```
**********************************************************
  103640 Wed Sep 14 16:21:02 2016
new/usr/src/uts/common/fs/nfs/nfs3_srv.c
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_

2660 void
2661 rfs3_rename(RENAME3args *args, RENAME3res *resp, struct exportinfo *exi,
2662     struct svc_req *req, cred_t *cr, bool_t ro)
2663 {
2664         int error = 0;
2665         vnode_t *fvp;
2666         vnode_t *tvp;
2667         vnode_t *targvp;
2668         struct vattr *fbvap;
2669         struct vattr fbva;
2670         struct vattr *favap;
2671         struct vattr fava;
2672         struct vattr *tbvap;
2673         struct vattr tbva;
2674         struct vattr *tavap;
2675         struct vattr tava;
2676         nfs_fh3 *fh3;
2677         struct exportinfo *to_exi;
2678         vnode_t *srcvp = NULL;
2679         bslabel_t *clabel;
2680         struct sockaddr *ca;
2681         char *name = NULL;
2682         char *toname = NULL;

2684         fbvap = NULL;
2685         favap = NULL;
2686         tbvap = NULL;
2687         tavap = NULL;
2688         tvp = NULL;

2690         fvp = nfs3_fhtovp(&args->from.dir, exi);

2692         DTRACE_NFSV3_4(op__rename__start, struct svc_req *, req,
2693             cred_t *, cr, vnode_t *, fvp, RENAME3args *, args);

2695         if (fvp == NULL) {
2696                 error = ESTALE;
2697                 goto err;
2698         }

2700         if (is_system_labeled()) {
2701                 clabel = req->rq_label;
2702                 ASSERT(clabel != NULL);
2703                 DTRACE_PROBE2(tx__rfs3__log__info__oprename__clabel, char *,
2704                     "got client label from request(1)", struct svc_req *, req);

2706                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
2707                         if (!do_rfs_label_check(clabel, fvp, EQUALITY_CHECK,
2708                             exi)) {
2709                                 resp->status = NFS3ERR_ACCES;
2710                                 goto err1;
2711                         }
2712                 }
2713         }

2715         fbva.va_mask = AT_ALL;
2716         fbvap = VOP_GETATTR(fvp, &fbva, 0, cr, NULL) ? NULL : &fbva;
2717         favap = fbvap;
```



```
2719         fh3 = &args->to.dir;
2720         to_exi = checkexport(&fh3->fh3_fsid, FH3TOXFIDP(fh3), NULL);
2720         to_exi = checkexport(&fh3->fh3_fsid, FH3TOXFIDP(fh3));
2721         if (to_exi == NULL) {
2722                 resp->status = NFS3ERR_ACCES;
2723                 goto err1;
2724         }
2725         exi_rele(to_exi);

2727         if (to_exi != exi) {
2728                 resp->status = NFS3ERR_XDEV;
2729                 goto err1;
2730         }

2732         tvp = nfs3_fhtovp(&args->to.dir, exi);
2733         if (tvp == NULL) {
2734                 error = ESTALE;
2735                 goto err;
2736         }

2738         tbva.va_mask = AT_ALL;
2739         tbvap = VOP_GETATTR(tvp, &tbva, 0, cr, NULL) ? NULL : &tbva;
2740         tavap = tbvap;

2742         if (fvp->v_type != VDIR || tvp->v_type != VDIR) {
2743                 resp->status = NFS3ERR_NOTDIR;
2744                 goto err1;
2745         }

2747         if (args->from.name == nfs3nametoolong ||
2748             args->to.name == nfs3nametoolong) {
2749                 resp->status = NFS3ERR_NAMETOOLONG;
2750                 goto err1;
2751         }
2752         if (args->from.name == NULL || *(args->from.name) == '\0' ||
2753             args->to.name == NULL || *(args->to.name) == '\0') {
2754                 resp->status = NFS3ERR_ACCES;
2755                 goto err1;
2756         }

2758         if (rdonly(ro, tvp)) {
2759                 resp->status = NFS3ERR_ROFS;
2760                 goto err1;
2761         }

2763         if (is_system_labeled()) {
2764                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
2765                         if (!do_rfs_label_check(clabel, tvp, EQUALITY_CHECK,
2766                             exi)) {
2767                                 resp->status = NFS3ERR_ACCES;
2768                                 goto err1;
2769                         }
2770                 }
2771         }

2773         ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
2774         name = nfscmd_convname(ca, exi, args->from.name,
2775             NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);

2777         if (name == NULL) {
2778                 resp->status = NFS3ERR_INVAL;
2779                 goto err1;
2780         }

2782         toname = nfscmd_convname(ca, exi, args->to.name,
2783             NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);
```

```
2785         if (toname == NULL) {
2786                 resp->status = NFS3ERR_INVAL;
2787                 goto err1;
2788         }

2790         /*
2791          * Check for a conflict with a non-blocking mandatory share
2792          * reservation or V4 delegations.
2793          */
2794         error = VOP_LOOKUP(fvp, name, &srcvp, NULL, 0,
2795             NULL, cr, NULL, NULL, NULL);
2796         if (error != 0)
2797                 goto err;

2799         /*
2800          * If we rename a delegated file we should recall the
2801          * delegation, since future opens should fail or would
2802          * refer to a new file.
2803          */
2804         if (rfs4_check_delegated(FWRITE, srcvp, FALSE)) {
2805                 resp->status = NFS3ERR_JUKEBOX;
2806                 goto err1;
2807         }

2809         /*
2810          * Check for renaming over a delegated file.  Check rfs4_deleg_policy
2811          * first to avoid VOP_LOOKUP if possible.
2812          */
2813         if (rfs4_deleg_policy != SRV_NEVER_DELEGATE &&
2814             VOP_LOOKUP(tvp, toname, &targvp, NULL, 0, NULL, cr,
2815             NULL, NULL, NULL) == 0) {

2817                 if (rfs4_check_delegated(FWRITE, targvp, TRUE)) {
2818                         VN_RELE(targvp);
2819                         resp->status = NFS3ERR_JUKEBOX;
2820                         goto err1;
2821                 }
2822                 VN_RELE(targvp);
2823         }

2825         if (!nbl_need_check(srcvp)) {
2826                 error = VOP_RENAME(fvp, name, tvp, toname, cr, NULL, 0);
2827         } else {
2828                 nbl_start_crit(srcvp, RW_READER);
2829                 if (nbl_conflict(srcvp, NBL_RENAME, 0, 0, 0, NULL))
2830                         error = EACCES;
2831                 else
2832                         error = VOP_RENAME(fvp, name, tvp, toname, cr, NULL, 0);
2833                 nbl_end_crit(srcvp);
2834         }
2835         if (error == 0)
2836                 vn_renamepath(tvp, srcvp, args->to.name,
2837                     strlen(args->to.name));
2838         VN_RELE(srcvp);
2839         srcvp = NULL;

2841         fava.va_mask = AT_ALL;
2842         favap = VOP_GETATTR(fvp, &fava, 0, cr, NULL) ? NULL : &fava;
2843         tava.va_mask = AT_ALL;
2844         tavap = VOP_GETATTR(tvp, &tava, 0, cr, NULL) ? NULL : &tava;

2846         /*
2847          * Force modified data and metadata out to stable storage.
2848          */
2849         (void) VOP_FSYNC(fvp, 0, cr, NULL);
```



```
2850         (void) VOP_FSYNC(tvp, 0, cr, NULL);

2852         if (error)
2853                 goto err;

2855         resp->status = NFS3_OK;
2856         vattr_to_wcc_data(fbvap, favap, &resp->resok.fromdir_wcc);
2857         vattr_to_wcc_data(tbvap, tavap, &resp->resok.todir_wcc);
2858         goto out;

2860 err:
2861         if (curthread->t_flag & T_WOULDBLOCK) {
2862                 curthread->t_flag &= ~T_WOULDBLOCK;
2863                 resp->status = NFS3ERR_JUKEBOX;
2864         } else {
2865                 resp->status = puterrno3(error);
2866         }
2867 err1:
2868         vattr_to_wcc_data(fbvap, favap, &resp->resfail.fromdir_wcc);
2869         vattr_to_wcc_data(tbvap, tavap, &resp->resfail.todir_wcc);

2871 out:
2872         if (name != NULL && name != args->from.name)
2873                 kmem_free(name, MAXPATHLEN + 1);
2874         if (toname != NULL && toname != args->to.name)
2875                 kmem_free(toname, MAXPATHLEN + 1);

2877         DTRACE_NFSV3_4(op__rename__done, struct svc_req *, req,
2878             cred_t *, cr, vnode_t *, fvp, RENAME3res *, resp);
2879         if (fvp != NULL)
2880                 VN_RELE(fvp);
2881         if (tvp != NULL)
2882                 VN_RELE(tvp);
2883 }
______unchanged_portion_omitted_

2892 void
2893 rfs3_link(LINK3args *args, LINK3res *resp, struct exportinfo *exi,
2894     struct svc_req *req, cred_t *cr, bool_t ro)
2895 {
2896         int error;
2897         vnode_t *vp;
2898         vnode_t *dvp;
2899         struct vattr *vap;
2900         struct vattr va;
2901         struct vattr *bvap;
2902         struct vattr bva;
2903         struct vattr *avap;
2904         struct vattr ava;
2905         nfs_fh3 *fh3;
2906         struct exportinfo *to_exi;
2907         bslabel_t *clabel;
2908         struct sockaddr *ca;
2909         char *name = NULL;

2911         vap = NULL;
2912         bvap = NULL;
2913         avap = NULL;
2914         dvp = NULL;

2916         vp = nfs3_fhtovp(&args->file, exi);

2918         DTRACE_NFSV3_4(op__link__start, struct svc_req *, req,
2919             cred_t *, cr, vnode_t *, vp, LINK3args *, args);

2921         if (vp == NULL) {
```

```
2922                 error = ESTALE;
2923                 goto out;
2924         }

2926         va.va_mask = AT_ALL;
2927         vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;

2929         fh3 = &args->link.dir;
2930         to_exi = checkexport(&fh3->fh3_fsid, FH3TOXFIDP(fh3), NULL);
2930         to_exi = checkexport(&fh3->fh3_fsid, FH3TOXFIDP(fh3));
2931         if (to_exi == NULL) {
2932                 resp->status = NFS3ERR_ACCES;
2933                 goto out1;
2934         }
2935         exi_rele(to_exi);

2937         if (to_exi != exi) {
2938                 resp->status = NFS3ERR_XDEV;
2939                 goto out1;
2940         }

2942         if (is_system_labeled()) {
2943                 clabel = req->rq_label;

2945                 ASSERT(clabel != NULL);
2946                 DTRACE_PROBE2(tx__rfs3__log__info__oplink__clabel, char *,
2947                     "got client label from request(1)", struct svc_req *, req);

2949                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
2950                         if (!do_rfs_label_check(clabel, vp, DOMINANCE_CHECK,
2951                             exi)) {
2952                                 resp->status = NFS3ERR_ACCES;
2953                                 goto out1;
2954                         }
2955                 }
2956         }

2958         dvp = nfs3_fhtovp(&args->link.dir, exi);
2959         if (dvp == NULL) {
2960                 error = ESTALE;
2961                 goto out;
2962         }

2964         bva.va_mask = AT_ALL;
2965         bvap = VOP_GETATTR(dvp, &bva, 0, cr, NULL) ? NULL : &bva;

2967         if (dvp->v_type != VDIR) {
2968                 resp->status = NFS3ERR_NOTDIR;
2969                 goto out1;
2970         }

2972         if (args->link.name == nfs3nametoolong) {
2973                 resp->status = NFS3ERR_NAMETOOLONG;
2974                 goto out1;
2975         }

2977         if (args->link.name == NULL || *(args->link.name) == '\0') {
2978                 resp->status = NFS3ERR_ACCES;
2979                 goto out1;
2980         }

2982         if (rdonly(ro, dvp)) {
2983                 resp->status = NFS3ERR_ROFS;
2984                 goto out1;
2985         }
```



```
2987         if (is_system_labeled()) {
2988                 DTRACE_PROBE2(tx__rfs3__log__info__oplinkdir__clabel, char *,
2989                     "got client label from request(1)", struct svc_req *, req);

2991                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
2992                         if (!do_rfs_label_check(clabel, dvp, EQUALITY_CHECK,
2993                             exi)) {
2994                                 resp->status = NFS3ERR_ACCES;
2995                                 goto out1;
2996                         }
2997                 }
2998         }

3000         ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
3001         name = nfscmd_convname(ca, exi, args->link.name,
3002             NFSCMD_CONV_INBOUND, MAXPATHLEN + 1);

3004         if (name == NULL) {
3005                 resp->status = NFS3ERR_SERVERFAULT;
3006                 goto out1;
3007         }

3009         error = VOP_LINK(dvp, vp, name, cr, NULL, 0);

3011         va.va_mask = AT_ALL;
3012         vap = VOP_GETATTR(vp, &va, 0, cr, NULL) ? NULL : &va;
3013         ava.va_mask = AT_ALL;
3014         avap = VOP_GETATTR(dvp, &ava, 0, cr, NULL) ? NULL : &ava;

3016         /*
3017          * Force modified data and metadata out to stable storage.
3018          */
3019         (void) VOP_FSYNC(vp, FNODSYNC, cr, NULL);
3020         (void) VOP_FSYNC(dvp, 0, cr, NULL);

3022         if (error)
3023                 goto out;

3025         VN_RELE(dvp);

3027         resp->status = NFS3_OK;
3028         vattr_to_post_op_attr(vap, &resp->resok.file_attributes);
3029         vattr_to_wcc_data(bvap, avap, &resp->resok.linkdir_wcc);

3031         DTRACE_NFSV3_4(op__link__done, struct svc_req *, req,
3032             cred_t *, cr, vnode_t *, vp, LINK3res *, resp);

3034         VN_RELE(vp);

3036         return;

3038 out:
3039         if (curthread->t_flag & T_WOULDBLOCK) {
3040                 curthread->t_flag &= ~T_WOULDBLOCK;
3041                 resp->status = NFS3ERR_JUKEBOX;
3042         } else
3043                 resp->status = puterrno3(error);
3044 out1:
3045         if (name != NULL && name != args->link.name)
3046                 kmem_free(name, MAXPATHLEN + 1);

3048         DTRACE_NFSV3_4(op__link__done, struct svc_req *, req,
3049             cred_t *, cr, vnode_t *, vp, LINK3res *, resp);

3051         if (vp != NULL)
3052                 VN_RELE(vp);
```

```
3053         if (dvp != NULL)
3054                 VN_RELE(dvp);
3055         vattr_to_post_op_attr(vap, &resp->resfail.file_attributes);
3056         vattr_to_wcc_data(bvap, avap, &resp->resfail.linkdir_wcc);
3057 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  251102 Wed Sep 14 16:21:02 2016
new/usr/src/uts/common/fs/nfs/nfs4_srv.c
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_

 864 /*
 865  * Used by rfs4_op_secinfo to get the security information from the
 866  * export structure associated with the component.
 867  */
 868 /* ARGSUSED */
 869 static nfsstat4
 870 do_rfs4_op_secinfo(struct compound_state *cs, char *nm, SECINFO4res *resp)
 871 {
 872         int error, different_export = 0;
 873         vnode_t *dvp, *vp;
 874         struct exportinfo *exi = NULL;
 875         struct exportinfo *oexi = NULL;
 876 #endif /* ! codereview */
 877         fid_t fid;
 878         uint_t count, i;
 879         secinfo4 *resok_val;
 880         struct secinfo *secp;
 881         seconfig_t *si;
 882         bool_t did_traverse = FALSE;
 883         int dotdot, walk;

 885         dvp = cs->vp;
 886         dotdot = (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0');

 888         /*
 889          * If dotdotting, then need to check whether it's above the
 890          * root of a filesystem, or above an export point.
 891          */
 892         if (dotdot) {

 894                 /*
 895                  * If dotdotting at the root of a filesystem, then
 896                  * need to traverse back to the mounted-on filesystem
 897                  * and do the dotdot lookup there.
 898                  */
 899                 if (cs->vp->v_flag & VROOT) {

 901                         /*
 902                          * If at the system root, then can
 903                          * go up no further.
 904                          */
 905                         if (VN_CMP(dvp, rootdir))
 906                                 return (puterrno4(ENOENT));

 908                         /*
 909                          * Traverse back to the mounted-on filesystem
 910                          */
 911                         dvp = untraverse(cs->vp);

 913                         /*
 914                          * Set the different_export flag so we remember
 915                          * to pick up a new exportinfo entry for
 916                          * this new filesystem.
 917                          */
 918                         different_export = 1;
 919                 } else {

 921                         /*
 922                          * If dotdotting above an export point then set
```



```
 923                          * the different_export to get new export info.
 924                          */
 925                         different_export = nfs_exported(cs->exi, cs->vp);
 926                 }
 927         }

 929         /*
 930          * Get the vnode for the component "nm".
 931          */
 932         error = VOP_LOOKUP(dvp, nm, &vp, NULL, 0, NULL, cs->cr,
 933             NULL, NULL, NULL);
 934         if (error)
 935                 return (puterrno4(error));

 937         /*
 938          * If the vnode is in a pseudo filesystem, or if the security flavor
 939          * used in the request is valid but not an explicitly shared flavor,
 940          * or the access bit indicates that this is a limited access,
 941          * check whether this vnode is visible.
 942          */
 943         if (!different_export &&
 944             (PSEUDO(cs->exi) || ! is_exported_sec(cs->nfsflavor, cs->exi) ||
 945             cs->access & CS_ACCESS_LIMITED)) {
 946                 if (! nfs_visible(cs->exi, vp, &different_export)) {
 947                         VN_RELE(vp);
 948                         return (puterrno4(ENOENT));
 949                 }
 950         }

 952         /*
 953          * If it's a mountpoint, then traverse it.
 954          */
 955         if (vn_ismntpt(vp)) {
 956                 if ((error = traverse(&vp)) != 0) {
 957                         VN_RELE(vp);
 958                         return (puterrno4(error));
 959                 }
 960                 /* remember that we had to traverse mountpoint */
 961                 did_traverse = TRUE;
 962                 different_export = 1;
 963         } else if (vp->v_vfsp != dvp->v_vfsp) {
 964                 /*
 965                  * If vp isn't a mountpoint and the vfs ptrs aren't the same,
 966                  * then vp is probably an LOFS object.  We don't need the
 967                  * realvp, we just need to know that we might have crossed
 968                  * a server fs boundary and need to call checkexport.
 875                  * a server fs boundary and need to call checkexport4.
 969                  * (LOFS lookup hides server fs mountpoints, and actually calls
 970                  * traverse)
 971                  */
 972                 different_export = 1;
 973         }

 975         /*
 976          * Get the export information for it.
 977          */
 978         if (different_export) {

 980                 bzero(&fid, sizeof (fid));
 981                 fid.fid_len = MAXFIDSZ;
 982                 error = vop_fid_pseudo(vp, &fid);
 983                 if (error) {
 984                         VN_RELE(vp);
 985                         return (puterrno4(error));
 986                 }
```

```
 988                 if (dotdot)
 989                         oexi = nfs_vptoexi(NULL, vp, cs->cr, &walk, NULL, TRUE);
 896                         exi = nfs_vptoexi(NULL, vp, cs->cr, &walk, NULL, TRUE);
 990                 else
 991                         oexi = checkexport(&vp->v_vfsp->vfs_fsid, &fid, vp);
 898                         exi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);

 993                 if (oexi == NULL) {
 900                 if (exi == NULL) {
 994                         if (did_traverse == TRUE) {
 995                                 /*
 996                                  * If this vnode is a mounted-on vnode,
 997                                  * but the mounted-on file system is not
 998                                  * exported, send back the secinfo for
 999                                  * the exported node that the mounted-on
1000                                  * vnode lives in.
1001                                  */
1002                                 exi = cs->exi;
1003                         } else {
1004                                 VN_RELE(vp);
1005                                 return (puterrno4(EACCES));
1006                         }
1007                 } else {
1008                         exi = oexi;
1009 #endif /* ! codereview */
1010                 }
1011         } else {
1012                 exi = cs->exi;
1013         }
1014         ASSERT(exi != NULL);


1017         /*
1018          * Create the secinfo result based on the security information
1019          * from the exportinfo structure (exi).
1020          *
1021          * Return all flavors for a pseudo node.
1022          * For a real export node, return the flavor that the client
1023          * has access with.
1024          */
1025         rw_enter(&exported_lock, RW_READER);
 914         ASSERT(RW_LOCK_HELD(&exported_lock));
1026         if (PSEUDO(exi)) {
1027                 count = exi->exi_export.ex_seccnt; /* total sec count */
1028                 resok_val = kmem_alloc(count * sizeof (secinfo4), KM_SLEEP);
1029                 secp = exi->exi_export.ex_secinfo;

1031                 for (i = 0; i < count; i++) {
1032                         si = &secp[i].s_secinfo;
1033                         resok_val[i].flavor = si->sc_rpcnum;
1034                         if (resok_val[i].flavor == RPCSEC_GSS) {
1035                                 rpcsec_gss_info *info;

1037                                 info = &resok_val[i].flavor_info;
1038                                 info->qop = si->sc_qop;
1039                                 info->service = (rpc_gss_svc_t)si->sc_service;

1041                                 /* get oid opaque data */
1042                                 info->oid.sec_oid4_len =
1043                                     si->sc_gss_mech_type->length;
1044                                 info->oid.sec_oid4_val = kmem_alloc(
1045                                     si->sc_gss_mech_type->length, KM_SLEEP);
1046                                 bcopy(
1047                                     si->sc_gss_mech_type->elements,
1048                                     info->oid.sec_oid4_val,
1049                                     info->oid.sec_oid4_len);
```



```
1050                         }
1051                 }
1052                 resp->SECINFO4resok_len = count;
1053                 resp->SECINFO4resok_val = resok_val;
1054         } else {
1055                 int ret_cnt = 0, k = 0;
1056                 int *flavor_list;

1058                 count = exi->exi_export.ex_seccnt; /* total sec count */
1059                 secp = exi->exi_export.ex_secinfo;

1061                 flavor_list = kmem_alloc(count * sizeof (int), KM_SLEEP);
1062                 /* find out which flavors to return */
1063                 for (i = 0; i < count; i ++) {
1064                         int access, flavor, perm;

1066                         flavor = secp[i].s_secinfo.sc_nfsnum;
1067                         perm = secp[i].s_flags;

1069                         access = nfsauth4_secinfo_access(exi, cs->req,
1070                             flavor, perm, cs->basecr);

1072                         if (! (access & NFSAUTH_DENIED) &&
1073                             ! (access & NFSAUTH_WRONGSEC)) {
1074                                 flavor_list[ret_cnt] = flavor;
1075                                 ret_cnt++;
1076                         }
1077                 }

1079                 /* Create the returning SECINFO value */
1080                 resok_val = kmem_alloc(ret_cnt * sizeof (secinfo4), KM_SLEEP);

1082                 for (i = 0; i < count; i++) {
1083                         /*
1084                          * If the flavor is in the flavor list,
1085                          * fill in resok_val.
1086                          */
1087                         si = &secp[i].s_secinfo;
1088                         if (in_flavor_list(si->sc_nfsnum,
1089                             flavor_list, ret_cnt)) {
1090                                 resok_val[k].flavor = si->sc_rpcnum;
1091                                 if (resok_val[k].flavor == RPCSEC_GSS) {
1092                                         rpcsec_gss_info *info;

1094                                         info = &resok_val[k].flavor_info;
1095                                         info->qop = si->sc_qop;
1096                                         info->service = (rpc_gss_svc_t)
1097                                             si->sc_service;

1099                                         /* get oid opaque data */
1100                                         info->oid.sec_oid4_len =
1101                                             si->sc_gss_mech_type->length;
1102                                         info->oid.sec_oid4_val = kmem_alloc(
1103                                             si->sc_gss_mech_type->length,
1104                                             KM_SLEEP);
1105                                         bcopy(si->sc_gss_mech_type->elements,
1106                                             info->oid.sec_oid4_val,
1107                                             info->oid.sec_oid4_len);
1108                                 }
1109                                 k++;
1110                         }
1111                         if (k >= ret_cnt)
1112                                 break;
1113                 }
1114                 resp->SECINFO4resok_len = ret_cnt;
1115                 resp->SECINFO4resok_val = resok_val;
```

```
1116                    kmem_free(flavor_list, count * sizeof (int));
1117            }
1118            rw_exit(&exported_lock);
1119            if (oexi)
1120                    exi_rele(oexi);

1121            VN_RELE(vp);
1122            return (NFS4_OK);
1123    }
_____unchanged_portion_omitted_

2610    /*
2611     * Used by rfs4_op_lookup and rfs4_op_lookupp to do the actual work.
2612     */

2614    /* ARGSUSED */
2615    static nfsstat4
2616    do_rfs4_op_lookup(char *nm, struct svc_req *req, struct compound_state *cs)
2617    {
2618            int error;
2619            int different_export = 0;
2620            vnode_t *vp, *pre_tvp = NULL, *oldvp = NULL;
2621            struct exportinfo *exi = NULL, *pre_exi = NULL, *oexi = NULL;
2508            struct exportinfo *exi = NULL, *pre_exi = NULL;
2622            nfsstat4 stat;
2623            fid_t fid;
2624            int attrdir, dotdot, walk;
2625            bool_t is_newvp = FALSE;

2627            if (cs->vp->v_flag & V_XATTRDIR) {
2628                    attrdir = 1;
2629                    ASSERT(get_fh4_flag(&cs->fh, FH4_ATTRDIR));
2630            } else {
2631                    attrdir = 0;
2632                    ASSERT(! get_fh4_flag(&cs->fh, FH4_ATTRDIR));
2633            }

2635            dotdot = (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0');

2637            /*
2638             * If dotdotting, then need to check whether it's
2639             * above the root of a filesystem, or above an
2640             * export point.
2641             */
2642            if (dotdot) {

2644                    /*
2645                     * If dotdotting at the root of a filesystem, then
2646                     * need to traverse back to the mounted-on filesystem
2647                     * and do the dotdot lookup there.
2648                     */
2649                    if (cs->vp->v_flag & VROOT) {

2651                            /*
2652                             * If at the system root, then can
2653                             * go up no further.
2654                             */
2655                            if (VN_CMP(cs->vp, rootdir))
2656                                    return (puterrno4(ENOENT));

2658                            /*
2659                             * Traverse back to the mounted-on filesystem
2660                             */
2661                            cs->vp = untraverse(cs->vp);

2663                            /*
```



```
2664                             * Set the different_export flag so we remember
2665                             * to pick up a new exportinfo entry for
2666                             * this new filesystem.
2667                             */
2668                            different_export = 1;
2669                    } else {

2671                            /*
2672                             * If dotdotting above an export point then set
2673                             * the different_export to get new export info.
2674                             */
2675                            different_export = nfs_exported(cs->exi, cs->vp);
2676                    }
2677            }

2679            error = VOP_LOOKUP(cs->vp, nm, &vp, NULL, 0, NULL, cs->cr,
2680                NULL, NULL, NULL);
2681            if (error)
2682                    return (puterrno4(error));

2684            /*
2685             * If the vnode is in a pseudo filesystem, check whether it is visible.
2686             *
2687             * XXX if the vnode is a symlink and it is not visible in
2688             * a pseudo filesystem, return ENOENT (not following symlink).
2689             * V4 client can not mount such symlink. This is a regression
2690             * from V2/V3.
2691             *
2692             * In the same exported filesystem, if the security flavor used
2693             * is not an explicitly shared flavor, limit the view to the visible
2694             * list entries only. This is not a WRONGSEC case because it's already
2695             * checked via PUTROOTFH/PUTPUBFH or PUTFH.
2696             */
2697            if (!different_export &&
2698                (PSEUDO(cs->exi) || ! is_exported_sec(cs->nfsflavor, cs->exi) ||
2699                cs->access & CS_ACCESS_LIMITED)) {
2700                    if (! nfs_visible(cs->exi, vp, &different_export)) {
2701                            VN_RELE(vp);
2702                            return (puterrno4(ENOENT));
2703                    }
2704            }

2706            /*
2707             * If it's a mountpoint, then traverse it.
2708             */
2709            if (vn_ismntpt(vp)) {
2710                    pre_exi = cs->exi;      /* save pre-traversed exportinfo */
2711                    pre_tvp = vp;           /* save pre-traversed vnode     */

2713                    /*
2714                     * hold pre_tvp to counteract rele by traverse.  We will
2715                     * need pre_tvp below if checkexport fails
2602                     * need pre_tvp below if checkexport4 fails
2716                     */
2717                    VN_HOLD(pre_tvp);
2718                    if ((error = traverse(&vp)) != 0) {
2719                            VN_RELE(vp);
2720                            VN_RELE(pre_tvp);
2721                            return (puterrno4(error));
2722                    }
2723                    different_export = 1;
2724            } else if (vp->v_vfsp != cs->vp->v_vfsp) {
2725                    /*
2726                     * The vfsp comparison is to handle the case where
2727                     * a LOFS mount is shared.  lo_lookup traverses mount points,
2728                     * and NFS is unaware of local fs transistions because
```

```
2729                 * v_vfsmountedhere isn't set.  For this special LOFS case,
2730                 * the dir and the obj returned by lookup will have different
2731                 * vfs ptrs.
2732                 */
2733                different_export = 1;
2734        }

2736        if (different_export) {

2738                bzero(&fid, sizeof (fid));
2739                fid.fid_len = MAXFIDSZ;
2740                error = vop_fid_pseudo(vp, &fid);
2741                if (error) {
2742                        VN_RELE(vp);
2743                        if (pre_tvp)
2744                                VN_RELE(pre_tvp);
2745                        return (puterrno4(error));
2746                }

2748                if (dotdot)
2749                        exi = nfs_vptoexi(NULL, vp, cs->cr, &walk, NULL, TRUE);
2750                else
2751                        exi = checkexport(&vp->v_vfsp->vfs_fsid, &fid, vp);
2638                        exi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);

2753                if (exi == NULL) {
2754                        if (pre_tvp) {
2755                                /*
2756                                 * If this vnode is a mounted-on vnode,
2757                                 * but the mounted-on file system is not
2758                                 * exported, send back the filehandle for
2759                                 * the mounted-on vnode, not the root of
2760                                 * the mounted-on file system.
2761                                 */
2762                                VN_RELE(vp);
2763                                vp = pre_tvp;
2764                                exi = pre_exi;
2765                                if (exi)
2766                                        exi_hold(exi);
2767 #endif /* ! codereview */
2768                        } else {
2769                                VN_RELE(vp);
2770                                return (puterrno4(EACCES));
2771                        }
2772                } else if (pre_tvp) {
2773                        /* we're done with pre_tvp now. release extra hold */
2774                        VN_RELE(pre_tvp);
2775                }

2777                if (cs->exi)
2778                        exi_rele(cs->exi);
2779 #endif /* ! codereview */
2780                cs->exi = exi;

2782                /*
2783                 * Now we do a checkauth4. The reason is that
2784                 * this client/user may not have access to the new
2785                 * exported file system, and if he does,
2786                 * the client/user may be mapped to a different uid.
2787                 *
2788                 * We start with a new cr, because the checkauth4 done
2789                 * in the PUT*FH operation over wrote the cred's uid,
2790                 * gid, etc, and we want the real thing before calling
2791                 * checkauth4()
2792                 */
2793                crfree(cs->cr);
```



```
2794                cs->cr = crdup(cs->basecr);

2796                oldvp = cs->vp;
2797                cs->vp = vp;
2798                is_newvp = TRUE;

2800                stat = call_checkauth4(cs, req);
2801                if (stat != NFS4_OK) {
2802                        VN_RELE(cs->vp);
2803                        cs->vp = oldvp;
2804                        return (stat);
2805                }
2806        }

2808        /*
2809         * After various NFS checks, do a label check on the path
2810         * component. The label on this path should either be the
2811         * global zone's label or a zone's label. We are only
2812         * interested in the zone's label because exported files
2813         * in global zone is accessible (though read-only) to
2814         * clients. The exportability/visibility check is already
2815         * done before reaching this code.
2816         */
2817        if (is_system_labeled()) {
2818                bslabel_t *clabel;

2820                ASSERT(req->rq_label != NULL);
2821                clabel = req->rq_label;
2822                DTRACE_PROBE2(tx__rfs4__log__info__oplookup__clabel, char *,
2823                    "got client label from request(1)", struct svc_req *, req);

2825                if (!blequal(&l_admin_low->tsl_label, clabel)) {
2826                        if (!do_rfs_label_check(clabel, vp, DOMINANCE_CHECK,
2827                            cs->exi)) {
2828                                error = EACCES;
2829                                goto err_out;
2830                        }
2831                } else {
2832                        /*
2833                         * We grant access to admin_low label clients
2834                         * only if the client is trusted, i.e. also
2835                         * running Solaris Trusted Extension.
2836                         */
2837                        struct sockaddr *ca;
2838                        int             addr_type;
2839                        void            *ipaddr;
2840                        tsol_tpc_t      *tp;

2842                        ca = (struct sockaddr *)svc_getrpccaller(
2843                            req->rq_xprt)->buf;
2844                        if (ca->sa_family == AF_INET) {
2845                                addr_type = IPV4_VERSION;
2846                                ipaddr = &((struct sockaddr_in *)ca)->sin_addr;
2847                        } else if (ca->sa_family == AF_INET6) {
2848                                addr_type = IPV6_VERSION;
2849                                ipaddr = &((struct sockaddr_in6 *)
2850                                    ca)->sin6_addr;
2851                        }
2852                        tp = find_tpc(ipaddr, addr_type, B_FALSE);
2853                        if (tp == NULL || tp->tpc_tp.tp_doi !=
2854                            l_admin_low->tsl_doi || tp->tpc_tp.host_type !=
2855                            SUN_CIPSO) {
2856                                if (tp != NULL)
2857                                        TPC_RELE(tp);
2858                                error = EACCES;
2859                                goto err_out;
```

```
2860                        }
2861                        TPC_RELE(tp);
2862                }
2863        }

2865        error = makefh4(&cs->fh, vp, cs->exi);

2867 err_out:
2868        if (error) {
2869                if (is_newvp) {
2870                        VN_RELE(cs->vp);
2871                        cs->vp = oldvp;
2872                } else
2873                        VN_RELE(vp);
2874                return (puterrno4(error));
2875        }

2877        if (!is_newvp) {
2878                if (cs->vp)
2879                        VN_RELE(cs->vp);
2880                cs->vp = vp;
2881        } else if (oldvp)
2882                VN_RELE(oldvp);

2884        /*
2885         * if did lookup on attrdir and didn't lookup .., set named
2886         * attr fh flag
2887         */
2888        if (attrdir && ! dotdot)
2889                set_fh4_flag(&cs->fh, FH4_NAMEDATTR);

2891        /* Assume false for now, open proc will set this */
2892        cs->mandlock = FALSE;

2894        return (NFS4_OK);
2895 }

2897 /* ARGSUSED */
2898 static void
2899 rfs4_op_lookup(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
2900     struct compound_state *cs)
2901 {
2902        LOOKUP4args *args = &argop->nfs_argop4_u.oplookup;
2903        LOOKUP4res *resp = &resop->nfs_resop4_u.oplookup;
2904        char *nm;
2905        uint_t len;
2906        struct sockaddr *ca;
2907        char *name = NULL;
2908        nfsstat4 status;

2910        DTRACE_NFSV4_2(op__lookup__start, struct compound_state *, cs,
2911            LOOKUP4args *, args);

2913        if (cs->vp == NULL) {
2914                *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
2915                goto out;
2916        }

2918        if (cs->vp->v_type == VLNK) {
2919                *cs->statusp = resp->status = NFS4ERR_SYMLINK;
2920                goto out;
2921        }

2923        if (cs->vp->v_type != VDIR) {
2924                *cs->statusp = resp->status = NFS4ERR_NOTDIR;
2925                goto out;
```



```
2926        }

2928        status = utf8_dir_verify(&args->objname);
2929        if (status != NFS4_OK) {
2930                *cs->statusp = resp->status = status;
2931                goto out;
2932        }

2934        nm = utf8_to_str(&args->objname, &len, NULL);
2935        if (nm == NULL) {
2936                *cs->statusp = resp->status = NFS4ERR_INVAL;
2937                goto out;
2938        }

2940        if (len > MAXNAMELEN) {
2941                *cs->statusp = resp->status = NFS4ERR_NAMETOOLONG;
2942                kmem_free(nm, len);
2943                goto out;
2944        }

2946        ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
2947        name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND,
2948            MAXPATHLEN  + 1);

2950        if (name == NULL) {
2951                *cs->statusp = resp->status = NFS4ERR_INVAL;
2952                kmem_free(nm, len);
2953                goto out;
2954        }

2956        *cs->statusp = resp->status = do_rfs4_op_lookup(name, req, cs);

2958        if (name != nm)
2959                kmem_free(name, MAXPATHLEN + 1);
2960        kmem_free(nm, len);

2962 out:
2963        DTRACE_NFSV4_2(op__lookup__done, struct compound_state *, cs,
2964            LOOKUP4res *, resp);
2965 }

2967 /* ARGSUSED */
2968 static void
2969 rfs4_op_lookupp(nfs_argop4 *args, nfs_resop4 *resop, struct svc_req *req,
2970     struct compound_state *cs)
2971 {
2972        LOOKUPP4res *resp = &resop->nfs_resop4_u.oplookupp;

2974        DTRACE_NFSV4_1(op__lookupp__start, struct compound_state *, cs);

2976        if (cs->vp == NULL) {
2977                *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
2978                goto out;
2979        }

2981        if (cs->vp->v_type != VDIR) {
2982                *cs->statusp = resp->status = NFS4ERR_NOTDIR;
2983                goto out;
2984        }

2986        *cs->statusp = resp->status = do_rfs4_op_lookup("..", req, cs);

2988        /*
2989         * From NFSV4 Specification, LOOKUPP should not check for
2990         * NFS4ERR_WRONGSEC. Retrun NFS4_OK instead.
