****
430161 Thu Sep 22 17:42:43 2016
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
6785 nfs4_attr_cache deadlock
****
_____***unchanged_portion_omitted_***

```
2402 /*
2403  * Assumes you already have the open seqid sync grabbed as well as the
2404  * 'os_sync_lock'.  Note: this will release the open seqid sync and
2405  * 'os_sync_lock' if client recovery starts.  Calling functions have to
2406  * be prepared to handle this.
2407  *
2408  * 'recov' is returned as 1 if the CLOSE operation detected client recovery
2409  * was needed and was started, and that the calling function should retry
2410  * this function; otherwise it is returned as 0.
2411  *
2412  * Errors are returned via the nfs4_error_t parameter.
2413  */
2414 static void
2415 nfs4close_otw(rnode4_t *rp, cred_t *cred_otw, nfs4_open_owner_t *oop,
2416     nfs4_open_stream_t *osp, int *recov, int *did_start_seqid_syncp,
2417     nfs4_close_type_t close_type, nfs4_error_t *ep, int *have_sync_lockp)
2418 {
2419         COMPOUND4args_clnt args;
2420         COMPOUND4res_clnt res;
2421         CLOSE4args *close_args;
2422         nfs_resop4 *resop;
2423         nfs_argop4 argop[3];
2424         int doqueue = 1;
2425         mntinfo4_t *mi;
2426         seqid4 seqid;
2427         vnode_t *vp;
2428         bool_t needrecov = FALSE;
2429         nfs4_lost_rqst_t lost_rqst;
2430         hrtime_t t;

2432         ASSERT(nfs_zone() == VTOMI4(RTOV4(rp))->mi_zone);

2434         ASSERT(MUTEX_HELD(&osp->os_sync_lock));

2436         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4close_otw"));

2438         /* Only set this to 1 if recovery is started */
2439         *recov = 0;

2441         /* do the OTW call to close the file */

2443         if (close_type == CLOSE_RESEND)
2444                 args.ctag = TAG_CLOSE_LOST;
2445         else if (close_type == CLOSE_AFTER_RESEND)
2446                 args.ctag = TAG_CLOSE_UNDO;
2447         else
2448                 args.ctag = TAG_CLOSE;

2450         args.array_len = 3;
2451         args.array = argop;

2453         vp = RTOV4(rp);

2455         mi = VTOMI4(vp);

2457         /* putfh target fh */
2458         argop[0].argop = OP_CPUTFH;
2459         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;
```



```
2461         argop[1].argop = OP_GETATTR;
2462         argop[1].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
2463         argop[1].nfs_argop4_u.opgetattr.mi = mi;

2465         argop[2].argop = OP_CLOSE;
2466         close_args = &argop[2].nfs_argop4_u.opclose;

2468         seqid = nfs4_get_open_seqid(oop) + 1;

2470         close_args->seqid = seqid;
2471         close_args->open_stateid = osp->open_stateid;

2473         NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
2474             "nfs4close_otw: %s call, rp %s", needrecov ? "recov" : "first",
2475             rnode4info(rp)));

2477         t = gethrtime();

2479         rfs4call(mi, &args, &res, cred_otw, &doqueue, 0, ep);

2481         if (!ep->error && nfs4_need_to_bump_seqid(&res)) {
2482                 nfs4_set_open_seqid(seqid, oop, args.ctag);
2483         }

2485         needrecov = nfs4_needs_recovery(ep, TRUE, mi->mi_vfsp);
2486         if (ep->error && !needrecov) {
2487                 /*
2488                  * if there was an error and no recovery is to be done
2489                  * then then set up the file to flush its cache if
2490                  * needed for the next caller.
2491                  */
2492                 mutex_enter(&rp->r_statelock);
2493                 PURGE_ATTRCACHE4_LOCKED(rp);
2494                 rp->r_flags &= ~R4WRITEMODIFIED;
2495                 mutex_exit(&rp->r_statelock);
2496                 return;
2497         }

2499         if (needrecov) {
2500                 bool_t abort;
2501                 nfs4_bseqid_entry_t *bsep = NULL;

2503                 if (close_type != CLOSE_RESEND)
2504                         nfs4close_save_lost_rqst(ep->error, &lost_rqst, oop,
2505                             osp, cred_otw, vp);

2507                 if (!ep->error && res.status == NFS4ERR_BAD_SEQID)
2508                         bsep = nfs4_create_bseqid_entry(oop, NULL, vp,
2509                             0, args.ctag, close_args->seqid);

2511                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
2512                     "nfs4close_otw: initiating recovery. error %d "
2513                     "res.status %d", ep->error, res.status));

2515                 /*
2516                  * Drop the 'os_sync_lock' here so we don't hit
2517                  * a potential recursive mutex_enter via an
2518                  * 'open_stream_hold()'.
2519                  */
2520                 mutex_exit(&osp->os_sync_lock);
2521                 *have_sync_lockp = 0;
2522                 abort = nfs4_start_recovery(ep, VTOMI4(vp), vp, NULL, NULL,
2523                     (close_type != CLOSE_RESEND &&
2524                     lost_rqst.lr_op == OP_CLOSE) ? &lost_rqst : NULL,
2525                     OP_CLOSE, bsep, NULL, NULL);
```

```
2527                /* drop open seq sync, and let the calling function regrab it */
2528                nfs4_end_open_seqid_sync(oop);
2529                *did_start_seqid_syncp = 0;
2530
2531                if (bsep)
2532                        kmem_free(bsep, sizeof (*bsep));
2533                /*
2534                 * For signals, the caller wants to quit, so don't say to
2535                 * retry.  For forced unmount, if it's a user thread, it
2536                 * wants to quit.  If it's a recovery thread, the retry
2537                 * will happen higher-up on the call stack.  Either way,
2538                 * don't say to retry.
2539                 */
2540                if (abort == FALSE && ep->error != EINTR &&
2541                    !NFS4_FRC_UNMT_ERR(ep->error, mi->mi_vfsp) &&
2542                    close_type != CLOSE_RESEND &&
2543                    close_type != CLOSE_AFTER_RESEND)
2544                        *recov = 1;
2545                else
2546                        *recov = 0;
2547
2548                if (!ep->error)
2549                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2550                return;
2551        }
2552
2553        if (res.status) {
2554                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2555                return;
2556        }
2557
2558        mutex_enter(&rp->r_statev4_lock);
2559        rp->created_v4 = 0;
2560        mutex_exit(&rp->r_statev4_lock);
2561
2562        resop = &res.array[2];
2563        osp->open_stateid = resop->nfs_resop4_u.opclose.open_stateid;
2564        osp->os_valid = 0;
2565
2566        /*
2567         * This removes the reference obtained at OPEN; ie, when the
2568         * open stream structure was created.
2569         *
2570         * We don't have to worry about calling 'open_stream_rele'
2571         * since we our currently holding a reference to the open
2572         * stream which means the count cannot go to 0 with this
2573         * decrement.
2574         */
2575        ASSERT(osp->os_ref_count >= 2);
2576        osp->os_ref_count--;
2577
2578        if (ep->error == 0) {
2579                /*
2580                 * Avoid a deadlock with the r_serial thread waiting for
2581                 * os_sync_lock in nfs4_get_otw_cred_by_osp() which might be
2582                 * held by us. We will wait in nfs4_attr_cache() for the
2583                 * completion of the r_serial thread.
2584                 */
2585                mutex_exit(&osp->os_sync_lock);
2586                *have_sync_lockp = 0;
2587
2588                nfs4_attr_cache(vp,
2589                    &res.array[1].nfs_resop4_u.opgetattr.ga_res,
2590                    t, cred_otw, TRUE, NULL);
2591        }
```



```
2592 #endif /* ! codereview */
2593
2594         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4close_otw:"
2595             " returning %d", ep->error));
2596
2597         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2598 }
2599
2600 /* ARGSUSED */
2601 static int
2602 nfs4_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
2603     caller_context_t *ct)
2604 {
2605         rnode4_t *rp;
2606         u_offset_t off;
2607         offset_t diff;
2608         uint_t on;
2609         uint_t n;
2610         caddr_t base;
2611         uint_t flags;
2612         int error;
2613         mntinfo4_t *mi;
2614
2615         rp = VTOR4(vp);
2616
2617         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));
2618
2619         if (IS_SHADOW(vp, rp))
2620                 vp = RTOV4(rp);
2621
2622         if (vp->v_type != VREG)
2623                 return (EISDIR);
2624
2625         mi = VTOMI4(vp);
2626
2627         if (nfs_zone() != mi->mi_zone)
2628                 return (EIO);
2629
2630         if (uiop->uio_resid == 0)
2631                 return (0);
2632
2633         if (uiop->uio_loffset < 0 || uiop->uio_loffset + uiop->uio_resid < 0)
2634                 return (EINVAL);
2635
2636         mutex_enter(&rp->r_statelock);
2637         if (rp->r_flags & R4RECOVERRP)
2638                 error = (rp->r_error ? rp->r_error : EIO);
2639         else
2640                 error = 0;
2641         mutex_exit(&rp->r_statelock);
2642         if (error)
2643                 return (error);
2644
2645         /*
2646          * Bypass VM if caching has been disabled (e.g., locking) or if
2647          * using client-side direct I/O and the file is not mmap'd and
2648          * there are no cached pages.
2649          */
2650         if ((vp->v_flag & VNOCACHE) ||
2651             (((rp->r_flags & R4DIRECTIO) || (mi->mi_flags & MI4_DIRECTIO)) &&
2652             rp->r_mapcnt == 0 && rp->r_inmap == 0 && !nfs4_has_pages(vp))) {
2653                 size_t resid = 0;
2654
2655                 return (nfs4read(vp, NULL, uiop->uio_loffset,
2656                     uiop->uio_resid, &resid, cr, FALSE, uiop));
2657         }
```

```
2659         error = 0;

2661         do {
2662                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
2663                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
2664                 n = MIN(MAXBSIZE - on, uiop->uio_resid);

2666                 if (error = nfs4_validate_caches(vp, cr))
2667                         break;

2669                 mutex_enter(&rp->r_statelock);
2670                 while (rp->r_flags & R4INCACHEPURGE) {
2671                         if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
2672                                 mutex_exit(&rp->r_statelock);
2673                                 return (EINTR);
2674                         }
2675                 }
2676                 diff = rp->r_size - uiop->uio_loffset;
2677                 mutex_exit(&rp->r_statelock);
2678                 if (diff <= 0)
2679                         break;
2680                 if (diff < n)
2681                         n = (uint_t)diff;

2683                 if (vpm_enable) {
2684                         /*
2685                          * Copy data.
2686                          */
2687                         error = vpm_data_copy(vp, off + on, n, uiop,
2688                             1, NULL, 0, S_READ);
2689                 } else {
2690                         base = segmap_getmapflt(segkmap, vp, off + on, n, 1,
2691                             S_READ);

2693                         error = uiomove(base + on, n, UIO_READ, uiop);
2694                 }

2696                 if (!error) {
2697                         /*
2698                          * If read a whole block or read to eof,
2699                          * won't need this buffer again soon.
2700                          */
2701                         mutex_enter(&rp->r_statelock);
2702                         if (n + on == MAXBSIZE ||
2703                             uiop->uio_loffset == rp->r_size)
2704                                 flags = SM_DONTNEED;
2705                         else
2706                                 flags = 0;
2707                         mutex_exit(&rp->r_statelock);
2708                         if (vpm_enable) {
2709                                 error = vpm_sync_pages(vp, off, n, flags);
2710                         } else {
2711                                 error = segmap_release(segkmap, base, flags);
2712                         }
2713                 } else {
2714                         if (vpm_enable) {
2715                                 (void) vpm_sync_pages(vp, off, n, 0);
2716                         } else {
2717                                 (void) segmap_release(segkmap, base, 0);
2718                         }
2719                 }
2720         } while (!error && uiop->uio_resid > 0);

2722         return (error);
2723 }
```



```
2725 /* ARGSUSED */
2726 static int
2727 nfs4_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
2728     caller_context_t *ct)
2729 {
2730         rlim64_t limit = uiop->uio_llimit;
2731         rnode4_t *rp;
2732         u_offset_t off;
2733         caddr_t base;
2734         uint_t flags;
2735         int remainder;
2736         size_t n;
2737         int on;
2738         int error;
2739         int resid;
2740         u_offset_t offset;
2741         mntinfo4_t *mi;
2742         uint_t bsize;

2744         rp = VTOR4(vp);

2746         if (IS_SHADOW(vp, rp))
2747                 vp = RTOV4(rp);

2749         if (vp->v_type != VREG)
2750                 return (EISDIR);

2752         mi = VTOMI4(vp);

2754         if (nfs_zone() != mi->mi_zone)
2755                 return (EIO);

2757         if (uiop->uio_resid == 0)
2758                 return (0);

2760         mutex_enter(&rp->r_statelock);
2761         if (rp->r_flags & R4RECOVERRP)
2762                 error = (rp->r_error ? rp->r_error : EIO);
2763         else
2764                 error = 0;
2765         mutex_exit(&rp->r_statelock);
2766         if (error)
2767                 return (error);

2769         if (ioflag & FAPPEND) {
2770                 struct vattr va;

2772                 /*
2773                  * Must serialize if appending.
2774                  */
2775                 if (nfs_rw_lock_held(&rp->r_rwlock, RW_READER)) {
2776                         nfs_rw_exit(&rp->r_rwlock);
2777                         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER,
2778                             INTR4(vp)))
2779                                 return (EINTR);
2780                 }

2782                 va.va_mask = AT_SIZE;
2783                 error = nfs4getattr(vp, &va, cr);
2784                 if (error)
2785                         return (error);
2786                 uiop->uio_loffset = va.va_size;
2787         }

2789         offset = uiop->uio_loffset + uiop->uio_resid;
```

```
2791         if (uiop->uio_loffset < (offset_t)0 || offset < 0)
2792                 return (EINVAL);

2794         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
2795                 limit = MAXOFFSET_T;

2797         /*
2798          * Check to make sure that the process will not exceed
2799          * its limit on file size.  It is okay to write up to
2800          * the limit, but not beyond.  Thus, the write which
2801          * reaches the limit will be short and the next write
2802          * will return an error.
2803          */
2804         remainder = 0;
2805         if (offset > uiop->uio_llimit) {
2806                 remainder = offset - uiop->uio_llimit;
2807                 uiop->uio_resid = uiop->uio_llimit - uiop->uio_loffset;
2808                 if (uiop->uio_resid <= 0) {
2809                         proc_t *p = ttoproc(curthread);

2811                         uiop->uio_resid += remainder;
2812                         mutex_enter(&p->p_lock);
2813                         (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE],
2814                             p->p_rctls, p, RCA_UNSAFE_SIGINFO);
2815                         mutex_exit(&p->p_lock);
2816                         return (EFBIG);
2817                 }
2818         }

2820         /* update the change attribute, if we have a write delegation */

2822         mutex_enter(&rp->r_statev4_lock);
2823         if (rp->r_deleg_type == OPEN_DELEGATE_WRITE)
2824                 rp->r_deleg_change++;

2826         mutex_exit(&rp->r_statev4_lock);

2828         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR4(vp)))
2829                 return (EINTR);

2831         /*
2832          * Bypass VM if caching has been disabled (e.g., locking) or if
2833          * using client-side direct I/O and the file is not mmap'd and
2834          * there are no cached pages.
2835          */
2836         if ((vp->v_flag & VNOCACHE) ||
2837             (((rp->r_flags & R4DIRECTIO) || (mi->mi_flags & MI4_DIRECTIO)) &&
2838             rp->r_mapcnt == 0 && rp->r_inmap == 0 && !nfs4_has_pages(vp))) {
2839                 size_t bufsize;
2840                 int count;
2841                 u_offset_t org_offset;
2842                 stable_how4 stab_comm;
2843 nfs4_fwrite:
2844                 if (rp->r_flags & R4STALE) {
2845                         resid = uiop->uio_resid;
2846                         offset = uiop->uio_loffset;
2847                         error = rp->r_error;
2848                         /*
2849                          * A close may have cleared r_error, if so,
2850                          * propagate ESTALE error return properly
2851                          */
2852                         if (error == 0)
2853                                 error = ESTALE;
2854                         goto bottom;
2855                 }
```



```
2857                 bufsize = MIN(uiop->uio_resid, mi->mi_stsize);
2858                 base = kmem_alloc(bufsize, KM_SLEEP);
2859                 do {
2860                         if (ioflag & FDSYNC)
2861                                 stab_comm = DATA_SYNC4;
2862                         else
2863                                 stab_comm = FILE_SYNC4;
2864                         resid = uiop->uio_resid;
2865                         offset = uiop->uio_loffset;
2866                         count = MIN(uiop->uio_resid, bufsize);
2867                         org_offset = uiop->uio_loffset;
2868                         error = uiomove(base, count, UIO_WRITE, uiop);
2869                         if (!error) {
2870                                 error = nfs4write(vp, base, org_offset,
2871                                     count, cr, &stab_comm);
2872                                 if (!error) {
2873                                         mutex_enter(&rp->r_statelock);
2874                                         if (rp->r_size < uiop->uio_loffset)
2875                                                 rp->r_size = uiop->uio_loffset;
2876                                         mutex_exit(&rp->r_statelock);
2877                                 }
2878                         }
2879                 } while (!error && uiop->uio_resid > 0);
2880                 kmem_free(base, bufsize);
2881                 goto bottom;
2882         }

2884         bsize = vp->v_vfsp->vfs_bsize;

2886         do {
2887                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
2888                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
2889                 n = MIN(MAXBSIZE - on, uiop->uio_resid);

2891                 resid = uiop->uio_resid;
2892                 offset = uiop->uio_loffset;

2894                 if (rp->r_flags & R4STALE) {
2895                         error = rp->r_error;
2896                         /*
2897                          * A close may have cleared r_error, if so,
2898                          * propagate ESTALE error return properly
2899                          */
2900                         if (error == 0)
2901                                 error = ESTALE;
2902                         break;
2903                 }

2905                 /*
2906                  * Don't create dirty pages faster than they
2907                  * can be cleaned so that the system doesn't
2908                  * get imbalanced.  If the async queue is
2909                  * maxed out, then wait for it to drain before
2910                  * creating more dirty pages.  Also, wait for
2911                  * any threads doing pagewalks in the vop_getattr
2912                  * entry points so that they don't block for
2913                  * long periods.
2914                  */
2915                 mutex_enter(&rp->r_statelock);
2916                 while ((mi->mi_max_threads != 0 &&
2917                     rp->r_awcount > 2 * mi->mi_max_threads) ||
2918                     rp->r_gcount > 0) {
2919                         if (INTR4(vp)) {
2920                                 klwp_t *lwp = ttolwp(curthread);
```

```
2922                                if (lwp != NULL)
2923                                        lwp->lwp_nostop++;
2924                                if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
2925                                        mutex_exit(&rp->r_statelock);
2926                                        if (lwp != NULL)
2927                                                lwp->lwp_nostop--;
2928                                        error = EINTR;
2929                                        goto bottom;
2930                                }
2931                                if (lwp != NULL)
2932                                        lwp->lwp_nostop--;
2933                        } else
2934                                cv_wait(&rp->r_cv, &rp->r_statelock);
2935                }
2936                mutex_exit(&rp->r_statelock);

2938                /*
2939                 * Touch the page and fault it in if it is not in core
2940                 * before segmap_getmapflt or vpm_data_copy can lock it.
2941                 * This is to avoid the deadlock if the buffer is mapped
2942                 * to the same file through mmap which we want to write.
2943                 */
2944                uio_prefaultpages((long)n, uiop);

2946                if (vpm_enable) {
2947                        /*
2948                         * It will use kpm mappings, so no need to
2949                         * pass an address.
2950                         */
2951                        error = writerp4(rp, NULL, n, uiop, 0);
2952                } else  {
2953                        if (segmap_kpm) {
2954                                int pon = uiop->uio_loffset & PAGEOFFSET;
2955                                size_t pn = MIN(PAGESIZE - pon,
2956                                    uiop->uio_resid);
2957                                int pagecreate;

2959                                mutex_enter(&rp->r_statelock);
2960                                pagecreate = (pon == 0) && (pn == PAGESIZE ||
2961                                    uiop->uio_loffset + pn >= rp->r_size);
2962                                mutex_exit(&rp->r_statelock);

2964                                base = segmap_getmapflt(segkmap, vp, off + on,
2965                                    pn, !pagecreate, S_WRITE);

2967                                error = writerp4(rp, base + pon, n, uiop,
2968                                    pagecreate);

2970                        } else {
2971                                base = segmap_getmapflt(segkmap, vp, off + on,
2972                                    n, 0, S_READ);
2973                                error = writerp4(rp, base + on, n, uiop, 0);
2974                        }
2975                }

2977                if (!error) {
2978                        if (mi->mi_flags & MI4_NOAC)
2979                                flags = SM_WRITE;
2980                        else if ((uiop->uio_loffset % bsize) == 0 ||
2981                            IS_SWAPVP(vp)) {
2982                                /*
2983                                 * Have written a whole block.
2984                                 * Start an asynchronous write
2985                                 * and mark the buffer to
2986                                 * indicate that it won't be
2987                                 * needed again soon.
```



```
2988                                 */
2989                                flags = SM_WRITE | SM_ASYNC | SM_DONTNEED;
2990                        } else
2991                                flags = 0;
2992                        if ((ioflag & (FSYNC|FDSYNC)) ||
2993                            (rp->r_flags & R4OUTOFSPACE)) {
2994                                flags &= ~SM_ASYNC;
2995                                flags |= SM_WRITE;
2996                        }
2997                        if (vpm_enable) {
2998                                error = vpm_sync_pages(vp, off, n, flags);
2999                        } else {
3000                                error = segmap_release(segkmap, base, flags);
3001                        }
3002                } else {
3003                        if (vpm_enable) {
3004                                (void) vpm_sync_pages(vp, off, n, 0);
3005                        } else {
3006                                (void) segmap_release(segkmap, base, 0);
3007                        }
3008                        /*
3009                         * In the event that we got an access error while
3010                         * faulting in a page for a write-only file just
3011                         * force a write.
3012                         */
3013                        if (error == EACCES)
3014                                goto nfs4_fwrite;
3015                }
3016        } while (!error && uiop->uio_resid > 0);

3018 bottom:
3019        if (error) {
3020                uiop->uio_resid = resid + remainder;
3021                uiop->uio_loffset = offset;
3022        } else {
3023                uiop->uio_resid += remainder;

3025                mutex_enter(&rp->r_statev4_lock);
3026                if (rp->r_deleg_type == OPEN_DELEGATE_WRITE) {
3027                        gethrestime(&rp->r_attr.va_mtime);
3028                        rp->r_attr.va_ctime = rp->r_attr.va_mtime;
3029                }
3030                mutex_exit(&rp->r_statev4_lock);
3031        }

3033        nfs_rw_exit(&rp->r_lkserlock);

3035        return (error);
3036 }

3038 /*
3039  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED}
3040  */
3041 static int
3042 nfs4_rdwrlbn(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
3043     int flags, cred_t *cr)
3044 {
3045        struct buf *bp;
3046        int error;
3047        page_t *savepp;
3048        uchar_t fsdata;
3049        stable_how4 stab_comm;

3051        ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
3052        bp = pageio_setup(pp, len, vp, flags);
3053        ASSERT(bp != NULL);
```

```
3055         /*
3056          * pageio_setup should have set b_addr to 0.  This
3057          * is correct since we want to do I/O on a page
3058          * boundary.  bp_mapin will use this addr to calculate
3059          * an offset, and then set b_addr to the kernel virtual
3060          * address it allocated for us.
3061          */
3062         ASSERT(bp->b_un.b_addr == 0);

3064         bp->b_edev = 0;
3065         bp->b_dev = 0;
3066         bp->b_lblkno = lbtodb(off);
3067         bp->b_file = vp;
3068         bp->b_offset = (offset_t)off;
3069         bp_mapin(bp);

3071         if ((flags & (B_WRITE|B_ASYNC)) == (B_WRITE|B_ASYNC) &&
3072             freemem > desfree)
3073                 stab_comm = UNSTABLE4;
3074         else
3075                 stab_comm = FILE_SYNC4;

3077         error = nfs4_bio(bp, &stab_comm, cr, FALSE);

3079         bp_mapout(bp);
3080         pageio_done(bp);

3082         if (stab_comm == UNSTABLE4)
3083                 fsdata = C_DELAYCOMMIT;
3084         else
3085                 fsdata = C_NOCOMMIT;

3087         savepp = pp;
3088         do {
3089                 pp->p_fsdata = fsdata;
3090         } while ((pp = pp->p_next) != savepp);

3092         return (error);
3093 }

3095 /*
3096  */
3097 static int
3098 nfs4rdwr_check_osid(vnode_t *vp, nfs4_error_t *ep, cred_t *cr)
3099 {
3100         nfs4_open_owner_t       *oop;
3101         nfs4_open_stream_t      *osp;
3102         rnode4_t                *rp = VTOR4(vp);
3103         mntinfo4_t              *mi = VTOMI4(vp);
3104         int                     reopen_needed;

3106         ASSERT(nfs_zone() == mi->mi_zone);


3109         oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
3110         if (!oop)
3111                 return (EIO);

3113         /* returns with 'os_sync_lock' held */
3114         osp = find_open_stream(oop, rp);
3115         if (!osp) {
3116                 open_owner_rele(oop);
3117                 return (EIO);
3118         }
```



```
3120         if (osp->os_failed_reopen) {
3121                 mutex_exit(&osp->os_sync_lock);
3122                 open_stream_rele(osp, rp);
3123                 open_owner_rele(oop);
3124                 return (EIO);
3125         }

3127         /*
3128          * Determine whether a reopen is needed.  If this
3129          * is a delegation open stream, then the os_delegation bit
3130          * should be set.
3131          */

3133         reopen_needed = osp->os_delegation;

3135         mutex_exit(&osp->os_sync_lock);
3136         open_owner_rele(oop);

3138         if (reopen_needed) {
3139                 nfs4_error_zinit(ep);
3140                 nfs4_reopen(vp, osp, ep, CLAIM_NULL, FALSE, FALSE);
3141                 mutex_enter(&osp->os_sync_lock);
3142                 if (ep->error || ep->stat || osp->os_failed_reopen) {
3143                         mutex_exit(&osp->os_sync_lock);
3144                         open_stream_rele(osp, rp);
3145                         return (EIO);
3146                 }
3147                 mutex_exit(&osp->os_sync_lock);
3148         }
3149         open_stream_rele(osp, rp);

3151         return (0);
3152 }

3154 /*
3155  * Write to file.  Writes to remote server in largest size
3156  * chunks that the server can handle.  Write is synchronous.
3157  */
3158 static int
3159 nfs4write(vnode_t *vp, caddr_t base, u_offset_t offset, int count, cred_t *cr,
3160     stable_how4 *stab_comm)
3161 {
3162         mntinfo4_t *mi;
3163         COMPOUND4args_clnt args;
3164         COMPOUND4res_clnt res;
3165         WRITE4args *wargs;
3166         WRITE4res *wres;
3167         nfs_argop4 argop[2];
3168         nfs_resop4 *resop;
3169         int tsize;
3170         stable_how4 stable;
3171         rnode4_t *rp;
3172         int doqueue = 1;
3173         bool_t needrecov;
3174         nfs4_recov_state_t recov_state;
3175         nfs4_stateid_types_t sid_types;
3176         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
3177         int recov;

3179         rp = VTOR4(vp);
3180         mi = VTOMI4(vp);

3182         ASSERT(nfs_zone() == mi->mi_zone);

3184         stable = *stab_comm;
3185         *stab_comm = FILE_SYNC4;
```

```
3187         needrecov = FALSE;
3188         recov_state.rs_flags = 0;
3189         recov_state.rs_num_retry_despite_err = 0;
3190         nfs4_init_stateid_types(&sid_types);

3192         /* Is curthread the recovery thread? */
3193         mutex_enter(&mi->mi_lock);
3194         recov = (mi->mi_recovthread == curthread);
3195         mutex_exit(&mi->mi_lock);

3197 recov_retry:
3198         args.ctag = TAG_WRITE;
3199         args.array_len = 2;
3200         args.array = argop;

3202         if (!recov) {
3203                 e.error = nfs4_start_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3204                     &recov_state, NULL);
3205                 if (e.error)
3206                         return (e.error);
3207         }

3209         /* 0. putfh target fh */
3210         argop[0].argop = OP_CPUTFH;
3211         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

3213         /* 1. write */
3214         nfs4args_write(&argop[1], stable, rp, cr, &wargs, &sid_types);

3216         do {

3218                 wargs->offset = (offset4)offset;
3219                 wargs->data_val = base;

3221                 if (mi->mi_io_kstats) {
3222                         mutex_enter(&mi->mi_lock);
3223                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
3224                         mutex_exit(&mi->mi_lock);
3225                 }

3227                 if ((vp->v_flag & VNOCACHE) ||
3228                     (rp->r_flags & R4DIRECTIO) ||
3229                     (mi->mi_flags & MI4_DIRECTIO))
3230                         tsize = MIN(mi->mi_stsize, count);
3231                 else
3232                         tsize = MIN(mi->mi_curwrite, count);
3233                 wargs->data_len = (uint_t)tsize;
3234                 rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

3236                 if (mi->mi_io_kstats) {
3237                         mutex_enter(&mi->mi_lock);
3238                         kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
3239                         mutex_exit(&mi->mi_lock);
3240                 }

3242                 if (!recov) {
3243                         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
3244                         if (e.error && !needrecov) {
3245                                 nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3246                                     &recov_state, needrecov);
3247                                 return (e.error);
3248                         }
3249                 } else {
3250                         if (e.error)
3251                                 return (e.error);
```



```
3252                 }

3254                 /*
3255                  * Do handling of OLD_STATEID outside
3256                  * of the normal recovery framework.
