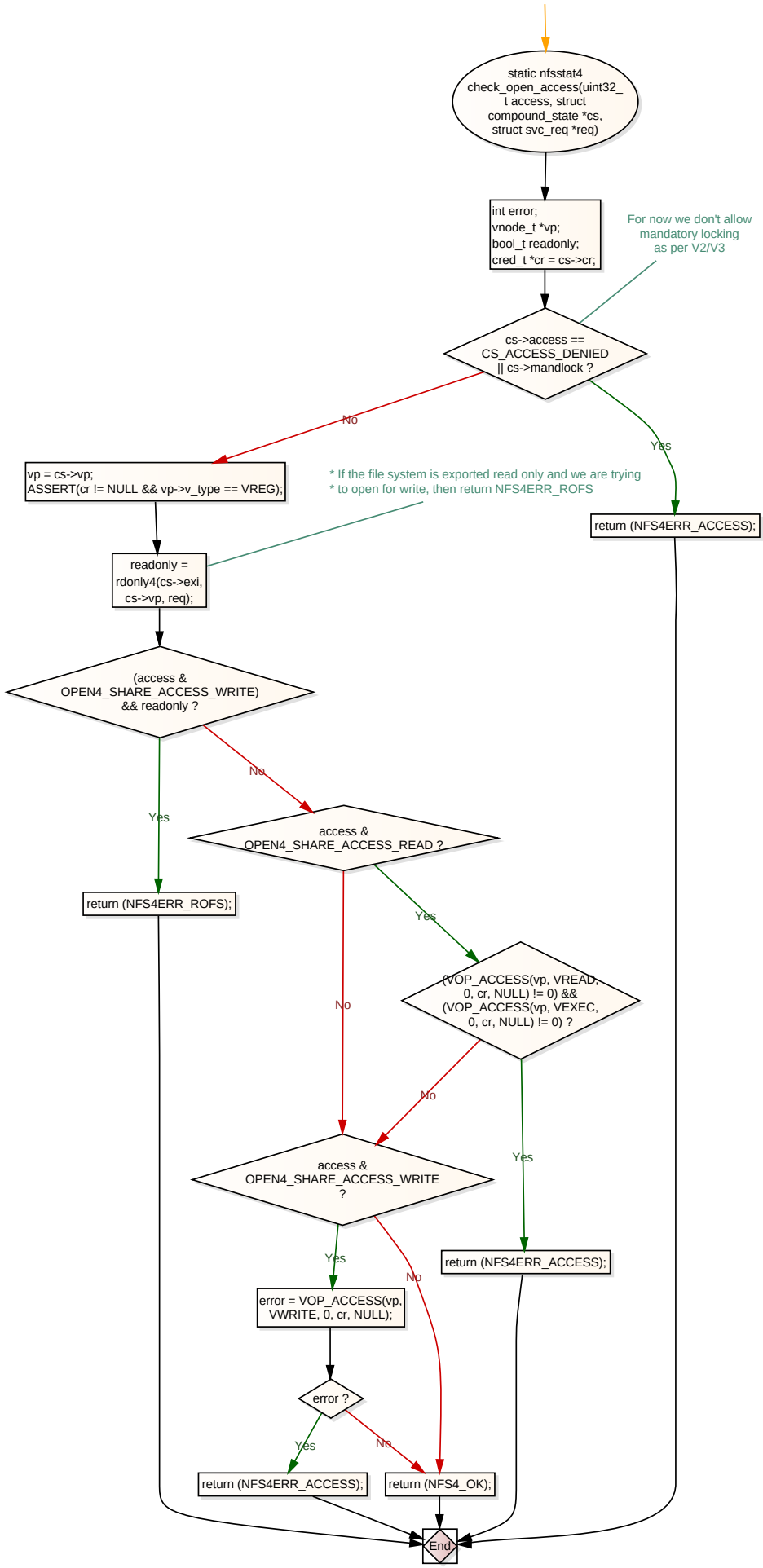

static void
lock_print(char *str,
int operation,
struct flock64 *flk)
char *op, *type;
switch (operation)
F_GETLK ?
No
Yes
op = "F_GETLK";
F_SETLK ?
No
Yes
op = "F_SETLK";
F_SETLK_NBMAND ?
Yes
op = "F_SETLK_NBMAND";
default
op = "F_UNKNOWN";
switch (flk->l_type)
F_UNLCK ?
No
Yes
type = "F_UNLCK";
F_RDLCK ?
No
Yes
type = "F_RDLCK";
F_WRLCK ?
Yes
type = "F_WRLCK";
default
type = "F_UNKNOWN";
ASSERT(flk->l_whence == 0);
cmn_err(CE_NOTE, "%s: %s, type = %s, off = %llx len = %llx pid
= %d", str, op, type, (longlong_t)flk->l_start, flk->l_len ?
(longlong_t)flk->l_len : ~0LL, flk->l_pid);
End

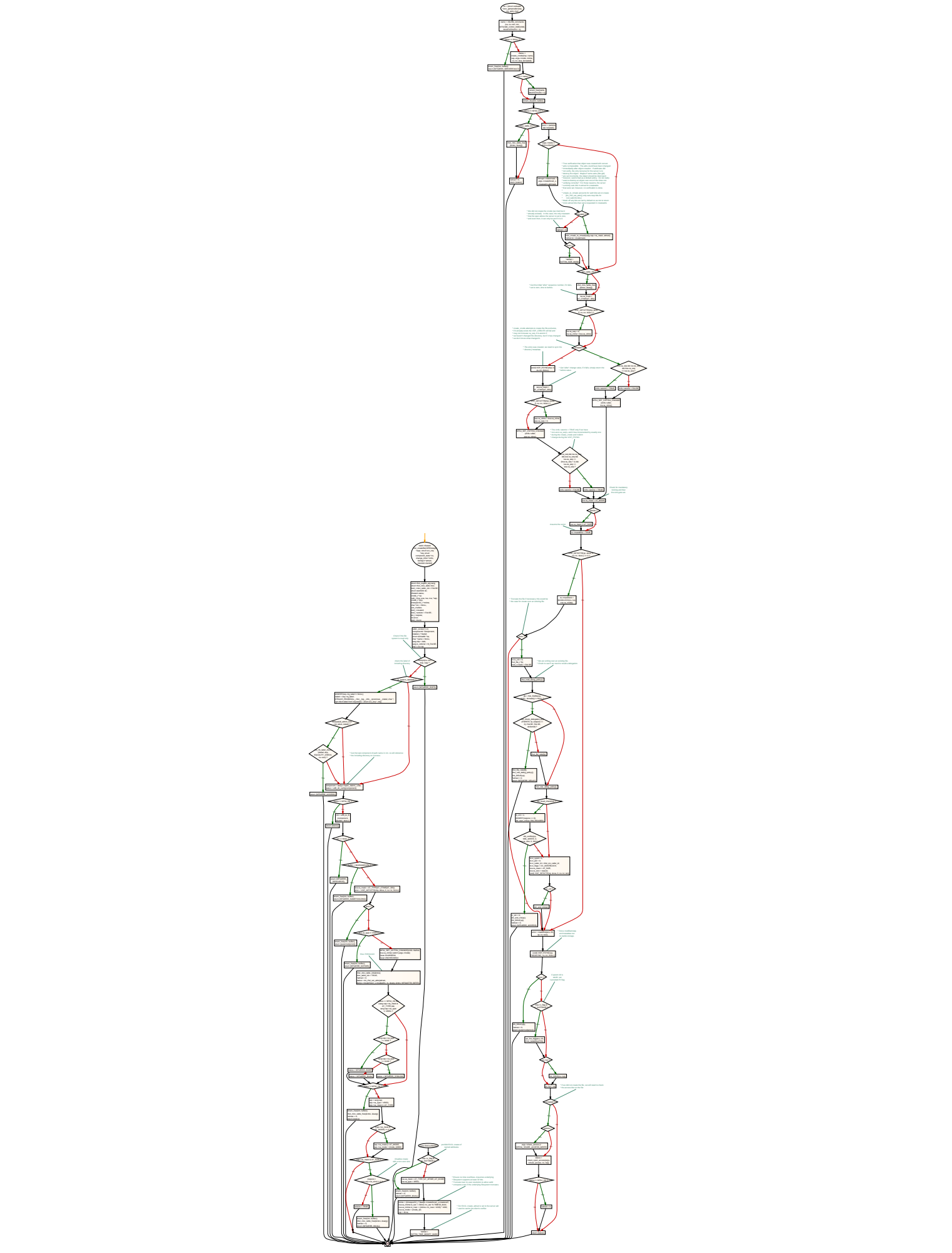

static bool_t creds_ok(cred_set_t cr_set, struct svc_req *req, struct compound_state *cs)
ARGSUSED
return (TRUE);
End

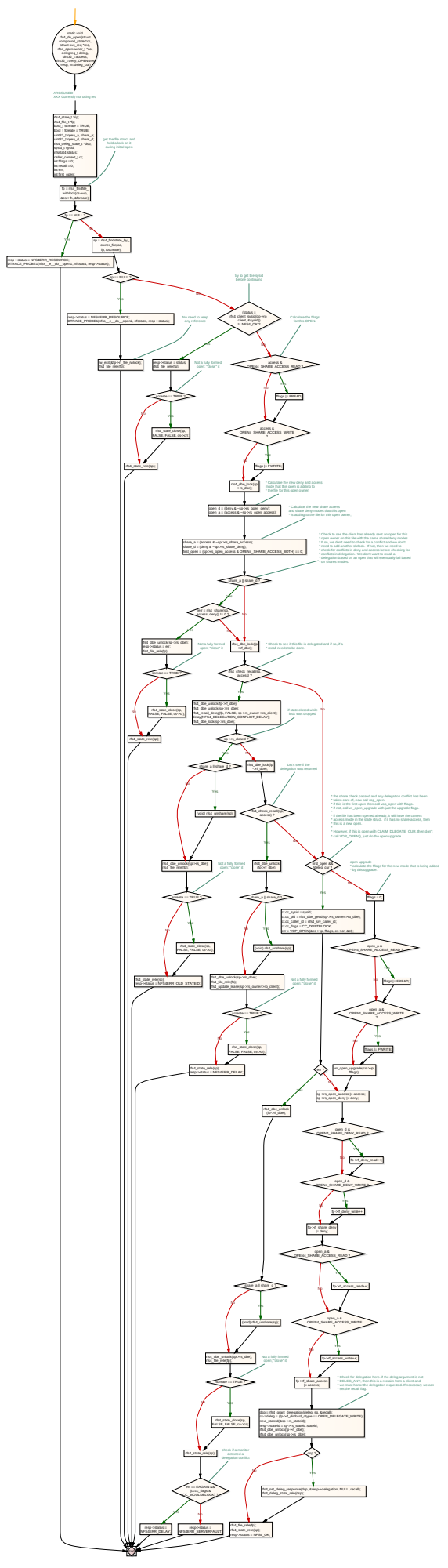

static nfsstat4 rfs4_lookup(component4 *component, struct svc_req *req, struct compound_state *cs)
* Look up the pathname using the vp in cs as the directory vnode.
* cs->vp will be the vnode for the file on success
char *nm;
uint32_t len;
nfsstat4 status;
struct sockaddr *ca;
char *name;
cs->vp == NULL ?
No
Yes
cs->vp->v_type != VDIR ?
return (NFS4ERR_NOFILEHANDLE);
No
Yes
status = utf8_dir_verify (component);
return (NFS4ERR_NOTDIR);
status != NFS4_OK ?
No
Yes
nm = utf8_to_fn (component, &len, NULL);
return (status);
nm == NULL ?
No
Yes
len > MAXNAMELEN ?
No
Yes
return (NFS4ERR_INVAL);
ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
kmem_free(nm, len);
return (NFS4ERR_NAMETOOLONG);
name == NULL ?
No
Yes
status = do_rfs4_op_lookup(name, req, cs);
kmem_free(nm, len);
return (NFS4ERR_INVAL);
name != nm ?
Yes
No
kmem_free(name, MAXPATHLEN + 1);
kmem_free(nm, len);
return (status);
end

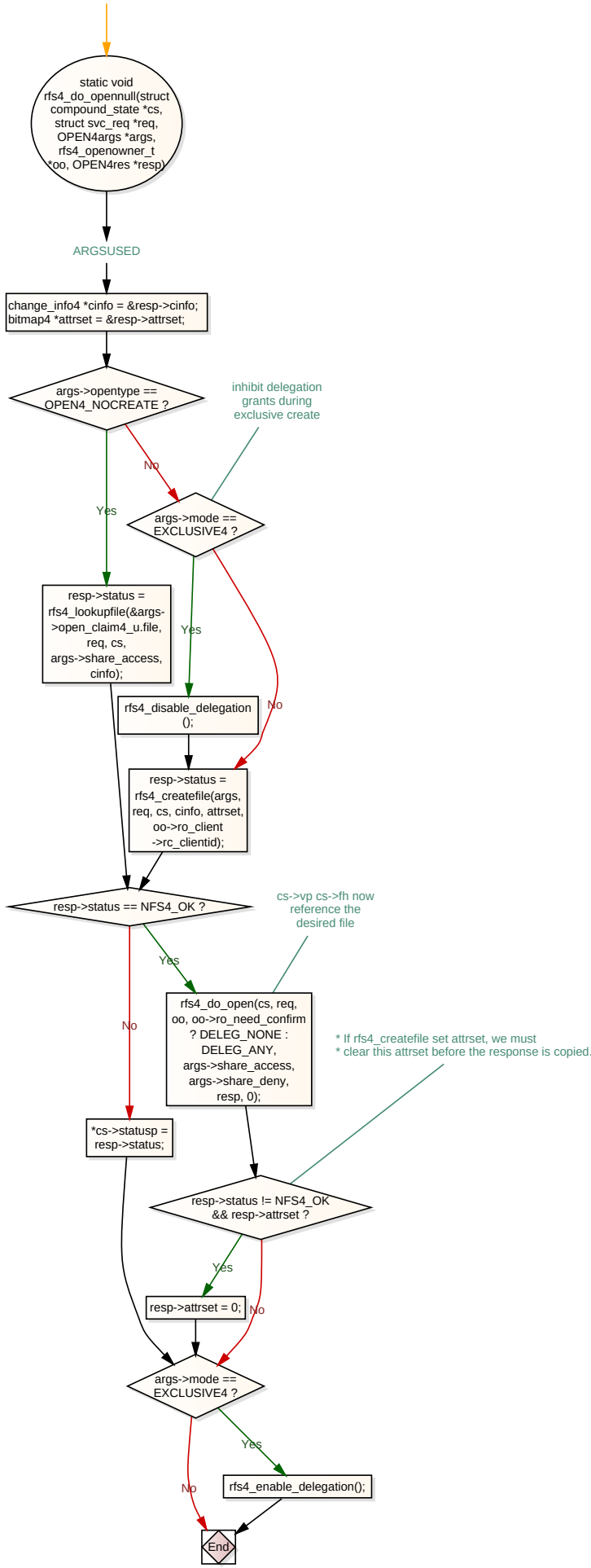

static nfsstat4
rfs4_lookupfile
(component4 *component,
struct svc_req *req,
struct compound_state
*cs, uint32_t access,
change_info4 *cinfo)
nfsstat4 status;
vnode_t *dvp = cs->vp;
vattr_t bva, ava, fva;
int error;
Get "before" change value
bva.va_mask = AT_CTIME|AT_SEQ;
error = VOP_GETATTR(dvp, &bva, 0, cs->cr, NULL);
error ?
No
Yes
rfs4_lookup may
VN_RELE directory
VN_HOLD(dvp);
status = rfs4_lookup(component, req, cs);
return (puterrno4
(error));
status != NFS4_OK ?
* Get "after" change value, if it fails, simply return the
* before value.
Yes
No
ava.va_mask =
AT_CTIME|AT_SEQ;
VN_RELE(dvp);
return (status);
VOP_GETATTR(dvp, &ava,
0, cs->cr, NULL) ?
Yes
No
ava.va_ctime = bva.va_ctime;
ava.va_seq = 0;
VN_RELE(dvp);
* Validate the
file is a file
fva.va_mask = AT_TYPE|AT_MODE;
error = VOP_GETATTR(cs->vp, &fva, 0, cs->cr, NULL);
error ?
Yes
No
fva.va_type != VREG ?
return (puterrno4
(error));
No
Yes
NFS4_SET_FATTR4_CHANGE(cinfo->before, bva.va_ctime);
NFS4_SET_FATTR4_CHANGE(cinfo->after, ava.va_ctime);
* It is undefined if VOP_LOOKUP will change va_seq, so
* cinfo.atomic = TRUE only if we have
* non-zero va_seq's, and they have not changed.
fva.va_type == VDIR ?
bva.va_seq && ava.va_seq
&& ava.va_seq
== bva.va_seq ?
Yes
No
Yes
No
fva.va_type == VLNK ?
Check for
mandatory locking
cinfo->atomic = TRUE;
cinfo->atomic = FALSE;
return (NFS4ERR_ISDIR);
No
Yes
return (NFS4ERR_INVAL);
return (NFS4ERR_SYMLINK);
cs->mandlock = MANDLOCK(cs->vp, fva.va_mode);
return (check_open_access(access, cs, req));
End

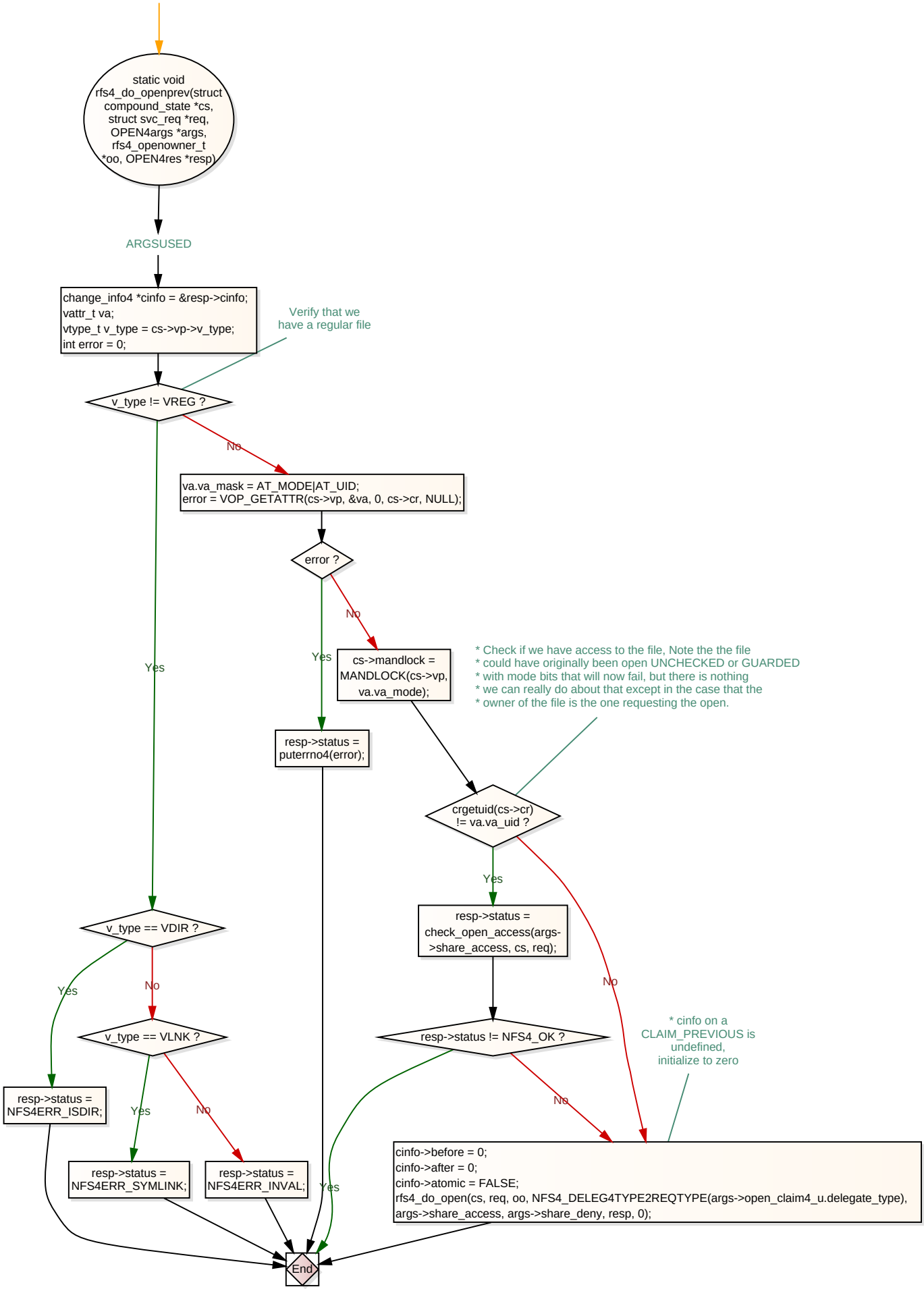

static nfsstat4 create_vnode(vnode_t *dvp, char *nm, vattr_t *vap, createmode4 mode, timespec32_t *mtime, cred_t *cr, vnode_t **vpp, bool_t *created)
int error;
nfsstat4 status = NFS4_OK;
vattr_t va;
tryagain: * The file open mode used is VWRITE. If the client needs * some other semantic, then it should do the access checking * itself. It would have been nice to have the file open mode * passed as part of the arguments.
*created = TRUE;
error = VOP_CREATE(dvp, nm, vap, EXCL, VWRITE, vpp, cr, 0, NULL, NULL);
error ?
No
Yes
* If we got something other than file already exists * then just return this error. Otherwise, we got * EEXIST. If we were doing a GUARDED create, then * just return this error. Otherwise, we need to * make sure that this wasn't a duplicate of an * exclusive create request. * * The assumption is made that a non-exclusive create * request will never return EEXIST.
*created = FALSE;
return (NFS4_OK);
error != EEXIST || mode == GUARDED4 ?
Yes
No
error = VOP_LOOKUP(dvp, nm, vpp, NULL, 0, NULL, cr, NULL, NULL, NULL);
status = puterrno4(error); return (status);
error ?
Yes
* We couldn't find the file that we thought that * we just created. So, we'll just try creating * it again.
No
Yes
mode == UNCHECKED4 ?
error == ENOENT ?
existing object must be regular file
Yes
No
(*vpp)->v_type != VREG ?
Check for duplicate request
status = puterrno4(error); return (status);
Yes
No
(*vpp)->v_type == VDIR ?
ASSERT(mtime != 0); va.va_mask = AT_MTIME; error = VOP_GETATTR(*vpp, &va, 0, cr, NULL);
Yes
No
(*vpp)->v_type == VLNK ?
return (NFS4_OK);
!error ?
We found the file
status = NFS4ERR_ISDIR;
Yes
No
Yes
status = NFS4ERR_INVAL;
status = NFS4ERR_SYMLINK;
No
but its not our creation
va.va_mtime.tv_sec != mtime->tv_sec || va.va_mtime.tv_nsec != mtime->tv_nsec ?
retrans of create == created
VN_RELE(*vpp); return (NFS4ERR_EXIST);
Yes
No
VN_RELE(*vpp); return (status);
VN_RELE(*vpp); return (NFS4ERR_EXIST);
*created = TRUE; return (NFS4_OK);
End