2991         */
```

```
2992         if (resp->status == NFS4ERR_WRONGSEC) {
2993                 *cs->statusp = resp->status = NFS4_OK;
2994         }

2996 out:
2997         DTRACE_NFSV4_2(op__lookupp__done, struct compound_state *, cs,
2998             LOOKUPP4res *, resp);
2999 }


3002 /*ARGSUSED2*/
3003 static void
3004 rfs4_op_openattr(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
3005     struct compound_state *cs)
3006 {
3007         OPENATTR4args   *args = &argop->nfs_argop4_u.opopenattr;
3008         OPENATTR4res    *resp = &resop->nfs_resop4_u.opopenattr;
3009         vnode_t         *avp = NULL;
3010         int             lookup_flags = LOOKUP_XATTR, error;
3011         int             exp_ro = 0;

3013         DTRACE_NFSV4_2(op__openattr__start, struct compound_state *, cs,
3014             OPENATTR4args *, args);

3016         if (cs->vp == NULL) {
3017                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
3018                 goto out;
3019         }

3021         if ((cs->vp->v_vfsp->vfs_flag & VFS_XATTR) == 0 &&
3022             !vfs_has_feature(cs->vp->v_vfsp, VFSFT_SYSATTR_VIEWS)) {
3023                 *cs->statusp = resp->status = puterrno4(ENOTSUP);
3024                 goto out;
3025         }

3027         /*
3028          * If file system supports passing ACE mask to VOP_ACCESS then
3029          * check for ACE_READ_NAMED_ATTRS, otherwise do legacy checks
3030          */

3032         if (vfs_has_feature(cs->vp->v_vfsp, VFSFT_ACEMASKONACCESS))
3033                 error = VOP_ACCESS(cs->vp, ACE_READ_NAMED_ATTRS,
3034                     V_ACE_MASK, cs->cr, NULL);
3035         else
3036                 error = ((VOP_ACCESS(cs->vp, VREAD, 0, cs->cr, NULL) != 0) &&
3037                     (VOP_ACCESS(cs->vp, VWRITE, 0, cs->cr, NULL) != 0) &&
3038                     (VOP_ACCESS(cs->vp, VEXEC, 0, cs->cr, NULL) != 0));

3040         if (error) {
3041                 *cs->statusp = resp->status = puterrno4(EACCES);
3042                 goto out;
3043         }

3045         /*
3046          * The CREATE_XATTR_DIR VOP flag cannot be specified if
3047          * the file system is exported read-only -- regardless of
3048          * createdir flag.  Otherwise the attrdir would be created
3049          * (assuming server fs isn't mounted readonly locally).  If
3050          * VOP_LOOKUP returns ENOENT in this case, the error will
3051          * be translated into EROFS.  ENOSYS is mapped to ENOTSUP
3052          * because specfs has no VOP_LOOKUP op, so the macro would
3053          * return ENOSYS.  EINVAL is returned by all (current)
3054          * Solaris file system implementations when any of their
3055          * restrictions are violated (xattr(dir) can't have xattrdir).
3056          * Returning NOTSUPP is more appropriate in this case
3057          * because the object will never be able to have an attrdir.
```



```
3058          */
3059         if (args->createdir && ! (exp_ro = rdonly4(req, cs)))
3060                 lookup_flags |= CREATE_XATTR_DIR;

3062         error = VOP_LOOKUP(cs->vp, "", &avp, NULL, lookup_flags, NULL, cs->cr,
3063             NULL, NULL, NULL);

3065         if (error) {
3066                 if (error == ENOENT && args->createdir && exp_ro)
3067                         *cs->statusp = resp->status = puterrno4(EROFS);
3068                 else if (error == EINVAL || error == ENOSYS)
3069                         *cs->statusp = resp->status = puterrno4(ENOTSUP);
3070                 else
3071                         *cs->statusp = resp->status = puterrno4(error);
3072                 goto out;
3073         }

3075         ASSERT(avp->v_flag & V_XATTRDIR);

3077         error = makefh4(&cs->fh, avp, cs->exi);

3079         if (error) {
3080                 VN_RELE(avp);
3081                 *cs->statusp = resp->status = puterrno4(error);
3082                 goto out;
3083         }

3085         VN_RELE(cs->vp);
3086         cs->vp = avp;

3088         /*
3089          * There is no requirement for an attrdir fh flag
3090          * because the attrdir has a vnode flag to distinguish
3091          * it from regular (non-xattr) directories.  The
3092          * FH4_ATTRDIR flag is set for future sanity checks.
3093          */
3094         set_fh4_flag(&cs->fh, FH4_ATTRDIR);
3095         *cs->statusp = resp->status = NFS4_OK;

3097 out:
3098         DTRACE_NFSV4_2(op__openattr__done, struct compound_state *, cs,
3099             OPENATTR4res *, resp);
3100 }

3102 static int
3103 do_io(int direction, vnode_t *vp, struct uio *uio, int ioflag, cred_t *cred,
3104     caller_context_t *ct)
3105 {
3106         int error;
3107         int i;
3108         clock_t delaytime;

3110         delaytime = MSEC_TO_TICK_ROUNDUP(rfs4_lock_delay);

3112         /*
3113          * Don't block on mandatory locks. If this routine returns
3114          * EAGAIN, the caller should return NFS4ERR_LOCKED.
3115          */
3116         uio->uio_fmode = FNONBLOCK;

3118         for (i = 0; i < rfs4_maxlock_tries; i++) {


3121                 if (direction == FREAD) {
3122                         (void) VOP_RWLOCK(vp, V_WRITELOCK_FALSE, ct);
3123                         error = VOP_READ(vp, uio, ioflag, cred, ct);
```

```
3124                        VOP_RWUNLOCK(vp, V_WRITELOCK_FALSE, ct);
3125                } else {
3126                        (void) VOP_RWLOCK(vp, V_WRITELOCK_TRUE, ct);
3127                        error = VOP_WRITE(vp, uio, ioflag, cred, ct);
3128                        VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, ct);
3129                }

3131                if (error != EAGAIN)
3132                        break;

3134                if (i < rfs4_maxlock_tries - 1) {
3135                        delay(delaytime);
3136                        delaytime *= 2;
3137                }
3138        }

3140        return (error);
3141 }

3143 /* ARGSUSED */
3144 static void
3145 rfs4_op_read(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
3146     struct compound_state *cs)
3147 {
3148        READ4args *args = &argop->nfs_argop4_u.opread;
3149        READ4res *resp = &resop->nfs_resop4_u.opread;
3150        int error;
3151        int verror;
3152        vnode_t *vp;
3153        struct vattr va;
3154        struct iovec iov, *iovp = NULL;
3155        int iovcnt;
3156        struct uio uio;
3157        u_offset_t offset;
3158        bool_t *deleg = &cs->deleg;
3159        nfsstat4 stat;
3160        int in_crit = 0;
3161        mblk_t *mp = NULL;
3162        int alloc_err = 0;
3163        int rdma_used = 0;
3164        int loaned_buffers;
3165        caller_context_t ct;
3166        struct uio *uiop;

3168        DTRACE_NFSV4_2(op__read__start, struct compound_state *, cs,
3169            READ4args, args);

3171        vp = cs->vp;
3172        if (vp == NULL) {
3173                *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
3174                goto out;
3175        }
3176        if (cs->access == CS_ACCESS_DENIED) {
3177                *cs->statusp = resp->status = NFS4ERR_ACCESS;
3178                goto out;
3179        }

3181        if ((stat = rfs4_check_stateid(FREAD, vp, &args->stateid, FALSE,
3182            deleg, TRUE, &ct)) != NFS4_OK) {
3183                *cs->statusp = resp->status = stat;
3184                goto out;
3185        }

3187        /*
3188         * Enter the critical region before calling VOP_RWLOCK
3189         * to avoid a deadlock with write requests.
```



```
3190         */
3191        if (nbl_need_check(vp)) {
3192                nbl_start_crit(vp, RW_READER);
3193                in_crit = 1;
3194                if (nbl_conflict(vp, NBL_READ, args->offset, args->count, 0,
3195                    &ct)) {
3196                        *cs->statusp = resp->status = NFS4ERR_LOCKED;
3197                        goto out;
3198                }
3199        }

3201        if (args->wlist) {
3202                if (args->count > clist_len(args->wlist)) {
3203                        *cs->statusp = resp->status = NFS4ERR_INVAL;
3204                        goto out;
3205                }
3206                rdma_used = 1;
3207        }

3209        /* use loaned buffers for TCP */
3210        loaned_buffers = (nfs_loaned_buffers && !rdma_used) ? 1 : 0;

3212        va.va_mask = AT_MODE|AT_SIZE|AT_UID;
3213        verror = VOP_GETATTR(vp, &va, 0, cs->cr, &ct);

3215        /*
3216         * If we can't get the attributes, then we can't do the
3217         * right access checking.  So, we'll fail the request.
3218         */
3219        if (verror) {
3220                *cs->statusp = resp->status = puterrno4(verror);
3221                goto out;
3222        }

3224        if (vp->v_type != VREG) {
3225                *cs->statusp = resp->status =
3226                    ((vp->v_type == VDIR) ? NFS4ERR_ISDIR : NFS4ERR_INVAL);
3227                goto out;
3228        }

3230        if (crgetuid(cs->cr) != va.va_uid &&
3231            (error = VOP_ACCESS(vp, VREAD, 0, cs->cr, &ct)) &&
3232            (error = VOP_ACCESS(vp, VEXEC, 0, cs->cr, &ct))) {
3233                *cs->statusp = resp->status = puterrno4(error);
3234                goto out;
3235        }

3237        if (MANDLOCK(vp, va.va_mode)) { /* XXX - V4 supports mand locking */
3238                *cs->statusp = resp->status = NFS4ERR_ACCESS;
3239                goto out;
3240        }

3242        offset = args->offset;
3243        if (offset >= va.va_size) {
3244                *cs->statusp = resp->status = NFS4_OK;
3245                resp->eof = TRUE;
3246                resp->data_len = 0;
3247                resp->data_val = NULL;
3248                resp->mblk = NULL;
3249                /* RDMA */
3250                resp->wlist = args->wlist;
3251                resp->wlist_len = resp->data_len;
3252                *cs->statusp = resp->status = NFS4_OK;
3253                if (resp->wlist)
3254                        clist_zero_len(resp->wlist);
3255                goto out;
```

```
3256         }

3258         if (args->count == 0) {
3259                 *cs->statusp = resp->status = NFS4_OK;
3260                 resp->eof = FALSE;
3261                 resp->data_len = 0;
3262                 resp->data_val = NULL;
3263                 resp->mblk = NULL;
3264                 /* RDMA */
3265                 resp->wlist = args->wlist;
3266                 resp->wlist_len = resp->data_len;
3267                 if (resp->wlist)
3268                         clist_zero_len(resp->wlist);
3269                 goto out;
3270         }

3272         /*
3273          * Do not allocate memory more than maximum allowed
3274          * transfer size
3275          */
3276         if (args->count > rfs4_tsize(req))
3277                 args->count = rfs4_tsize(req);

3279         if (loaned_buffers) {
3280                 uiop = (uio_t *)rfs_setup_xuio(vp);
3281                 ASSERT(uiop != NULL);
3282                 uiop->uio_segflg = UIO_SYSSPACE;
3283                 uiop->uio_loffset = args->offset;
3284                 uiop->uio_resid = args->count;

3286                 /* Jump to do the read if successful */
3287                 if (!VOP_REQZCBUF(vp, UIO_READ, (xuio_t *)uiop, cs->cr, &ct)) {
3288                         /*
3289                          * Need to hold the vnode until after VOP_RETZCBUF()
3290                          * is called.
3291                          */
3292                         VN_HOLD(vp);
3293                         goto doio_read;
3294                 }

3296                 DTRACE_PROBE2(nfss__i__reqzcbuf_failed, int,
3297                     uiop->uio_loffset, int, uiop->uio_resid);

3299                 uiop->uio_extflg = 0;

3301                 /* failure to setup for zero copy */
3302                 rfs_free_xuio((void *)uiop);
3303                 loaned_buffers = 0;
3304         }

3306         /*
3307          * If returning data via RDMA Write, then grab the chunk list. If we
3308          * aren't returning READ data w/RDMA_WRITE, then grab a mblk.
3309          */
3310         if (rdma_used) {
3311                 mp = NULL;
3312                 (void) rdma_get_wchunk(req, &iov, args->wlist);
3313                 uio.uio_iov = &iov;
3314                 uio.uio_iovcnt = 1;
3315         } else {
3316                 /*
3317                  * mp will contain the data to be sent out in the read reply.
3318                  * It will be freed after the reply has been sent.
3319                  */
3320                 mp = rfs_read_alloc(args->count, &iovp, &iovcnt);
3321                 ASSERT(mp != NULL);
```



```
3322                 ASSERT(alloc_err == 0);
3323                 uio.uio_iov = iovp;
3324                 uio.uio_iovcnt = iovcnt;
3325         }

3327         uio.uio_segflg = UIO_SYSSPACE;
3328         uio.uio_extflg = UIO_COPY_CACHED;
3329         uio.uio_loffset = args->offset;
3330         uio.uio_resid = args->count;
3331         uiop = &uio;

3333 doio_read:
3334         error = do_io(FREAD, vp, uiop, 0, cs->cr, &ct);

3336         va.va_mask = AT_SIZE;
3337         verror = VOP_GETATTR(vp, &va, 0, cs->cr, &ct);

3339         if (error) {
3340                 if (mp)
3341                         freemsg(mp);
3342                 *cs->statusp = resp->status = puterrno4(error);
3343                 goto out;
3344         }

3346         /* make mblk using zc buffers */
3347         if (loaned_buffers) {
3348                 mp = uio_to_mblk(uiop);
3349                 ASSERT(mp != NULL);
3350         }

3352         *cs->statusp = resp->status = NFS4_OK;

3354         ASSERT(uiop->uio_resid >= 0);
3355         resp->data_len = args->count - uiop->uio_resid;
3356         if (mp) {
3357                 resp->data_val = (char *)mp->b_datap->db_base;
3358                 rfs_rndup_mblks(mp, resp->data_len, loaned_buffers);
3359         } else {
3360                 resp->data_val = (caddr_t)iov.iov_base;
3361         }

3363         resp->mblk = mp;

3365         if (!verror && offset + resp->data_len == va.va_size)
3366                 resp->eof = TRUE;
3367         else
3368                 resp->eof = FALSE;

3370         if (rdma_used) {
3371                 if (!rdma_setup_read_data4(args, resp)) {
3372                         *cs->statusp = resp->status = NFS4ERR_INVAL;
3373                 }
3374         } else {
3375                 resp->wlist = NULL;
3376         }

3378 out:
3379         if (in_crit)
3380                 nbl_end_crit(vp);

3382         if (iovp != NULL)
3383                 kmem_free(iovp, iovcnt * sizeof (struct iovec));

3385         DTRACE_NFSV4_2(op__read__done, struct compound_state *, cs,
3386             READ4res *, resp);
3387 }
```

```
3389 static void
3390 rfs4_op_read_free(nfs_resop4 *resop)
3391 {
3392         READ4res        *resp = &resop->nfs_resop4_u.opread;

3394         if (resp->status == NFS4_OK && resp->mblk != NULL) {
3395                 freemsg(resp->mblk);
3396                 resp->mblk = NULL;
3397                 resp->data_val = NULL;
3398                 resp->data_len = 0;
3399         }
3400 }

3402 static void
3403 rfs4_op_readdir_free(nfs_resop4 * resop)
3404 {
3405         READDIR4res    *resp = &resop->nfs_resop4_u.opreaddir;

3407         if (resp->status == NFS4_OK && resp->mblk != NULL) {
3408                 freeb(resp->mblk);
3409                 resp->mblk = NULL;
3410                 resp->data_len = 0;
3411         }
3412 }


3415 /* ARGSUSED */
3416 static void
3417 rfs4_op_putpubfh(nfs_argop4 *args, nfs_resop4 *resop, struct svc_req *req,
3418     struct compound_state *cs)
3419 {
3420         PUTPUBFH4res    *resp = &resop->nfs_resop4_u.opputpubfh;
3421         int             error;
3422         vnode_t         *vp;
3423         struct exportinfo *exi, *sav_exi;
3424         nfs_fh4_fmt_t   *fh_fmtp;

3426         DTRACE_NFSV4_1(op__putpubfh__start, struct compound_state *, cs);

3428         if (cs->vp) {
3429                 VN_RELE(cs->vp);
3430                 cs->vp = NULL;
3431         }

3433         if (cs->cr)
3434                 crfree(cs->cr);

3436         cs->cr = crdup(cs->basecr);

3438         vp = exi_public->exi_vp;
3439         if (vp == NULL) {
3440                 *cs->statusp = resp->status = NFS4ERR_SERVERFAULT;
3441                 goto out;
3442         }

3444         error = makefh4(&cs->fh, vp, exi_public);
3445         if (error != 0) {
3446                 *cs->statusp = resp->status = puterrno4(error);
3447                 goto out;
3448         }
3449         sav_exi = cs->exi;
3450         if (exi_public == exi_root) {
3451                 /*
3452                  * No filesystem is actually shared public, so we default
3453                  * to exi_root. In this case, we must check whether root
```



```
3454                  * is exported.
3455                  */
3456                 fh_fmtp = (nfs_fh4_fmt_t *)cs->fh.nfs_fh4_val;

3458                 /*
3459                  * if root filesystem is exported, the exportinfo struct that we
3460                  * should use is what checkexport returns, because root_exi is
2652                  * should use is what checkexport4 returns, because root_exi is
3461                  * actually a mostly empty struct.
3462                  */
3463                 exi = checkexport(&fh_fmtp->fh4_fsid,
2655                 exi = checkexport4(&fh_fmtp->fh4_fsid,
3464                     (fid_t *)&fh_fmtp->fh4_xlen, NULL);
3465                 if (exi) {
3466                         cs->exi = exi;
3467                 } else {
3468                         exi_hold(exi_public);
3469                         cs->exi = exi_public;
3470                 }
2657                 cs->exi = ((exi != NULL) ? exi : exi_public);
3471         } else {
3472                 /*
3473                  * it's a properly shared filesystem
3474                  */
3475                 exi_hold(exi_public);
3476 #endif /* ! codereview */
3477                 cs->exi = exi_public;
3478         }

3480         if (is_system_labeled()) {
3481                 bslabel_t *clabel;

3483                 ASSERT(req->rq_label != NULL);
3484                 clabel = req->rq_label;
3485                 DTRACE_PROBE2(tx__rfs4__log__info__opputpubfh__clabel, char *,
3486                     "got client label from request(1)",
3487                     struct svc_req *, req);
3488                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
3489                         if (!do_rfs_label_check(clabel, vp, DOMINANCE_CHECK,
3490                             cs->exi)) {
3491                                 *cs->statusp = resp->status =
3492                                     NFS4ERR_SERVERFAULT;
3493                                 if (sav_exi)
3494                                         exi_rele(sav_exi);
3495 #endif /* ! codereview */
3496                                 goto out;
3497                         }
3498                 }
3499         }

3501         VN_HOLD(vp);
3502         cs->vp = vp;

3504         if ((resp->status = call_checkauth4(cs, req)) != NFS4_OK) {
3505                 VN_RELE(cs->vp);
3506                 cs->vp = NULL;
3507                 exi_rele(cs->exi);
3508 #endif /* ! codereview */
3509                 cs->exi = sav_exi;
3510                 goto out;
3511         }
3512         if (sav_exi)
3513                 exi_rele(sav_exi);
3514 #endif /* ! codereview */

3516         *cs->statusp = resp->status = NFS4_OK;
```

```
3517 out:
3518         DTRACE_NFSV4_2(op__putpubfh__done, struct compound_state *, cs,
3519             PUTPUBFH4res *, resp);
3520 }

3522 /*
3523  * XXX - issue with put*fh operations. Suppose /export/home is exported.
3524  * Suppose an NFS client goes to mount /export/home/joe. If /export, home,
3525  * or joe have restrictive search permissions, then we shouldn't let
3526  * the client get a file handle. This is easy to enforce. However, we
3527  * don't know what security flavor should be used until we resolve the
3528  * path name. Another complication is uid mapping. If root is
3529  * the user, then it will be mapped to the anonymous user by default,
3530  * but we won't know that till we've resolved the path name. And we won't
3531  * know what the anonymous user is.
3532  * Luckily, SECINFO is specified to take a full filename.
3533  * So what we will have to in rfs4_op_lookup is check that flavor of
3534  * the target object matches that of the request, and if root was the
3535  * caller, check for the root= and anon= options, and if necessary,
3536  * repeat the lookup using the right cred_t. But that's not done yet.
3537  */
3538 /* ARGSUSED */
3539 static void
3540 rfs4_op_putfh(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
3541     struct compound_state *cs)
3542 {
3543         PUTFH4args *args = &argop->nfs_argop4_u.opputfh;
3544         PUTFH4res *resp = &resop->nfs_resop4_u.opputfh;
3545         nfs_fh4_fmt_t *fh_fmtp;

3547         DTRACE_NFSV4_2(op__putfh__start, struct compound_state *, cs,
3548             PUTFH4args *, args);

3550         if (cs->vp) {
3551                 VN_RELE(cs->vp);
3552                 cs->vp = NULL;
3553         }

3555         if (cs->cr) {
3556                 crfree(cs->cr);
3557                 cs->cr = NULL;
3558         }


3561         if (args->object.nfs_fh4_len < NFS_FH4_LEN) {
3562                 *cs->statusp = resp->status = NFS4ERR_BADHANDLE;
3563                 goto out;
3564         }

3566         fh_fmtp = (nfs_fh4_fmt_t *)args->object.nfs_fh4_val;
3567         if (cs->exi)
3568                 exi_rele(cs->exi);
3569         cs->exi = checkexport(&fh_fmtp->fh4_fsid, (fid_t *)&fh_fmtp->fh4_xlen,
2662         cs->exi = checkexport4(&fh_fmtp->fh4_fsid, (fid_t *)&fh_fmtp->fh4_xlen,
3570             NULL);

3572         if (cs->exi == NULL) {
3573                 *cs->statusp = resp->status = NFS4ERR_STALE;
3574                 goto out;
3575         }

3577         cs->cr = crdup(cs->basecr);

3579         ASSERT(cs->cr != NULL);

3581         if (! (cs->vp = nfs4_fhtovp(&args->object, cs->exi, &resp->status))) {
```



```
3582                 *cs->statusp = resp->status;
3583                 goto out;
3584         }

3586         if ((resp->status = call_checkauth4(cs, req)) != NFS4_OK) {
3587                 VN_RELE(cs->vp);
3588                 cs->vp = NULL;
3589                 goto out;
3590         }

3592         nfs_fh4_copy(&args->object, &cs->fh);
3593         *cs->statusp = resp->status = NFS4_OK;
3594         cs->deleg = FALSE;

3596 out:
3597         DTRACE_NFSV4_2(op__putfh__done, struct compound_state *, cs,
3598             PUTFH4res *, resp);
3599 }

3601 /* ARGSUSED */
3602 static void
3603 rfs4_op_putrootfh(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
3604     struct compound_state *cs)
3605 {
3606         PUTROOTFH4res *resp = &resop->nfs_resop4_u.opputrootfh;
3607         int error;
3608         fid_t fid;
3609         struct exportinfo *exi, *sav_exi;

3611         DTRACE_NFSV4_1(op__putrootfh__start, struct compound_state *, cs);

3613         if (cs->vp) {
3614                 VN_RELE(cs->vp);
3615                 cs->vp = NULL;
3616         }

3618         if (cs->cr)
3619                 crfree(cs->cr);

3621         cs->cr = crdup(cs->basecr);

3623         /*
3624          * Using rootdir, the system root vnode,
3625          * get its fid.
3626          */
3627         bzero(&fid, sizeof (fid));
3628         fid.fid_len = MAXFIDSZ;
3629         error = vop_fid_pseudo(rootdir, &fid);
3630         if (error != 0) {
3631                 *cs->statusp = resp->status = puterrno4(error);
3632                 goto out;
3633         }

3635         /*
3636          * Then use the root fsid & fid it to find out if it's exported
3637          *
3638          * If the server root isn't exported directly, then
3639          * it should at least be a pseudo export based on
3640          * one or more exports further down in the server's
3641          * file tree.
3642          */
3643         exi = checkexport(&rootdir->v_vfsp->vfs_fsid, &fid, NULL);
2736         exi = checkexport4(&rootdir->v_vfsp->vfs_fsid, &fid, NULL);
3644         if (exi == NULL || exi->exi_export.ex_flags & EX_PUBLIC) {
3645                 NFS4_DEBUG(rfs4_debug,
3646                     (CE_WARN, "rfs4_op_putrootfh: export check failure"));
```

```
3647                *cs->statusp = resp->status = NFS4ERR_SERVERFAULT;
3648                if (exi)
3649                        exi_rele(exi);
3650 #endif /* ! codereview */
3651                goto out;
3652        }

3654        /*
3655         * Now make a filehandle based on the root
3656         * export and root vnode.
3657         */
3658        error = makefh4(&cs->fh, rootdir, exi);
3659        if (error != 0) {
3660                *cs->statusp = resp->status = puterrno4(error);
3661                exi_rele(exi);
3662 #endif /* ! codereview */
3663                goto out;
3664        }

3666        sav_exi = cs->exi;
3667        cs->exi = exi;

3669        VN_HOLD(rootdir);
3670        cs->vp = rootdir;

3672        if ((resp->status = call_checkauth4(cs, req)) != NFS4_OK) {
3673                VN_RELE(rootdir);
3674                cs->vp = NULL;
3675                exi_rele(exi);
3676 #endif /* ! codereview */
3677                cs->exi = sav_exi;
3678                goto out;
3679        }
3680        if (sav_exi)
3681                exi_rele(sav_exi);
3682 #endif /* ! codereview */

3684        *cs->statusp = resp->status = NFS4_OK;
3685        cs->deleg = FALSE;
3686 out:
3687        DTRACE_NFSV4_2(op__putrootfh__done, struct compound_state *, cs,
3688            PUTROOTFH4res *, resp);
3689 }

3691 /*
3692  * set_rdattr_params sets up the variables used to manage what information
3693  * to get for each directory entry.
3694  */
3695 static nfsstat4
3696 set_rdattr_params(struct nfs4_svgetit_arg *sargp,
3697     bitmap4 attrs, bool_t *need_to_lookup)
3698 {
3699        uint_t  va_mask;
3700        nfsstat4 status;
3701        bitmap4 objbits;

3703        status = bitmap4_to_attrmask(attrs, sargp);
3704        if (status != NFS4_OK) {
3705                /*
3706                 * could not even figure attr mask
3707                 */
3708                return (status);
3709        }
3710        va_mask = sargp->vap->va_mask;

3712        /*
```



```
3713         * dirent's d_ino is always correct value for mounted_on_fileid.
3714         * mntdfid_set is set once here, but mounted_on_fileid is
3715         * set in main dirent processing loop for each dirent.
3716         * The mntdfid_set is a simple optimization that lets the
3717         * server attr code avoid work when caller is readdir.
3718         */
3719        sargp->mntdfid_set = TRUE;

3721        /*
3722         * Lookup entry only if client asked for any of the following:
3723         * a) vattr attrs
3724         * b) vfs attrs
3725         * c) attrs w/per-object scope requested (change, filehandle, etc)
3726         *    other than mounted_on_fileid (which we can take from dirent)
3727         */
3728        objbits = attrs ? attrs & NFS4_VP_ATTR_MASK : 0;

3730        if (va_mask || sargp->sbp || (objbits & ~FATTR4_MOUNTED_ON_FILEID_MASK))
3731                *need_to_lookup = TRUE;
3732        else
3733                *need_to_lookup = FALSE;

3735        if (sargp->sbp == NULL)
3736                return (NFS4_OK);

3738        /*
3739         * If filesystem attrs are requested, get them now from the
3740         * directory vp, as most entries will have same filesystem. The only
3741         * exception are mounted over entries but we handle
3742         * those as we go (XXX mounted over detection not yet implemented).
3743         */
3744        sargp->vap->va_mask = 0;        /* to avoid VOP_GETATTR */
3745        status = bitmap4_get_sysattrs(sargp);
3746        sargp->vap->va_mask = va_mask;

3748        if ((status != NFS4_OK) && sargp->rdattr_error_req) {
3749                /*
3750                 * Failed to get filesystem attributes.
3751                 * Return a rdattr_error for each entry, but don't fail.
3752                 * However, don't get any obj-dependent attrs.
3753                 */
3754                sargp->rdattr_error = status;   /* for rdattr_error */
3755                *need_to_lookup = FALSE;
3756                /*
3757                 * At least get fileid for regular readdir output
3758                 */
3759                sargp->vap->va_mask &= AT_NODEID;
3760                status = NFS4_OK;
3761        }

3763        return (status);
3764 }

3766 /*
3767  * readlink: args: CURRENT_FH.
3768  *      res: status. If success - CURRENT_FH unchanged, return linktext.
3769  */

3771 /* ARGSUSED */
3772 static void
3773 rfs4_op_readlink(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
3774     struct compound_state *cs)
3775 {
3776        READLINK4res *resp = &resop->nfs_resop4_u.opreadlink;
3777        int error;
3778        vnode_t *vp;
```

```
3779         struct iovec iov;
3780         struct vattr va;
3781         struct uio uio;
3782         char *data;
3783         struct sockaddr *ca;
3784         char *name = NULL;
3785         int is_referral;

3787         DTRACE_NFSV4_1(op__readlink__start, struct compound_state *, cs);

3789         /* CURRENT_FH: directory */
3790         vp = cs->vp;
3791         if (vp == NULL) {
3792                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
3793                 goto out;
3794         }

3796         if (cs->access == CS_ACCESS_DENIED) {
3797                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
3798                 goto out;
3799         }

3801         /* Is it a referral? */
3802         if (vn_is_nfs_reparse(vp, cs->cr) && client_is_downrev(req)) {

3804                 is_referral = 1;

3806         } else {

3808                 is_referral = 0;

3810                 if (vp->v_type == VDIR) {
3811                         *cs->statusp = resp->status = NFS4ERR_ISDIR;
3812                         goto out;
3813                 }

3815                 if (vp->v_type != VLNK) {
3816                         *cs->statusp = resp->status = NFS4ERR_INVAL;
3817                         goto out;
3818                 }

3820         }

3822         va.va_mask = AT_MODE;
3823         error = VOP_GETATTR(vp, &va, 0, cs->cr, NULL);
3824         if (error) {
3825                 *cs->statusp = resp->status = puterrno4(error);
3826                 goto out;
3827         }

3829         if (MANDLOCK(vp, va.va_mode)) {
3830                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
3831                 goto out;
3832         }

3834         data = kmem_alloc(MAXPATHLEN + 1, KM_SLEEP);

3836         if (is_referral) {
3837                 char *s;
3838                 size_t strsz;

3840                 /* Get an artificial symlink based on a referral */
3841                 s = build_symlink(vp, cs->cr, &strsz);
3842                 global_svstat_ptr[4][NFS_REFERLINKS].value.ui64++;
3843                 DTRACE_PROBE2(nfs4serv__func__referral__reflink,
3844                     vnode_t *, vp, char *, s);
```



```
3845                 if (s == NULL)
3846                         error = EINVAL;
3847                 else {
3848                         error = 0;
3849                         (void) strlcpy(data, s, MAXPATHLEN + 1);
3850                         kmem_free(s, strsz);
3851                 }

3853         } else {

3855                 iov.iov_base = data;
3856                 iov.iov_len = MAXPATHLEN;
3857                 uio.uio_iov = &iov;
3858                 uio.uio_iovcnt = 1;
3859                 uio.uio_segflg = UIO_SYSSPACE;
3860                 uio.uio_extflg = UIO_COPY_CACHED;
3861                 uio.uio_loffset = 0;
3862                 uio.uio_resid = MAXPATHLEN;

3864                 error = VOP_READLINK(vp, &uio, cs->cr, NULL);

3866                 if (!error)
3867                         *(data + MAXPATHLEN - uio.uio_resid) = '\0';
3868         }

3870         if (error) {
3871                 kmem_free((caddr_t)data, (uint_t)MAXPATHLEN + 1);
3872                 *cs->statusp = resp->status = puterrno4(error);
3873                 goto out;
3874         }

3876         ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
3877         name = nfscmd_convname(ca, cs->exi, data, NFSCMD_CONV_OUTBOUND,
3878             MAXPATHLEN  + 1);

3880         if (name == NULL) {
3881                 /*
3882                  * Even though the conversion failed, we return
3883                  * something. We just don't translate it.
3884                  */
3885                 name = data;
3886         }

3888         /*
3889          * treat link name as data
3890          */
3891         (void) str_to_utf8(name, (utf8string *)&resp->link);

3893         if (name != data)
3894                 kmem_free(name, MAXPATHLEN + 1);
3895         kmem_free((caddr_t)data, (uint_t)MAXPATHLEN + 1);
3896         *cs->statusp = resp->status = NFS4_OK;

3898 out:
3899         DTRACE_NFSV4_2(op__readlink__done, struct compound_state *, cs,
3900             READLINK4res *, resp);
3901 }

3903 static void
3904 rfs4_op_readlink_free(nfs_resop4 *resop)
3905 {
3906         READLINK4res *resp = &resop->nfs_resop4_u.opreadlink;
3907         utf8string *symlink = (utf8string *)&resp->link;

3909         if (symlink->utf8string_val) {
3910                 UTF8STRING_FREE(*symlink)
```

```
3911         }
3912 }

3914 /*
3915  * release_lockowner:
3916  *      Release any state associated with the supplied
3917  *      lockowner. Note if any lo_state is holding locks we will not
3918  *      rele that lo_state and thus the lockowner will not be destroyed.
3919  *      A client using lock after the lock owner stateid has been released
3920  *      will suffer the consequence of NFS4ERR_BAD_STATEID and would have
3921  *      to reissue the lock with new_lock_owner set to TRUE.
3922  *      args: lock_owner
3923  *      res:  status
3924  */
3925 /* ARGSUSED */
3926 static void
3927 rfs4_op_release_lockowner(nfs_argop4 *argop, nfs_resop4 *resop,
3928     struct svc_req *req, struct compound_state *cs)
3929 {
3930         RELEASE_LOCKOWNER4args *ap = &argop->nfs_argop4_u.oprelease_lockowner;
3931         RELEASE_LOCKOWNER4res *resp = &resop->nfs_resop4_u.oprelease_lockowner;
3932         rfs4_lockowner_t *lo;
3933         rfs4_openowner_t *oo;
3934         rfs4_state_t *sp;
3935         rfs4_lo_state_t *lsp;
3936         rfs4_client_t *cp;
3937         bool_t create = FALSE;
3938         locklist_t *llist;
3939         sysid_t sysid;

3941         DTRACE_NFSV4_2(op__release__lockowner__start, struct compound_state *,
3942             cs, RELEASE_LOCKOWNER4args *, ap);

3944         /* Make sure there is a clientid around for this request */
3945         cp = rfs4_findclient_by_id(ap->lock_owner.clientid, FALSE);

3947         if (cp == NULL) {
3948                 *cs->statusp = resp->status =
3949                     rfs4_check_clientid(&ap->lock_owner.clientid, 0);
3950                 goto out;
3951         }
3952         rfs4_client_rele(cp);

3954         lo = rfs4_findlockowner(&ap->lock_owner, &create);
3955         if (lo == NULL) {
3956                 *cs->statusp = resp->status = NFS4_OK;
3957                 goto out;
3958         }
3959         ASSERT(lo->rl_client != NULL);

3961         /*
3962          * Check for EXPIRED client. If so will reap state with in a lease
3963          * period or on next set_clientid_confirm step
3964          */
3965         if (rfs4_lease_expired(lo->rl_client)) {
3966                 rfs4_lockowner_rele(lo);
3967                 *cs->statusp = resp->status = NFS4ERR_EXPIRED;
3968                 goto out;
3969         }

3971         /*
3972          * If no sysid has been assigned, then no locks exist; just return.
3973          */
3974         rfs4_dbe_lock(lo->rl_client->rc_dbe);
3975         if (lo->rl_client->rc_sysidt == LM_NOSYSID) {
3976                 rfs4_lockowner_rele(lo);
```



```
3977                 rfs4_dbe_unlock(lo->rl_client->rc_dbe);
3978                 goto out;
3979         }

3981         sysid = lo->rl_client->rc_sysidt;
3982         rfs4_dbe_unlock(lo->rl_client->rc_dbe);

3984         /*
3985          * Mark the lockowner invalid.
3986          */
3987         rfs4_dbe_hide(lo->rl_dbe);

3989         /*
3990          * sysid-pid pair should now not be used since the lockowner is
3991          * invalid. If the client were to instantiate the lockowner again
3992          * it would be assigned a new pid. Thus we can get the list of
3993          * current locks.
3994          */

3996         llist = flk_get_active_locks(sysid, lo->rl_pid);
3997         /* If we are still holding locks fail */
3998         if (llist != NULL) {

4000                 *cs->statusp = resp->status = NFS4ERR_LOCKS_HELD;

4002                 flk_free_locklist(llist);
4003                 /*
4004                  * We need to unhide the lockowner so the client can
4005                  * try it again. The bad thing here is if the client
4006                  * has a logic error that took it here in the first place
4007                  * he probably has lost accounting of the locks that it
4008                  * is holding. So we may have dangling state until the
4009                  * open owner state is reaped via close. One scenario
4010                  * that could possibly occur is that the client has
4011                  * sent the unlock request(s) in separate threads
4012                  * and has not waited for the replies before sending the
4013                  * RELEASE_LOCKOWNER request. Presumably, it would expect
4014                  * and deal appropriately with NFS4ERR_LOCKS_HELD, by
4015                  * reissuing the request.
4016                  */
4017                 rfs4_dbe_unhide(lo->rl_dbe);
4018                 rfs4_lockowner_rele(lo);
4019                 goto out;
4020         }

4022         /*
4023          * For the corresponding client we need to check each open
4024          * owner for any opens that have lockowner state associated
4025          * with this lockowner.
4026          */

4028         rfs4_dbe_lock(lo->rl_client->rc_dbe);
4029         for (oo = list_head(&lo->rl_client->rc_openownerlist); oo != NULL;
4030             oo = list_next(&lo->rl_client->rc_openownerlist, oo)) {

4032                 rfs4_dbe_lock(oo->ro_dbe);
4033                 for (sp = list_head(&oo->ro_statelist); sp != NULL;
4034                     sp = list_next(&oo->ro_statelist, sp)) {

4036                         rfs4_dbe_lock(sp->rs_dbe);
4037                         for (lsp = list_head(&sp->rs_lostatelist);
4038                             lsp != NULL;
4039                             lsp = list_next(&sp->rs_lostatelist, lsp)) {
4040                                 if (lsp->rls_locker == lo) {
4041                                         rfs4_dbe_lock(lsp->rls_dbe);
4042                                         rfs4_dbe_invalidate(lsp->rls_dbe);
```

```
4043                                    rfs4_dbe_unlock(lsp->rls_dbe);
4044                            }
4045                    }
4046                    rfs4_dbe_unlock(sp->rs_dbe);
4047            }
4048            rfs4_dbe_unlock(oo->ro_dbe);
4049    }
4050    rfs4_dbe_unlock(lo->rl_client->rc_dbe);

4052    rfs4_lockowner_rele(lo);

4054    *cs->statusp = resp->status = NFS4_OK;

4056 out:
4057    DTRACE_NFSV4_2(op__release__lockowner__done, struct compound_state *,
4058        cs, RELEASE_LOCKOWNER4res *, resp);
4059 }

4061 /*
4062  * short utility function to lookup a file and recall the delegation
4063  */
4064 static rfs4_file_t *
4065 rfs4_lookup_and_findfile(vnode_t *dvp, char *nm, vnode_t **vpp,
4066     int *lkup_error, cred_t *cr)
4067 {
4068    vnode_t *vp;
4069    rfs4_file_t *fp = NULL;
4070    bool_t fcreate = FALSE;
4071    int error;

4073    if (vpp)
4074            *vpp = NULL;

4076    if ((error = VOP_LOOKUP(dvp, nm, &vp, NULL, 0, NULL, cr, NULL, NULL,
4077        NULL)) == 0) {
4078            if (vp->v_type == VREG)
4079                    fp = rfs4_findfile(vp, NULL, &fcreate);
4080            if (vpp)
4081                    *vpp = vp;
4082            else
4083                    VN_RELE(vp);
4084    }

4086    if (lkup_error)
4087            *lkup_error = error;

4089    return (fp);
4090 }

4092 /*
4093  * remove: args: CURRENT_FH: directory; name.