3257                  *
3258                  * If write receives a BAD stateid error while using a
3259                  * delegation stateid, retry using the open stateid (if it
3260                  * exists).  If it doesn't have an open stateid, reopen the
3261                  * file first, then retry.
3262                  */
3263                 if (!e.error && res.status == NFS4ERR_OLD_STATEID &&
3264                     sid_types.cur_sid_type != SPEC_SID) {
3265                         nfs4_save_stateid(&wargs->stateid, &sid_types);
3266                         if (!recov)
3267                                 nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3268                                     &recov_state, needrecov);
3269                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3270                         goto recov_retry;
3271                 } else if (e.error == 0 && res.status == NFS4ERR_BAD_STATEID &&
3272                     sid_types.cur_sid_type == DEL_SID) {
3273                         nfs4_save_stateid(&wargs->stateid, &sid_types);
3274                         mutex_enter(&rp->r_statev4_lock);
3275                         rp->r_deleg_return_pending = TRUE;
3276                         mutex_exit(&rp->r_statev4_lock);
3277                         if (nfs4rdwr_check_osid(vp, &e, cr)) {
3278                                 if (!recov)
3279                                         nfs4_end_fop(mi, vp, NULL, OH_WRITE,
3280                                             &recov_state, needrecov);
3281                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3282                                     (caddr_t)&res);
3283                                 return (EIO);
3284                         }
3285                         if (!recov)
3286                                 nfs4_end_fop(mi, vp, NULL, OH_WRITE,
3287                                     &recov_state, needrecov);
3288                         /* hold needed for nfs4delegreturn_thread */
3289                         VN_HOLD(vp);
3290                         nfs4delegreturn_async(rp, (NFS4_DR_PUSH|NFS4_DR_REOPEN|
3291                             NFS4_DR_DISCARD), FALSE);
3292                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3293                         goto recov_retry;
3294                 }

3296                 if (needrecov) {
3297                         bool_t abort;

3299                         NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
3300                             "nfs4write: client got error %d, res.status %d"
3301                             ", so start recovery", e.error, res.status));

3303                         abort = nfs4_start_recovery(&e,
3304                             VTOMI4(vp), vp, NULL, &wargs->stateid,
3305                             NULL, OP_WRITE, NULL, NULL, NULL);
3306                         if (!e.error) {
3307                                 e.error = geterrno4(res.status);
3308                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3309                                     (caddr_t)&res);
3310                         }
3311                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3312                             &recov_state, needrecov);
3313                         if (abort == FALSE)
3314                                 goto recov_retry;
3315                         return (e.error);
3316                 }
```

```
3318                if (res.status) {
3319                        e.error = geterrno4(res.status);
3320                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3321                        if (!recov)
3322                                nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3323                                    &recov_state, needrecov);
3324                        return (e.error);
3325                }

3327                resop = &res.array[1];  /* write res */
3328                wres = &resop->nfs_resop4_u.opwrite;

3330                if ((int)wres->count > tsize) {
3331                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

3333                        zcmn_err(getzoneid(), CE_WARN,
3334                            "nfs4write: server wrote %u, requested was %u",
3335                            (int)wres->count, tsize);
3336                        if (!recov)
3337                                nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3338                                    &recov_state, needrecov);
3339                        return (EIO);
3340                }
3341                if (wres->committed == UNSTABLE4) {
3342                        *stab_comm = UNSTABLE4;
3343                        if (wargs->stable == DATA_SYNC4 ||
3344                            wargs->stable == FILE_SYNC4) {
3345                                (void) xdr_free(xdr_COMPOUND4res_clnt,
3346                                    (caddr_t)&res);
3347                                zcmn_err(getzoneid(), CE_WARN,
3348                                    "nfs4write: server %s did not commit "
3349                                    "to stable storage",
3350                                    rp->r_server->sv_hostname);
3351                                if (!recov)
3352                                        nfs4_end_fop(VTOMI4(vp), vp, NULL,
3353                                            OH_WRITE, &recov_state, needrecov);
3354                                return (EIO);
3355                        }
3356                }

3358                tsize = (int)wres->count;
3359                count -= tsize;
3360                base += tsize;
3361                offset += tsize;
3362                if (mi->mi_io_kstats) {
3363                        mutex_enter(&mi->mi_lock);
3364                        KSTAT_IO_PTR(mi->mi_io_kstats)->writes++;
3365                        KSTAT_IO_PTR(mi->mi_io_kstats)->nwritten +=
3366                            tsize;
3367                        mutex_exit(&mi->mi_lock);
3368                }
3369                lwp_stat_update(LWP_STAT_OUBLK, 1);
3370                mutex_enter(&rp->r_statelock);
3371                if (rp->r_flags & R4HAVEVERF) {
3372                        if (rp->r_writeverf != wres->writeverf) {
3373                                nfs4_set_mod(vp);
3374                                rp->r_writeverf = wres->writeverf;
3375                        }
3376                } else {
3377                        rp->r_writeverf = wres->writeverf;
3378                        rp->r_flags |= R4HAVEVERF;
3379                }
3380                PURGE_ATTRCACHE4_LOCKED(rp);
3381                rp->r_flags |= R4WRITEMODIFIED;
3382                gethrestime(&rp->r_attr.va_mtime);
3383                rp->r_attr.va_ctime = rp->r_attr.va_mtime;
```



```
3384                mutex_exit(&rp->r_statelock);
3385                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3386        } while (count);

3388        if (!recov)
3389                nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE, &recov_state,
3390                    needrecov);

3392        return (e.error);
3393 }

3395 /*
3396  * Read from a file.  Reads data in largest chunks our interface can handle.
3397  */
3398 static int
3399 nfs4read(vnode_t *vp, caddr_t base, offset_t offset, int count,
3400     size_t *residp, cred_t *cr, bool_t async, struct uio *uiop)
3401 {
3402        mntinfo4_t *mi;
3403        COMPOUND4args_clnt args;
3404        COMPOUND4res_clnt res;
3405        READ4args *rargs;
3406        nfs_argop4 argop[2];
3407        int tsize;
3408        int doqueue;
3409        rnode4_t *rp;
3410        int data_len;
3411        bool_t is_eof;
3412        bool_t needrecov = FALSE;
3413        nfs4_recov_state_t recov_state;
3414        nfs4_stateid_types_t sid_types;
3415        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

3417        rp = VTOR4(vp);
3418        mi = VTOMI4(vp);
3419        doqueue = 1;

3421        ASSERT(nfs_zone() == mi->mi_zone);

3423        args.ctag = async ? TAG_READAHEAD : TAG_READ;

3425        args.array_len = 2;
3426        args.array = argop;

3428        nfs4_init_stateid_types(&sid_types);

3430        recov_state.rs_flags = 0;
3431        recov_state.rs_num_retry_despite_err = 0;

3433 recov_retry:
3434        e.error = nfs4_start_fop(mi, vp, NULL, OH_READ,
3435            &recov_state, NULL);
3436        if (e.error)
3437                return (e.error);

3439        /* putfh target fh */
3440        argop[0].argop = OP_CPUTFH;
3441        argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

3443        /* read */
3444        argop[1].argop = OP_READ;
3445        rargs = &argop[1].nfs_argop4_u.opread;
3446        rargs->stateid = nfs4_get_stateid(cr, rp, curproc->p_pidp->pid_id, mi,
3447            OP_READ, &sid_types, async);

3449        do {
```

```
3450                 if (mi->mi_io_kstats) {
3451                         mutex_enter(&mi->mi_lock);
3452                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
3453                         mutex_exit(&mi->mi_lock);
3454                 }

3456                 NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
3457                     "nfs4read: %s call, rp %s",
3458                     needrecov ? "recov" : "first",
3459                     rnode4info(rp)));

3461                 if ((vp->v_flag & VNOCACHE) ||
3462                     (rp->r_flags & R4DIRECTIO) ||
3463                     (mi->mi_flags & MI4_DIRECTIO))
3464                         tsize = MIN(mi->mi_tsize, count);
3465                 else
3466                         tsize = MIN(mi->mi_curread, count);

3468                 rargs->offset = (offset4)offset;
3469                 rargs->count = (count4)tsize;
3470                 rargs->res_data_val_alt = NULL;
3471                 rargs->res_mblk = NULL;
3472                 rargs->res_uiop = NULL;
3473                 rargs->res_maxsize = 0;
3474                 rargs->wlist = NULL;

3476                 if (uiop)
3477                         rargs->res_uiop = uiop;
3478                 else
3479                         rargs->res_data_val_alt = base;
3480                 rargs->res_maxsize = tsize;

3482                 rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);
3483 #ifdef  DEBUG
3484                 if (nfs4read_error_inject) {
3485                         res.status = nfs4read_error_inject;
3486                         nfs4read_error_inject = 0;
3487                 }
3488 #endif

3490                 if (mi->mi_io_kstats) {
3491                         mutex_enter(&mi->mi_lock);
3492                         kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
3493                         mutex_exit(&mi->mi_lock);
3494                 }

3496                 needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
3497                 if (e.error != 0 && !needrecov) {
3498                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3499                             &recov_state, needrecov);
3500                         return (e.error);
3501                 }

3503                 /*
3504                  * Do proper retry for OLD and BAD stateid errors outside
3505                  * of the normal recovery framework.  There are two differences
3506                  * between async and sync reads.  The first is that we allow
3507                  * retry on BAD_STATEID for async reads, but not sync reads.
3508                  * The second is that we mark the file dead for a failed
3509                  * attempt with a special stateid for sync reads, but just
3510                  * return EIO for async reads.
3511                  *
3512                  * If a sync read receives a BAD stateid error while using a
3513                  * delegation stateid, retry using the open stateid (if it
3514                  * exists).  If it doesn't have an open stateid, reopen the
3515                  * file first, then retry.
```



```
3516                  */
3517                 if (e.error == 0 && (res.status == NFS4ERR_OLD_STATEID ||
3518                     res.status == NFS4ERR_BAD_STATEID) && async) {
3519                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3520                             &recov_state, needrecov);
3521                         if (sid_types.cur_sid_type == SPEC_SID) {
3522                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3523                                     (caddr_t)&res);
3524                                 return (EIO);
3525                         }
3526                         nfs4_save_stateid(&rargs->stateid, &sid_types);
3527                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3528                         goto recov_retry;
3529                 } else if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
3530                     !async && sid_types.cur_sid_type != SPEC_SID) {
3531                         nfs4_save_stateid(&rargs->stateid, &sid_types);
3532                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3533                             &recov_state, needrecov);
3534                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3535                         goto recov_retry;
3536                 } else if (e.error == 0 && res.status == NFS4ERR_BAD_STATEID &&
3537                     sid_types.cur_sid_type == DEL_SID) {
3538                         nfs4_save_stateid(&rargs->stateid, &sid_types);
3539                         mutex_enter(&rp->r_statev4_lock);
3540                         rp->r_deleg_return_pending = TRUE;
3541                         mutex_exit(&rp->r_statev4_lock);
3542                         if (nfs4rdwr_check_osid(vp, &e, cr)) {
3543                                 nfs4_end_fop(mi, vp, NULL, OH_READ,
3544                                     &recov_state, needrecov);
3545                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3546                                     (caddr_t)&res);
3547                                 return (EIO);
3548                         }
3549                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3550                             &recov_state, needrecov);
3551                         /* hold needed for nfs4delegreturn_thread */
3552                         VN_HOLD(vp);
3553                         nfs4delegreturn_async(rp, (NFS4_DR_PUSH|NFS4_DR_REOPEN|
3554                             NFS4_DR_DISCARD), FALSE);
3555                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3556                         goto recov_retry;
3557                 }
3558                 if (needrecov) {
3559                         bool_t abort;

3561                         NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
3562                             "nfs4read: initiating recovery\n"));
3563                         abort = nfs4_start_recovery(&e,
3564                             mi, vp, NULL, &rargs->stateid,
3565                             NULL, OP_READ, NULL, NULL, NULL);
3566                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3567                             &recov_state, needrecov);
3568                         /*
3569                          * Do not retry if we got OLD_STATEID using a special
3570                          * stateid.  This avoids looping with a broken server.
3571                          */
3572                         if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
3573                             sid_types.cur_sid_type == SPEC_SID)
3574                                 abort = TRUE;

3576                         if (abort == FALSE) {
3577                                 /*
3578                                  * Need to retry all possible stateids in
3579                                  * case the recovery error wasn't stateid
3580                                  * related or the stateids have become
3581                                  * stale (server reboot).
```

```
3582                                 */
3583                                nfs4_init_stateid_types(&sid_types);
3584                                (void) xdr_free(xdr_COMPOUND4res_clnt,
3585                                    (caddr_t)&res);
3586                                goto recov_retry;
3587                        }

3589                        if (!e.error) {
3590                                e.error = geterrno4(res.status);
3591                                (void) xdr_free(xdr_COMPOUND4res_clnt,
3592                                    (caddr_t)&res);
3593                        }
3594                        return (e.error);
3595                }

3597                if (res.status) {
3598                        e.error = geterrno4(res.status);
3599                        nfs4_end_fop(mi, vp, NULL, OH_READ,
3600                            &recov_state, needrecov);
3601                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3602                        return (e.error);
3603                }

3605                data_len = res.array[1].nfs_resop4_u.opread.data_len;
3606                count -= data_len;
3607                if (base)
3608                        base += data_len;
3609                offset += data_len;
3610                if (mi->mi_io_kstats) {
3611                        mutex_enter(&mi->mi_lock);
3612                        KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
3613                        KSTAT_IO_PTR(mi->mi_io_kstats)->nread += data_len;
3614                        mutex_exit(&mi->mi_lock);
3615                }
3616                lwp_stat_update(LWP_STAT_INBLK, 1);
3617                is_eof = res.array[1].nfs_resop4_u.opread.eof;
3618                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

3620        } while (count && !is_eof);

3622        *residp = count;

3624        nfs4_end_fop(mi, vp, NULL, OH_READ, &recov_state, needrecov);

3626        return (e.error);
3627 }

3629 /* ARGSUSED */
3630 static int
3631 nfs4_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp,
3632     caller_context_t *ct)
3633 {
3634        if (nfs_zone() != VTOMI4(vp)->mi_zone)
3635                return (EIO);
3636        switch (cmd) {
3637                case _FIODIRECTIO:
3638                        return (nfs4_directio(vp, (int)arg, cr));
3639                default:
3640                        return (ENOTTY);
3641        }
3642 }

3644 /* ARGSUSED */
3645 int
3646 nfs4_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
3647     caller_context_t *ct)
```



```
3648 {
3649        int error;
3650        rnode4_t *rp = VTOR4(vp);

3652        if (nfs_zone() != VTOMI4(vp)->mi_zone)
3653                return (EIO);
3654        /*
3655         * If it has been specified that the return value will
3656         * just be used as a hint, and we are only being asked
3657         * for size, fsid or rdevid, then return the client's
3658         * notion of these values without checking to make sure
3659         * that the attribute cache is up to date.
3660         * The whole point is to avoid an over the wire GETATTR
3661         * call.
3662         */
3663        if (flags & ATTR_HINT) {
3664                if (!(vap->va_mask & ~(AT_SIZE | AT_FSID | AT_RDEV))) {
3665                        mutex_enter(&rp->r_statelock);
3666                        if (vap->va_mask & AT_SIZE)
3667                                vap->va_size = rp->r_size;
3668                        if (vap->va_mask & AT_FSID)
3669                                vap->va_fsid = rp->r_attr.va_fsid;
3670                        if (vap->va_mask & AT_RDEV)
3671                                vap->va_rdev = rp->r_attr.va_rdev;
3672                        mutex_exit(&rp->r_statelock);
3673                        return (0);
3674                }
3675        }

3677        /*
3678         * Only need to flush pages if asking for the mtime
3679         * and if there any dirty pages or any outstanding
3680         * asynchronous (write) requests for this file.
3681         */
3682        if (vap->va_mask & AT_MTIME) {
3683                rp = VTOR4(vp);
3684                if (nfs4_has_pages(vp)) {
3685                        mutex_enter(&rp->r_statev4_lock);
3686                        if (rp->r_deleg_type != OPEN_DELEGATE_WRITE) {
3687                                mutex_exit(&rp->r_statev4_lock);
3688                                if (rp->r_flags & R4DIRTY ||
3689                                    rp->r_awcount > 0) {
3690                                        mutex_enter(&rp->r_statelock);
3691                                        rp->r_gcount++;
3692                                        mutex_exit(&rp->r_statelock);
3693                                        error =
3694                                            nfs4_putpage(vp, (u_offset_t)0,
3695                                            0, 0, cr, NULL);
3696                                        mutex_enter(&rp->r_statelock);
3697                                        if (error && (error == ENOSPC ||
3698                                            error == EDQUOT)) {
3699                                                if (!rp->r_error)
3700                                                        rp->r_error = error;
3701                                        }
3702                                        if (--rp->r_gcount == 0)
3703                                                cv_broadcast(&rp->r_cv);
3704                                        mutex_exit(&rp->r_statelock);
3705                                }
3706                        } else {
3707                                mutex_exit(&rp->r_statev4_lock);
3708                        }
3709                }
3710        }
3711        return (nfs4getattr(vp, vap, cr));
3712 }
```

```
3714 int
3715 nfs4_compare_modes(mode_t from_server, mode_t on_client)
3716 {
3717         /*
3718          * If these are the only two bits cleared
3719          * on the server then return 0 (OK) else
3720          * return 1 (BAD).
3721          */
3722         on_client &= ~(S_ISUID|S_ISGID);
3723         if (on_client == from_server)
3724                 return (0);
3725         else
3726                 return (1);
3727 }

3729 /*ARGSUSED4*/
3730 static int
3731 nfs4_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
3732     caller_context_t *ct)
3733 {
3734         int error;

3736         if (vap->va_mask & AT_NOSET)
3737                 return (EINVAL);

3739         if (nfs_zone() != VTOMI4(vp)->mi_zone)
3740                 return (EIO);

3742         /*
3743          * Don't call secpolicy_vnode_setattr, the client cannot
3744          * use its cached attributes to make security decisions
3745          * as the server may be faking mode bits or mapping uid/gid.
3746          * Always just let the server to the checking.
3747          * If we provide the ability to remove basic priviledges
3748          * to setattr (e.g. basic without chmod) then we will
3749          * need to add a check here before calling the server.
3750          */
3751         error = nfs4setattr(vp, vap, flags, cr, NULL);

3753         if (error == 0 && (vap->va_mask & AT_SIZE) && vap->va_size == 0)
3754                 vnevent_truncate(vp, ct);

3756         return (error);
3757 }

3759 /*
3760  * To replace the "guarded" version 3 setattr, we use two types of compound
3761  * setattr requests:
3762  * 1. The "normal" setattr, used when the size of the file isn't being
3763  *    changed - { Putfh <fh>; Setattr; Getattr }/
3764  * 2. If the size is changed, precede Setattr with: Getattr; Verify
3765  *    with only ctime as the argument. If the server ctime differs from
3766  *    what is cached on the client, the verify will fail, but we would
3767  *    already have the ctime from the preceding getattr, so just set it
3768  *    and retry. Thus the compound here is - { Putfh <fh>; Getattr; Verify;
3769  *      Setattr; Getattr }.
3770  *
3771  * The vsecattr_t * input parameter will be non-NULL if ACLs are being set in
3772  * this setattr and NULL if they are not.
3773  */
3774 static int
3775 nfs4setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
3776     vsecattr_t *vsap)
3777 {
3778         COMPOUND4args_clnt args;
3779         COMPOUND4res_clnt res, *resp = NULL;
```



```
3780         nfs4_ga_res_t *garp = NULL;
3781         int numops = 3;                 /* { Putfh; Setattr; Getattr } */
3782         nfs_argop4 argop[5];
3783         int verify_argop = -1;
3784         int setattr_argop = 1;
3785         nfs_resop4 *resop;
3786         vattr_t va;
3787         rnode4_t *rp;
3788         int doqueue = 1;
3789         uint_t mask = vap->va_mask;
3790         mode_t omode;
3791         vsecattr_t *vsp;
3792         timestruc_t ctime;
3793         bool_t needrecov = FALSE;
3794         nfs4_recov_state_t recov_state;
3795         nfs4_stateid_types_t sid_types;
3796         stateid4 stateid;
3797         hrtime_t t;
3798         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
3799         servinfo4_t *svp;
3800         bitmap4 supp_attrs;

3802         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
3803         rp = VTOR4(vp);
3804         nfs4_init_stateid_types(&sid_types);

3806         /*
3807          * Only need to flush pages if there are any pages and
3808          * if the file is marked as dirty in some fashion.  The
3809          * file must be flushed so that we can accurately
3810          * determine the size of the file and the cached data
3811          * after the SETATTR returns.  A file is considered to
3812          * be dirty if it is either marked with R4DIRTY, has
3813          * outstanding i/o's active, or is mmap'd.  In this
3814          * last case, we can't tell whether there are dirty
3815          * pages, so we flush just to be sure.
3816          */
3817         if (nfs4_has_pages(vp) &&
3818             ((rp->r_flags & R4DIRTY) ||
3819             rp->r_count > 0 ||
3820             rp->r_mapcnt > 0)) {
3821                 ASSERT(vp->v_type != VCHR);
3822                 e.error = nfs4_putpage(vp, (offset_t)0, 0, 0, cr, NULL);
3823                 if (e.error && (e.error == ENOSPC || e.error == EDQUOT)) {
3824                         mutex_enter(&rp->r_statelock);
3825                         if (!rp->r_error)
3826                                 rp->r_error = e.error;
3827                         mutex_exit(&rp->r_statelock);
3828                 }
3829         }

3831         if (mask & AT_SIZE) {
3832                 /*
3833                  * Verification setattr compound for non-deleg AT_SIZE:
3834                  *      { Putfh; Getattr; Verify; Setattr; Getattr }
3835                  * Set ctime local here (outside the do_again label)
3836                  * so that subsequent retries (after failed VERIFY)
3837                  * will use ctime from GETATTR results (from failed
3838                  * verify compound) as VERIFY arg.
3839                  * If file has delegation, then VERIFY(time_metadata)
3840                  * is of little added value, so don't bother.
3841                  */
3842                 mutex_enter(&rp->r_statev4_lock);
3843                 if (rp->r_deleg_type == OPEN_DELEGATE_NONE ||
3844                     rp->r_deleg_return_pending) {
3845                         numops = 5;
```

```
3846                        ctime = rp->r_attr.va_ctime;
3847                }
3848                mutex_exit(&rp->r_statev4_lock);
3849        }

3851        recov_state.rs_flags = 0;
3852        recov_state.rs_num_retry_despite_err = 0;

3854        args.ctag = TAG_SETATTR;
3855 do_again:
3856 recov_retry:
3857        setattr_argop = numops - 2;

3859        args.array = argop;
3860        args.array_len = numops;

3862        e.error = nfs4_start_op(VTOMI4(vp), vp, NULL, &recov_state);
3863        if (e.error)
3864                return (e.error);


3867        /* putfh target fh */
3868        argop[0].argop = OP_CPUTFH;
3869        argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

3871        if (numops == 5) {
3872                /*
3873                 * We only care about the ctime, but need to get mtime
3874                 * and size for proper cache update.
3875                 */
3876                /* getattr */
3877                argop[1].argop = OP_GETATTR;
3878                argop[1].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
3879                argop[1].nfs_argop4_u.opgetattr.mi = VTOMI4(vp);

3881                /* verify - set later in loop */
3882                verify_argop = 2;
3883        }

3885        /* setattr */
3886        svp = rp->r_server;
3887        (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
3888        supp_attrs = svp->sv_supp_attrs;
3889        nfs_rw_exit(&svp->sv_lock);

3891        nfs4args_setattr(&argop[setattr_argop], vap, vsap, flags, rp, cr,
3892            supp_attrs, &e.error, &sid_types);
3893        stateid = argop[setattr_argop].nfs_argop4_u.opsetattr.stateid;
3894        if (e.error) {
3895                /* req time field(s) overflow - return immediately */
3896                nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state, needrecov);
3897                nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
3898                    opsetattr.obj_attributes);
3899                return (e.error);
3900        }
3901        omode = rp->r_attr.va_mode;

3903        /* getattr */
3904        argop[numops-1].argop = OP_GETATTR;
3905        argop[numops-1].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
3906        /*
3907         * If we are setting the ACL (indicated only by vsap != NULL), request
3908         * the ACL in this getattr.  The ACL returned from this getattr will be
3909         * used in updating the ACL cache.
3910         */
3911        if (vsap != NULL)
```



```
3912                argop[numops-1].nfs_argop4_u.opgetattr.attr_request |=
3913                    FATTR4_ACL_MASK;
3914        argop[numops-1].nfs_argop4_u.opgetattr.mi = VTOMI4(vp);

3916        /*
3917         * setattr iterates if the object size is set and the cached ctime
3918         * does not match the file ctime. In that case, verify the ctime first.
3919         */

3921        do {
3922                if (verify_argop != -1) {
3923                        /*
3924                         * Verify that the ctime match before doing setattr.
3925                         */
3926                        va.va_mask = AT_CTIME;
3927                        va.va_ctime = ctime;
3928                        svp = rp->r_server;
3929                        (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
3930                        supp_attrs = svp->sv_supp_attrs;
3931                        nfs_rw_exit(&svp->sv_lock);
3932                        e.error = nfs4args_verify(&argop[verify_argop], &va,
3933                            OP_VERIFY, supp_attrs);
3934                        if (e.error) {
3935                                /* req time field(s) overflow - return */
3936                                nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3937                                    needrecov);
3938                                break;
3939                        }
3940                }

3942                doqueue = 1;

3944                t = gethrtime();

3946                rfs4call(VTOMI4(vp), &args, &res, cr, &doqueue, 0, &e);

3948                /*
3949                 * Purge the access cache and ACL cache if changing either the
3950                 * owner of the file, the group owner, or the mode.  These may
3951                 * change the access permissions of the file, so purge old
3952                 * information and start over again.
3953                 */
3954                if (mask & (AT_UID | AT_GID | AT_MODE)) {
3955                        (void) nfs4_access_purge_rp(rp);
3956                        if (rp->r_secattr != NULL) {
3957                                mutex_enter(&rp->r_statelock);
3958                                vsp = rp->r_secattr;
3959                                rp->r_secattr = NULL;
3960                                mutex_exit(&rp->r_statelock);
3961                                if (vsp != NULL)
3962                                        nfs4_acl_free_cache(vsp);
3963                        }
3964                }

3966                /*
3967                 * If res.array_len == numops, then everything succeeded,
3968                 * except for possibly the final getattr.  If only the
3969                 * last getattr failed, give up, and don't try recovery.
3970                 */
3971                if (res.array_len == numops) {
3972                        nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3973                            needrecov);
3974                        if (! e.error)
3975                                resp = &res;
3976                        break;
3977                }
```

```
3979                 /*
3980                  * if either rpc call failed or completely succeeded - done
3981                  */
3982                 needrecov = nfs4_needs_recovery(&e, FALSE, vp->v_vfsp);
3983                 if (e.error) {
3984                         PURGE_ATTRCACHE4(vp);
3985                         if (!needrecov) {
3986                                 nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3987                                     needrecov);
3988                                 break;
3989                         }
3990                 }

3992                 /*
3993                  * Do proper retry for OLD_STATEID outside of the normal
3994                  * recovery framework.