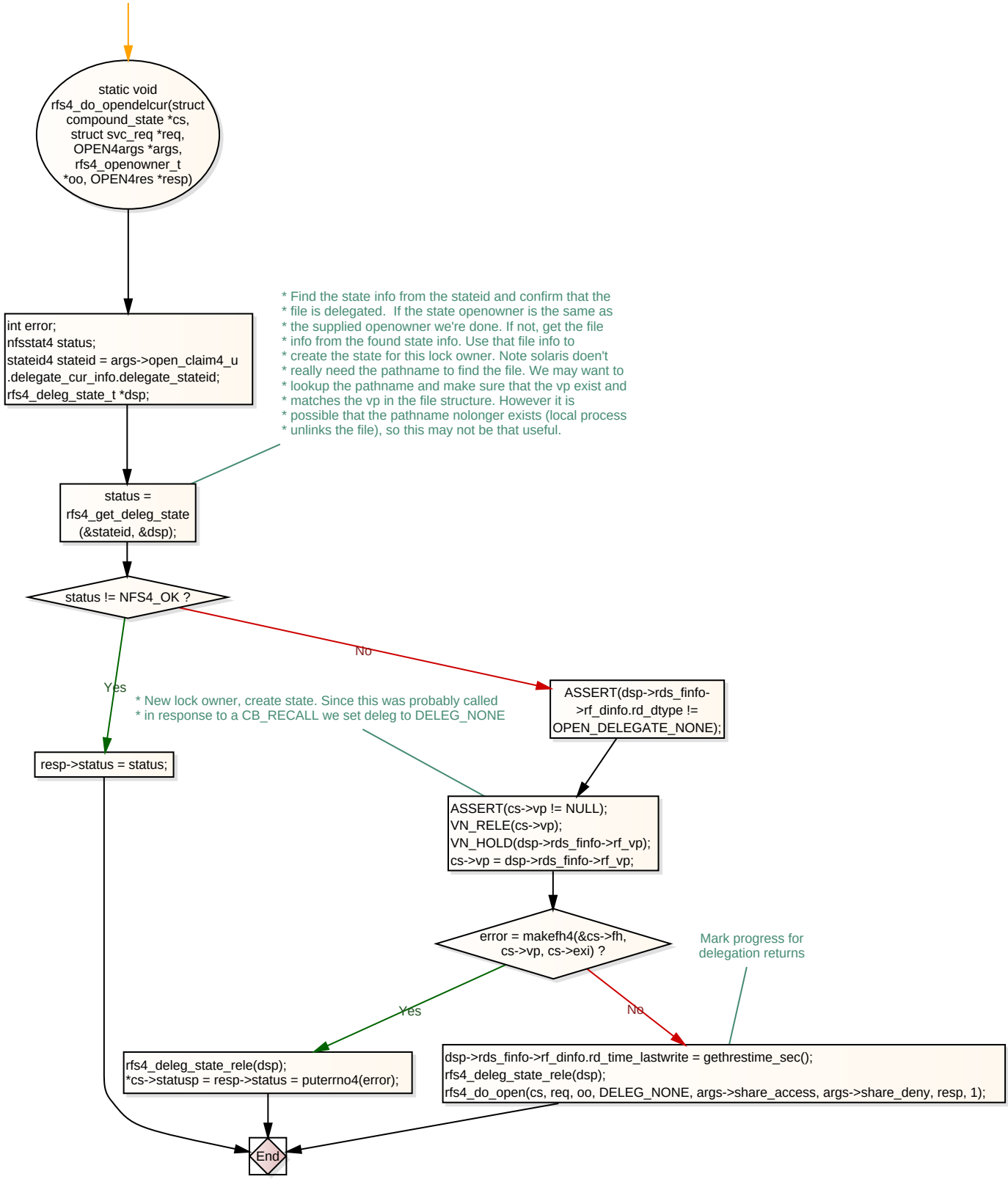

static nfsstat4 check_open_access(uint32_ t access, struct compound_state *cs, struct svc_req *req)
int error; vnode_t *vp; bool_t readonly; cred_t *cr = cs->cr;
For now we don't allow mandatory locking as per V2/V3
cs->access == CS_ACCESS_DENIED || cs->mandlock ?
No
Yes
vp = cs->vp; ASSERT(cr != NULL && vp->v_type == VREG);
* If the file system is exported read only and we are trying
* to open for write, then return NFS4ERR_ROFS
return (NFS4ERR_ACCESS);
readonly = rdonly4(cs->exi, cs->vp, req);
(access & OPEN4_SHARE_ACCESS_WRITE) && readonly ?
No
Yes
access & OPEN4_SHARE_ACCESS_READ ?
return (NFS4ERR_ROFS);
Yes
No
(VOP_ACCESS(vp, VREAD, 0, cr, NULL) != 0) && (VOP_ACCESS(vp, VEXEC, 0, cr, NULL) != 0) ?
No
Yes
access & OPEN4_SHARE_ACCESS_WRITE ?
Yes
No
return (NFS4ERR_ACCESS);
error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL);
error ?
Yes
No
return (NFS4ERR_ACCESS);
return (NFS4_OK);
End

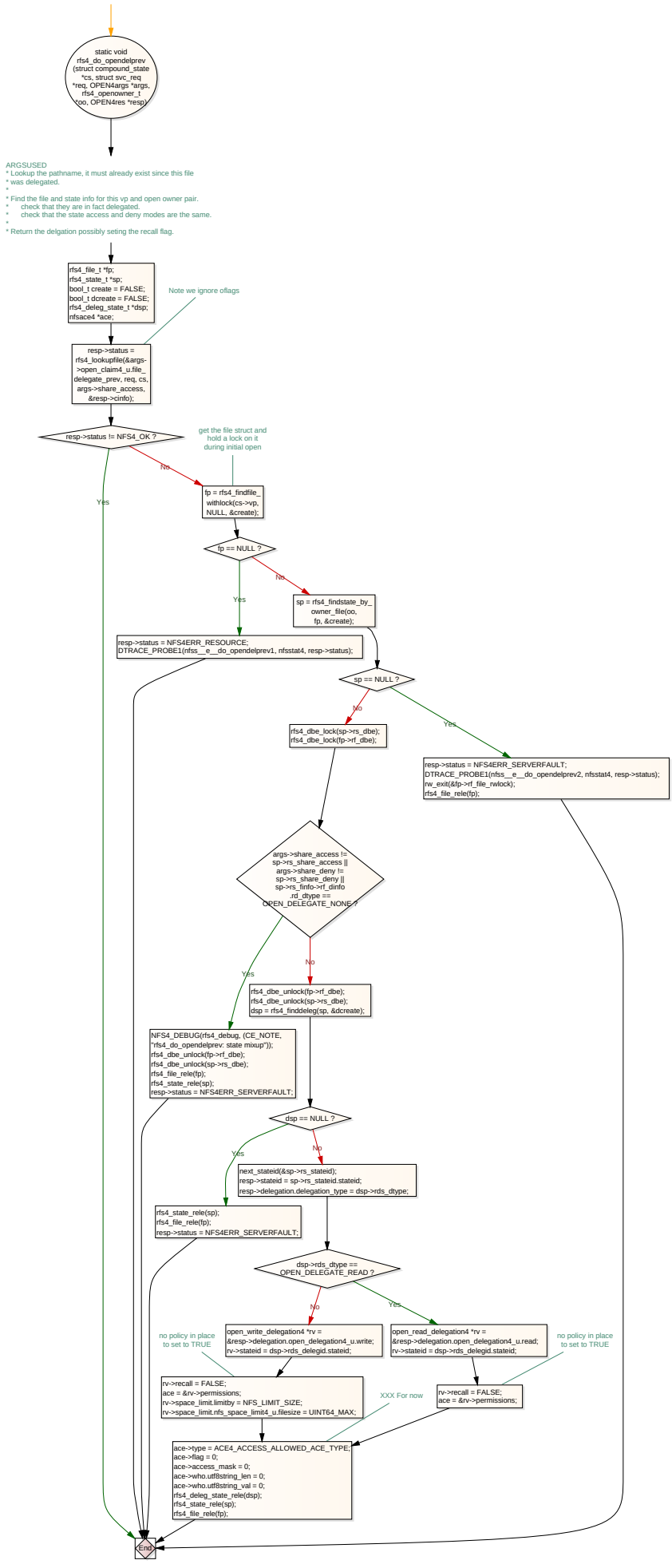


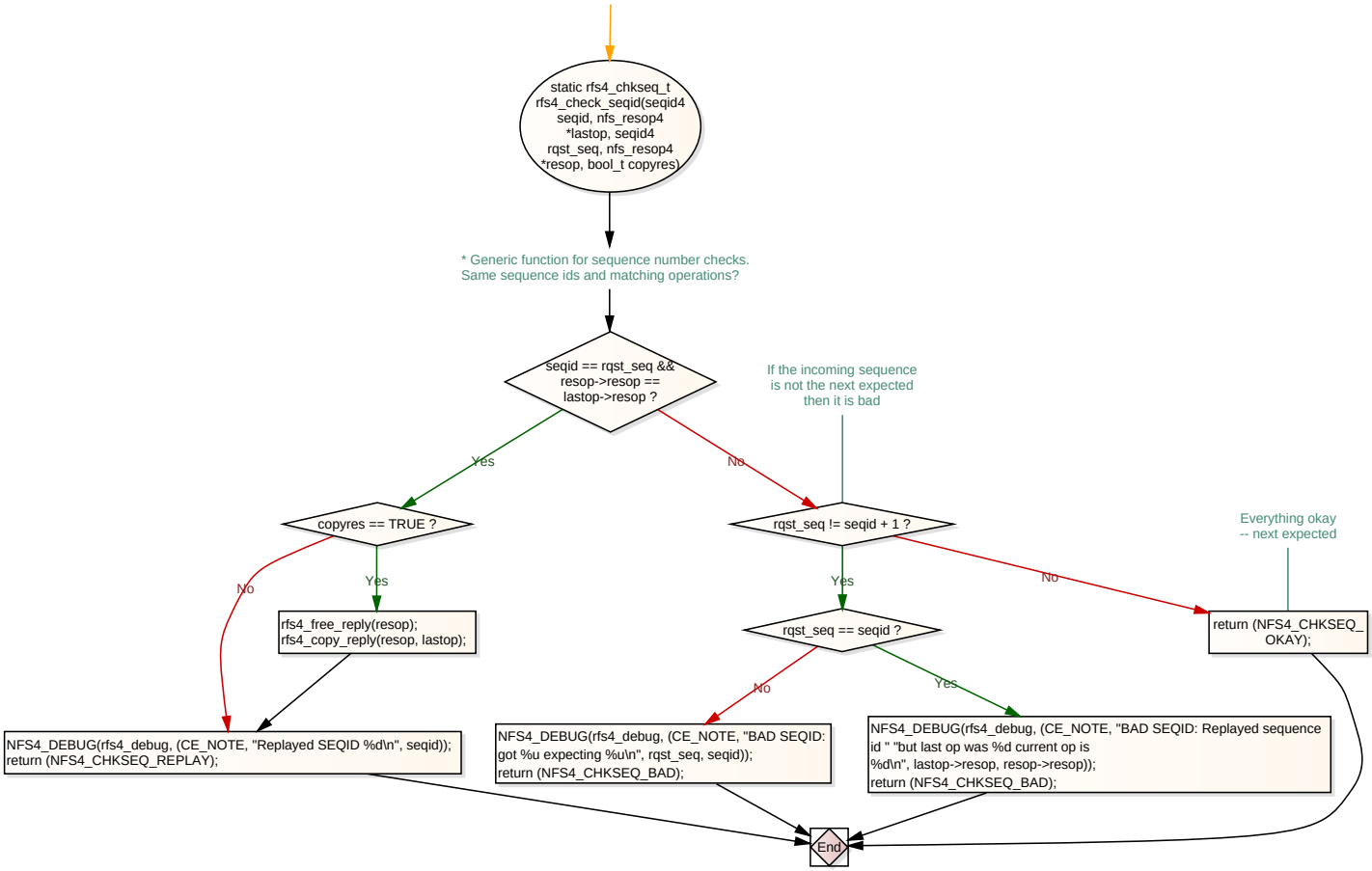


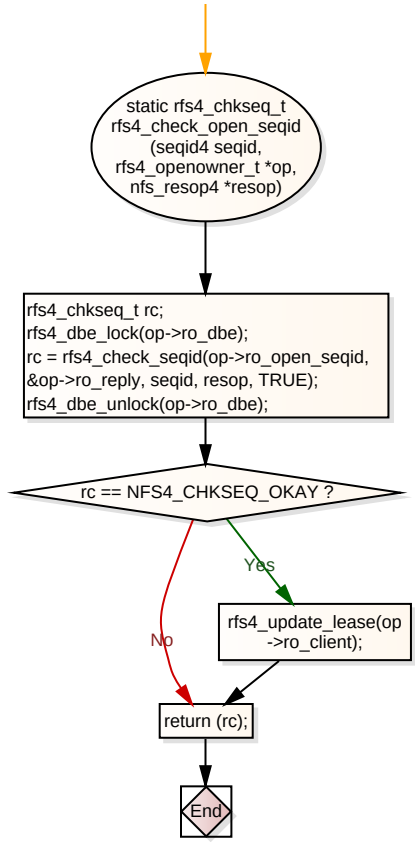

static void rfs4_do_opennull(struct compound_state *cs, struct svc_req *req, OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
ARGSUSED
change_info4 *cinfo = &resp->cinfo; bitmap4 *attrset = &resp->attrset;
args->opentype == OPEN4_NOCREATE ?
inhibit delegation grants during exclusive create
No
Yes
args->mode == EXCLUSIVE4 ?
resp->status = rfs4_lookupfile(&args->open_claim4_u.file, req, cs, args->share_access, cinfo);
Yes
rfs4_disable_delegation();
No
resp->status = rfs4_createfile(args, req, cs, cinfo, attrset, oo->ro_client ->rc_clientid);
resp->status == NFS4_OK ?
cs->vp cs->fh now reference the desired file
Yes
No
rfs4_do_open(cs, req, oo, oo->ro_need_confirm ? DELEG_NONE : DELEG_ANY, args->share_access, args->share_deny, resp, 0);
* If rfs4_createfile set attrset, we must
* clear this attrset before the response is copied.
*cs->statusp = resp->status;
resp->status != NFS4_OK && resp->attrset ?
Yes
resp->attrset = 0;
No
args->mode == EXCLUSIVE4 ?
Yes
No
rfs4_enable_delegation();
End

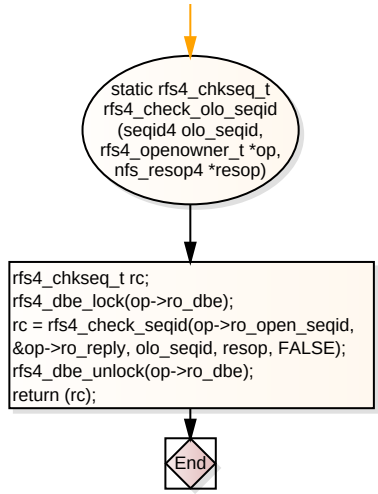

static void rfs4_do_openprev(struct compound_state *cs, struct svc_req *req, OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
ARGSUSED
change_info4 *cinfo = &resp->cinfo;
vattr_t va;
vtype_t v_type = cs->vp->v_type;
int error = 0;
Verify that we have a regular file
v_type != VREG ?
No
Yes
va.va_mask = AT_MODE|AT_UID;
error = VOP_GETATTR(cs->vp, &va, 0, cs->cr, NULL);
error ?
No
Yes
cs->mandlock = MANDLOCK(cs->vp, va.va_mode);
* Check if we have access to the file, Note the the file
* could have originally been open UNCHECKED or GUARDED
* with mode bits that will now fail, but there is nothing
* we can really do about that except in the case that the
* owner of the file is the one requesting the open.
resp->status = puterrno4(error);
crgetuid(cs->cr) != va.va_uid ?
Yes
No
resp->status = check_open_access(args->share_access, cs, req);
resp->status != NFS4_OK ?
No
Yes
v_type == VDIR ?
Yes
No
v_type == VLNK ?
Yes
No
resp->status = NFS4ERR_ISDIR;
resp->status = NFS4ERR_SYMLINK;
resp->status = NFS4ERR_INVAL;
* cinfo on a CLAIM_PREVIOUS is undefined, initialize to zero
cinfo->before = 0;
cinfo->after = 0;
cinfo->atomic = FALSE;
rfs4_do_open(cs, req, oo, NFS4_DELEG4TYPE2REQTYPE(args->open_claim4_u.delegate_type), args->share_access, args->share_deny, resp, 0);
End