4094  *      res: status. If success - CURRENT_FH unchanged, return change_info
4095  *           for directory.
4096  */
4097 /* ARGSUSED */
4098 static void
4099 rfs4_op_remove(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
4100     struct compound_state *cs)
4101 {
4102    REMOVE4args *args = &argop->nfs_argop4_u.opremove;
4103    REMOVE4res *resp = &resop->nfs_resop4_u.opremove;
4104    int error;
4105    vnode_t *dvp, *vp;
4106    struct vattr bdva, idva, adva;
4107    char *nm;
4108    uint_t len;
```



```
4109    rfs4_file_t *fp;
4110    int in_crit = 0;
4111    bslabel_t *clabel;
4112    struct sockaddr *ca;
4113    char *name = NULL;
4114    nfsstat4 status;

4116    DTRACE_NFSV4_2(op__remove__start, struct compound_state *, cs,
4117        REMOVE4args *, args);

4119    /* CURRENT_FH: directory */
4120    dvp = cs->vp;
4121    if (dvp == NULL) {
4122            *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
4123            goto out;
4124    }

4126    if (cs->access == CS_ACCESS_DENIED) {
4127            *cs->statusp = resp->status = NFS4ERR_ACCESS;
4128            goto out;
4129    }

4131    /*
4132     * If there is an unshared filesystem mounted on this vnode,
4133     * Do not allow to remove anything in this directory.
4134     */
4135    if (vn_ismntpt(dvp)) {
4136            *cs->statusp = resp->status = NFS4ERR_ACCESS;
4137            goto out;
4138    }

4140    if (dvp->v_type != VDIR) {
4141            *cs->statusp = resp->status = NFS4ERR_NOTDIR;
4142            goto out;
4143    }

4145    status = utf8_dir_verify(&args->target);
4146    if (status != NFS4_OK) {
4147            *cs->statusp = resp->status = status;
4148            goto out;
4149    }

4151    /*
4152     * Lookup the file so that we can check if it's a directory
4153     */
4154    nm = utf8_to_fn(&args->target, &len, NULL);
4155    if (nm == NULL) {
4156            *cs->statusp = resp->status = NFS4ERR_INVAL;
4157            goto out;
4158    }

4160    if (len > MAXNAMELEN) {
4161            *cs->statusp = resp->status = NFS4ERR_NAMETOOLONG;
4162            kmem_free(nm, len);
4163            goto out;
4164    }

4166    if (rdonly4(req, cs)) {
4167            *cs->statusp = resp->status = NFS4ERR_ROFS;
4168            kmem_free(nm, len);
4169            goto out;
4170    }

4172    ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
4173    name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND,
4174        MAXPATHLEN  + 1);
```

```
4176         if (name == NULL) {
4177                 *cs->statusp = resp->status = NFS4ERR_INVAL;
4178                 kmem_free(nm, len);
4179                 goto out;
4180         }

4182         /*
4183          * Lookup the file to determine type and while we are see if
4184          * there is a file struct around and check for delegation.
4185          * We don't need to acquire va_seq before this lookup, if
4186          * it causes an update, cinfo.before will not match, which will
4187          * trigger a cache flush even if atomic is TRUE.
4188          */
4189         if (fp = rfs4_lookup_and_findfile(dvp, name, &vp, &error, cs->cr)) {
4190                 if (rfs4_check_delegated_byfp(FWRITE, fp, TRUE, TRUE, TRUE,
4191                     NULL)) {
4192                         VN_RELE(vp);
4193                         rfs4_file_rele(fp);
4194                         *cs->statusp = resp->status = NFS4ERR_DELAY;
4195                         if (nm != name)
4196                                 kmem_free(name, MAXPATHLEN + 1);
4197                         kmem_free(nm, len);
4198                         goto out;
4199                 }
4200         }

4202         /* Didn't find anything to remove */
4203         if (vp == NULL) {
4204                 *cs->statusp = resp->status = error;
4205                 if (nm != name)
4206                         kmem_free(name, MAXPATHLEN + 1);
4207                 kmem_free(nm, len);
4208                 goto out;
4209         }

4211         if (nbl_need_check(vp)) {
4212                 nbl_start_crit(vp, RW_READER);
4213                 in_crit = 1;
4214                 if (nbl_conflict(vp, NBL_REMOVE, 0, 0, 0, NULL)) {
4215                         *cs->statusp = resp->status = NFS4ERR_FILE_OPEN;
4216                         if (nm != name)
4217                                 kmem_free(name, MAXPATHLEN + 1);
4218                         kmem_free(nm, len);
4219                         nbl_end_crit(vp);
4220                         VN_RELE(vp);
4221                         if (fp) {
4222                                 rfs4_clear_dont_grant(fp);
4223                                 rfs4_file_rele(fp);
4224                         }
4225                         goto out;
4226                 }
4227         }

4229         /* check label before allowing removal */
4230         if (is_system_labeled()) {
4231                 ASSERT(req->rq_label != NULL);
4232                 clabel = req->rq_label;
4233                 DTRACE_PROBE2(tx__rfs4__log__info__opremove__clabel, char *,
4234                     "got client label from request(1)",
4235                     struct svc_req *, req);
4236                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
4237                         if (!do_rfs_label_check(clabel, vp, EQUALITY_CHECK,
4238                             cs->exi)) {
4239                                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
4240                                 if (name != nm)
```



```
4241                                         kmem_free(name, MAXPATHLEN + 1);
4242                                 kmem_free(nm, len);
4243                                 if (in_crit)
4244                                         nbl_end_crit(vp);
4245                                 VN_RELE(vp);
4246                                 if (fp) {
4247                                         rfs4_clear_dont_grant(fp);
4248                                         rfs4_file_rele(fp);
4249                                 }
4250                                 goto out;
4251                         }
4252                 }
4253         }

4255         /* Get dir "before" change value */
4256         bdva.va_mask = AT_CTIME|AT_SEQ;
4257         error = VOP_GETATTR(dvp, &bdva, 0, cs->cr, NULL);
4258         if (error) {
4259                 *cs->statusp = resp->status = puterrno4(error);
4260                 if (nm != name)
4261                         kmem_free(name, MAXPATHLEN + 1);
4262                 kmem_free(nm, len);
4263                 if (in_crit)
4264                         nbl_end_crit(vp);
4265                 VN_RELE(vp);
4266                 if (fp) {
4267                         rfs4_clear_dont_grant(fp);
4268                         rfs4_file_rele(fp);
4269                 }
4270                 goto out;
4271         }
4272         NFS4_SET_FATTR4_CHANGE(resp->cinfo.before, bdva.va_ctime)

4274         /* Actually do the REMOVE operation */
4275         if (vp->v_type == VDIR) {
4276                 /*
4277                  * Can't remove a directory that has a mounted-on filesystem.
4278                  */
4279                 if (vn_ismntpt(vp)) {
4280                         error = EACCES;
4281                 } else {
4282                         /*
4283                          * System V defines rmdir to return EEXIST,
4284                          * not ENOTEMPTY, if the directory is not
4285                          * empty.  A System V NFS server needs to map
4286                          * NFS4ERR_EXIST to NFS4ERR_NOTEMPTY to
4287                          * transmit over the wire.
4288                          */
4289                         if ((error = VOP_RMDIR(dvp, name, rootdir, cs->cr,
4290                             NULL, 0)) == EEXIST)
4291                                 error = ENOTEMPTY;
4292                 }
4293         } else {
4294                 if ((error = VOP_REMOVE(dvp, name, cs->cr, NULL, 0)) == 0 &&
4295                     fp != NULL) {
4296                         struct vattr va;
4297                         vnode_t *tvp;

4299                         rfs4_dbe_lock(fp->rf_dbe);
4300                         tvp = fp->rf_vp;
4301                         if (tvp)
4302                                 VN_HOLD(tvp);
4303                         rfs4_dbe_unlock(fp->rf_dbe);

4305                         if (tvp) {
4306                                 /*
```

```
4307                                 * This is va_seq safe because we are not
4308                                 * manipulating dvp.
4309                                 */
4310                                va.va_mask = AT_NLINK;
4311                                if (!VOP_GETATTR(tvp, &va, 0, cs->cr, NULL) &&
4312                                    va.va_nlink == 0) {
4313                                        /* Remove state on file remove */
4314                                        if (in_crit) {
4315                                                nbl_end_crit(vp);
4316                                                in_crit = 0;
4317                                        }
4318                                        rfs4_close_all_state(fp);
4319                                }
4320                                VN_RELE(tvp);
4321                        }
4322                }
4323        }

4325        if (in_crit)
4326                nbl_end_crit(vp);
4327        VN_RELE(vp);

4329        if (fp) {
4330                rfs4_clear_dont_grant(fp);
4331                rfs4_file_rele(fp);
4332        }
4333        if (nm != name)
4334                kmem_free(name, MAXPATHLEN + 1);
4335        kmem_free(nm, len);

4337        if (error) {
4338                *cs->statusp = resp->status = puterrno4(error);
4339                goto out;
4340        }

4342        /*
4343         * Get the initial "after" sequence number, if it fails, set to zero
4344         */
4345        idva.va_mask = AT_SEQ;
4346        if (VOP_GETATTR(dvp, &idva, 0, cs->cr, NULL))
4347                idva.va_seq = 0;

4349        /*
4350         * Force modified data and metadata out to stable storage.
4351         */
4352        (void) VOP_FSYNC(dvp, 0, cs->cr, NULL);

4354        /*
4355         * Get "after" change value, if it fails, simply return the
4356         * before value.
4357         */
4358        adva.va_mask = AT_CTIME|AT_SEQ;
4359        if (VOP_GETATTR(dvp, &adva, 0, cs->cr, NULL)) {
4360                adva.va_ctime = bdva.va_ctime;
4361                adva.va_seq = 0;
4362        }

4364        NFS4_SET_FATTR4_CHANGE(resp->cinfo.after, adva.va_ctime)

4366        /*
4367         * The cinfo.atomic = TRUE only if we have
4368         * non-zero va_seq's, and it has incremented by exactly one
4369         * during the VOP_REMOVE/RMDIR and it didn't change during
4370         * the VOP_FSYNC.
4371         */
4372        if (bdva.va_seq && idva.va_seq && adva.va_seq &&
```



```
4373            idva.va_seq == (bdva.va_seq + 1) && idva.va_seq == adva.va_seq)
4374                resp->cinfo.atomic = TRUE;
4375        else
4376                resp->cinfo.atomic = FALSE;

4378        *cs->statusp = resp->status = NFS4_OK;

4380 out:
4381        DTRACE_NFSV4_2(op__remove__done, struct compound_state *, cs,
4382            REMOVE4res *, resp);
4383 }

4385 /*
4386  * rename: args: SAVED_FH: from directory, CURRENT_FH: target directory,
4387  *               oldname and newname.
4388  *       res: status. If success - CURRENT_FH unchanged, return change_info
4389  *               for both from and target directories.
4390  */
4391 /* ARGSUSED */
4392 static void
4393 rfs4_op_rename(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
4394     struct compound_state *cs)
4395 {
4396        RENAME4args *args = &argop->nfs_argop4_u.oprename;
4397        RENAME4res *resp = &resop->nfs_resop4_u.oprename;
4398        int error;
4399        vnode_t *odvp;
4400        vnode_t *ndvp;
4401        vnode_t *srcvp, *targvp;
4402        struct vattr obdva, oidva, oadva;
4403        struct vattr nbdva, nidva, nadva;
4404        char *onm, *nnm;
4405        uint_t olen, nlen;
4406        rfs4_file_t *fp, *sfp;
4407        int in_crit_src, in_crit_targ;
4408        int fp_rele_grant_hold, sfp_rele_grant_hold;
4409        bslabel_t *clabel;
4410        struct sockaddr *ca;
4411        char *converted_onm = NULL;
4412        char *converted_nnm = NULL;
4413        nfsstat4 status;

4415        DTRACE_NFSV4_2(op__rename__start, struct compound_state *, cs,
4416            RENAME4args *, args);

4418        fp = sfp = NULL;
4419        srcvp = targvp = NULL;
4420        in_crit_src = in_crit_targ = 0;
4421        fp_rele_grant_hold = sfp_rele_grant_hold = 0;

4423        /* CURRENT_FH: target directory */
4424        ndvp = cs->vp;
4425        if (ndvp == NULL) {
4426                *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
4427                goto out;
4428        }

4430        /* SAVED_FH: from directory */
4431        odvp = cs->saved_vp;
4432        if (odvp == NULL) {
4433                *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
4434                goto out;
4435        }

4437        if (cs->access == CS_ACCESS_DENIED) {
4438                *cs->statusp = resp->status = NFS4ERR_ACCESS;
```

```
4439                goto out;
4440        }

4442        /*
4443         * If there is an unshared filesystem mounted on this vnode,
4444         * do not allow to rename objects in this directory.
4445         */
4446        if (vn_ismntpt(odvp)) {
4447                *cs->statusp = resp->status = NFS4ERR_ACCESS;
4448                goto out;
4449        }

4451        /*
4452         * If there is an unshared filesystem mounted on this vnode,
4453         * do not allow to rename to this directory.
4454         */
4455        if (vn_ismntpt(ndvp)) {
4456                *cs->statusp = resp->status = NFS4ERR_ACCESS;
4457                goto out;
4458        }

4460        if (odvp->v_type != VDIR || ndvp->v_type != VDIR) {
4461                *cs->statusp = resp->status = NFS4ERR_NOTDIR;
4462                goto out;
4463        }

4465        if (cs->saved_exi != cs->exi) {
4466                *cs->statusp = resp->status = NFS4ERR_XDEV;
4467                goto out;
4468        }

4470        status = utf8_dir_verify(&args->oldname);
4471        if (status != NFS4_OK) {
4472                *cs->statusp = resp->status = status;
4473                goto out;
4474        }

4476        status = utf8_dir_verify(&args->newname);
4477        if (status != NFS4_OK) {
4478                *cs->statusp = resp->status = status;
4479                goto out;
4480        }

4482        onm = utf8_to_fn(&args->oldname, &olen, NULL);
4483        if (onm == NULL) {
4484                *cs->statusp = resp->status = NFS4ERR_INVAL;
4485                goto out;
4486        }
4487        ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
4488        nlen = MAXPATHLEN + 1;
4489        converted_onm = nfscmd_convname(ca, cs->exi, onm, NFSCMD_CONV_INBOUND,
4490            nlen);

4492        if (converted_onm == NULL) {
4493                *cs->statusp = resp->status = NFS4ERR_INVAL;
4494                kmem_free(onm, olen);
4495                goto out;
4496        }

4498        nnm = utf8_to_fn(&args->newname, &nlen, NULL);
4499        if (nnm == NULL) {
4500                *cs->statusp = resp->status = NFS4ERR_INVAL;
4501                if (onm != converted_onm)
4502                        kmem_free(converted_onm, MAXPATHLEN + 1);
4503                kmem_free(onm, olen);
4504                goto out;
```



```
4505        }
4506        converted_nnm = nfscmd_convname(ca, cs->exi, nnm, NFSCMD_CONV_INBOUND,
4507            MAXPATHLEN  + 1);

4509        if (converted_nnm == NULL) {
4510                *cs->statusp = resp->status = NFS4ERR_INVAL;
4511                kmem_free(nnm, nlen);
4512                nnm = NULL;
4513                if (onm != converted_onm)
4514                        kmem_free(converted_onm, MAXPATHLEN + 1);
4515                kmem_free(onm, olen);
4516                goto out;
4517        }


4520        if (olen > MAXNAMELEN || nlen > MAXNAMELEN) {
4521                *cs->statusp = resp->status = NFS4ERR_NAMETOOLONG;
4522                kmem_free(onm, olen);
4523                kmem_free(nnm, nlen);
4524                goto out;
4525        }


4528        if (rdonly4(req, cs)) {
4529                *cs->statusp = resp->status = NFS4ERR_ROFS;
4530                if (onm != converted_onm)
4531                        kmem_free(converted_onm, MAXPATHLEN + 1);
4532                kmem_free(onm, olen);
4533                if (nnm != converted_nnm)
4534                        kmem_free(converted_nnm, MAXPATHLEN + 1);
4535                kmem_free(nnm, nlen);
4536                goto out;
4537        }

4539        /* check label of the target dir */
4540        if (is_system_labeled()) {
4541                ASSERT(req->rq_label != NULL);
4542                clabel = req->rq_label;
4543                DTRACE_PROBE2(tx__rfs4__log__info__oprename__clabel, char *,
4544                    "got client label from request(1)",
4545                    struct svc_req *, req);
4546                if (!blequal(&l_admin_low->tsl_label, clabel)) {
4547                        if (!do_rfs_label_check(clabel, ndvp,
4548                            EQUALITY_CHECK, cs->exi)) {
4549                                *cs->statusp = resp->status = NFS4ERR_ACCESS;
4550                                goto err_out;
4551                        }
4552                }
4553        }

4555        /*
4556         * Is the source a file and have a delegation?
4557         * We don't need to acquire va_seq before these lookups, if
4558         * it causes an update, cinfo.before will not match, which will
4559         * trigger a cache flush even if atomic is TRUE.
4560         */
4561        if (sfp = rfs4_lookup_and_findfile(odvp, converted_onm, &srcvp,
4562            &error, cs->cr)) {
4563                if (rfs4_check_delegated_byfp(FWRITE, sfp, TRUE, TRUE, TRUE,
4564                    NULL)) {
4565                        *cs->statusp = resp->status = NFS4ERR_DELAY;
4566                        goto err_out;
4567                }
4568        }

4570        if (srcvp == NULL) {
```

```
4571                *cs->statusp = resp->status = puterrno4(error);
4572                if (onm != converted_onm)
4573                        kmem_free(converted_onm, MAXPATHLEN + 1);
4574                kmem_free(onm, olen);
4575                if (nnm != converted_nnm)
4576                        kmem_free(converted_nnm, MAXPATHLEN + 1);
4577                kmem_free(nnm, nlen);
4578                goto out;
4579        }

4581        sfp_rele_grant_hold = 1;

4583        /* Does the destination exist and a file and have a delegation? */
4584        if (fp = rfs4_lookup_and_findfile(ndvp, converted_nnm, &targvp,
4585            NULL, cs->cr)) {
4586                if (rfs4_check_delegated_byfp(FWRITE, fp, TRUE, TRUE, TRUE,
4587                    NULL)) {
4588                        *cs->statusp = resp->status = NFS4ERR_DELAY;
4589                        goto err_out;
4590                }
4591        }
4592        fp_rele_grant_hold = 1;


4595        /* Check for NBMAND lock on both source and target */
4596        if (nbl_need_check(srcvp)) {
4597                nbl_start_crit(srcvp, RW_READER);
4598                in_crit_src = 1;
4599                if (nbl_conflict(srcvp, NBL_RENAME, 0, 0, 0, NULL)) {
4600                        *cs->statusp = resp->status = NFS4ERR_FILE_OPEN;
4601                        goto err_out;
4602                }
4603        }

4605        if (targvp && nbl_need_check(targvp)) {
4606                nbl_start_crit(targvp, RW_READER);
4607                in_crit_targ = 1;
4608                if (nbl_conflict(targvp, NBL_REMOVE, 0, 0, 0, NULL)) {
4609                        *cs->statusp = resp->status = NFS4ERR_FILE_OPEN;
4610                        goto err_out;
4611                }
4612        }

4614        /* Get source "before" change value */
4615        obdva.va_mask = AT_CTIME|AT_SEQ;
4616        error = VOP_GETATTR(odvp, &obdva, 0, cs->cr, NULL);
4617        if (!error) {
4618                nbdva.va_mask = AT_CTIME|AT_SEQ;
4619                error = VOP_GETATTR(ndvp, &nbdva, 0, cs->cr, NULL);
4620        }
4621        if (error) {
4622                *cs->statusp = resp->status = puterrno4(error);
4623                goto err_out;
4624        }

4626        NFS4_SET_FATTR4_CHANGE(resp->source_cinfo.before, obdva.va_ctime)
4627        NFS4_SET_FATTR4_CHANGE(resp->target_cinfo.before, nbdva.va_ctime)

4629        if ((error = VOP_RENAME(odvp, converted_onm, ndvp, converted_nnm,
4630            cs->cr, NULL, 0)) == 0 && fp != NULL) {
4631                struct vattr va;
4632                vnode_t *tvp;

4634                rfs4_dbe_lock(fp->rf_dbe);
4635                tvp = fp->rf_vp;
4636                if (tvp)
```



```
4637                        VN_HOLD(tvp);
4638                rfs4_dbe_unlock(fp->rf_dbe);

4640                if (tvp) {
4641                        va.va_mask = AT_NLINK;
4642                        if (!VOP_GETATTR(tvp, &va, 0, cs->cr, NULL) &&
4643                            va.va_nlink == 0) {
4644                                /* The file is gone and so should the state */
4645                                if (in_crit_targ) {
4646                                        nbl_end_crit(targvp);
4647                                        in_crit_targ = 0;
4648                                }
4649                                rfs4_close_all_state(fp);
4650                        }
4651                        VN_RELE(tvp);
4652                }
4653        }
4654        if (error == 0)
4655                vn_renamepath(ndvp, srcvp, nnm, nlen - 1);

4657        if (in_crit_src)
4658                nbl_end_crit(srcvp);
4659        if (srcvp)
4660                VN_RELE(srcvp);
4661        if (in_crit_targ)
4662                nbl_end_crit(targvp);
4663        if (targvp)
4664                VN_RELE(targvp);

4666        if (sfp) {
4667                rfs4_clear_dont_grant(sfp);
4668                rfs4_file_rele(sfp);
4669        }
4670        if (fp) {
4671                rfs4_clear_dont_grant(fp);
4672                rfs4_file_rele(fp);
4673        }

4675        if (converted_onm != onm)
4676                kmem_free(converted_onm, MAXPATHLEN + 1);
4677        kmem_free(onm, olen);
4678        if (converted_nnm != nnm)
4679                kmem_free(converted_nnm, MAXPATHLEN + 1);
4680        kmem_free(nnm, nlen);

4682        /*
4683         * Get the initial "after" sequence number, if it fails, set to zero
4684         */
4685        oidva.va_mask = AT_SEQ;
4686        if (VOP_GETATTR(odvp, &oidva, 0, cs->cr, NULL))
4687                oidva.va_seq = 0;

4689        nidva.va_mask = AT_SEQ;
4690        if (VOP_GETATTR(ndvp, &nidva, 0, cs->cr, NULL))
4691                nidva.va_seq = 0;

4693        /*
4694         * Force modified data and metadata out to stable storage.
4695         */
4696        (void) VOP_FSYNC(odvp, 0, cs->cr, NULL);
4697        (void) VOP_FSYNC(ndvp, 0, cs->cr, NULL);

4699        if (error) {
4700                *cs->statusp = resp->status = puterrno4(error);
4701                goto out;
4702        }
```

```
4704         /*
4705          * Get "after" change values, if it fails, simply return the
4706          * before value.
4707          */
4708         oadva.va_mask = AT_CTIME|AT_SEQ;
4709         if (VOP_GETATTR(odvp, &oadva, 0, cs->cr, NULL)) {
4710                 oadva.va_ctime = obdva.va_ctime;
4711                 oadva.va_seq = 0;
4712         }

4714         nadva.va_mask = AT_CTIME|AT_SEQ;
4715         if (VOP_GETATTR(odvp, &nadva, 0, cs->cr, NULL)) {
4716                 nadva.va_ctime = nbdva.va_ctime;
4717                 nadva.va_seq = 0;
4718         }

4720         NFS4_SET_FATTR4_CHANGE(resp->source_cinfo.after, oadva.va_ctime)
4721         NFS4_SET_FATTR4_CHANGE(resp->target_cinfo.after, nadva.va_ctime)

4723         /*
4724          * The cinfo.atomic = TRUE only if we have
4725          * non-zero va_seq's, and it has incremented by exactly one
4726          * during the VOP_RENAME and it didn't change during the VOP_FSYNC.
4727          */
4728         if (obdva.va_seq && oidva.va_seq && oadva.va_seq &&
4729             oidva.va_seq == (obdva.va_seq + 1) && oidva.va_seq == oadva.va_seq)
4730                 resp->source_cinfo.atomic = TRUE;
4731         else
4732                 resp->source_cinfo.atomic = FALSE;

4734         if (nbdva.va_seq && nidva.va_seq && nadva.va_seq &&
4735             nidva.va_seq == (nbdva.va_seq + 1) && nidva.va_seq == nadva.va_seq)
4736                 resp->target_cinfo.atomic = TRUE;
4737         else
4738                 resp->target_cinfo.atomic = FALSE;

4740 #ifdef  VOLATILE_FH_TEST
4741         {
4742         extern void add_volrnm_fh(struct exportinfo *, vnode_t *);

4744         /*
4745          * Add the renamed file handle to the volatile rename list
4746          */
4747         if (cs->exi->exi_export.ex_flags & EX_VOLRNM) {
4748                 /* file handles may expire on rename */
4749                 vnode_t *vp;

4751                 nnm = utf8_to_fn(&args->newname, &nlen, NULL);
4752                 /*
4753                  * Already know that nnm will be a valid string
4754                  */
4755                 error = VOP_LOOKUP(ndvp, nnm, &vp, NULL, 0, NULL, cs->cr,
4756                     NULL, NULL, NULL);
4757                 kmem_free(nnm, nlen);
4758                 if (!error) {
4759                         add_volrnm_fh(cs->exi, vp);
4760                         VN_RELE(vp);
4761                 }
4762         }
4763         }
4764 #endif  /* VOLATILE_FH_TEST */

4766         *cs->statusp = resp->status = NFS4_OK;
4767 out:
4768         DTRACE_NFSV4_2(op__rename__done, struct compound_state *, cs,
```



```
4769             RENAME4res *, resp);
4770         return;

4772 err_out:
4773         if (onm != converted_onm)
4774                 kmem_free(converted_onm, MAXPATHLEN + 1);
4775         if (onm != NULL)
4776                 kmem_free(onm, olen);
4777         if (nnm != converted_nnm)
4778                 kmem_free(converted_nnm, MAXPATHLEN + 1);
4779         if (nnm != NULL)
4780                 kmem_free(nnm, nlen);

4782         if (in_crit_src) nbl_end_crit(srcvp);
4783         if (in_crit_targ) nbl_end_crit(targvp);
4784         if (targvp) VN_RELE(targvp);
4785         if (srcvp) VN_RELE(srcvp);
4786         if (sfp) {
4787                 if (sfp_rele_grant_hold) rfs4_clear_dont_grant(sfp);
4788                 rfs4_file_rele(sfp);
4789         }
4790         if (fp) {
4791                 if (fp_rele_grant_hold) rfs4_clear_dont_grant(fp);
4792                 rfs4_file_rele(fp);
4793         }

4795         DTRACE_NFSV4_2(op__rename__done, struct compound_state *, cs,
4796             RENAME4res *, resp);
4797 }

4799 /* ARGSUSED */
4800 static void
4801 rfs4_op_renew(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
4802     struct compound_state *cs)
4803 {
4804         RENEW4args *args = &argop->nfs_argop4_u.oprenew;
4805         RENEW4res *resp = &resop->nfs_resop4_u.oprenew;
4806         rfs4_client_t *cp;

4808         DTRACE_NFSV4_2(op__renew__start, struct compound_state *, cs,
4809             RENEW4args *, args);

4811         if ((cp = rfs4_findclient_by_id(args->clientid, FALSE)) == NULL) {
4812                 *cs->statusp = resp->status =
4813                     rfs4_check_clientid(&args->clientid, 0);
4814                 goto out;
4815         }

4817         if (rfs4_lease_expired(cp)) {
4818                 rfs4_client_rele(cp);
4819                 *cs->statusp = resp->status = NFS4ERR_EXPIRED;
4820                 goto out;
4821         }

4823         rfs4_update_lease(cp);

4825         mutex_enter(cp->rc_cbinfo.cb_lock);
4826         if (cp->rc_cbinfo.cb_notified_of_cb_path_down == FALSE) {
4827                 cp->rc_cbinfo.cb_notified_of_cb_path_down = TRUE;
4828                 *cs->statusp = resp->status = NFS4ERR_CB_PATH_DOWN;
4829         } else {
4830                 *cs->statusp = resp->status = NFS4_OK;
4831         }
4832         mutex_exit(cp->rc_cbinfo.cb_lock);

4834         rfs4_client_rele(cp);
```

```
4836 out:
4837         DTRACE_NFSV4_2(op__renew__done, struct compound_state *, cs,
4838             RENEW4res *, resp);
4839 }

4841 /* ARGSUSED */
4842 static void
4843 rfs4_op_restorefh(nfs_argop4 *args, nfs_resop4 *resop, struct svc_req *req,
4844     struct compound_state *cs)
4845 {
4846         RESTOREFH4res *resp = &resop->nfs_resop4_u.oprestorefh;

4848         DTRACE_NFSV4_1(op__restorefh__start, struct compound_state *, cs);

4850         /* No need to check cs->access - we are not accessing any object */
4851         if ((cs->saved_vp == NULL) || (cs->saved_fh.nfs_fh4_val == NULL)) {
4852                 *cs->statusp = resp->status = NFS4ERR_RESTOREFH;
4853                 goto out;
4854         }
4855         if (cs->vp != NULL) {
4856                 VN_RELE(cs->vp);
4857         }
4858         cs->vp = cs->saved_vp;
4859         cs->saved_vp = NULL;
4860         if (cs->exi)
4861                 exi_rele(cs->exi);
4862 #endif /* ! codereview */
4863         cs->exi = cs->saved_exi;
4864         if (cs->exi)
4865                 exi_hold(cs->exi);
4866 #endif /* ! codereview */
4867         nfs_fh4_copy(&cs->saved_fh, &cs->fh);
4868         *cs->statusp = resp->status = NFS4_OK;
4869         cs->deleg = FALSE;

4871 out:
4872         DTRACE_NFSV4_2(op__restorefh__done, struct compound_state *, cs,
4873             RESTOREFH4res *, resp);
4874 }

4876 /* ARGSUSED */
4877 static void
4878 rfs4_op_savefh(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
4879     struct compound_state *cs)
4880 {
4881         SAVEFH4res *resp = &resop->nfs_resop4_u.opsavefh;

4883         DTRACE_NFSV4_1(op__savefh__start, struct compound_state *, cs);

4885         /* No need to check cs->access - we are not accessing any object */
4886         if (cs->vp == NULL) {
4887                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
4888                 goto out;
4889         }
4890         if (cs->saved_vp != NULL) {
4891                 VN_RELE(cs->saved_vp);
4892         }
4893         cs->saved_vp = cs->vp;
4894         VN_HOLD(cs->saved_vp);
4895         if (cs->saved_exi)
4896                 exi_rele(cs->saved_exi);
4897 #endif /* ! codereview */
4898         cs->saved_exi = cs->exi;
4899         if (cs->saved_exi)
4900                 exi_hold(cs->saved_exi);
```



```
4901 #endif /* ! codereview */
4902         /*
4903          * since SAVEFH is fairly rare, don't alloc space for its fh
4904          * unless necessary.
4905          */
4906         if (cs->saved_fh.nfs_fh4_val == NULL) {
4907                 cs->saved_fh.nfs_fh4_val = kmem_alloc(NFS4_FHSIZE, KM_SLEEP);
4908         }
4909         nfs_fh4_copy(&cs->fh, &cs->saved_fh);
4910         *cs->statusp = resp->status = NFS4_OK;

4912 out:
4913         DTRACE_NFSV4_2(op__savefh__done, struct compound_state *, cs,
4914             SAVEFH4res *, resp);
4915 }

4917 /*
4918  * rfs4_verify_attr is called when nfsv4 Setattr failed, but we wish to
4919  * return the bitmap of attrs that were set successfully. It is also
4920  * called by Verify/Nverify to test the vattr/vfsstat attrs. It should
4921  * always be called only after rfs4_do_set_attrs().
4922  *
4923  * Verify that the attributes are same as the expected ones. sargp->vap
4924  * and sargp->sbp contain the input attributes as translated from fattr4.
4925  *
4926  * This function verifies only the attrs that correspond to a vattr or
4927  * vfsstat struct. That is because of the extra step needed to get the
4928  * corresponding system structs. Other attributes have already been set or
4929  * verified by do_rfs4_set_attrs.
4930  *
4931  * Return 0 if all attrs match, -1 if some don't, error if error processing.
4932  */
4933 static int
4934 rfs4_verify_attr(struct nfs4_svgetit_arg *sargp,
4935     bitmap4 *resp, struct nfs4_ntov_table *ntovp)
4936 {
4937         int error, ret_error = 0;
4938         int i, k;
4939         uint_t sva_mask = sargp->vap->va_mask;
4940         uint_t vbit;
4941         union nfs4_attr_u *na;
4942         uint8_t *amap;
4943         bool_t getsb = ntovp->vfsstat;

4945         if (sva_mask != 0) {
4946                 /*
4947                  * Okay to overwrite sargp->vap because we verify based
4948                  * on the incoming values.
4949                  */
4950                 ret_error = VOP_GETATTR(sargp->cs->vp, sargp->vap, 0,
4951                     sargp->cs->cr, NULL);
4952                 if (ret_error) {
4953                         if (resp == NULL)
4954                                 return (ret_error);
4955                         /*
4956                          * Must return bitmap of successful attrs
4957                          */
4958                         sva_mask = 0;   /* to prevent checking vap later */
4959                 } else {
4960                         /*
4961                          * Some file systems clobber va_mask. it is probably
4962                          * wrong of them to do so, nonethless we practice
4963                          * defensive coding.
4964                          * See bug id 4276830.
4965                          */
4966                         sargp->vap->va_mask = sva_mask;
```

```
4967                }
4968        }

4970        if (getsb) {
4971                /*
4972                 * Now get the superblock and loop on the bitmap, as there is
4973                 * no simple way of translating from superblock to bitmap4.
4974                 */
4975                ret_error = VFS_STATVFS(sargp->cs->vp->v_vfsp, sargp->sbp);
4976                if (ret_error) {
4977                        if (resp == NULL)
4978                                goto errout;
4979                        getsb = FALSE;
4980                }
4981        }

4983        /*
4984         * Now loop and verify each attribute which getattr returned
4985         * whether it's the same as the input.
4986         */
4987        if (resp == NULL && !getsb && (sva_mask == 0))
4988                goto errout;

4990        na = ntovp->na;
4991        amap = ntovp->amap;
4992        k = 0;
4993        for (i = 0; i < ntovp->attrcnt; i++, na++, amap++) {
4994                k = *amap;
4995                ASSERT(nfs4_ntov_map[k].nval == k);
4996                vbit = nfs4_ntov_map[k].vbit;

4998                /*
4999                 * If vattr attribute but VOP_GETATTR failed, or it's
5000                 * superblock attribute but VFS_STATVFS failed, skip
5001                 */
5002                if (vbit) {
5003                        if ((vbit & sva_mask) == 0)
5004                                continue;
5005                } else if (!(getsb && nfs4_ntov_map[k].vfsstat)) {
5006                        continue;
5007                }
5008                error = (*nfs4_ntov_map[k].sv_getit)(NFS4ATTR_VERIT, sargp, na);
5009                if (resp != NULL) {
5010                        if (error)
5011                                ret_error = -1; /* not all match */
5012                        else    /* update response bitmap */
5013                                *resp |= nfs4_ntov_map[k].fbit;
5014                        continue;
5015                }
5016                if (error) {
5017                        ret_error = -1; /* not all match */
5018                        break;
5019                }
5020        }
5021 errout:
5022        return (ret_error);
5023 }

5025 /*
5026  * Decode the attribute to be set/verified. If the attr requires a sys op
5027  * (VOP_GETATTR, VFS_VFSSTAT), and the request is to verify, then don't
5028  * call the sv_getit function for it, because the sys op hasn't yet been done.
5029  * Return 0 for success, error code if failed.
5030  *
5031  * Note: the decoded arg is not freed here but in nfs4_ntov_table_free.
5032  */
```



```
5033 static int
5034 decode_fattr4_attr(nfs4_attr_cmd_t cmd, struct nfs4_svgetit_arg *sargp,
5035     int k, XDR *xdrp, bitmap4 *resp_bval, union nfs4_attr_u *nap)
5036 {
5037        int error = 0;
5038        bool_t set_later;

5040        sargp->vap->va_mask |= nfs4_ntov_map[k].vbit;

5042        if ((*nfs4_ntov_map[k].xfunc)(xdrp, nap)) {
5043                set_later = nfs4_ntov_map[k].vbit || nfs4_ntov_map[k].vfsstat;
5044                /*
5045                 * don't verify yet if a vattr or sb dependent attr,
5046                 * because we don't have their sys values yet.
5047                 * Will be done later.
5048                 */
5049                if (! (set_later && (cmd == NFS4ATTR_VERIT))) {
5050                        /*
5051                         * ACLs are a special case, since setting the MODE
5052                         * conflicts with setting the ACL.  We delay setting
5053                         * the ACL until all other attributes have been set.
5054                         * The ACL gets set in do_rfs4_op_setattr().
5055                         */
5056                        if (nfs4_ntov_map[k].fbit != FATTR4_ACL_MASK) {
5057                                error = (*nfs4_ntov_map[k].sv_getit)(cmd,
5058                                    sargp, nap);
5059                                if (error) {
5060                                        xdr_free(nfs4_ntov_map[k].xfunc,
5061                                            (caddr_t)nap);
5062                                }
5063                        }
5064                }
5065        } else {
5066 #ifdef  DEBUG
5067                cmn_err(CE_NOTE, "decode_fattr4_attr: error "
5068                    "decoding attribute %d\n", k);
5069 #endif
5070                error = EINVAL;
5071        }
5072        if (!error && resp_bval && !set_later) {
5073                *resp_bval |= nfs4_ntov_map[k].fbit;
5074        }

5076        return (error);
5077 }

5079 /*
5080  * Set vattr based on incoming fattr4 attrs - used by setattr.
5081  * Set response mask. Ignore any values that are not writable vattr attrs.
5082  */
5083 static nfsstat4
5084 do_rfs4_set_attrs(bitmap4 *resp, fattr4 *fattrp, struct compound_state *cs,
5085     struct nfs4_svgetit_arg *sargp, struct nfs4_ntov_table *ntovp,
5086     nfs4_attr_cmd_t cmd)
5087 {
5088        int error = 0;
5089        int i;
5090        char *attrs = fattrp->attrlist4;
5091        uint32_t attrslen = fattrp->attrlist4_len;
5092        XDR xdr;
5093        nfsstat4 status = NFS4_OK;
5094        vnode_t *vp = cs->vp;
5095        union nfs4_attr_u *na;
5096        uint8_t *amap;

5098 #ifndef lint
```

```
5099         /*
5100          * Make sure that maximum attribute number can be expressed as an
5101          * 8 bit quantity.