3995                  */
3996                 if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
3997                     sid_types.cur_sid_type != SPEC_SID &&
3998                     sid_types.cur_sid_type != NO_SID) {
3999                         nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
4000                             needrecov);
4001                         nfs4_save_stateid(&stateid, &sid_types);
4002                         nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4003                             opsetattr.obj_attributes);
4004                         if (verify_argop != -1) {
4005                                 nfs4args_verify_free(&argop[verify_argop]);
4006                                 verify_argop = -1;
4007                         }
4008                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4009                         goto recov_retry;
4010                 }

4012                 if (needrecov) {
4013                         bool_t abort;

4015                         abort = nfs4_start_recovery(&e,
4016                             VTOMI4(vp), vp, NULL, NULL, NULL,
4017                             OP_SETATTR, NULL, NULL, NULL);
4018                         nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
4019                             needrecov);
4020                         /*
4021                          * Do not retry if we failed with OLD_STATEID using
4022                          * a special stateid.  This is done to avoid looping
4023                          * with a broken server.
4024                          */
4025                         if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
4026                             (sid_types.cur_sid_type == SPEC_SID ||
4027                             sid_types.cur_sid_type == NO_SID))
4028                                 abort = TRUE;
4029                         if (!e.error) {
4030                                 if (res.status == NFS4ERR_BADOWNER)
4031                                         nfs4_log_badowner(VTOMI4(vp),
4032                                             OP_SETATTR);

4034                                 e.error = geterrno4(res.status);
4035                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4036                                     (caddr_t)&res);
4037                         }
4038                         nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4039                             opsetattr.obj_attributes);
4040                         if (verify_argop != -1) {
4041                                 nfs4args_verify_free(&argop[verify_argop]);
4042                                 verify_argop = -1;
4043                         }
```



```
4044                         if (abort == FALSE) {
4045                                 /*
4046                                  * Need to retry all possible stateids in
4047                                  * case the recovery error wasn't stateid
4048                                  * related or the stateids have become
4049                                  * stale (server reboot).
4050                                  */
4051                                 nfs4_init_stateid_types(&sid_types);
4052                                 goto recov_retry;
4053                         }
4054                         return (e.error);
4055                 }

4057                 /*
4058                  * Need to call nfs4_end_op before nfs4getattr to
4059                  * avoid potential nfs4_start_op deadlock. See RFE
4060                  * 4777612.  Calls to nfs4_invalidate_pages() and
4061                  * nfs4_purge_stale_fh() might also generate over the
4062                  * wire calls which my cause nfs4_start_op() deadlock.
4063                  */
4064                 nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state, needrecov);

4066                 /*
4067                  * Check to update lease.
4068                  */
4069                 resp = &res;
4070                 if (res.status == NFS4_OK) {
4071                         break;
4072                 }

4074                 /*
4075                  * Check if verify failed to see if try again
4076                  */
4077                 if ((verify_argop == -1) || (res.array_len != 3)) {
4078                         /*
4079                          * can't continue...
4080                          */
4081                         if (res.status == NFS4ERR_BADOWNER)
4082                                 nfs4_log_badowner(VTOMI4(vp), OP_SETATTR);

4084                         e.error = geterrno4(res.status);
4085                 } else {
4086                         /*
4087                          * When the verify request fails, the client ctime is
4088                          * not in sync with the server. This is the same as
4089                          * the version 3 "not synchronized" error, and we
4090                          * handle it in a similar manner (XXX do we need to???).
4091                          * Use the ctime returned in the first getattr for
4092                          * the input to the next verify.
4093                          * If we couldn't get the attributes, then we give up
4094                          * because we can't complete the operation as required.
4095                          */
4096                         garp = &res.array[1].nfs_resop4_u.opgetattr.ga_res;
4097                 }
4098                 if (e.error) {
4099                         PURGE_ATTRCACHE4(vp);
4100                         nfs4_purge_stale_fh(e.error, vp, cr);
4101                 } else {
4102                         /*
4103                          * retry with a new verify value
4104                          */
4105                         ctime = garp->n4g_va.va_ctime;
4106                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4107                         resp = NULL;
4108                 }
4109                 if (!e.error) {
```

```
4110                        nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4111                            opsetattr.obj_attributes);
4112                        if (verify_argop != -1) {
4113                                nfs4args_verify_free(&argop[verify_argop]);
4114                                verify_argop = -1;
4115                        }
4116                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4117                        goto do_again;
4118                }
4119        } while (!e.error);

4121        if (e.error) {
4122                /*
4123                 * If we are here, rfs4call has an irrecoverable error - return
4124                 */
4125                nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4126                    opsetattr.obj_attributes);
4127                if (verify_argop != -1) {
4128                        nfs4args_verify_free(&argop[verify_argop]);
4129                        verify_argop = -1;
4130                }
4131                if (resp)
4132                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
4133                return (e.error);
4134        }




4138        /*
4139         * If changing the size of the file, invalidate
4140         * any local cached data which is no longer part
4141         * of the file.  We also possibly invalidate the
4142         * last page in the file.  We could use
4143         * pvn_vpzero(), but this would mark the page as
4144         * modified and require it to be written back to
4145         * the server for no particularly good reason.
4146         * This way, if we access it, then we bring it
4147         * back in.  A read should be cheaper than a
4148         * write.
4149         */
4150        if (mask & AT_SIZE) {
4151                nfs4_invalidate_pages(vp, (vap->va_size & PAGEMASK), cr);
4152        }

4154        /* either no error or one of the postop getattr failed */

4156        /*
4157         * XXX Perform a simplified version of wcc checking. Instead of
4158         * have another getattr to get pre-op, just purge cache if
4159         * any of the ops prior to and including the getattr failed.
4160         * If the getattr succeeded then update the attrcache accordingly.
4161         */

4163        garp = NULL;
4164        if (res.status == NFS4_OK) {
4165                /*
4166                 * Last getattr
4167                 */
4168                resop = &res.array[numops - 1];
4169                garp = &resop->nfs_resop4_u.opgetattr.ga_res;
4170        }
4171        /*
4172         * In certain cases, nfs4_update_attrcache() will purge the attrcache,
4173         * rather than filling it.  See the function itself for details.
4174         */
4175        e.error = nfs4_update_attrcache(res.status, garp, t, vp, cr);
```



```
4176        if (garp != NULL) {
4177                if (garp->n4g_resbmap & FATTR4_ACL_MASK) {
4178                        nfs4_acl_fill_cache(rp, &garp->n4g_vsa);
4179                        vs_ace4_destroy(&garp->n4g_vsa);
4180                } else {
4181                        if (vsap != NULL) {
4182                                /*
4183                                 * The ACL was supposed to be set and to be
4184                                 * returned in the last getattr of this
4185                                 * compound, but for some reason the getattr
4186                                 * result doesn't contain the ACL.  In this
4187                                 * case, purge the ACL cache.
4188                                 */
4189                                if (rp->r_secattr != NULL) {
4190                                        mutex_enter(&rp->r_statelock);
4191                                        vsp = rp->r_secattr;
4192                                        rp->r_secattr = NULL;
4193                                        mutex_exit(&rp->r_statelock);
4194                                        if (vsp != NULL)
4195                                                nfs4_acl_free_cache(vsp);
4196                                }
4197                        }
4198                }
4199        }

4201        if (res.status == NFS4_OK && (mask & AT_SIZE)) {
4202                /*
4203                 * Set the size, rather than relying on getting it updated
4204                 * via a GETATTR.  With delegations the client tries to
4205                 * suppress GETATTR calls.
4206                 */
4207                mutex_enter(&rp->r_statelock);
4208                rp->r_size = vap->va_size;
4209                mutex_exit(&rp->r_statelock);
4210        }

4212        /*
4213         * Can free up request args and res
4214         */
4215        nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4216            opsetattr.obj_attributes);
4217        if (verify_argop != -1) {
4218                nfs4args_verify_free(&argop[verify_argop]);
4219                verify_argop = -1;
4220        }
4221        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

4223        /*
4224         * Some servers will change the mode to clear the setuid
4225         * and setgid bits when changing the uid or gid.  The
4226         * client needs to compensate appropriately.
4227         */
4228        if (mask & (AT_UID | AT_GID)) {
4229                int terror, do_setattr;

4231                do_setattr = 0;
4232                va.va_mask = AT_MODE;
4233                terror = nfs4getattr(vp, &va, cr);
4234                if (!terror &&
4235                    (((mask & AT_MODE) && va.va_mode != vap->va_mode) ||
4236                    (!(mask & AT_MODE) && va.va_mode != omode))) {
4237                        va.va_mask = AT_MODE;
4238                        if (mask & AT_MODE) {
4239                                /*
4240                                 * We asked the mode to be changed and what
4241                                 * we just got from the server in getattr is
```

```
4242                                 * not what we wanted it to be, so set it now.
4243                                 */
4244                                va.va_mode = vap->va_mode;
4245                                do_setattr = 1;
4246                        } else {
4247                                /*
4248                                 * We did not ask the mode to be changed,
4249                                 * Check to see that the server just cleared
4250                                 * I_SUID and I_GUID from it. If not then
4251                                 * set mode to omode with UID/GID cleared.
4252                                 */
4253                                if (nfs4_compare_modes(va.va_mode, omode)) {
4254                                        omode &= ~(S_ISUID|S_ISGID);
4255                                        va.va_mode = omode;
4256                                        do_setattr = 1;
4257                                }
4258                        }
4259
4260                        if (do_setattr)
4261                                (void) nfs4setattr(vp, &va, 0, cr, NULL);
4262                }
4263        }
4264
4265        return (e.error);
4266 }
4267
4268 /* ARGSUSED */
4269 static int
4270 nfs4_access(vnode_t *vp, int mode, int flags, cred_t *cr, caller_context_t *ct)
4271 {
4272        COMPOUND4args_clnt args;
4273        COMPOUND4res_clnt res;
4274        int doqueue;
4275        uint32_t acc, resacc, argacc;
4276        rnode4_t *rp;
4277        cred_t *cred, *ncr, *ncrfree = NULL;
4278        nfs4_access_type_t cacc;
4279        int num_ops;
4280        nfs_argop4 argop[3];
4281        nfs_resop4 *resop;
4282        bool_t needrecov = FALSE, do_getattr;
4283        nfs4_recov_state_t recov_state;
4284        int rpc_error;
4285        hrtime_t t;
4286        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
4287        mntinfo4_t *mi = VTOMI4(vp);
4288
4289        if (nfs_zone() != mi->mi_zone)
4290                return (EIO);
4291
4292        acc = 0;
4293        if (mode & VREAD)
4294                acc |= ACCESS4_READ;
4295        if (mode & VWRITE) {
4296                if ((vp->v_vfsp->vfs_flag & VFS_RDONLY) && !ISVDEV(vp->v_type))
4297                        return (EROFS);
4298                if (vp->v_type == VDIR)
4299                        acc |= ACCESS4_DELETE;
4300                acc |= ACCESS4_MODIFY | ACCESS4_EXTEND;
4301        }
4302        if (mode & VEXEC) {
4303                if (vp->v_type == VDIR)
4304                        acc |= ACCESS4_LOOKUP;
4305                else
4306                        acc |= ACCESS4_EXECUTE;
4307        }
```



```
4309        if (VTOR4(vp)->r_acache != NULL) {
4310                e.error = nfs4_validate_caches(vp, cr);
4311                if (e.error)
4312                        return (e.error);
4313        }
4314
4315        rp = VTOR4(vp);
4316        if (vp->v_type == VDIR)
4317                argacc = ACCESS4_READ | ACCESS4_DELETE | ACCESS4_MODIFY |
4318                    ACCESS4_EXTEND | ACCESS4_LOOKUP;
4319        else
4320                argacc = ACCESS4_READ | ACCESS4_MODIFY | ACCESS4_EXTEND |
4321                    ACCESS4_EXECUTE;
4322        recov_state.rs_flags = 0;
4323        recov_state.rs_num_retry_despite_err = 0;
4324
4325        cred = cr;
4326        /*
4327         * ncr and ncrfree both initially
4328         * point to the memory area returned
4329         * by crnetadjust();
4330         * ncrfree not NULL when exiting means
4331         * that we need to release it
4332         */
4333        ncr = crnetadjust(cred);
4334        ncrfree = ncr;
4335
4336 tryagain:
4337        cacc = nfs4_access_check(rp, acc, cred);
4338        if (cacc == NFS4_ACCESS_ALLOWED) {
4339                if (ncrfree != NULL)
4340                        crfree(ncrfree);
4341                return (0);
4342        }
4343        if (cacc == NFS4_ACCESS_DENIED) {
4344                /*
4345                 * If the cred can be adjusted, try again
4346                 * with the new cred.
4347                 */
4348                if (ncr != NULL) {
4349                        cred = ncr;
4350                        ncr = NULL;
4351                        goto tryagain;
4352                }
4353                if (ncrfree != NULL)
4354                        crfree(ncrfree);
4355                return (EACCES);
4356        }
4357
4358 recov_retry:
4359        /*
4360         * Don't take with r_statev4_lock here. r_deleg_type could
4361         * change as soon as lock is released.  Since it is an int,
4362         * there is no atomicity issue.
4363         */
4364        do_getattr = (rp->r_deleg_type == OPEN_DELEGATE_NONE);
4365        num_ops = do_getattr ? 3 : 2;
4366
4367        args.ctag = TAG_ACCESS;
4368
4369        args.array_len = num_ops;
4370        args.array = argop;
4371
4372        if (e.error = nfs4_start_fop(mi, vp, NULL, OH_ACCESS,
4373            &recov_state, NULL)) {
```

```
4374                 if (ncrfree != NULL)
4375                         crfree(ncrfree);
4376                 return (e.error);
4377         }

4379         /* putfh target fh */
4380         argop[0].argop = OP_CPUTFH;
4381         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(vp)->r_fh;

4383         /* access */
4384         argop[1].argop = OP_ACCESS;
4385         argop[1].nfs_argop4_u.opaccess.access = argacc;

4387         /* getattr */
4388         if (do_getattr) {
4389                 argop[2].argop = OP_GETATTR;
4390                 argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
4391                 argop[2].nfs_argop4_u.opgetattr.mi = mi;
4392         }

4394         NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
4395             "nfs4_access: %s call, rp %s", needrecov ? "recov" : "first",
4396             rnode4info(VTOR4(vp))));

4398         doqueue = 1;
4399         t = gethrtime();
4400         rfs4call(VTOMI4(vp), &args, &res, cred, &doqueue, 0, &e);
4401         rpc_error = e.error;

4403         needrecov = nfs4_needs_recovery(&e, FALSE, vp->v_vfsp);
4404         if (needrecov) {
4405                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
4406                     "nfs4_access: initiating recovery\n"));

4408                 if (nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
4409                     NULL, OP_ACCESS, NULL, NULL, NULL) == FALSE) {
4410                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_ACCESS,
4411                             &recov_state, needrecov);
4412                         if (!e.error)
4413                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4414                                     (caddr_t)&res);
4415                         goto recov_retry;
4416                 }
4417         }
4418         nfs4_end_fop(mi, vp, NULL, OH_ACCESS, &recov_state, needrecov);

4420         if (e.error)
4421                 goto out;

4423         if (res.status) {
4424                 e.error = geterrno4(res.status);
4425                 /*
4426                  * This might generate over the wire calls throught
4427                  * nfs4_invalidate_pages. Hence we need to call nfs4_end_op()
4428                  * here to avoid a deadlock.
4429                  */
4430                 nfs4_purge_stale_fh(e.error, vp, cr);
4431                 goto out;
4432         }
4433         resop = &res.array[1];  /* access res */

4435         resacc = resop->nfs_resop4_u.opaccess.access;

4437         if (do_getattr) {
4438                 resop++;        /* getattr res */
4439                 nfs4_attr_cache(vp, &resop->nfs_resop4_u.opgetattr.ga_res,
```



```
4440                     t, cr, FALSE, NULL);
4441         }

4443         if (!e.error) {
4444                 nfs4_access_cache(rp, argacc, resacc, cred);
4445                 /*
4446                  * we just cached results with cred; if cred is the
4447                  * adjusted credentials from crnetadjust, we do not want
4448                  * to release them before exiting: hence setting ncrfree
4449                  * to NULL
4450                  */
4451                 if (cred != cr)
4452                         ncrfree = NULL;
4453                 /* XXX check the supported bits too? */
4454                 if ((acc & resacc) != acc) {
4455                         /*
4456                          * The following code implements the semantic
4457                          * that a setuid root program has *at least* the
4458                          * permissions of the user that is running the
4459                          * program.  See rfs3call() for more portions
4460                          * of the implementation of this functionality.
4461                          */
4462                         /* XXX-LP */
4463                         if (ncr != NULL) {
4464                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4465                                     (caddr_t)&res);
4466                                 cred = ncr;
4467                                 ncr = NULL;
4468                                 goto tryagain;
4469                         }
4470                         e.error = EACCES;
4471                 }
4472         }

4474 out:
4475         if (!rpc_error)
4476                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

4478         if (ncrfree != NULL)
4479                 crfree(ncrfree);

4481         return (e.error);
4482 }

4484 /* ARGSUSED */
4485 static int
4486 nfs4_readlink(vnode_t *vp, struct uio *uiop, cred_t *cr, caller_context_t *ct)
4487 {
4488         COMPOUND4args_clnt args;
4489         COMPOUND4res_clnt res;
4490         int doqueue;
4491         rnode4_t *rp;
4492         nfs_argop4 argop[3];
4493         nfs_resop4 *resop;
4494         READLINK4res *lr_res;
4495         nfs4_ga_res_t *garp;
4496         uint_t len;
4497         char *linkdata;
4498         bool_t needrecov = FALSE;
4499         nfs4_recov_state_t recov_state;
4500         hrtime_t t;
4501         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

4503         if (nfs_zone() != VTOMI4(vp)->mi_zone)
4504                 return (EIO);
4505         /*
```

```
4506          * Can't readlink anything other than a symbolic link.
4507          */
4508         if (vp->v_type != VLNK)
4509                 return (EINVAL);

4511         rp = VTOR4(vp);
4512         if (nfs4_do_symlink_cache && rp->r_symlink.contents != NULL) {
4513                 e.error = nfs4_validate_caches(vp, cr);
4514                 if (e.error)
4515                         return (e.error);
4516                 mutex_enter(&rp->r_statelock);
4517                 if (rp->r_symlink.contents != NULL) {
4518                         e.error = uiomove(rp->r_symlink.contents,
4519                             rp->r_symlink.len, UIO_READ, uiop);
4520                         mutex_exit(&rp->r_statelock);
4521                         return (e.error);
4522                 }
4523                 mutex_exit(&rp->r_statelock);
4524         }
4525         recov_state.rs_flags = 0;
4526         recov_state.rs_num_retry_despite_err = 0;

4528 recov_retry:
4529         args.array_len = 3;
4530         args.array = argop;
4531         args.ctag = TAG_READLINK;

4533         e.error = nfs4_start_op(VTOMI4(vp), vp, NULL, &recov_state);
4534         if (e.error) {
4535                 return (e.error);
4536         }

4538         /* 0. putfh symlink fh */
4539         argop[0].argop = OP_CPUTFH;
4540         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(vp)->r_fh;

4542         /* 1. readlink */
4543         argop[1].argop = OP_READLINK;

4545         /* 2. getattr */
4546         argop[2].argop = OP_GETATTR;
4547         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
4548         argop[2].nfs_argop4_u.opgetattr.mi = VTOMI4(vp);

4550         doqueue = 1;

4552         NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
4553             "nfs4_readlink: %s call, rp %s", needrecov ? "recov" : "first",
4554             rnode4info(VTOR4(vp))));

4556         t = gethrtime();

4558         rfs4call(VTOMI4(vp), &args, &res, cr, &doqueue, 0, &e);

4560         needrecov = nfs4_needs_recovery(&e, FALSE, vp->v_vfsp);
4561         if (needrecov) {
4562                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
4563                     "nfs4_readlink: initiating recovery\n"));

4565                 if (nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
4566                     NULL, OP_READLINK, NULL, NULL, NULL) == FALSE) {
4567                         if (!e.error)
4568                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4569                                     (caddr_t)&res);

4571                         nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
```



```
4572                             needrecov);
4573                         goto recov_retry;
4574                 }
4575         }

4577         nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state, needrecov);

4579         if (e.error)
4580                 return (e.error);

4582         /*
4583          * There is an path in the code below which calls
4584          * nfs4_purge_stale_fh(), which may generate otw calls through
4585          * nfs4_invalidate_pages. Hence we need to call nfs4_end_op()
4586          * here to avoid nfs4_start_op() deadlock.
4587          */

4589         if (res.status && (res.array_len < args.array_len)) {
4590                 /*
4591                  * either Putfh or Link failed
4592                  */
4593                 e.error = geterrno4(res.status);
4594                 nfs4_purge_stale_fh(e.error, vp, cr);
4595                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4596                 return (e.error);
4597         }

4599         resop = &res.array[1];  /* readlink res */
4600         lr_res = &resop->nfs_resop4_u.opreadlink;

4602         /*
4603          * treat symlink names as data
4604          */
4605         linkdata = utf8_to_str((utf8string *)&lr_res->link, &len, NULL);
4606         if (linkdata != NULL) {
4607                 int uio_len = len - 1;
4608                 /* len includes null byte, which we won't uiomove */
4609                 e.error = uiomove(linkdata, uio_len, UIO_READ, uiop);
4610                 if (nfs4_do_symlink_cache && rp->r_symlink.contents == NULL) {
4611                         mutex_enter(&rp->r_statelock);
4612                         if (rp->r_symlink.contents == NULL) {
4613                                 rp->r_symlink.contents = linkdata;
4614                                 rp->r_symlink.len = uio_len;
4615                                 rp->r_symlink.size = len;
4616                                 mutex_exit(&rp->r_statelock);
4617                         } else {
4618                                 mutex_exit(&rp->r_statelock);
4619                                 kmem_free(linkdata, len);
4620                         }
4621                 } else {
4622                         kmem_free(linkdata, len);
4623                 }
4624         }
4625         if (res.status == NFS4_OK) {
4626                 resop++;        /* getattr res */
4627                 garp = &resop->nfs_resop4_u.opgetattr.ga_res;
4628         }
4629         e.error = nfs4_update_attrcache(res.status, garp, t, vp, cr);

4631         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

4633         /*
4634          * The over the wire error for attempting to readlink something
4635          * other than a symbolic link is ENXIO.  However, we need to
4636          * return EINVAL instead of ENXIO, so we map it here.
4637          */
```

```
4638         return (e.error == ENXIO ? EINVAL : e.error);
4639 }

4641 /*
4642  * Flush local dirty pages to stable storage on the server.
4643  *
4644  * If FNODSYNC is specified, then there is nothing to do because
4645  * metadata changes are not cached on the client before being
4646  * sent to the server.
4647  */
4648 /* ARGSUSED */
4649 static int
4650 nfs4_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
4651 {
4652         int error;

4654         if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
4655                 return (0);
4656         if (nfs_zone() != VTOMI4(vp)->mi_zone)
4657                 return (EIO);
4658         error = nfs4_putpage_commit(vp, (offset_t)0, 0, cr);
4659         if (!error)
4660                 error = VTOR4(vp)->r_error;
4661         return (error);
4662 }

4664 /*
4665  * Weirdness: if the file was removed or the target of a rename
4666  * operation while it was open, it got renamed instead.  Here we
4667  * remove the renamed file.
4668  */
4669 /* ARGSUSED */
4670 void
4671 nfs4_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
4672 {
4673         rnode4_t *rp;

4675         ASSERT(vp != DNLC_NO_VNODE);

4677         rp = VTOR4(vp);

4679         if (IS_SHADOW(vp, rp)) {
4680                 sv_inactive(vp);
4681                 return;
4682         }

4684         /*
4685          * If this is coming from the wrong zone, we let someone in the right
4686          * zone take care of it asynchronously.  We can get here due to
4687          * VN_RELE() being called from pageout() or fsflush().  This call may
4688          * potentially turn into an expensive no-op if, for instance, v_count
4689          * gets incremented in the meantime, but it's still correct.
4690          */
4691         if (nfs_zone() != VTOMI4(vp)->mi_zone) {
4692                 nfs4_async_inactive(vp, cr);
4693                 return;
4694         }

4696         /*
4697          * Some of the cleanup steps might require over-the-wire
4698          * operations.  Since VOP_INACTIVE can get called as a result of
4699          * other over-the-wire operations (e.g., an attribute cache update
4700          * can lead to a DNLC purge), doing those steps now would lead to a
4701          * nested call to the recovery framework, which can deadlock.  So
4702          * do any over-the-wire cleanups asynchronously, in a separate
4703          * thread.
```



```
4704          */

4706         mutex_enter(&rp->r_os_lock);
4707         mutex_enter(&rp->r_statelock);
4708         mutex_enter(&rp->r_statev4_lock);

4710         if (vp->v_type == VREG && list_head(&rp->r_open_streams) != NULL) {
4711                 mutex_exit(&rp->r_statev4_lock);
4712                 mutex_exit(&rp->r_statelock);
4713                 mutex_exit(&rp->r_os_lock);
4714                 nfs4_async_inactive(vp, cr);
4715                 return;
4716         }

4718         if (rp->r_deleg_type == OPEN_DELEGATE_READ ||
4719             rp->r_deleg_type == OPEN_DELEGATE_WRITE) {
4720                 mutex_exit(&rp->r_statev4_lock);
4721                 mutex_exit(&rp->r_statelock);
4722                 mutex_exit(&rp->r_os_lock);
4723                 nfs4_async_inactive(vp, cr);
4724                 return;
4725         }

4727         if (rp->r_unldvp != NULL) {
4728                 mutex_exit(&rp->r_statev4_lock);
4729                 mutex_exit(&rp->r_statelock);
4730                 mutex_exit(&rp->r_os_lock);
4731                 nfs4_async_inactive(vp, cr);
4732                 return;
4733         }
4734         mutex_exit(&rp->r_statev4_lock);
4735         mutex_exit(&rp->r_statelock);
4736         mutex_exit(&rp->r_os_lock);

4738         rp4_addfree(rp, cr);
4739 }

4741 /*
4742  * nfs4_inactive_otw - nfs4_inactive, plus over-the-wire calls to free up
4743  * various bits of state.  The caller must not refer to vp after this call.
4744  */

4746 void
4747 nfs4_inactive_otw(vnode_t *vp, cred_t *cr)
4748 {
4749         rnode4_t *rp = VTOR4(vp);
4750         nfs4_recov_state_t recov_state;
4751         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
4752         vnode_t *unldvp;
4753         char *unlname;
4754         cred_t *unlcred;
4755         COMPOUND4args_clnt args;
4756         COMPOUND4res_clnt res, *resp;
4757         nfs_argop4 argop[2];
4758         int doqueue;
4759 #ifdef DEBUG
4760         char *name;
4761 #endif

4763         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
4764         ASSERT(!IS_SHADOW(vp, rp));

4766 #ifdef DEBUG
4767         name = fn_name(VTOSV(vp)->sv_name);
4768         NFS4_DEBUG(nfs4_client_inactive_debug, (CE_NOTE, "nfs4_inactive_otw: "
4769             "release vnode %s", name));
```

```
4770         kmem_free(name, MAXNAMELEN);
4771 #endif
4772
4773         if (vp->v_type == VREG) {
4774                 bool_t recov_failed = FALSE;
4775
4776                 e.error = nfs4close_all(vp, cr);
4777                 if (e.error) {
4778                         /* Check to see if recovery failed */
4779                         mutex_enter(&(VTOMI4(vp)->mi_lock));
4780                         if (VTOMI4(vp)->mi_flags & MI4_RECOV_FAIL)
4781                                 recov_failed = TRUE;
4782                         mutex_exit(&(VTOMI4(vp)->mi_lock));
4783                         if (!recov_failed) {
4784                                 mutex_enter(&rp->r_statelock);
4785                                 if (rp->r_flags & R4RECOVERR)
4786                                         recov_failed = TRUE;
4787                                 mutex_exit(&rp->r_statelock);
4788                         }
4789                         if (recov_failed) {
4790                                 NFS4_DEBUG(nfs4_client_recov_debug,
4791                                     (CE_NOTE, "nfs4_inactive_otw: "
4792                                     "close failed (recovery failure)"));
4793                         }
4794                 }
4795         }
4796
4797 redo:
4798         if (rp->r_unldvp == NULL) {
4799                 rp4_addfree(rp, cr);
4800                 return;
4801         }
4802
4803         /*
4804          * Save the vnode pointer for the directory where the
4805          * unlinked-open file got renamed, then set it to NULL
4806          * to prevent another thread from getting here before
4807          * we're done with the remove.  While we have the
4808          * statelock, make local copies of the pertinent rnode
4809          * fields.  If we weren't to do this in an atomic way, the
4810          * the unl* fields could become inconsistent with respect
4811          * to each other due to a race condition between this
4812          * code and nfs_remove().  See bug report 1034328.