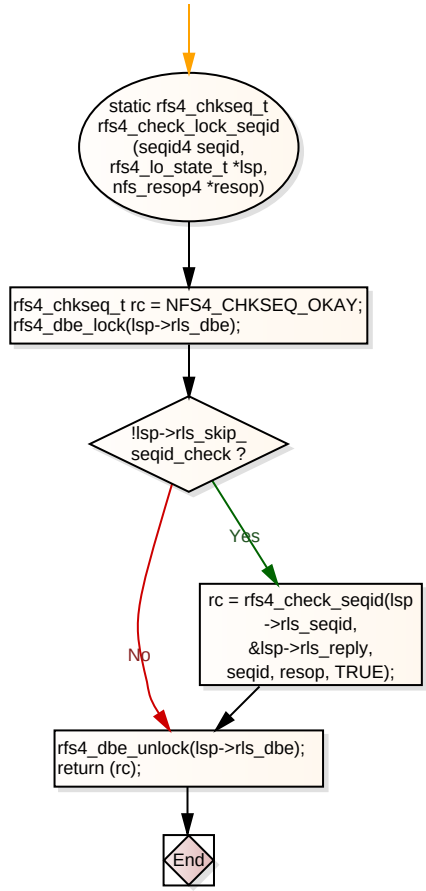

static void rfs4_do_opendelcur(struct compound_state *cs, struct svc_req *req, OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
int error;
nfsstat4 status;
stateid4 stateid = args->open_claim4_u .delegate_cur_info.delegate_stateid;
rfs4_deleg_state_t *dsp;
* Find the state info from the stateid and confirm that the
* file is delegated. If the state openowner is the same as
* the supplied openowner we're done. If not, get the file
* info from the found state info. Use that file info to
* create the state for this lock owner. Note solaris doen't
* really need the pathname to find the file. We may want to
* lookup the pathname and make sure that the vp exist and
* matches the vp in the file structure. However it is
* possible that the pathname nolonger exists (local process
* unlinks the file), so this may not be that useful.
status = rfs4_get_deleg_state (&stateid, &dsp);
status != NFS4_OK ?
No
Yes
ASSERT(dsp->rds_finfo- >rf_dinfo.rd_dtype != OPEN_DELEGATE_NONE);
* New lock owner, create state. Since this was probably called
* in response to a CB_RECALL we set deleg to DELEG_NONE
resp->status = status;
ASSERT(cs->vp != NULL);
VN_RELE(cs->vp);
VN_HOLD(dsp->rds_finfo->rf_vp);
cs->vp = dsp->rds_finfo->rf_vp;
error = makefh4(&cs->fh, cs->vp, cs->exi) ?
Mark progress for delegation returns
Yes
No
rfs4_deleg_state_rele(dsp);
*cs->statusp = resp->status = puterrno4(error);
dsp->rds_finfo->rf_dinfo.rd_time_lastwrite = gethrestime_sec();
rfs4_deleg_state_rele(dsp);
rfs4_do_open(cs, req, oo, DELEG_NONE, args->share_access, args->share_deny, resp, 1);
End

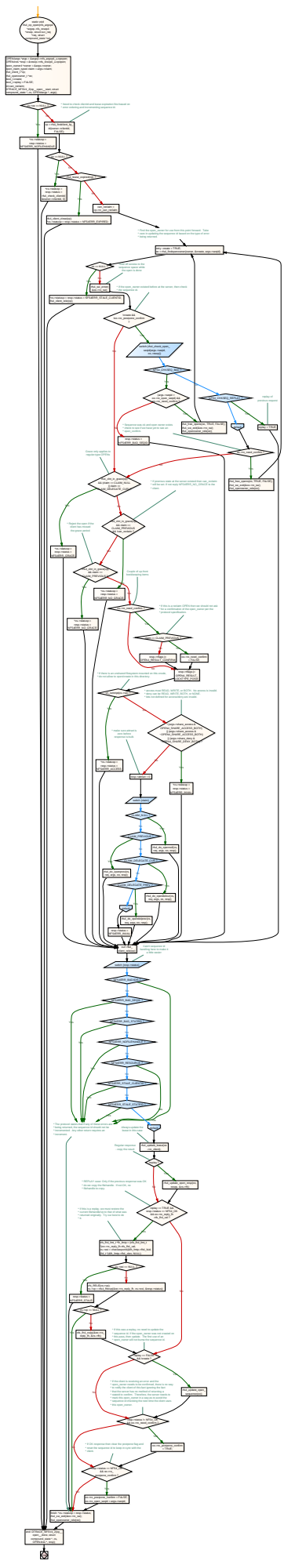

static void rfs4_do_opendelprev (struct compound_state *cs, struct svc_req *req, OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
ARGSUSED
* Lookup the pathname, it must already exist since this file
* was delegated.
*
* Find the file and state info for this vp and open owner pair.
* check that they are in fact delegated.
* check that the state access and deny modes are the same.
*
* Return the delgation possibly seting the recall flag.
rfs4_file_t *fp;
rfs4_state_t *sp;
bool_t create = FALSE;
bool_t dcreate = FALSE;
rfs4_deleg_state_t *dsp;
nfsace4 *ace;
Note we ignore oflags
resp->status = rfs4_lookupfile(&args->open_claim4_u.file_ delegate_prev, req, cs, args->share_access, &resp->cinfo);
resp->status != NFS4_OK ?
No
Yes
get the file struct and hold a lock on it during initial open
fp = rfs4_findfile_ withlock(cs->vp, NULL, &create);
fp == NULL ?
No
Yes
sp = rfs4_findstate_by_ owner_file(oo, fp, &create);
resp->status = NFS4ERR_RESOURCE;
DTRACE_PROBE1(nfss__e__do_opendelprev1, nfsstat4, resp->status);
sp == NULL ?
No
Yes
rfs4_dbe_lock(sp->rs_dbe);
rfs4_dbe_lock(fp->rf_dbe);
resp->status = NFS4ERR_SERVERFAULT;
DTRACE_PROBE1(nfss__e__do_opendelprev2, nfsstat4, resp->status);
rw_exit(&fp->rf_file_rwlock);
rfs4_file_rele(fp);
args->share_access != sp->rs_share_access || args->share_deny != sp->rs_share_deny || sp->rs_finfo->rf_dinfo .rd_dtype == OPEN_DELEGATE_NONE ?
Yes
No
rfs4_dbe_unlock(fp->rf_dbe);
rfs4_dbe_unlock(sp->rs_dbe);
dsp = rfs4_finddeleg(sp, &dcreate);
NFS4_DEBUG(rfs4_debug, (CE_NOTE, "rfs4_do_opendelprev: state mixup"));
rfs4_dbe_unlock(fp->rf_dbe);
rfs4_dbe_unlock(sp->rs_dbe);
rfs4_file_rele(fp);
rfs4_state_rele(sp);
resp->status = NFS4ERR_SERVERFAULT;
dsp == NULL ?
Yes
No
next_stateid(&sp->rs_stateid);
resp->stateid = sp->rs_stateid.stateid;
resp->delegation.delegation_type = dsp->rds_dtype;
rfs4_state_rele(sp);
rfs4_file_rele(fp);
resp->status = NFS4ERR_SERVERFAULT;
dsp->rds_dtype == OPEN_DELEGATE_READ ?
No
Yes
no policy in place to set to TRUE
open_write_delegation4 *rv = &resp->delegation.open_delegation4_u.write;
rv->stateid = dsp->rds_delegid.stateid;
open_read_delegation4 *rv = &resp->delegation.open_delegation4_u.read;
rv->stateid = dsp->rds_delegid.stateid;
no policy in place to set to TRUE
rv->recall = FALSE;
ace = &rv->permissions;
rv->space_limit.limitby = NFS_LIMIT_SIZE;
rv->space_limit.nfs_space_limit4_u.filesize = UINT64_MAX;
XXX For now
rv->recall = FALSE;
ace = &rv->permissions;
ace->type = ACE4_ACCESS_ALLOWED_ACE_TYPE;
ace->flag = 0;
ace->access_mask = 0;
ace->who.utf8string_len = 0;
ace->who.utf8string_val = 0;
rfs4_deleg_state_rele(dsp);
rfs4_state_rele(sp);
rfs4_file_rele(fp);
End

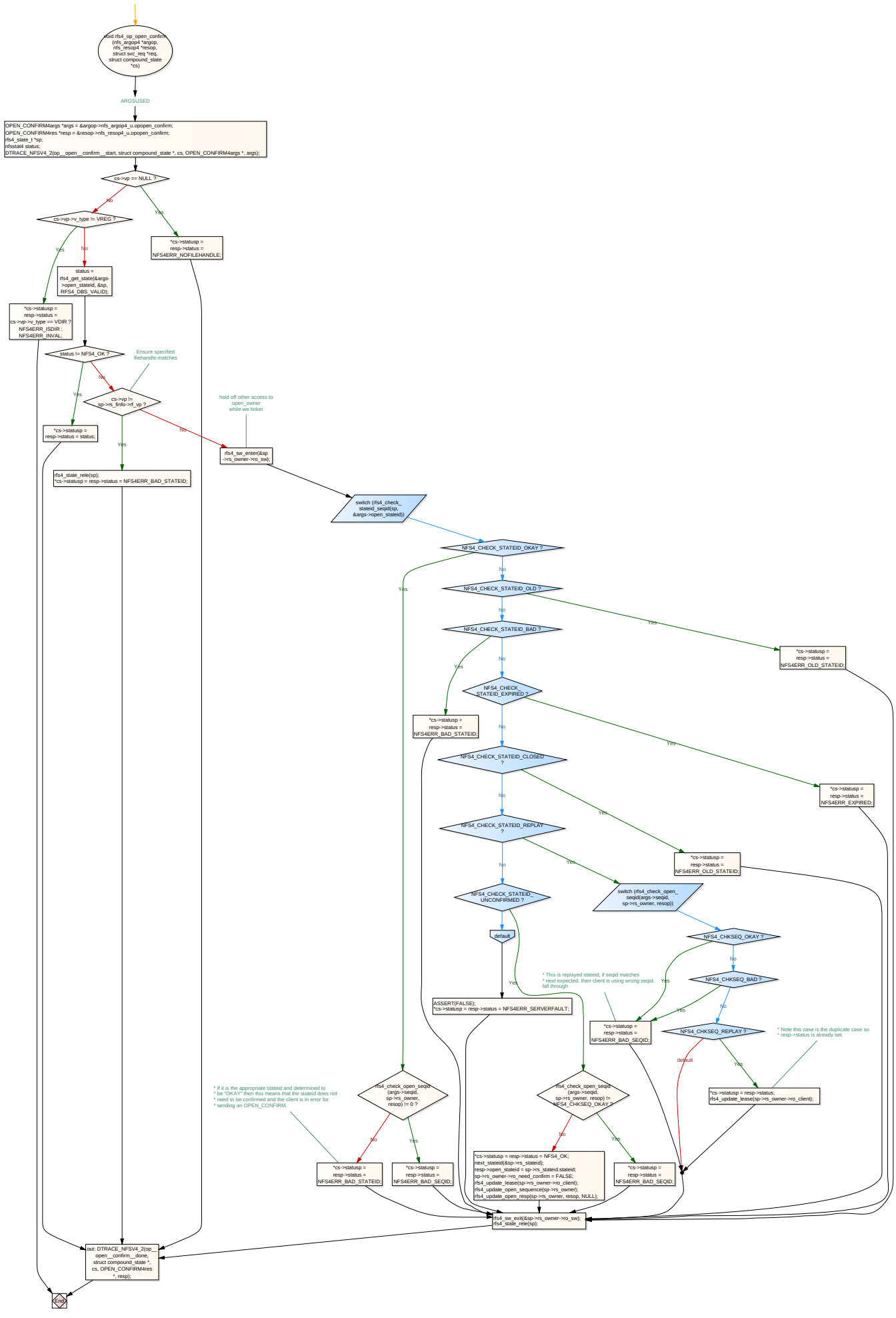

static rfs4_chkseq_t
rfs4_check_seqid(seqid4
seqid, nfs_resop4
*lastop, seqid4
rqst_seq, nfs_resop4
*resop, bool_t copyres)
* Generic function for sequence number checks.
Same sequence ids and matching operations?
seqid == rqst_seq &&
resop->resop ==
lastop->resop ?
Yes
No
If the incoming sequence
is not the next expected
then it is bad
copyres == TRUE ?
rqst_seq != seqid + 1 ?
Everything okay
-- next expected
No
Yes
Yes
No
rfs4_free_reply(resop);
rfs4_copy_reply(resop, lastop);
rqst_seq == seqid ?
return (NFS4_CHKSEQ_
OKAY);
No
Yes
NFS4_DEBUG(rfs4_debug, (CE_NOTE, "Replayed SEQID %d\n", seqid));
return (NFS4_CHKSEQ_REPLAY);
NFS4_DEBUG(rfs4_debug, (CE_NOTE, "BAD SEQID:
got %u expecting %u\n", rqst_seq, seqid));
return (NFS4_CHKSEQ_BAD);
NFS4_DEBUG(rfs4_debug, (CE_NOTE, "BAD SEQID: Replayed sequence
id " "but last op was %d current op is
%d\n", lastop->resop, resop->resop));
return (NFS4_CHKSEQ_BAD);
End

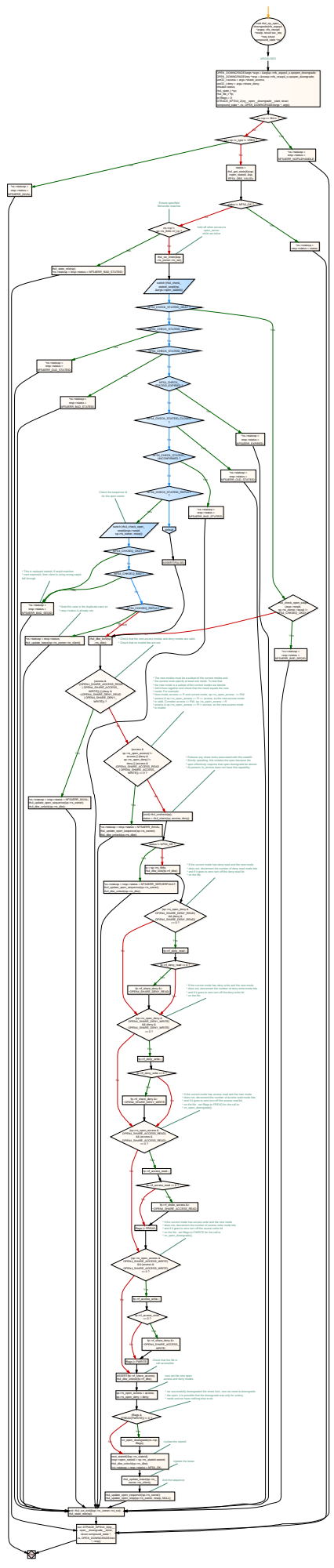

static rfs4_chkseq_t
rfs4_check_open_seqid
(seqid4 seqid,
rfs4_openowner_t *op,
nfs_resop4 *resop)
rfs4_chkseq_t rc;
rfs4_dbe_lock(op->ro_dbe);
rc = rfs4_check_seqid(op->ro_open_seqid,
&op->ro_reply, seqid, resop, TRUE);
rfs4_dbe_unlock(op->ro_dbe);
rc == NFS4_CHKSEQ_OKAY ?
Yes
No
rfs4_update_lease(op
->ro_client);
return (rc);
End

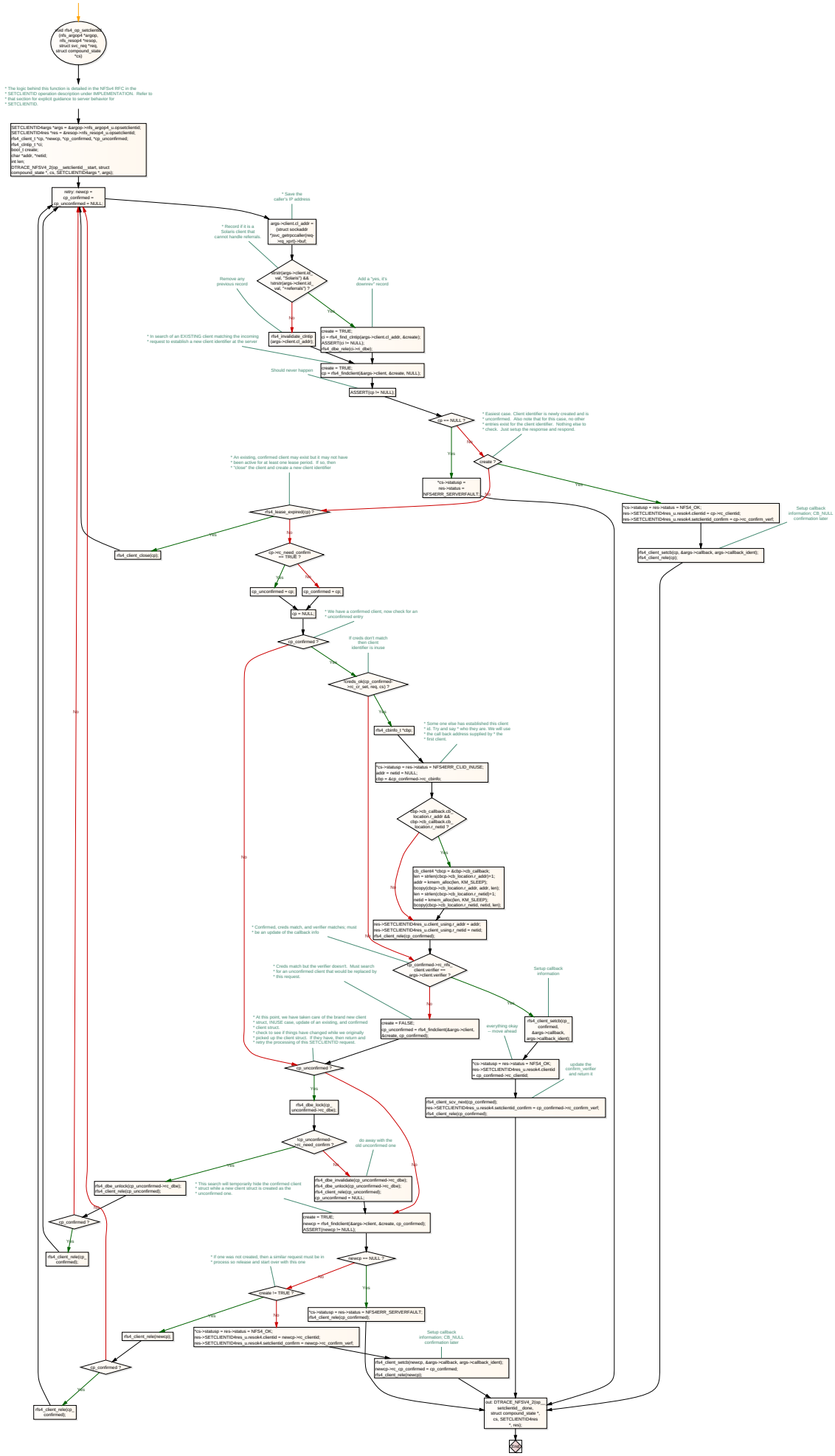

static rfs4_chkseq_t
rfs4_check_olo_seqid
(seqid4 olo_seqid,
rfs4_openowner_t *op,
nfs_resop4 *resop)
rfs4_chkseq_t rc;
rfs4_dbe_lock(op->ro_dbe);
rc = rfs4_check_seqid(op->ro_open_seqid,
&op->ro_reply, olo_seqid, resop, FALSE);
rfs4_dbe_unlock(op->ro_dbe);
return (rc);
End

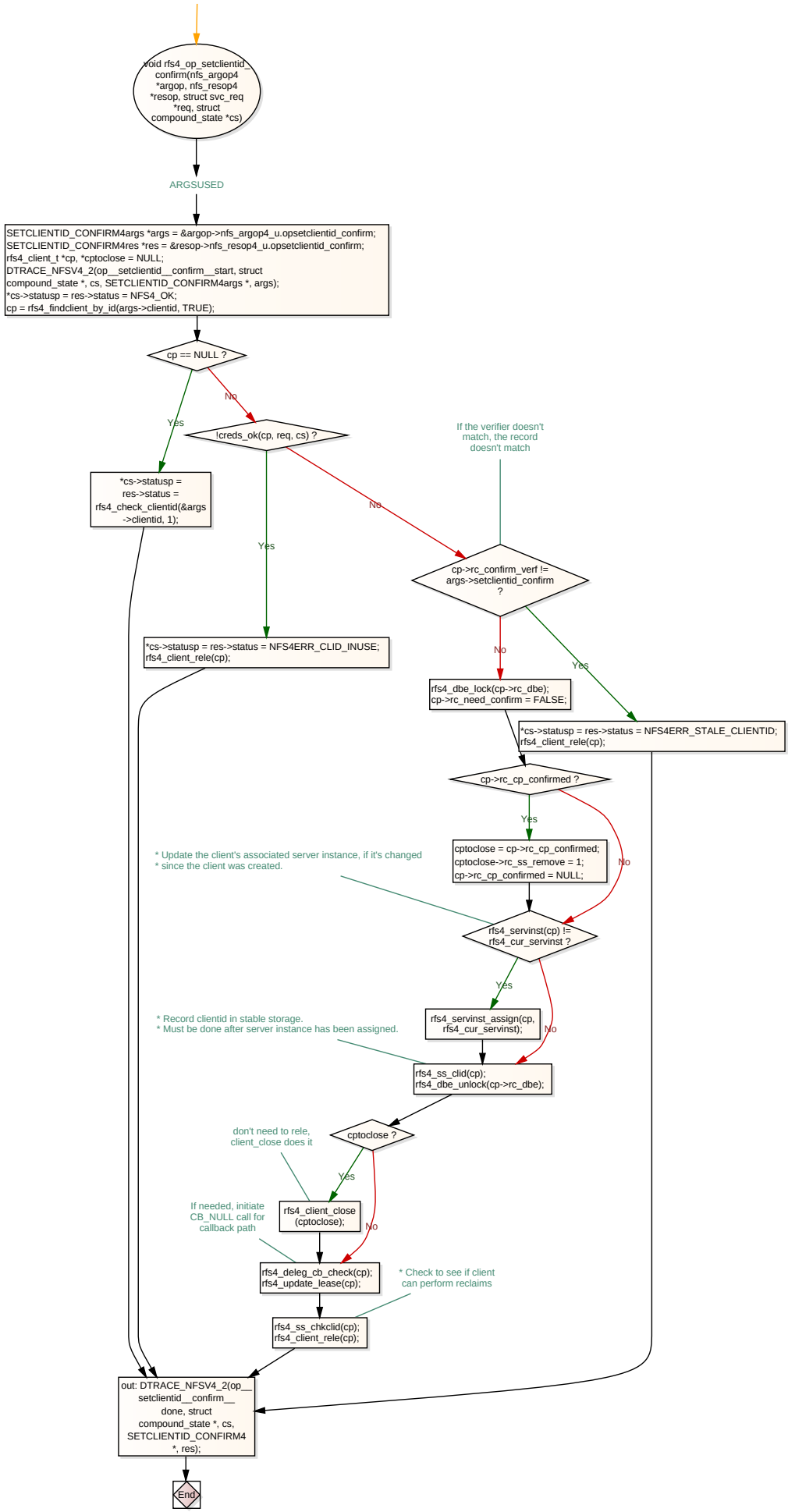

static rfs4_chkseq_t
rfs4_check_lock_seqid
(seqid4 seqid,
rfs4_lo_state_t *lsp,
nfs_resop4 *resop)
rfs4_chkseq_t rc = NFS4_CHKSEQ_OKAY;
rfs4_dbe_lock(lsp->rls_dbe);
!lsp->rls_skip_
seqid_check ?
Yes
No
rc = rfs4_check_seqid(lsp
->rls_seqid,
&lsp->rls_reply,
seqid, resop, TRUE);
rfs4_dbe_unlock(lsp->rls_dbe);
return (rc);
End