5102          */
5103         ASSERT(NFS4_MAXNUM_ATTRS <= (UINT8_MAX + 1));
5104 #endif

5106         if (vp == NULL) {
5107                 if (resp)
5108                         *resp = 0;
5109                 return (NFS4ERR_NOFILEHANDLE);
5110         }
5111         if (cs->access == CS_ACCESS_DENIED) {
5112                 if (resp)
5113                         *resp = 0;
5114                 return (NFS4ERR_ACCESS);
5115         }

5117         sargp->op = cmd;
5118         sargp->cs = cs;
5119         sargp->flag = 0;        /* may be set later */
5120         sargp->vap->va_mask = 0;
5121         sargp->rdattr_error = NFS4_OK;
5122         sargp->rdattr_error_req = FALSE;
5123         /* sargp->sbp is set by the caller */

5125         xdrmem_create(&xdr, attrs, attrslen, XDR_DECODE);

5127         na = ntovp->na;
5128         amap = ntovp->amap;

5130         /*
5131          * The following loop iterates on the nfs4_ntov_map checking
5132          * if the fbit is set in the requested bitmap.
5133          * If set then we process the arguments using the
5134          * rfs4_fattr4 conversion functions to populate the setattr
5135          * vattr and va_mask. Any settable attrs that are not using vattr
5136          * will be set in this loop.
5137          */
5138         for (i = 0; i < nfs4_ntov_map_size; i++) {
5139                 if (!(fattrp->attrmask & nfs4_ntov_map[i].fbit)) {
5140                         continue;
5141                 }
5142                 /*
5143                  * If setattr, must be a writable attr.
5144                  * If verify/nverify, must be a readable attr.
5145                  */
5146                 if ((error = (*nfs4_ntov_map[i].sv_getit)(
5147                     NFS4ATTR_SUPPORTED, sargp, NULL)) != 0) {
5148                         /*
5149                          * Client tries to set/verify an
5150                          * unsupported attribute, tries to set
5151                          * a read only attr or verify a write
5152                          * only one - error!
5153                          */
5154                         break;
5155                 }
5156                 /*
5157                  * Decode the attribute to set/verify
5158                  */
5159                 error = decode_fattr4_attr(cmd, sargp, nfs4_ntov_map[i].nval,
5160                     &xdr, resp ? resp : NULL, na);
5161                 if (error)
5162                         break;
5163                 *amap++ = (uint8_t)nfs4_ntov_map[i].nval;
5164                 na++;
```



```
5165                 (ntovp->attrcnt)++;
5166                 if (nfs4_ntov_map[i].vfsstat)
5167                         ntovp->vfsstat = TRUE;
5168         }

5170         if (error != 0)
5171                 status = (error == ENOTSUP ? NFS4ERR_ATTRNOTSUPP :
5172                     puterrno4(error));
5173         /* xdrmem_destroy(&xdrs); */    /* NO-OP */
5174         return (status);
5175 }

5177 static nfsstat4
5178 do_rfs4_op_setattr(bitmap4 *resp, fattr4 *fattrp, struct compound_state *cs,
5179     stateid4 *stateid)
5180 {
5181         int error = 0;
5182         struct nfs4_svgetit_arg sarg;
5183         bool_t trunc;

5185         nfsstat4 status = NFS4_OK;
5186         cred_t *cr = cs->cr;
5187         vnode_t *vp = cs->vp;
5188         struct nfs4_ntov_table ntov;
5189         struct statvfs64 sb;
5190         struct vattr bva;
5191         struct flock64 bf;
5192         int in_crit = 0;
5193         uint_t saved_mask = 0;
5194         caller_context_t ct;

5196         *resp = 0;
5197         sarg.sbp = &sb;
5198         sarg.is_referral = B_FALSE;
5199         nfs4_ntov_table_init(&ntov);
5200         status = do_rfs4_set_attrs(resp, fattrp, cs, &sarg, &ntov,
5201             NFS4ATTR_SETIT);
5202         if (status != NFS4_OK) {
5203                 /*
5204                  * failed set attrs
5205                  */
5206                 goto done;
5207         }
5208         if ((sarg.vap->va_mask == 0) &&
5209             (! (fattrp->attrmask & FATTR4_ACL_MASK))) {
5210                 /*
5211                  * no further work to be done
5212                  */
5213                 goto done;
5214         }

5216         /*
5217          * If we got a request to set the ACL and the MODE, only
5218          * allow changing VSUID, VSGID, and VSVTX.  Attempting
5219          * to change any other bits, along with setting an ACL,
5220          * gives NFS4ERR_INVAL.
5221          */
5222         if ((fattrp->attrmask & FATTR4_ACL_MASK) &&
5223             (fattrp->attrmask & FATTR4_MODE_MASK)) {
5224                 vattr_t va;

5226                 va.va_mask = AT_MODE;
5227                 error = VOP_GETATTR(vp, &va, 0, cs->cr, NULL);
5228                 if (error) {
5229                         status = puterrno4(error);
5230                         goto done;
```

```
5231                }
5232                if ((sarg.vap->va_mode ^ va.va_mode) &
5233                    ~(VSUID | VSGID | VSVTX)) {
5234                        status = NFS4ERR_INVAL;
5235                        goto done;
5236                }
5237        }

5239        /* Check stateid only if size has been set */
5240        if (sarg.vap->va_mask & AT_SIZE) {
5241                trunc = (sarg.vap->va_size == 0);
5242                status = rfs4_check_stateid(FWRITE, cs->vp, stateid,
5243                    trunc, &cs->deleg, sarg.vap->va_mask & AT_SIZE, &ct);
5244                if (status != NFS4_OK)
5245                        goto done;
5246        } else {
5247                ct.cc_sysid = 0;
5248                ct.cc_pid = 0;
5249                ct.cc_caller_id = nfs4_srv_caller_id;
5250                ct.cc_flags = CC_DONTBLOCK;
5251        }

5253        /* XXX start of possible race with delegations */

5255        /*
5256         * We need to specially handle size changes because it is
5257         * possible for the client to create a file with read-only
5258         * modes, but with the file opened for writing. If the client
5259         * then tries to set the file size, e.g. ftruncate(3C),
5260         * fcntl(F_FREESP), the normal access checking done in
5261         * VOP_SETATTR would prevent the client from doing it even though
5262         * it should be allowed to do so.  To get around this, we do the
5263         * access checking for ourselves and use VOP_SPACE which doesn't
5264         * do the access checking.
5265         * Also the client should not be allowed to change the file
5266         * size if there is a conflicting non-blocking mandatory lock in
5267         * the region of the change.
5268         */
5269        if (vp->v_type == VREG && (sarg.vap->va_mask & AT_SIZE)) {
5270                u_offset_t offset;
5271                ssize_t length;

5273                /*
5274                 * ufs_setattr clears AT_SIZE from vap->va_mask, but
5275                 * before returning, sarg.vap->va_mask is used to
5276                 * generate the setattr reply bitmap.  We also clear
5277                 * AT_SIZE below before calling VOP_SPACE.  For both
5278                 * of these cases, the va_mask needs to be saved here
5279                 * and restored after calling VOP_SETATTR.
5280                 */
5281                saved_mask = sarg.vap->va_mask;

5283                /*
5284                 * Check any possible conflict due to NBMAND locks.
5285                 * Get into critical region before VOP_GETATTR, so the
5286                 * size attribute is valid when checking conflicts.
5287                 */
5288                if (nbl_need_check(vp)) {
5289                        nbl_start_crit(vp, RW_READER);
5290                        in_crit = 1;
5291                }

5293                bva.va_mask = AT_UID|AT_SIZE;
5294                if (error = VOP_GETATTR(vp, &bva, 0, cr, &ct)) {
5295                        status = puterrno4(error);
5296                        goto done;
```



```
5297                }

5299                if (in_crit) {
5300                        if (sarg.vap->va_size < bva.va_size) {
5301                                offset = sarg.vap->va_size;
5302                                length = bva.va_size - sarg.vap->va_size;
5303                        } else {
5304                                offset = bva.va_size;
5305                                length = sarg.vap->va_size - bva.va_size;
5306                        }
5307                        if (nbl_conflict(vp, NBL_WRITE, offset, length, 0,
5308                            &ct)) {
5309                                status = NFS4ERR_LOCKED;
5310                                goto done;
5311                        }
5312                }

5314                if (crgetuid(cr) == bva.va_uid) {
5315                        sarg.vap->va_mask &= ~AT_SIZE;
5316                        bf.l_type = F_WRLCK;
5317                        bf.l_whence = 0;
5318                        bf.l_start = (off64_t)sarg.vap->va_size;
5319                        bf.l_len = 0;
5320                        bf.l_sysid = 0;
5321                        bf.l_pid = 0;
5322                        error = VOP_SPACE(vp, F_FREESP, &bf, FWRITE,
5323                            (offset_t)sarg.vap->va_size, cr, &ct);
5324                }
5325        }

5327        if (!error && sarg.vap->va_mask != 0)
5328                error = VOP_SETATTR(vp, sarg.vap, sarg.flag, cr, &ct);

5330        /* restore va_mask -- ufs_setattr clears AT_SIZE */
5331        if (saved_mask & AT_SIZE)
5332                sarg.vap->va_mask |= AT_SIZE;

5334        /*
5335         * If an ACL was being set, it has been delayed until now,
5336         * in order to set the mode (via the VOP_SETATTR() above) first.
5337         */
5338        if ((! error) && (fattrp->attrmask & FATTR4_ACL_MASK)) {
5339                int i;

5341                for (i = 0; i < NFS4_MAXNUM_ATTRS; i++)
5342                        if (ntov.amap[i] == FATTR4_ACL)
5343                                break;
5344                if (i < NFS4_MAXNUM_ATTRS) {
5345                        error = (*nfs4_ntov_map[FATTR4_ACL].sv_getit)(
5346                            NFS4ATTR_SETIT, &sarg, &ntov.na[i]);
5347                        if (error == 0) {
5348                                *resp |= FATTR4_ACL_MASK;
5349                        } else if (error == ENOTSUP) {
5350                                (void) rfs4_verify_attr(&sarg, resp, &ntov);
5351                                status = NFS4ERR_ATTRNOTSUPP;
5352                                goto done;
5353                        }
5354                } else {
5355                        NFS4_DEBUG(rfs4_debug,
5356                            (CE_NOTE, "do_rfs4_op_setattr: "
5357                            "unable to find ACL in fattr4"));
5358                        error = EINVAL;
5359                }
5360        }

5362        if (error) {
```

```
5363                 /* check if a monitor detected a delegation conflict */
5364                 if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK))
5365                         status = NFS4ERR_DELAY;
5366                 else
5367                         status = puterrno4(error);

5369                 /*
5370                  * Set the response bitmap when setattr failed.
5371                  * If VOP_SETATTR partially succeeded, test by doing a
5372                  * VOP_GETATTR on the object and comparing the data
5373                  * to the setattr arguments.
5374                  */
5375                 (void) rfs4_verify_attr(&sarg, resp, &ntov);
5376         } else {
5377                 /*
5378                  * Force modified metadata out to stable storage.
5379                  */
5380                 (void) VOP_FSYNC(vp, FNODSYNC, cr, &ct);
5381                 /*
5382                  * Set response bitmap
5383                  */
5384                 nfs4_vmask_to_nmask_set(sarg.vap->va_mask, resp);
5385         }

5387 /* Return early and already have a NFSv4 error */
5388 done:
5389         /*
5390          * Except for nfs4_vmask_to_nmask_set(), vattr --> fattr
5391          * conversion sets both readable and writeable NFS4 attrs
5392          * for AT_MTIME and AT_ATIME.  The line below masks out
5393          * unrequested attrs from the setattr result bitmap.  This
5394          * is placed after the done: label to catch the ATTRNOTSUP
5395          * case.
5396          */
5397         *resp &= fattrp->attrmask;

5399         if (in_crit)
5400                 nbl_end_crit(vp);

5402         nfs4_ntov_table_free(&ntov, &sarg);

5404         return (status);
5405 }

5407 /* ARGSUSED */
5408 static void
5409 rfs4_op_setattr(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
5410     struct compound_state *cs)
5411 {
5412         SETATTR4args *args = &argop->nfs_argop4_u.opsetattr;
5413         SETATTR4res *resp = &resop->nfs_resop4_u.opsetattr;
5414         bslabel_t *clabel;

5416         DTRACE_NFSV4_2(op__setattr__start, struct compound_state *, cs,
5417             SETATTR4args *, args);

5419         if (cs->vp == NULL) {
5420                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
5421                 goto out;
5422         }

5424         /*
5425          * If there is an unshared filesystem mounted on this vnode,
5426          * do not allow to setattr on this vnode.
5427          */
5428         if (vn_ismntpt(cs->vp)) {
```



```
5429                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
5430                 goto out;
5431         }

5433         resp->attrsset = 0;

5435         if (rdonly4(req, cs)) {
5436                 *cs->statusp = resp->status = NFS4ERR_ROFS;
5437                 goto out;
5438         }

5440         /* check label before setting attributes */
5441         if (is_system_labeled()) {
5442                 ASSERT(req->rq_label != NULL);
5443                 clabel = req->rq_label;
5444                 DTRACE_PROBE2(tx__rfs4__log__info__opsetattr__clabel, char *,
5445                     "got client label from request(1)",
5446                     struct svc_req *, req);
5447                 if (!blequal(&l_admin_low->tsl_label, clabel)) {
5448                         if (!do_rfs_label_check(clabel, cs->vp,
5449                             EQUALITY_CHECK, cs->exi)) {
5450                                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
5451                                 goto out;
5452                         }
5453                 }
5454         }

5456         *cs->statusp = resp->status =
5457             do_rfs4_op_setattr(&resp->attrsset, &args->obj_attributes, cs,
5458             &args->stateid);

5460 out:
5461         DTRACE_NFSV4_2(op__setattr__done, struct compound_state *, cs,
5462             SETATTR4res *, resp);
5463 }

5465 /* ARGSUSED */
5466 static void
5467 rfs4_op_verify(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
5468     struct compound_state *cs)
5469 {
5470         /*
5471          * verify and nverify are exactly the same, except that nverify
5472          * succeeds when some argument changed, and verify succeeds when
5473          * when none changed.
5474          */

5476         VERIFY4args  *args = &argop->nfs_argop4_u.opverify;
5477         VERIFY4res *resp = &resop->nfs_resop4_u.opverify;

5479         int error;
5480         struct nfs4_svgetit_arg sarg;
5481         struct statvfs64 sb;
5482         struct nfs4_ntov_table ntov;

5484         DTRACE_NFSV4_2(op__verify__start, struct compound_state *, cs,
5485             VERIFY4args *, args);

5487         if (cs->vp == NULL) {
5488                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
5489                 goto out;
5490         }

5492         sarg.sbp = &sb;
5493         sarg.is_referral = B_FALSE;
5494         nfs4_ntov_table_init(&ntov);
```

```
5495         resp->status = do_rfs4_set_attrs(NULL, &args->obj_attributes, cs,
5496             &sarg, &ntov, NFS4ATTR_VERIT);
5497         if (resp->status != NFS4_OK) {
5498                 /*
5499                  * do_rfs4_set_attrs will try to verify systemwide attrs,
5500                  * so could return -1 for "no match".
5501                  */
5502                 if (resp->status == -1)
5503                         resp->status = NFS4ERR_NOT_SAME;
5504                 goto done;
5505         }
5506         error = rfs4_verify_attr(&sarg, NULL, &ntov);
5507         switch (error) {
5508         case 0:
5509                 resp->status = NFS4_OK;
5510                 break;
5511         case -1:
5512                 resp->status = NFS4ERR_NOT_SAME;
5513                 break;
5514         default:
5515                 resp->status = puterrno4(error);
5516                 break;
5517         }
5518 done:
5519         *cs->statusp = resp->status;
5520         nfs4_ntov_table_free(&ntov, &sarg);
5521 out:
5522         DTRACE_NFSV4_2(op__verify__done, struct compound_state *, cs,
5523             VERIFY4res *, resp);
5524 }

5526 /* ARGSUSED */
5527 static void
5528 rfs4_op_nverify(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
5529     struct compound_state *cs)
5530 {
5531         /*
5532          * verify and nverify are exactly the same, except that nverify
5533          * succeeds when some argument changed, and verify succeeds when
5534          * when none changed.
5535          */

5537         NVERIFY4args  *args = &argop->nfs_argop4_u.opnverify;
5538         NVERIFY4res *resp = &resop->nfs_resop4_u.opnverify;

5540         int error;
5541         struct nfs4_svgetit_arg sarg;
5542         struct statvfs64 sb;
5543         struct nfs4_ntov_table ntov;

5545         DTRACE_NFSV4_2(op__nverify__start, struct compound_state *, cs,
5546             NVERIFY4args *, args);

5548         if (cs->vp == NULL) {
5549                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
5550                 DTRACE_NFSV4_2(op__nverify__done, struct compound_state *, cs,
5551                     NVERIFY4res *, resp);
5552                 return;
5553         }
5554         sarg.sbp = &sb;
5555         sarg.is_referral = B_FALSE;
5556         nfs4_ntov_table_init(&ntov);
5557         resp->status = do_rfs4_set_attrs(NULL, &args->obj_attributes, cs,
5558             &sarg, &ntov, NFS4ATTR_VERIT);
5559         if (resp->status != NFS4_OK) {
5560                 /*
```



```
5561                  * do_rfs4_set_attrs will try to verify systemwide attrs,
5562                  * so could return -1 for "no match".
5563                  */
5564                 if (resp->status == -1)
5565                         resp->status = NFS4_OK;
5566                 goto done;
5567         }
5568         error = rfs4_verify_attr(&sarg, NULL, &ntov);
5569         switch (error) {
5570         case 0:
5571                 resp->status = NFS4ERR_SAME;
5572                 break;
5573         case -1:
5574                 resp->status = NFS4_OK;
5575                 break;
5576         default:
5577                 resp->status = puterrno4(error);
5578                 break;
5579         }
5580 done:
5581         *cs->statusp = resp->status;
5582         nfs4_ntov_table_free(&ntov, &sarg);

5584         DTRACE_NFSV4_2(op__nverify__done, struct compound_state *, cs,
5585             NVERIFY4res *, resp);
5586 }

5588 /*
5589  * XXX - This should live in an NFS header file.
5590  */
5591 #define MAX_IOVECS      12

5593 /* ARGSUSED */
5594 static void
5595 rfs4_op_write(nfs_argop4 *argop, nfs_resop4 *resop, struct svc_req *req,
5596     struct compound_state *cs)
5597 {
5598         WRITE4args *args = &argop->nfs_argop4_u.opwrite;
5599         WRITE4res *resp = &resop->nfs_resop4_u.opwrite;
5600         int error;
5601         vnode_t *vp;
5602         struct vattr bva;
5603         u_offset_t rlimit;
5604         struct uio uio;
5605         struct iovec iov[MAX_IOVECS];
5606         struct iovec *iovp;
5607         int iovcnt;
5608         int ioflag;
5609         cred_t *savecred, *cr;
5610         bool_t *deleg = &cs->deleg;
5611         nfsstat4 stat;
5612         int in_crit = 0;
5613         caller_context_t ct;

5615         DTRACE_NFSV4_2(op__write__start, struct compound_state *, cs,
5616             WRITE4args *, args);

5618         vp = cs->vp;
5619         if (vp == NULL) {
5620                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
5621                 goto out;
5622         }
5623         if (cs->access == CS_ACCESS_DENIED) {
5624                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
5625                 goto out;
5626         }
```

```
5628         cr = cs->cr;

5630         if ((stat = rfs4_check_stateid(FWRITE, vp, &args->stateid, FALSE,
5631             deleg, TRUE, &ct)) != NFS4_OK) {
5632                 *cs->statusp = resp->status = stat;
5633                 goto out;
5634         }

5636         /*
5637          * We have to enter the critical region before calling VOP_RWLOCK
5638          * to avoid a deadlock with ufs.
5639          */
5640         if (nbl_need_check(vp)) {
5641                 nbl_start_crit(vp, RW_READER);
5642                 in_crit = 1;
5643                 if (nbl_conflict(vp, NBL_WRITE,
5644                     args->offset, args->data_len, 0, &ct)) {
5645                         *cs->statusp = resp->status = NFS4ERR_LOCKED;
5646                         goto out;
5647                 }
5648         }

5650         bva.va_mask = AT_MODE | AT_UID;
5651         error = VOP_GETATTR(vp, &bva, 0, cr, &ct);

5653         /*
5654          * If we can't get the attributes, then we can't do the
5655          * right access checking.  So, we'll fail the request.
5656          */
5657         if (error) {
5658                 *cs->statusp = resp->status = puterrno4(error);
5659                 goto out;
5660         }

5662         if (rdonly4(req, cs)) {
5663                 *cs->statusp = resp->status = NFS4ERR_ROFS;
5664                 goto out;
5665         }

5667         if (vp->v_type != VREG) {
5668                 *cs->statusp = resp->status =
5669                     ((vp->v_type == VDIR) ? NFS4ERR_ISDIR : NFS4ERR_INVAL);
5670                 goto out;
5671         }

5673         if (crgetuid(cr) != bva.va_uid &&
5674             (error = VOP_ACCESS(vp, VWRITE, 0, cr, &ct))) {
5675                 *cs->statusp = resp->status = puterrno4(error);
5676                 goto out;
5677         }

5679         if (MANDLOCK(vp, bva.va_mode)) {
5680                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
5681                 goto out;
5682         }

5684         if (args->data_len == 0) {
5685                 *cs->statusp = resp->status = NFS4_OK;
5686                 resp->count = 0;
5687                 resp->committed = args->stable;
5688                 resp->writeverf = Write4verf;
5689                 goto out;
5690         }

5692         if (args->mblk != NULL) {
```



```
5693                 mblk_t *m;
5694                 uint_t bytes, round_len;

5696                 iovcnt = 0;
5697                 bytes = 0;
5698                 round_len = roundup(args->data_len, BYTES_PER_XDR_UNIT);
5699                 for (m = args->mblk;
5700                     m != NULL && bytes < round_len;
5701                     m = m->b_cont) {
5702                         iovcnt++;
5703                         bytes += MBLKL(m);
5704                 }
5705 #ifdef DEBUG
5706                 /* should have ended on an mblk boundary */
5707                 if (bytes != round_len) {
5708                         printf("bytes=0x%x, round_len=0x%x, req len=0x%x\n",
5709                             bytes, round_len, args->data_len);
5710                         printf("args=%p, args->mblk=%p, m=%p", (void *)args,
5711                             (void *)args->mblk, (void *)m);
5712                         ASSERT(bytes == round_len);
5713                 }
5714 #endif
5715                 if (iovcnt <= MAX_IOVECS) {
5716                         iovp = iov;
5717                 } else {
5718                         iovp = kmem_alloc(sizeof (*iovp) * iovcnt, KM_SLEEP);
5719                 }
5720                 mblk_to_iov(args->mblk, iovcnt, iovp);
5721         } else if (args->rlist != NULL) {
5722                 iovcnt = 1;
5723                 iovp = iov;
5724                 iovp->iov_base = (char *)((args->rlist)->u.c_daddr3);
5725                 iovp->iov_len = args->data_len;
5726         } else {
5727                 iovcnt = 1;
5728                 iovp = iov;
5729                 iovp->iov_base = args->data_val;
5730                 iovp->iov_len = args->data_len;
5731         }

5733         uio.uio_iov = iovp;
5734         uio.uio_iovcnt = iovcnt;

5736         uio.uio_segflg = UIO_SYSSPACE;
5737         uio.uio_extflg = UIO_COPY_DEFAULT;
5738         uio.uio_loffset = args->offset;
5739         uio.uio_resid = args->data_len;
5740         uio.uio_llimit = curproc->p_fsz_ctl;
5741         rlimit = uio.uio_llimit - args->offset;
5742         if (rlimit < (u_offset_t)uio.uio_resid)
5743                 uio.uio_resid = (int)rlimit;

5745         if (args->stable == UNSTABLE4)
5746                 ioflag = 0;
5747         else if (args->stable == FILE_SYNC4)
5748                 ioflag = FSYNC;
5749         else if (args->stable == DATA_SYNC4)
5750                 ioflag = FDSYNC;
5751         else {
5752                 if (iovp != iov)
5753                         kmem_free(iovp, sizeof (*iovp) * iovcnt);
5754                 *cs->statusp = resp->status = NFS4ERR_INVAL;
5755                 goto out;
5756         }

5758         /*
```

```
5759          * We're changing creds because VM may fault and we need
5760          * the cred of the current thread to be used if quota
5761          * checking is enabled.
5762          */
5763         savecred = curthread->t_cred;
5764         curthread->t_cred = cr;
5765         error = do_io(FWRITE, vp, &uio, ioflag, cr, &ct);
5766         curthread->t_cred = savecred;

5768         if (iovp != iov)
5769                 kmem_free(iovp, sizeof (*iovp) * iovcnt);

5771         if (error) {
5772                 *cs->statusp = resp->status = puterrno4(error);
5773                 goto out;
5774         }

5776         *cs->statusp = resp->status = NFS4_OK;
5777         resp->count = args->data_len - uio.uio_resid;

5779         if (ioflag == 0)
5780                 resp->committed = UNSTABLE4;
5781         else
5782                 resp->committed = FILE_SYNC4;

5784         resp->writeverf = Write4verf;

5786 out:
5787         if (in_crit)
5788                 nbl_end_crit(vp);

5790         DTRACE_NFSV4_2(op__write__done, struct compound_state *, cs,
5791             WRITE4res *, resp);
5792 }


5795 /* XXX put in a header file */
5796 extern int      sec_svc_getcred(struct svc_req *, cred_t *,  caddr_t *, int *);

5798 void
5799 rfs4_compound(COMPOUND4args *args, COMPOUND4res *resp, struct exportinfo *exi,
5800     struct svc_req *req, cred_t *cr, int *rv)
5801 {
5802         uint_t i;
5803         struct compound_state cs;

5805         if (rv != NULL)
5806                 *rv = 0;
5807         rfs4_init_compound_state(&cs);
5808         /*
5809          * Form a reply tag by copying over the reqeuest tag.
5810          */
5811         resp->tag.utf8string_val =
5812             kmem_alloc(args->tag.utf8string_len, KM_SLEEP);
5813         resp->tag.utf8string_len = args->tag.utf8string_len;
5814         bcopy(args->tag.utf8string_val, resp->tag.utf8string_val,
5815             resp->tag.utf8string_len);

5817         cs.statusp = &resp->status;
5818         cs.req = req;
5819         resp->array = NULL;
5820         resp->array_len = 0;

5822         /*
5823          * XXX for now, minorversion should be zero
5824          */
```



```
5825         if (args->minorversion != NFS4_MINORVERSION) {
5826                 DTRACE_NFSV4_2(compound__start, struct compound_state *,
5827                     &cs, COMPOUND4args *, args);
5828                 resp->status = NFS4ERR_MINOR_VERS_MISMATCH;
5829                 DTRACE_NFSV4_2(compound__done, struct compound_state *,
5830                     &cs, COMPOUND4res *, resp);
5831                 return;
5832         }

5834         if (args->array_len == 0) {
5835                 resp->status = NFS4_OK;
5836                 return;
5837         }

5839         ASSERT(exi == NULL);
5840         ASSERT(cr == NULL);

5842         cr = crget();
5843         ASSERT(cr != NULL);

5845         if (sec_svc_getcred(req, cr, &cs.principal, &cs.nfsflavor) == 0) {
5846                 DTRACE_NFSV4_2(compound__start, struct compound_state *,
5847                     &cs, COMPOUND4args *, args);
5848                 crfree(cr);
5849                 DTRACE_NFSV4_2(compound__done, struct compound_state *,
5850                     &cs, COMPOUND4res *, resp);
5851                 svcerr_badcred(req->rq_xprt);
5852                 if (rv != NULL)
5853                         *rv = 1;
5854                 return;
5855         }
5856         resp->array_len = args->array_len;
5857         resp->array = kmem_zalloc(args->array_len * sizeof (nfs_resop4),
5858             KM_SLEEP);

5860         cs.basecr = cr;

5862         DTRACE_NFSV4_2(compound__start, struct compound_state *, &cs,
5863             COMPOUND4args *, args);

5865         /*
2741          * For now, NFS4 compound processing must be protected by
2742          * exported_lock because it can access more than one exportinfo
2743          * per compound and share/unshare can now change multiple
2744          * exinfo structs.  The NFS2/3 code only refs 1 exportinfo
2745          * per proc (excluding public exinfo), and exi_count design
2746          * is sufficient to protect concurrent execution of NFS2/3
2747          * ops along with unexport.  This lock will be removed as
2748          * part of the NFSv4 phase 2 namespace redesign work.
2749          */
2750         rw_enter(&exported_lock, RW_READER);

2752         /*
5866          * If this is the first compound we've seen, we need to start all
5867          * new instances' grace periods.
5868          */
5869         if (rfs4_seen_first_compound == 0) {
5870                 rfs4_grace_start_new();
5871                 /*
5872                  * This must be set after rfs4_grace_start_new(), otherwise
5873                  * another thread could proceed past here before the former
5874                  * is finished.
5875                  */
5876                 rfs4_seen_first_compound = 1;
5877         }
```

```
5879         for (i = 0; i < args->array_len && cs.cont; i++) {
5880                 nfs_argop4 *argop;
5881                 nfs_resop4 *resop;
5882                 uint_t op;

5884                 argop = &args->array[i];
5885                 resop = &resp->array[i];
5886                 resop->resop = argop->argop;
5887                 op = (uint_t)resop->resop;

5889                 if (op < rfsv4disp_cnt) {
5890                         /*
5891                          * Count the individual ops here; NULL and COMPOUND
5892                          * are counted in common_dispatch()
5893                          */
5894                         rfsproccnt_v4_ptr[op].value.ui64++;

5896                         NFS4_DEBUG(rfs4_debug > 1,
5897                             (CE_NOTE, "Executing %s", rfs4_op_string[op]));
5898                         (*rfsv4disptab[op].dis_proc)(argop, resop, req, &cs);
5899                         NFS4_DEBUG(rfs4_debug > 1, (CE_NOTE, "%s returned %d",
5900                             rfs4_op_string[op], *cs.statusp));
5901                         if (*cs.statusp != NFS4_OK)
5902                                 cs.cont = FALSE;
5903                 } else {
5904                         /*
5905                          * This is effectively dead code since XDR code
5906                          * will have already returned BADXDR if op doesn't
5907                          * decode to legal value.  This only done for a
5908                          * day when XDR code doesn't verify v4 opcodes.
5909                          */
5910                         op = OP_ILLEGAL;
5911                         rfsproccnt_v4_ptr[OP_ILLEGAL_IDX].value.ui64++;

5913                         rfs4_op_illegal(argop, resop, req, &cs);
5914                         cs.cont = FALSE;
5915                 }

5917                 /*
5918                  * If not at last op, and if we are to stop, then
5919                  * compact the results array.
5920                  */
5921                 if ((i + 1) < args->array_len && !cs.cont) {
5922                         nfs_resop4 *new_res = kmem_alloc(
5923                             (i+1) * sizeof (nfs_resop4), KM_SLEEP);
5924                         bcopy(resp->array,
5925                             new_res, (i+1) * sizeof (nfs_resop4));
5926                         kmem_free(resp->array,
5927                             args->array_len * sizeof (nfs_resop4));

5929                         resp->array_len =  i + 1;
5930                         resp->array = new_res;
5931                 }
5932         }

2821         rw_exit(&exported_lock);

5935         DTRACE_NFSV4_2(compound__done, struct compound_state *, &cs,
5936             COMPOUND4res *, resp);

5938         if (cs.exi)
5939                 exi_rele(cs.exi);
5940         if (cs.saved_exi)
5941                 exi_rele(cs.saved_exi);
5942 #endif /* ! codereview */
5943         if (cs.vp)
```



```
5944                 VN_RELE(cs.vp);
5945         if (cs.saved_vp)
5946                 VN_RELE(cs.saved_vp);
5947         if (cs.saved_fh.nfs_fh4_val)
5948                 kmem_free(cs.saved_fh.nfs_fh4_val, NFS4_FHSIZE);

5950         if (cs.basecr)
5951                 crfree(cs.basecr);
5952         if (cs.cr)
5953                 crfree(cs.cr);
5954         /*
5955          * done with this compound request, free the label
5956          */

5958         if (req->rq_label != NULL) {
5959                 kmem_free(req->rq_label, sizeof (bslabel_t));
5960                 req->rq_label = NULL;
5961         }
5962 }

5964 /*
5965  * XXX because of what appears to be duplicate calls to rfs4_compound_free
5966  * XXX zero out the tag and array values. Need to investigate why the
5967  * XXX calls occur, but at least prevent the panic for now.
5968  */
5969 void
5970 rfs4_compound_free(COMPOUND4res *resp)
5971 {
5972         uint_t i;

5974         if (resp->tag.utf8string_val) {
5975                 UTF8STRING_FREE(resp->tag)
5976         }

5978         for (i = 0; i < resp->array_len; i++) {
5979                 nfs_resop4 *resop;
5980                 uint_t op;

5982                 resop = &resp->array[i];
5983                 op = (uint_t)resop->resop;
5984                 if (op < rfsv4disp_cnt) {
5985                         (*rfsv4disptab[op].dis_resfree)(resop);
5986                 }
5987         }
5988         if (resp->array != NULL) {
5989                 kmem_free(resp->array, resp->array_len * sizeof (nfs_resop4));
5990         }
5991 }

5993 /*
5994  * Process the value of the compound request rpc flags, as a bit-AND
5995  * of the individual per-op flags (idempotent, allowork, publicfh_ok)
5996  */
5997 void
5998 rfs4_compound_flagproc(COMPOUND4args *args, int *flagp)
5999 {
6000         int i;
6001         int flag = RPC_ALL;

6003         for (i = 0; flag && i < args->array_len; i++) {
6004                 uint_t op;

6006                 op = (uint_t)args->array[i].argop;

6008                 if (op < rfsv4disp_cnt)
6009                         flag &= rfsv4disptab[op].dis_flags;
```

```
6010                 else
6011                         flag = 0;
6012         }
6013         *flagp = flag;
6014 }

6016 nfsstat4
6017 rfs4_client_sysid(rfs4_client_t *cp, sysid_t *sp)
6018 {
6019         nfsstat4 e;

6021         rfs4_dbe_lock(cp->rc_dbe);

6023         if (cp->rc_sysidt != LM_NOSYSID) {
6024                 *sp = cp->rc_sysidt;
6025                 e = NFS4_OK;

6027         } else if ((cp->rc_sysidt = lm_alloc_sysidt()) != LM_NOSYSID) {
6028                 *sp = cp->rc_sysidt;
6029                 e = NFS4_OK;

6031                 NFS4_DEBUG(rfs4_debug, (CE_NOTE,
6032                     "rfs4_client_sysid: allocated 0x%x\n", *sp));
6033         } else
6034                 e = NFS4ERR_DELAY;

6036         rfs4_dbe_unlock(cp->rc_dbe);
6037         return (e);
6038 }

6040 #if defined(DEBUG) && ! defined(lint)
6041 static void lock_print(char *str, int operation, struct flock64 *flk)
6042 {
6043         char *op, *type;

6045         switch (operation) {
6046         case F_GETLK: op = "F_GETLK";
6047                 break;
6048         case F_SETLK: op = "F_SETLK";
6049                 break;
6050         case F_SETLK_NBMAND: op = "F_SETLK_NBMAND";
6051                 break;
6052         default: op = "F_UNKNOWN";
6053                 break;
6054         }
6055         switch (flk->l_type) {
6056         case F_UNLCK: type = "F_UNLCK";
6057                 break;
6058         case F_RDLCK: type = "F_RDLCK";
6059                 break;
6060         case F_WRLCK: type = "F_WRLCK";
6061                 break;
6062         default: type = "F_UNKNOWN";
6063                 break;
6064         }

6066         ASSERT(flk->l_whence == 0);
6067         cmn_err(CE_NOTE, "%s:  %s, type = %s, off = %llx len = %llx pid = %d",
6068             str, op, type, (longlong_t)flk->l_start,
6069             flk->l_len ? (longlong_t)flk->l_len : ~0LL, flk->l_pid);
6070 }

6072 #define LOCK_PRINT(d, s, t, f) if (d) lock_print(s, t, f)
6073 #else
6074 #define LOCK_PRINT(d, s, t, f)
6075 #endif
```



```
6077 /*ARGSUSED*/
6078 static bool_t
6079 creds_ok(cred_set_t cr_set, struct svc_req *req, struct compound_state *cs)
6080 {
6081         return (TRUE);
6082 }

6084 /*
6085  * Look up the pathname using the vp in cs as the directory vnode.
6086  * cs->vp will be the vnode for the file on success
6087  */

6089 static nfsstat4
6090 rfs4_lookup(component4 *component, struct svc_req *req,
6091     struct compound_state *cs)
6092 {
6093         char *nm;
6094         uint32_t len;
6095         nfsstat4 status;
6096         struct sockaddr *ca;
6097         char *name;

6099         if (cs->vp == NULL) {
6100                 return (NFS4ERR_NOFILEHANDLE);
6101         }
6102         if (cs->vp->v_type != VDIR) {
6103                 return (NFS4ERR_NOTDIR);
6104         }

6106         status = utf8_dir_verify(component);
6107         if (status != NFS4_OK)
6108                 return (status);

6110         nm = utf8_to_fn(component, &len, NULL);
6111         if (nm == NULL) {
6112                 return (NFS4ERR_INVAL);
6113         }

6115         if (len > MAXNAMELEN) {
6116                 kmem_free(nm, len);
6117                 return (NFS4ERR_NAMETOOLONG);
6118         }

6120         ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
6121         name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND,
6122             MAXPATHLEN + 1);

6124         if (name == NULL) {
6125                 kmem_free(nm, len);
6126                 return (NFS4ERR_INVAL);
6127         }

6129         status = do_rfs4_op_lookup(name, req, cs);

6131         if (name != nm)
6132                 kmem_free(name, MAXPATHLEN + 1);

6134         kmem_free(nm, len);

6136         return (status);
6137 }

6139 static nfsstat4
6140 rfs4_lookupfile(component4 *component, struct svc_req *req,
6141     struct compound_state *cs, uint32_t access, change_info4 *cinfo)
```

```
6142 {
6143         nfsstat4 status;
6144         vnode_t *dvp = cs->vp;
6145         vattr_t bva, ava, fva;
6146         int error;
6147
6148         /* Get "before" change value */
6149         bva.va_mask = AT_CTIME|AT_SEQ;
6150         error = VOP_GETATTR(dvp, &bva, 0, cs->cr, NULL);
6151         if (error)
6152                 return (puterrno4(error));
6153
6154         /* rfs4_lookup may VN_RELE directory */
6155         VN_HOLD(dvp);
6156
6157         status = rfs4_lookup(component, req, cs);
6158         if (status != NFS4_OK) {
6159                 VN_RELE(dvp);
6160                 return (status);
6161         }
6162
6163         /*
6164          * Get "after" change value, if it fails, simply return the
6165          * before value.
6166          */
6167         ava.va_mask = AT_CTIME|AT_SEQ;
6168         if (VOP_GETATTR(dvp, &ava, 0, cs->cr, NULL)) {
6169                 ava.va_ctime = bva.va_ctime;
6170                 ava.va_seq = 0;
6171         }
6172         VN_RELE(dvp);
6173
6174         /*
6175          * Validate the file is a file
6176          */
6177         fva.va_mask = AT_TYPE|AT_MODE;
6178         error = VOP_GETATTR(cs->vp, &fva, 0, cs->cr, NULL);
6179         if (error)
6180                 return (puterrno4(error));
6181
6182         if (fva.va_type != VREG) {
6183                 if (fva.va_type == VDIR)
6184                         return (NFS4ERR_ISDIR);
6185                 if (fva.va_type == VLNK)
6186                         return (NFS4ERR_SYMLINK);
6187                 return (NFS4ERR_INVAL);
6188         }
6189
6190         NFS4_SET_FATTR4_CHANGE(cinfo->before, bva.va_ctime);
6191         NFS4_SET_FATTR4_CHANGE(cinfo->after, ava.va_ctime);
6192
6193         /*
6194          * It is undefined if VOP_LOOKUP will change va_seq, so
6195          * cinfo.atomic = TRUE only if we have
6196          * non-zero va_seq's, and they have not changed.