4813          */
4814         mutex_enter(&rp->r_statelock);
4815         if (rp->r_unldvp == NULL) {
4816                 mutex_exit(&rp->r_statelock);
4817                 rp4_addfree(rp, cr);
4818                 return;
4819         }
4820
4821         unldvp = rp->r_unldvp;
4822         rp->r_unldvp = NULL;
4823         unlname = rp->r_unlname;
4824         rp->r_unlname = NULL;
4825         unlcred = rp->r_unlcred;
4826         rp->r_unlcred = NULL;
4827         mutex_exit(&rp->r_statelock);
4828
4829         /*
4830          * If there are any dirty pages left, then flush
4831          * them.  This is unfortunate because they just
4832          * may get thrown away during the remove operation,
4833          * but we have to do this for correctness.
4834          */
4835         if (nfs4_has_pages(vp) &&
```



```
4836             ((rp->r_flags & R4DIRTY) || rp->r_count > 0)) {
4837                 ASSERT(vp->v_type != VCHR);
4838                 e.error = nfs4_putpage(vp, (u_offset_t)0, 0, 0, cr, NULL);
4839                 if (e.error) {
4840                         mutex_enter(&rp->r_statelock);
4841                         if (!rp->r_error)
4842                                 rp->r_error = e.error;
4843                         mutex_exit(&rp->r_statelock);
4844                 }
4845         }
4846
4847         recov_state.rs_flags = 0;
4848         recov_state.rs_num_retry_despite_err = 0;
4849 recov_retry_remove:
4850         /*
4851          * Do the remove operation on the renamed file
4852          */
4853         args.ctag = TAG_INACTIVE;
4854
4855         /*
4856          * Remove ops: putfh dir; remove
4857          */
4858         args.array_len = 2;
4859         args.array = argop;
4860
4861         e.error = nfs4_start_op(VTOMI4(unldvp), unldvp, NULL, &recov_state);
4862         if (e.error) {
4863                 kmem_free(unlname, MAXNAMELEN);
4864                 crfree(unlcred);
4865                 VN_RELE(unldvp);
4866                 /*
4867                  * Try again; this time around r_unldvp will be NULL, so we'll
4868                  * just call rp4_addfree() and return.
4869                  */
4870                 goto redo;
4871         }
4872
4873         /* putfh directory */
4874         argop[0].argop = OP_CPUTFH;
4875         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(unldvp)->r_fh;
4876
4877         /* remove */
4878         argop[1].argop = OP_CREMOVE;
4879         argop[1].nfs_argop4_u.opcremove.ctarget = unlname;
4880
4881         doqueue = 1;
4882         resp = &res;
4883
4884 #if 0 /* notyet */
4885         /*
4886          * Can't do this yet.  We may be being called from
4887          * dnlc_purge_XXX while that routine is holding a
4888          * mutex lock to the nc_rele list.  The calls to
4889          * nfs3_cache_wcc_data may result in calls to
4890          * dnlc_purge_XXX.  This will result in a deadlock.
4891          */
4892         rfs4call(VTOMI4(unldvp), &args, &res, unlcred, &doqueue, 0, &e);
4893         if (e.error) {
4894                 PURGE_ATTRCACHE4(unldvp);
4895                 resp = NULL;
4896         } else if (res.status) {
4897                 e.error = geterrno4(res.status);
4898                 PURGE_ATTRCACHE4(unldvp);
4899                 /*
4900                  * This code is inactive right now
4901                  * but if made active there should
```

```
4902                  * be a nfs4_end_op() call before
4903                  * nfs4_purge_stale_fh to avoid start_op()
4904                  * deadlock. See BugId: 4948726
4905                  */
4906                 nfs4_purge_stale_fh(error, unldvp, cr);
4907         } else {
4908                 nfs_resop4 *resop;
4909                 REMOVE4res *rm_res;

4911                 resop = &res.array[1];
4912                 rm_res = &resop->nfs_resop4_u.opremove;
4913                 /*
4914                  * Update directory cache attribute,
4915                  * readdir and dnlc caches.
4916                  */
4917                 nfs4_update_dircaches(&rm_res->cinfo, unldvp, NULL, NULL, NULL);
4918         }
4919 #else
4920         rfs4call(VTOMI4(unldvp), &args, &res, unlcred, &doqueue, 0, &e);

4922         PURGE_ATTRCACHE4(unldvp);
4923 #endif

4925         if (nfs4_needs_recovery(&e, FALSE, unldvp->v_vfsp)) {
4926                 if (nfs4_start_recovery(&e, VTOMI4(unldvp), unldvp, NULL,
4927                     NULL, NULL, OP_REMOVE, NULL, NULL, NULL) == FALSE) {
4928                         if (!e.error)
4929                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4930                                     (caddr_t)&res);
4931                         nfs4_end_op(VTOMI4(unldvp), unldvp, NULL,
4932                             &recov_state, TRUE);
4933                         goto recov_retry_remove;
4934                 }
4935         }
4936         nfs4_end_op(VTOMI4(unldvp), unldvp, NULL, &recov_state, FALSE);

4938         /*
4939          * Release stuff held for the remove
4940          */
4941         VN_RELE(unldvp);
4942         if (!e.error && resp)
4943                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

4945         kmem_free(unlname, MAXNAMELEN);
4946         crfree(unlcred);
4947         goto redo;
4948 }

4950 /*
4951  * Remote file system operations having to do with directory manipulation.
4952  */
4953 /* ARGSUSED3 */
4954 int
4955 nfs4_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
4956     int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
4957     int *direntflags, pathname_t *realpnp)
4958 {
4959         int error;
4960         vnode_t *vp, *avp = NULL;
4961         rnode4_t *drp;

4963         *vpp = NULL;
4964         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
4965                 return (EPERM);
4966         /*
4967          * if LOOKUP_XATTR, must replace dvp (object) with
```



```
4968          * object's attrdir before continuing with lookup
4969          */
4970         if (flags & LOOKUP_XATTR) {
4971                 error = nfs4lookup_xattr(dvp, nm, &avp, flags, cr);
4972                 if (error)
4973                         return (error);

4975                 dvp = avp;

4977                 /*
4978                  * If lookup is for "", just return dvp now.  The attrdir
4979                  * has already been activated (from nfs4lookup_xattr), and
4980                  * the caller will RELE the original dvp -- not
4981                  * the attrdir.  So, set vpp and return.
4982                  * Currently, when the LOOKUP_XATTR flag is
4983                  * passed to VOP_LOOKUP, the name is always empty, and
4984                  * shortcircuiting here avoids 3 unneeded lock/unlock
4985                  * pairs.
4986                  *
4987                  * If a non-empty name was provided, then it is the
4988                  * attribute name, and it will be looked up below.
4989                  */
4990                 if (*nm == '\0') {
4991                         *vpp = dvp;
4992                         return (0);
4993                 }

4995                 /*
4996                  * The vfs layer never sends a name when asking for the
4997                  * attrdir, so we should never get here (unless of course
4998                  * name is passed at some time in future -- at which time
4999                  * we'll blow up here).
5000                  */
5001                 ASSERT(0);
5002         }

5004         drp = VTOR4(dvp);
5005         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp)))
5006                 return (EINTR);

5008         error = nfs4lookup(dvp, nm, vpp, cr, 0);
5009         nfs_rw_exit(&drp->r_rwlock);

5011         /*
5012          * If vnode is a device, create special vnode.
5013          */
5014         if (!error && ISVDEV((*vpp)->v_type)) {
5015                 vp = *vpp;
5016                 *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
5017                 VN_RELE(vp);
5018         }

5020         return (error);
5021 }

5023 /* ARGSUSED */
5024 static int
5025 nfs4lookup_xattr(vnode_t *dvp, char *nm, vnode_t **vpp, int flags, cred_t *cr)
5026 {
5027         int error;
5028         rnode4_t *drp;
5029         int cflag = ((flags & CREATE_XATTR_DIR) != 0);
5030         mntinfo4_t *mi;

5032         mi = VTOMI4(dvp);
5033         if (!(mi->mi_vfsp->vfs_flag & VFS_XATTR) &&
```

```
5034             !vfs_has_feature(mi->mi_vfsp, VFSFT_SYSATTR_VIEWS))
5035                 return (EINVAL);
5036
5037         drp = VTOR4(dvp);
5038         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp)))
5039                 return (EINTR);
5040
5041         mutex_enter(&drp->r_statelock);
5042         /*
5043          * If the server doesn't support xattrs just return EINVAL
5044          */
5045         if (drp->r_xattr_dir == NFS4_XATTR_DIR_NOTSUPP) {
5046                 mutex_exit(&drp->r_statelock);
5047                 nfs_rw_exit(&drp->r_rwlock);
5048                 return (EINVAL);
5049         }
5050
5051         /*
5052          * If there is a cached xattr directory entry,
5053          * use it as long as the attributes are valid. If the
5054          * attributes are not valid, take the simple approach and
5055          * free the cached value and re-fetch a new value.
5056          *
5057          * We don't negative entry cache for now, if we did we
5058          * would need to check if the file has changed on every
5059          * lookup. But xattrs don't exist very often and failing
5060          * an openattr is not much more expensive than and NVERIFY or GETATTR
5061          * so do an openattr over the wire for now.
5062          */
5063         if (drp->r_xattr_dir != NULL) {
5064                 if (ATTRCACHE4_VALID(dvp)) {
5065                         VN_HOLD(drp->r_xattr_dir);
5066                         *vpp = drp->r_xattr_dir;
5067                         mutex_exit(&drp->r_statelock);
5068                         nfs_rw_exit(&drp->r_rwlock);
5069                         return (0);
5070                 }
5071                 VN_RELE(drp->r_xattr_dir);
5072                 drp->r_xattr_dir = NULL;
5073         }
5074         mutex_exit(&drp->r_statelock);
5075
5076         error = nfs4openattr(dvp, vpp, cflag, cr);
5077
5078         nfs_rw_exit(&drp->r_rwlock);
5079
5080         return (error);
5081 }
5082
5083 static int
5084 nfs4lookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr, int skipdnlc)
5085 {
5086         int error;
5087         rnode4_t *drp;
5088
5089         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);
5090
5091         /*
5092          * If lookup is for "", just return dvp.  Don't need
5093          * to send it over the wire, look it up in the dnlc,
5094          * or perform any access checks.
5095          */
5096         if (*nm == '\0') {
5097                 VN_HOLD(dvp);
5098                 *vpp = dvp;
5099                 return (0);
```



```
5100         }
5101
5102         /*
5103          * Can't do lookups in non-directories.
5104          */
5105         if (dvp->v_type != VDIR)
5106                 return (ENOTDIR);
5107
5108         /*
5109          * If lookup is for ".", just return dvp.  Don't need
5110          * to send it over the wire or look it up in the dnlc,
5111          * just need to check access.
5112          */
5113         if (nm[0] == '.' && nm[1] == '\0') {
5114                 error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5115                 if (error)
5116                         return (error);
5117                 VN_HOLD(dvp);
5118                 *vpp = dvp;
5119                 return (0);
5120         }
5121
5122         drp = VTOR4(dvp);
5123         if (!(drp->r_flags & R4LOOKUP)) {
5124                 mutex_enter(&drp->r_statelock);
5125                 drp->r_flags |= R4LOOKUP;
5126                 mutex_exit(&drp->r_statelock);
5127         }
5128
5129         *vpp = NULL;
5130         /*
5131          * Lookup this name in the DNLC.  If there is no entry
5132          * lookup over the wire.
5133          */
5134         if (!skipdnlc)
5135                 *vpp = dnlc_lookup(dvp, nm);
5136         if (*vpp == NULL) {
5137                 /*
5138                  * We need to go over the wire to lookup the name.
5139                  */
5140                 return (nfs4lookupnew_otw(dvp, nm, vpp, cr));
5141         }
5142
5143         /*
5144          * We hit on the dnlc
5145          */
5146         if (*vpp != DNLC_NO_VNODE ||
5147             (dvp->v_vfsp->vfs_flag & VFS_RDONLY)) {
5148                 /*
5149                  * But our attrs may not be valid.
5150                  */
5151                 if (ATTRCACHE4_VALID(dvp)) {
5152                         error = nfs4_waitfor_purge_complete(dvp);
5153                         if (error) {
5154                                 VN_RELE(*vpp);
5155                                 *vpp = NULL;
5156                                 return (error);
5157                         }
5158
5159                         /*
5160                          * If after the purge completes, check to make sure
5161                          * our attrs are still valid.
5162                          */
5163                         if (ATTRCACHE4_VALID(dvp)) {
5164                                 /*
5165                                  * If we waited for a purge we may have
```

```
5166                                 * lost our vnode so look it up again.
5167                                 */
5168                                VN_RELE(*vpp);
5169                                *vpp = dnlc_lookup(dvp, nm);
5170                                if (*vpp == NULL)
5171                                        return (nfs4lookupnew_otw(dvp,
5172                                            nm, vpp, cr));
5173
5174                                /*
5175                                 * The access cache should almost always hit
5176                                 */
5177                                error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5178
5179                                if (error) {
5180                                        VN_RELE(*vpp);
5181                                        *vpp = NULL;
5182                                        return (error);
5183                                }
5184                                if (*vpp == DNLC_NO_VNODE) {
5185                                        VN_RELE(*vpp);
5186                                        *vpp = NULL;
5187                                        return (ENOENT);
5188                                }
5189                                return (0);
5190                        }
5191                }
5192        }
5193
5194        ASSERT(*vpp != NULL);
5195
5196        /*
5197         * We may have gotten here we have one of the following cases:
5198         *      1) vpp != DNLC_NO_VNODE, our attrs have timed out so we
5199         *              need to validate them.
5200         *      2) vpp == DNLC_NO_VNODE, a negative entry that we always
5201         *              must validate.
5202         *
5203         * Go to the server and check if the directory has changed, if
5204         * it hasn't we are done and can use the dnlc entry.
5205         */
5206        return (nfs4lookupvalidate_otw(dvp, nm, vpp, cr));
5207 }
5208
5209 /*
5210  * Go to the server and check if the directory has changed, if
5211  * it hasn't we are done and can use the dnlc entry.  If it
5212  * has changed we get a new copy of its attributes and check
5213  * the access for VEXEC, then relookup the filename and
5214  * get its filehandle and attributes.
5215  *
5216  * PUTFH dfh NVERIFY GETATTR ACCESS LOOKUP GETFH GETATTR
5217  *      if the NVERIFY failed we must
5218  *              purge the caches
5219  *              cache new attributes (will set r_time_attr_inval)
5220  *              cache new access
5221  *              recheck VEXEC access
5222  *              add name to dnlc, possibly negative
5223  *              if LOOKUP succeeded
5224  *                      cache new attributes
5225  *      else
5226  *              set a new r_time_attr_inval for dvp
5227  *              check to make sure we have access
5228  *
5229  * The vpp returned is the vnode passed in if the directory is valid,
5230  * a new vnode if successful lookup, or NULL on error.
5231  */
```



```
5232 static int
5233 nfs4lookupvalidate_otw(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr)
5234 {
5235         COMPOUND4args_clnt args;
5236         COMPOUND4res_clnt res;
5237         fattr4 *ver_fattr;
5238         fattr4_change dchange;
5239         int32_t *ptr;
5240         int argoplist_size  = 7 * sizeof (nfs_argop4);
5241         nfs_argop4 *argop;
5242         int doqueue;
5243         mntinfo4_t *mi;
5244         nfs4_recov_state_t recov_state;
5245         hrtime_t t;
5246         int isdotdot;
5247         vnode_t *nvp;
5248         nfs_fh4 *fhp;
5249         nfs4_sharedfh_t *sfhp;
5250         nfs4_access_type_t cacc;
5251         rnode4_t *nrp;
5252         rnode4_t *drp = VTOR4(dvp);
5253         nfs4_ga_res_t *garp = NULL;
5254         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
5255
5256         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);
5257         ASSERT(nm != NULL);
5258         ASSERT(nm[0] != '\0');
5259         ASSERT(dvp->v_type == VDIR);
5260         ASSERT(nm[0] != '.' || nm[1] != '\0');
5261         ASSERT(*vpp != NULL);
5262
5263         if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0') {
5264                 isdotdot = 1;
5265                 args.ctag = TAG_LOOKUP_VPARENT;
5266         } else {
5267                 /*
5268                  * If dvp were a stub, it should have triggered and caused
5269                  * a mount for us to get this far.
5270                  */
5271                 ASSERT(!RP_ISSTUB(VTOR4(dvp)));
5272
5273                 isdotdot = 0;
5274                 args.ctag = TAG_LOOKUP_VALID;
5275         }
5276
5277         mi = VTOMI4(dvp);
5278         recov_state.rs_flags = 0;
5279         recov_state.rs_num_retry_despite_err = 0;
5280
5281         nvp = NULL;
5282
5283         /* Save the original mount point security information */
5284         (void) save_mnt_secinfo(mi->mi_curr_serv);
5285
5286 recov_retry:
5287         e.error = nfs4_start_fop(mi, dvp, NULL, OH_LOOKUP,
5288             &recov_state, NULL);
5289         if (e.error) {
5290                 (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5291                 VN_RELE(*vpp);
5292                 *vpp = NULL;
5293                 return (e.error);
5294         }
5295
5296         argop = kmem_alloc(argoplist_size, KM_SLEEP);
```

```
5298         /* PUTFH dfh NVERIFY GETATTR ACCESS LOOKUP GETFH GETATTR */
5299         args.array_len = 7;
5300         args.array = argop;

5302         /* 0. putfh file */
5303         argop[0].argop = OP_CPUTFH;
5304         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(dvp)->r_fh;

5306         /* 1. nverify the change info */
5307         argop[1].argop = OP_NVERIFY;
5308         ver_fattr = &argop[1].nfs_argop4_u.opnverify.obj_attributes;
5309         ver_fattr->attrmask = FATTR4_CHANGE_MASK;
5310         ver_fattr->attrlist4 = (char *)&dchange;
5311         ptr = (int32_t *)&dchange;
5312         IXDR_PUT_HYPER(ptr, VTOR4(dvp)->r_change);
5313         ver_fattr->attrlist4_len = sizeof (fattr4_change);

5315         /* 2. getattr directory */
5316         argop[2].argop = OP_GETATTR;
5317         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5318         argop[2].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5320         /* 3. access directory */
5321         argop[3].argop = OP_ACCESS;
5322         argop[3].nfs_argop4_u.opaccess.access = ACCESS4_READ | ACCESS4_DELETE |
5323             ACCESS4_MODIFY | ACCESS4_EXTEND | ACCESS4_LOOKUP;

5325         /* 4. lookup name */
5326         if (isdotdot) {
5327                 argop[4].argop = OP_LOOKUPP;
5328         } else {
5329                 argop[4].argop = OP_CLOOKUP;
5330                 argop[4].nfs_argop4_u.opclookup.cname = nm;
5331         }

5333         /* 5. resulting file handle */
5334         argop[5].argop = OP_GETFH;

5336         /* 6. resulting file attributes */
5337         argop[6].argop = OP_GETATTR;
5338         argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5339         argop[6].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5341         doqueue = 1;
5342         t = gethrtime();

5344         rfs4call(VTOMI4(dvp), &args, &res, cr, &doqueue, 0, &e);

5346         if (!isdotdot && res.status == NFS4ERR_MOVED) {
5347                 e.error = nfs4_setup_referral(dvp, nm, vpp, cr);
5348                 if (e.error != 0 && *vpp != NULL)
5349                         VN_RELE(*vpp);
5350                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5351                     &recov_state, FALSE);
5352                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5353                 kmem_free(argop, argoplist_size);
5354                 return (e.error);
5355         }

5357         if (nfs4_needs_recovery(&e, FALSE, dvp->v_vfsp)) {
5358                 /*
5359                  * For WRONGSEC of a non-dotdot case, send secinfo directly
5360                  * from this thread, do not go thru the recovery thread since
5361                  * we need the nm information.
5362                  *
5363                  * Not doing dotdot case because there is no specification
```



```
5364                  * for (PUTFH, SECINFO "..") yet.
5365                  */
5366                 if (!isdotdot && res.status == NFS4ERR_WRONGSEC) {
5367                         if ((e.error = nfs4_secinfo_vnode_otw(dvp, nm, cr)))
5368                                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5369                                     &recov_state, FALSE);
5370                         else
5371                                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5372                                     &recov_state, TRUE);
5373                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5374                         kmem_free(argop, argoplist_size);
5375                         if (!e.error)
5376                                 goto recov_retry;
5377                         (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5378                         VN_RELE(*vpp);
5379                         *vpp = NULL;
5380                         return (e.error);
5381                 }

5383                 if (nfs4_start_recovery(&e, mi, dvp, NULL, NULL, NULL,
5384                     OP_LOOKUP, NULL, NULL, NULL) == FALSE) {
5385                         nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5386                             &recov_state, TRUE);

5388                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5389                         kmem_free(argop, argoplist_size);
5390                         goto recov_retry;
5391                 }
5392         }

5394         nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP, &recov_state, FALSE);

5396         if (e.error || res.array_len == 0) {
5397                 /*
5398                  * If e.error isn't set, then reply has no ops (or we couldn't
5399                  * be here).  The only legal way to reply without an op array
5400                  * is via NFS4ERR_MINOR_VERS_MISMATCH.  An ops array should
5401                  * be in the reply for all other status values.
5402                  *
5403                  * For valid replies without an ops array, return ENOTSUP
5404                  * (geterrno4 xlation of VERS_MISMATCH).  For illegal replies,
5405                  * return EIO -- don't trust status.
5406                  */
5407                 if (e.error == 0)
5408                         e.error = (res.status == NFS4ERR_MINOR_VERS_MISMATCH) ?
5409                             ENOTSUP : EIO;
5410                 VN_RELE(*vpp);
5411                 *vpp = NULL;
5412                 kmem_free(argop, argoplist_size);
5413                 (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5414                 return (e.error);
5415         }

5417         if (res.status != NFS4ERR_SAME) {
5418                 e.error = geterrno4(res.status);

5420                 /*
5421                  * The NVERIFY "failed" so the directory has changed
5422                  * First make sure PUTFH succeeded and NVERIFY "failed"
5423                  * cleanly.
5424                  */
5425                 if ((res.array[0].nfs_resop4_u.opputfh.status != NFS4_OK) ||
5426                     (res.array[1].nfs_resop4_u.opnverify.status != NFS4_OK)) {
5427                         nfs4_purge_stale_fh(e.error, dvp, cr);
5428                         VN_RELE(*vpp);
5429                         *vpp = NULL;
```

```
5430                    goto exit;
5431            }
5433            /*
5434             * We know the NVERIFY "failed" so we must:
5435             *      purge the caches (access and indirectly dnlc if needed)
5436             */
5437            nfs4_purge_caches(dvp, NFS4_NOPURGE_DNLC, cr, TRUE);
5439            if (res.array[2].nfs_resop4_u.opgetattr.status != NFS4_OK) {
5440                    nfs4_purge_stale_fh(e.error, dvp, cr);
5441                    VN_RELE(*vpp);
5442                    *vpp = NULL;
5443                    goto exit;
5444            }
5446            /*
5447             * Install new cached attributes for the directory
5448             */
5449            nfs4_attr_cache(dvp,
5450                &res.array[2].nfs_resop4_u.opgetattr.ga_res,
5451                t, cr, FALSE, NULL);
5453            if (res.array[3].nfs_resop4_u.opaccess.status != NFS4_OK) {
5454                    nfs4_purge_stale_fh(e.error, dvp, cr);
5455                    VN_RELE(*vpp);
5456                    *vpp = NULL;
5457                    e.error = geterrno4(res.status);
5458                    goto exit;
5459            }
5461            /*
5462             * Now we know the directory is valid,
5463             * cache new directory access
5464             */
5465            nfs4_access_cache(drp,
5466                args.array[3].nfs_argop4_u.opaccess.access,
5467                res.array[3].nfs_resop4_u.opaccess.access, cr);
5469            /*
5470             * recheck VEXEC access
5471             */
5472            cacc = nfs4_access_check(drp, ACCESS4_LOOKUP, cr);
5473            if (cacc != NFS4_ACCESS_ALLOWED) {
5474                    /*
5475                     * Directory permissions might have been revoked
5476                     */
5477                    if (cacc == NFS4_ACCESS_DENIED) {
5478                            e.error = EACCES;
5479                            VN_RELE(*vpp);
5480                            *vpp = NULL;
5481                            goto exit;
5482                    }
5484                    /*
5485                     * Somehow we must not have asked for enough
5486                     * so try a singleton ACCESS, should never happen.
5487                     */
5488                    e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5489                    if (e.error) {
5490                            VN_RELE(*vpp);
5491                            *vpp = NULL;
5492                            goto exit;
5493                    }
5494            }
```



```
5496            e.error = geterrno4(res.status);
5497            if (res.array[4].nfs_resop4_u.oplookup.status != NFS4_OK) {
5498                    /*
5499                     * The lookup failed, probably no entry
5500                     */
5501                    if (e.error == ENOENT && nfs4_lookup_neg_cache) {
5502                            dnlc_update(dvp, nm, DNLC_NO_VNODE);
5503                    } else {
5504                            /*
5505                             * Might be some other error, so remove
5506                             * the dnlc entry to make sure we start all
5507                             * over again, next time.
5508                             */
5509                            dnlc_remove(dvp, nm);
5510                    }
5511                    VN_RELE(*vpp);
5512                    *vpp = NULL;
5513                    goto exit;
5514            }
5516            if (res.array[5].nfs_resop4_u.opgetfh.status != NFS4_OK) {
5517                    /*
5518                     * The file exists but we can't get its fh for
5519                     * some unknown reason.  Remove it from the dnlc
5520                     * and error out to be safe.
5521                     */
5522                    dnlc_remove(dvp, nm);
5523                    VN_RELE(*vpp);
5524                    *vpp = NULL;
5525                    goto exit;
5526            }
5527            fhp = &res.array[5].nfs_resop4_u.opgetfh.object;
5528            if (fhp->nfs_fh4_len == 0) {
5529                    /*
5530                     * The file exists but a bogus fh
5531                     * some unknown reason.  Remove it from the dnlc
5532                     * and error out to be safe.
5533                     */
5534                    e.error = ENOENT;
5535                    dnlc_remove(dvp, nm);
5536                    VN_RELE(*vpp);
5537                    *vpp = NULL;
5538                    goto exit;
5539            }
5540            sfhp = sfh4_get(fhp, mi);
5542            if (res.array[6].nfs_resop4_u.opgetattr.status == NFS4_OK)
5543                    garp = &res.array[6].nfs_resop4_u.opgetattr.ga_res;
5545            /*
5546             * Make the new rnode
5547             */
5548            if (isdotdot) {
5549                    e.error = nfs4_make_dotdot(sfhp, t, dvp, cr, &nvp, 1);
5550                    if (e.error) {
5551                            sfh4_rele(&sfhp);
5552                            VN_RELE(*vpp);
5553                            *vpp = NULL;
5554                            goto exit;
5555                    }
5556                    /*
5557                     * XXX if nfs4_make_dotdot uses an existing rnode
5558                     * XXX it doesn't update the attributes.
5559                     * XXX for now just save them again to save an OTW
5560                     */
5561                    nfs4_attr_cache(nvp, garp, t, cr, FALSE, NULL);
```

```
5562                } else {
5563                        nvp = makenfs4node(sfhp, garp, dvp->v_vfsp, t, cr,
5564                            dvp, fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
5565                        /*
5566                         * If v_type == VNON, then garp was NULL because
5567                         * the last op in the compound failed and makenfs4node
5568                         * could not find the vnode for sfhp. It created
5569                         * a new vnode, so we have nothing to purge here.
5570                         */
5571                        if (nvp->v_type == VNON) {
5572                                vattr_t vattr;

5574                                vattr.va_mask = AT_TYPE;
5575                                /*
5576                                 * N.B. We've already called nfs4_end_fop above.