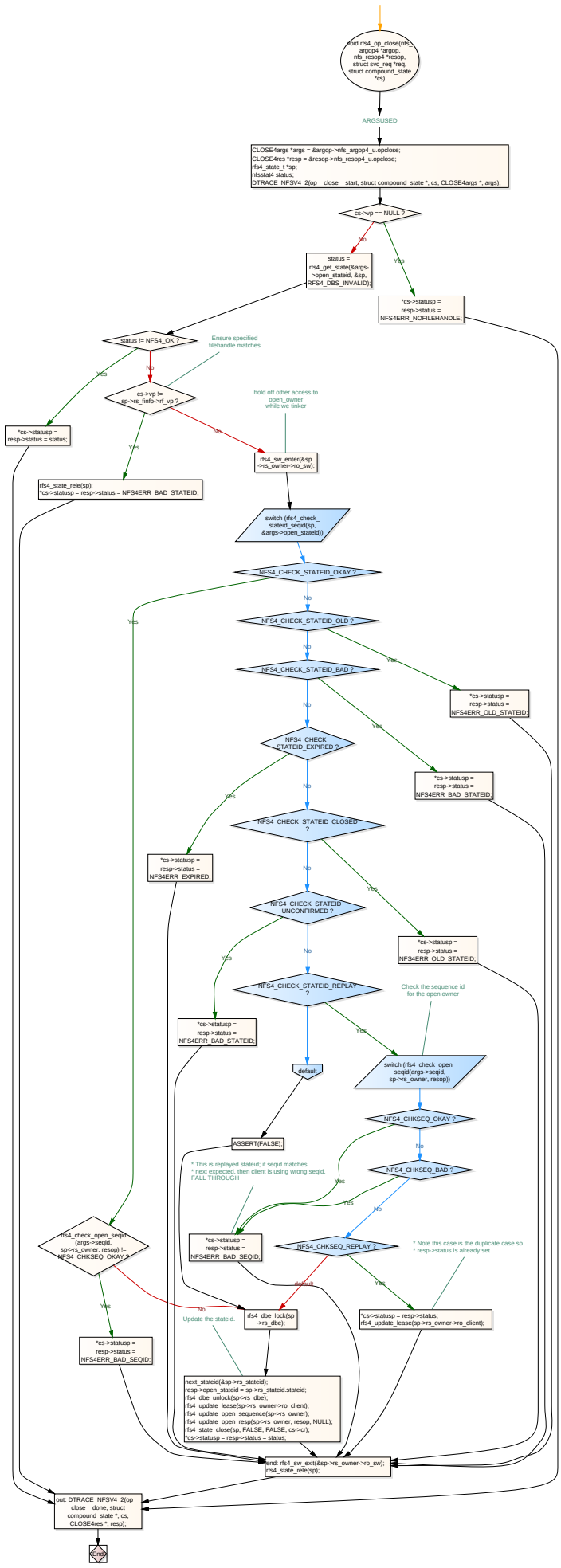


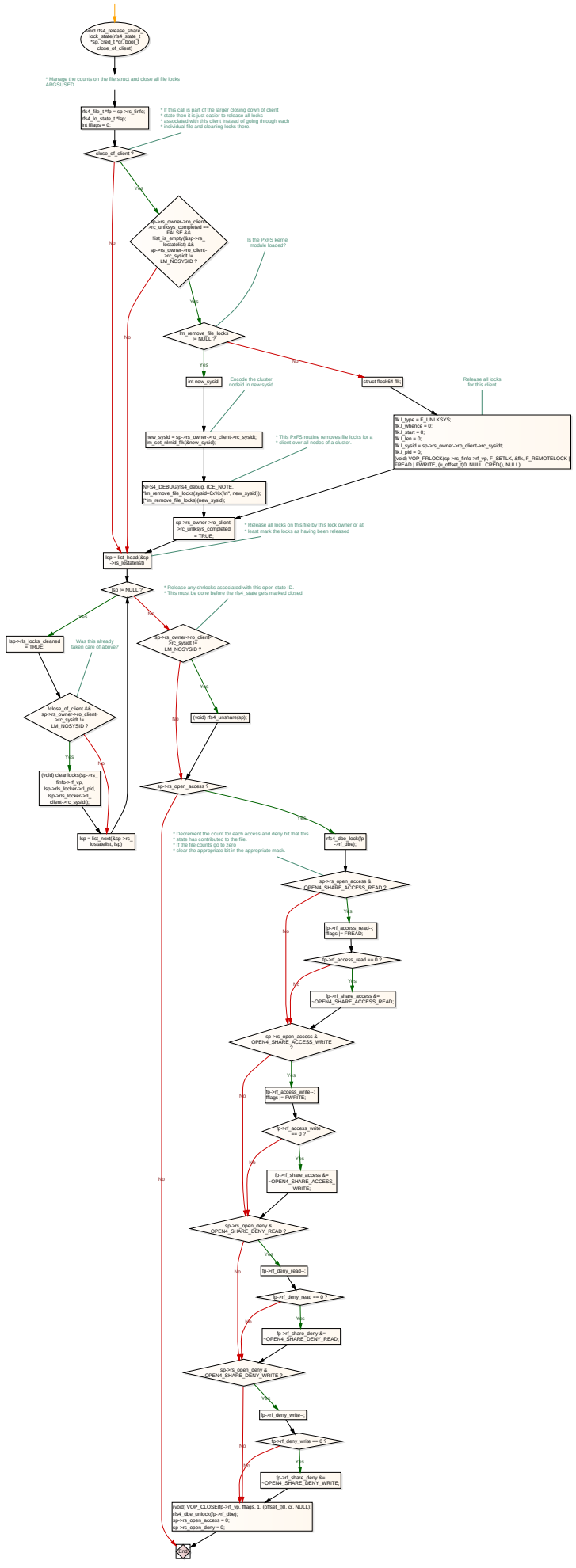

void rfs4_op_open_confirm (nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
OPEN_CONFIRM4args *args = &argop->nfs_argop4_u.opopen_confirm;
OPEN_CONFIRM4res *resp = &resop->nfs_resop4_u.opopen_confirm;
rfs4_state_t *sp;
nfsstat4 status;
DTRACE_NFSV4_2(op__open__confirm__start, struct compound_state *, cs, OPEN_CONFIRM4args *, args);
cs->vp == NULL ?
No
Yes
cs->vp->v_type != VREG ?
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
Yes
No
status = rfs4_get_state(&args->open_stateid, &sp, RFS4_DBS_VALID);
*cs->statusp = resp->status = cs->vp->v_type == VDIR ? NFS4ERR_ISDIR : NFS4ERR_INVAL;
status != NFS4_OK ?
Ensure specified filehandle matches
No
Yes
cs->vp != sp->rs_finfo->rf_vp ?
No
Yes
*cs->statusp = resp->status = status;
rfs4_state_rele(sp);
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
hold off other access to open_owner while we tinker
rfs4_sw_enter(&sp->rs_owner->ro_sw);
switch (rfs4_check_stateid_seqid(sp, &args->open_stateid))
NFS4_CHECK_STATEID_OKAY ?
Yes
No
NFS4_CHECK_STATEID_OLD ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_OLD_STATEID;
NFS4_CHECK_STATEID_BAD ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
NFS4_CHECK_STATEID_EXPIRED ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_EXPIRED;
NFS4_CHECK_STATEID_CLOSED ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_OLD_STATEID;
NFS4_CHECK_STATEID_REPLAY ?
No
Yes
switch (rfs4_check_open_seqid(args->seqid, sp->rs_owner, resop))
NFS4_CHECK_STATEID_UNCONFIRMED ?
default
Yes
ASSERT(FALSE);
*cs->statusp = resp->status = NFS4ERR_SERVERFAULT;
NFS4_CHKSEQ_OKAY ?
No
NFS4_CHKSEQ_BAD ?
Yes
No
* This is replayed stateid; if seqid matches
* next expected, then client is using wrong seqid.
fall through
Yes
*cs->statusp = resp->status = NFS4ERR_BAD_SEQID;
NFS4_CHKSEQ_REPLAY ?
default
Yes
* Note this case is the duplicate case so
* resp->status is already set.
*cs->statusp = resp->status; rfs4_update_lease(sp->rs_owner->ro_client);
* If it is the appropriate stateid and determined to
* be "OKAY" then this means that the stateid does not
* need to be confirmed and the client is in error for
* sending an OPEN_CONFIRM.
rfs4_check_open_seqid (args->seqid, sp->rs_owner, resop) != 0 ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
*cs->statusp = resp->status = NFS4ERR_BAD_SEQID;
rfs4_check_open_seqid (args->seqid, sp->rs_owner, resop) != NFS4_CHKSEQ_OKAY ?
No
Yes
*cs->statusp = resp->status = NFS4_OK; next_stateid(&sp->rs_stateid); resp->open_stateid = sp->rs_stateid.stateid; sp->rs_owner->ro_need_confirm = FALSE; rfs4_update_lease(sp->rs_owner->ro_client); rfs4_update_open_sequence(sp->rs_owner); rfs4_update_open_resp(sp->rs_owner, resop, NULL);
*cs->statusp = resp->status = NFS4ERR_BAD_SEQID;
rfs4_sw_exit(&sp->rs_owner->ro_sw); rfs4_state_rele(sp);
out: DTRACE_NFSV4_2(op__open__confirm__done, struct compound_state *, cs, OPEN_CONFIRM4res *, resp);
Emd