6197          */
6198         if (bva.va_seq && ava.va_seq && ava.va_seq == bva.va_seq)
6199                 cinfo->atomic = TRUE;
6200         else
6201                 cinfo->atomic = FALSE;
6202
6203         /* Check for mandatory locking */
6204         cs->mandlock = MANDLOCK(cs->vp, fva.va_mode);
6205         return (check_open_access(access, cs, req));
6206 }
```



```
6208 static nfsstat4
6209 create_vnode(vnode_t *dvp, char *nm,  vattr_t *vap, createmode4 mode,
6210     timespec32_t *mtime, cred_t *cr, vnode_t **vpp, bool_t *created)
6211 {
6212         int error;
6213         nfsstat4 status = NFS4_OK;
6214         vattr_t va;
6215
6216 tryagain:
6217
6218         /*
6219          * The file open mode used is VWRITE.  If the client needs
6220          * some other semantic, then it should do the access checking
6221          * itself.  It would have been nice to have the file open mode
6222          * passed as part of the arguments.
6223          */
6224
6225         *created = TRUE;
6226         error = VOP_CREATE(dvp, nm, vap, EXCL, VWRITE, vpp, cr, 0, NULL, NULL);
6227
6228         if (error) {
6229                 *created = FALSE;
6230
6231                 /*
6232                  * If we got something other than file already exists
6233                  * then just return this error.  Otherwise, we got
6234                  * EEXIST.  If we were doing a GUARDED create, then
6235                  * just return this error.  Otherwise, we need to
6236                  * make sure that this wasn't a duplicate of an
6237                  * exclusive create request.
6238                  *
6239                  * The assumption is made that a non-exclusive create
6240                  * request will never return EEXIST.
6241                  */
6242
6243                 if (error != EEXIST || mode == GUARDED4) {
6244                         status = puterrno4(error);
6245                         return (status);
6246                 }
6247                 error = VOP_LOOKUP(dvp, nm, vpp, NULL, 0, NULL, cr,
6248                     NULL, NULL, NULL);
6249
6250                 if (error) {
6251                         /*
6252                          * We couldn't find the file that we thought that
6253                          * we just created.  So, we'll just try creating
6254                          * it again.
6255                          */
6256                         if (error == ENOENT)
6257                                 goto tryagain;
6258
6259                         status = puterrno4(error);
6260                         return (status);
6261                 }
6262
6263                 if (mode == UNCHECKED4) {
6264                         /* existing object must be regular file */
6265                         if ((*vpp)->v_type != VREG) {
6266                                 if ((*vpp)->v_type == VDIR)
6267                                         status = NFS4ERR_ISDIR;
6268                                 else if ((*vpp)->v_type == VLNK)
6269                                         status = NFS4ERR_SYMLINK;
6270                                 else
6271                                         status = NFS4ERR_INVAL;
6272                                 VN_RELE(*vpp);
6273                                 return (status);
```

```
6274                        }

6276                        return (NFS4_OK);
6277                }

6279                /* Check for duplicate request */
6280                ASSERT(mtime != 0);
6281                va.va_mask = AT_MTIME;
6282                error = VOP_GETATTR(*vpp, &va, 0, cr, NULL);
6283                if (!error) {
6284                        /* We found the file */
6285                        if (va.va_mtime.tv_sec != mtime->tv_sec ||
6286                            va.va_mtime.tv_nsec != mtime->tv_nsec) {
6287                                /* but its not our creation */
6288                                VN_RELE(*vpp);
6289                                return (NFS4ERR_EXIST);
6290                        }
6291                        *created = TRUE; /* retrans of create == created */
6292                        return (NFS4_OK);
6293                }
6294                VN_RELE(*vpp);
6295                return (NFS4ERR_EXIST);
6296        }

6298        return (NFS4_OK);
6299 }

6301 static nfsstat4
6302 check_open_access(uint32_t access, struct compound_state *cs,
6303     struct svc_req *req)
6304 {
6305        int error;
6306        vnode_t *vp;
6307        bool_t readonly;
6308        cred_t *cr = cs->cr;

6310        /* For now we don't allow mandatory locking as per V2/V3 */
6311        if (cs->access == CS_ACCESS_DENIED || cs->mandlock) {
6312                return (NFS4ERR_ACCESS);
6313        }

6315        vp = cs->vp;
6316        ASSERT(cr != NULL && vp->v_type == VREG);

6318        /*
6319         * If the file system is exported read only and we are trying
6320         * to open for write, then return NFS4ERR_ROFS
6321         */

6323        readonly = rdonly4(req, cs);

6325        if ((access & OPEN4_SHARE_ACCESS_WRITE) && readonly)
6326                return (NFS4ERR_ROFS);

6328        if (access & OPEN4_SHARE_ACCESS_READ) {
6329                if ((VOP_ACCESS(vp, VREAD, 0, cr, NULL) != 0) &&
6330                    (VOP_ACCESS(vp, VEXEC, 0, cr, NULL) != 0)) {
6331                        return (NFS4ERR_ACCESS);
6332                }
6333        }

6335        if (access & OPEN4_SHARE_ACCESS_WRITE) {
6336                error = VOP_ACCESS(vp, VWRITE, 0, cr, NULL);
6337                if (error)
6338                        return (NFS4ERR_ACCESS);
6339        }
```



```
6341        return (NFS4_OK);
6342 }

6344 static nfsstat4
6345 rfs4_createfile(OPEN4args *args, struct svc_req *req, struct compound_state *cs,
6346     change_info4 *cinfo, bitmap4 *attrset, clientid4 clientid)
6347 {
6348        struct nfs4_svgetit_arg sarg;
6349        struct nfs4_ntov_table ntov;

6351        bool_t ntov_table_init = FALSE;
6352        struct statvfs64 sb;
6353        nfsstat4 status;
6354        vnode_t *vp;
6355        vattr_t bva, ava, iva, cva, *vap;
6356        vnode_t *dvp;
6357        timespec32_t *mtime;
6358        char *nm = NULL;
6359        uint_t buflen;
6360        bool_t created;
6361        bool_t setsize = FALSE;
6362        len_t reqsize;
6363        int error;
6364        bool_t trunc;
6365        caller_context_t ct;
6366        component4 *component;
6367        bslabel_t *clabel;
6368        struct sockaddr *ca;
6369        char *name = NULL;

6371        sarg.sbp = &sb;
6372        sarg.is_referral = B_FALSE;

6374        dvp = cs->vp;

6376        /* Check if the file system is read only */
6377        if (rdonly4(req, cs))
6378                return (NFS4ERR_ROFS);

6380        /* check the label of including directory */
6381        if (is_system_labeled()) {
6382                ASSERT(req->rq_label != NULL);
6383                clabel = req->rq_label;
6384                DTRACE_PROBE2(tx__rfs4__log__info__opremove__clabel, char *,
6385                    "got client label from request(1)",
6386                    struct svc_req *, req);
6387                if (!blequal(&l_admin_low->tsl_label, clabel)) {
6388                        if (!do_rfs_label_check(clabel, dvp, EQUALITY_CHECK,
6389                            cs->exi)) {
6390                                return (NFS4ERR_ACCESS);
6391                        }
6392                }
6393        }

6395        /*
6396         * Get the last component of path name in nm. cs will reference
6397         * the including directory on success.
6398         */
6399        component = &args->open_claim4_u.file;
6400        status = utf8_dir_verify(component);
6401        if (status != NFS4_OK)
6402                return (status);

6404        nm = utf8_to_fn(component, &buflen, NULL);
```

```
6406         if (nm == NULL)
6407                 return (NFS4ERR_RESOURCE);

6409         if (buflen > MAXNAMELEN) {
6410                 kmem_free(nm, buflen);
6411                 return (NFS4ERR_NAMETOOLONG);
6412         }

6414         bva.va_mask = AT_TYPE|AT_CTIME|AT_SEQ;
6415         error = VOP_GETATTR(dvp, &bva, 0, cs->cr, NULL);
6416         if (error) {
6417                 kmem_free(nm, buflen);
6418                 return (puterrno4(error));
6419         }

6421         if (bva.va_type != VDIR) {
6422                 kmem_free(nm, buflen);
6423                 return (NFS4ERR_NOTDIR);
6424         }

6426         NFS4_SET_FATTR4_CHANGE(cinfo->before, bva.va_ctime)

6428         switch (args->mode) {
6429         case GUARDED4:
6430                 /*FALLTHROUGH*/
6431         case UNCHECKED4:
6432                 nfs4_ntov_table_init(&ntov);
6433                 ntov_table_init = TRUE;

6435                 *attrset = 0;
6436                 status = do_rfs4_set_attrs(attrset,
6437                     &args->createhow4_u.createattrs,
6438                     cs, &sarg, &ntov, NFS4ATTR_SETIT);

6440                 if (status == NFS4_OK && (sarg.vap->va_mask & AT_TYPE) &&
6441                     sarg.vap->va_type != VREG) {
6442                         if (sarg.vap->va_type == VDIR)
6443                                 status = NFS4ERR_ISDIR;
6444                         else if (sarg.vap->va_type == VLNK)
6445                                 status = NFS4ERR_SYMLINK;
6446                         else
6447                                 status = NFS4ERR_INVAL;
6448                 }

6450                 if (status != NFS4_OK) {
6451                         kmem_free(nm, buflen);
6452                         nfs4_ntov_table_free(&ntov, &sarg);
6453                         *attrset = 0;
6454                         return (status);
6455                 }

6457                 vap = sarg.vap;
6458                 vap->va_type = VREG;
6459                 vap->va_mask |= AT_TYPE;

6461                 if ((vap->va_mask & AT_MODE) == 0) {
6462                         vap->va_mask |= AT_MODE;
6463                         vap->va_mode = (mode_t)0600;
6464                 }

6466                 if (vap->va_mask & AT_SIZE) {

6468                         /* Disallow create with a non-zero size */

6470                         if ((reqsize = sarg.vap->va_size) != 0) {
6471                                 kmem_free(nm, buflen);
```



```
6472                                 nfs4_ntov_table_free(&ntov, &sarg);
6473                                 *attrset = 0;
6474                                 return (NFS4ERR_INVAL);
6475                         }
6476                         setsize = TRUE;
6477                 }
6478                 break;

6480         case EXCLUSIVE4:
6481                 /* prohibit EXCL create of named attributes */
6482                 if (dvp->v_flag & V_XATTRDIR) {
6483                         kmem_free(nm, buflen);
6484                         *attrset = 0;
6485                         return (NFS4ERR_INVAL);
6486                 }

6488                 cva.va_mask = AT_TYPE | AT_MTIME | AT_MODE;
6489                 cva.va_type = VREG;
6490                 /*
6491                  * Ensure no time overflows. Assumes underlying
6492                  * filesystem supports at least 32 bits.
6493                  * Truncate nsec to usec resolution to allow valid
6494                  * compares even if the underlying filesystem truncates.
6495                  */
6496                 mtime = (timespec32_t *)&args->createhow4_u.createverf;
6497                 cva.va_mtime.tv_sec = mtime->tv_sec % TIME32_MAX;
6498                 cva.va_mtime.tv_nsec = (mtime->tv_nsec / 1000) * 1000;
6499                 cva.va_mode = (mode_t)0;
6500                 vap = &cva;

6502                 /*
6503                  * For EXCL create, attrset is set to the server attr
6504                  * used to cache the client's verifier.
6505                  */
6506                 *attrset = FATTR4_TIME_MODIFY_MASK;
6507                 break;
6508         }

6510         ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;
6511         name = nfscmd_convname(ca, cs->exi, nm, NFSCMD_CONV_INBOUND,
6512             MAXPATHLEN  + 1);

6514         if (name == NULL) {
6515                 kmem_free(nm, buflen);
6516                 return (NFS4ERR_SERVERFAULT);
6517         }

6519         status = create_vnode(dvp, name, vap, args->mode, mtime,
6520             cs->cr, &vp, &created);
6521         if (nm != name)
6522                 kmem_free(name, MAXPATHLEN + 1);
6523         kmem_free(nm, buflen);

6525         if (status != NFS4_OK) {
6526                 if (ntov_table_init)
6527                         nfs4_ntov_table_free(&ntov, &sarg);
6528                 *attrset = 0;
6529                 return (status);
6530         }

6532         trunc = (setsize && !created);

6534         if (args->mode != EXCLUSIVE4) {
6535                 bitmap4 createmask = args->createhow4_u.createattrs.attrmask;

6537                 /*
```

```
6538                 * True verification that object was created with correct
6539                 * attrs is impossible.  The attrs could have been changed
6540                 * immediately after object creation.  If attributes did
6541                 * not verify, the only recourse for the server is to
6542                 * destroy the object.  Maybe if some attrs (like gid)
6543                 * are set incorrectly, the object should be destroyed;
6544                 * however, seems bad as a default policy.  Do we really
6545                 * want to destroy an object over one of the times not
6546                 * verifying correctly?  For these reasons, the server
6547                 * currently sets bits in attrset for createattrs
6548                 * that were set; however, no verification is done.
6549                 *
6550                 * vmask_to_nmask accounts for vattr bits set on create
6551                 *      [do_rfs4_set_attrs() only sets resp bits for
6552                 *       non-vattr/vfs bits.]
6553                 * Mask off any bits we set by default so as not to return
6554                 * more attrset bits than were requested in createattrs
6555                 */
6556                if (created) {
6557                        nfs4_vmask_to_nmask(sarg.vap->va_mask, attrset);
6558                        *attrset &= createmask;
6559                } else {
6560                        /*
6561                         * We did not create the vnode (we tried but it
6562                         * already existed).  In this case, the only createattr
6563                         * that the spec allows the server to set is size,
6564                         * and even then, it can only be set if it is 0.
6565                         */
6566                        *attrset = 0;
6567                        if (trunc)
6568                                *attrset = FATTR4_SIZE_MASK;
6569                }
6570        }
6571        if (ntov_table_init)
6572                nfs4_ntov_table_free(&ntov, &sarg);

6574        /*
6575         * Get the initial "after" sequence number, if it fails,
6576         * set to zero, time to before.
6577         */
6578        iva.va_mask = AT_CTIME|AT_SEQ;
6579        if (VOP_GETATTR(dvp, &iva, 0, cs->cr, NULL)) {
6580                iva.va_seq = 0;
6581                iva.va_ctime = bva.va_ctime;
6582        }

6584        /*
6585         * create_vnode attempts to create the file exclusive,
6586         * if it already exists the VOP_CREATE will fail and
6587         * may not increase va_seq. It is atomic if
6588         * we haven't changed the directory, but if it has changed
6589         * we don't know what changed it.
6590         */
6591        if (!created) {
6592                if (bva.va_seq && iva.va_seq &&
6593                    bva.va_seq == iva.va_seq)
6594                        cinfo->atomic = TRUE;
6595                else
6596                        cinfo->atomic = FALSE;
6597                NFS4_SET_FATTR4_CHANGE(cinfo->after, iva.va_ctime);
6598        } else {
6599                /*
6600                 * The entry was created, we need to sync the
6601                 * directory metadata.
6602                 */
6603                (void) VOP_FSYNC(dvp, 0, cs->cr, NULL);
```



```
6605                /*
6606                 * Get "after" change value, if it fails, simply return the
6607                 * before value.
6608                 */
6609                ava.va_mask = AT_CTIME|AT_SEQ;
6610                if (VOP_GETATTR(dvp, &ava, 0, cs->cr, NULL)) {
6611                        ava.va_ctime = bva.va_ctime;
6612                        ava.va_seq = 0;
6613                }

6615                NFS4_SET_FATTR4_CHANGE(cinfo->after, ava.va_ctime);

6617                /*
6618                 * The cinfo->atomic = TRUE only if we have
6619                 * non-zero va_seq's, and it has incremented by exactly one
6620                 * during the create_vnode and it didn't
6621                 * change during the VOP_FSYNC.
6622                 */
6623                if (bva.va_seq && iva.va_seq && ava.va_seq &&
6624                    iva.va_seq == (bva.va_seq + 1) && iva.va_seq == ava.va_seq)
6625                        cinfo->atomic = TRUE;
6626                else
6627                        cinfo->atomic = FALSE;
6628        }

6630        /* Check for mandatory locking and that the size gets set. */
6631        cva.va_mask = AT_MODE;
6632        if (setsize)
6633                cva.va_mask |= AT_SIZE;

6635        /* Assume the worst */
6636        cs->mandlock = TRUE;

6638        if (VOP_GETATTR(vp, &cva, 0, cs->cr, NULL) == 0) {
6639                cs->mandlock = MANDLOCK(cs->vp, cva.va_mode);

6641                /*
6642                 * Truncate the file if necessary; this would be
6643                 * the case for create over an existing file.
6644                 */

6646                if (trunc) {
6647                        int in_crit = 0;
6648                        rfs4_file_t *fp;
6649                        bool_t create = FALSE;

6651                        /*
6652                         * We are writing over an existing file.
6653                         * Check to see if we need to recall a delegation.
6654                         */
6655                        rfs4_hold_deleg_policy();
6656                        if ((fp = rfs4_findfile(vp, NULL, &create)) != NULL) {
6657                                if (rfs4_check_delegated_byfp(FWRITE, fp,
6658                                    (reqsize == 0), FALSE, FALSE, &clientid)) {
6659                                        rfs4_file_rele(fp);
6660                                        rfs4_rele_deleg_policy();
6661                                        VN_RELE(vp);
6662                                        *attrset = 0;
6663                                        return (NFS4ERR_DELAY);
6664                                }
6665                                rfs4_file_rele(fp);
6666                        }
6667                        rfs4_rele_deleg_policy();

6669                        if (nbl_need_check(vp)) {
```

```
6670                                in_crit = 1;

6672                                ASSERT(reqsize == 0);

6674                                nbl_start_crit(vp, RW_READER);
6675                                if (nbl_conflict(vp, NBL_WRITE, 0,
6676                                    cva.va_size, 0, NULL)) {
6677                                        in_crit = 0;
6678                                        nbl_end_crit(vp);
6679                                        VN_RELE(vp);
6680                                        *attrset = 0;
6681                                        return (NFS4ERR_ACCESS);
6682                                }
6683                        }
6684                        ct.cc_sysid = 0;
6685                        ct.cc_pid = 0;
6686                        ct.cc_caller_id = nfs4_srv_caller_id;
6687                        ct.cc_flags = CC_DONTBLOCK;

6689                        cva.va_mask = AT_SIZE;
6690                        cva.va_size = reqsize;
6691                        (void) VOP_SETATTR(vp, &cva, 0, cs->cr, &ct);
6692                        if (in_crit)
6693                                nbl_end_crit(vp);
6694                }
6695        }

6697        error = makefh4(&cs->fh, vp, cs->exi);

6699        /*
6700         * Force modified data and metadata out to stable storage.
6701         */
6702        (void) VOP_FSYNC(vp, FNODSYNC, cs->cr, NULL);

6704        if (error) {
6705                VN_RELE(vp);
6706                *attrset = 0;
6707                return (puterrno4(error));
6708        }

6710        /* if parent dir is attrdir, set namedattr fh flag */
6711        if (dvp->v_flag & V_XATTRDIR)
6712                set_fh4_flag(&cs->fh, FH4_NAMEDATTR);

6714        if (cs->vp)
6715                VN_RELE(cs->vp);

6717        cs->vp = vp;

6719        /*
6720         * if we did not create the file, we will need to check
6721         * the access bits on the file
6722         */

6724        if (!created) {
6725                if (setsize)
6726                        args->share_access |= OPEN4_SHARE_ACCESS_WRITE;
6727                status = check_open_access(args->share_access, cs, req);
6728                if (status != NFS4_OK)
6729                        *attrset = 0;
6730        }
6731        return (status);
6732 }

6734 /*ARGSUSED*/
6735 static void
```



```
6736 rfs4_do_open(struct compound_state *cs, struct svc_req *req,
6737     rfs4_openowner_t *oo, delegreq_t deleg,
6738     uint32_t access, uint32_t deny,
6739     OPEN4res *resp, int deleg_cur)
6740 {
6741         /* XXX Currently not using req  */
6742         rfs4_state_t *sp;
6743         rfs4_file_t *fp;
6744         bool_t screate = TRUE;
6745         bool_t fcreate = TRUE;
6746         uint32_t open_a, share_a;
6747         uint32_t open_d, share_d;
6748         rfs4_deleg_state_t *dsp;
6749         sysid_t sysid;
6750         nfsstat4 status;
6751         caller_context_t ct;
6752         int fflags = 0;
6753         int recall = 0;
6754         int err;
6755         int first_open;

6757         /* get the file struct and hold a lock on it during initial open */
6758         fp = rfs4_findfile_withlock(cs->vp, &cs->fh, &fcreate);
6759         if (fp == NULL) {
6760                 resp->status = NFS4ERR_RESOURCE;
6761                 DTRACE_PROBE1(nfss__e__do__open1, nfsstat4, resp->status);
6762                 return;
6763         }

6765         sp = rfs4_findstate_by_owner_file(oo, fp, &screate);
6766         if (sp == NULL) {
6767                 resp->status = NFS4ERR_RESOURCE;
6768                 DTRACE_PROBE1(nfss__e__do__open2, nfsstat4, resp->status);
6769                 /* No need to keep any reference */
6770                 rw_exit(&fp->rf_file_rwlock);
6771                 rfs4_file_rele(fp);
6772                 return;
6773         }

6775         /* try to get the sysid before continuing */
6776         if ((status = rfs4_client_sysid(oo->ro_client, &sysid)) != NFS4_OK) {
6777                 resp->status = status;
6778                 rfs4_file_rele(fp);
6779                 /* Not a fully formed open; "close" it */
6780                 if (screate == TRUE)
6781                         rfs4_state_close(sp, FALSE, FALSE, cs->cr);
6782                 rfs4_state_rele(sp);
6783                 return;
6784         }

6786         /* Calculate the fflags for this OPEN. */
6787         if (access & OPEN4_SHARE_ACCESS_READ)
6788                 fflags |= FREAD;
6789         if (access & OPEN4_SHARE_ACCESS_WRITE)
6790                 fflags |= FWRITE;

6792         rfs4_dbe_lock(sp->rs_dbe);

6794         /*
6795          * Calculate the new deny and access mode that this open is adding to
6796          * the file for this open owner;
6797          */
6798         open_d = (deny & ~sp->rs_open_deny);
6799         open_a = (access & ~sp->rs_open_access);

6801         /*
```

```
6802          * Calculate the new share access and share deny modes that this open
6803          * is adding to the file for this open owner;
6804          */
6805         share_a = (access & ~sp->rs_share_access);
6806         share_d = (deny & ~sp->rs_share_deny);
6807
6808         first_open = (sp->rs_open_access & OPEN4_SHARE_ACCESS_BOTH) == 0;
6809
6810         /*
6811          * Check to see the client has already sent an open for this
6812          * open owner on this file with the same share/deny modes.
6813          * If so, we don't need to check for a conflict and we don't
6814          * need to add another shrlock.  If not, then we need to
6815          * check for conflicts in deny and access before checking for
6816          * conflicts in delegation.  We don't want to recall a
6817          * delegation based on an open that will eventually fail based
6818          * on shares modes.
6819          */
6820
6821         if (share_a || share_d) {
6822                 if ((err = rfs4_share(sp, access, deny)) != 0) {
6823                         rfs4_dbe_unlock(sp->rs_dbe);
6824                         resp->status = err;
6825
6826                         rfs4_file_rele(fp);
6827                         /* Not a fully formed open; "close" it */
6828                         if (screate == TRUE)
6829                                 rfs4_state_close(sp, FALSE, FALSE, cs->cr);
6830                         rfs4_state_rele(sp);
6831                         return;
6832                 }
6833         }
6834
6835         rfs4_dbe_lock(fp->rf_dbe);
6836
6837         /*
6838          * Check to see if this file is delegated and if so, if a
6839          * recall needs to be done.
6840          */
6841         if (rfs4_check_recall(sp, access)) {
6842                 rfs4_dbe_unlock(fp->rf_dbe);
6843                 rfs4_dbe_unlock(sp->rs_dbe);
6844                 rfs4_recall_deleg(fp, FALSE, sp->rs_owner->ro_client);
6845                 delay(NFS4_DELEGATION_CONFLICT_DELAY);
6846                 rfs4_dbe_lock(sp->rs_dbe);
6847
6848                 /* if state closed while lock was dropped */
6849                 if (sp->rs_closed) {
6850                         if (share_a || share_d)
6851                                 (void) rfs4_unshare(sp);
6852                         rfs4_dbe_unlock(sp->rs_dbe);
6853                         rfs4_file_rele(fp);
6854                         /* Not a fully formed open; "close" it */
6855                         if (screate == TRUE)
6856                                 rfs4_state_close(sp, FALSE, FALSE, cs->cr);
6857                         rfs4_state_rele(sp);
6858                         resp->status = NFS4ERR_OLD_STATEID;
6859                         return;
6860                 }
6861
6862                 rfs4_dbe_lock(fp->rf_dbe);
6863                 /* Let's see if the delegation was returned */
6864                 if (rfs4_check_recall(sp, access)) {
6865                         rfs4_dbe_unlock(fp->rf_dbe);
6866                         if (share_a || share_d)
6867                                 (void) rfs4_unshare(sp);
```



```
6868                         rfs4_dbe_unlock(sp->rs_dbe);
6869                         rfs4_file_rele(fp);
6870                         rfs4_update_lease(sp->rs_owner->ro_client);
6871
6872                         /* Not a fully formed open; "close" it */
6873                         if (screate == TRUE)
6874                                 rfs4_state_close(sp, FALSE, FALSE, cs->cr);
6875                         rfs4_state_rele(sp);
6876                         resp->status = NFS4ERR_DELAY;
6877                         return;
6878                 }
6879         }
6880         /*
6881          * the share check passed and any delegation conflict has been
6882          * taken care of, now call vop_open.
6883          * if this is the first open then call vop_open with fflags.
6884          * if not, call vn_open_upgrade with just the upgrade flags.
6885          *
6886          * if the file has been opened already, it will have the current
6887          * access mode in the state struct.  if it has no share access, then
6888          * this is a new open.
6889          *
6890          * However, if this is open with CLAIM_DLEGATE_CUR, then don't
6891          * call VOP_OPEN(), just do the open upgrade.
6892          */
6893         if (first_open && !deleg_cur) {
6894                 ct.cc_sysid = sysid;
6895                 ct.cc_pid = rfs4_dbe_getid(sp->rs_owner->ro_dbe);
6896                 ct.cc_caller_id = nfs4_srv_caller_id;
6897                 ct.cc_flags = CC_DONTBLOCK;
6898                 err = VOP_OPEN(&cs->vp, fflags, cs->cr, &ct);
6899                 if (err) {
6900                         rfs4_dbe_unlock(fp->rf_dbe);
6901                         if (share_a || share_d)
6902                                 (void) rfs4_unshare(sp);
6903                         rfs4_dbe_unlock(sp->rs_dbe);
6904                         rfs4_file_rele(fp);
6905
6906                         /* Not a fully formed open; "close" it */
6907                         if (screate == TRUE)
6908                                 rfs4_state_close(sp, FALSE, FALSE, cs->cr);
6909                         rfs4_state_rele(sp);
6910                         /* check if a monitor detected a delegation conflict */
6911                         if (err == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK))
6912                                 resp->status = NFS4ERR_DELAY;
6913                         else
6914                                 resp->status = NFS4ERR_SERVERFAULT;
6915                         return;
6916                 }
6917         } else { /* open upgrade */
6918                 /*
6919                  * calculate the fflags for the new mode that is being added
6920                  * by this upgrade.
6921                  */
6922                 fflags = 0;
6923                 if (open_a & OPEN4_SHARE_ACCESS_READ)
6924                         fflags |= FREAD;
6925                 if (open_a & OPEN4_SHARE_ACCESS_WRITE)
6926                         fflags |= FWRITE;
6927                 vn_open_upgrade(cs->vp, fflags);
6928         }
6929         sp->rs_open_access |= access;
6930         sp->rs_open_deny |= deny;
6931
6932         if (open_d & OPEN4_SHARE_DENY_READ)
6933                 fp->rf_deny_read++;
```

```
6934         if (open_d & OPEN4_SHARE_DENY_WRITE)
6935                 fp->rf_deny_write++;
6936         fp->rf_share_deny |= deny;

6938         if (open_a & OPEN4_SHARE_ACCESS_READ)
6939                 fp->rf_access_read++;
6940         if (open_a & OPEN4_SHARE_ACCESS_WRITE)
6941                 fp->rf_access_write++;
6942         fp->rf_share_access |= access;

6944         /*
6945          * Check for delegation here. if the deleg argument is not
6946          * DELEG_ANY, then this is a reclaim from a client and
6947          * we must honor the delegation requested. If necessary we can
6948          * set the recall flag.
6949          */

6951         dsp = rfs4_grant_delegation(deleg, sp, &recall);

6953         cs->deleg = (fp->rf_dinfo.rd_dtype == OPEN_DELEGATE_WRITE);

6955         next_stateid(&sp->rs_stateid);

6957         resp->stateid = sp->rs_stateid.stateid;

6959         rfs4_dbe_unlock(fp->rf_dbe);
6960         rfs4_dbe_unlock(sp->rs_dbe);

6962         if (dsp) {
6963                 rfs4_set_deleg_response(dsp, &resp->delegation, NULL, recall);
6964                 rfs4_deleg_state_rele(dsp);
6965         }

6967         rfs4_file_rele(fp);
6968         rfs4_state_rele(sp);

6970         resp->status = NFS4_OK;
6971 }

6973 /*ARGSUSED*/
6974 static void
6975 rfs4_do_opennull(struct compound_state *cs, struct svc_req *req,
6976     OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
6977 {
6978         change_info4 *cinfo = &resp->cinfo;
6979         bitmap4 *attrset = &resp->attrset;

6981         if (args->opentype == OPEN4_NOCREATE)
6982                 resp->status = rfs4_lookupfile(&args->open_claim4_u.file,
6983                     req, cs, args->share_access, cinfo);
6984         else {
6985                 /* inhibit delegation grants during exclusive create */

6987                 if (args->mode == EXCLUSIVE4)
6988                         rfs4_disable_delegation();

6990                 resp->status = rfs4_createfile(args, req, cs, cinfo, attrset,
6991                     oo->ro_client->rc_clientid);
6992         }

6994         if (resp->status == NFS4_OK) {

6996                 /* cs->vp cs->fh now reference the desired file */

6998                 rfs4_do_open(cs, req, oo,
6999                     oo->ro_need_confirm ? DELEG_NONE : DELEG_ANY,
```



```
7000                     args->share_access, args->share_deny, resp, 0);

7002                 /*
7003                  * If rfs4_createfile set attrset, we must
7004                  * clear this attrset before the response is copied.
7005                  */
7006                 if (resp->status != NFS4_OK && resp->attrset) {
7007                         resp->attrset = 0;
7008                 }
7009         }
7010         else
7011                 *cs->statusp = resp->status;

7013         if (args->mode == EXCLUSIVE4)
7014                 rfs4_enable_delegation();
7015 }

7017 /*ARGSUSED*/
7018 static void
7019 rfs4_do_openprev(struct compound_state *cs, struct svc_req *req,
7020     OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
7021 {
7022         change_info4 *cinfo = &resp->cinfo;
7023         vattr_t va;
7024         vtype_t v_type = cs->vp->v_type;
7025         int error = 0;

7027         /* Verify that we have a regular file */
7028         if (v_type != VREG) {
7029                 if (v_type == VDIR)
7030                         resp->status = NFS4ERR_ISDIR;
7031                 else if (v_type == VLNK)
7032                         resp->status = NFS4ERR_SYMLINK;
7033                 else
7034                         resp->status = NFS4ERR_INVAL;
7035                 return;
7036         }

7038         va.va_mask = AT_MODE|AT_UID;
7039         error = VOP_GETATTR(cs->vp, &va, 0, cs->cr, NULL);
7040         if (error) {
7041                 resp->status = puterrno4(error);
7042                 return;
7043         }

7045         cs->mandlock = MANDLOCK(cs->vp, va.va_mode);

7047         /*
7048          * Check if we have access to the file, Note the the file
7049          * could have originally been open UNCHECKED or GUARDED
7050          * with mode bits that will now fail, but there is nothing
7051          * we can really do about that except in the case that the
7052          * owner of the file is the one requesting the open.
7053          */
7054         if (crgetuid(cs->cr) != va.va_uid) {
7055                 resp->status = check_open_access(args->share_access, cs, req);
7056                 if (resp->status != NFS4_OK) {
7057                         return;
7058                 }
7059         }

7061         /*
7062          * cinfo on a CLAIM_PREVIOUS is undefined, initialize to zero
7063          */
7064         cinfo->before = 0;
7065         cinfo->after = 0;
```

```
7066         cinfo->atomic = FALSE;

7068         rfs4_do_open(cs, req, oo,
7069             NFS4_DELEG4TYPE2REQTYPE(args->open_claim4_u.delegate_type),
7070             args->share_access, args->share_deny, resp, 0);
7071 }

7073 static void
7074 rfs4_do_opendelcur(struct compound_state *cs, struct svc_req *req,
7075     OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
7076 {
7077         int error;
7078         nfsstat4 status;
7079         stateid4 stateid =
7080             args->open_claim4_u.delegate_cur_info.delegate_stateid;
7081         rfs4_deleg_state_t *dsp;

7083         /*
7084          * Find the state info from the stateid and confirm that the
7085          * file is delegated.  If the state openowner is the same as
7086          * the supplied openowner we're done. If not, get the file
7087          * info from the found state info. Use that file info to
7088          * create the state for this lock owner. Note solaris doen't
7089          * really need the pathname to find the file. We may want to
7090          * lookup the pathname and make sure that the vp exist and
7091          * matches the vp in the file structure. However it is
7092          * possible that the pathname nolonger exists (local process
7093          * unlinks the file), so this may not be that useful.
7094          */

7096         status = rfs4_get_deleg_state(&stateid, &dsp);
7097         if (status != NFS4_OK) {
7098                 resp->status = status;
7099                 return;
7100         }

7102         ASSERT(dsp->rds_finfo->rf_dinfo.rd_dtype != OPEN_DELEGATE_NONE);

7104         /*
7105          * New lock owner, create state. Since this was probably called
7106          * in response to a CB_RECALL we set deleg to DELEG_NONE
7107          */

7109         ASSERT(cs->vp != NULL);
7110         VN_RELE(cs->vp);
7111         VN_HOLD(dsp->rds_finfo->rf_vp);
7112         cs->vp = dsp->rds_finfo->rf_vp;

7114         if (error = makefh4(&cs->fh, cs->vp, cs->exi)) {
7115                 rfs4_deleg_state_rele(dsp);
7116                 *cs->statusp = resp->status = puterrno4(error);
7117                 return;
7118         }

7120         /* Mark progress for delegation returns */
7121         dsp->rds_finfo->rf_dinfo.rd_time_lastwrite = gethrestime_sec();
7122         rfs4_deleg_state_rele(dsp);
7123         rfs4_do_open(cs, req, oo, DELEG_NONE,
7124             args->share_access, args->share_deny, resp, 1);
7125 }

7127 /*ARGSUSED*/
7128 static void
7129 rfs4_do_opendelprev(struct compound_state *cs, struct svc_req *req,
7130     OPEN4args *args, rfs4_openowner_t *oo, OPEN4res *resp)
7131 {
```



```
7132         /*
7133          * Lookup the pathname, it must already exist since this file
7134          * was delegated.
7135          *
7136          * Find the file and state info for this vp and open owner pair.
7137          *      check that they are in fact delegated.
7138          *      check that the state access and deny modes are the same.
7139          *
7140          * Return the delgation possibly seting the recall flag.
7141          */
7142         rfs4_file_t *fp;
7143         rfs4_state_t *sp;
7144         bool_t create = FALSE;
7145         bool_t dcreate = FALSE;
7146         rfs4_deleg_state_t *dsp;
7147         nfsace4 *ace;

7149         /* Note we ignore oflags */
7150         resp->status = rfs4_lookupfile(&args->open_claim4_u.file_delegate_prev,
7151             req, cs, args->share_access, &resp->cinfo);

7153         if (resp->status != NFS4_OK) {
7154                 return;
7155         }

7157         /* get the file struct and hold a lock on it during initial open */
7158         fp = rfs4_findfile_withlock(cs->vp, NULL, &create);
7159         if (fp == NULL) {
7160                 resp->status = NFS4ERR_RESOURCE;
7161                 DTRACE_PROBE1(nfss__e__do_opendelprev1, nfsstat4, resp->status);
7162                 return;
7163         }

7165         sp = rfs4_findstate_by_owner_file(oo, fp, &create);
7166         if (sp == NULL) {
7167                 resp->status = NFS4ERR_SERVERFAULT;
7168                 DTRACE_PROBE1(nfss__e__do_opendelprev2, nfsstat4, resp->status);
7169                 rw_exit(&fp->rf_file_rwlock);
7170                 rfs4_file_rele(fp);
7171                 return;
7172         }

7174         rfs4_dbe_lock(sp->rs_dbe);
7175         rfs4_dbe_lock(fp->rf_dbe);
7176         if (args->share_access != sp->rs_share_access ||
7177             args->share_deny != sp->rs_share_deny ||
7178             sp->rs_finfo->rf_dinfo.rd_dtype == OPEN_DELEGATE_NONE) {
7179                 NFS4_DEBUG(rfs4_debug,
7180                     (CE_NOTE, "rfs4_do_opendelprev: state mixup"));
7181                 rfs4_dbe_unlock(fp->rf_dbe);
7182                 rfs4_dbe_unlock(sp->rs_dbe);
7183                 rfs4_file_rele(fp);
7184                 rfs4_state_rele(sp);
7185                 resp->status = NFS4ERR_SERVERFAULT;
7186                 return;
7187         }
7188         rfs4_dbe_unlock(fp->rf_dbe);
7189         rfs4_dbe_unlock(sp->rs_dbe);

7191         dsp = rfs4_finddeleg(sp, &dcreate);
7192         if (dsp == NULL) {
7193                 rfs4_state_rele(sp);
7194                 rfs4_file_rele(fp);
7195                 resp->status = NFS4ERR_SERVERFAULT;
7196                 return;
7197         }
```

```
7199         next_stateid(&sp->rs_stateid);

7201         resp->stateid = sp->rs_stateid.stateid;

7203         resp->delegation.delegation_type = dsp->rds_dtype;

7205         if (dsp->rds_dtype == OPEN_DELEGATE_READ) {
7206                 open_read_delegation4 *rv =
7207                     &resp->delegation.open_delegation4_u.read;

7209                 rv->stateid = dsp->rds_delegid.stateid;
7210                 rv->recall = FALSE; /* no policy in place to set to TRUE */
7211                 ace = &rv->permissions;
7212         } else {
7213                 open_write_delegation4 *rv =
7214                     &resp->delegation.open_delegation4_u.write;

7216                 rv->stateid = dsp->rds_delegid.stateid;
7217                 rv->recall = FALSE;  /* no policy in place to set to TRUE */
7218                 ace = &rv->permissions;
7219                 rv->space_limit.limitby = NFS_LIMIT_SIZE;
7220                 rv->space_limit.nfs_space_limit4_u.filesize = UINT64_MAX;
7221         }

7223         /* XXX For now */
7224         ace->type = ACE4_ACCESS_ALLOWED_ACE_TYPE;
7225         ace->flag = 0;
7226         ace->access_mask = 0;
7227         ace->who.utf8string_len = 0;
7228         ace->who.utf8string_val = 0;

7230         rfs4_deleg_state_rele(dsp);
7231         rfs4_state_rele(sp);
7232         rfs4_file_rele(fp);
7233 }

7235 typedef enum {
7236         NFS4_CHKSEQ_OKAY = 0,
7237         NFS4_CHKSEQ_REPLAY = 1,
7238         NFS4_CHKSEQ_BAD = 2
7239 } rfs4_chkseq_t;

7241 /*
7242  * Generic function for sequence number checks.