5577                                 */
5578                                e.error = nfs4getattr(nvp, &vattr, cr);
5579                                if (e.error) {
5580                                        sfh4_rele(&sfhp);
5581                                        VN_RELE(*vpp);
5582                                        *vpp = NULL;
5583                                        VN_RELE(nvp);
5584                                        goto exit;
5585                                }
5586                                nvp->v_type = vattr.va_type;
5587                        }
5588                }
5589                sfh4_rele(&sfhp);

5591                nrp = VTOR4(nvp);
5592                mutex_enter(&nrp->r_statev4_lock);
5593                if (!nrp->created_v4) {
5594                        mutex_exit(&nrp->r_statev4_lock);
5595                        dnlc_update(dvp, nm, nvp);
5596                } else
5597                        mutex_exit(&nrp->r_statev4_lock);

5599                VN_RELE(*vpp);
5600                *vpp = nvp;
5601        } else {
5602                hrtime_t now;
5603                hrtime_t delta = 0;

5605                e.error = 0;

5607                /*
5608                 * Because the NVERIFY "succeeded" we know that the
5609                 * directory attributes are still valid
5610                 * so update r_time_attr_inval
5611                 */
5612                now = gethrtime();
5613                mutex_enter(&drp->r_statelock);
5614                if (!(mi->mi_flags & MI4_NOAC) && !(dvp->v_flag & VNOCACHE)) {
5615                        delta = now - drp->r_time_attr_saved;
5616                        if (delta < mi->mi_acdirmin)
5617                                delta = mi->mi_acdirmin;
5618                        else if (delta > mi->mi_acdirmax)
5619                                delta = mi->mi_acdirmax;
5620                }
5621                drp->r_time_attr_inval = now + delta;
5622                mutex_exit(&drp->r_statelock);
5623                dnlc_update(dvp, nm, *vpp);

5625                /*
5626                 * Even though we have a valid directory attr cache
5627                 * and dnlc entry, we may not have access.
```



```
5628                 * This should almost always hit the cache.
5629                 */
5630                e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5631                if (e.error) {
5632                        VN_RELE(*vpp);
5633                        *vpp = NULL;
5634                }

5636                if (*vpp == DNLC_NO_VNODE) {
5637                        VN_RELE(*vpp);
5638                        *vpp = NULL;
5639                        e.error = ENOENT;
5640                }
5641        }

5643 exit:
5644        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5645        kmem_free(argop, argoplist_size);
5646        (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5647        return (e.error);
5648 }

5650 /*
5651  * We need to go over the wire to lookup the name, but
5652  * while we are there verify the directory has not
5653  * changed but if it has, get new attributes and check access
5654  *
5655  * PUTFH dfh SAVEFH LOOKUP nm GETFH GETATTR RESTOREFH
5656  *                                      NVERIFY GETATTR ACCESS
5657  *
5658  * With the results:
5659  *      if the NVERIFY failed we must purge the caches, add new attributes,
5660  *              and cache new access.
5661  *      set a new r_time_attr_inval
5662  *      add name to dnlc, possibly negative
5663  *      if LOOKUP succeeded
5664  *              cache new attributes
5665  */
5666 static int
5667 nfs4lookupnew_otw(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr)
5668 {
5669        COMPOUND4args_clnt args;
5670        COMPOUND4res_clnt res;
5671        fattr4 *ver_fattr;
5672        fattr4_change dchange;
5673        int32_t *ptr;
5674        nfs4_ga_res_t *garp = NULL;
5675        int argoplist_size  = 9 * sizeof (nfs_argop4);
5676        nfs_argop4 *argop;
5677        int doqueue;
5678        mntinfo4_t *mi;
5679        nfs4_recov_state_t recov_state;
5680        hrtime_t t;
5681        int isdotdot;
5682        vnode_t *nvp;
5683        nfs_fh4 *fhp;
5684        nfs4_sharedfh_t *sfhp;
5685        nfs4_access_type_t cacc;
5686        rnode4_t *nrp;
5687        rnode4_t *drp = VTOR4(dvp);
5688        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

5690        ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);
5691        ASSERT(nm != NULL);
5692        ASSERT(nm[0] != '\0');
5693        ASSERT(dvp->v_type == VDIR);
```

```
5694         ASSERT(nm[0] != '.' || nm[1] != '\0');
5695         ASSERT(*vpp == NULL);

5697         if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0') {
5698                 isdotdot = 1;
5699                 args.ctag = TAG_LOOKUP_PARENT;
5700         } else {
5701                 /*
5702                  * If dvp were a stub, it should have triggered and caused
5703                  * a mount for us to get this far.
5704                  */
5705                 ASSERT(!RP_ISSTUB(VTOR4(dvp)));

5707                 isdotdot = 0;
5708                 args.ctag = TAG_LOOKUP;
5709         }

5711         mi = VTOMI4(dvp);
5712         recov_state.rs_flags = 0;
5713         recov_state.rs_num_retry_despite_err = 0;

5715         nvp = NULL;

5717         /* Save the original mount point security information */
5718         (void) save_mnt_secinfo(mi->mi_curr_serv);

5720 recov_retry:
5721         e.error = nfs4_start_fop(mi, dvp, NULL, OH_LOOKUP,
5722             &recov_state, NULL);
5723         if (e.error) {
5724                 (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5725                 return (e.error);
5726         }

5728         argop = kmem_alloc(argoplist_size, KM_SLEEP);

5730         /* PUTFH SAVEFH LOOKUP GETFH GETATTR RESTOREFH NVERIFY GETATTR ACCESS */
5731         args.array_len = 9;
5732         args.array = argop;

5734         /* 0. putfh file */
5735         argop[0].argop = OP_CPUTFH;
5736         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(dvp)->r_fh;

5738         /* 1. savefh for the nverify */
5739         argop[1].argop = OP_SAVEFH;

5741         /* 2. lookup name */
5742         if (isdotdot) {
5743                 argop[2].argop = OP_LOOKUPP;
5744         } else {
5745                 argop[2].argop = OP_CLOOKUP;
5746                 argop[2].nfs_argop4_u.opclookup.cname = nm;
5747         }

5749         /* 3. resulting file handle */
5750         argop[3].argop = OP_GETFH;

5752         /* 4. resulting file attributes */
5753         argop[4].argop = OP_GETATTR;
5754         argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5755         argop[4].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5757         /* 5. restorefh back the directory for the nverify */
5758         argop[5].argop = OP_RESTOREFH;
```



```
5760         /* 6. nverify the change info */
5761         argop[6].argop = OP_NVERIFY;
5762         ver_fattr = &argop[6].nfs_argop4_u.opnverify.obj_attributes;
5763         ver_fattr->attrmask = FATTR4_CHANGE_MASK;
5764         ver_fattr->attrlist4 = (char *)&dchange;
5765         ptr = (int32_t *)&dchange;
5766         IXDR_PUT_HYPER(ptr, VTOR4(dvp)->r_change);
5767         ver_fattr->attrlist4_len = sizeof (fattr4_change);

5769         /* 7. getattr directory */
5770         argop[7].argop = OP_GETATTR;
5771         argop[7].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5772         argop[7].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5774         /* 8. access directory */
5775         argop[8].argop = OP_ACCESS;
5776         argop[8].nfs_argop4_u.opaccess.access = ACCESS4_READ | ACCESS4_DELETE |
5777             ACCESS4_MODIFY | ACCESS4_EXTEND | ACCESS4_LOOKUP;

5779         doqueue = 1;
5780         t = gethrtime();

5782         rfs4call(VTOMI4(dvp), &args, &res, cr, &doqueue, 0, &e);

5784         if (!isdotdot && res.status == NFS4ERR_MOVED) {
5785                 e.error = nfs4_setup_referral(dvp, nm, vpp, cr);
5786                 if (e.error != 0 && *vpp != NULL)
5787                         VN_RELE(*vpp);
5788                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5789                     &recov_state, FALSE);
5790                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5791                 kmem_free(argop, argoplist_size);
5792                 return (e.error);
5793         }

5795         if (nfs4_needs_recovery(&e, FALSE, dvp->v_vfsp)) {
5796                 /*
5797                  * For WRONGSEC of a non-dotdot case, send secinfo directly
5798                  * from this thread, do not go thru the recovery thread since
5799                  * we need the nm information.
5800                  *
5801                  * Not doing dotdot case because there is no specification
5802                  * for (PUTFH, SECINFO "..") yet.
5803                  */
5804                 if (!isdotdot && res.status == NFS4ERR_WRONGSEC) {
5805                         if ((e.error = nfs4_secinfo_vnode_otw(dvp, nm, cr)))
5806                                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5807                                     &recov_state, FALSE);
5808                         else
5809                                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5810                                     &recov_state, TRUE);
5811                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5812                         kmem_free(argop, argoplist_size);
5813                         if (!e.error)
5814                                 goto recov_retry;
5815                         (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5816                         return (e.error);
5817                 }

5819                 if (nfs4_start_recovery(&e, mi, dvp, NULL, NULL, NULL,
5820                     OP_LOOKUP, NULL, NULL, NULL) == FALSE) {
5821                         nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5822                             &recov_state, TRUE);

5824                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5825                         kmem_free(argop, argoplist_size);
```

```
5826                        goto recov_retry;
5827                }
5828        }

5830        nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP, &recov_state, FALSE);

5832        if (e.error || res.array_len == 0) {
5833                /*
5834                 * If e.error isn't set, then reply has no ops (or we couldn't
5835                 * be here).  The only legal way to reply without an op array
5836                 * is via NFS4ERR_MINOR_VERS_MISMATCH.  An ops array should
5837                 * be in the reply for all other status values.
5838                 *
5839                 * For valid replies without an ops array, return ENOTSUP
5840                 * (geterrno4 xlation of VERS_MISMATCH).  For illegal replies,
5841                 * return EIO -- don't trust status.
5842                 */
5843                if (e.error == 0)
5844                        e.error = (res.status == NFS4ERR_MINOR_VERS_MISMATCH) ?
5845                            ENOTSUP : EIO;

5847                kmem_free(argop, argoplist_size);
5848                (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5849                return (e.error);
5850        }

5852        e.error = geterrno4(res.status);

5854        /*
5855         * The PUTFH and SAVEFH may have failed.
5856         */
5857        if ((res.array[0].nfs_resop4_u.opputfh.status != NFS4_OK) ||
5858            (res.array[1].nfs_resop4_u.opsavefh.status != NFS4_OK)) {
5859                nfs4_purge_stale_fh(e.error, dvp, cr);
5860                goto exit;
5861        }

5863        /*
5864         * Check if the file exists, if it does delay entering
5865         * into the dnlc until after we update the directory
5866         * attributes so we don't cause it to get purged immediately.
5867         */
5868        if (res.array[2].nfs_resop4_u.oplookup.status != NFS4_OK) {
5869                /*
5870                 * The lookup failed, probably no entry
5871                 */
5872                if (e.error == ENOENT && nfs4_lookup_neg_cache)
5873                        dnlc_update(dvp, nm, DNLC_NO_VNODE);
5874                goto exit;
5875        }

5877        if (res.array[3].nfs_resop4_u.opgetfh.status != NFS4_OK) {
5878                /*
5879                 * The file exists but we can't get its fh for
5880                 * some unknown reason. Error out to be safe.
5881                 */
5882                goto exit;
5883        }

5885        fhp = &res.array[3].nfs_resop4_u.opgetfh.object;
5886        if (fhp->nfs_fh4_len == 0) {
5887                /*
5888                 * The file exists but a bogus fh
5889                 * some unknown reason.  Error out to be safe.
5890                 */
5891                e.error = EIO;
```



```
5892                goto exit;
5893        }
5894        sfhp = sfh4_get(fhp, mi);

5896        if (res.array[4].nfs_resop4_u.opgetattr.status != NFS4_OK) {
5897                sfh4_rele(&sfhp);
5898                goto exit;
5899        }
5900        garp = &res.array[4].nfs_resop4_u.opgetattr.ga_res;

5902        /*
5903         * The RESTOREFH may have failed
5904         */
5905        if (res.array[5].nfs_resop4_u.oprestorefh.status != NFS4_OK) {
5906                sfh4_rele(&sfhp);
5907                e.error = EIO;
5908                goto exit;
5909        }

5911        if (res.array[6].nfs_resop4_u.opnverify.status != NFS4ERR_SAME) {
5912                /*
5913                 * First make sure the NVERIFY failed as we expected,
5914                 * if it didn't then be conservative and error out
5915                 * as we can't trust the directory.
5916                 */
5917                if (res.array[6].nfs_resop4_u.opnverify.status != NFS4_OK) {
5918                        sfh4_rele(&sfhp);
5919                        e.error = EIO;
5920                        goto exit;
5921                }

5923                /*
5924                 * We know the NVERIFY "failed" so the directory has changed,
5925                 * so we must:
5926                 *      purge the caches (access and indirectly dnlc if needed)
5927                 */
5928                nfs4_purge_caches(dvp, NFS4_NOPURGE_DNLC, cr, TRUE);

5930                if (res.array[7].nfs_resop4_u.opgetattr.status != NFS4_OK) {
5931                        sfh4_rele(&sfhp);
5932                        goto exit;
5933                }
5934                nfs4_attr_cache(dvp,
5935                    &res.array[7].nfs_resop4_u.opgetattr.ga_res,
5936                    t, cr, FALSE, NULL);

5938                if (res.array[8].nfs_resop4_u.opaccess.status != NFS4_OK) {
5939                        nfs4_purge_stale_fh(e.error, dvp, cr);
5940                        sfh4_rele(&sfhp);
5941                        e.error = geterrno4(res.status);
5942                        goto exit;
5943                }

5945                /*
5946                 * Now we know the directory is valid,
5947                 * cache new directory access
5948                 */
5949                nfs4_access_cache(drp,
5950                    args.array[8].nfs_argop4_u.opaccess.access,
5951                    res.array[8].nfs_resop4_u.opaccess.access, cr);

5953                /*
5954                 * recheck VEXEC access
5955                 */
5956                cacc = nfs4_access_check(drp, ACCESS4_LOOKUP, cr);
5957                if (cacc != NFS4_ACCESS_ALLOWED) {
```

```
5958                        /*
5959                         * Directory permissions might have been revoked
5960                         */
5961                        if (cacc == NFS4_ACCESS_DENIED) {
5962                                sfh4_rele(&sfhp);
5963                                e.error = EACCES;
5964                                goto exit;
5965                        }

5967                        /*
5968                         * Somehow we must not have asked for enough
5969                         * so try a singleton ACCESS should never happen
5970                         */
5971                        e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5972                        if (e.error) {
5973                                sfh4_rele(&sfhp);
5974                                goto exit;
5975                        }
5976                }

5978                e.error = geterrno4(res.status);
5979        } else {
5980                hrtime_t now;
5981                hrtime_t delta = 0;

5983                e.error = 0;

5985                /*
5986                 * Because the NVERIFY "succeeded" we know that the
5987                 * directory attributes are still valid
5988                 * so update r_time_attr_inval
5989                 */
5990                now = gethrtime();
5991                mutex_enter(&drp->r_statelock);
5992                if (!(mi->mi_flags & MI4_NOAC) && !(dvp->v_flag & VNOCACHE)) {
5993                        delta = now - drp->r_time_attr_saved;
5994                        if (delta < mi->mi_acdirmin)
5995                                delta = mi->mi_acdirmin;
5996                        else if (delta > mi->mi_acdirmax)
5997                                delta = mi->mi_acdirmax;
5998                }
5999                drp->r_time_attr_inval = now + delta;
6000                mutex_exit(&drp->r_statelock);

6002                /*
6003                 * Even though we have a valid directory attr cache,
6004                 * we may not have access.
6005                 * This should almost always hit the cache.
6006                 */
6007                e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
6008                if (e.error) {
6009                        sfh4_rele(&sfhp);
6010                        goto exit;
6011                }
6012        }

6014        /*
6015         * Now we have successfully completed the lookup, if the
6016         * directory has changed we now have the valid attributes.
6017         * We also know we have directory access.
6018         * Create the new rnode and insert it in the dnlc.
6019         */
6020        if (isdotdot) {
6021                e.error = nfs4_make_dotdot(sfhp, t, dvp, cr, &nvp, 1);
6022                if (e.error) {
6023                        sfh4_rele(&sfhp);
```



```
6024                            goto exit;
6025                    }
6026                    /*
6027                     * XXX if nfs4_make_dotdot uses an existing rnode
6028                     * XXX it doesn't update the attributes.
6029                     * XXX for now just save them again to save an OTW
6030                     */
6031                    nfs4_attr_cache(nvp, garp, t, cr, FALSE, NULL);
6032            } else {
6033                    nvp = makenfs4node(sfhp, garp, dvp->v_vfsp, t, cr,
6034                        dvp, fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
6035            }
6036            sfh4_rele(&sfhp);

6038            nrp = VTOR4(nvp);
6039            mutex_enter(&nrp->r_statev4_lock);
6040            if (!nrp->created_v4) {
6041                    mutex_exit(&nrp->r_statev4_lock);
6042                    dnlc_update(dvp, nm, nvp);
6043            } else
6044                    mutex_exit(&nrp->r_statev4_lock);

6046            *vpp = nvp;

6048    exit:
6049            (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6050            kmem_free(argop, argoplist_size);
6051            (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
6052            return (e.error);
6053    }

6055    #ifdef DEBUG
6056    void
6057    nfs4lookup_dump_compound(char *where, nfs_argop4 *argbase, int argcnt)
6058    {
6059            uint_t i, len;
6060            zoneid_t zoneid = getzoneid();
6061            char *s;

6063            zcmn_err(zoneid, CE_NOTE, "%s: dumping cmpd", where);
6064            for (i = 0; i < argcnt; i++) {
6065                    nfs_argop4 *op = &argbase[i];
6066                    switch (op->argop) {
6067                    case OP_CPUTFH:
6068                    case OP_PUTFH:
6069                            zcmn_err(zoneid, CE_NOTE, "\t op %d, putfh", i);
6070                            break;
6071                    case OP_PUTROOTFH:
6072                            zcmn_err(zoneid, CE_NOTE, "\t op %d, putrootfh", i);
6073                            break;
6074                    case OP_CLOOKUP:
6075                            s = op->nfs_argop4_u.opclookup.cname;
6076                            zcmn_err(zoneid, CE_NOTE, "\t op %d, lookup %s", i, s);
6077                            break;
6078                    case OP_LOOKUP:
6079                            s = utf8_to_str(&op->nfs_argop4_u.oplookup.objname,
6080                                &len, NULL);
6081                            zcmn_err(zoneid, CE_NOTE, "\t op %d, lookup %s", i, s);
6082                            kmem_free(s, len);
6083                            break;
6084                    case OP_LOOKUPP:
6085                            zcmn_err(zoneid, CE_NOTE, "\t op %d, lookupp ..", i);
6086                            break;
6087                    case OP_GETFH:
6088                            zcmn_err(zoneid, CE_NOTE, "\t op %d, getfh", i);
6089                            break;
```

```
6090                 case OP_GETATTR:
6091                         zcmn_err(zoneid, CE_NOTE, "\t op %d, getattr", i);
6092                         break;
6093                 case OP_OPENATTR:
6094                         zcmn_err(zoneid, CE_NOTE, "\t op %d, openattr", i);
6095                         break;
6096                 default:
6097                         zcmn_err(zoneid, CE_NOTE, "\t op %d, opcode %d", i,
6098                             op->argop);
6099                         break;
6100                 }
6101         }
6102 }
6103 #endif

6105 /*
6106  * nfs4lookup_setup - constructs a multi-lookup compound request.
6107  *
6108  * Given the path "nm1/nm2/.../nmn", the following compound requests
6109  * may be created:
6110  *
6111  * Note: Getfh is not be needed because filehandle attr is mandatory, but it
6112  * is faster, for now.
6113  *
6114  * l4_getattrs indicates the type of compound requested.
6115  *
6116  * LKP4_NO_ATTRIBUTE - no attributes (used by secinfo):
6117  *
6118  *      compound { Put*fh; Lookup {nm1}; Lookup {nm2}; ...  Lookup {nmn} }
6119  *
6120  *   total number of ops is n + 1.
6121  *
6122  * LKP4_LAST_NAMED_ATTR - multi-component path for a named
6123  *      attribute: create lookups plus one OPENATTR/GETFH/GETATTR
6124  *      before the last component, and only get attributes
6125  *      for the last component.  Note that the second-to-last
6126  *      pathname component is XATTR_RPATH, which does NOT go
6127  *      over-the-wire as a lookup.
6128  *
6129  *      compound { Put*fh; Lookup {nm1}; Lookup {nm2}; ... Lookup {nmn-2};
6130  *              Openattr; Getfh; Getattr; Lookup {nmn}; Getfh; Getattr }
6131  *
6132  *   and total number of ops is n + 5.
6133  *
6134  * LKP4_LAST_ATTRDIR - multi-component path for the hidden named
6135  *      attribute directory: create lookups plus an OPENATTR
6136  *      replacing the last lookup.  Note that the last pathname
6137  *      component is XATTR_RPATH, which does NOT go over-the-wire
6138  *      as a lookup.
6139  *
6140  *      compound { Put*fh; Lookup {nm1}; Lookup {nm2}; ... Getfh; Getattr;
6141  *              Openattr; Getfh; Getattr }
6142  *
6143  *   and total number of ops is n + 5.
6144  *
6145  * LKP4_ALL_ATTRIBUTES - create lookups and get attributes for intermediate
6146  *      nodes too.
6147  *
6148  *      compound { Put*fh; Lookup {nm1}; Getfh; Getattr;
6149  *              Lookup {nm2}; ...  Lookup {nmn}; Getfh; Getattr }
6150  *
6151  *   and total number of ops is 3*n + 1.
6152  *
6153  * All cases: returns the index in the arg array of the final LOOKUP op, or
6154  * -1 if no LOOKUPs were used.
6155  */
```



```
6156 int
6157 nfs4lookup_setup(char *nm, lookup4_param_t *lookupargp, int needgetfh)
6158 {
6159         enum lkp4_attr_setup l4_getattrs = lookupargp->l4_getattrs;
6160         nfs_argop4 *argbase, *argop;
6161         int arglen, argcnt;
6162         int n = 1;      /* number of components */
6163         int nga = 1;    /* number of Getattr's in request */
6164         char c = '\0', *s, *p;
6165         int lookup_idx = -1;
6166         int argoplist_size;

6168         /* set lookuparg response result to 0 */
6169         lookupargp->resp->status = NFS4_OK;

6171         /* skip leading "/" or "." e.g. ".//./" if there is */
6172         for (; ; nm++) {
6173                 if (*nm != '/' && *nm != '.')
6174                         break;

6176                 /* ".." is counted as 1 component */
6177                 if (*nm == '.' && *(nm + 1) != '/')
6178                         break;
6179         }

6181         /*
6182          * Find n = number of components - nm must be null terminated
6183          * Skip "." components.
6184          */
6185         if (*nm != '\0')
6186                 for (n = 1, s = nm; *s != '\0'; s++) {
6187                         if ((*s == '/') && (*(s + 1) != '/') &&
6188                             (*(s + 1) != '\0') &&
6189                             !(*(s + 1) == '.' && (*(s + 2) == '/' ||
6190                             *(s + 2) == '\0')))
6191                                 n++;
6192                 }
6193         else
6194                 n = 0;

6196         /*
6197          * nga is number of components that need Getfh+Getattr
6198          */
6199         switch (l4_getattrs) {
6200         case LKP4_NO_ATTRIBUTES:
6201                 nga = 0;
6202                 break;
6203         case LKP4_ALL_ATTRIBUTES:
6204                 nga = n;
6205                 /*
6206                  * Always have at least 1 getfh, getattr pair
6207                  */
6208                 if (nga == 0)
6209                         nga++;
6210                 break;
6211         case LKP4_LAST_ATTRDIR:
6212         case LKP4_LAST_NAMED_ATTR:
6213                 nga = n+1;
6214                 break;
6215         }

6217         /*
6218          * If change to use the filehandle attr instead of getfh
6219          * the following line can be deleted.
6220          */
6221         nga *= 2;
```

```
6223         /*
6224          * calculate number of ops in request as
6225          * header + trailer + lookups + getattrs
6226          */
6227         arglen = lookupargp->header_len + lookupargp->trailer_len + n + nga;

6229         argoplist_size = arglen * sizeof (nfs_argop4);
6230         argop = argbase = kmem_alloc(argoplist_size, KM_SLEEP);
6231         lookupargp->argsp->array = argop;

6233         argcnt = lookupargp->header_len;
6234         argop += argcnt;

6236         /*
6237          * loop and create a lookup op and possibly getattr/getfh for
6238          * each component. Skip "." components.
6239          */
6240         for (s = nm; *s != '\0'; s = p) {
6241                 /*
6242                  * Set up a pathname struct for each component if needed
6243                  */
6244                 while (*s == '/')
6245                         s++;
6246                 if (*s == '\0')
6247                         break;

6249                 for (p = s; (*p != '/') && (*p != '\0'); p++)
6250                         ;
6251                 c = *p;
6252                 *p = '\0';

6254                 if (s[0] == '.' && s[1] == '\0') {
6255                         *p = c;
6256                         continue;
6257                 }
6258                 if (l4_getattrs == LKP4_LAST_ATTRDIR &&
6259                     strcmp(s, XATTR_RPATH) == 0) {
6260                         /* getfh XXX may not be needed in future */
6261                         argop->argop = OP_GETFH;
6262                         argop++;
6263                         argcnt++;

6265                         /* getattr */
6266                         argop->argop = OP_GETATTR;
6267                         argop->nfs_argop4_u.opgetattr.attr_request =
6268                             lookupargp->ga_bits;
6269                         argop->nfs_argop4_u.opgetattr.mi =
6270                             lookupargp->mi;
6271                         argop++;
6272                         argcnt++;

6274                         /* openattr */
6275                         argop->argop = OP_OPENATTR;
6276                 } else if (l4_getattrs == LKP4_LAST_NAMED_ATTR &&
6277                     strcmp(s, XATTR_RPATH) == 0) {
6278                         /* openattr */
6279                         argop->argop = OP_OPENATTR;
6280                         argop++;
6281                         argcnt++;

6283                         /* getfh XXX may not be needed in future */
6284                         argop->argop = OP_GETFH;
6285                         argop++;
6286                         argcnt++;
```



```
6288                         /* getattr */
6289                         argop->argop = OP_GETATTR;
6290                         argop->nfs_argop4_u.opgetattr.attr_request =
6291                             lookupargp->ga_bits;
6292                         argop->nfs_argop4_u.opgetattr.mi =
6293                             lookupargp->mi;
6294                         argop++;
6295                         argcnt++;
6296                         *p = c;
6297                         continue;
6298                 } else if (s[0] == '.' && s[1] == '.' && s[2] == '\0') {
6299                         /* lookupp */
6300                         argop->argop = OP_LOOKUPP;
6301                 } else {
6302                         /* lookup */
6303                         argop->argop = OP_LOOKUP;
6304                         (void) str_to_utf8(s,
6305                             &argop->nfs_argop4_u.oplookup.objname);
6306                 }
6307                 lookup_idx = argcnt;
6308                 argop++;
6309                 argcnt++;

6311                 *p = c;

6313                 if (l4_getattrs == LKP4_ALL_ATTRIBUTES) {
6314                         /* getfh XXX may not be needed in future */
6315                         argop->argop = OP_GETFH;
6316                         argop++;
6317                         argcnt++;

6319                         /* getattr */
6320                         argop->argop = OP_GETATTR;
6321                         argop->nfs_argop4_u.opgetattr.attr_request =
6322                             lookupargp->ga_bits;
6323                         argop->nfs_argop4_u.opgetattr.mi =
6324                             lookupargp->mi;
6325                         argop++;
6326                         argcnt++;
6327                 }
6328         }

6330         if ((l4_getattrs != LKP4_NO_ATTRIBUTES) &&
6331             ((l4_getattrs != LKP4_ALL_ATTRIBUTES) || (lookup_idx < 0))) {
6332                 if (needgetfh) {
6333                         /* stick in a post-lookup getfh */
6334                         argop->argop = OP_GETFH;
6335                         argcnt++;
6336                         argop++;
6337                 }
6338                 /* post-lookup getattr */
6339                 argop->argop = OP_GETATTR;
6340                 argop->nfs_argop4_u.opgetattr.attr_request =
6341                     lookupargp->ga_bits;
6342                 argop->nfs_argop4_u.opgetattr.mi = lookupargp->mi;
6343                 argcnt++;
6344         }
6345         argcnt += lookupargp->trailer_len;      /* actual op count */
6346         lookupargp->argsp->array_len = argcnt;
6347         lookupargp->arglen = arglen;

6349 #ifdef DEBUG
6350         if (nfs4_client_lookup_debug)
6351                 nfs4lookup_dump_compound("nfs4lookup_setup", argbase, argcnt);
6352 #endif
```

```
6354         return (lookup_idx);
6355 }

6357 static int
6358 nfs4openattr(vnode_t *dvp, vnode_t **avp, int cflag, cred_t *cr)
6359 {
6360         COMPOUND4args_clnt      args;
6361         COMPOUND4res_clnt       res;
6362         GETFH4res       *gf_res = NULL;
6363         nfs_argop4      argop[4];
6364         nfs_resop4      *resop = NULL;
6365         nfs4_sharedfh_t *sfhp;
6366         hrtime_t t;
6367         nfs4_error_t    e;

6369         rnode4_t        *drp;
6370         int             doqueue = 1;
6371         vnode_t         *vp;
6372         int             needrecov = 0;
6373         nfs4_recov_state_t recov_state;

6375         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

6377         *avp = NULL;
6378         recov_state.rs_flags = 0;
6379         recov_state.rs_num_retry_despite_err = 0;

6381 recov_retry:
6382         /* COMPOUND: putfh, openattr, getfh, getattr */
6383         args.array_len = 4;
6384         args.array = argop;
6385         args.ctag = TAG_OPENATTR;

6387         e.error = nfs4_start_op(VTOMI4(dvp), dvp, NULL, &recov_state);
6388         if (e.error)
6389                 return (e.error);

6391         drp = VTOR4(dvp);

6393         /* putfh */
6394         argop[0].argop = OP_CPUTFH;
6395         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

6397         /* openattr */
6398         argop[1].argop = OP_OPENATTR;
6399         argop[1].nfs_argop4_u.opopenattr.createdir = (cflag ? TRUE : FALSE);

6401         /* getfh */
6402         argop[2].argop = OP_GETFH;

6404         /* getattr */
6405         argop[3].argop = OP_GETATTR;
6406         argop[3].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
6407         argop[3].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

6409         NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
6410             "nfs4openattr: %s call, drp %s", needrecov ? "recov" : "first",
6411             rnode4info(drp)));

6413         t = gethrtime();

6415         rfs4call(VTOMI4(dvp), &args, &res, cr, &doqueue, 0, &e);

6417         needrecov = nfs4_needs_recovery(&e, FALSE, dvp->v_vfsp);
6418         if (needrecov) {
6419                 bool_t abort;
```



```
6421                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
6422                     "nfs4openattr: initiating recovery\n"));

6424                 abort = nfs4_start_recovery(&e,
6425                     VTOMI4(dvp), dvp, NULL, NULL, NULL,
6426                     OP_OPENATTR, NULL, NULL, NULL);
6427                 nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);
6428                 if (!e.error) {
6429                         e.error = geterrno4(res.status);
6430                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6431                 }
6432                 if (abort == FALSE)
6433                         goto recov_retry;
6434                 return (e.error);
6435         }

6437         if (e.error) {
6438                 nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);
6439                 return (e.error);
6440         }

6442         if (res.status) {
6443                 /*
6444                  * If OTW errro is NOTSUPP, then it should be
6445                  * translated to EINVAL.  All Solaris file system
6446                  * implementations return EINVAL to the syscall layer
6447                  * when the attrdir cannot be created due to an
6448                  * implementation restriction or noxattr mount option.