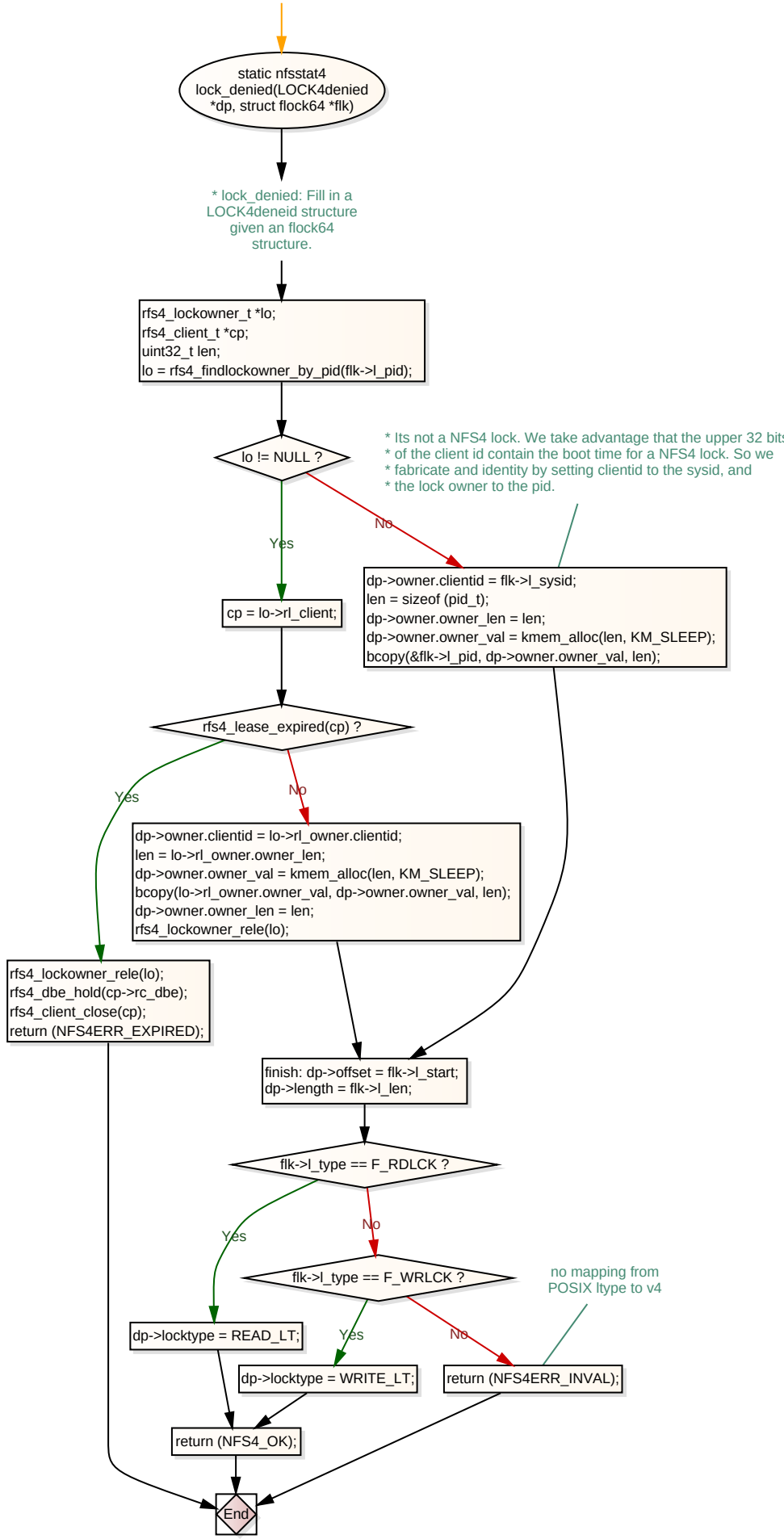


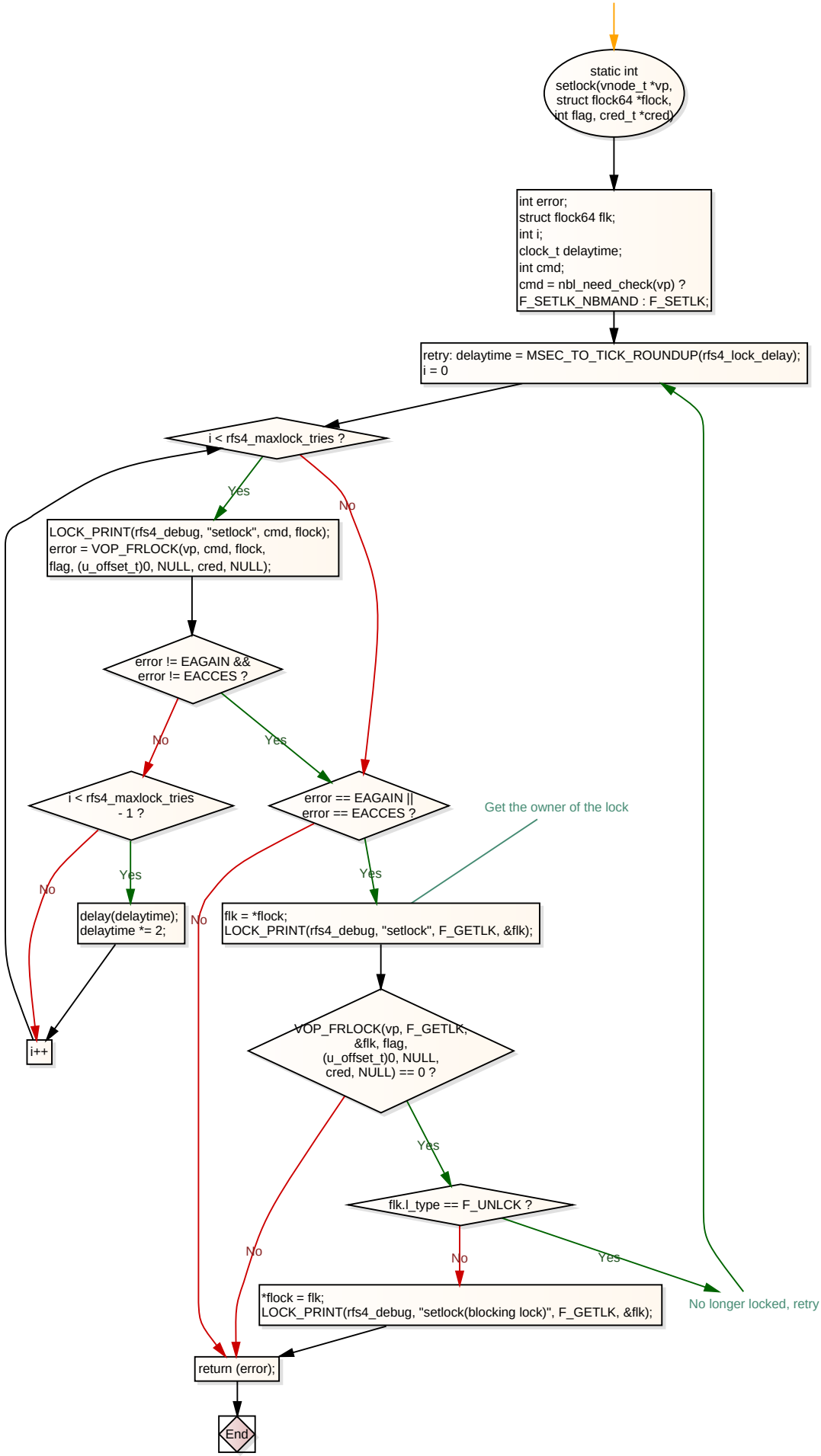

nfsd4 nfs4_op_setclientid (nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* The logic behind this function is detailed in the NFSv4 RFC in the * SETCLIENTID operation description under IMPLEMENTATION. Refer to * that section for explicit guidance to server behavior for * SETCLIENTID.
SETCLIENTID4args *args = &argop->nfs_argop4_u.opsetclientid; SETCLIENTID4res *res = &resop->nfs_resop4_u.opsetclientid; nfs4_client_t *cp, *newcp, *cp_confirmed, *cp_unconfirmed; nfs4_clntip_t *ci; bool_t create; char *addr, *netid; int len; DTRACE_NFSV4_2(cp__setclientid__start, struct compound_state *, cs, SETCLIENTID4args *, args);
retry: newcp = cp_confirmed = cp_unconfirmed = NULL;
* Save the caller's IP address
args->client.cl_addr = (struct sockaddr *)svc_getrpccaller(req)->xp_rtaddr.buf;
* Record if it is a Solaris client that cannot handle referrals.
strstr(args->client.id.val, "Solaris") && !strstr(args->client.id.val, "+referrals") ?
Remove any previous record
Add a "yes, it's downrev" record
No
Yes
nfs4_invalidate_clntip (args->client.cl_addr)
create = TRUE; ci = nfs4_find_clntip(args->client.cl_addr, &create); ASSERT(ci != NULL); nfs4_dbe_rele(ci->ri_dbe);
* In search of an EXISTING client matching the incoming * request to establish a new client identifier at the server
Should never happen
create = TRUE; cp = nfs4_findclient(&args->client, &create, NULL);
ASSERT(cp != NULL);
cp == NULL ?
* Easiest case. Client identifier is newly created and is * unconfirmed. Also note that for this case, no other * entries exist for the client identifier. Nothing else to * check. Just setup the response and respond.
create ?
* An existing, confirmed client may exist but it may not have * been active for at least one lease period. If so, then * "close" the client and create a new client identifier
*cs->statusp = res->status = NFS4ERR_SERVERFAULT;
*cs->statusp = res->status = NFS4_OK; res->SETCLIENTID4res_u.resok4.clientid = cp->rc_clientid; res->SETCLIENTID4res_u.resok4.setclientid_confirm = cp->rc_confirm_verf;
Setup callback information; CB_NULL confirmation later
nfs4_client_setcb(cp, &args->callback, args->callback_ident); nfs4_client_rele(cp);
nfs4_lease_expired(cp) ?
Yes
No
nfs4_client_close(cp);
cp->rc_need_confirm == TRUE ?
Yes
No
cp_unconfirmed = cp;
cp_confirmed = cp;
cp = NULL;
* We have a confirmed client, now check for an * unconfirmed entry
cp_confirmed ?
If creds don't match then client identifier is inuse
Yes
No
*creds_ok(cp_confirmed->rc_cr_set, req, cs) ?
Yes
nfs4_clntip_t *clnp;
* Some one else has established this client * id. Try and say * who they are. We will use * the call back address supplied by * the * first client.
*cs->statusp = res->status = NFS4ERR_CLID_INUSE; addr = netid = NULL; clnp = &cp_confirmed->rc_clntip;
clnp->cb_callback.cb_location.r_addr && clnp->cb_callback.cb_location.r_netid ?
Yes
No
cb_client4 *cbcp = &clnp->cb_callback; len = strlen(cbcp->cb_location.r_addr)+1; addr = kmem_alloc(len, KM_SLEEP); bcopy(cbcp->cb_location.r_addr, addr, len); len = strlen(cbcp->cb_location.r_netid)+1; netid = kmem_alloc(len, KM_SLEEP); bcopy(cbcp->cb_location.r_netid, netid, len);
res->SETCLIENTID4res_u.client_using.r_addr = addr; res->SETCLIENTID4res_u.client_using.r_netid = netid; nfs4_client_rele(cp_confirmed);
* Confirmed, creds match, and verifier matches; must * be an update of the callback info
cp_confirmed->rc_nfs_client.verifier == args->client.verifier ?
* Creds match but the verifier doesn't. Must search * for an unconfirmed client that would be replaced by * this request.
Setup callback information
Yes
No
* At this point, we have taken care of the brand new client * struct, INUSE case, update of an existing, and confirmed * client struct. * check to see if things have changed while we originally * picked up the client struct. If they have, then return and * retry the processing of this SETCLIENTID request.
create = FALSE; cp_unconfirmed = nfs4_findclient(&args->client, &create, cp_confirmed);
everything okay -- move ahead
nfs4_client_setcb(cp_confirmed, &args->callback, args->callback_ident);
*cs->statusp = res->status = NFS4_OK; res->SETCLIENTID4res_u.resok4.clientid = cp_confirmed->rc_clientid;
update the confirm_verifier and return it
cp_unconfirmed ?
Yes
nfs4_dbe_lock(cp_unconfirmed->rc_dbe);
nfs4_client_scv_next(cp_confirmed); res->SETCLIENTID4res_u.resok4.setclientid_confirm = cp_confirmed->rc_confirm_verf; nfs4_client_rele(cp_confirmed);
cp_unconfirmed->rc_need_confirm ?
do away with the old unconfirmed one
Yes
No
nfs4_dbe_unlock(cp_unconfirmed->rc_dbe); nfs4_client_rele(cp_unconfirmed);
* This search will temporarily hide the confirmed client * struct while a new client struct is created as the * unconfirmed one.
nfs4_dbe_invalidate(cp_unconfirmed->rc_dbe); nfs4_dbe_unlock(cp_unconfirmed->rc_dbe); nfs4_client_rele(cp_unconfirmed); cp_unconfirmed = NULL;
cp_confirmed ?
Yes
No
create = TRUE; newcp = nfs4_findclient(&args->client, &create, cp_confirmed); ASSERT(newcp != NULL);
nfs4_client_rele(cp_confirmed);
* If one was not created, then a similar request must be in * process so release and start over with this one
newcp == NULL ?
No
Yes
Create != TRUE ?
Yes
No
*cs->statusp = res->status = NFS4ERR_SERVERFAULT; nfs4_client_rele(cp_confirmed);
nfs4_client_rele(newcp);
*cs->statusp = res->status = NFS4_OK; res->SETCLIENTID4res_u.resok4.clientid = newcp->rc_clientid; res->SETCLIENTID4res_u.resok4.setclientid_confirm = newcp->rc_confirm_verf;
Setup callback information; CB_NULL confirmation later
nfs4_client_setcb(newcp, &args->callback, args->callback_ident); newcp->rc_cp_confirmed = cp_confirmed; nfs4_client_rele(newcp);
cp_confirmed ?
Yes
nfs4_client_rele(cp_confirmed);
out: DTRACE_NFSV4_2(cp__setclientid__done, struct compound_state *, cs, SETCLIENTID4res *, res);
end

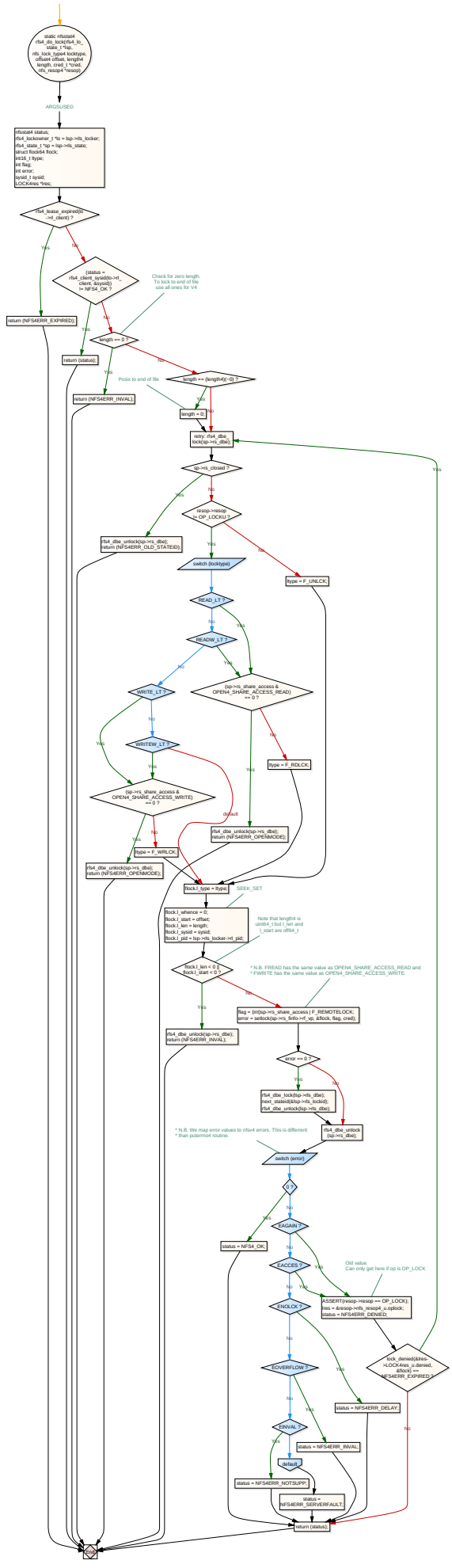

void rfs4_op_setclientid_confirm(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
SETCLIENTID_CONFIRM4args *args = &argop->nfs_argop4_u.opsetclientid_confirm;
SETCLIENTID_CONFIRM4res *res = &resop->nfs_resop4_u.opsetclientid_confirm;
rfs4_client_t *cp, *cptoclose = NULL;
DTRACE_NFSV4_2(op__setclientid__confirm__start, struct compound_state *, cs, SETCLIENTID_CONFIRM4args *, args);
*cs->statusp = res->status = NFS4_OK;
cp = rfs4_findclient_by_id(args->clientid, TRUE);
cp == NULL ?
Yes
No
*cs->statusp = res->status = rfs4_check_clientid(&args->clientid, 1);
!creds_ok(cp, req, cs) ?
Yes
No
*cs->statusp = res->status = NFS4ERR_CLID_INUSE;
rfs4_client_rele(cp);
If the verifier doesn't match, the record doesn't match
cp->rc_confirm_verf != args->setclientid_confirm ?
No
Yes
rfs4_dbe_lock(cp->rc_dbe);
cp->rc_need_confirm = FALSE;
*cs->statusp = res->status = NFS4ERR_STALE_CLIENTID;
rfs4_client_rele(cp);
cp->rc_cp_confirmed ?
Yes
No
cptoclose = cp->rc_cp_confirmed;
cptoclose->rc_ss_remove = 1;
cp->rc_cp_confirmed = NULL;
* Update the client's associated server instance, if it's changed
* since the client was created.
rfs4_servinst(cp) != rfs4_cur_servinst ?
Yes
No
rfs4_servinst_assign(cp, rfs4_cur_servinst);
* Record clientid in stable storage.
* Must be done after server instance has been assigned.
rfs4_ss_clid(cp);
rfs4_dbe_unlock(cp->rc_dbe);
cptoclose ?
don't need to rele, client_close does it
Yes
No
rfs4_client_close (cptoclose);
If needed, initiate CB_NULL call for callback path
rfs4_deleg_cb_check(cp);
rfs4_update_lease(cp);
* Check to see if client can perform reclaims
rfs4_ss_chkclid(cp);
rfs4_client_rele(cp);
out: DTRACE_NFSV4_2(op__setclientid__confirm__done, struct compound_state *, cs, SETCLIENTID_CONFIRM4 *, res);
End

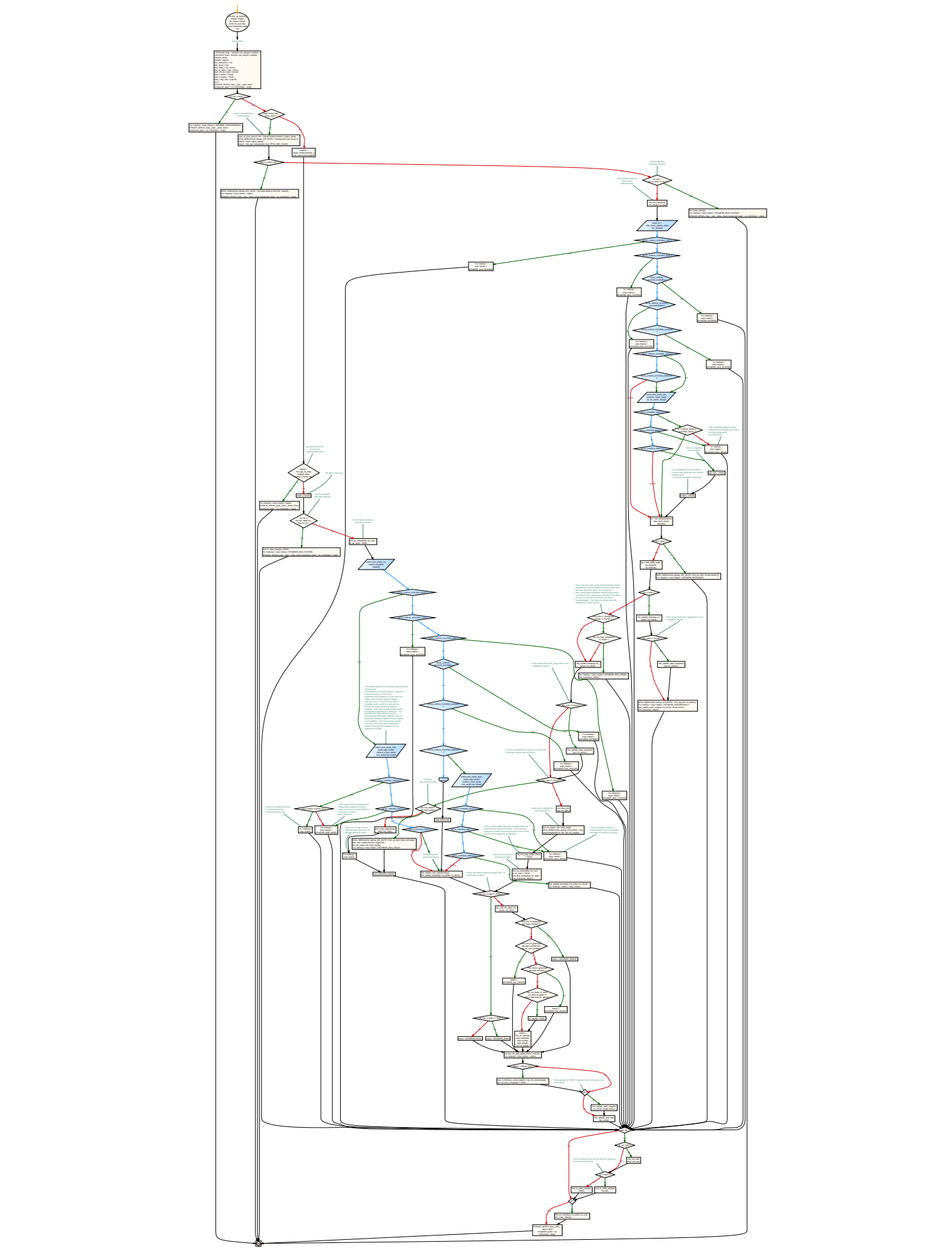

void rfs4_op_close(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
ARGSUSED
CLOSE4args *args = &argop->nfs_argop4_u.opclose;
CLOSE4res *resp = &resop->nfs_resop4_u.opclose;
rfs4_state_t *sp;
nfsstat4 status;
DTRACE_NFSV4_2(op__close__start, struct compound_state *, cs, CLOSE4args *, args);
cs->vp == NULL ?
No
Yes
status = rfs4_get_state(&args->open_stateid, &sp, RFS4_DBS_INVALID);
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
status != NFS4_OK ?
Yes
No
Ensure specified filehandle matches
cs->vp != sp->rs_finfo->rf_vp ?
No
Yes
hold off other access to open_owner while we tinker
*cs->statusp = resp->status = status;
rfs4_sw_enter(&sp->rs_owner->ro_sw);
rfs4_state_rele(sp);
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
switch (rfs4_check_stateid_seqid(sp, &args->open_stateid))
NFS4_CHECK_STATEID_OKAY ?
No
Yes
NFS4_CHECK_STATEID_OLD ?
No
Yes
NFS4_CHECK_STATEID_BAD ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_OLD_STATEID;
NFS4_CHECK_STATEID_EXPIRED ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
NFS4_CHECK_STATEID_CLOSED ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_EXPIRED;
NFS4_CHECK_STATEID_UNCONFIRMED ?
No
Yes
*cs->statusp = resp->status = NFS4ERR_OLD_STATEID;
NFS4_CHECK_STATEID_REPLAY ?
Yes
Check the sequence id for the open owner
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
default
switch (rfs4_check_open_seqid(args->seqid, sp->rs_owner, resop))
NFS4_CHKSEQ_OKAY ?
No
ASSERT(FALSE);
NFS4_CHKSEQ_BAD ?
Yes
No
* This is replayed stateid, if seqid matches
* next expected, then client is using wrong seqid.
FALL THROUGH
rfs4_check_open_seqid(args->seqid, sp->rs_owner, resop) != NFS4_CHKSEQ_OKAY ?
Yes
No
*cs->statusp = resp->status = NFS4ERR_BAD_SEQID;
NFS4_CHKSEQ_REPLAY ?
default
Yes
* Note this case is the duplicate case so
* resp->status is already set.
*cs->statusp = resp->status; rfs4_update_lease(sp->rs_owner->ro_client);
rfs4_dbe_lock(sp->rs_dbe);
Update the stateid.
*cs->statusp = resp->status = NFS4ERR_BAD_SEQID;
next_stateid(&sp->rs_stateid);
resp->open_stateid = sp->rs_stateid.stateid;
rfs4_dbe_unlock(sp->rs_dbe);
rfs4_update_lease(sp->rs_owner->ro_client);
rfs4_update_open_sequence(sp->rs_owner);
rfs4_update_open_resp(sp->rs_owner, resop, NULL);
rfs4_state_close(sp, FALSE, FALSE, cs->cr);
*cs->statusp = resp->status = status;
end: rfs4_sw_exit(&sp->rs_owner->ro_sw);
rfs4_state_rele(sp);
out: DTRACE_NFSV4_2(op__close__done, struct compound_state *, cs, CLOSE4res *, resp);
End