7243  */
7244 static rfs4_chkseq_t
7245 rfs4_check_seqid(seqid4 seqid, nfs_resop4 *lastop,
7246     seqid4 rqst_seq, nfs_resop4 *resop, bool_t copyres)
7247 {
7248         /* Same sequence ids and matching operations? */
7249         if (seqid == rqst_seq && resop->resop == lastop->resop) {
7250                 if (copyres == TRUE) {
7251                         rfs4_free_reply(resop);
7252                         rfs4_copy_reply(resop, lastop);
7253                 }
7254                 NFS4_DEBUG(rfs4_debug, (CE_NOTE,
7255                     "Replayed SEQID %d\n", seqid));
7256                 return (NFS4_CHKSEQ_REPLAY);
7257         }

7259         /* If the incoming sequence is not the next expected then it is bad */
7260         if (rqst_seq != seqid + 1) {
7261                 if (rqst_seq == seqid) {
7262                         NFS4_DEBUG(rfs4_debug,
7263                             (CE_NOTE, "BAD SEQID: Replayed sequence id "
```



```
7264                             "but last op was %d current op is %d\n",
7265                             lastop->resop, resop->resop));
7266                         return (NFS4_CHKSEQ_BAD);
7267                 }
7268                 NFS4_DEBUG(rfs4_debug,
7269                     (CE_NOTE, "BAD SEQID: got %u expecting %u\n",
7270                     rqst_seq, seqid));
7271                 return (NFS4_CHKSEQ_BAD);
7272         }

7274         /* Everything okay -- next expected */
7275         return (NFS4_CHKSEQ_OKAY);
7276 }


7279 static rfs4_chkseq_t
7280 rfs4_check_open_seqid(seqid4 seqid, rfs4_openowner_t *op, nfs_resop4 *resop)
7281 {
7282         rfs4_chkseq_t rc;

7284         rfs4_dbe_lock(op->ro_dbe);
7285         rc = rfs4_check_seqid(op->ro_open_seqid, &op->ro_reply, seqid, resop,
7286             TRUE);
7287         rfs4_dbe_unlock(op->ro_dbe);

7289         if (rc == NFS4_CHKSEQ_OKAY)
7290                 rfs4_update_lease(op->ro_client);

7292         return (rc);
7293 }

7295 static rfs4_chkseq_t
7296 rfs4_check_olo_seqid(seqid4 olo_seqid, rfs4_openowner_t *op, nfs_resop4 *resop)
7297 {
7298         rfs4_chkseq_t rc;

7300         rfs4_dbe_lock(op->ro_dbe);
7301         rc = rfs4_check_seqid(op->ro_open_seqid, &op->ro_reply,
7302             olo_seqid, resop, FALSE);
7303         rfs4_dbe_unlock(op->ro_dbe);

7305         return (rc);
7306 }

7308 static rfs4_chkseq_t
7309 rfs4_check_lock_seqid(seqid4 seqid, rfs4_lo_state_t *lsp, nfs_resop4 *resop)
7310 {
7311         rfs4_chkseq_t rc = NFS4_CHKSEQ_OKAY;

7313         rfs4_dbe_lock(lsp->rls_dbe);
7314         if (!lsp->rls_skip_seqid_check)
7315                 rc = rfs4_check_seqid(lsp->rls_seqid, &lsp->rls_reply, seqid,
7316                     resop, TRUE);
7317         rfs4_dbe_unlock(lsp->rls_dbe);

7319         return (rc);
7320 }

7322 static void
7323 rfs4_op_open(nfs_argop4 *argop, nfs_resop4 *resop,
7324     struct svc_req *req, struct compound_state *cs)
7325 {
7326         OPEN4args *args = &argop->nfs_argop4_u.opopen;
7327         OPEN4res *resp = &resop->nfs_resop4_u.opopen;
7328         open_owner4 *owner = &args->owner;
7329         open_claim_type4 claim = args->claim;
```

```
7330         rfs4_client_t *cp;
7331         rfs4_openowner_t *oo;
7332         bool_t create;
7333         bool_t replay = FALSE;
7334         int can_reclaim;

7336         DTRACE_NFSV4_2(op__open__start, struct compound_state *, cs,
7337             OPEN4args *, args);

7339         if (cs->vp == NULL) {
7340                 *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
7341                 goto end;
7342         }

7344         /*
7345          * Need to check clientid and lease expiration first based on
7346          * error ordering and incrementing sequence id.
7347          */
7348         cp = rfs4_findclient_by_id(owner->clientid, FALSE);
7349         if (cp == NULL) {
7350                 *cs->statusp = resp->status =
7351                     rfs4_check_clientid(&owner->clientid, 0);
7352                 goto end;
7353         }

7355         if (rfs4_lease_expired(cp)) {
7356                 rfs4_client_close(cp);
7357                 *cs->statusp = resp->status = NFS4ERR_EXPIRED;
7358                 goto end;
7359         }
7360         can_reclaim = cp->rc_can_reclaim;

7362         /*
7363          * Find the open_owner for use from this point forward.  Take
7364          * care in updating the sequence id based on the type of error
7365          * being returned.
7366          */
7367 retry:
7368         create = TRUE;
7369         oo = rfs4_findopenowner(owner, &create, args->seqid);
7370         if (oo == NULL) {
7371                 *cs->statusp = resp->status = NFS4ERR_RESOURCE;
7372                 rfs4_client_rele(cp);
7373                 goto end;
7374         }

7376         /* Hold off access to the sequence space while the open is done */
7377         rfs4_sw_enter(&oo->ro_sw);

7379         /*
7380          * If the open_owner existed before at the server, then check
7381          * the sequence id.
7382          */
7383         if (!create && !oo->ro_postpone_confirm) {
7384                 switch (rfs4_check_open_seqid(args->seqid, oo, resop)) {
7385                 case NFS4_CHKSEQ_BAD:
7386                         if ((args->seqid > oo->ro_open_seqid) &&
7387                             oo->ro_need_confirm) {
7388                                 rfs4_free_opens(oo, TRUE, FALSE);
7389                                 rfs4_sw_exit(&oo->ro_sw);
7390                                 rfs4_openowner_rele(oo);
7391                                 goto retry;
7392                         }
7393                         resp->status = NFS4ERR_BAD_SEQID;
7394                         goto out;
7395                 case NFS4_CHKSEQ_REPLAY: /* replay of previous request */
```



```
7396                         replay = TRUE;
7397                         goto out;
7398                 default:
7399                         break;
7400                 }

7402                 /*
7403                  * Sequence was ok and open owner exists
7404                  * check to see if we have yet to see an
7405                  * open_confirm.
7406                  */
7407                 if (oo->ro_need_confirm) {
7408                         rfs4_free_opens(oo, TRUE, FALSE);
7409                         rfs4_sw_exit(&oo->ro_sw);
7410                         rfs4_openowner_rele(oo);
7411                         goto retry;
7412                 }
7413         }
7414         /* Grace only applies to regular-type OPENs */
7415         if (rfs4_clnt_in_grace(cp) &&
7416             (claim == CLAIM_NULL || claim == CLAIM_DELEGATE_CUR)) {
7417                 *cs->statusp = resp->status = NFS4ERR_GRACE;
7418                 goto out;
7419         }

7421         /*
7422          * If previous state at the server existed then can_reclaim
7423          * will be set. If not reply NFS4ERR_NO_GRACE to the
7424          * client.
7425          */
7426         if (rfs4_clnt_in_grace(cp) && claim == CLAIM_PREVIOUS && !can_reclaim) {
7427                 *cs->statusp = resp->status = NFS4ERR_NO_GRACE;
7428                 goto out;
7429         }


7432         /*
7433          * Reject the open if the client has missed the grace period
7434          */
7435         if (!rfs4_clnt_in_grace(cp) && claim == CLAIM_PREVIOUS) {
7436                 *cs->statusp = resp->status = NFS4ERR_NO_GRACE;
7437                 goto out;
7438         }

7440         /* Couple of up-front bookkeeping items */
7441         if (oo->ro_need_confirm) {
7442                 /*
7443                  * If this is a reclaim OPEN then we should not ask
7444                  * for a confirmation of the open_owner per the
7445                  * protocol specification.
7446                  */
7447                 if (claim == CLAIM_PREVIOUS)
7448                         oo->ro_need_confirm = FALSE;
7449                 else
7450                         resp->rflags |= OPEN4_RESULT_CONFIRM;
7451         }
7452         resp->rflags |= OPEN4_RESULT_LOCKTYPE_POSIX;

7454         /*
7455          * If there is an unshared filesystem mounted on this vnode,
7456          * do not allow to open/create in this directory.
7457          */
7458         if (vn_ismntpt(cs->vp)) {
7459                 *cs->statusp = resp->status = NFS4ERR_ACCESS;
7460                 goto out;
7461         }
```

```
7463         /*
7464          * access must READ, WRITE, or BOTH.  No access is invalid.
7465          * deny can be READ, WRITE, BOTH, or NONE.
7466          * bits not defined for access/deny are invalid.
7467          */
7468         if (! (args->share_access & OPEN4_SHARE_ACCESS_BOTH) ||
7469             (args->share_access & ~OPEN4_SHARE_ACCESS_BOTH) ||
7470             (args->share_deny & ~OPEN4_SHARE_DENY_BOTH)) {
7471                 *cs->statusp = resp->status = NFS4ERR_INVAL;
7472                 goto out;
7473         }


7476         /*
7477          * make sure attrset is zero before response is built.
7478          */
7479         resp->attrset = 0;

7481         switch (claim) {
7482         case CLAIM_NULL:
7483                 rfs4_do_opennull(cs, req, args, oo, resp);
7484                 break;
7485         case CLAIM_PREVIOUS:
7486                 rfs4_do_openprev(cs, req, args, oo, resp);
7487                 break;
7488         case CLAIM_DELEGATE_CUR:
7489                 rfs4_do_opendelcur(cs, req, args, oo, resp);
7490                 break;
7491         case CLAIM_DELEGATE_PREV:
7492                 rfs4_do_opendelprev(cs, req, args, oo, resp);
7493                 break;
7494         default:
7495                 resp->status = NFS4ERR_INVAL;
7496                 break;
7497         }

7499 out:
7500         rfs4_client_rele(cp);

7502         /* Catch sequence id handling here to make it a little easier */
7503         switch (resp->status) {
7504         case NFS4ERR_BADXDR:
7505         case NFS4ERR_BAD_SEQID:
7506         case NFS4ERR_BAD_STATEID:
7507         case NFS4ERR_NOFILEHANDLE:
7508         case NFS4ERR_RESOURCE:
7509         case NFS4ERR_STALE_CLIENTID:
7510         case NFS4ERR_STALE_STATEID:
7511                 /*
7512                  * The protocol states that if any of these errors are
7513                  * being returned, the sequence id should not be
7514                  * incremented.  Any other return requires an
7515                  * increment.
7516                  */
7517                 break;
7518         default:
7519                 /* Always update the lease in this case */
7520                 rfs4_update_lease(oo->ro_client);

7522                 /* Regular response - copy the result */
7523                 if (!replay)
7524                         rfs4_update_open_resp(oo, resop, &cs->fh);

7526                 /*
7527                  * REPLAY case: Only if the previous response was OK
```



```
7528                  * do we copy the filehandle.  If not OK, no
7529                  * filehandle to copy.
7530                  */
7531                 if (replay == TRUE &&
7532                     resp->status == NFS4_OK &&
7533                     oo->ro_reply_fh.nfs_fh4_val) {
7534                         /*
7535                          * If this is a replay, we must restore the
7536                          * current filehandle/vp to that of what was
7537                          * returned originally.  Try our best to do
7538                          * it.
7539                          */
7540                         nfs_fh4_fmt_t *fh_fmtp =
7541                             (nfs_fh4_fmt_t *)oo->ro_reply_fh.nfs_fh4_val;

7543                         if (cs->exi)
7544                                 exi_rele(cs->exi);
7545                         cs->exi = checkexport(&fh_fmtp->fh4_fsid,
2826                         cs->exi = checkexport4(&fh_fmtp->fh4_fsid,
7546                             (fid_t *)&fh_fmtp->fh4_xlen, NULL);

7548                         if (cs->exi == NULL) {
7549                                 resp->status = NFS4ERR_STALE;
7550                                 goto finish;
7551                         }

7553                         VN_RELE(cs->vp);

7555                         cs->vp = nfs4_fhtovp(&oo->ro_reply_fh, cs->exi,
7556                             &resp->status);

7558                         if (cs->vp == NULL)
7559                                 goto finish;

7561                         nfs_fh4_copy(&oo->ro_reply_fh, &cs->fh);
7562                 }

7564                 /*
7565                  * If this was a replay, no need to update the
7566                  * sequence id. If the open_owner was not created on
7567                  * this pass, then update.  The first use of an
7568                  * open_owner will not bump the sequence id.
7569                  */
7570                 if (replay == FALSE && !create)
7571                         rfs4_update_open_sequence(oo);
7572                 /*
7573                  * If the client is receiving an error and the
7574                  * open_owner needs to be confirmed, there is no way
7575                  * to notify the client of this fact ignoring the fact
7576                  * that the server has no method of returning a
7577                  * stateid to confirm.  Therefore, the server needs to
7578                  * mark this open_owner in a way as to avoid the
7579                  * sequence id checking the next time the client uses
7580                  * this open_owner.
7581                  */
7582                 if (resp->status != NFS4_OK && oo->ro_need_confirm)
7583                         oo->ro_postpone_confirm = TRUE;
7584                 /*
7585                  * If OK response then clear the postpone flag and
7586                  * reset the sequence id to keep in sync with the
7587                  * client.
7588                  */
7589                 if (resp->status == NFS4_OK && oo->ro_postpone_confirm) {
7590                         oo->ro_postpone_confirm = FALSE;
7591                         oo->ro_open_seqid = args->seqid;
7592                 }
```

```
7593                    break;
7594            }

7596 finish:
7597            *cs->statusp = resp->status;

7599            rfs4_sw_exit(&oo->ro_sw);
7600            rfs4_openowner_rele(oo);

7602 end:
7603            DTRACE_NFSV4_2(op__open__done, struct compound_state *, cs,
7604                OPEN4res *, resp);
7605 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   31301 Wed Sep 14 16:21:03 2016
new/usr/src/uts/common/fs/nfs/nfs4_srv_ns.c
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_

 556 /*
 557  * This function checks the path to a new export to
 558  * check whether all the pathname components are
 559  * exported. It works by climbing the file tree one
 560  * component at a time via "..", crossing mountpoints
 561  * if necessary until an export entry is found, or the
 562  * system root is reached.
 563  *
 564  * If an unexported mountpoint is found, then
 565  * a new pseudo export is added and the pathname from
 566  * the mountpoint down to the export is added to the
 567  * visible list for the new pseudo export.  If an existing
 568  * pseudo export is found, then the pathname is added
 569  * to its visible list.
 570  *
 571  * Note that there's some tests for exportdir.
 572  * The exportinfo entry that's passed as a parameter
 573  * is that of the real export and exportdir is set
 574  * for this case.
 575  *
 576  * Here is an example of a possible setup:
 577  *
 578  * () - a new fs; fs mount point
 579  * EXPORT - a real exported node
 580  * PSEUDO - a pseudo node
 581  * vis - visible list
 582  * f# - security flavor#
 583  * (f#) - security flavor# propagated from its descendents
 584  * "" - covered vnode
 585  *
 586  *
 587  *                 /
 588  *                 |
 589  *                 (a) PSEUDO (f1,f2)
 590  *                 |   vis: b,b,"c","n"
 591  *                 |
 592  *                 b
 593  *        ---------|------------------
 594  *        |                          |
 595  *        (c) EXPORT,f1(f2)          (n) PSEUDO (f1,f2)
 596  *        |   vis: "e","d"           |   vis: m,m,,p,q,"o"
 597  *        |                          |
 598  *  ------------------          -------------------
 599  *  |       |        |         |                   |
 600  *  (d)     (e)      f         m EXPORT,f1(f2)     p
 601  *  EXPORT  EXPORT             |                   |
 602  *  f1      f2                 |                   |
 603  *          |                  |                   |
 604  *          j                 (o) EXPORT,f2        q EXPORT f2
 605  *
 606  */
 607 int
 608 treeclimb_export(struct exportinfo *exip)
 609 {
 610         vnode_t *dvp, *vp;
 611         fid_t fid;
 612         int error;
 613         int exportdir;
 614         struct exportinfo *exi = NULL;
```



```
 615         struct exportinfo *new_exi = exip;
 616         struct exp_visible *visp;
 617         struct exp_visible *vis_head = NULL;
 618         struct vattr va;
 619         treenode_t *tree_head = NULL;

 621         ASSERT(RW_WRITE_HELD(&exported_lock));

 623         vp = exip->exi_vp;
 624         VN_HOLD(vp);
 625         exportdir = 1;

 627         for (;;) {

 629                 bzero(&fid, sizeof (fid));
 630                 fid.fid_len = MAXFIDSZ;
 631                 error = vop_fid_pseudo(vp, &fid);
 632                 if (error)
 633                         break;

 635                 if (! exportdir) {
 636                         /*
 637                          * Check if this exportroot is a VROOT dir.  If so,
 638                          * then attach the pseudonodes.  If not, then
 639                          * continue .. traversal until we hit a VROOT
 640                          * export (pseudo or real).
 641                          */
 642                         exi = checkexport_nohold(&vp->v_vfsp->vfs_fsid, &fid,
 643                             vp);
 642                         exi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);
 644                         if (exi != NULL && vp->v_flag & VROOT) {
 645                                 /*
 646                                  * Found an export info
 647                                  *
 648                                  * Extend the list of visible
 649                                  * directories whether it's a pseudo
 650                                  * or a real export.
 651                                  */
 652                                 more_visible(exi, tree_head);
 653                                 break;  /* and climb no further */
 654                         }
 655                 }

 657                 /*
 658                  * If at the root of the filesystem, need
 659                  * to traverse across the mountpoint
 660                  * and continue the climb on the mounted-on
 661                  * filesystem.
 662                  */
 663                 if (vp->v_flag & VROOT) {

 665                         if (! exportdir) {
 666                                 /*
 667                                  * Found the root directory of a filesystem
 668                                  * that isn't exported.  Need to export
 669                                  * this as a pseudo export so that an NFS v4
 670                                  * client can do lookups in it.
 671                                  */
 672                                 new_exi = pseudo_exportfs(vp, &fid, vis_head,
 673                                     NULL);
 674                                 vis_head = NULL;
 675                         }

 677                         if (VN_CMP(vp, rootdir)) {
 678                                 /* at system root */
 679                                 /*
```

```
680                                  * If sharing "/", new_exi is shared exportinfo
681                                  * (exip). Otherwise, new_exi is exportinfo
682                                  * created in pseudo_exportfs() above.
683                                  */
684                                 ns_root = tree_prepend_node(tree_head, 0,
685                                     new_exi);
686                                 break;
687                         }

689                         vp = untraverse(vp);
690                         exportdir = 0;
691                         continue;
692                 }

694                 /*
695                  * Do a getattr to obtain the nodeid (inode num)
696                  * for this vnode.
697                  */
698                 va.va_mask = AT_NODEID;
699                 error = VOP_GETATTR(vp, &va, 0, CRED(), NULL);
700                 if (error)
701                         break;

703                 /*
704                  *  Add this directory fid to visible list
705                  */
706                 visp = kmem_alloc(sizeof (*visp), KM_SLEEP);
707                 VN_HOLD(vp);
708                 visp->vis_vp = vp;
709                 visp->vis_fid = fid;            /* structure copy */
710                 visp->vis_ino = va.va_nodeid;
711                 visp->vis_count = 1;
712                 visp->vis_exported = exportdir;
713                 visp->vis_secinfo = NULL;
714                 visp->vis_seccnt = 0;
715                 visp->vis_next = vis_head;
716                 vis_head = visp;


719                 /*
720                  * Will set treenode's pointer to exportinfo to
721                  * 1. shared exportinfo (exip) - if first visit here
722                  * 2. freshly allocated pseudo export (if any)
723                  * 3. null otherwise
724                  */
725                 tree_head = tree_prepend_node(tree_head, visp, new_exi);
726                 new_exi = NULL;

728                 /*
729                  * Now, do a ".." to find parent dir of vp.
730                  */
731                 error = VOP_LOOKUP(vp, "..", &dvp, NULL, 0, NULL, CRED(),
732                     NULL, NULL, NULL);

734                 if (error == ENOTDIR && exportdir) {
735                         dvp = exip->exi_dvp;
736                         ASSERT(dvp != NULL);
737                         VN_HOLD(dvp);
738                         error = 0;
739                 }

741                 if (error)
742                         break;

744                 exportdir = 0;
745                 VN_RELE(vp);
```



```
746                         vp = dvp;
747                 }

749                 VN_RELE(vp);

751                 /*
752                  * We can have set error due to error in:
753                  * 1. vop_fid_pseudo()
754                  * 2. VOP_GETATTR()
755                  * 3. VOP_LOOKUP()
756                  * We must free pseudo exportinfos, visibles and treenodes.
757                  * Visibles are referenced from treenode_t::tree_vis and
758                  * exportinfo_t::exi_visible. To avoid double freeing, only
759                  * exi_visible pointer is used, via exi_rele(), for the clean-up.
760                  */
761                 if (error) {
762                         /* Free unconnected visibles, if there are any. */
763                         if (vis_head)
764                                 free_visible(vis_head);

766                         /* Connect unconnected exportinfo, if there is any. */
767                         if (new_exi && new_exi != exip)
768                                 tree_head = tree_prepend_node(tree_head, 0, new_exi);

770                         while (tree_head) {
771                                 treenode_t *t2 = tree_head;
772                                 exportinfo_t *e  = tree_head->tree_exi;
773                                 /* exip will be freed in exportfs() */
774                                 if (e && e != exip) {
775                                         export_unlink(e);
776                                         exi_rele(e);
777                                 }
778                                 tree_head = tree_head->tree_child_first;
779                                 kmem_free(t2, sizeof (*t2));
780                         }
781                 }

783                 return (error);
784 }
```

_____***unchanged_portion_omitted_***

****
40103 Wed Sep 14 16:21:03 2016
new/usr/src/uts/common/fs/nfs/nfs4_srv_readdir.c
7378 exported_lock held during nfs4 compound processing
****
_____unchanged_portion_omitted_

```
105 int
106 nfs4_readdir_getvp(vnode_t *dvp, char *d_name, vnode_t **vpp,
107                 struct exportinfo **exi, struct svc_req *req,
108                 struct compound_state *cs, int expseudo)
109 {
110         int error;
111         int ismntpt;
112         fid_t fid;
113         vnode_t *vp, *pre_tvp;
114         nfsstat4 status;
115         struct exportinfo *newexi, *saveexi;
116         cred_t *scr;

118         *vpp = vp = NULL;

120         if (error = VOP_LOOKUP(dvp, d_name, &vp, NULL, 0, NULL, cs->cr,
121             NULL, NULL, NULL))
122                 return (error);

124         /*
125          * If the directory is a referral point, don't return the
126          * attrs, instead set rdattr_error to MOVED.
127          */
128         if (vn_is_nfs_reparse(vp, cs->cr) && !client_is_downrev(req)) {
129                 VN_RELE(vp);
130                 DTRACE_PROBE2(nfs4serv__func__referral__moved,
131                     vnode_t *, vp, char *, "nfs4_readdir_getvp");
132                 return (NFS4ERR_MOVED);
133         }

135         /* Is this object mounted upon? */
136         ismntpt = vn_ismntpt(vp);

138         /*
139          * Nothing more to do if object is not a mount point or
140          * a possible LOFS shadow of an LOFS mount (which won't
141          * have v_vfsmountedhere set)
142          */
143         if (ismntpt == 0 && dvp->v_vfsp == vp->v_vfsp && expseudo == 0) {
144                 *vpp = vp;
145                 return (0);
146         }

148         if (ismntpt) {
149                 /*
150                  * Something is mounted here. Traverse and manage the
151                  * namespace
152                  */
153                 pre_tvp = vp;
154                 VN_HOLD(pre_tvp);

156                 if ((error = traverse(&vp)) != 0) {
157                         VN_RELE(vp);
158                         VN_RELE(pre_tvp);
159                         return (error);
160                 }
161                 if (vn_is_nfs_reparse(vp, cs->cr)) {
162                         VN_RELE(vp);
163                         VN_RELE(pre_tvp);
```



```
164                         DTRACE_PROBE2(nfs4serv__func__referral__moved,
165                             vnode_t *, vp, char *, "nfs4_readdir_getvp");
166                         return (NFS4ERR_MOVED);
167                 }
168         }

170         bzero(&fid, sizeof (fid));
171         fid.fid_len = MAXFIDSZ;

173         /*
174          * If VOP_FID not supported by underlying fs (mntfs, procfs,
175          * etc.), then return attrs for stub instead of VROOT object.
176          * If it fails for any other reason, then return the error.
177          */
178         if (error = VOP_FID(vp, &fid, NULL)) {
179                 if (ismntpt == 0) {
180                         VN_RELE(vp);
181                         return (error);
182                 }

184                 if (error != ENOSYS && error != ENOTSUP) {
185                         VN_RELE(vp);
186                         VN_RELE(pre_tvp);
187                         return (error);
188                 }
189                 /* go back to vnode that is "under" mount */
190                 VN_RELE(vp);
191                 *vpp = pre_tvp;
192                 return (0);
193         }

195         newexi = checkexport(&vp->v_vfsp->vfs_fsid, &fid, vp);
195         newexi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);
196         if (newexi == NULL) {
197                 if (ismntpt == 0) {
198                         *vpp = vp;
199                 } else {
200                         VN_RELE(vp);
201                         *vpp = pre_tvp;
202                 }
203                 return (0);
204         }

206         if (ismntpt)
207                 VN_RELE(pre_tvp);

209         /* Save the exi and present the new one to checkauth4() */
210         saveexi = cs->exi;
211         cs->exi = newexi;

213         /* Get the right cred like lookup does */
214         scr = cs->cr;
215         cs->cr = crdup(cs->basecr);

217         status = call_checkauth4(cs, req);

219         crfree(cs->cr);
220         cs->cr = scr;
221         cs->exi = saveexi;

223         /* Reset what call_checkauth4() may have set */
224         *cs->statusp = NFS4_OK;

226         if (status != NFS4_OK) {
227                 VN_RELE(vp);
228                 exi_rele(newexi);
```

```
#endif /* ! codereview */
                if (status == NFS4ERR_DELAY)
                        status = NFS4ERR_ACCESS;
                return (status);
        }
        *vpp = vp;
        *exi = newexi;

        return (0);
}

/* This is the set of pathconf data for vfs */
typedef struct {
        uint64_t maxfilesize;
        uint32_t maxlink;
        uint32_t maxname;
} rfs4_pc_encode_t;


static int
rfs4_get_pc_encode(vnode_t *vp, rfs4_pc_encode_t *pce, bitmap4 ar, cred_t *cr)
{
        int error;
        ulong_t pc_val;

        pce->maxfilesize = 0;
        pce->maxlink = 0;
        pce->maxname = 0;

        if (ar & FATTR4_MAXFILESIZE_MASK) {
                /* Maximum File Size */
                error = VOP_PATHCONF(vp, _PC_FILESIZEBITS, &pc_val, cr, NULL);
                if (error)
                        return (error);

                /*
                 * If the underlying file system does not support
                 * _PC_FILESIZEBITS, return a reasonable default. Note that
                 * error code on VOP_PATHCONF will be 0, even if the underlying
                 * file system does not support _PC_FILESIZEBITS.
                 */
                if (pc_val == (ulong_t)-1) {
                        pce->maxfilesize = MAXOFF32_T;
                } else {
                        if (pc_val >= (sizeof (uint64_t) * 8))
                                pce->maxfilesize = INT64_MAX;
                        else
                                pce->maxfilesize = ((1LL << (pc_val - 1)) - 1);
                }
        }

        if (ar & FATTR4_MAXLINK_MASK) {
                /* Maximum Link Count */
                error = VOP_PATHCONF(vp, _PC_LINK_MAX, &pc_val, cr, NULL);
                if (error)
                        return (error);

                pce->maxlink = pc_val;
        }

        if (ar & FATTR4_MAXNAME_MASK) {
                /* Maximum Name Length */
                error = VOP_PATHCONF(vp, _PC_NAME_MAX, &pc_val, cr, NULL);
                if (error)
                        return (error);
```



```
                pce->maxname = pc_val;
        }

        return (0);
}

/* This is the set of statvfs data that is ready for encoding */
typedef struct {
        uint64_t space_avail;
        uint64_t space_free;
        uint64_t space_total;
        u_longlong_t fa;
        u_longlong_t ff;
        u_longlong_t ft;
} rfs4_sb_encode_t;

static int
rfs4_get_sb_encode(vfs_t *vfsp, rfs4_sb_encode_t *psbe)
{
        int error;
        struct statvfs64 sb;

        /* Grab the per filesystem info */
        if (error = VFS_STATVFS(vfsp, &sb)) {
                return (error);
        }

        /* Calculate space available */
        if (sb.f_bavail != (fsblkcnt64_t)-1) {
                psbe->space_avail =
                    (fattr4_space_avail) sb.f_frsize *
                    (fattr4_space_avail) sb.f_bavail;
        } else {
                psbe->space_avail =
                    (fattr4_space_avail) sb.f_bavail;
        }

        /* Calculate space free */
        if (sb.f_bfree != (fsblkcnt64_t)-1) {
                psbe->space_free =
                    (fattr4_space_free) sb.f_frsize *
                    (fattr4_space_free) sb.f_bfree;
        } else {
                psbe->space_free =
                    (fattr4_space_free) sb.f_bfree;
        }

        /* Calculate space total */
        if (sb.f_blocks != (fsblkcnt64_t)-1) {
                psbe->space_total =
                    (fattr4_space_total) sb.f_frsize *
                    (fattr4_space_total) sb.f_blocks;
        } else {
                psbe->space_total =
                    (fattr4_space_total) sb.f_blocks;
        }

        /* For use later on attr encode */
        psbe->fa = sb.f_favail;
        psbe->ff = sb.f_ffree;
        psbe->ft = sb.f_files;

        return (0);
}

/*
```

```
 * Macros to handle if we have don't have enough space for the requested
 * attributes and this is the first entry and the
 * requested attributes are more than the minimal useful
 * set, reset the attributes to the minimal set and
 * retry the encoding. If the client has asked for both
 * mounted_on_fileid and fileid, prefer mounted_on_fileid.
 */
#define MINIMAL_RD_ATTRS                                        \
        (FATTR4_MOUNTED_ON_FILEID_MASK|                         \
        FATTR4_FILEID_MASK|                                     \
        FATTR4_RDATTR_ERROR_MASK)

#define MINIMIZE_ATTR_MASK(m) {                                 \
        if ((m) & FATTR4_MOUNTED_ON_FILEID_MASK)                \
            (m) &= FATTR4_RDATTR_ERROR_MASK|FATTR4_MOUNTED_ON_FILEID_MASK;\
        else                                                    \
            (m) &= FATTR4_RDATTR_ERROR_MASK|FATTR4_FILEID_MASK;         \
}

#define IS_MIN_ATTR_MASK(m)     (((m) & ~MINIMAL_RD_ATTRS) == 0)
/*
 * If readdir only needs to return FILEID, we can take it from the
 * dirent struct and save doing the lookup.
 */
/* ARGSUSED */
void
rfs4_op_readdir(nfs_argop4 *argop, nfs_resop4 *resop,
        struct svc_req *req, struct compound_state *cs)
{
        READDIR4args *args = &argop->nfs_argop4_u.opreaddir;
        READDIR4res *resp = &resop->nfs_resop4_u.opreaddir;
        struct exportinfo *newexi = NULL;
        int error;
        mblk_t *mp;
        uint_t mpcount;
        int alloc_err = 0;
        vnode_t *dvp = cs->vp;
        vnode_t *vp;
        vattr_t va;
        struct dirent64 *dp;
        rfs4_sb_encode_t dsbe, sbe;
        int vfs_different;
        int rddir_data_len, rddir_result_size;
        caddr_t rddir_data;
        offset_t rddir_next_offset;
        int dircount;
        int no_space;
        int iseofdir;
        uint_t eof;
        struct iovec iov;
        struct uio uio;
        int tsize;
        int check_visible;
        int expseudo = 0;

        uint32_t *ptr, *ptr_redzone;
        uint32_t *beginning_ptr;
        uint32_t *lastentry_ptr;
        uint32_t *attrmask_ptr;
        uint32_t *attr_offset_ptr;
        uint32_t attr_length;
        uint32_t rndup;
        uint32_t namelen;
        uint32_t rddirattr_error = 0;
        int nents;
        bitmap4 ar = args->attr_request & NFS4_SRV_RDDIR_SUPPORTED_ATTRS;
```



```
        bitmap4 ae;
        rfs4_pc_encode_t dpce, pce;
        ulong_t pc_val;
        uint64_t maxread;
        uint64_t maxwrite;
        uint_t true = TRUE;
        uint_t false = FALSE;
        uid_t lastuid;
        gid_t lastgid;
        int lu_set, lg_set;
        utf8string owner, group;
        int owner_error, group_error;
        struct sockaddr *ca;
        char *name = NULL;

        DTRACE_NFSV4_2(op__readdir__start, struct compound_state *, cs,
            READDIR4args *, args);

        lu_set = lg_set = 0;
        owner.utf8string_len = group.utf8string_len = 0;
        owner.utf8string_val = group.utf8string_val = NULL;

        resp->mblk = NULL;

        /* Maximum read and write size */
        maxread = maxwrite = rfs4_tsize(req);

        if (dvp == NULL) {
                *cs->statusp = resp->status = NFS4ERR_NOFILEHANDLE;
                goto out;
        }

        /*
         * If there is an unshared filesystem mounted on this vnode,
         * do not allow readdir in this directory.
         */
        if (vn_ismntpt(dvp)) {
                *cs->statusp = resp->status = NFS4ERR_ACCESS;
                goto out;
        }

        if (dvp->v_type != VDIR) {
                *cs->statusp = resp->status = NFS4ERR_NOTDIR;
                goto out;
        }

        if (args->maxcount <= RFS4_MINLEN_RDDIR4) {
                *cs->statusp = resp->status = NFS4ERR_TOOSMALL;
                goto out;
        }

        /*
         * If write-only attrs are requested, then fail the readdir op
         */
        if (args->attr_request &
            (FATTR4_TIME_MODIFY_SET_MASK | FATTR4_TIME_ACCESS_SET_MASK)) {
                *cs->statusp = resp->status = NFS4ERR_INVAL;
                goto out;
        }

        error = VOP_ACCESS(dvp, VREAD, 0, cs->cr, NULL);
        if (error) {
                *cs->statusp = resp->status = puterrno4(error);
                goto out;
        }
```

```
493         if (args->cookieverf != Readdir4verf) {
494                 *cs->statusp = resp->status = NFS4ERR_NOT_SAME;
495                 goto out;
496         }
497
498         /* Is there pseudo-fs work that is needed for this readdir? */
499         check_visible = PSEUDO(cs->exi) ||
500             ! is_exported_sec(cs->nfsflavor, cs->exi) ||
501             cs->access & CS_ACCESS_LIMITED;
502
503         /* Check the requested attributes and only do the work if needed */
504
505         if (ar & (FATTR4_MAXFILESIZE_MASK |
506             FATTR4_MAXLINK_MASK |
507             FATTR4_MAXNAME_MASK)) {
508                 if (error = rfs4_get_pc_encode(cs->vp, &dpce, ar, cs->cr)) {
509                         *cs->statusp = resp->status = puterrno4(error);
510                         goto out;
511                 }
512                 pce = dpce;
513         }
514
515         /* If there is statvfs data requested, pick it up once */
516         if (ar &
517             (FATTR4_FILES_AVAIL_MASK |
518             FATTR4_FILES_FREE_MASK |
519             FATTR4_FILES_TOTAL_MASK |
520             FATTR4_FILES_AVAIL_MASK |
521             FATTR4_FILES_FREE_MASK |
522             FATTR4_FILES_TOTAL_MASK)) {
523                 if (error = rfs4_get_sb_encode(dvp->v_vfsp, &dsbe)) {
524                         *cs->statusp = resp->status = puterrno4(error);
525                         goto out;
526                 }
527                 sbe = dsbe;
528         }
529
530         /*
531          * Max transfer size of the server is the absolute limite.
532          * If the client has decided to max out with something really
533          * tiny, then return toosmall.  Otherwise, move forward and
534          * see if a single entry can be encoded.
535          */
536         tsize = rfs4_tsize(req);
537         if (args->maxcount > tsize)
538                 args->maxcount = tsize;
539         else if (args->maxcount < RFS4_MINLEN_RDDIR_BUF) {
540                 if (args->maxcount < RFS4_MINLEN_ENTRY4) {
541                         *cs->statusp = resp->status = NFS4ERR_TOOSMALL;
542                         goto out;
543                 }
544         }
545
546         /*
547          * How large should the mblk be for outgoing encoding.
548          */
549         if (args->maxcount < MAXBSIZE)
550                 mpcount = MAXBSIZE;
551         else
552                 mpcount = args->maxcount;
553
554         /*
555          * mp will contain the data to be sent out in the readdir reply.
556          * It will be freed after the reply has been sent.
557          * Let's roundup the data to a BYTES_PER_XDR_UNIX multiple,
558          * so that the call to xdrmblk_putmblk() never fails.
```



```
559          */
560         mp = allocb(RNDUP(mpcount), BPRI_MED);
561
562         if (mp == NULL) {
563                 /*
564                  * The allocation of the client's requested size has
565                  * failed.  It may be that the size is too large for
566                  * current system utilization; step down to a "common"
567                  * size and wait for the allocation to occur.