6449                  */
6450                 if (res.status == NFS4ERR_NOTSUPP) {
6451                         mutex_enter(&drp->r_statelock);
6452                         if (drp->r_xattr_dir)
6453                                 VN_RELE(drp->r_xattr_dir);
6454                         VN_HOLD(NFS4_XATTR_DIR_NOTSUPP);
6455                         drp->r_xattr_dir = NFS4_XATTR_DIR_NOTSUPP;
6456                         mutex_exit(&drp->r_statelock);

6458                         e.error = EINVAL;
6459                 } else {
6460                         e.error = geterrno4(res.status);
6461                 }

6463                 if (e.error) {
6464                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6465                         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state,
6466                             needrecov);
6467                         return (e.error);
6468                 }
6469         }

6471         resop = &res.array[0];  /* putfh res */
6472         ASSERT(resop->nfs_resop4_u.opgetfh.status == NFS4_OK);

6474         resop = &res.array[1];  /* openattr res */
6475         ASSERT(resop->nfs_resop4_u.opopenattr.status == NFS4_OK);

6477         resop = &res.array[2];  /* getfh res */
6478         gf_res = &resop->nfs_resop4_u.opgetfh;
6479         if (gf_res->object.nfs_fh4_len == 0) {
6480                 *avp = NULL;
6481                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6482                 nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);
6483                 return (ENOENT);
6484         }
```

```
6486         sfhp = sfh4_get(&gf_res->object, VTOMI4(dvp));
6487         vp = makenfs4node(sfhp, &res.array[3].nfs_resop4_u.opgetattr.ga_res,
6488             dvp->v_vfsp, t, cr, dvp,
6489             fn_get(VTOSV(dvp)->sv_name, XATTR_RPATH, sfhp));
6490         sfh4_rele(&sfhp);

6492         if (e.error)
6493                 PURGE_ATTRCACHE4(vp);

6495         mutex_enter(&vp->v_lock);
6496         vp->v_flag |= V_XATTRDIR;
6497         mutex_exit(&vp->v_lock);

6499         *avp = vp;

6501         mutex_enter(&drp->r_statelock);
6502         if (drp->r_xattr_dir)
6503                 VN_RELE(drp->r_xattr_dir);
6504         VN_HOLD(vp);
6505         drp->r_xattr_dir = vp;

6507         /*
6508          * Invalidate pathconf4 cache because r_xattr_dir is no longer
6509          * NULL.  xattrs could be created at any time, and we have no
6510          * way to update pc4_xattr_exists in the base object if/when
6511          * it happens.
6512          */
6513         drp->r_pathconf.pc4_xattr_valid = 0;

6515         mutex_exit(&drp->r_statelock);

6517         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);

6519         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

6521         return (0);
6522 }

6524 /* ARGSUSED */
6525 static int
6526 nfs4_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
6527     int mode, vnode_t **vpp, cred_t *cr, int flags, caller_context_t *ct,
6528     vsecattr_t *vsecp)
6529 {
6530         int error;
6531         vnode_t *vp = NULL;
6532         rnode4_t *rp;
6533         struct vattr vattr;
6534         rnode4_t *drp;
6535         vnode_t *tempvp;
6536         enum createmode4 createmode;
6537         bool_t must_trunc = FALSE;
6538         int     truncating = 0;

6540         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
6541                 return (EPERM);
6542         if (exclusive == EXCL && (dvp->v_flag & V_XATTRDIR)) {
6543                 return (EINVAL);
6544         }

6546         /* . and .. have special meaning in the protocol, reject them. */

6548         if (nm[0] == '.' && (nm[1] == '\0' || (nm[1] == '.' && nm[2] == '\0')))
6549                 return (EISDIR);

6551         drp = VTOR4(dvp);
```



```
6553         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp)))
6554                 return (EINTR);

6556 top:
6557         /*
6558          * We make a copy of the attributes because the caller does not
6559          * expect us to change what va points to.
6560          */
6561         vattr = *va;

6563         /*
6564          * If the pathname is "", then dvp is the root vnode of
6565          * a remote file mounted over a local directory.
6566          * All that needs to be done is access
6567          * checking and truncation.  Note that we avoid doing
6568          * open w/ create because the parent directory might
6569          * be in pseudo-fs and the open would fail.
6570          */
6571         if (*nm == '\0') {
6572                 error = 0;
6573                 VN_HOLD(dvp);
6574                 vp = dvp;
6575                 must_trunc = TRUE;
6576         } else {
6577                 /*
6578                  * We need to go over the wire, just to be sure whether the
6579                  * file exists or not.  Using the DNLC can be dangerous in
6580                  * this case when making a decision regarding existence.
6581                  */
6582                 error = nfs4lookup(dvp, nm, &vp, cr, 1);
6583         }

6585         if (exclusive)
6586                 createmode = EXCLUSIVE4;
6587         else
6588                 createmode = GUARDED4;

6590         /*
6591          * error would be set if the file does not exist on the
6592          * server, so lets go create it.
6593          */
6594         if (error) {
6595                 goto create_otw;
6596         }

6598         /*
6599          * File does exist on the server
6600          */
6601         if (exclusive == EXCL)
6602                 error = EEXIST;
6603         else if (vp->v_type == VDIR && (mode & VWRITE))
6604                 error = EISDIR;
6605         else {
6606                 /*
6607                  * If vnode is a device, create special vnode.
6608                  */
6609                 if (ISVDEV(vp->v_type)) {
6610                         tempvp = vp;
6611                         vp = specvp(vp, vp->v_rdev, vp->v_type, cr);
6612                         VN_RELE(tempvp);
6613                 }
6614                 if (!(error = VOP_ACCESS(vp, mode, 0, cr, ct))) {
6615                         if ((vattr.va_mask & AT_SIZE) &&
6616                             vp->v_type == VREG) {
6617                                 rp = VTOR4(vp);
```

```
6618                                /*
6619                                 * Check here for large file handled
6620                                 * by LF-unaware process (as
6621                                 * ufs_create() does)
6622                                 */
6623                                if (!(flags & FOFFMAX)) {
6624                                        mutex_enter(&rp->r_statelock);
6625                                        if (rp->r_size > MAXOFF32_T)
6626                                                error = EOVERFLOW;
6627                                        mutex_exit(&rp->r_statelock);
6628                                }

6630                                /* if error is set then we need to return */
6631                                if (error) {
6632                                        nfs_rw_exit(&drp->r_rwlock);
6633                                        VN_RELE(vp);
6634                                        return (error);
6635                                }

6637                                if (must_trunc) {
6638                                        vattr.va_mask = AT_SIZE;
6639                                        error = nfs4setattr(vp, &vattr, 0, cr,
6640                                            NULL);
6641                                } else {
6642                                        /*
6643                                         * we know we have a regular file that already
6644                                         * exists and we may end up truncating the file
6645                                         * as a result of the open_otw, so flush out
6646                                         * any dirty pages for this file first.
6647                                         */
6648                                        if (nfs4_has_pages(vp) &&
6649                                            ((rp->r_flags & R4DIRTY) ||
6650                                            rp->r_count > 0 ||
6651                                            rp->r_mapcnt > 0)) {
6652                                                error = nfs4_putpage(vp,
6653                                                    (offset_t)0, 0, 0, cr, ct);
6654                                                if (error && (error == ENOSPC ||
6655                                                    error == EDQUOT)) {
6656                                                        mutex_enter(
6657                                                            &rp->r_statelock);
6658                                                        if (!rp->r_error)
6659                                                                rp->r_error =
6660                                                                    error;
6661                                                        mutex_exit(
6662                                                            &rp->r_statelock);
6663                                                }
6664                                        }
6665                                        vattr.va_mask = (AT_SIZE |
6666                                            AT_TYPE | AT_MODE);
6667                                        vattr.va_type = VREG;
6668                                        createmode = UNCHECKED4;
6669                                        truncating = 1;
6670                                        goto create_otw;
6671                                }
6672                        }
6673                }
6674        }
6675        nfs_rw_exit(&drp->r_rwlock);
6676        if (error) {
6677                VN_RELE(vp);
6678        } else {
6679                vnode_t *tvp;
6680                rnode4_t *trp;
6681                tvp = vp;
6682                if (vp->v_type == VREG) {
6683                        trp = VTOR4(vp);
```



```
6684                            if (IS_SHADOW(vp, trp))
6685                                    tvp = RTOV4(trp);
6686                    }

6688                    if (must_trunc) {
6689                            /*
6690                             * existing file got truncated, notify.
6691                             */
6692                            vnevent_create(tvp, ct);
6693                    }

6695                    *vpp = vp;
6696            }
6697            return (error);

6699    create_otw:
6700            dnlc_remove(dvp, nm);

6702            ASSERT(vattr.va_mask & AT_TYPE);

6704            /*
6705             * If not a regular file let nfs4mknod() handle it.
6706             */
6707            if (vattr.va_type != VREG) {
6708                    error = nfs4mknod(dvp, nm, &vattr, exclusive, mode, vpp, cr);
6709                    nfs_rw_exit(&drp->r_rwlock);
6710                    return (error);
6711            }

6713            /*
6714             * It _is_ a regular file.
6715             */
6716            ASSERT(vattr.va_mask & AT_MODE);
6717            if (MANDMODE(vattr.va_mode)) {
6718                    nfs_rw_exit(&drp->r_rwlock);
6719                    return (EACCES);
6720            }

6722            /*
6723             * If this happens to be a mknod of a regular file, then flags will
6724             * have neither FREAD or FWRITE.  However, we must set at least one
6725             * for the call to nfs4open_otw.  If it's open(O_CREAT) driving
6726             * nfs4_create, then either FREAD, FWRITE, or FRDWR has already been
6727             * set (based on openmode specified by app).
6728             */
6729            if ((flags & (FREAD|FWRITE)) == 0)
6730                    flags |= (FREAD|FWRITE);

6732            error = nfs4open_otw(dvp, nm, &vattr, vpp, cr, 1, flags, createmode, 0);

6734            if (vp != NULL) {
6735                    /* if create was successful, throw away the file's pages */
6736                    if (!error && (vattr.va_mask & AT_SIZE))
6737                            nfs4_invalidate_pages(vp, (vattr.va_size & PAGEMASK),
6738                                cr);
6739                    /* release the lookup hold */
6740                    VN_RELE(vp);
6741                    vp = NULL;
6742            }

6744            /*
6745             * validate that we opened a regular file. This handles a misbehaving
6746             * server that returns an incorrect FH.
6747             */
6748            if ((error == 0) && *vpp && (*vpp)->v_type != VREG) {
6749                    error = EISDIR;
```

```
6750                 VN_RELE(*vpp);
6751         }

6753         /*
6754          * If this is not an exclusive create, then the CREATE
6755          * request will be made with the GUARDED mode set.  This
6756          * means that the server will return EEXIST if the file
6757          * exists.  The file could exist because of a retransmitted
6758          * request.  In this case, we recover by starting over and
6759          * checking to see whether the file exists.  This second
6760          * time through it should and a CREATE request will not be
6761          * sent.
6762          *
6763          * This handles the problem of a dangling CREATE request
6764          * which contains attributes which indicate that the file
6765          * should be truncated.  This retransmitted request could
6766          * possibly truncate valid data in the file if not caught
6767          * by the duplicate request mechanism on the server or if
6768          * not caught by other means.  The scenario is:
6769          *
6770          * Client transmits CREATE request with size = 0
6771          * Client times out, retransmits request.
6772          * Response to the first request arrives from the server
6773          *  and the client proceeds on.
6774          * Client writes data to the file.
6775          * The server now processes retransmitted CREATE request
6776          *  and truncates file.
6777          *
6778          * The use of the GUARDED CREATE request prevents this from
6779          * happening because the retransmitted CREATE would fail
6780          * with EEXIST and would not truncate the file.
6781          */
6782         if (error == EEXIST && exclusive == NONEXCL) {
6783 #ifdef DEBUG
6784                 nfs4_create_misses++;
6785 #endif
6786                 goto top;
6787         }
6788         nfs_rw_exit(&drp->r_rwlock);
6789         if (truncating && !error && *vpp) {
6790                 vnode_t *tvp;
6791                 rnode4_t *trp;
6792                 /*
6793                  * existing file got truncated, notify.
6794                  */
6795                 tvp = *vpp;
6796                 trp = VTOR4(tvp);
6797                 if (IS_SHADOW(tvp, trp))
6798                         tvp = RTOV4(trp);
6799                 vnevent_create(tvp, ct);
6800         }
6801         return (error);
6802 }

6804 /*
6805  * Create compound (for mkdir, mknod, symlink):
6806  * { Putfh <dfh>; Create; Getfh; Getattr }
6807  * It's okay if setattr failed to set gid - this is not considered
6808  * an error, but purge attrs in that case.
6809  */
6810 static int
6811 call_nfs4_create_req(vnode_t *dvp, char *nm, void *data, struct vattr *va,
6812     vnode_t **vpp, cred_t *cr, nfs_ftype4 type)
6813 {
6814         int need_end_op = FALSE;
6815         COMPOUND4args_clnt args;
```



```
6816         COMPOUND4res_clnt res, *resp = NULL;
6817         nfs_argop4 *argop;
6818         nfs_resop4 *resop;
6819         int doqueue;
6820         mntinfo4_t *mi;
6821         rnode4_t *drp = VTOR4(dvp);
6822         change_info4 *cinfo;
6823         GETFH4res *gf_res;
6824         struct vattr vattr;
6825         vnode_t *vp;
6826         fattr4 *crattr;
6827         bool_t needrecov = FALSE;
6828         nfs4_recov_state_t recov_state;
6829         nfs4_sharedfh_t *sfhp = NULL;
6830         hrtime_t t;
6831         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
6832         int numops, argoplist_size, setgid_flag, idx_create, idx_fattr;
6833         dirattr_info_t dinfo, *dinfop;
6834         servinfo4_t *svp;
6835         bitmap4 supp_attrs;

6837         ASSERT(type == NF4DIR || type == NF4LNK || type == NF4BLK ||
6838             type == NF4CHR || type == NF4SOCK || type == NF4FIFO);

6840         mi = VTOMI4(dvp);

6842         /*
6843          * Make sure we properly deal with setting the right gid
6844          * on a new directory to reflect the parent's setgid bit
6845          */
6846         setgid_flag = 0;
6847         if (type == NF4DIR) {
6848                 struct vattr dva;

6850                 va->va_mode &= ~VSGID;
6851                 dva.va_mask = AT_MODE | AT_GID;
6852                 if (VOP_GETATTR(dvp, &dva, 0, cr, NULL) == 0) {

6854                         /*
6855                          * If the parent's directory has the setgid bit set
6856                          * _and_ the client was able to get a valid mapping
6857                          * for the parent dir's owner_group, we want to
6858                          * append NVERIFY(owner_group == dva.va_gid) and
6859                          * SETTATTR to the CREATE compound.
6860                          */
6861                         if (mi->mi_flags & MI4_GRPID || dva.va_mode & VSGID) {
6862                                 setgid_flag = 1;
6863                                 va->va_mode |= VSGID;
6864                                 if (dva.va_gid != GID_NOBODY) {
6865                                         va->va_mask |= AT_GID;
6866                                         va->va_gid = dva.va_gid;
6867                                 }
6868                         }
6869                 }
6870         }

6872         /*
6873          * Create ops:
6874          *      0:putfh(dir) 1:savefh(dir) 2:create 3:getfh(new) 4:getattr(new)
6875          *      5:restorefh(dir) 6:getattr(dir)
6876          *
6877          * if (setgid)
6878          *      0:putfh(dir) 1:create 2:getfh(new) 3:getattr(new)
6879          *      4:savefh(new) 5:putfh(dir) 6:getattr(dir) 7:restorefh(new)
6880          *      8:nverify 9:setattr
6881          */
```

```
6882         if (setgid_flag) {
6883                 numops = 10;
6884                 idx_create = 1;
6885                 idx_fattr = 3;
6886         } else {
6887                 numops = 7;
6888                 idx_create = 2;
6889                 idx_fattr = 4;
6890         }

6892         ASSERT(nfs_zone() == mi->mi_zone);
6893         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp))) {
6894                 return (EINTR);
6895         }
6896         recov_state.rs_flags = 0;
6897         recov_state.rs_num_retry_despite_err = 0;

6899         argoplist_size = numops * sizeof (nfs_argop4);
6900         argop = kmem_alloc(argoplist_size, KM_SLEEP);

6902 recov_retry:
6903         if (type == NF4LNK)
6904                 args.ctag = TAG_SYMLINK;
6905         else if (type == NF4DIR)
6906                 args.ctag = TAG_MKDIR;
6907         else
6908                 args.ctag = TAG_MKNOD;

6910         args.array_len = numops;
6911         args.array = argop;

6913         if (e.error = nfs4_start_op(mi, dvp, NULL, &recov_state)) {
6914                 nfs_rw_exit(&drp->r_rwlock);
6915                 kmem_free(argop, argoplist_size);
6916                 return (e.error);
6917         }
6918         need_end_op = TRUE;


6921         /* 0: putfh directory */
6922         argop[0].argop = OP_CPUTFH;
6923         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

6925         /* 1/2: Create object */
6926         argop[idx_create].argop = OP_CCREATE;
6927         argop[idx_create].nfs_argop4_u.opccreate.cname = nm;
6928         argop[idx_create].nfs_argop4_u.opccreate.type = type;
6929         if (type == NF4LNK) {
6930                 /*
6931                  * symlink, treat name as data
6932                  */
6933                 ASSERT(data != NULL);
6934                 argop[idx_create].nfs_argop4_u.opccreate.ftype4_u.clinkdata =
6935                     (char *)data;
6936         }
6937         if (type == NF4BLK || type == NF4CHR) {
6938                 ASSERT(data != NULL);
6939                 argop[idx_create].nfs_argop4_u.opccreate.ftype4_u.devdata =
6940                     *((specdata4 *)data);
6941         }

6943         crattr = &argop[idx_create].nfs_argop4_u.opccreate.createattrs;

6945         svp = drp->r_server;
6946         (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
6947         supp_attrs = svp->sv_supp_attrs;
```



```
6948         nfs_rw_exit(&svp->sv_lock);

6950         if (vattr_to_fattr4(va, NULL, crattr, 0, OP_CREATE, supp_attrs)) {
6951                 nfs_rw_exit(&drp->r_rwlock);
6952                 nfs4_end_op(mi, dvp, NULL, &recov_state, needrecov);
6953                 e.error = EINVAL;
6954                 kmem_free(argop, argoplist_size);
6955                 return (e.error);
6956         }

6958         /* 2/3: getfh fh of created object */
6959         ASSERT(idx_create + 1 == idx_fattr - 1);
6960         argop[idx_create + 1].argop = OP_GETFH;

6962         /* 3/4: getattr of new object */
6963         argop[idx_fattr].argop = OP_GETATTR;
6964         argop[idx_fattr].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
6965         argop[idx_fattr].nfs_argop4_u.opgetattr.mi = mi;

6967         if (setgid_flag) {
6968                 vattr_t _v;

6970                 argop[4].argop = OP_SAVEFH;

6972                 argop[5].argop = OP_CPUTFH;
6973                 argop[5].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

6975                 argop[6].argop = OP_GETATTR;
6976                 argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
6977                 argop[6].nfs_argop4_u.opgetattr.mi = mi;

6979                 argop[7].argop = OP_RESTOREFH;

6981                 /*
6982                  * nverify
6983                  *
6984                  * XXX - Revisit the last argument to nfs4_end_op()
6985                  *       once 5020486 is fixed.
6986                  */
6987                 _v.va_mask = AT_GID;
6988                 _v.va_gid = va->va_gid;
6989                 if (e.error = nfs4args_verify(&argop[8], &_v, OP_NVERIFY,
6990                     supp_attrs)) {
6991                         nfs4_end_op(mi, dvp, *vpp, &recov_state, TRUE);
6992                         nfs_rw_exit(&drp->r_rwlock);
6993                         nfs4_fattr4_free(crattr);
6994                         kmem_free(argop, argoplist_size);
6995                         return (e.error);
6996                 }

6998                 /*
6999                  * setattr
7000                  *
7001                  * We _know_ we're not messing with AT_SIZE or AT_XTIME,
7002                  * so no need for stateid or flags. Also we specify NULL
7003                  * rp since we're only interested in setting owner_group
7004                  * attributes.
7005                  */
7006                 nfs4args_setattr(&argop[9], &_v, NULL, 0, NULL, cr, supp_attrs,
7007                     &e.error, 0);

7009                 if (e.error) {
7010                         nfs4_end_op(mi, dvp, *vpp, &recov_state, TRUE);
7011                         nfs_rw_exit(&drp->r_rwlock);
7012                         nfs4_fattr4_free(crattr);
7013                         nfs4args_verify_free(&argop[8]);
```

```
7014                        kmem_free(argop, argoplist_size);
7015                        return (e.error);
7016                }
7017        } else {
7018                argop[1].argop = OP_SAVEFH;

7020                argop[5].argop = OP_RESTOREFH;

7022                argop[6].argop = OP_GETATTR;
7023                argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7024                argop[6].nfs_argop4_u.opgetattr.mi = mi;
7025        }

7027        dnlc_remove(dvp, nm);

7029        doqueue = 1;
7030        t = gethrtime();
7031        rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

7033        needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
7034        if (e.error) {
7035                PURGE_ATTRCACHE4(dvp);
7036                if (!needrecov)
7037                        goto out;
7038        }

7040        if (needrecov) {
7041                if (nfs4_start_recovery(&e, mi, dvp, NULL, NULL, NULL,
7042                    OP_CREATE, NULL, NULL, NULL) == FALSE) {
7043                        nfs4_end_op(mi, dvp, NULL, &recov_state,
7044                            needrecov);
7045                        need_end_op = FALSE;
7046                        nfs4_fattr4_free(crattr);
7047                        if (setgid_flag) {
7048                                nfs4args_verify_free(&argop[8]);
7049                                nfs4args_setattr_free(&argop[9]);
7050                        }
7051                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
7052                        goto recov_retry;
7053                }
7054        }

7056        resp = &res;

7058        if (res.status != NFS4_OK && res.array_len <= idx_fattr + 1) {

7060                if (res.status == NFS4ERR_BADOWNER)
7061                        nfs4_log_badowner(mi, OP_CREATE);

7063                e.error = geterrno4(res.status);

7065                /*
7066                 * This check is left over from when create was implemented
7067                 * using a setattr op (instead of createattrs).  If the
7068                 * putfh/create/getfh failed, the error was returned.  If
7069                 * setattr/getattr failed, we keep going.
7070                 *
7071                 * It might be better to get rid of the GETFH also, and just
7072                 * do PUTFH/CREATE/GETATTR since the FH attr is mandatory.
7073                 * Then if any of the operations failed, we could return the
7074                 * error now, and remove much of the error code below.
7075                 */
7076                if (res.array_len <= idx_fattr) {
7077                        /*
7078                         * Either Putfh, Create or Getfh failed.
7079                         */
```



```
7080                        PURGE_ATTRCACHE4(dvp);
7081                        /*
7082                         * nfs4_purge_stale_fh() may generate otw calls through
7083                         * nfs4_invalidate_pages. Hence the need to call
7084                         * nfs4_end_op() here to avoid nfs4_start_op() deadlock.
7085                         */
7086                        nfs4_end_op(mi, dvp, NULL, &recov_state,
7087                            needrecov);
7088                        need_end_op = FALSE;
7089                        nfs4_purge_stale_fh(e.error, dvp, cr);
7090                        goto out;
7091                }
7092        }

7094        resop = &res.array[idx_create]; /* create res */
7095        cinfo = &resop->nfs_resop4_u.opcreate.cinfo;

7097        resop = &res.array[idx_create + 1]; /* getfh res */
7098        gf_res = &resop->nfs_resop4_u.opgetfh;

7100        sfhp = sfh4_get(&gf_res->object, mi);
7101        if (e.error) {
7102                *vpp = vp = makenfs4node(sfhp, NULL, dvp->v_vfsp, t, cr, dvp,
7103                    fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
7104                if (vp->v_type == VNON) {
7105                        vattr.va_mask = AT_TYPE;
7106                        /*
7107                         * Need to call nfs4_end_op before nfs4getattr to avoid
7108                         * potential nfs4_start_op deadlock. See RFE 4777612.