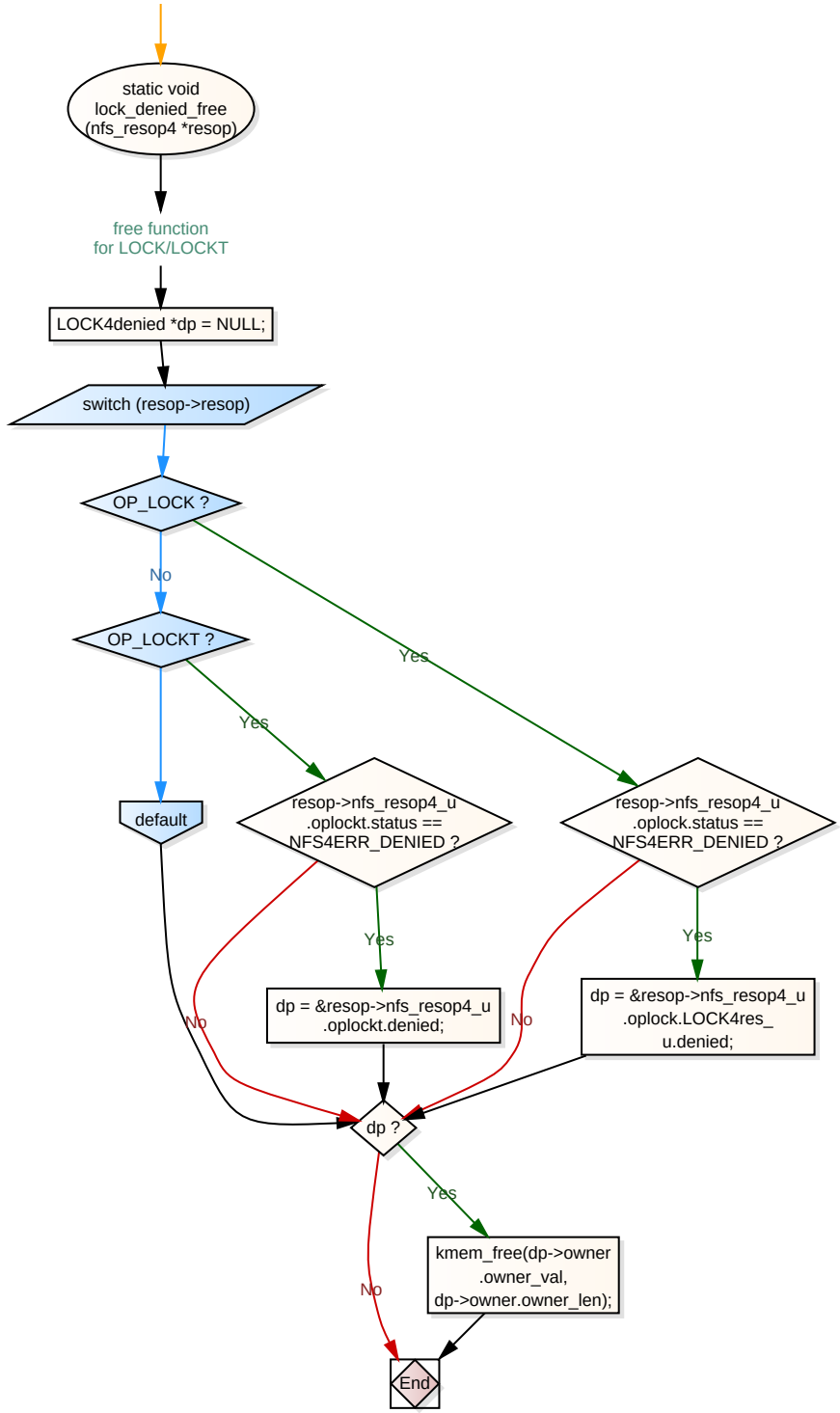

void rfs4_release_share_lock_state(rfs4_state_t *sp, cred_t *cr, bool_t close_of_client)
* Manage the counts on the file struct and close all file locks
ARGSUSED
rfs4_file_t *fp = sp->rs_finfo;
rfs4_lo_state_t *lsp;
int fflags = 0;
* If this call is part of the larger closing down of client
* state then it is just easier to release all locks
* associated with this client instead of going through each
* individual file and cleaning locks there.
close_of_client ?
Yes
No
sp->rs_owner->ro_client->rc_unlksys_completed == FALSE &&
!list_is_empty(&sp->rs_lostatelist) &&
sp->rs_owner->ro_client->rc_sysidt != LM_NOSYSID ?
Is the PxFS kernel module loaded?
Yes
No
lm_remove_file_locks != NULL ?
Yes
No
int new_sysid;
Encode the cluster nodeid in new sysid
new_sysid = sp->rs_owner->ro_client->rc_sysidt;
lm_set_nlmid_flk(&new_sysid);
* This PxFS routine removes file locks for a
* client over all nodes of a cluster.
NFS4_DEBUG(rfs4_debug, (CE_NOTE,
"lm_remove_file_locks(sysid=0x%x)\n", new_sysid));
(*lm_remove_file_locks)(new_sysid);
struct flock64 flk;
Release all locks for this client
flk.l_type = F_UNLKSYS;
flk.l_whence = 0;
flk.l_start = 0;
flk.l_len = 0;
flk.l_sysid = sp->rs_owner->ro_client->rc_sysidt;
flk.l_pid = 0;
(void) VOP_FRLOCK(sp->rs_finfo->rf_vp, F_SETLK, &flk, F_REMOTELOCK | FREAD | FWRITE, (u_offset_t0, NULL, CRED(), NULL);
sp->rs_owner->ro_client->rc_unlksys_completed = TRUE;
* Release all locks on this file by this lock owner or at
* least mark the locks as having been released
lsp = list_head(&sp->rs_lostatelist)
lsp != NULL ?
* Release any shrlocks associated with this open state ID.
* This must be done before the rfs4_state gets marked closed.
Yes
No
lsp->rls_locks_cleaned = TRUE;
Was this already taken care of above?
sp->rs_owner->ro_client->rc_sysidt != LM_NOSYSID ?
Yes
No
(close_of_client && sp->rs_owner->ro_client->rc_sysidt != LM_NOSYSID ?
Yes
No
(void) cleanlocks(sp->rs_finfo->rf_vp, lsp->rls_locker->rl_pid, lsp->rls_locker->rl_client->rc_sysidt);
(void) rfs4_unshare(sp);
lsp = list_next(&sp->rs_lostatelist, lsp)
sp->rs_open_access ?
Yes
No
* Decrement the count for each access and deny bit that this
* state has contributed to the file.
* If this file counts go to zero
* clear the appropriate bit in the appropriate mask.
rfs4_dbe_lock(fp->rf_dbe);
sp->rs_open_access & OPEN4_SHARE_ACCESS_READ ?
Yes
No
fp->rf_access_read--;
fflags |= FREAD;
fp->rf_access_read == 0 ?
Yes
No
fp->rf_share_access &= ~OPEN4_SHARE_ACCESS_READ;
sp->rs_open_access & OPEN4_SHARE_ACCESS_WRITE ?
Yes
No
fp->rf_access_write--;
fflags |= FWRITE;
fp->rf_access_write == 0 ?
Yes
No
fp->rf_share_access &= ~OPEN4_SHARE_ACCESS_WRITE;
sp->rs_open_deny & OPEN4_SHARE_DENY_READ ?
Yes
No
fp->rf_deny_read--;
fp->rf_deny_read == 0 ?
Yes
No
fp->rf_share_deny &= ~OPEN4_SHARE_DENY_READ;
sp->rs_open_deny & OPEN4_SHARE_DENY_WRITE ?
Yes
No
fp->rf_deny_write--;
fp->rf_deny_write == 0 ?
Yes
No
fp->rf_share_deny &= ~OPEN4_SHARE_DENY_WRITE;
(void) VOP_CLOSE(fp->rf_vp, fflags, 1, (offset_t)0, cr, NULL);
rfs4_dbe_unlock(fp->rf_dbe);
sp->rs_open_access = 0;
sp->rs_open_deny = 0;
END

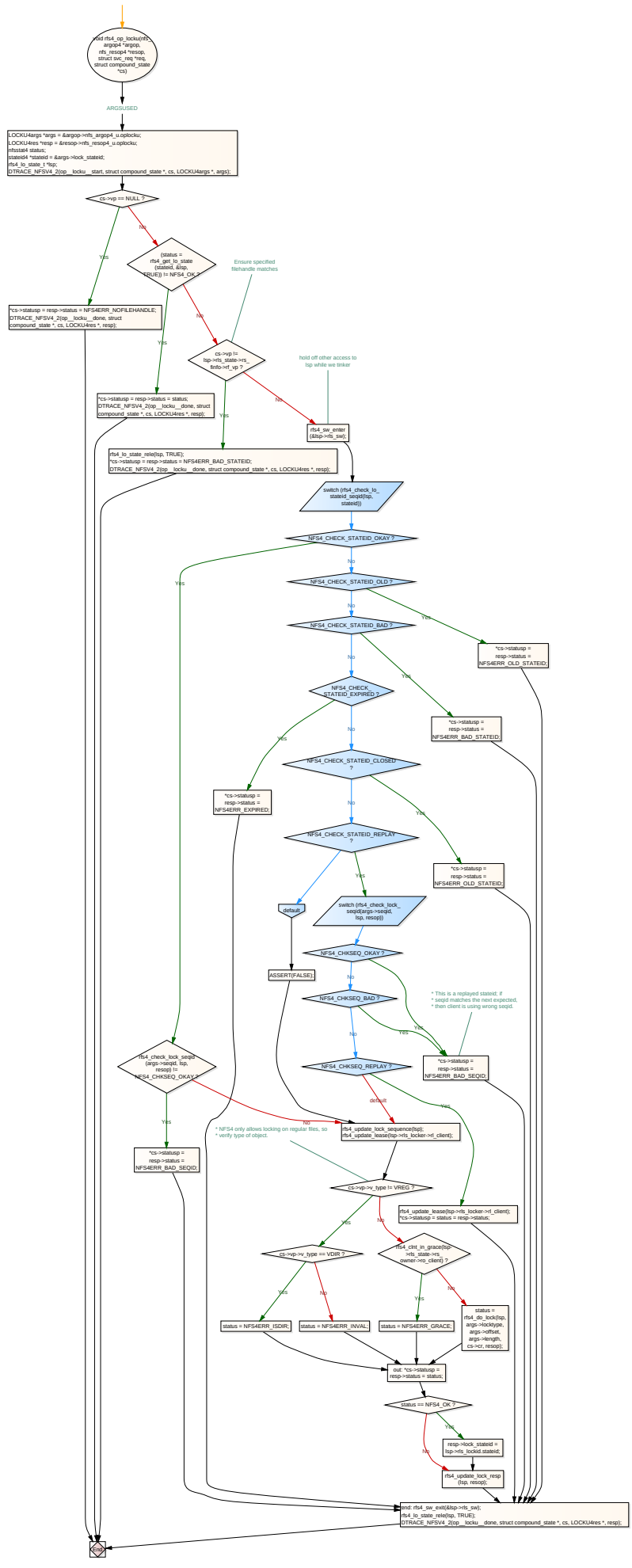

static nfsstat4 lock_denied(LOCK4denied *dp, struct flock64 *flk)
* lock_denied: Fill in a LOCK4deneid structure given an flock64 structure.
rfs4_lockowner_t *lo;
rfs4_client_t *cp;
uint32_t len;
lo = rfs4_findlockowner_by_pid(flk->l_pid);
lo != NULL ?
Yes
No
* Its not a NFS4 lock. We take advantage that the upper 32 bits
* of the client id contain the boot time for a NFS4 lock. So we
* fabricate and identity by setting clientid to the sysid, and
* the lock owner to the pid.
cp = lo->rl_client;
dp->owner.clientid = flk->l_sysid;
len = sizeof (pid_t);
dp->owner.owner_len = len;
dp->owner.owner_val = kmem_alloc(len, KM_SLEEP);
bcopy(&flk->l_pid, dp->owner.owner_val, len);
rfs4_lease_expired(cp) ?
Yes
No
dp->owner.clientid = lo->rl_owner.clientid;
len = lo->rl_owner.owner_len;
dp->owner.owner_val = kmem_alloc(len, KM_SLEEP);
bcopy(lo->rl_owner.owner_val, dp->owner.owner_val, len);
dp->owner.owner_len = len;
rfs4_lockowner_rele(lo);
rfs4_lockowner_rele(lo);
rfs4_dbe_hold(cp->rc_dbe);
rfs4_client_close(cp);
return (NFS4ERR_EXPIRED);
finish: dp->offset = flk->l_start;
dp->length = flk->l_len;
flk->l_type == F_RDLCK ?
Yes
No
flk->l_type == F_WRLCK ?
no mapping from POSIX ltype to v4
dp->locktype = READ_LT;
Yes
No
dp->locktype = WRITE_LT;
return (NFS4ERR_INVAL);
return (NFS4_OK);
End

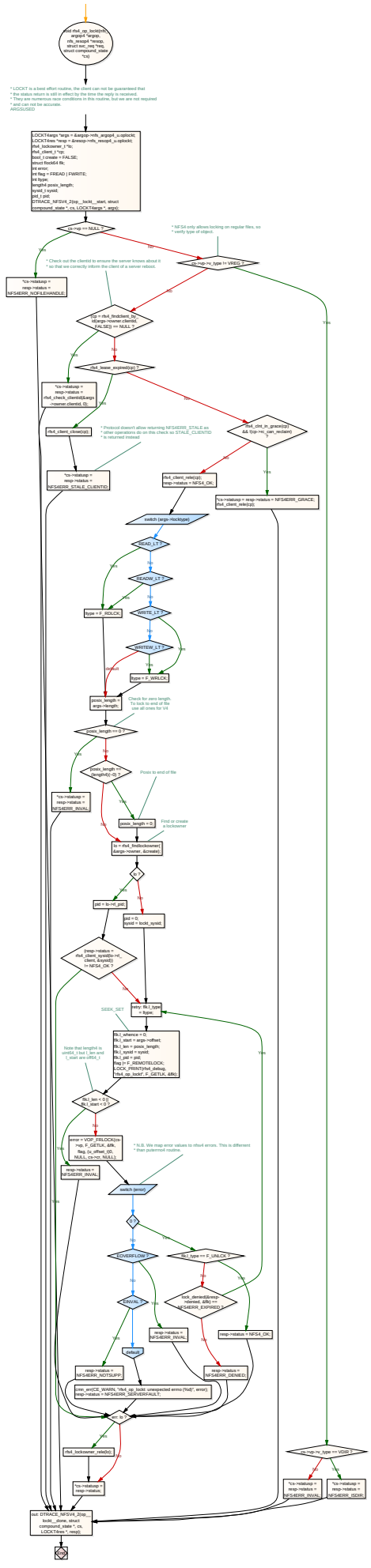

static int
setlock(vnode_t *vp,
struct flock64 *flock,
int flag, cred_t *cred)
int error;
struct flock64 flk;
int i;
clock_t delaytime;
int cmd;
cmd = nbl_need_check(vp) ?
F_SETLK_NBMAND : F_SETLK;
retry: delaytime = MSEC_TO_TICK_ROUNDUP(rfs4_lock_delay);
i = 0
i < rfs4_maxlock_tries ?
Yes
No
LOCK_PRINT(rfs4_debug, "setlock", cmd, flock);
error = VOP_FRLOCK(vp, cmd, flock,
flag, (u_offset_t)0, NULL, cred, NULL);
error != EAGAIN &&
error != EACCES ?
No
Yes
i < rfs4_maxlock_tries
- 1 ?
error == EAGAIN ||
error == EACCES ?
Get the owner of the lock
No
Yes
Yes
delay(delaytime);
delaytime *= 2;
No
flk = *flock;
LOCK_PRINT(rfs4_debug, "setlock", F_GETLK, &flk);
i++
VOP_FRLOCK(vp, F_GETLK,
&flk, flag,
(u_offset_t)0, NULL,
cred, NULL) == 0 ?
Yes
flk.l_type == F_UNLCK ?
No
Yes
No
No longer locked, retry
*flock = flk;
LOCK_PRINT(rfs4_debug, "setlock(blocking lock)", F_GETLK, &flk);
return (error);
End

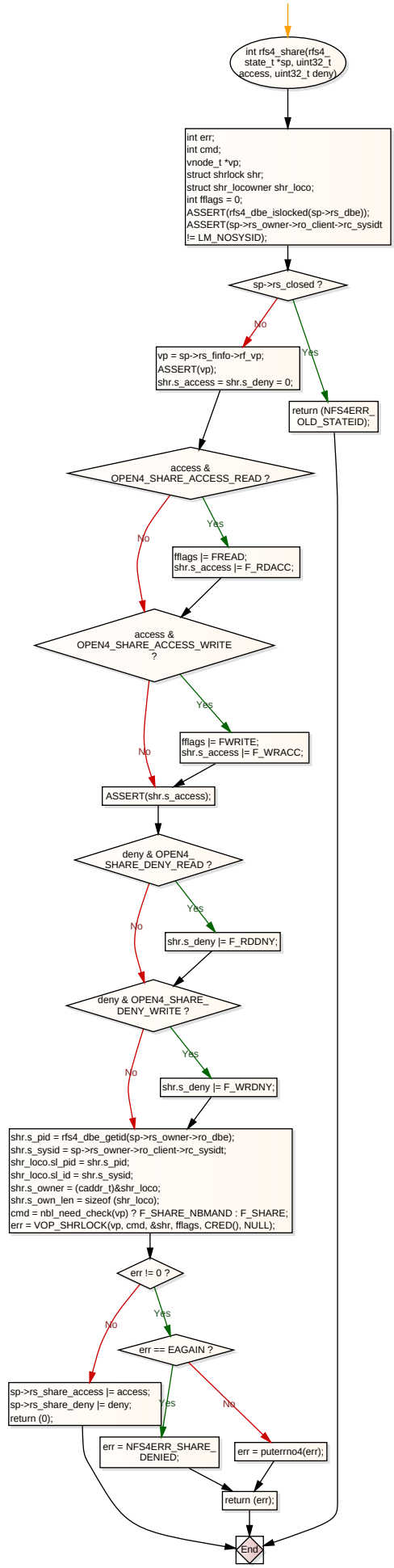

static nfsstat4
nfs4_do_lock(nfs4_lo_
state_t *lsp,
nfs_lock_type4 locktype,
offset4 offset, length4
length, cred_t *cred,
nfs_resop4 *resop)
ARGSUSED
nfsstat4 status;
lockowner_t *lo = lsp->ls_locker;
nfs4_state_t *sp = lsp->ls_state;
struct flock64 flock;
int16_t ltype;
int flag;
int error;
sysid_t sysid;
LOCK4res *lres;
rfs4_lease_expired(lo
->rl_client) ?
Yes
No
(status =
rfs4_client_sysid(lo->rl_
client, &sysid))
!= NFS4_OK ?
Check for zero length.
To lock to end of file
use all ones for V4
return (NFS4ERR_EXPIRED);
Yes
No
length == 0 ?
return (status);
Yes
No
length == (length4)(-1) ?
Yes
return (NFS4ERR_INVAL);
Posix to end of file
Yes
No
length = 0;
retry: rfs4_dbe_
lock(sp->rs_dbe);
sp->rs_closed ?
Yes
No
Yes
resop->resop
!= OP_LOCKU ?
rfs4_dbe_unlock(sp->rs_dbe);
return (NFS4ERR_OLD_STATEID);
Yes
No
switch (locktype)
ltype = F_UNLCK;
READ_LT ?
No
Yes
READW_LT ?
No
Yes
WRITE_LT ?
(lsp->rs_share_access &
OPEN4_SHARE_ACCESS_READ)
== 0 ?
Yes
No
WRITEW_LT ?
No
Yes
ltype = F_RDLCK;
Yes
default
(lsp->rs_share_access &
OPEN4_SHARE_ACCESS_WRITE)
== 0 ?
Yes
No
rfs4_dbe_unlock(lsp->rs_dbe);
return (NFS4ERR_OPENMODE);
ltype = F_WRLCK;
rfs4_dbe_unlock(sp->rs_dbe);
return (NFS4ERR_OPENMODE);
flock.l_type = ltype;
SEEK_SET
flock.l_whence = 0;
flock.l_start = offset;
flock.l_len = length;
flock.l_sysid = sysid;
flock.l_pid = lsp->ls_locker->rl_pid;
Note that length4 is
uint64_t but l_len and
l_start are off64_t
flock.l_len < 0 ||
flock.l_start < 0 ?
* N.B. FREAD has the same value as OPEN4_SHARE_ACCESS_READ and
* FWRITE has the same value as OPEN4_SHARE_ACCESS_WRITE.
Yes
No
flag = (int)lsp->rs_share_access | F_REMOTELOCK;
error = setlock(sp->rs_finfo->rf_vp, &flock, flag, cred);
rfs4_dbe_unlock(sp->rs_dbe);
return (NFS4ERR_INVAL);
error == 0 ?
Yes
No
rfs4_dbe_lock(lsp->ls_dbe);
next_stateid(&lsp->ls_lockid);
rfs4_dbe_unlock(lsp->ls_dbe);
rfs4_dbe_unlock
(lsp->rs_dbe);
* N.B. We map error values to nfsv4 errors. This is different
* than puterrno4 routine.
switch (error)
0 ?
No
Yes
EAGAIN ?
status = NFS4_OK;
No
Yes
EACCES ?
No
Yes
Old value
Can only get here if op is OP_LOCK
ENOLCK ?
No
Yes
ASSERT(resop->resop == OP_LOCK);
lres = &resop->nfs_resop4_u.oplock;
status = NFS4ERR_DENIED;
EOVERFLOW ?
No
Yes
lock_denied(&lres
->LOCK4res_u.denied,
&flock) ==
NFS4ERR_EXPIRED ?
Yes
No
status = NFS4ERR_DELAY;
EINVAL ?
No
Yes
status = NFS4ERR_INVAL;
default
status = NFS4ERR_NOTSUPP;
status =
NFS4ERR_SERVERFAULT;
return (status);
Yes

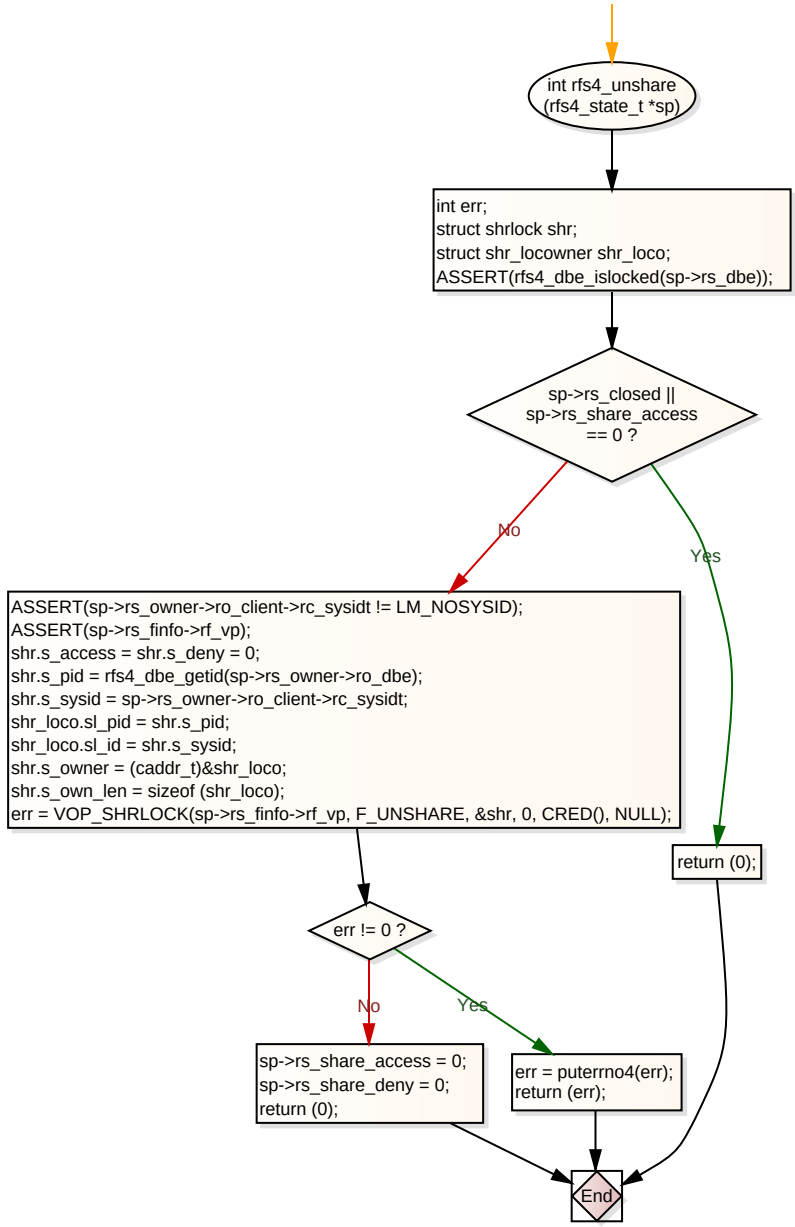


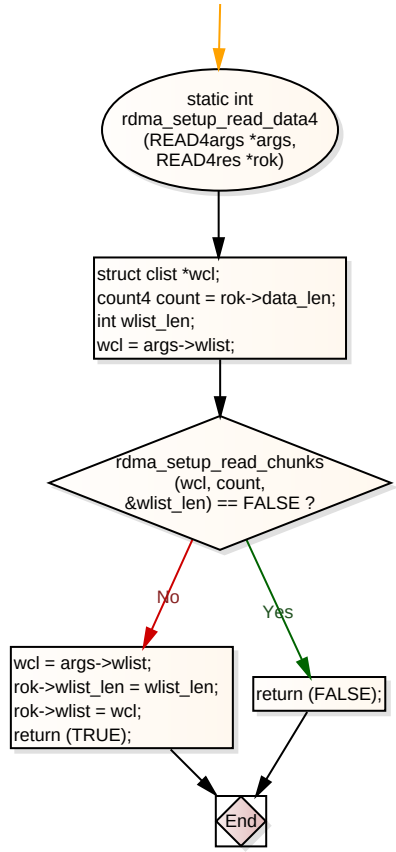

static void
lock_denied_free
(nfs_resop4 *resop)
free function
for LOCK/LOCKT
LOCK4denied *dp = NULL;
switch (resop->resop)
OP_LOCK ?
No
Yes
OP_LOCKT ?
Yes
default
resop->nfs_resop4_u
.oplockt.status ==
NFS4ERR_DENIED ?
resop->nfs_resop4_u
.oplock.status ==
NFS4ERR_DENIED ?
Yes
Yes
No
No
dp = &resop->nfs_resop4_u
.oplockt.denied;
dp = &resop->nfs_resop4_u
.oplock.LOCK4res_
u.denied;
dp ?
Yes
No
kmem_free(dp->owner
.owner_val,
dp->owner.owner_len);
End