568                  */
569                 if (mpcount > MAXBSIZE)
570                         args->maxcount = mpcount = MAXBSIZE;
571                 mp = allocb_wait(RNDUP(mpcount), BPRI_MED,
572                     STR_NOSIG, &alloc_err);
573         }
574
575         ASSERT(mp != NULL);
576         ASSERT(alloc_err == 0);
577
578         resp->mblk = mp;
579
580         ptr = beginning_ptr = (uint32_t *)mp->b_datap->db_base;
581
582         /*
583          * The "redzone" at the end of the encoding buffer is used
584          * to deal with xdr encoding length.  Instead of checking
585          * each encoding of an attribute value before it is done,
586          * make the assumption that it will fit into the buffer and
587          * check occasionally.
588          *
589          * The largest block of attributes that are encoded without
590          * checking the redzone is 18 * BYTES_PER_XDR_UNIT (72 bytes)
591          * "round" to 128 as the redzone size.
592          */
593         if (args->maxcount < (mpcount - 128))
594                 ptr_redzone =
595                     (uint32_t *)(((char *)ptr) + RNDUP(args->maxcount));
596         else
597                 ptr_redzone =
598                     (uint32_t *)((((char *)ptr) + RNDUP(mpcount)) - 128);
599
600         /*
601          * Set the dircount; this will be used as the size for the
602          * readdir of the underlying filesystem.  First make sure
603          * that it is large enough to do a reasonable readdir (client
604          * may have short changed us - it is an advisory number);
605          * then make sure that it isn't too large.
606          * After all of that, if maxcount is "small" then just use
607          * that for the dircount number.
608          */
609         dircount = (args->dircount < MAXBSIZE) ? MAXBSIZE : args->dircount;
610         dircount = (dircount > tsize) ? tsize : dircount;
611         if (dircount > args->maxcount)
612                 dircount = args->maxcount;
613         if (args->maxcount <= MAXBSIZE) {
614                 if (args->maxcount < RFS4_MINLEN_RDDIR_BUF)
615                         dircount = RFS4_MINLEN_RDDIR_BUF;
616                 else
617                         dircount = args->maxcount;
618         }
619
620         /* number of entries fully encoded in outgoing buffer */
621         nents = 0;
622
623         /* ENCODE READDIR4res.cookieverf */
624         IXDR_PUT_HYPER(ptr, Readdir4verf);
```

```
626         rddir_data_len = dircount;
627         rddir_data = kmem_alloc(rddir_data_len, KM_NOSLEEP);
628         if (rddir_data == NULL) {
629                 /* The allocation failed; downsize and wait for it this time */
630                 if (rddir_data_len > MAXBSIZE)
631                         rddir_data_len = dircount = MAXBSIZE;
632                 rddir_data = kmem_alloc(rddir_data_len, KM_SLEEP);
633         }

635         rddir_next_offset = (offset_t)args->cookie;

637         ca = (struct sockaddr *)svc_getrpccaller(req->rq_xprt)->buf;

639 readagain:

641         no_space = FALSE;
642         iseofdir = FALSE;

644         vp = NULL;

646         /* Move on to reading the directory contents */
647         iov.iov_base = rddir_data;
648         iov.iov_len = rddir_data_len;
649         uio.uio_iov = &iov;
650         uio.uio_iovcnt = 1;
651         uio.uio_segflg = UIO_SYSSPACE;
652         uio.uio_extflg = UIO_COPY_CACHED;
653         uio.uio_loffset = rddir_next_offset;
654         uio.uio_resid = rddir_data_len;

656         (void) VOP_RWLOCK(dvp, V_WRITELOCK_FALSE, NULL);

658         error = VOP_READDIR(dvp, &uio, cs->cr, &iseofdir, NULL, 0);

660         VOP_RWUNLOCK(dvp, V_WRITELOCK_FALSE, NULL);

662         if (error) {
663                 kmem_free((caddr_t)rddir_data, rddir_data_len);
664                 freeb(resp->mblk);
665                 resp->mblk = NULL;
666                 resp->data_len = 0;
667                 *cs->statusp = resp->status = puterrno4(error);
668                 goto out;
669         }


672         rddir_result_size = rddir_data_len - uio.uio_resid;

674         /* No data were read. Check if we reached the end of the directory. */
675         if (rddir_result_size == 0) {
676                 /* encode the BOOLEAN marking no further entries */
677                 IXDR_PUT_U_INT32(ptr, false);
678                 /* encode the BOOLEAN signifying end of directory */
679                 IXDR_PUT_U_INT32(ptr, iseofdir ? true : false);
680                 resp->data_len = (char *)ptr - (char *)beginning_ptr;
681                 resp->mblk->b_wptr += resp->data_len;
682                 kmem_free((caddr_t)rddir_data, rddir_data_len);
683                 *cs->statusp = resp->status = NFS4_OK;
684                 goto out;
685         }

687         lastentry_ptr = ptr;
688         no_space = 0;
689         for (dp = (struct dirent64 *)rddir_data;
690             !no_space && rddir_result_size > 0; dp = nextdp(dp)) {
```



```
692                 /* reset expseudo */
693                 expseudo = 0;

695                 if (vp) {
696                         VN_RELE(vp);
697                         vp = NULL;
698                 }

700                 if (newexi) {
701                         exi_rele(newexi);
228                 if (newexi)
702                         newexi = NULL;
703                 }
704 #endif /* ! codereview */

706                 rddir_result_size -= dp->d_reclen;

708                 /* skip "." and ".." entries */
709                 if (dp->d_ino == 0 || NFS_IS_DOTNAME(dp->d_name)) {
710                         rddir_next_offset = dp->d_off;
711                         continue;
712                 }

714                 if (check_visible &&
715                     !nfs_visible_inode(cs->exi, dp->d_ino, &expseudo)) {
716                         rddir_next_offset = dp->d_off;
717                         continue;
718                 }

720                 /*
721                  * Only if the client requested attributes...
722                  * If the VOP_LOOKUP fails ENOENT, then skip this entry
723                  * for the readdir response.  If there was another error,
724                  * then set the rddirattr_error and the error will be
725                  * encoded later in the "attributes" section.
726                  */
727                 ae = ar;
728                 if (ar == 0)
729                         goto reencode_attrs;

731                 error = nfs4_readdir_getvp(dvp, dp->d_name,
732                     &vp, &newexi, req, cs, expseudo);
733                 if (error == ENOENT) {
734                         rddir_next_offset = dp->d_off;
735                         continue;
736                 }

738                 rddirattr_error = error;

740                 /*
741                  * The vp obtained from above may be from a
742                  * different filesystem mount and the vfs-like
743                  * attributes should be obtained from that
744                  * different vfs; only do this if appropriate.
745                  */
746                 if (vp &&
747                     (vfs_different = (dvp->v_vfsp != vp->v_vfsp))) {
748                         if (ar & (FATTR4_FILES_AVAIL_MASK |
749                             FATTR4_FILES_FREE_MASK |
750                             FATTR4_FILES_TOTAL_MASK |
751                             FATTR4_FILES_AVAIL_MASK |
752                             FATTR4_FILES_FREE_MASK |
753                             FATTR4_FILES_TOTAL_MASK)) {
754                                 if (error =
755                                     rfs4_get_sb_encode(dvp->v_vfsp,
```

```
                                    &sbe)) {
                                        /* Remove attrs from encode */
                                        ae &= ~(FATTR4_FILES_AVAIL_MASK |
                                            FATTR4_FILES_FREE_MASK |
                                            FATTR4_FILES_TOTAL_MASK |
                                            FATTR4_FILES_AVAIL_MASK |
                                            FATTR4_FILES_FREE_MASK |
                                            FATTR4_FILES_TOTAL_MASK);
                                        rddirattr_error = error;
                                }
                        }
                        if (ar & (FATTR4_MAXFILESIZE_MASK |
                            FATTR4_MAXLINK_MASK |
                            FATTR4_MAXNAME_MASK)) {
                                if (error = rfs4_get_pc_encode(cs->vp,
                                    &pce, ar, cs->cr)) {
                                        ar &= ~(FATTR4_MAXFILESIZE_MASK |
                                            FATTR4_MAXLINK_MASK |
                                            FATTR4_MAXNAME_MASK);
                                        rddirattr_error = error;
                                }
                        }
                }

reencode_attrs:
                /* encode the BOOLEAN for the existence of the next entry */
                IXDR_PUT_U_INT32(ptr, true);
                /* encode the COOKIE for the entry */
                IXDR_PUT_U_HYPER(ptr, dp->d_off);

                name = nfscmd_convname(ca, cs->exi, dp->d_name,
                    NFSCMD_CONV_OUTBOUND, MAXPATHLEN + 1);

                if (name == NULL) {
                        rddir_next_offset = dp->d_off;
                        continue;
                }
                /* Calculate the dirent name length */
                namelen = strlen(name);

                rndup = RNDUP(namelen) / BYTES_PER_XDR_UNIT;

                /* room for LENGTH + string ? */
                if ((ptr + (1 + rndup)) > ptr_redzone) {
                        no_space = TRUE;
                        continue;
                }

                /* encode the LENGTH of the name */
                IXDR_PUT_U_INT32(ptr, namelen);
                /* encode the RNDUP FILL first */
                ptr[rndup - 1] = 0;
                /* encode the NAME of the entry */
                bcopy(name, (char *)ptr, namelen);
                /* now bump the ptr after... */
                ptr += rndup;

                if (name != dp->d_name)
                        kmem_free(name, MAXPATHLEN + 1);

                /*
                 * Keep checking on the dircount to see if we have
                 * reached the limit; from the RFC, dircount is to be
                 * the XDR encoded limit of the cookie plus name.
                 * So the count is the name, XDR_UNIT of length for
                 * that name and 2 * XDR_UNIT bytes of cookie;
```



```
                 * However, use the regular DIRENT64 to match most
                 * client's APIs.
                 */
                dircount -= DIRENT64_RECLEN(namelen);
                if (nents != 0 && dircount < 0) {
                        no_space = TRUE;
                        continue;
                }

                /*
                 * Attributes requested?
                 * Gather up the attribute info and the previous VOP_LOOKUP()
                 * succeeded; if an error occurs on the VOP_GETATTR() then
                 * return just the error (again if it is requested).
                 * Note that the previous VOP_LOOKUP() could have failed
                 * itself which leaves this code without anything for
                 * a VOP_GETATTR().
                 * Also note that the readdir_attr_error is left in the
                 * encoding mask if requested and so is the mounted_on_fileid.
                 */
                if (ae != 0) {
                        if (!vp) {
                                ae = ar & (FATTR4_RDATTR_ERROR_MASK |
                                    FATTR4_MOUNTED_ON_FILEID_MASK);
                        } else {
                                va.va_mask = AT_ALL;
                                rddirattr_error =
                                    VOP_GETATTR(vp, &va, 0, cs->cr, NULL);
                                if (rddirattr_error) {
                                        ae = ar & (FATTR4_RDATTR_ERROR_MASK |
                                            FATTR4_MOUNTED_ON_FILEID_MASK);
                                } else {
                                        /*
                                         * We may lie about the object
                                         * type for a referral
                                         */
                                        if (vn_is_nfs_reparse(vp, cs->cr) &&
                                            client_is_downrev(req))
                                                va.va_type = VLNK;
                                }
                        }
                }

                /* START OF ATTRIBUTE ENCODING */

                /* encode the LENGTH of the BITMAP4 array */
                IXDR_PUT_U_INT32(ptr, 2);
                /* encode the BITMAP4 */
                attrmask_ptr = ptr;
                IXDR_PUT_HYPER(ptr, ae);
                attr_offset_ptr = ptr;
                /* encode the default LENGTH of the attributes for entry */
                IXDR_PUT_U_INT32(ptr, 0);

                if (ptr > ptr_redzone) {
                        no_space = TRUE;
                        continue;
                }

                /* Check if any of the first 32 attributes are being encoded */
                if (ae & 0xffffffff00000000) {
                        /*
                         * Redzone check is done at the end of this section.
                         * This particular section will encode a maximum of
                         * 18 * BYTES_PER_XDR_UNIT of data
                         */
```

```
888                         if (ae &
889                             (FATTR4_SUPPORTED_ATTRS_MASK |
890                             FATTR4_TYPE_MASK |
891                             FATTR4_FH_EXPIRE_TYPE_MASK |
892                             FATTR4_CHANGE_MASK |
893                             FATTR4_SIZE_MASK |
894                             FATTR4_LINK_SUPPORT_MASK |
895                             FATTR4_SYMLINK_SUPPORT_MASK |
896                             FATTR4_NAMED_ATTR_MASK |
897                             FATTR4_FSID_MASK |
898                             FATTR4_UNIQUE_HANDLES_MASK |
899                             FATTR4_LEASE_TIME_MASK |
900                             FATTR4_RDATTR_ERROR_MASK)) {
901
902                                 if (ae & FATTR4_SUPPORTED_ATTRS_MASK) {
903                                         IXDR_PUT_INT32(ptr, 2);
904                                         IXDR_PUT_HYPER(ptr,
905                                             rfs4_supported_attrs);
906                                 }
907                                 if (ae & FATTR4_TYPE_MASK) {
908                                         uint_t ftype = vt_to_nf4[va.va_type];
909                                         if (dvp->v_flag & V_XATTRDIR) {
910                                                 if (va.va_type == VDIR)
911                                                         ftype = NF4ATTRDIR;
912                                                 else
913                                                         ftype = NF4NAMEDATTR;
914                                         }
915                                         IXDR_PUT_U_INT32(ptr, ftype);
916                                 }
917                                 if (ae & FATTR4_FH_EXPIRE_TYPE_MASK) {
918                                         uint_t expire_type = FH4_PERSISTENT;
919                                         IXDR_PUT_U_INT32(ptr, expire_type);
920                                 }
921                                 if (ae & FATTR4_CHANGE_MASK) {
922                                         u_longlong_t change;
923                                         NFS4_SET_FATTR4_CHANGE(change,
924                                             va.va_ctime);
925                                         IXDR_PUT_HYPER(ptr, change);
926                                 }
927                                 if (ae & FATTR4_SIZE_MASK) {
928                                         u_longlong_t size = va.va_size;
929                                         IXDR_PUT_HYPER(ptr, size);
930                                 }
931                                 if (ae & FATTR4_LINK_SUPPORT_MASK) {
932                                         IXDR_PUT_U_INT32(ptr, true);
933                                 }
934                                 if (ae & FATTR4_SYMLINK_SUPPORT_MASK) {
935                                         IXDR_PUT_U_INT32(ptr, true);
936                                 }
937                                 if (ae & FATTR4_NAMED_ATTR_MASK) {
938                                         uint_t isit;
939                                         pc_val = FALSE;
940                                         int sattr_error;
941
942                                         if (!(vp->v_vfsp->vfs_flag &
943                                             VFS_XATTR)) {
944                                                 isit = FALSE;
945                                         } else {
946                                                 sattr_error = VOP_PATHCONF(vp,
947                                                     _PC_SATTR_EXISTS,
948                                                     &pc_val, cs->cr, NULL);
949                                                 if (sattr_error || pc_val == 0)
950                                                         (void) VOP_PATHCONF(vp,
951                                                             _PC_XATTR_EXISTS,
952                                                             &pc_val,
953                                                             cs->cr, NULL);
```



```
954                                         }
955                                         isit = (pc_val ? TRUE : FALSE);
956                                         IXDR_PUT_U_INT32(ptr, isit);
957                                 }
958                                 if (ae & FATTR4_FSID_MASK) {
959                                         u_longlong_t major, minor;
960                                         struct exportinfo *exi;
961
962                                         exi = newexi ? newexi : cs->exi;
963                                         if (exi->exi_volatile_dev) {
964                                                 int *pmaj = (int *)&major;
965
966                                                 pmaj[0] = exi->exi_fsid.val[0];
967                                                 pmaj[1] = exi->exi_fsid.val[1];
968                                                 minor = 0;
969                                         } else {
970                                                 major = getmajor(va.va_fsid);
971                                                 minor = getminor(va.va_fsid);
972                                         }
973                                         IXDR_PUT_HYPER(ptr, major);
974                                         IXDR_PUT_HYPER(ptr, minor);
975                                 }
976                                 if (ae & FATTR4_UNIQUE_HANDLES_MASK) {
977                                         IXDR_PUT_U_INT32(ptr, false);
978                                 }
979                                 if (ae & FATTR4_LEASE_TIME_MASK) {
980                                         uint_t lt = rfs4_lease_time;
981                                         IXDR_PUT_U_INT32(ptr, lt);
982                                 }
983                                 if (ae & FATTR4_RDATTR_ERROR_MASK) {
984                                         rddirattr_error =
985                                             (rddirattr_error == 0 ?
986                                             0 : puterrno4(rddirattr_error));
987                                         IXDR_PUT_U_INT32(ptr, rddirattr_error);
988                                 }
989
990                                 /* Check the redzone boundary */
991                                 if (ptr > ptr_redzone) {
992                                         if (nents || IS_MIN_ATTR_MASK(ar)) {
993                                                 no_space = TRUE;
994                                                 continue;
995                                         }
996                                         MINIMIZE_ATTR_MASK(ar);
997                                         ae = ar;
998                                         ptr = lastentry_ptr;
999                                         goto reencode_attrs;
1000                                }
1001                        }
1002                        /*
1003                         * Redzone check is done at the end of this section.
1004                         * This particular section will encode a maximum of
1005                         * 4 * BYTES_PER_XDR_UNIT of data.
1006                         * NOTE: that if ACLs are supported that the
1007                         * redzone calculations will need to change.
1008                         */
1009                        if (ae &
1010                            (FATTR4_ACL_MASK |
1011                            FATTR4_ACLSUPPORT_MASK |
1012                            FATTR4_ARCHIVE_MASK |
1013                            FATTR4_CANSETTIME_MASK |
1014                            FATTR4_CASE_INSENSITIVE_MASK |
1015                            FATTR4_CASE_PRESERVING_MASK |
1016                            FATTR4_CHOWN_RESTRICTED_MASK)) {
1017
1018                                if (ae & FATTR4_ACL_MASK) {
1019                                        ASSERT(0);
```

```
1020                                }
1021                                if (ae & FATTR4_ACLSUPPORT_MASK) {
1022                                        ASSERT(0);
1023                                }
1024                                if (ae & FATTR4_ARCHIVE_MASK) {
1025                                        ASSERT(0);
1026                                }
1027                                if (ae & FATTR4_CANSETTIME_MASK) {
1028                                        IXDR_PUT_U_INT32(ptr, true);
1029                                }
1030                                if (ae & FATTR4_CASE_INSENSITIVE_MASK) {
1031                                        IXDR_PUT_U_INT32(ptr, false);
1032                                }
1033                                if (ae & FATTR4_CASE_PRESERVING_MASK) {
1034                                        IXDR_PUT_U_INT32(ptr, true);
1035                                }
1036                                if (ae & FATTR4_CHOWN_RESTRICTED_MASK) {
1037                                        uint_t isit;
1038                                        pc_val = FALSE;
1039                                        (void) VOP_PATHCONF(vp,
1040                                            _PC_CHOWN_RESTRICTED,
1041                                            &pc_val, cs->cr, NULL);
1042                                        isit = (pc_val ? TRUE : FALSE);
1043                                        IXDR_PUT_U_INT32(ptr, isit);
1044                                }
1045                                /* Check the redzone boundary */
1046                                if (ptr > ptr_redzone) {
1047                                        if (nents || IS_MIN_ATTR_MASK(ar)) {
1048                                                no_space = TRUE;
1049                                                continue;
1050                                        }
1051                                        MINIMIZE_ATTR_MASK(ar);
1052                                        ae = ar;
1053                                        ptr = lastentry_ptr;
1054                                        goto reencode_attrs;
1055                                }
1056                        }
1057                        /*
1058                         * Redzone check is done before the filehandle
1059                         * is encoded.
1060                         */
1061                        if (ae &
1062                            (FATTR4_FILEHANDLE_MASK |
1063                            FATTR4_FILEID_MASK)) {
1064
1065                                if (ae & FATTR4_FILEHANDLE_MASK) {
1066                                        struct {
1067                                                uint_t len;
1068                                                char *val;
1069                                                char fh[NFS_FH4_LEN];
1070                                        } fh;
1071                                        fh.len = 0;
1072                                        fh.val = fh.fh;
1073                                        (void) makefh4((nfs_fh4 *)&fh, vp,
1074                                            (newexi ? newexi : cs->exi));
1075
1076                                        if (dvp->v_flag & V_XATTRDIR)
1077                                                set_fh4_flag((nfs_fh4 *)&fh,
1078                                                    FH4_NAMEDATTR);
1079
1080                                        if (!xdr_inline_encode_nfs_fh4(
1081                                            &ptr, ptr_redzone,
1082                                            (nfs_fh4_fmt_t *)fh.val)) {
1083                                                if (nents ||
1084                                                    IS_MIN_ATTR_MASK(ar)) {
1085                                                        no_space = TRUE;
```



```
1086                                                        continue;
1087                                                }
1088                                                MINIMIZE_ATTR_MASK(ar);
1089                                                ae = ar;
1090                                                ptr = lastentry_ptr;
1091                                                goto reencode_attrs;
1092                                        }
1093                                }
1094                                if (ae & FATTR4_FILEID_MASK) {
1095                                        IXDR_PUT_HYPER(ptr, va.va_nodeid);
1096                                }
1097                                /* Check the redzone boundary */
1098                                if (ptr > ptr_redzone) {
1099                                        if (nents || IS_MIN_ATTR_MASK(ar)) {
1100                                                no_space = TRUE;
1101                                                continue;
1102                                        }
1103                                        MINIMIZE_ATTR_MASK(ar);
1104                                        ae = ar;
1105                                        ptr = lastentry_ptr;
1106                                        goto reencode_attrs;
1107                                }
1108                        }
1109                        /*
1110                         * Redzone check is done at the end of this section.
1111                         * This particular section will encode a maximum of
1112                         * 15 * BYTES_PER_XDR_UNIT of data.
1113                         */
1114                        if (ae &
1115                            (FATTR4_FILES_AVAIL_MASK |
1116                            FATTR4_FILES_FREE_MASK |
1117                            FATTR4_FILES_TOTAL_MASK |
1118                            FATTR4_FS_LOCATIONS_MASK |
1119                            FATTR4_HIDDEN_MASK |
1120                            FATTR4_HOMOGENEOUS_MASK |
1121                            FATTR4_MAXFILESIZE_MASK |
1122                            FATTR4_MAXLINK_MASK |
1123                            FATTR4_MAXNAME_MASK |
1124                            FATTR4_MAXREAD_MASK |
1125                            FATTR4_MAXWRITE_MASK)) {
1126
1127                                if (ae & FATTR4_FILES_AVAIL_MASK) {
1128                                        IXDR_PUT_HYPER(ptr, sbe.fa);
1129                                }
1130                                if (ae & FATTR4_FILES_FREE_MASK) {
1131                                        IXDR_PUT_HYPER(ptr, sbe.ff);
1132                                }
1133                                if (ae & FATTR4_FILES_TOTAL_MASK) {
1134                                        IXDR_PUT_HYPER(ptr, sbe.ft);
1135                                }
1136                                if (ae & FATTR4_FS_LOCATIONS_MASK) {
1137                                        ASSERT(0);
1138                                }
1139                                if (ae & FATTR4_HIDDEN_MASK) {
1140                                        ASSERT(0);
1141                                }
1142                                if (ae & FATTR4_HOMOGENEOUS_MASK) {
1143                                        IXDR_PUT_U_INT32(ptr, true);
1144                                }
1145                                if (ae & FATTR4_MAXFILESIZE_MASK) {
1146                                        IXDR_PUT_HYPER(ptr, pce.maxfilesize);
1147                                }
1148                                if (ae & FATTR4_MAXLINK_MASK) {
1149                                        IXDR_PUT_U_INT32(ptr, pce.maxlink);
1150                                }
1151                                if (ae & FATTR4_MAXNAME_MASK) {
```

```
1152                                    IXDR_PUT_U_INT32(ptr, pce.maxname);
1153                            }
1154                            if (ae & FATTR4_MAXREAD_MASK) {
1155                                    IXDR_PUT_HYPER(ptr, maxread);
1156                            }
1157                            if (ae & FATTR4_MAXWRITE_MASK) {
1158                                    IXDR_PUT_HYPER(ptr, maxwrite);
1159                            }
1160                            /* Check the redzone boundary */
1161                            if (ptr > ptr_redzone) {
1162                                    if (nents || IS_MIN_ATTR_MASK(ar)) {
1163                                            no_space = TRUE;
1164                                            continue;
1165                                    }
1166                                    MINIMIZE_ATTR_MASK(ar);
1167                                    ae = ar;
1168                                    ptr = lastentry_ptr;
1169                                    goto reencode_attrs;
1170                            }
1171                    }
1172            }
1173            if (ae & 0x00000000ffffffff) {
1174                    /*
1175                     * Redzone check is done at the end of this section.
1176                     * This particular section will encode a maximum of
1177                     * 3 * BYTES_PER_XDR_UNIT of data.
1178                     */
1179                    if (ae &
1180                        (FATTR4_MIMETYPE_MASK |
1181                        FATTR4_MODE_MASK |
1182                        FATTR4_NO_TRUNC_MASK |
1183                        FATTR4_NUMLINKS_MASK)) {
1184
1185                            if (ae & FATTR4_MIMETYPE_MASK) {
1186                                    ASSERT(0);
1187                            }
1188                            if (ae & FATTR4_MODE_MASK) {
1189                                    uint_t m = va.va_mode;
1190                                    IXDR_PUT_U_INT32(ptr, m);
1191                            }
1192                            if (ae & FATTR4_NO_TRUNC_MASK) {
1193                                    IXDR_PUT_U_INT32(ptr, true);
1194                            }
1195                            if (ae & FATTR4_NUMLINKS_MASK) {
1196                                    IXDR_PUT_U_INT32(ptr, va.va_nlink);
1197                            }
1198                            /* Check the redzone boundary */
1199                            if (ptr > ptr_redzone) {
1200                                    if (nents || IS_MIN_ATTR_MASK(ar)) {
1201                                            no_space = TRUE;
1202                                            continue;
1203                                    }
1204                                    MINIMIZE_ATTR_MASK(ar);
1205                                    ae = ar;
1206                                    ptr = lastentry_ptr;
1207                                    goto reencode_attrs;
1208                            }
1209                    }
1210                    /*
1211                     * Redzone check is done before the encoding of the
1212                     * owner string since the length is indeterminate.
1213                     */
1214                    if (ae & FATTR4_OWNER_MASK) {
1215                            if (!lu_set) {
1216                                    owner_error = nfs_idmap_uid_str(
1217                                        va.va_uid, &owner, TRUE);
```



```
1218                                    if (!owner_error) {
1219                                            lu_set = TRUE;
1220                                            lastuid = va.va_uid;
1221                                    }
1222                            } else  if (va.va_uid != lastuid) {
1223                                    if (owner.utf8string_len != 0) {
1224                                            kmem_free(owner.utf8string_val,
1225                                                owner.utf8string_len);
1226                                            owner.utf8string_len = 0;
1227                                            owner.utf8string_val = NULL;
1228                                    }
1229                                    owner_error = nfs_idmap_uid_str(
1230                                        va.va_uid, &owner, TRUE);
1231                                    if (!owner_error) {
1232                                            lastuid = va.va_uid;
1233                                    } else {
1234                                            lu_set = FALSE;
1235                                    }
1236                            }
1237                            if (!owner_error) {
1238                                    if ((ptr +
1239                                        (owner.utf8string_len /
1240                                        BYTES_PER_XDR_UNIT)
1241                                        + 2) > ptr_redzone) {
1242                                            if (nents ||
1243                                                IS_MIN_ATTR_MASK(ar)) {
1244                                                    no_space = TRUE;
1245                                                    continue;
1246                                            }
1247                                            MINIMIZE_ATTR_MASK(ar);
1248                                            ae = ar;
1249                                            ptr = lastentry_ptr;
1250                                            goto reencode_attrs;
1251                                    }
1252                                    /* encode the LENGTH of owner string */
1253                                    IXDR_PUT_U_INT32(ptr,
1254                                        owner.utf8string_len);
1255                                    /* encode the RNDUP FILL first */
1256                                    rndup = RNDUP(owner.utf8string_len) /
1257                                        BYTES_PER_XDR_UNIT;
1258                                    ptr[rndup - 1] = 0;
1259                                    /* encode the OWNER */
1260                                    bcopy(owner.utf8string_val, ptr,
1261                                        owner.utf8string_len);
1262                                    ptr += rndup;
1263                            }
1264                    }
1265                    /*
1266                     * Redzone check is done before the encoding of the
1267                     * group string since the length is indeterminate.
1268                     */
1269                    if (ae & FATTR4_OWNER_GROUP_MASK) {
1270                            if (!lg_set) {
1271                                    group_error =
1272                                        nfs_idmap_gid_str(va.va_gid,
1273                                        &group, TRUE);
1274                                    if (!group_error) {
1275                                            lg_set = TRUE;
1276                                            lastgid = va.va_gid;
1277                                    }
1278                            } else if (va.va_gid != lastgid) {
1279                                    if (group.utf8string_len != 0) {
1280                                            kmem_free(
1281                                                group.utf8string_val,
1282                                                group.utf8string_len);
1283                                            group.utf8string_len = 0;
```

```
1284                                        group.utf8string_val = NULL;
1285                                }
1286                                group_error =
1287                                    nfs_idmap_gid_str(va.va_gid,
1288                                    &group, TRUE);
1289                                if (!group_error)
1290                                        lastgid = va.va_gid;
1291                                else
1292                                        lg_set = FALSE;
1293                        }
1294                        if (!group_error) {
1295                                if ((ptr +
1296                                    (group.utf8string_len /
1297                                    BYTES_PER_XDR_UNIT)
1298                                    + 2) > ptr_redzone) {
1299                                        if (nents ||
1300                                            IS_MIN_ATTR_MASK(ar)) {
1301                                                no_space = TRUE;
1302                                                continue;
1303                                        }
1304                                        MINIMIZE_ATTR_MASK(ar);
1305                                        ae = ar;
1306                                        ptr = lastentry_ptr;
1307                                        goto reencode_attrs;
1308                                }
1309                                /* encode the LENGTH of owner string */
1310                                IXDR_PUT_U_INT32(ptr,
1311                                    group.utf8string_len);
1312                                /* encode the RNDUP FILL first */
1313                                rndup = RNDUP(group.utf8string_len) /
1314                                    BYTES_PER_XDR_UNIT;
1315                                ptr[rndup - 1] = 0;
1316                                /* encode the OWNER */
1317                                bcopy(group.utf8string_val, ptr,
1318                                    group.utf8string_len);
1319                                ptr += rndup;
1320                        }
1321                }
1322                if (ae &
1323                    (FATTR4_QUOTA_AVAIL_HARD_MASK |
1324                    FATTR4_QUOTA_AVAIL_SOFT_MASK |
1325                    FATTR4_QUOTA_USED_MASK)) {
1326                        if (ae & FATTR4_QUOTA_AVAIL_HARD_MASK) {
1327                                ASSERT(0);
1328                        }
1329                        if (ae & FATTR4_QUOTA_AVAIL_SOFT_MASK) {
1330                                ASSERT(0);
1331                        }
1332                        if (ae & FATTR4_QUOTA_USED_MASK) {
1333                                ASSERT(0);
1334                        }
1335                }
1336                /*
1337                 * Redzone check is done at the end of this section.
1338                 * This particular section will encode a maximum of
1339                 * 10 * BYTES_PER_XDR_UNIT of data.
1340                 */
1341                if (ae &
1342                    (FATTR4_RAWDEV_MASK |
1343                    FATTR4_SPACE_AVAIL_MASK |
1344                    FATTR4_SPACE_FREE_MASK |
1345                    FATTR4_SPACE_TOTAL_MASK |
1346                    FATTR4_SPACE_USED_MASK |
1347                    FATTR4_SYSTEM_MASK)) {

1349                        if (ae & FATTR4_RAWDEV_MASK) {
```



```
1350                                fattr4_rawdev rd;
1351                                rd.specdata1 =
1352                                    (uint32)getmajor(va.va_rdev);
1353                                rd.specdata2 =
1354                                    (uint32)getminor(va.va_rdev);
1355                                IXDR_PUT_U_INT32(ptr, rd.specdata1);
1356                                IXDR_PUT_U_INT32(ptr, rd.specdata2);
1357                        }
1358                        if (ae & FATTR4_SPACE_AVAIL_MASK) {
1359                                IXDR_PUT_HYPER(ptr, sbe.space_avail);
1360                        }
1361                        if (ae & FATTR4_SPACE_FREE_MASK) {
1362                                IXDR_PUT_HYPER(ptr, sbe.space_free);
1363                        }
1364                        if (ae & FATTR4_SPACE_TOTAL_MASK) {
1365                                IXDR_PUT_HYPER(ptr, sbe.space_total);
1366                        }
1367                        if (ae & FATTR4_SPACE_USED_MASK) {
1368                                u_longlong_t su;
1369                                su = (fattr4_space_used) DEV_BSIZE *
1370                                    (fattr4_space_used) va.va_nblocks;
1371                                IXDR_PUT_HYPER(ptr, su);
1372                        }
1373                        if (ae & FATTR4_SYSTEM_MASK) {
1374                                ASSERT(0);
1375                        }
1376                        /* Check the redzone boundary */
1377                        if (ptr > ptr_redzone) {
1378                                if (nents || IS_MIN_ATTR_MASK(ar)) {
1379                                        no_space = TRUE;
1380                                        continue;
1381                                }
1382                                MINIMIZE_ATTR_MASK(ar);
1383                                ae = ar;
1384                                ptr = lastentry_ptr;
1385                                goto reencode_attrs;
1386                        }
1387                }
1388                /*
1389                 * Redzone check is done at the end of this section.
1390                 * This particular section will encode a maximum of
1391                 * 14 * BYTES_PER_XDR_UNIT of data.
1392                 */
1393                if (ae &
1394                    (FATTR4_TIME_ACCESS_MASK |
1395                    FATTR4_TIME_ACCESS_SET_MASK |
1396                    FATTR4_TIME_BACKUP_MASK |
1397                    FATTR4_TIME_CREATE_MASK |
1398                    FATTR4_TIME_DELTA_MASK |
1399                    FATTR4_TIME_METADATA_MASK |
1400                    FATTR4_TIME_MODIFY_MASK |
1401                    FATTR4_TIME_MODIFY_SET_MASK |
1402                    FATTR4_MOUNTED_ON_FILEID_MASK)) {

1404                        if (ae & FATTR4_TIME_ACCESS_MASK) {
1405                                u_longlong_t sec =
1406                                    (u_longlong_t)va.va_atime.tv_sec;
1407                                uint_t nsec =
1408                                    (uint_t)va.va_atime.tv_nsec;
1409                                IXDR_PUT_HYPER(ptr, sec);
1410                                IXDR_PUT_INT32(ptr, nsec);
1411                        }
1412                        if (ae & FATTR4_TIME_ACCESS_SET_MASK) {
1413                                ASSERT(0);
1414                        }
1415                        if (ae & FATTR4_TIME_BACKUP_MASK) {
```

```
1416                                    ASSERT(0);
1417                            }
1418                            if (ae & FATTR4_TIME_CREATE_MASK) {
1419                                    ASSERT(0);
1420                            }
1421                            if (ae & FATTR4_TIME_DELTA_MASK) {
1422                                    u_longlong_t sec = 0;
1423                                    uint_t nsec = 1000;
1424                                    IXDR_PUT_HYPER(ptr, sec);
1425                                    IXDR_PUT_INT32(ptr, nsec);
1426                            }
1427                            if (ae & FATTR4_TIME_METADATA_MASK) {
1428                                    u_longlong_t sec =
1429                                        (u_longlong_t)va.va_ctime.tv_sec;
1430                                    uint_t nsec =
1431                                        (uint_t)va.va_ctime.tv_nsec;
1432                                    IXDR_PUT_HYPER(ptr, sec);
1433                                    IXDR_PUT_INT32(ptr, nsec);
1434                            }
1435                            if (ae & FATTR4_TIME_MODIFY_MASK) {
1436                                    u_longlong_t sec =
1437                                        (u_longlong_t)va.va_mtime.tv_sec;
1438                                    uint_t nsec =
1439                                        (uint_t)va.va_mtime.tv_nsec;
1440                                    IXDR_PUT_HYPER(ptr, sec);
1441                                    IXDR_PUT_INT32(ptr, nsec);
1442                            }
1443                            if (ae & FATTR4_TIME_MODIFY_SET_MASK) {
1444                                    ASSERT(0);
1445                            }
1446                            if (ae & FATTR4_MOUNTED_ON_FILEID_MASK) {
1447                                    IXDR_PUT_HYPER(ptr, dp->d_ino);
1448                            }
1449                            /* Check the redzone boundary */
1450                            if (ptr > ptr_redzone) {
1451                                    if (nents || IS_MIN_ATTR_MASK(ar)) {
1452                                            no_space = TRUE;
1453                                            continue;
1454                                    }
1455                                    MINIMIZE_ATTR_MASK(ar);
1456                                    ae = ar;
1457                                    ptr = lastentry_ptr;
1458                                    goto reencode_attrs;
1459                            }
1460                    }
1461            }

1463            /* Reset to directory's vfs info when encoding complete */
1464            if (vfs_different) {
1465                    dsbe = sbe;
1466                    dpce = pce;
1467                    vfs_different = 0;
1468            }

1470            /* "go back" and encode the attributes' length */
1471            attr_length =
1472                (char *)ptr -
1473                (char *)attr_offset_ptr -
1474                BYTES_PER_XDR_UNIT;
1475            IXDR_PUT_U_INT32(attr_offset_ptr, attr_length);

1477            /*
1478             * If there was trouble obtaining a mapping for either
1479             * the owner or group attributes, then remove them from
1480             * bitmap4 for this entry and reset the bitmap value
1481             * in the data stream.
```



```
1482                     */
1483                    if (owner_error || group_error) {
1484                            if (owner_error)
1485                                    ae &= ~FATTR4_OWNER_MASK;
1486                            if (group_error)
1487                                    ae &= ~FATTR4_OWNER_GROUP_MASK;
1488                            IXDR_PUT_HYPER(attrmask_ptr, ae);
1489                    }

1491                    /* END OF ATTRIBUTE ENCODING */

1493                    lastentry_ptr = ptr;
1494                    nents++;
1495                    rddir_next_offset = dp->d_off;
1496            }

1498            if (newexi) {
1499                    exi_rele(newexi);
1500                    newexi = NULL;
1501            }

1503    #endif /* ! codereview */
1504            /*
1505             * Check for the case that another VOP_READDIR() has to be done.
1506             * - no space encoding error
1507             * - no entry successfully encoded
1508             * - still more directory to read
1509             */
1510            if (!no_space && nents == 0 && !iseofdir)
1511                    goto readagain;

1513            *cs->statusp = resp->status = NFS4_OK;

1515            /*
1516             * If no_space is set then we terminated prematurely,
1517             * rewind to the last entry and this can never be EOF.
1518             */
1519            if (no_space) {
1520                    ptr = lastentry_ptr;
1521                    eof = FALSE; /* ended encoded prematurely */
1522            } else {
1523                    eof = (iseofdir ? TRUE : FALSE);
1524            }

1526            /*
1527             * If we have entries, always return them, otherwise only error
1528             * if we ran out of space.