7109                         */
7110                        nfs4_end_op(mi, dvp, NULL, &recov_state,
7111                            needrecov);
7112                        need_end_op = FALSE;
7113                        e.error = nfs4getattr(vp, &vattr, cr);
7114                        if (e.error) {
7115                                VN_RELE(vp);
7116                                *vpp = NULL;
7117                                goto out;
7118                        }
7119                        vp->v_type = vattr.va_type;
7120                }
7121                e.error = 0;
7122        } else {
7123                *vpp = vp = makenfs4node(sfhp,
7124                    &res.array[idx_fattr].nfs_resop4_u.opgetattr.ga_res,
7125                    dvp->v_vfsp, t, cr,
7126                    dvp, fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
7127        }

7129        /*
7130         * If compound succeeded, then update dir attrs
7131         */
7132        if (res.status == NFS4_OK) {
7133                dinfo.di_garp = &res.array[6].nfs_resop4_u.opgetattr.ga_res;
7134                dinfo.di_cred = cr;
7135                dinfo.di_time_call = t;
7136                dinfop = &dinfo;
7137        } else
7138                dinfop = NULL;

7140        /* Update directory cache attribute, readdir and dnlc caches */
7141        nfs4_update_dircaches(cinfo, dvp, vp, nm, dinfop);

7143 out:
7144        if (sfhp != NULL)
7145                sfh4_rele(&sfhp);
```

```
7146         nfs_rw_exit(&drp->r_rwlock);
7147         nfs4_fattr4_free(crattr);
7148         if (setgid_flag) {
7149                 nfs4args_verify_free(&argop[8]);
7150                 nfs4args_setattr_free(&argop[9]);
7151         }
7152         if (resp)
7153                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
7154         if (need_end_op)
7155                 nfs4_end_op(mi, dvp, NULL, &recov_state, needrecov);

7157         kmem_free(argop, argoplist_size);
7158         return (e.error);
7159 }

7161 /* ARGSUSED */
7162 static int
7163 nfs4mknod(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
7164     int mode, vnode_t **vpp, cred_t *cr)
7165 {
7166         int error;
7167         vnode_t *vp;
7168         nfs_ftype4 type;
7169         specdata4 spec, *specp = NULL;

7171         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

7173         switch (va->va_type) {
7174         case VCHR:
7175         case VBLK:
7176                 type = (va->va_type == VCHR) ? NF4CHR : NF4BLK;
7177                 spec.specdata1 = getmajor(va->va_rdev);
7178                 spec.specdata2 = getminor(va->va_rdev);
7179                 specp = &spec;
7180                 break;

7182         case VFIFO:
7183                 type = NF4FIFO;
7184                 break;
7185         case VSOCK:
7186                 type = NF4SOCK;
7187                 break;

7189         default:
7190                 return (EINVAL);
7191         }

7193         error = call_nfs4_create_req(dvp, nm, specp, va, &vp, cr, type);
7194         if (error) {
7195                 return (error);
7196         }

7198         /*
7199          * This might not be needed any more; special case to deal
7200          * with problematic v2/v3 servers.  Since create was unable
7201          * to set group correctly, not sure what hope setattr has.
7202          */
7203         if (va->va_gid != VTOR4(vp)->r_attr.va_gid) {
7204                 va->va_mask = AT_GID;
7205                 (void) nfs4setattr(vp, va, 0, cr, NULL);
7206         }

7208         /*
7209          * If vnode is a device create special vnode
7210          */
7211         if (ISVDEV(vp->v_type)) {
```



```
7212                 *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
7213                 VN_RELE(vp);
7214         } else {
7215                 *vpp = vp;
7216         }
7217         return (error);
7218 }

7220 /*
7221  * Remove requires that the current fh be the target directory.
7222  * After the operation, the current fh is unchanged.
7223  * The compound op structure is:
7224  *      PUTFH(targetdir), REMOVE
7225  *
7226  * Weirdness: if the vnode to be removed is open
7227  * we rename it instead of removing it and nfs_inactive
7228  * will remove the new name.
7229  */
7230 /* ARGSUSED */
7231 static int
7232 nfs4_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct, int flags)
7233 {
7234         COMPOUND4args_clnt args;
7235         COMPOUND4res_clnt res, *resp = NULL;
7236         REMOVE4res *rm_res;
7237         nfs_argop4 argop[3];
7238         nfs_resop4 *resop;
7239         vnode_t *vp;
7240         char *tmpname;
7241         int doqueue;
7242         mntinfo4_t *mi;
7243         rnode4_t *rp;
7244         rnode4_t *drp;
7245         int needrecov = 0;
7246         nfs4_recov_state_t recov_state;
7247         int isopen;
7248         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
7249         dirattr_info_t dinfo;

7251         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
7252                 return (EPERM);
7253         drp = VTOR4(dvp);
7254         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp)))
7255                 return (EINTR);

7257         e.error = nfs4lookup(dvp, nm, &vp, cr, 0);
7258         if (e.error) {
7259                 nfs_rw_exit(&drp->r_rwlock);
7260                 return (e.error);
7261         }

7263         if (vp->v_type == VDIR) {
7264                 VN_RELE(vp);
7265                 nfs_rw_exit(&drp->r_rwlock);
7266                 return (EISDIR);
7267         }

7269         /*
7270          * First just remove the entry from the name cache, as it
7271          * is most likely the only entry for this vp.
7272          */
7273         dnlc_remove(dvp, nm);

7275         rp = VTOR4(vp);

7277         /*
```

```
7278          * For regular file types, check to see if the file is open by looking
7279          * at the open streams.
7280          * For all other types, check the reference count on the vnode.  Since
7281          * they are not opened OTW they never have an open stream.
7282          *
7283          * If the file is open, rename it to .nfsXXXX.
7284          */
7285         if (vp->v_type != VREG) {
7286                 /*
7287                  * If the file has a v_count > 1 then there may be more than one
7288                  * entry in the name cache due multiple links or an open file,
7289                  * but we don't have the real reference count so flush all
7290                  * possible entries.
7291                  */
7292                 if (vp->v_count > 1)
7293                         dnlc_purge_vp(vp);
7294
7295                 /*
7296                  * Now we have the real reference count.
7297                  */
7298                 isopen = vp->v_count > 1;
7299         } else {
7300                 mutex_enter(&rp->r_os_lock);
7301                 isopen = list_head(&rp->r_open_streams) != NULL;
7302                 mutex_exit(&rp->r_os_lock);
7303         }
7304
7305         mutex_enter(&rp->r_statelock);
7306         if (isopen &&
7307             (rp->r_unldvp == NULL || strcmp(nm, rp->r_unlname) == 0)) {
7308                 mutex_exit(&rp->r_statelock);
7309                 tmpname = newname();
7310                 e.error = nfs4rename(dvp, nm, dvp, tmpname, cr, ct);
7311                 if (e.error)
7312                         kmem_free(tmpname, MAXNAMELEN);
7313                 else {
7314                         mutex_enter(&rp->r_statelock);
7315                         if (rp->r_unldvp == NULL) {
7316                                 VN_HOLD(dvp);
7317                                 rp->r_unldvp = dvp;
7318                                 if (rp->r_unlcred != NULL)
7319                                         crfree(rp->r_unlcred);
7320                                 crhold(cr);
7321                                 rp->r_unlcred = cr;
7322                                 rp->r_unlname = tmpname;
7323                         } else {
7324                                 kmem_free(rp->r_unlname, MAXNAMELEN);
7325                                 rp->r_unlname = tmpname;
7326                         }
7327                         mutex_exit(&rp->r_statelock);
7328                 }
7329                 VN_RELE(vp);
7330                 nfs_rw_exit(&drp->r_rwlock);
7331                 return (e.error);
7332         }
7333         /*
7334          * Actually remove the file/dir
7335          */
7336         mutex_exit(&rp->r_statelock);
7337
7338         /*
7339          * We need to flush any dirty pages which happen to
7340          * be hanging around before removing the file.
7341          * This shouldn't happen very often since in NFSv4
7342          * we should be close to open consistent.
7343          */
```



```
7344         if (nfs4_has_pages(vp) &&
7345             ((rp->r_flags & R4DIRTY) || rp->r_count > 0)) {
7346                 e.error = nfs4_putpage(vp, (u_offset_t)0, 0, 0, cr, ct);
7347                 if (e.error && (e.error == ENOSPC || e.error == EDQUOT)) {
7348                         mutex_enter(&rp->r_statelock);
7349                         if (!rp->r_error)
7350                                 rp->r_error = e.error;
7351                         mutex_exit(&rp->r_statelock);
7352                 }
7353         }
7354
7355         mi = VTOMI4(dvp);
7356
7357         (void) nfs4delegreturn(rp, NFS4_DR_REOPEN);
7358         recov_state.rs_flags = 0;
7359         recov_state.rs_num_retry_despite_err = 0;
7360
7361 recov_retry:
7362         /*
7363          * Remove ops: putfh dir; remove
7364          */
7365         args.ctag = TAG_REMOVE;
7366         args.array_len = 3;
7367         args.array = argop;
7368
7369         e.error = nfs4_start_op(VTOMI4(dvp), dvp, NULL, &recov_state);
7370         if (e.error) {
7371                 nfs_rw_exit(&drp->r_rwlock);
7372                 VN_RELE(vp);
7373                 return (e.error);
7374         }
7375
7376         /* putfh directory */
7377         argop[0].argop = OP_CPUTFH;
7378         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;
7379
7380         /* remove */
7381         argop[1].argop = OP_CREMOVE;
7382         argop[1].nfs_argop4_u.opcremove.ctarget = nm;
7383
7384         /* getattr dir */
7385         argop[2].argop = OP_GETATTR;
7386         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7387         argop[2].nfs_argop4_u.opgetattr.mi = mi;
7388
7389         doqueue = 1;
7390         dinfo.di_time_call = gethrtime();
7391         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);
7392
7393         PURGE_ATTRCACHE4(vp);
7394
7395         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
7396         if (e.error)
7397                 PURGE_ATTRCACHE4(dvp);
7398
7399         if (needrecov) {
7400                 if (nfs4_start_recovery(&e, VTOMI4(dvp), dvp,
7401                     NULL, NULL, NULL, OP_REMOVE, NULL, NULL, NULL) == FALSE) {
7402                         if (!e.error)
7403                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
7404                                     (caddr_t)&res);
7405                         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state,
7406                             needrecov);
7407                         goto recov_retry;
7408                 }
7409         }
```

```
7411         /*
7412          * Matching nfs4_end_op() for start_op() above.
7413          * There is a path in the code below which calls
7414          * nfs4_purge_stale_fh(), which may generate otw calls through
7415          * nfs4_invalidate_pages. Hence we need to call nfs4_end_op()
7416          * here to avoid nfs4_start_op() deadlock.
7417          */
7418         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);

7420         if (!e.error) {
7421                 resp = &res;

7423                 if (res.status) {
7424                         e.error = geterrno4(res.status);
7425                         PURGE_ATTRCACHE4(dvp);
7426                         nfs4_purge_stale_fh(e.error, dvp, cr);
7427                 } else {
7428                         resop = &res.array[1];  /* remove res */
7429                         rm_res = &resop->nfs_resop4_u.opremove;

7431                         dinfo.di_garp =
7432                             &res.array[2].nfs_resop4_u.opgetattr.ga_res;
7433                         dinfo.di_cred = cr;

7435                         /* Update directory attr, readdir and dnlc caches */
7436                         nfs4_update_dircaches(&rm_res->cinfo, dvp, NULL, NULL,
7437                             &dinfo);
7438                 }
7439         }
7440         nfs_rw_exit(&drp->r_rwlock);
7441         if (resp)
7442                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

7444         if (e.error == 0) {
7445                 vnode_t *tvp;
7446                 rnode4_t *trp;
7447                 trp = VTOR4(vp);
7448                 tvp = vp;
7449                 if (IS_SHADOW(vp, trp))
7450                         tvp = RTOV4(trp);
7451                 vnevent_remove(tvp, dvp, nm, ct);
7452         }
7453         VN_RELE(vp);
7454         return (e.error);
7455 }

7457 /*
7458  * Link requires that the current fh be the target directory and the
7459  * saved fh be the source fh. After the operation, the current fh is unchanged.
7460  * Thus the compound op structure is:
7461  *      PUTFH(file), SAVEFH, PUTFH(targetdir), LINK, RESTOREFH,
7462  *      GETATTR(file)
7463  */
7464 /* ARGSUSED */
7465 static int
7466 nfs4_link(vnode_t *tdvp, vnode_t *svp, char *tnm, cred_t *cr,
7467     caller_context_t *ct, int flags)
7468 {
7469         COMPOUND4args_clnt args;
7470         COMPOUND4res_clnt res, *resp = NULL;
7471         LINK4res *ln_res;
7472         int argoplist_size  = 7 * sizeof (nfs_argop4);
7473         nfs_argop4 *argop;
7474         nfs_resop4 *resop;
7475         vnode_t *realvp, *nvp;
```



```
7476         int doqueue;
7477         mntinfo4_t *mi;
7478         rnode4_t *tdrp;
7479         bool_t needrecov = FALSE;
7480         nfs4_recov_state_t recov_state;
7481         hrtime_t t;
7482         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
7483         dirattr_info_t dinfo;

7485         ASSERT(*tnm != '\0');
7486         ASSERT(tdvp->v_type == VDIR);
7487         ASSERT(nfs4_consistent_type(tdvp));
7488         ASSERT(nfs4_consistent_type(svp));

7490         if (nfs_zone() != VTOMI4(tdvp)->mi_zone)
7491                 return (EPERM);
7492         if (VOP_REALVP(svp, &realvp, ct) == 0) {
7493                 svp = realvp;
7494                 ASSERT(nfs4_consistent_type(svp));
7495         }

7497         tdrp = VTOR4(tdvp);
7498         mi = VTOMI4(svp);

7500         if (!(mi->mi_flags & MI4_LINK)) {
7501                 return (EOPNOTSUPP);
7502         }
7503         recov_state.rs_flags = 0;
7504         recov_state.rs_num_retry_despite_err = 0;

7506         if (nfs_rw_enter_sig(&tdrp->r_rwlock, RW_WRITER, INTR4(tdvp)))
7507                 return (EINTR);

7509 recov_retry:
7510         argop = kmem_alloc(argoplist_size, KM_SLEEP);

7512         args.ctag = TAG_LINK;

7514         /*
7515          * Link ops: putfh fl; savefh; putfh tdir; link; getattr(dir);
7516          * restorefh; getattr(fl)
7517          */
7518         args.array_len = 7;
7519         args.array = argop;

7521         e.error = nfs4_start_op(VTOMI4(svp), svp, tdvp, &recov_state);
7522         if (e.error) {
7523                 kmem_free(argop, argoplist_size);
7524                 nfs_rw_exit(&tdrp->r_rwlock);
7525                 return (e.error);
7526         }

7528         /* 0. putfh file */
7529         argop[0].argop = OP_CPUTFH;
7530         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(svp)->r_fh;

7532         /* 1. save current fh to free up the space for the dir */
7533         argop[1].argop = OP_SAVEFH;

7535         /* 2. putfh targetdir */
7536         argop[2].argop = OP_CPUTFH;
7537         argop[2].nfs_argop4_u.opcputfh.sfh = tdrp->r_fh;

7539         /* 3. link: current_fh is targetdir, saved_fh is source */
7540         argop[3].argop = OP_CLINK;
7541         argop[3].nfs_argop4_u.opclink.cnewname = tnm;
```

```
7543         /* 4. Get attributes of dir */
7544         argop[4].argop = OP_GETATTR;
7545         argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7546         argop[4].nfs_argop4_u.opgetattr.mi = mi;

7548         /* 5. If link was successful, restore current vp to file */
7549         argop[5].argop = OP_RESTOREFH;

7551         /* 6. Get attributes of linked object */
7552         argop[6].argop = OP_GETATTR;
7553         argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7554         argop[6].nfs_argop4_u.opgetattr.mi = mi;

7556         dnlc_remove(tdvp, tnm);

7558         doqueue = 1;
7559         t = gethrtime();

7561         rfs4call(VTOMI4(svp), &args, &res, cr, &doqueue, 0, &e);

7563         needrecov = nfs4_needs_recovery(&e, FALSE, svp->v_vfsp);
7564         if (e.error != 0 && !needrecov) {
7565                 PURGE_ATTRCACHE4(tdvp);
7566                 PURGE_ATTRCACHE4(svp);
7567                 nfs4_end_op(VTOMI4(svp), svp, tdvp, &recov_state, needrecov);
7568                 goto out;
7569         }

7571         if (needrecov) {
7572                 bool_t abort;

7574                 abort = nfs4_start_recovery(&e, VTOMI4(svp), svp, tdvp,
7575                     NULL, NULL, OP_LINK, NULL, NULL, NULL);
7576                 if (abort == FALSE) {
7577                         nfs4_end_op(VTOMI4(svp), svp, tdvp, &recov_state,
7578                             needrecov);
7579                         kmem_free(argop, argoplist_size);
7580                         if (!e.error)
7581                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
7582                                     (caddr_t)&res);
7583                         goto recov_retry;
7584                 } else {
7585                         if (e.error != 0) {
7586                                 PURGE_ATTRCACHE4(tdvp);
7587                                 PURGE_ATTRCACHE4(svp);
7588                                 nfs4_end_op(VTOMI4(svp), svp, tdvp,
7589                                     &recov_state, needrecov);
7590                                 goto out;
7591                         }
7592                         /* fall through for res.status case */
7593                 }
7594         }

7596         nfs4_end_op(VTOMI4(svp), svp, tdvp, &recov_state, needrecov);

7598         resp = &res;
7599         if (res.status) {
7600                 /* If link succeeded, then don't return error */
7601                 e.error = geterrno4(res.status);
7602                 if (res.array_len <= 4) {
7603                         /*
7604                          * Either Putfh, Savefh, Putfh dir, or Link failed
7605                          */
7606                         PURGE_ATTRCACHE4(svp);
7607                         PURGE_ATTRCACHE4(tdvp);
```



```
7608                         if (e.error == EOPNOTSUPP) {
7609                                 mutex_enter(&mi->mi_lock);
7610                                 mi->mi_flags &= ~MI4_LINK;
7611                                 mutex_exit(&mi->mi_lock);
7612                         }
7613                         /* Remap EISDIR to EPERM for non-root user for SVVS */
7614                         /* XXX-LP */
7615                         if (e.error == EISDIR && crgetuid(cr) != 0)
7616                                 e.error = EPERM;
7617                         goto out;
7618                 }
7619         }

7621         /* either no error or one of the postop getattr failed */

7623         /*
7624          * XXX - if LINK succeeded, but no attrs were returned for link
7625          * file, purge its cache.
7626          *
7627          * XXX Perform a simplified version of wcc checking. Instead of
7628          * have another getattr to get pre-op, just purge cache if
7629          * any of the ops prior to and including the getattr failed.
7630          * If the getattr succeeded then update the attrcache accordingly.
7631          */

7633         /*
7634          * update cache with link file postattrs.
7635          * Note: at this point resop points to link res.
7636          */
7637         resop = &res.array[3];  /* link res */
7638         ln_res = &resop->nfs_resop4_u.oplink;
7639         if (res.status == NFS4_OK)
7640                 e.error = nfs4_update_attrcache(res.status,
7641                     &res.array[6].nfs_resop4_u.opgetattr.ga_res,
7642                     t, svp, cr);

7644         /*
7645          * Call makenfs4node to create the new shadow vp for tnm.
7646          * We pass NULL attrs because we just cached attrs for
7647          * the src object.  All we're trying to accomplish is to
7648          * to create the new shadow vnode.
7649          */
7650         nvp = makenfs4node(VTOR4(svp)->r_fh, NULL, tdvp->v_vfsp, t, cr,
7651             tdvp, fn_get(VTOSV(tdvp)->sv_name, tnm, VTOR4(svp)->r_fh));

7653         /* Update target cache attribute, readdir and dnlc caches */
7654         dinfo.di_garp = &res.array[4].nfs_resop4_u.opgetattr.ga_res;
7655         dinfo.di_time_call = t;
7656         dinfo.di_cred = cr;

7658         nfs4_update_dircaches(&ln_res->cinfo, tdvp, nvp, tnm, &dinfo);
7659         ASSERT(nfs4_consistent_type(tdvp));
7660         ASSERT(nfs4_consistent_type(svp));
7661         ASSERT(nfs4_consistent_type(nvp));
7662         VN_RELE(nvp);

7664         if (!e.error) {
7665                 vnode_t *tvp;
7666                 rnode4_t *trp;
7667                 /*
7668                  * Notify the source file of this link operation.
7669                  */
7670                 trp = VTOR4(svp);
7671                 tvp = svp;
7672                 if (IS_SHADOW(svp, trp))
7673                         tvp = RTOV4(trp);
```

```
7674                    vnevent_link(tvp, ct);
7675            }
7676    out:
7677            kmem_free(argop, argoplist_size);
7678            if (resp)
7679                    (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

7681            nfs_rw_exit(&tdrp->r_rwlock);

7683            return (e.error);
7684    }

7686    /* ARGSUSED */
7687    static int
7688    nfs4_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
7689        caller_context_t *ct, int flags)
7690    {
7691            vnode_t *realvp;

7693            if (nfs_zone() != VTOMI4(odvp)->mi_zone)
7694                    return (EPERM);
7695            if (VOP_REALVP(ndvp, &realvp, ct) == 0)
7696                    ndvp = realvp;

7698            return (nfs4rename(odvp, onm, ndvp, nnm, cr, ct));
7699    }

7701    /*
7702     * nfs4rename does the real work of renaming in NFS Version 4.
7703     *
7704     * A file handle is considered volatile for renaming purposes if either
7705     * of the volatile bits are turned on. However, the compound may differ
7706     * based on the likelihood of the filehandle to change during rename.
7707     */
7708    static int
7709    nfs4rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
7710        caller_context_t *ct)
7711    {
7712            int error;
7713            mntinfo4_t *mi;
7714            vnode_t *nvp = NULL;
7715            vnode_t *ovp = NULL;
7716            char *tmpname = NULL;
7717            rnode4_t *rp;
7718            rnode4_t *odrp;
7719            rnode4_t *ndrp;
7720            int did_link = 0;
7721            int do_link = 1;
7722            nfsstat4 stat = NFS4_OK;

7724            ASSERT(nfs_zone() == VTOMI4(odvp)->mi_zone);
7725            ASSERT(nfs4_consistent_type(odvp));
7726            ASSERT(nfs4_consistent_type(ndvp));

7728            if (onm[0] == '.' && (onm[1] == '\0' ||
7729                (onm[1] == '.' && onm[2] == '\0')))
7730                    return (EINVAL);

7732            if (nnm[0] == '.' && (nnm[1] == '\0' ||
7733                (nnm[1] == '.' && nnm[2] == '\0')))
7734                    return (EINVAL);

7736            odrp = VTOR4(odvp);
7737            ndrp = VTOR4(ndvp);
7738            if ((intptr_t)odrp < (intptr_t)ndrp) {
7739                    if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR4(odvp)))
```



```
7740                            return (EINTR);
7741                    if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR4(ndvp))) {
7742                            nfs_rw_exit(&odrp->r_rwlock);
7743                            return (EINTR);
7744                    }
7745            } else {
7746                    if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR4(ndvp)))
7747                            return (EINTR);
7748                    if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR4(odvp))) {
7749                            nfs_rw_exit(&ndrp->r_rwlock);
7750                            return (EINTR);
7751                    }
7752            }

7754            /*
7755             * Lookup the target file.  If it exists, it needs to be
7756             * checked to see whether it is a mount point and whether
7757             * it is active (open).
7758             */
7759            error = nfs4lookup(ndvp, nnm, &nvp, cr, 0);
7760            if (!error) {
7761                    int     isactive;

7763                    ASSERT(nfs4_consistent_type(nvp));
7764                    /*
7765                     * If this file has been mounted on, then just
7766                     * return busy because renaming to it would remove
7767                     * the mounted file system from the name space.
7768                     */
7769                    if (vn_ismntpt(nvp)) {
7770                            VN_RELE(nvp);
7771                            nfs_rw_exit(&odrp->r_rwlock);
7772                            nfs_rw_exit(&ndrp->r_rwlock);
7773                            return (EBUSY);
7774                    }

7776                    /*
7777                     * First just remove the entry from the name cache, as it
7778                     * is most likely the only entry for this vp.
7779                     */
7780                    dnlc_remove(ndvp, nnm);

7782                    rp = VTOR4(nvp);

7784                    if (nvp->v_type != VREG) {
7785                            /*
7786                             * Purge the name cache of all references to this vnode
7787                             * so that we can check the reference count to infer
7788                             * whether it is active or not.
7789                             */
7790                            if (nvp->v_count > 1)
7791                                    dnlc_purge_vp(nvp);

7793                            isactive = nvp->v_count > 1;
7794                    } else {
7795                            mutex_enter(&rp->r_os_lock);
7796                            isactive = list_head(&rp->r_open_streams) != NULL;
7797                            mutex_exit(&rp->r_os_lock);
7798                    }

7800                    /*
7801                     * If the vnode is active and is not a directory,
7802                     * arrange to rename it to a
7803                     * temporary file so that it will continue to be
7804                     * accessible.  This implements the "unlink-open-file"
7805                     * semantics for the target of a rename operation.
```

```
7806                     * Before doing this though, make sure that the
7807                     * source and target files are not already the same.
7808                     */
7809                    if (isactive && nvp->v_type != VDIR) {
7810                            /*
7811                             * Lookup the source name.
7812                             */
7813                            error = nfs4lookup(odvp, onm, &ovp, cr, 0);

7815                            /*
7816                             * The source name *should* already exist.
7817                             */
7818                            if (error) {
7819                                    VN_RELE(nvp);
7820                                    nfs_rw_exit(&odrp->r_rwlock);
7821                                    nfs_rw_exit(&ndrp->r_rwlock);
7822                                    return (error);
7823                            }

7825                            ASSERT(nfs4_consistent_type(ovp));

7827                            /*
7828                             * Compare the two vnodes.  If they are the same,
7829                             * just release all held vnodes and return success.
7830                             */
7831                            if (VN_CMP(ovp, nvp)) {
7832                                    VN_RELE(ovp);
7833                                    VN_RELE(nvp);
7834                                    nfs_rw_exit(&odrp->r_rwlock);
7835                                    nfs_rw_exit(&ndrp->r_rwlock);
7836                                    return (0);
7837                            }

7839                            /*
7840                             * Can't mix and match directories and non-
7841                             * directories in rename operations.  We already
7842                             * know that the target is not a directory.  If
7843                             * the source is a directory, return an error.
7844                             */
7845                            if (ovp->v_type == VDIR) {
7846                                    VN_RELE(ovp);
7847                                    VN_RELE(nvp);
7848                                    nfs_rw_exit(&odrp->r_rwlock);
7849                                    nfs_rw_exit(&ndrp->r_rwlock);
7850                                    return (ENOTDIR);
7851                            }
7852 link_call:
7853                            /*
7854                             * The target file exists, is not the same as
7855                             * the source file, and is active.  We first
7856                             * try to Link it to a temporary filename to
7857                             * avoid having the server removing the file
7858                             * completely (which could cause data loss to
7859                             * the user's POV in the event the Rename fails
7860                             * -- see bug 1165874).
7861                             */
7862                            /*
7863                             * The do_link and did_link booleans are
7864                             * introduced in the event we get NFS4ERR_FILE_OPEN
7865                             * returned for the Rename.  Some servers can
7866                             * not Rename over an Open file, so they return
7867                             * this error.  The client needs to Remove the
7868                             * newly created Link and do two Renames, just
7869                             * as if the server didn't support LINK.
7870                             */
7871                            tmpname = newname();
```



```
7872                            error = 0;

7874                            if (do_link) {
7875                                    error = nfs4_link(ndvp, nvp, tmpname, cr,
7876                                        NULL, 0);
7877                            }
7878                            if (error == EOPNOTSUPP || !do_link) {
7879                                    error = nfs4_rename(ndvp, nnm, ndvp, tmpname,
7880                                        cr, NULL, 0);
7881                                    did_link = 0;
7882                            } else {
7883                                    did_link = 1;
7884                            }
7885                            if (error) {
7886                                    kmem_free(tmpname, MAXNAMELEN);
7887                                    VN_RELE(ovp);
7888                                    VN_RELE(nvp);
7889                                    nfs_rw_exit(&odrp->r_rwlock);
7890                                    nfs_rw_exit(&ndrp->r_rwlock);
7891                                    return (error);
7892                            }

7894                            mutex_enter(&rp->r_statelock);
7895                            if (rp->r_unldvp == NULL) {
7896                                    VN_HOLD(ndvp);
7897                                    rp->r_unldvp = ndvp;
7898                                    if (rp->r_unlcred != NULL)
7899                                            crfree(rp->r_unlcred);
7900                                    crhold(cr);
7901                                    rp->r_unlcred = cr;
7902                                    rp->r_unlname = tmpname;
7903                            } else {
7904                                    if (rp->r_unlname)
7905                                            kmem_free(rp->r_unlname, MAXNAMELEN);
7906                                    rp->r_unlname = tmpname;
7907                            }
7908                            mutex_exit(&rp->r_statelock);
7909                    }

7911                    (void) nfs4delegreturn(VTOR4(nvp), NFS4_DR_PUSH|NFS4_DR_REOPEN);

7913                    ASSERT(nfs4_consistent_type(nvp));
7914            }

7916            if (ovp == NULL) {
7917                    /*
7918                     * When renaming directories to be a subdirectory of a
7919                     * different parent, the dnlc entry for ".." will no
7920                     * longer be valid, so it must be removed.
7921                     *
7922                     * We do a lookup here to determine whether we are renaming
7923                     * a directory and we need to check if we are renaming
7924                     * an unlinked file.  This might have already been done
7925                     * in previous code, so we check ovp == NULL to avoid
7926                     * doing it twice.