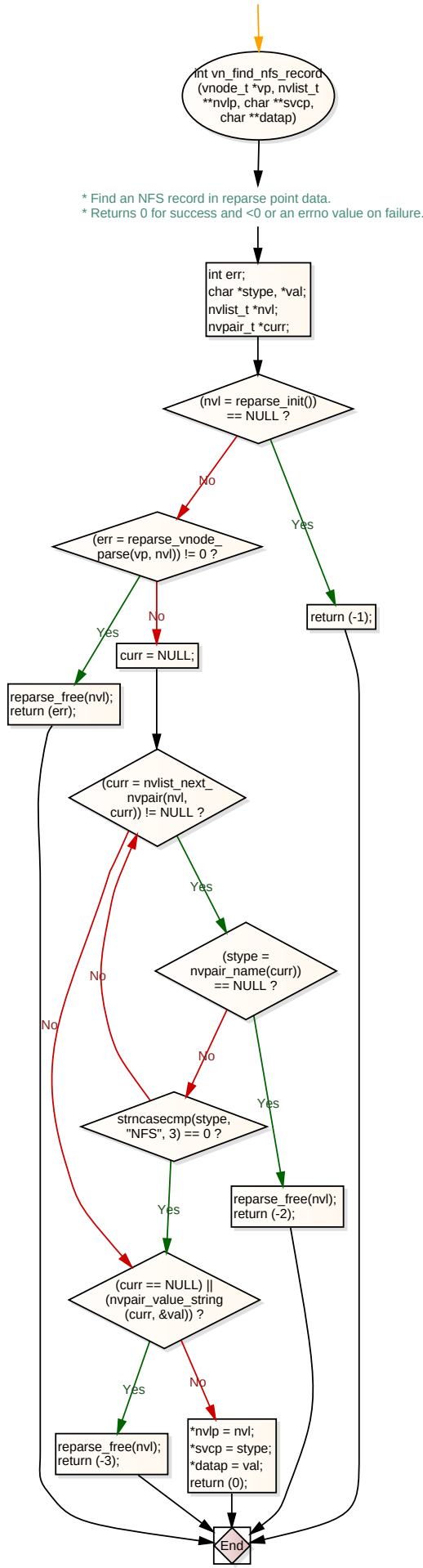

void nfs4_op_locku(nfs_
argop4 *argop,
nfs_resop4 *resop,
struct svc_req *req,
struct compound_state
*cs)
ARGSUSED
LOCKU4args *args = &argop->nfs_argop4_u.oplocku;
LOCKU4res *resp = &resop->nfs_resop4_u.oplocku;
nfsstat4 status;
stateid4 *stateid = &args->lock_stateid;
nfs4_lo_state_t *lsp;
DTRACE_NFSV4_2(op__locku__start, struct compound_state *, cs, LOCKU4args *, args);
cs->vp == NULL ?
Yes
No
(status =
nfs4_get_lo_state
(stateid, &lsp,
TRUE)) != NFS4_OK ?
Ensure specified
filehandle matches
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
DTRACE_NFSV4_2(op__locku__done, struct
compound_state *, cs, LOCKU4res *, resp);
cs->vp !=
lsp->ls_state->rs_
finfo->rf_vp ?
hold off other access to
lsp while we tinker
*cs->statusp = resp->status = status;
DTRACE_NFSV4_2(op__locku__done, struct
compound_state *, cs, LOCKU4res *, resp);
nfs4_sw_enter
(&lsp->ls_sw);
nfs4_lo_state_rele(lsp, TRUE);
*cs->statusp = resp->status = NFS4ERR_BAD_STATEID;
DTRACE_NFSV4_2(op__locku__done, struct compound_state *, cs, LOCKU4res *, resp);
switch (nfs4_check_lo_
stateid_seqid(lsp,
stateid))
NFS4_CHECK_STATEID_OKAY ?
NFS4_CHECK_STATEID_OLD ?
NFS4_CHECK_STATEID_BAD ?
*cs->statusp =
resp->status =
NFS4ERR_OLD_STATEID;
NFS4_CHECK_
STATEID_EXPIRED ?
*cs->statusp =
resp->status =
NFS4ERR_BAD_STATEID;
NFS4_CHECK_STATEID_CLOSED
?
*cs->statusp =
resp->status =
NFS4ERR_EXPIRED;
NFS4_CHECK_STATEID_REPLAY
?
*cs->statusp =
resp->status =
NFS4ERR_OLD_STATEID;
default
switch (nfs4_check_lock_
seqid(args->seqid,
lsp, resop))
NFS4_CHKSEQ_OKAY ?
ASSERT(FALSE);
NFS4_CHKSEQ_BAD ?
* This is a replayed stateid; if
* seqid matches the next expected,
* then client is using wrong seqid.
nfs4_check_lock_seqid
(args->seqid, lsp,
resop) !=
NFS4_CHKSEQ_OKAY ?
*cs->statusp =
resp->status =
NFS4ERR_BAD_SEQID;
NFS4_CHKSEQ_REPLAY ?
*cs->statusp =
resp->status =
NFS4ERR_BAD_SEQID;
default
* NFS4 only allows locking on regular files, so
* verify type of object.
nfs4_update_lock_sequence(lsp);
nfs4_update_lease(lsp->ls_locker->rl_client);
cs->vp->v_type != VREG ?
nfs4_update_lease(lsp->ls_locker->rl_client);
*cs->statusp = status = resp->status;
cs->vp->v_type == VDIR ?
nfs4_clnt_in_grace(lsp->
ls_state->rs_
owner->ro_client) ?
status = NFS4ERR_ISDIR;
status = NFS4ERR_INVAL;
status = NFS4ERR_GRACE;
status =
nfs4_do_lock(lsp,
args->locktype,
args->offset,
args->length,
cs->cr, resop);
out: *cs->statusp =
resp->status = status;
status == NFS4_OK ?
resp->lock_stateid =
lsp->ls_lockid.stateid;
nfs4_update_lock_resp
(lsp, resop);
end: nfs4_sw_exit(&lsp->ls_sw);
nfs4_lo_state_rele(lsp, TRUE);
DTRACE_NFSV4_2(op__locku__done, struct compound_state *, cs, LOCKU4res *, resp);
End

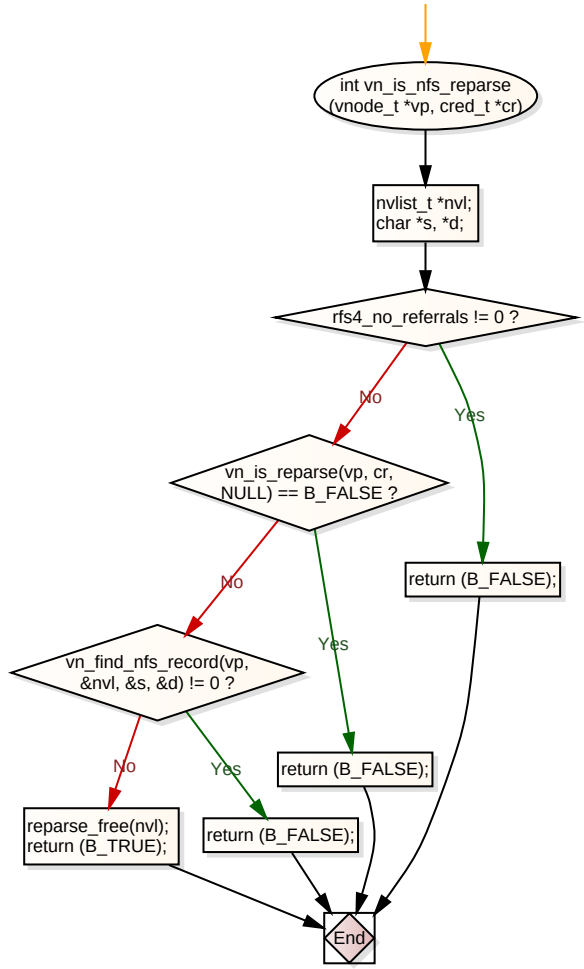

nfsd4_op_lock(nfs4_argsp4 *argsp, nfs_resop4 *resop, struct svc_req *req, struct compound_state *cs)
* LOCKT is a best effort routine, the client can not be guaranteed that
* the status return is still in effect by the time the reply is received.
* They are numerous race conditions in this routine, but we are not required
* and can not be accurate.
ARGSUSED
LOCK4resok *lresp = &resop->nfs_resop4_u.oplockt;
LOCKT4res *lresp = &resop->nfs_resop4_u.oplockt;
nfs4_lockowner_t *lo;
nfs4_client_t *cp;
bool_t create = FALSE;
struct flock64 fl;
int error;
int flag = FREAD | FWRITE;
int ltype;
length4 posix_length;
sysid_t sysid;
pid_t pid;
DTRACE_NFSV4_2(op__lock__start, struct compound_state *, cs, LOCK4args *, argsp);
cs->vp == NULL ?
Yes
No
* NFS4 only allows locking on regular files, so
* verify type of object.
cs->vp->v_type != VREG ?
* Check out the clientid to ensure the server knows about it
* so that we correctly inform the client of a server reboot.
*cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
cp = rfs4_findclient_by_id(argsp->owner.clientid, FALSE) == NULL ?
rfs4_lease_expired(cp) ?
*cs->statusp = resp->status = rfs4_check_clientid(&argsp->owner.clientid, 0);
rfs4_client_close(cp);
* Protocol doesn't allow returning NFS4ERR_STALE as
* other operations do on this check so STALE_CLIENTID
* is returned instead.
*cs->statusp = resp->status = NFS4ERR_STALE_CLIENTID;
rfs4_dbe_is_grace(cp->rc_dbe) && !(cp->rc_can_reclaim) ?
rfs4_client_rele(cp);
resp->status = NFS4_OK;
*cs->statusp = resp->status = NFS4ERR_GRACE;
rfs4_client_rele(cp);
switch (argsp->locktype)
READ_LT ?
READW_LT ?
ltype = F_RDLCK;
WRITE_LT ?
WRITEW_LT ?
default
ltype = F_WRLCK;
posix_length = args->length;
Check for zero length.
To lock to end of file
use all ones for V4
posix_length == 0 ?
posix_length != (length4)(~0) ?
*cs->statusp = resp->status = NFS4ERR_INVAL;
Posix to end of file
posix_length = 0;
Find or create a lockowner
lo = rfs4_findlockowner(&argsp->owner, &create);
lo ?
pid = lo->rl_pid;
pid = 0;
sysid = lockt_sysid;
(resp->status = rfs4_client_sysid((*cl), client, &sysid)) != NFS4_OK ?
retry: fl.l_type = ltype;
SEEK_SET
fl.l_whence = 0;
fl.l_start = argsp->offset;
fl.l_len = posix_length;
fl.l_sysid = sysid;
fl.l_pid = pid;
flag |= F_REMOTELOCK;
LOCK_PRINT(nfs4_debug, "nfs4_op_lock(F_F_GETLK, &fl);
Note that length4 is uint64_t but l_len and
l_start are off64_t
fl.l_len < 0 ?
fl.l_start < 0 ?
error = VOP_FRLOCK(cs->vp, F_GETLK, &fl, flag, (u_offset_t)0, NULL, cs->cr, NULL);
resp->status = NFS4ERR_INVAL;
* N.B. We map error values to nfsv4 errors. This is different
* than puterrno4 routine.
switch (error)
0 ?
EOVERFLOW
EINVAL ?
default
fl.l_type == F_UNLCK ?
lock_denied(&resp->denied, &fl) == NFS4ERR_EXPIRED ?
resp->status = NFS4_OK;
resp->status = NFS4ERR_INVAL;
resp->status = NFS4ERR_NOTSUPP;
resp->status = NFS4ERR_DENIED;
cmn_err(CE_WARN, "nfs4_op_lock: unexpected errno (%d)", error);
resp->status = NFS4ERR_SERVERFAULT;
lo ?
rfs4_lockowner_rele(lo);
*cs->statusp = resp->status;
cs->vp->v_type == VDIR ?
*cs->statusp = resp->status = NFS4ERR_INVAL;
*cs->statusp = resp->status = NFS4ERR_ISDIR;
out: DTRACE_NFSV4_2(op__lock__done, struct compound_state *, cs, LOCK4res *, resp);

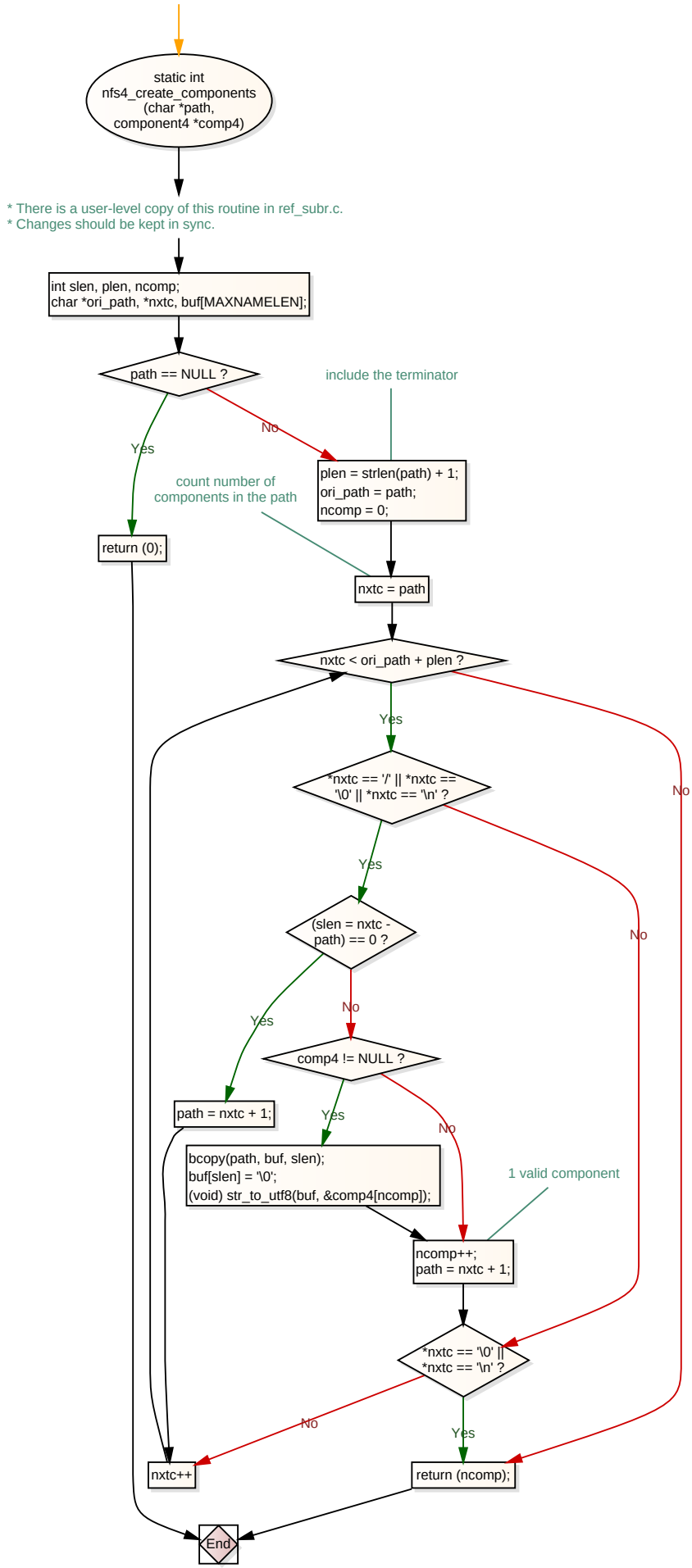

int rfs4_share(rfs4_
state_t *sp, uint32_t
access, uint32_t deny)
int err;
int cmd;
vnode_t *vp;
struct shrlock shr;
struct shr_locowner shr_loco;
int fflags = 0;
ASSERT(rfs4_dbe_islocked(sp->rs_dbe));
ASSERT(sp->rs_owner->ro_client->rc_sysidt
!= LM_NOSYSID);
sp->rs_closed ?
No
Yes
vp = sp->rs_finfo->rf_vp;
ASSERT(vp);
shr.s_access = shr.s_deny = 0;
return (NFS4ERR_
OLD_STATEID);
access &
OPEN4_SHARE_ACCESS_READ ?
No
Yes
fflags |= FREAD;
shr.s_access |= F_RDACC;
access &
OPEN4_SHARE_ACCESS_WRITE
?
No
Yes
fflags |= FWRITE;
shr.s_access |= F_WRACC;
ASSERT(shr.s_access);
deny & OPEN4_
SHARE_DENY_READ ?
No
Yes
shr.s_deny |= F_RDDNY;
deny & OPEN4_SHARE_
DENY_WRITE ?
No
Yes
shr.s_deny |= F_WRDNY;
shr.s_pid = rfs4_dbe_getid(sp->rs_owner->ro_dbe);
shr.s_sysid = sp->rs_owner->ro_client->rc_sysidt;
shr_loco.sl_pid = shr.s_pid;
shr_loco.sl_id = shr.s_sysid;
shr.s_owner = (caddr_t)&shr_loco;
shr.s_own_len = sizeof (shr_loco);
cmd = nbl_need_check(vp) ? F_SHARE_NBMAND : F_SHARE;
err = VOP_SHRLOCK(vp, cmd, &shr, fflags, CRED(), NULL);
err != 0 ?
No
Yes
err == EAGAIN ?
sp->rs_share_access |= access;
sp->rs_share_deny |= deny;
return (0);
Yes
No
err = NFS4ERR_SHARE_
DENIED;
err = puterrno4(err);
return (err);
End