1529             */
1530            if (nents || !no_space) {
1531                    ASSERT(ptr != NULL);
1532                    /* encode the BOOLEAN marking no further entries */
1533                    IXDR_PUT_U_INT32(ptr, false);
1534                    /* encode the BOOLEAN signifying end of directory */
1535                    IXDR_PUT_U_INT32(ptr, eof);

1537                    resp->data_len = (char *)ptr - (char *)beginning_ptr;
1538                    resp->mblk->b_wptr += resp->data_len;
1539            } else {
1540                    freeb(mp);
1541                    resp->mblk = NULL;
1542                    resp->data_len = 0;
1543                    *cs->statusp = resp->status = NFS4ERR_TOOSMALL;
1544            }

1546            kmem_free((caddr_t)rddir_data, rddir_data_len);
1547            if (vp)
```

```
1548                 VN_RELE(vp);
1549         if (owner.utf8string_len != 0)
1550                 kmem_free(owner.utf8string_val, owner.utf8string_len);
1551         if (group.utf8string_len != 0)
1552                 kmem_free(group.utf8string_val, group.utf8string_len);

1554 out:
1555         DTRACE_NFSV4_2(op__readdir__done, struct compound_state *, cs,
1556             READDIR4res *, resp);
1557 }
```

```
**********************************************************
   69864 Wed Sep 14 16:21:03 2016
new/usr/src/uts/common/fs/nfs/nfs_export.c
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_

1888 /*
1889  * Strategy: if vp is in the export list, then
1890  * return the associated file handle. Otherwise, ".."
1891  * once up the vp and try again, until the root of the
1892  * filesystem is reached.
1893  */
1894 struct   exportinfo *
1895 nfs_vptoexi(vnode_t *dvp, vnode_t *vp, cred_t *cr, int *walk,
1896         int *err,  bool_t v4srv)
1897 {
1898         fid_t fid;
1899         int error;
1900         struct exportinfo *exi;

1902         ASSERT(vp);
1903         VN_HOLD(vp);
1904         if (dvp != NULL) {
1905                 VN_HOLD(dvp);
1906         }
1907         if (walk != NULL)
1908                 *walk = 0;

1910         for (;;) {
1911                 bzero(&fid, sizeof (fid));
1912                 fid.fid_len = MAXFIDSZ;
1913                 error = vop_fid_pseudo(vp, &fid);
1914                 if (error) {
1915                         /*
1916                          * If vop_fid_pseudo returns ENOSPC then the fid
1917                          * supplied is too small. For now we simply
1918                          * return EREMOTE.
1919                          */
1920                         if (error == ENOSPC)
1921                                 error = EREMOTE;
1922                         break;
1923                 }

1925                 exi = checkexport(&vp->v_vfsp->vfs_fsid, &fid,
1926                     v4srv ? vp : NULL);
1925                 if (v4srv)
1926                         exi = checkexport4(&vp->v_vfsp->vfs_fsid, &fid, vp);
1927                 else
1928                         exi = checkexport(&vp->v_vfsp->vfs_fsid, &fid);

1927                 if (exi != NULL) {
1928                         /*
1929                          * Found the export info
1930                          */
1931                         break;
1932                 }

1934                 /*
1935                  * We have just failed finding a matching export.
1936                  * If we're at the root of this filesystem, then
1937                  * it's time to stop (with failure).
1938                  */
1939                 if (vp->v_flag & VROOT) {
1940                         error = EINVAL;
1941                         break;
```



```
1942                 }

1944                 if (walk != NULL)
1945                         (*walk)++;

1947                 /*
1948                  * Now, do a ".." up vp. If dvp is supplied, use it,
1949                  * otherwise, look it up.
1950                  */
1951                 if (dvp == NULL) {
1952                         error = VOP_LOOKUP(vp, "..", &dvp, NULL, 0, NULL, cr,
1953                             NULL, NULL, NULL);
1954                         if (error)
1955                                 break;
1956                 }
1957                 VN_RELE(vp);
1958                 vp = dvp;
1959                 dvp = NULL;
1960         }
1961         VN_RELE(vp);
1962         if (dvp != NULL) {
1963                 VN_RELE(dvp);
1964         }
1965         if (error != 0) {
1966                 if (err != NULL)
1967                         *err = error;
1968                 return (NULL);
1969         }
1970         return (exi);
1971 }
______unchanged_portion_omitted_

2434 /*
2435  * Find the export structure associated with the given filesystem.
2436  * If found, then increment the ref count (exi_count).
2437  */
2431 struct exportinfo *
2432 checkexport_nohold(fsid_t *fsid, fid_t *fid, vnode_t *vp)
2439 checkexport(fsid_t *fsid, fid_t *fid)
2433 {
2434         struct exportinfo *exi;

2443         rw_enter(&exported_lock, RW_READER);
2436         for (exi = exptable[exptablehash(fsid, fid)];
2437             exi != NULL;
2438             exi = exi->fid_hash.next) {
2439                 if (exportmatch(exi, fsid, fid)) {
2440                         /*
2441                          * If this is the place holder for the
2442                          * public file handle, then return the
2443                          * real export entry for the public file
2444                          * handle.
2445                          */
2446                         if (exi->exi_export.ex_flags & EX_PUBLIC) {
2447                                 exi = exi_public;
2448                         }

2450                         /*
2451                          * If vp is given, check if vp is the
2452                          * same vnode as the exported node.
2453                          *
2454                          * Since VOP_FID of a lofs node returns the
2455                          * fid of its real node (ufs), the exported
2456                          * node for lofs and (pseudo) ufs may have
2457                          * the same fsid and fid.
2458                          */
```

```
                        if (vp == NULL || vp == exi->exi_vp) {
                        exi_hold(exi);
                        rw_exit(&exported_lock);
                                return (exi);
                        }
                }
        }
        rw_exit(&exported_lock);
        return (NULL);
}


/*
 * Find the export structure associated with the given filesystem.
 * If found, then increment the ref count (exi_count).
 * "old school" version of checkexport() for NFS4.  NFS4
 * rfs4_compound holds exported_lock for duration of compound
 * processing.  This version doesn't manipulate exi_count
 * since NFS4 breaks fundamental assumptions in the exi_count
 * design.
 */
struct exportinfo *
checkexport(fsid_t *fsid, fid_t *fid, vnode_t *vp)
checkexport4(fsid_t *fsid, fid_t *fid, vnode_t *vp)
{
        struct exportinfo *exi;

        rw_enter(&exported_lock, RW_READER);
        exi = checkexport_nohold(fsid, fid, vp);
        if (exi)
                exi_hold(exi);
        rw_exit(&exported_lock);
        ASSERT(RW_LOCK_HELD(&exported_lock));

        for (exi = exptable[exptablehash(fsid, fid)];
            exi != NULL;
            exi = exi->fid_hash.next) {
                if (exportmatch(exi, fsid, fid)) {
                        /*
                         * If this is the place holder for the
                         * public file handle, then return the
                         * real export entry for the public file
                         * handle.
                         */
                        if (exi->exi_export.ex_flags & EX_PUBLIC) {
                                exi = exi_public;
                        }

                        /*
                         * If vp is given, check if vp is the
                         * same vnode as the exported node.
                         *
                         * Since VOP_FID of a lofs node returns the
                         * fid of its real node (ufs), the exported
                         * node for lofs and (pseudo) ufs may have
                         * the same fsid and fid.
                         */
                        if (vp == NULL || vp == exi->exi_vp)
        return (exi);
                        }
                }

        return (NULL);
}
```

_____unchanged_portion_omitted_

****
48876 Wed Sep 14 16:21:03 2016
new/usr/src/uts/common/fs/nfs/nfs_log.c
7378 exported_lock held during nfs4 compound processing
****
_____***unchanged_portion_omitted_***

```
1520 static int      nfslog_dispatch_table_arglen = sizeof (nfslog_dispatch_table) /
1521                                       sizeof (nfslog_dispatch_table[0]);

1523 /*
1524  * This function will determine the appropriate export info struct to use
1525  * and allocate a record id to be used in the written log buffer.
1526  * Usually this is a straightforward operation but the existence of the
1527  * multicomponent lookup and its semantics of crossing file system
1528  * boundaries add to the complexity.  See the comments below...
1529  */
1530 struct exportinfo *
1531 nfslog_get_exi(
1532         struct exportinfo *exi,
1533         struct svc_req *req,
1534         caddr_t res,
1535         unsigned int *nfslog_rec_id)
1536 {
1537         struct log_buffer *lb;
1538         struct exportinfo *exi_ret = NULL;
1539         fhandle_t               *fh;
1540         nfs_fh3                 *fh3;

1542         if (exi == NULL)
1543                 return (NULL);

1545         /*
1546          * If the exi is marked for logging, allocate a record id and return
1547          */
1548         if (exi->exi_export.ex_flags & EX_LOG) {
1549                 lb = exi->exi_logbuffer;

1551                 /* obtain the unique record id for the caller */
1552                 *nfslog_rec_id = atomic_add_32_nv(&lb->lb_rec_id, (int32_t)1);

1554                 /*
1555                  * The caller will expect to be able to exi_rele() it,
1556                  * so exi->exi_count must be incremented before it can
1557                  * be returned, to make it uniform with exi_ret->exi_count
1558                  */
1559                 exi_hold(exi);
1560                 return (exi);
1561         }

1563         if (exi != exi_public)
1564                 return (NULL);

1566         /*
1567          * Here we have an exi that is not marked for logging.
1568          * It is possible that this request is a multicomponent lookup
1569          * that was done from the public file handle (not logged) and
1570          * the resulting file handle being returned to the client exists
1571          * in a file system that is being logged.  If this is the case
1572          * we need to log this multicomponent lookup to the appropriate
1573          * log buffer.  This will allow for the appropriate path name
1574          * mapping to occur at user level.
1575          */
1576         if (req->rq_prog == NFS_PROGRAM) {
1577                 switch (req->rq_vers) {
1578                 case NFS_V3:
```



```
1579                         if ((req->rq_proc == NFSPROC3_LOOKUP) &&
1580                             (((LOOKUP3res *)res)->status == NFS3_OK)) {
1581                                 fh3 = &((LOOKUP3res *)res)->res_u.ok.object;
1582                                 exi_ret = checkexport(&fh3->fh3_fsid,
1583                                     FH3TOXFIDP(fh3), NULL);
1583                                     FH3TOXFIDP(fh3));
1584                         }
1585                         break;

1587                 case NFS_VERSION:
1588                         if ((req->rq_proc == RFS_LOOKUP) &&
1589                             (((struct nfsdiropres *)
1590                             res)->dr_status == NFS_OK)) {
1591                                 fh =  &((struct nfsdiropres *)res)->
1592                                     dr_u.dr_drok_u.drok_fhandle;
1593                                 exi_ret = checkexport(&fh->fh_fsid,
1594                                     (fid_t *)&fh->fh_xlen, NULL);
1594                                     (fid_t *)&fh->fh_xlen);
1595                         }
1596                         break;
1597                 default:
1598                         break;
1599                 }
1600         }

1602         if (exi_ret != NULL && exi_ret->exi_export.ex_flags & EX_LOG) {
1603                 lb = exi_ret->exi_logbuffer;
1604                 /* obtain the unique record id for the caller */
1605                 *nfslog_rec_id = atomic_add_32_nv(&lb->lb_rec_id, (int32_t)1);

1607                 return (exi_ret);
1608         }
1609         return (NULL);
1610 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   85730 Wed Sep 14 16:21:03 2016
new/usr/src/uts/common/fs/nfs/nfs_server.c
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_
```

```
1473 static void
1474 common_dispatch(struct svc_req *req, SVCXPRT *xprt, rpcvers_t min_vers,
1475                 rpcvers_t max_vers, char *pgmname,
1476                 struct rpc_disptable *disptable)
1477 {
1478         int which;
1479         rpcvers_t vers;
1480         char *args;
1481         union {
1482                         union rfs_args ra;
1483                         union acl_args aa;
1484                 } args_buf;
1485         char *res;
1486         union {
1487                         union rfs_res rr;
1488                         union acl_res ar;
1489                 } res_buf;
1490         struct rpcdisp *disp = NULL;
1491         int dis_flags = 0;
1492         cred_t *cr;
1493         int error = 0;
1494         int anon_ok;
1495         struct exportinfo *exi = NULL;
1496         unsigned int nfslog_rec_id;
1497         int dupstat;
1498         struct dupreq *dr;
1499         int authres;
1500         bool_t publicfh_ok = FALSE;
1501         enum_t auth_flavor;
1502         bool_t dupcached = FALSE;
1503         struct netbuf   nb;
1504         bool_t logging_enabled = FALSE;
1505         struct exportinfo *nfslog_exi = NULL;
1506         char **procnames;
1507         char cbuf[INET6_ADDRSTRLEN];    /* to hold both IPv4 and IPv6 addr */
1508         bool_t ro = FALSE;

1510         vers = req->rq_vers;

1512         if (vers < min_vers || vers > max_vers) {
1513                 svcerr_progvers(req->rq_xprt, min_vers, max_vers);
1514                 error++;
1515                 cmn_err(CE_NOTE, "%s: bad version number %u", pgmname, vers);
1516                 goto done;
1517         }
1518         vers -= min_vers;

1520         which = req->rq_proc;
1521         if (which < 0 || which >= disptable[(int)vers].dis_nprocs) {
1522                 svcerr_noproc(req->rq_xprt);
1523                 error++;
1524                 goto done;
1525         }

1527         (*(disptable[(int)vers].dis_proccntp))[which].value.ui64++;

1529         disp = &disptable[(int)vers].dis_table[which];
1530         procnames = disptable[(int)vers].dis_procnames;
```



```
1532         auth_flavor = req->rq_cred.oa_flavor;

1534         /*
1535          * Deserialize into the args struct.
1536          */
1537         args = (char *)&args_buf;

1539 #ifdef DEBUG
1540         if (rfs_no_fast_xdrargs || (auth_flavor == RPCSEC_GSS) ||
1541             disp->dis_fastxdrargs == NULL_xdrproc_t ||
1542             !SVC_GETARGS(xprt, disp->dis_fastxdrargs, (char *)&args))
1543 #else
1544         if ((auth_flavor == RPCSEC_GSS) ||
1545             disp->dis_fastxdrargs == NULL_xdrproc_t ||
1546             !SVC_GETARGS(xprt, disp->dis_fastxdrargs, (char *)&args))
1547 #endif
1548         {
1549                 bzero(args, disp->dis_argsz);
1550                 if (!SVC_GETARGS(xprt, disp->dis_xdrargs, args)) {
1551                         error++;
1552                         /*
1553                          * Check if we are outside our capabilities.
1554                          */
1555                         if (rfs4_minorvers_mismatch(req, xprt, (void *)args))
1556                                 goto done;

1558                         svcerr_decode(xprt);
1559                         cmn_err(CE_NOTE,
1560                             "Failed to decode arguments for %s version %u "
1561                             "procedure %s client %s%s",
1562                             pgmname, vers + min_vers, procnames[which],
1563                             client_name(req), client_addr(req, cbuf));
1564                         goto done;
1565                 }
1566         }

1568         /*
1569          * If Version 4 use that specific dispatch function.
1570          */
1571         if (req->rq_vers == 4) {
1572                 error += rfs4_dispatch(disp, req, xprt, args);
1573                 goto done;
1574         }

1576         dis_flags = disp->dis_flags;

1578         /*
1579          * Find export information and check authentication,
1580          * setting the credential if everything is ok.
1581          */
1582         if (disp->dis_getfh != NULL) {
1583                 void *fh;
1584                 fsid_t *fsid;
1585                 fid_t *fid, *xfid;
1586                 fhandle_t *fh2;
1587                 nfs_fh3 *fh3;

1589                 fh = (*disp->dis_getfh)(args);
1590                 switch (req->rq_vers) {
1591                 case NFS_VERSION:
1592                         fh2 = (fhandle_t *)fh;
1593                         fsid = &fh2->fh_fsid;
1594                         fid = (fid_t *)&fh2->fh_len;
1595                         xfid = (fid_t *)&fh2->fh_xlen;
1596                         break;
```

```
1597                 case NFS_V3:
1598                         fh3 = (nfs_fh3 *)fh;
1599                         fsid = &fh3->fh3_fsid;
1600                         fid = FH3TOFIDP(fh3);
1601                         xfid = FH3TOXFIDP(fh3);
1602                         break;
1603                 }

1605                 /*
1606                  * Fix for bug 1038302 - corbin
1607                  * There is a problem here if anonymous access is
1608                  * disallowed.  If the current request is part of the
1609                  * client's mount process for the requested filesystem,
1610                  * then it will carry root (uid 0) credentials on it, and
1611                  * will be denied by checkauth if that client does not
1612                  * have explicit root=0 permission.  This will cause the
1613                  * client's mount operation to fail.  As a work-around,
1614                  * we check here to see if the request is a getattr or
1615                  * statfs operation on the exported vnode itself, and
1616                  * pass a flag to checkauth with the result of this test.
1617                  *
1618                  * The filehandle refers to the mountpoint itself if
1619                  * the fh_data and fh_xdata portions of the filehandle
1620                  * are equal.
1621                  *
1622                  * Added anon_ok argument to checkauth().
1623                  */

1625                 if ((dis_flags & RPC_ALLOWANON) && EQFID(fid, xfid))
1626                         anon_ok = 1;
1627                 else
1628                         anon_ok = 0;

1630                 cr = xprt->xp_cred;
1631                 ASSERT(cr != NULL);
1632 #ifdef DEBUG
1633                 if (crgetref(cr) != 1) {
1634                         crfree(cr);
1635                         cr = crget();
1636                         xprt->xp_cred = cr;
1637                         cred_misses++;
1638                 } else
1639                         cred_hits++;
1640 #else
1641                 if (crgetref(cr) != 1) {
1642                         crfree(cr);
1643                         cr = crget();
1644                         xprt->xp_cred = cr;
1645                 }
1646 #endif

1648                 exi = checkexport(fsid, xfid, NULL);
1648                 exi = checkexport(fsid, xfid);

1650                 if (exi != NULL) {
1651                         publicfh_ok = PUBLICFH_CHECK(disp, exi, fsid, xfid);

1653                         /*
1654                          * Don't allow non-V4 clients access
1655                          * to pseudo exports
1656                          */
1657                         if (PSEUDO(exi)) {
1658                                 svcerr_weakauth(xprt);
1659                                 error++;
1660                                 goto done;
1661                         }
```



```
1663                         authres = checkauth(exi, req, cr, anon_ok, publicfh_ok,
1664                             &ro);
1665                         /*
1666                          * authres >  0: authentication OK - proceed
1667                          * authres == 0: authentication weak - return error
1668                          * authres <  0: authentication timeout - drop
1669                          */
1670                         if (authres <= 0) {
1671                                 if (authres == 0) {
1672                                         svcerr_weakauth(xprt);
1673                                         error++;
1674                                 }
1675                                 goto done;
1676                         }
1677                 }
1678         } else
1679                 cr = NULL;

1681         if ((dis_flags & RPC_MAPRESP) && (auth_flavor != RPCSEC_GSS)) {
1682                 res = (char *)SVC_GETRES(xprt, disp->dis_ressz);
1683                 if (res == NULL)
1684                         res = (char *)&res_buf;
1685         } else
1686                 res = (char *)&res_buf;

1688         if (!(dis_flags & RPC_IDEMPOTENT)) {
1689                 dupstat = SVC_DUP_EXT(xprt, req, res, disp->dis_ressz, &dr,
1690                     &dupcached);

1692                 switch (dupstat) {
1693                 case DUP_ERROR:
1694                         svcerr_systemerr(xprt);
1695                         error++;
1696                         goto done;
1697                         /* NOTREACHED */
1698                 case DUP_INPROGRESS:
1699                         if (res != (char *)&res_buf)
1700                                 SVC_FREERES(xprt);
1701                         error++;
1702                         goto done;
1703                         /* NOTREACHED */
1704                 case DUP_NEW:
1705                 case DUP_DROP:
1706                         curthread->t_flag |= T_DONTPEND;

1708                         (*disp->dis_proc)(args, res, exi, req, cr, ro);

1710                         curthread->t_flag &= ~T_DONTPEND;
1711                         if (curthread->t_flag & T_WOULDBLOCK) {
1712                                 curthread->t_flag &= ~T_WOULDBLOCK;
1713                                 SVC_DUPDONE_EXT(xprt, dr, res, NULL,
1714                                     disp->dis_ressz, DUP_DROP);
1715                                 if (res != (char *)&res_buf)
1716                                         SVC_FREERES(xprt);
1717                                 error++;
1718                                 goto done;
1719                         }
1720                         if (dis_flags & RPC_AVOIDWORK) {
1721                                 SVC_DUPDONE_EXT(xprt, dr, res, NULL,
1722                                     disp->dis_ressz, DUP_DROP);
1723                         } else {
1724                                 SVC_DUPDONE_EXT(xprt, dr, res,
1725                                     disp->dis_resfree == nullfree ? NULL :
1726                                     disp->dis_resfree,
1727                                     disp->dis_ressz, DUP_DONE);
```

```
1728                                    dupcached = TRUE;
1729                            }
1730                            break;
1731                    case DUP_DONE:
1732                            break;
1733                    }

1735            } else {
1736                    curthread->t_flag |= T_DONTPEND;

1738                    (*disp->dis_proc)(args, res, exi, req, cr, ro);

1740                    curthread->t_flag &= ~T_DONTPEND;
1741                    if (curthread->t_flag & T_WOULDBLOCK) {
1742                            curthread->t_flag &= ~T_WOULDBLOCK;
1743                            if (res != (char *)&res_buf)
1744                                    SVC_FREERES(xprt);
1745                            error++;
1746                            goto done;
1747                    }
1748            }

1750            if (auth_tooweak(req, res)) {
1751                    svcerr_weakauth(xprt);
1752                    error++;
1753                    goto done;
1754            }

1756            /*
1757             * Check to see if logging has been enabled on the server.
1758             * If so, then obtain the export info struct to be used for
1759             * the later writing of the log record.  This is done for
1760             * the case that a lookup is done across a non-logged public
1761             * file system.
1762             */
1763            if (nfslog_buffer_list != NULL) {
1764                    nfslog_exi = nfslog_get_exi(exi, req, res, &nfslog_rec_id);
1765                    /*
1766                     * Is logging enabled?
1767                     */
1768                    logging_enabled = (nfslog_exi != NULL);

1770                    /*
1771                     * Copy the netbuf for logging purposes, before it is
1772                     * freed by svc_sendreply().
1773                     */
1774                    if (logging_enabled) {
1775                            NFSLOG_COPY_NETBUF(nfslog_exi, xprt, &nb);
1776                            /*
1777                             * If RPC_MAPRESP flag set (i.e. in V2 ops) the
1778                             * res gets copied directly into the mbuf and
1779                             * may be freed soon after the sendreply. So we
1780                             * must copy it here to a safe place...
1781                             */
1782                            if (res != (char *)&res_buf) {
1783                                    bcopy(res, (char *)&res_buf, disp->dis_ressz);
1784                            }
1785                    }
1786            }

1788            /*
1789             * Serialize and send results struct
1790             */
1791 #ifdef DEBUG
1792            if (rfs_no_fast_xdrres == 0 && res != (char *)&res_buf)
1793 #else
```



```
1794            if (res != (char *)&res_buf)
1795 #endif
1796            {
1797                    if (!svc_sendreply(xprt, disp->dis_fastxdrres, res)) {
1798                            cmn_err(CE_NOTE, "%s: bad sendreply", pgmname);
1799                            svcerr_systemerr(xprt);
1800                            error++;
1801                    }
1802            } else {
1803                    if (!svc_sendreply(xprt, disp->dis_xdrres, res)) {
1804                            cmn_err(CE_NOTE, "%s: bad sendreply", pgmname);
1805                            svcerr_systemerr(xprt);
1806                            error++;
1807                    }
1808            }

1810            /*
1811             * Log if needed
1812             */
1813            if (logging_enabled) {
1814                    nfslog_write_record(nfslog_exi, req, args, (char *)&res_buf,
1815                        cr, &nb, nfslog_rec_id, NFSLOG_ONE_BUFFER);
1816                    exi_rele(nfslog_exi);
1817                    kmem_free((&nb)->buf, (&nb)->len);
1818            }

1820            /*
1821             * Free results struct. With the addition of NFS V4 we can
1822             * have non-idempotent procedures with functions.
1823             */
1824            if (disp->dis_resfree != nullfree && dupcached == FALSE) {
1825                    (*disp->dis_resfree)(res);
1826            }

1828 done:
1829            /*
1830             * Free arguments struct
1831             */
1832            if (disp) {
1833                    if (!SVC_FREEARGS(xprt, disp->dis_xdrargs, args)) {
1834                            cmn_err(CE_NOTE, "%s: bad freeargs", pgmname);
1835                            error++;
1836                    }
1837            } else {
1838                    if (!SVC_FREEARGS(xprt, (xdrproc_t)0, (caddr_t)0)) {
1839                            cmn_err(CE_NOTE, "%s: bad freeargs", pgmname);
1840                            error++;
1841                    }
1842            }

1844            if (exi != NULL)
1845                    exi_rele(exi);

1847            global_svstat_ptr[req->rq_vers][NFS_BADCALLS].value.ui64 += error;

1849            global_svstat_ptr[req->rq_vers][NFS_CALLS].value.ui64++;
1850 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   67795 Wed Sep 14 16:21:03 2016
new/usr/src/uts/common/fs/nfs/nfs_srv.c
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_

2028 /*
2029  * rename a file
2030  * Give a file (from) a new name (to).
2031  */
2032 /* ARGSUSED */
2033 void
2034 rfs_rename(struct nfsrnmargs *args, enum nfsstat *status,
2035     struct exportinfo *exi, struct svc_req *req, cred_t *cr, bool_t ro)
2036 {
2037         int error = 0;
2038         vnode_t *fromvp;
2039         vnode_t *tovp;
2040         struct exportinfo *to_exi;
2041         fhandle_t *fh;
2042         vnode_t *srcvp;
2043         vnode_t *targvp;
2044         int in_crit = 0;

2046         fromvp = nfs_fhtovp(args->rna_from.da_fhandle, exi);
2047         if (fromvp == NULL) {
2048                 *status = NFSERR_STALE;
2049                 return;
2050         }

2052         fh = args->rna_to.da_fhandle;
2053         to_exi = checkexport(&fh->fh_fsid, (fid_t *)&fh->fh_xlen, NULL);
2053         to_exi = checkexport(&fh->fh_fsid, (fid_t *)&fh->fh_xlen);
2054         if (to_exi == NULL) {
2055                 VN_RELE(fromvp);
2056                 *status = NFSERR_ACCES;
2057                 return;
2058         }
2059         exi_rele(to_exi);

2061         if (to_exi != exi) {
2062                 VN_RELE(fromvp);
2063                 *status = NFSERR_XDEV;
2064                 return;
2065         }

2067         tovp = nfs_fhtovp(args->rna_to.da_fhandle, exi);
2068         if (tovp == NULL) {
2069                 VN_RELE(fromvp);
2070                 *status = NFSERR_STALE;
2071                 return;
2072         }

2074         if (fromvp->v_type != VDIR || tovp->v_type != VDIR) {
2075                 VN_RELE(tovp);
2076                 VN_RELE(fromvp);
2077                 *status = NFSERR_NOTDIR;
2078                 return;
2079         }

2081         /*
2082          * Disallow NULL paths
2083          */
2084         if (args->rna_from.da_name == NULL || *args->rna_from.da_name == '\0' ||
2085             args->rna_to.da_name == NULL || *args->rna_to.da_name == '\0') {
```



```
2086                 VN_RELE(tovp);
2087                 VN_RELE(fromvp);
2088                 *status = NFSERR_ACCES;
2089                 return;
2090         }

2092         if (rdonly(ro, tovp)) {
2093                 VN_RELE(tovp);
2094                 VN_RELE(fromvp);
2095                 *status = NFSERR_ROFS;
2096                 return;
2097         }

2099         /*
2100          * Check for a conflict with a non-blocking mandatory share reservation.
2101          */
2102         error = VOP_LOOKUP(fromvp, args->rna_from.da_name, &srcvp, NULL, 0,
2103             NULL, cr, NULL, NULL, NULL);
2104         if (error != 0) {
2105                 VN_RELE(tovp);
2106                 VN_RELE(fromvp);
2107                 *status = puterrno(error);
2108                 return;
2109         }

2111         /* Check for delegations on the source file */

2113         if (rfs4_check_delegated(FWRITE, srcvp, FALSE)) {
2114                 VN_RELE(tovp);
2115                 VN_RELE(fromvp);
2116                 VN_RELE(srcvp);
2117                 curthread->t_flag |= T_WOULDBLOCK;
2118                 return;
2119         }

2121         /* Check for delegation on the file being renamed over, if it exists */

2123         if (rfs4_deleg_policy != SRV_NEVER_DELEGATE &&
2124             VOP_LOOKUP(tovp, args->rna_to.da_name, &targvp, NULL, 0, NULL, cr,
2125             NULL, NULL, NULL) == 0) {

2127                 if (rfs4_check_delegated(FWRITE, targvp, TRUE)) {
2128                         VN_RELE(tovp);
2129                         VN_RELE(fromvp);
2130                         VN_RELE(srcvp);
2131                         VN_RELE(targvp);
2132                         curthread->t_flag |= T_WOULDBLOCK;
2133                         return;
2134                 }
2135                 VN_RELE(targvp);
2136         }


2139         if (nbl_need_check(srcvp)) {
2140                 nbl_start_crit(srcvp, RW_READER);
2141                 in_crit = 1;
2142                 if (nbl_conflict(srcvp, NBL_RENAME, 0, 0, 0, NULL)) {
2143                         error = EACCES;
2144                         goto out;
2145                 }
2146         }

2148         error = VOP_RENAME(fromvp, args->rna_from.da_name,
2149             tovp, args->rna_to.da_name, cr, NULL, 0);

2151         if (error == 0)
```

```
2152                 vn_renamepath(tovp, srcvp, args->rna_to.da_name,
2153                     strlen(args->rna_to.da_name));

2155         /*
2156          * Force modified data and metadata out to stable storage.
2157          */
2158         (void) VOP_FSYNC(tovp, 0, cr, NULL);
2159         (void) VOP_FSYNC(fromvp, 0, cr, NULL);

2161 out:
2162         if (in_crit)
2163                 nbl_end_crit(srcvp);
2164         VN_RELE(srcvp);
2165         VN_RELE(tovp);
2166         VN_RELE(fromvp);

2168         *status = puterrno(error);

2170 }
```

*unchanged_portion_omitted_*

```
2177 /*
2178  * Link to a file.
2179  * Create a file (to) which is a hard link to the given file (from).
2180  */
2181 /* ARGSUSED */
2182 void
2183 rfs_link(struct nfslinkargs *args, enum nfsstat *status,
2184     struct exportinfo *exi, struct svc_req *req, cred_t *cr, bool_t ro)
2185 {
2186         int error;
2187         vnode_t *fromvp;
2188         vnode_t *tovp;
2189         struct exportinfo *to_exi;
2190         fhandle_t *fh;

2192         fromvp = nfs_fhtovp(args->la_from, exi);
2193         if (fromvp == NULL) {
2194                 *status = NFSERR_STALE;
2195                 return;
2196         }

2198         fh = args->la_to.da_fhandle;
2199         to_exi = checkexport(&fh->fh_fsid, (fid_t *)&fh->fh_xlen, NULL);
2199         to_exi = checkexport(&fh->fh_fsid, (fid_t *)&fh->fh_xlen);
2200         if (to_exi == NULL) {
2201                 VN_RELE(fromvp);
2202                 *status = NFSERR_ACCES;
2203                 return;
2204         }
2205         exi_rele(to_exi);

2207         if (to_exi != exi) {
2208                 VN_RELE(fromvp);
2209                 *status = NFSERR_XDEV;
2210                 return;
2211         }

2213         tovp = nfs_fhtovp(args->la_to.da_fhandle, exi);
2214         if (tovp == NULL) {
2215                 VN_RELE(fromvp);
2216                 *status = NFSERR_STALE;
2217                 return;
2218         }

2220         if (tovp->v_type != VDIR) {
```



```
2221                 VN_RELE(tovp);
2222                 VN_RELE(fromvp);
2223                 *status = NFSERR_NOTDIR;
2224                 return;
2225         }
2226         /*
2227          * Disallow NULL paths
2228          */
2229         if (args->la_to.da_name == NULL || *args->la_to.da_name == '\0') {
2230                 VN_RELE(tovp);
2231                 VN_RELE(fromvp);
2232                 *status = NFSERR_ACCES;
2233                 return;
2234         }

2236         if (rdonly(ro, tovp)) {
2237                 VN_RELE(tovp);
2238                 VN_RELE(fromvp);
2239                 *status = NFSERR_ROFS;
2240                 return;
2241         }

2243         error = VOP_LINK(tovp, fromvp, args->la_to.da_name, cr, NULL, 0);

2245         /*
2246          * Force modified data and metadata out to stable storage.
2247          */
2248         (void) VOP_FSYNC(tovp, 0, cr, NULL);
2249         (void) VOP_FSYNC(fromvp, FNODSYNC, cr, NULL);

2251         VN_RELE(tovp);
2252         VN_RELE(fromvp);

2254         *status = puterrno(error);

2256 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   24073 Wed Sep 14 16:21:04 2016
new/usr/src/uts/common/nfs/export.h
7378 exported_lock held during nfs4 compound processing
**********************************************************
______unchanged_portion_omitted_
 572 typedef struct exp_visible exp_visible_t;

 574 #define PSEUDO(exi)     ((exi)->exi_export.ex_flags & EX_PSEUDO)
 575 #define EXP_LINKED(exi) ((exi)->fid_hash.bckt != NULL)

 577 #define EQFSID(fsidp1, fsidp2)  \
 578         (((fsidp1)->val[0] == (fsidp2)->val[0]) && \
 579             ((fsidp1)->val[1] == (fsidp2)->val[1]))

 581 #define EQFID(fidp1, fidp2)     \
 582         ((fidp1)->fid_len == (fidp2)->fid_len && \
 583             bcmp((char *)(fidp1)->fid_data, (char *)(fidp2)->fid_data, \
 584             (uint_t)(fidp1)->fid_len) == 0)

 586 #define exportmatch(exi, fsid, fid)     \
 587         (EQFSID(&(exi)->exi_fsid, (fsid)) && EQFID(&(exi)->exi_fid, (fid)))

 589 /*
 590  * Returns true iff exported filesystem is read-only to the given host.
 591  *
 592  * Note:  this macro should be as fast as possible since it's called
 593  * on each NFS modification request.
 594  */
 595 #define rdonly(ro, vp)  ((ro) || vn_is_readonly(vp))
 596 #define rdonly4(req, cs)  \
 597         (vn_is_readonly((cs)->vp) || \
 598             (nfsauth4_access((cs)->exi, (cs)->vp, (req), (cs)->basecr, NULL, \
 599             NULL, NULL, NULL) & (NFSAUTH_RO | NFSAUTH_LIMITED)))

 601 extern int      nfsauth4_access(struct exportinfo *, vnode_t *,
 602     struct svc_req *, cred_t *, uid_t *, gid_t *, uint_t *, gid_t **);
 603 extern int      nfsauth4_secinfo_access(struct exportinfo *,
 604     struct svc_req *, int, int, cred_t *);
 605 extern int      nfsauth_cache_clnt_compar(const void *, const void *);
 606 extern int      nfs_fhbcmp(char *, char *, int);
 607 extern int      nfs_exportinit(void);
 608 extern void     nfs_exportfini(void);
 609 extern int      chk_clnt_sec(struct exportinfo *, struct svc_req *);
 610 extern int      makefh(fhandle_t *, struct vnode *, struct exportinfo *);
 611 extern int      makefh_ol(fhandle_t *, struct exportinfo *, uint_t);
 612 extern int      makefh3(nfs_fh3 *, struct vnode *, struct exportinfo *);
 613 extern int      makefh3_ol(nfs_fh3 *, struct exportinfo *, uint_t);
 614 extern vnode_t *nfs_fhtovp(fhandle_t *, struct exportinfo *);
 615 extern vnode_t *nfs3_fhtovp(nfs_fh3 *, struct exportinfo *);
 616 extern struct   exportinfo *checkexport(fsid_t *, struct fid *, vnode_t *);
 617 extern struct   exportinfo *checkexport_nohold(fsid_t *, struct fid *,
     vnode_t *);
 616 extern struct   exportinfo *checkexport(fsid_t *, struct fid *);
 617 extern struct   exportinfo *checkexport4(fsid_t *, struct fid *, vnode_t *);
 619 extern void     exi_hold(struct exportinfo *);
 620 extern void     exi_rele(struct exportinfo *);
 621 extern struct exportinfo *nfs_vptoexi(vnode_t *, vnode_t *, cred_t *, int *,
     int *, bool_t);
 623 extern int      nfs_check_vpexi(vnode_t *, vnode_t *, cred_t *,
                         struct exportinfo **);
 625 extern void     export_link(struct exportinfo *);
 626 extern void     export_unlink(struct exportinfo *);
 627 extern vnode_t *untraverse(vnode_t *);
 628 extern int      vn_is_nfs_reparse(vnode_t *, cred_t *);
 629 extern int      client_is_downrev(struct svc_req *);
```



```
 630 extern char    *build_symlink(vnode_t *, cred_t *, size_t *);

 632 /*
 633  * Functions that handle the NFSv4 server namespace
 634  */
 635 extern exportinfo_t *vis2exi(treenode_t *);
 636 extern int      treeclimb_export(struct exportinfo *);
 637 extern void     treeclimb_unexport(struct exportinfo *);
 638 extern int      nfs_visible(struct exportinfo *, vnode_t *, int *);
 639 extern int      nfs_visible_inode(struct exportinfo *, ino64_t, int *);
 640 extern int      has_visible(struct exportinfo *, vnode_t *);
 641 extern void     free_visible(struct exp_visible *);
 642 extern int      nfs_exported(struct exportinfo *, vnode_t *);
 643 extern struct exportinfo *pseudo_exportfs(vnode_t *, fid_t *,
 644     struct exp_visible *, struct exportdata *);
 645 extern int      vop_fid_pseudo(vnode_t *, fid_t *);
 646 extern int      nfs4_vget_pseudo(struct exportinfo *, vnode_t **, fid_t *);
 647 /*
 648  * Functions that handle the NFSv4 server namespace security flavors
 649  * information.
 650  */
 651 extern void     srv_secinfo_exp2pseu(struct exportdata *, struct exportdata *);
 652 extern void     srv_secinfo_list_free(struct secinfo *, int);

 654 /*
 655  * "public" and default (root) location for public filehandle
 656  */
 657 extern struct exportinfo *exi_public, *exi_root;
 658 extern fhandle_t nullfh2;       /* for comparing V2 filehandles */
 659 extern krwlock_t exported_lock;
 660 extern struct exportinfo *exptable[];

 662 /*
 663  * Two macros for identifying public filehandles.
 664  * A v2 public filehandle is 32 zero bytes.
 665  * A v3 public filehandle is zero length.
 666  */
 667 #define PUBLIC_FH2(fh) \
 668         ((fh)->fh_fsid.val[1] == 0 && \
 669         bcmp((fh), &nullfh2, sizeof (fhandle_t)) == 0)

 671 #define PUBLIC_FH3(fh) \
 672         ((fh)->fh3_length == 0)

 674 extern int      makefh4(nfs_fh4 *, struct vnode *, struct exportinfo *);
 675 extern vnode_t *nfs4_fhtovp(nfs_fh4 *, struct exportinfo *, nfsstat4 *);

 677 #endif /* _KERNEL */

 679 #ifdef  __cplusplus
 680 }
______unchanged_portion_omitted_
```