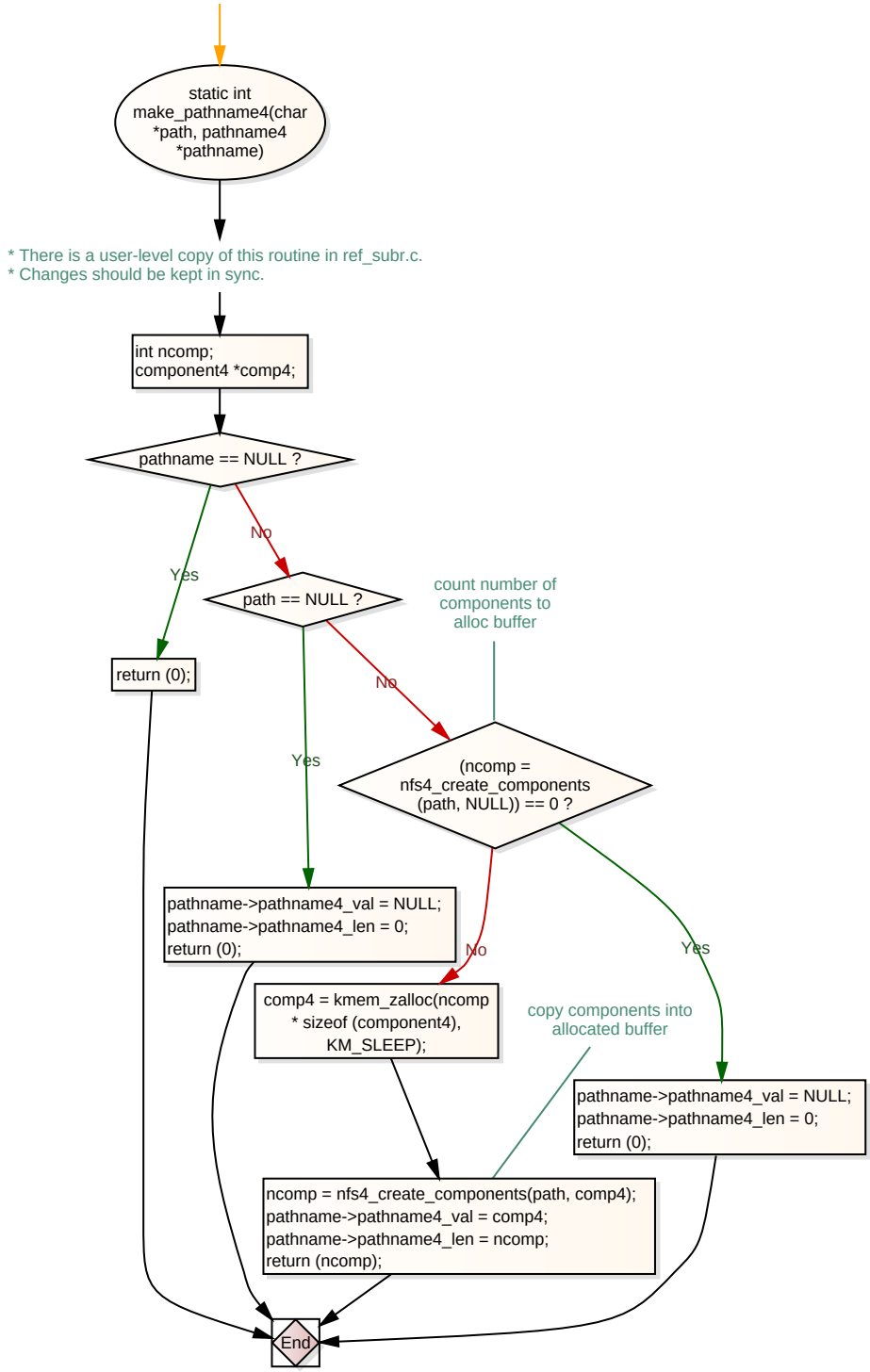

int rfs4_unshare
(rfs4_state_t *sp)
int err;
struct shrlock shr;
struct shr_locowner shr_loco;
ASSERT(rfs4_dbe_islocked(sp->rs_dbe));
sp->rs_closed ||
sp->rs_share_access
== 0 ?
No
Yes
ASSERT(sp->rs_owner->ro_client->rc_sysidt != LM_NOSYSID);
ASSERT(sp->rs_finfo->rf_vp);
shr.s_access = shr.s_deny = 0;
shr.s_pid = rfs4_dbe_getid(sp->rs_owner->ro_dbe);
shr.s_sysid = sp->rs_owner->ro_client->rc_sysidt;
shr_loco.sl_pid = shr.s_pid;
shr_loco.sl_id = shr.s_sysid;
shr.s_owner = (caddr_t)&shr_loco;
shr.s_own_len = sizeof (shr_loco);
err = VOP_SHRLOCK(sp->rs_finfo->rf_vp, F_UNSHARE, &shr, 0, CRED(), NULL);
return (0);
err != 0 ?
No
Yes
sp->rs_share_access = 0;
sp->rs_share_deny = 0;
return (0);
err = puterrno4(err);
return (err);
End

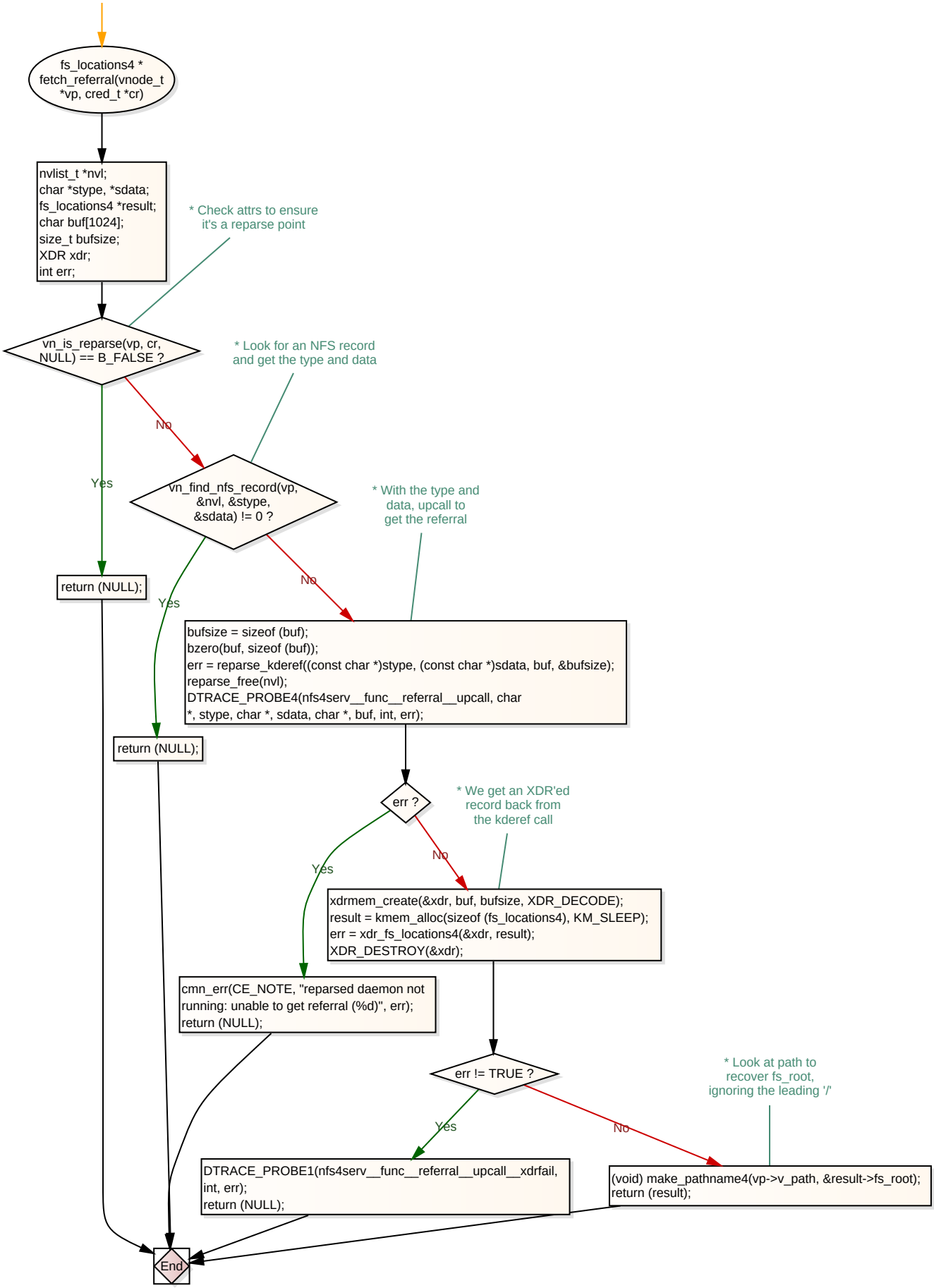

static int
rdma_setup_read_data4
(READ4args *args,
READ4res *rok)
struct clist *wcl;
count4 count = rok->data_len;
int wlist_len;
wcl = args->wlist;
rdma_setup_read_chunks
(wcl, count,
&wlist_len) == FALSE ?
No
Yes
wcl = args->wlist;
rok->wlist_len = wlist_len;
rok->wlist = wcl;
return (TRUE);
return (FALSE);
End

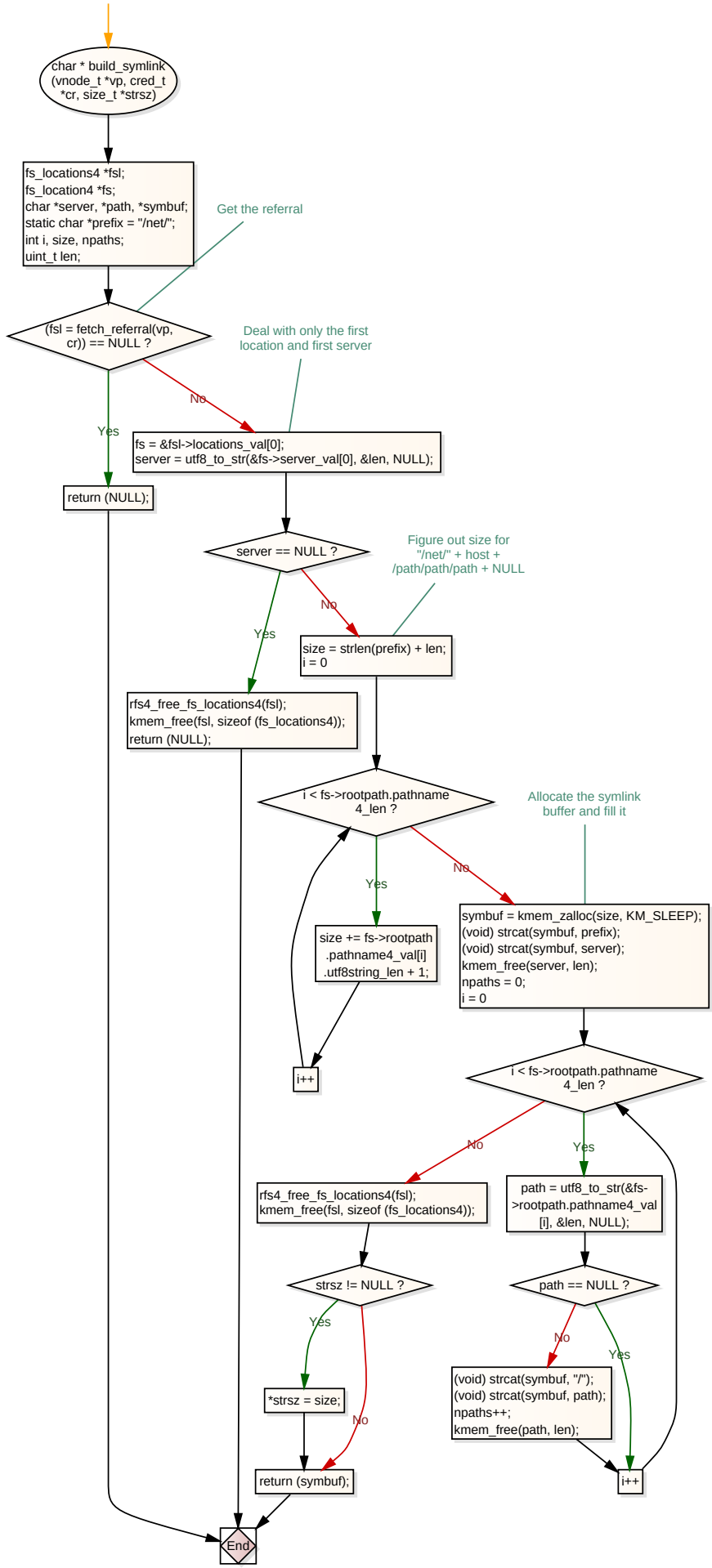

int vn_find_nfs_record
(vnode_t *vp, nvlist_t
**nvlp, char **svcp,
char **datap)
* Find an NFS record in reparse point data.
* Returns 0 for success and <0 or an errno value on failure.
int err;
char *stype, *val;
nvlist_t *nvl;
nvpair_t *curr;
(nvl = reparse_init())
== NULL ?
No
Yes
(err = reparse_vnode_
parse(vp, nvl)) != 0 ?
return (-1);
Yes
No
curr = NULL;
reparse_free(nvl);
return (err);
(curr = nvlist_next_
nvpair(nvl,
curr)) != NULL ?
Yes
(stype =
nvpair_name(curr))
== NULL ?
No
No
No
Yes
strncasecmp(stype,
"NFS", 3) == 0 ?
reparse_free(nvl);
return (-2);
Yes
(curr == NULL) ||
(nvpair_value_string
(curr, &val)) ?
Yes
No
reparse_free(nvl);
return (-3);
*nvlp = nvl;
*svcp = stype;
*datap = val;
return (0);
End

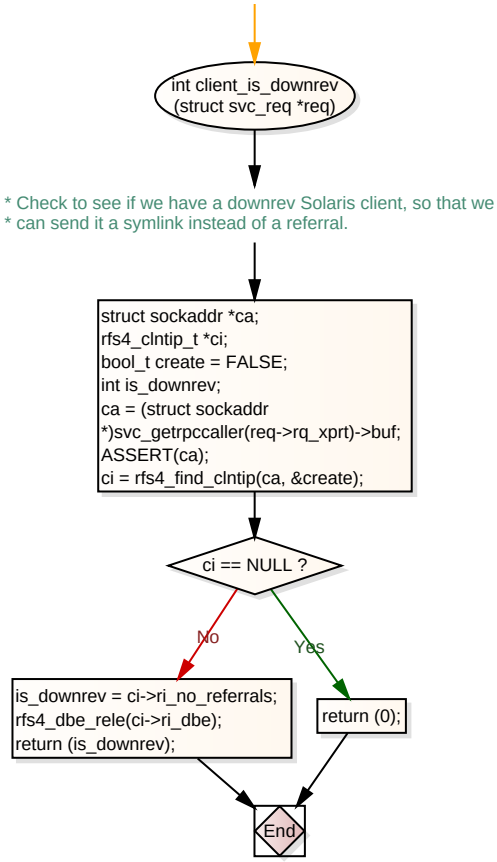

int vn_is_nfs_reparse
(vnode_t *vp, cred_t *cr)
nvlist_t *nvl;
char *s, *d;
rfs4_no_referrals != 0 ?
No
Yes
vn_is_reparse(vp, cr,
NULL) == B_FALSE ?
return (B_FALSE);
No
Yes
vn_find_nfs_record(vp,
&nvl, &s, &d) != 0 ?
No
Yes
return (B_FALSE);
reparse_free(nvl);
return (B_TRUE);
return (B_FALSE);
End


static int
nfs4_create_components
(char *path,
component4 *comp4)
* There is a user-level copy of this routine in ref_subr.c.
* Changes should be kept in sync.
int slen, plen, ncomp;
char *ori_path, *nxtc, buf[MAXNAMELEN];
path == NULL ?
Yes
No
return (0);
include the terminator
count number of
components in the path
plen = strlen(path) + 1;
ori_path = path;
ncomp = 0;
nxtc = path
nxtc < ori_path + plen ?
Yes
No
*nxtc == '/' || *nxtc ==
'\0' || *nxtc == '\n' ?
Yes
No
(slen = nxtc -
path) == 0 ?
Yes
No
comp4 != NULL ?
Yes
No
path = nxtc + 1;
bcopy(path, buf, slen);
buf[slen] = '\0';
(void) str_to_utf8(buf, &comp4[ncomp]);
1 valid component
ncomp++;
path = nxtc + 1;
*nxtc == '\0' ||
*nxtc == '\n' ?
No
Yes
nxtc++
return (ncomp);
End


static int
make_pathname4(char
*path, pathname4
*pathname)
* There is a user-level copy of this routine in ref_subr.c.
* Changes should be kept in sync.
int ncomp;
component4 *comp4;
pathname == NULL ?
Yes
No
path == NULL ?
count number of
components to
alloc buffer
return (0);
No
Yes
(ncomp =
nfs4_create_components
(path, NULL)) == 0 ?
pathname->pathname4_val = NULL;
pathname->pathname4_len = 0;
return (0);
No
Yes
comp4 = kmem_zalloc(ncomp
* sizeof (component4),
KM_SLEEP);
copy components into
allocated buffer
pathname->pathname4_val = NULL;
pathname->pathname4_len = 0;
return (0);
ncomp = nfs4_create_components(path, comp4);
pathname->pathname4_val = comp4;
pathname->pathname4_len = ncomp;
return (ncomp);
End


fs_locations4 * fetch_referral(vnode_t *vp, cred_t *cr)
nvlist_t *nvl;
char *stype, *sdata;
fs_locations4 *result;
char buf[1024];
size_t bufsize;
XDR xdr;
int err;
* Check attrs to ensure it's a reparse point
vn_is_reparse(vp, cr, NULL) == B_FALSE ?
Yes
No
* Look for an NFS record and get the type and data
vn_find_nfs_record(vp, &nvl, &stype, &sdata) != 0 ?
Yes
No
* With the type and data, upcall to get the referral
return (NULL);
bufsize = sizeof (buf);
bzero(buf, sizeof (buf));
err = reparse_kderef((const char *)stype, (const char *)sdata, buf, &bufsize);
reparse_free(nvl);
DTRACE_PROBE4(nfs4serv__func__referral__upcall, char *, stype, char *, sdata, char *, buf, int, err);
return (NULL);
err ?
Yes
No
* We get an XDR'ed record back from the kderef call
xdrmem_create(&xdr, buf, bufsize, XDR_DECODE);
result = kmem_alloc(sizeof (fs_locations4), KM_SLEEP);
err = xdr_fs_locations4(&xdr, result);
XDR_DESTROY(&xdr);
cmn_err(CE_NOTE, "reparsed daemon not running: unable to get referral (%d)", err);
return (NULL);
err != TRUE ?
Yes
No
* Look at path to recover fs_root, ignoring the leading '/'
DTRACE_PROBE1(nfs4serv__func__referral__upcall__xdrfail, int, err);
return (NULL);
(void) make_pathname4(vp->v_path, &result->fs_root);
return (result);
End


char * build_symlink (vnode_t *vp, cred_t *cr, size_t *strsz)
fs_locations4 *fsl;
fs_location4 *fs;
char *server, *path, *symbuf;
static char *prefix = "/net/";
int i, size, npaths;
uint_t len;
Get the referral
(fsl = fetch_referral(vp, cr)) == NULL ?
Deal with only the first location and first server
No
Yes
fs = &fsl->locations_val[0];
server = utf8_to_str(&fs->server_val[0], &len, NULL);
return (NULL);
server == NULL ?
Figure out size for "/net/" + host + /path/path/path + NULL
No
Yes
size = strlen(prefix) + len;
i = 0
rfs4_free_fs_locations4(fsl);
kmem_free(fsl, sizeof (fs_locations4));
return (NULL);
i < fs->rootpath.pathname 4_len ?
Allocate the symlink buffer and fill it
Yes
No
size += fs->rootpath .pathname4_val[i] .utf8string_len + 1;
symbuf = kmem_zalloc(size, KM_SLEEP);
(void) strcat(symbuf, prefix);
(void) strcat(symbuf, server);
kmem_free(server, len);
npaths = 0;
i = 0
i++
i < fs->rootpath.pathname 4_len ?
No
Yes
rfs4_free_fs_locations4(fsl);
kmem_free(fsl, sizeof (fs_locations4));
path = utf8_to_str(&fs->rootpath.pathname4_val [i], &len, NULL);
strsz != NULL ?
path == NULL ?
Yes
No
No
Yes
*strsz = size;
(void) strcat(symbuf, "/");
(void) strcat(symbuf, path);
npaths++;
kmem_free(path, len);
No
return (symbuf);
i++
End


int client_is_downrev
(struct svc_req *req)
* Check to see if we have a downrev Solaris client, so that we
* can send it a symlink instead of a referral.
struct sockaddr *ca;
rfs4_clntip_t *ci;
bool_t create = FALSE;
int is_downrev;
ca = (struct sockaddr
*)svc_getrpccaller(req->rq_xprt)->buf;
ASSERT(ca);
ci = rfs4_find_clntip(ca, &create);
ci == NULL ?
No
Yes
is_downrev = ci->ri_no_referrals;
rfs4_dbe_rele(ci->ri_dbe);
return (is_downrev);
return (0);
End