7927                     */
7928                    error = nfs4lookup(odvp, onm, &ovp, cr, 0);
7929                    /*
7930                     * The source name *should* already exist.
7931                     */
7932                    if (error) {
7933                            nfs_rw_exit(&odrp->r_rwlock);
7934                            nfs_rw_exit(&ndrp->r_rwlock);
7935                            if (nvp) {
7936                                    VN_RELE(nvp);
7937                            }
```

```
7938                        return (error);
7939                }
7940                ASSERT(ovp != NULL);
7941                ASSERT(nfs4_consistent_type(ovp));
7942        }

7944        /*
7945         * Is the object being renamed a dir, and if so, is
7946         * it being renamed to a child of itself?  The underlying
7947         * fs should ultimately return EINVAL for this case;
7948         * however, buggy beta non-Solaris NFSv4 servers at
7949         * interop testing events have allowed this behavior,
7950         * and it caused our client to panic due to a recursive
7951         * mutex_enter in fn_move.
7952         *
7953         * The tedious locking in fn_move could be changed to
7954         * deal with this case, and the client could avoid the
7955         * panic; however, the client would just confuse itself
7956         * later and misbehave.  A better way to handle the broken
7957         * server is to detect this condition and return EINVAL
7958         * without ever sending the the bogus rename to the server.
7959         * We know the rename is invalid -- just fail it now.
7960         */
7961        if (ovp->v_type == VDIR && VN_CMP(ndvp, ovp)) {
7962                VN_RELE(ovp);
7963                nfs_rw_exit(&odrp->r_rwlock);
7964                nfs_rw_exit(&ndrp->r_rwlock);
7965                if (nvp) {
7966                        VN_RELE(nvp);
7967                }
7968                return (EINVAL);
7969        }

7971        (void) nfs4delegreturn(VTOR4(ovp), NFS4_DR_PUSH|NFS4_DR_REOPEN);

7973        /*
7974         * If FH4_VOL_RENAME or FH4_VOLATILE_ANY bits are set, it is
7975         * possible for the filehandle to change due to the rename.
7976         * If neither of these bits is set, but FH4_VOL_MIGRATION is set,
7977         * the fh will not change because of the rename, but we still need
7978         * to update its rnode entry with the new name for
7979         * an eventual fh change due to migration. The FH4_NOEXPIRE_ON_OPEN
7980         * has no effect on these for now, but for future improvements,
7981         * we might want to use it too to simplify handling of files
7982         * that are open with that flag on. (XXX)
7983         */
7984        mi = VTOMI4(odvp);
7985        if (NFS4_VOLATILE_FH(mi))
7986                error = nfs4rename_volatile_fh(odvp, onm, ovp, ndvp, nnm, cr,
7987                    &stat);
7988        else
7989                error = nfs4rename_persistent_fh(odvp, onm, ovp, ndvp, nnm, cr,
7990                    &stat);

7992        ASSERT(nfs4_consistent_type(odvp));
7993        ASSERT(nfs4_consistent_type(ndvp));
7994        ASSERT(nfs4_consistent_type(ovp));

7996        if (stat == NFS4ERR_FILE_OPEN && did_link) {
7997                do_link = 0;
7998                /*
7999                 * Before the 'link_call' code, we did a nfs4_lookup
8000                 * that puts a VN_HOLD on nvp.  After the nfs4_link
8001                 * call we call VN_RELE to match that hold.  We need
8002                 * to place an additional VN_HOLD here since we will
8003                 * be hitting that VN_RELE again.
```



```
8004                 */
8005                VN_HOLD(nvp);

8007                (void) nfs4_remove(ndvp, tmpname, cr, NULL, 0);

8009                /* Undo the unlinked file naming stuff we just did */
8010                mutex_enter(&rp->r_statelock);
8011                if (rp->r_unldvp) {
8012                        VN_RELE(ndvp);
8013                        rp->r_unldvp = NULL;
8014                        if (rp->r_unlcred != NULL)
8015                                crfree(rp->r_unlcred);
8016                        rp->r_unlcred = NULL;
8017                        /* rp->r_unlanme points to tmpname */
8018                        if (rp->r_unlname)
8019                                kmem_free(rp->r_unlname, MAXNAMELEN);
8020                        rp->r_unlname = NULL;
8021                }
8022                mutex_exit(&rp->r_statelock);

8024                if (nvp) {
8025                        VN_RELE(nvp);
8026                }
8027                goto link_call;
8028        }

8030        if (error) {
8031                VN_RELE(ovp);
8032                nfs_rw_exit(&odrp->r_rwlock);
8033                nfs_rw_exit(&ndrp->r_rwlock);
8034                if (nvp) {
8035                        VN_RELE(nvp);
8036                }
8037                return (error);
8038        }

8040        /*
8041         * when renaming directories to be a subdirectory of a
8042         * different parent, the dnlc entry for ".." will no
8043         * longer be valid, so it must be removed
8044         */
8045        rp = VTOR4(ovp);
8046        if (ndvp != odvp) {
8047                if (ovp->v_type == VDIR) {
8048                        dnlc_remove(ovp, "..");
8049                        if (rp->r_dir != NULL)
8050                                nfs4_purge_rddir_cache(ovp);
8051                }
8052        }

8054        /*
8055         * If we are renaming the unlinked file, update the
8056         * r_unldvp and r_unlname as needed.
8057         */
8058        mutex_enter(&rp->r_statelock);
8059        if (rp->r_unldvp != NULL) {
8060                if (strcmp(rp->r_unlname, onm) == 0) {
8061                        (void) strncpy(rp->r_unlname, nnm, MAXNAMELEN);
8062                        rp->r_unlname[MAXNAMELEN - 1] = '\0';
8063                        if (ndvp != rp->r_unldvp) {
8064                                VN_RELE(rp->r_unldvp);
8065                                rp->r_unldvp = ndvp;
8066                                VN_HOLD(ndvp);
8067                        }
8068                }
8069        }
```

```
8070         mutex_exit(&rp->r_statelock);

8072         /*
8073          * Notify the rename vnevents to source vnode, and to the target
8074          * vnode if it already existed.
8075          */
8076         if (error == 0) {
8077                 vnode_t *tvp;
8078                 rnode4_t *trp;
8079                 /*
8080                  * Notify the vnode. Each links is represented by
8081                  * a different vnode, in nfsv4.
8082                  */
8083                 if (nvp) {
8084                         trp = VTOR4(nvp);
8085                         tvp = nvp;
8086                         if (IS_SHADOW(nvp, trp))
8087                                 tvp = RTOV4(trp);
8088                         vnevent_rename_dest(tvp, ndvp, nnm, ct);
8089                 }

8091                 /*
8092                  * if the source and destination directory are not the
8093                  * same notify the destination directory.
8094                  */
8095                 if (VTOR4(odvp) != VTOR4(ndvp)) {
8096                         trp = VTOR4(ndvp);
8097                         tvp = ndvp;
8098                         if (IS_SHADOW(ndvp, trp))
8099                                 tvp = RTOV4(trp);
8100                         vnevent_rename_dest_dir(tvp, ct);
8101                 }

8103                 trp = VTOR4(ovp);
8104                 tvp = ovp;
8105                 if (IS_SHADOW(ovp, trp))
8106                         tvp = RTOV4(trp);
8107                 vnevent_rename_src(tvp, odvp, onm, ct);
8108         }

8110         if (nvp) {
8111                 VN_RELE(nvp);
8112         }
8113         VN_RELE(ovp);

8115         nfs_rw_exit(&odrp->r_rwlock);
8116         nfs_rw_exit(&ndrp->r_rwlock);

8118         return (error);
8119 }

8121 /*
8122  * When the parent directory has changed, sv_dfh must be updated
8123  */
8124 static void
8125 update_parentdir_sfh(vnode_t *vp, vnode_t *ndvp)
8126 {
8127         svnode_t *sv = VTOSV(vp);
8128         nfs4_sharedfh_t *old_dfh = sv->sv_dfh;
8129         nfs4_sharedfh_t *new_dfh = VTOR4(ndvp)->r_fh;

8131         sfh4_hold(new_dfh);
8132         sv->sv_dfh = new_dfh;
8133         sfh4_rele(&old_dfh);
8134 }
```



```
8136 /*
8137  * nfs4rename_persistent does the otw portion of renaming in NFS Version 4,
8138  * when it is known that the filehandle is persistent through rename.
8139  *
8140  * Rename requires that the current fh be the target directory and the
8141  * saved fh be the source directory. After the operation, the current fh
8142  * is unchanged.
8143  * The compound op structure for persistent fh rename is:
8144  *      PUTFH(sourcdir), SAVEFH, PUTFH(targetdir), RENAME
8145  * Rather than bother with the directory postop args, we'll simply
8146  * update that a change occurred in the cache, so no post-op getattrs.
8147  */
8148 static int
8149 nfs4rename_persistent_fh(vnode_t *odvp, char *onm, vnode_t *renvp,
8150     vnode_t *ndvp, char *nnm, cred_t *cr, nfsstat4 *statp)
8151 {
8152         COMPOUND4args_clnt args;
8153         COMPOUND4res_clnt res, *resp = NULL;
8154         nfs_argop4 *argop;
8155         nfs_resop4 *resop;
8156         int doqueue, argoplist_size;
8157         mntinfo4_t *mi;
8158         rnode4_t *odrp = VTOR4(odvp);
8159         rnode4_t *ndrp = VTOR4(ndvp);
8160         RENAME4res *rn_res;
8161         bool_t needrecov;
8162         nfs4_recov_state_t recov_state;
8163         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
8164         dirattr_info_t dinfo, *dinfop;

8166         ASSERT(nfs_zone() == VTOMI4(odvp)->mi_zone);

8168         recov_state.rs_flags = 0;
8169         recov_state.rs_num_retry_despite_err = 0;

8171         /*
8172          * Rename ops: putfh sdir; savefh; putfh tdir; rename; getattr tdir
8173          *
8174          * If source/target are different dirs, then append putfh(src); getattr
8175          */
8176         args.array_len = (odvp == ndvp) ? 5 : 7;
8177         argoplist_size = args.array_len * sizeof (nfs_argop4);
8178         args.array = argop = kmem_alloc(argoplist_size, KM_SLEEP);

8180 recov_retry:
8181         *statp = NFS4_OK;

8183         /* No need to Lookup the file, persistent fh */
8184         args.ctag = TAG_RENAME;

8186         mi = VTOMI4(odvp);
8187         e.error = nfs4_start_op(mi, odvp, ndvp, &recov_state);
8188         if (e.error) {
8189                 kmem_free(argop, argoplist_size);
8190                 return (e.error);
8191         }

8193         /* 0: putfh source directory */
8194         argop[0].argop = OP_CPUTFH;
8195         argop[0].nfs_argop4_u.opcputfh.sfh = odrp->r_fh;

8197         /* 1: Save source fh to free up current for target */
8198         argop[1].argop = OP_SAVEFH;

8200         /* 2: putfh targetdir */
8201         argop[2].argop = OP_CPUTFH;
```

```
8202         argop[2].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;
8203
8204         /* 3: current_fh is targetdir, saved_fh is sourcedir */
8205         argop[3].argop = OP_CRENAME;
8206         argop[3].nfs_argop4_u.opcrename.coldname = onm;
8207         argop[3].nfs_argop4_u.opcrename.cnewname = nnm;
8208
8209         /* 4: getattr (targetdir) */
8210         argop[4].argop = OP_GETATTR;
8211         argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8212         argop[4].nfs_argop4_u.opgetattr.mi = mi;
8213
8214         if (ndvp != odvp) {
8215
8216                 /* 5: putfh (sourcedir) */
8217                 argop[5].argop = OP_CPUTFH;
8218                 argop[5].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;
8219
8220                 /* 6: getattr (sourcedir) */
8221                 argop[6].argop = OP_GETATTR;
8222                 argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8223                 argop[6].nfs_argop4_u.opgetattr.mi = mi;
8224         }
8225
8226         dnlc_remove(odvp, onm);
8227         dnlc_remove(ndvp, nnm);
8228
8229         doqueue = 1;
8230         dinfo.di_time_call = gethrtime();
8231         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);
8232
8233         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
8234         if (e.error) {
8235                 PURGE_ATTRCACHE4(odvp);
8236                 PURGE_ATTRCACHE4(ndvp);
8237         } else {
8238                 *statp = res.status;
8239         }
8240
8241         if (needrecov) {
8242                 if (nfs4_start_recovery(&e, mi, odvp, ndvp, NULL, NULL,
8243                     OP_RENAME, NULL, NULL, NULL) == FALSE) {
8244                         nfs4_end_op(mi, odvp, ndvp, &recov_state, needrecov);
8245                         if (!e.error)
8246                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
8247                                     (caddr_t)&res);
8248                         goto recov_retry;
8249                 }
8250         }
8251
8252         if (!e.error) {
8253                 resp = &res;
8254                 /*
8255                  * as long as OP_RENAME
8256                  */
8257                 if (res.status != NFS4_OK && res.array_len <= 4) {
8258                         e.error = geterrno4(res.status);
8259                         PURGE_ATTRCACHE4(odvp);
8260                         PURGE_ATTRCACHE4(ndvp);
8261                         /*
8262                          * System V defines rename to return EEXIST, not
8263                          * ENOTEMPTY if the target directory is not empty.
8264                          * Over the wire, the error is NFSERR_ENOTEMPTY
8265                          * which geterrno4 maps to ENOTEMPTY.
8266                          */
8267                         if (e.error == ENOTEMPTY)
```



```
8268                                 e.error = EEXIST;
8269                 } else {
8270
8271                         resop = &res.array[3];  /* rename res */
8272                         rn_res = &resop->nfs_resop4_u.oprename;
8273
8274                         if (res.status == NFS4_OK) {
8275                                 /*
8276                                  * Update target attribute, readdir and dnlc
8277                                  * caches.
8278                                  */
8279                                 dinfo.di_garp =
8280                                     &res.array[4].nfs_resop4_u.opgetattr.ga_res;
8281                                 dinfo.di_cred = cr;
8282                                 dinfop = &dinfo;
8283                         } else
8284                                 dinfop = NULL;
8285
8286                         nfs4_update_dircaches(&rn_res->target_cinfo,
8287                             ndvp, NULL, NULL, dinfop);
8288
8289                         /*
8290                          * Update source attribute, readdir and dnlc caches
8291                          *
8292                          */
8293                         if (ndvp != odvp) {
8294                                 update_parentdir_sfh(renvp, ndvp);
8295
8296                                 if (dinfop)
8297                                         dinfo.di_garp =
8298                                             &(res.array[6].nfs_resop4_u.
8299                                             opgetattr.ga_res);
8300
8301                                 nfs4_update_dircaches(&rn_res->source_cinfo,
8302                                     odvp, NULL, NULL, dinfop);
8303                         }
8304
8305                         fn_move(VTOSV(renvp)->sv_name, VTOSV(ndvp)->sv_name,
8306                             nnm);
8307                 }
8308         }
8309
8310         if (resp)
8311                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
8312         nfs4_end_op(mi, odvp, ndvp, &recov_state, needrecov);
8313         kmem_free(argop, argoplist_size);
8314
8315         return (e.error);
8316 }
8317
8318 /*
8319  * nfs4rename_volatile_fh does the otw part of renaming in NFS Version 4, when
8320  * it is possible for the filehandle to change due to the rename.
8321  *
8322  * The compound req in this case includes a post-rename lookup and getattr
8323  * to ensure that we have the correct fh and attributes for the object.
8324  *
8325  * Rename requires that the current fh be the target directory and the
8326  * saved fh be the source directory. After the operation, the current fh
8327  * is unchanged.
8328  *
8329  * We need the new filehandle (hence a LOOKUP and GETFH) so that we can
8330  * update the filehandle for the renamed object.  We also get the old
8331  * filehandle for historical reasons; this should be taken out sometime.
8332  * This results in a rather cumbersome compound...
8333  *
```

```
 *    PUTFH(sourcdir), SAVEFH, LOOKUP(src), GETFH(old),
 *    PUTFH(targetdir), RENAME, LOOKUP(trgt), GETFH(new), GETATTR
 *
 */
static int
nfs4rename_volatile_fh(vnode_t *odvp, char *onm, vnode_t *ovp,
    vnode_t *ndvp, char *nnm, cred_t *cr, nfsstat4 *statp)
{
        COMPOUND4args_clnt args;
        COMPOUND4res_clnt res, *resp = NULL;
        int argoplist_size;
        nfs_argop4 *argop;
        nfs_resop4 *resop;
        int doqueue;
        mntinfo4_t *mi;
        rnode4_t *odrp = VTOR4(odvp);   /* old directory */
        rnode4_t *ndrp = VTOR4(ndvp);   /* new directory */
        rnode4_t *orp = VTOR4(ovp);     /* object being renamed */
        RENAME4res *rn_res;
        GETFH4res *ngf_res;
        bool_t needrecov;
        nfs4_recov_state_t recov_state;
        hrtime_t t;
        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
        dirattr_info_t dinfo, *dinfop = &dinfo;

        ASSERT(nfs_zone() == VTOMI4(odvp)->mi_zone);

        recov_state.rs_flags = 0;
        recov_state.rs_num_retry_despite_err = 0;

recov_retry:
        *statp = NFS4_OK;

        /*
         * There is a window between the RPC and updating the path and
         * filehandle stored in the rnode.  Lock out the FHEXPIRED recovery
         * code, so that it doesn't try to use the old path during that
         * window.
         */
        mutex_enter(&orp->r_statelock);
        while (orp->r_flags & R4RECEXPFH) {
                klwp_t *lwp = ttolwp(curthread);

                if (lwp != NULL)
                        lwp->lwp_nostop++;
                if (cv_wait_sig(&orp->r_cv, &orp->r_statelock) == 0) {
                        mutex_exit(&orp->r_statelock);
                        if (lwp != NULL)
                                lwp->lwp_nostop--;
                        return (EINTR);
                }
                if (lwp != NULL)
                        lwp->lwp_nostop--;
        }
        orp->r_flags |= R4RECEXPFH;
        mutex_exit(&orp->r_statelock);

        mi = VTOMI4(odvp);

        args.ctag = TAG_RENAME_VFH;
        args.array_len = (odvp == ndvp) ? 10 : 12;
        argoplist_size  = args.array_len * sizeof (nfs_argop4);
        argop = kmem_alloc(argoplist_size, KM_SLEEP);

        /*
```



```
         * Rename ops:
         *    PUTFH(sourcdir), SAVEFH, LOOKUP(src), GETFH(old),
         *    PUTFH(targetdir), RENAME, GETATTR(targetdir)
         *    LOOKUP(trgt), GETFH(new), GETATTR,
         *
         *    if (odvp != ndvp)
         *      add putfh(sourcedir), getattr(sourcedir) }
         */
        args.array = argop;

        e.error = nfs4_start_fop(mi, odvp, ndvp, OH_VFH_RENAME,
            &recov_state, NULL);
        if (e.error) {
                kmem_free(argop, argoplist_size);
                mutex_enter(&orp->r_statelock);
                orp->r_flags &= ~R4RECEXPFH;
                cv_broadcast(&orp->r_cv);
                mutex_exit(&orp->r_statelock);
                return (e.error);
        }

        /* 0: putfh source directory */
        argop[0].argop = OP_CPUTFH;
        argop[0].nfs_argop4_u.opcputfh.sfh = odrp->r_fh;

        /* 1: Save source fh to free up current for target */
        argop[1].argop = OP_SAVEFH;

        /* 2: Lookup pre-rename fh of renamed object */
        argop[2].argop = OP_CLOOKUP;
        argop[2].nfs_argop4_u.opclookup.cname = onm;

        /* 3: getfh fh of renamed object (before rename) */
        argop[3].argop = OP_GETFH;

        /* 4: putfh targetdir */
        argop[4].argop = OP_CPUTFH;
        argop[4].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;

        /* 5: current_fh is targetdir, saved_fh is sourcedir */
        argop[5].argop = OP_CRENAME;
        argop[5].nfs_argop4_u.opcrename.coldname = onm;
        argop[5].nfs_argop4_u.opcrename.cnewname = nnm;

        /* 6: getattr of target dir (post op attrs) */
        argop[6].argop = OP_GETATTR;
        argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
        argop[6].nfs_argop4_u.opgetattr.mi = mi;

        /* 7: Lookup post-rename fh of renamed object */
        argop[7].argop = OP_CLOOKUP;
        argop[7].nfs_argop4_u.opclookup.cname = nnm;

        /* 8: getfh fh of renamed object (after rename) */
        argop[8].argop = OP_GETFH;

        /* 9: getattr of renamed object */
        argop[9].argop = OP_GETATTR;
        argop[9].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
        argop[9].nfs_argop4_u.opgetattr.mi = mi;

        /*
         * If source/target dirs are different, then get new post-op
         * attrs for source dir also.
         */
        if (ndvp != odvp) {
```

```
8466                /* 10: putfh (sourcedir) */
8467                argop[10].argop = OP_CPUTFH;
8468                argop[10].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;

8470                /* 11: getattr (sourcedir) */
8471                argop[11].argop = OP_GETATTR;
8472                argop[11].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8473                argop[11].nfs_argop4_u.opgetattr.mi = mi;
8474        }

8476        dnlc_remove(odvp, onm);
8477        dnlc_remove(ndvp, nnm);

8479        doqueue = 1;
8480        t = gethrtime();
8481        rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

8483        needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
8484        if (e.error) {
8485                PURGE_ATTRCACHE4(odvp);
8486                PURGE_ATTRCACHE4(ndvp);
8487                if (!needrecov) {
8488                        nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8489                            &recov_state, needrecov);
8490                        goto out;
8491                }
8492        } else {
8493                *statp = res.status;
8494        }

8496        if (needrecov) {
8497                bool_t abort;

8499                abort = nfs4_start_recovery(&e, mi, odvp, ndvp, NULL, NULL,
8500                    OP_RENAME, NULL, NULL, NULL);
8501                if (abort == FALSE) {
8502                        nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8503                            &recov_state, needrecov);
8504                        kmem_free(argop, argoplist_size);
8505                        if (!e.error)
8506                                (void) xdr_free(xdr_COMPOUND4res_clnt,
8507                                    (caddr_t)&res);
8508                        mutex_enter(&orp->r_statelock);
8509                        orp->r_flags &= ~R4RECEXPFH;
8510                        cv_broadcast(&orp->r_cv);
8511                        mutex_exit(&orp->r_statelock);
8512                        goto recov_retry;
8513                } else {
8514                        if (e.error != 0) {
8515                                nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8516                                    &recov_state, needrecov);
8517                                goto out;
8518                        }
8519                        /* fall through for res.status case */
8520                }
8521        }

8523        resp = &res;
8524        /*
8525         * If OP_RENAME (or any prev op) failed, then return an error.
8526         * OP_RENAME is index 5, so if array len <= 6 we return an error.
8527         */
8528        if ((res.status != NFS4_OK) && (res.array_len <= 6)) {
8529                /*
8530                 * Error in an op other than last Getattr
8531                 */
```



```
8532                e.error = geterrno4(res.status);
8533                PURGE_ATTRCACHE4(odvp);
8534                PURGE_ATTRCACHE4(ndvp);
8535                /*
8536                 * System V defines rename to return EEXIST, not
8537                 * ENOTEMPTY if the target directory is not empty.
8538                 * Over the wire, the error is NFSERR_ENOTEMPTY
8539                 * which geterrno4 maps to ENOTEMPTY.
8540                 */
8541                if (e.error == ENOTEMPTY)
8542                        e.error = EEXIST;
8543                nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME, &recov_state,
8544                    needrecov);
8545                goto out;
8546        }

8548        /* rename results */
8549        rn_res = &res.array[5].nfs_resop4_u.oprename;

8551        if (res.status == NFS4_OK) {
8552                /* Update target attribute, readdir and dnlc caches */
8553                dinfo.di_garp =
8554                    &res.array[6].nfs_resop4_u.opgetattr.ga_res;
8555                dinfo.di_cred = cr;
8556                dinfo.di_time_call = t;
8557        } else
8558                dinfop = NULL;

8560        /* Update source cache attribute, readdir and dnlc caches */
8561        nfs4_update_dircaches(&rn_res->target_cinfo, ndvp, NULL, NULL, dinfop);

8563        /* Update source cache attribute, readdir and dnlc caches */
8564        if (ndvp != odvp) {
8565                update_parentdir_sfh(ovp, ndvp);

8567                /*
8568                 * If dinfop is non-NULL, then compound succeded, so
8569                 * set di_garp to attrs for source dir.  dinfop is only
8570                 * set to NULL when compound fails.
8571                 */
8572                if (dinfop)
8573                        dinfo.di_garp =
8574                            &res.array[11].nfs_resop4_u.opgetattr.ga_res;
8575                nfs4_update_dircaches(&rn_res->source_cinfo, odvp, NULL, NULL,
8576                    dinfop);
8577        }

8579        /*
8580         * Update the rnode with the new component name and args,
8581         * and if the file handle changed, also update it with the new fh.
8582         * This is only necessary if the target object has an rnode
8583         * entry and there is no need to create one for it.
8584         */
8585        resop = &res.array[8];  /* getfh new res */
8586        ngf_res = &resop->nfs_resop4_u.opgetfh;

8588        /*
8589         * Update the path and filehandle for the renamed object.
8590         */
8591        nfs4rename_update(ovp, ndvp, &ngf_res->object, nnm);

8593        nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME, &recov_state, needrecov);

8595        if (res.status == NFS4_OK) {
8596                resop++;        /* getattr res */
8597                e.error = nfs4_update_attrcache(res.status,
```

```
8598                    &resop->nfs_resop4_u.opgetattr.ga_res,
8599                    t, ovp, cr);
8600        }

8602 out:
8603        kmem_free(argop, argoplist_size);
8604        if (resp)
8605                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
8606        mutex_enter(&orp->r_statelock);
8607        orp->r_flags &= ~R4RECEXPFH;
8608        cv_broadcast(&orp->r_cv);
8609        mutex_exit(&orp->r_statelock);

8611        return (e.error);
8612 }

8614 /* ARGSUSED */
8615 static int
8616 nfs4_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp, cred_t *cr,
8617     caller_context_t *ct, int flags, vsecattr_t *vsecp)
8618 {
8619        int error;
8620        vnode_t *vp;

8622        if (nfs_zone() != VTOMI4(dvp)->mi_zone)
8623                return (EPERM);
8624        /*
8625         * As ".." has special meaning and rather than send a mkdir
8626         * over the wire to just let the server freak out, we just
8627         * short circuit it here and return EEXIST
8628         */
8629        if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0')
8630                return (EEXIST);

8632        /*
8633         * Decision to get the right gid and setgid bit of the
8634         * new directory is now made in call_nfs4_create_req.
8635         */
8636        va->va_mask |= AT_MODE;
8637        error = call_nfs4_create_req(dvp, nm, NULL, va, &vp, cr, NF4DIR);
8638        if (error)
8639                return (error);

8641        *vpp = vp;
8642        return (0);
8643 }


8646 /*
8647  * rmdir is using the same remove v4 op as does remove.
8648  * Remove requires that the current fh be the target directory.
8649  * After the operation, the current fh is unchanged.
8650  * The compound op structure is:
8651  *      PUTFH(targetdir), REMOVE
8652  */
8653 /*ARGSUSED4*/
8654 static int
8655 nfs4_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
8656     caller_context_t *ct, int flags)
8657 {
8658        int need_end_op = FALSE;
8659        COMPOUND4args_clnt args;
8660        COMPOUND4res_clnt res, *resp = NULL;
8661        REMOVE4res *rm_res;
8662        nfs_argop4 argop[3];
8663        nfs_resop4 *resop;
```



```
8664        vnode_t *vp;
8665        int doqueue;
8666        mntinfo4_t *mi;
8667        rnode4_t *drp;
8668        bool_t needrecov = FALSE;
8669        nfs4_recov_state_t recov_state;
8670        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
8671        dirattr_info_t dinfo, *dinfop;

8673        if (nfs_zone() != VTOMI4(dvp)->mi_zone)
8674                return (EPERM);
8675        /*
8676         * As ".." has special meaning and rather than send a rmdir
8677         * over the wire to just let the server freak out, we just
8678         * short circuit it here and return EEXIST
8679         */
8680        if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0')
8681                return (EEXIST);

8683        drp = VTOR4(dvp);
8684        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp)))
8685                return (EINTR);

8687        /*
8688         * Attempt to prevent a rmdir(".") from succeeding.
8689         */
8690        e.error = nfs4lookup(dvp, nm, &vp, cr, 0);
8691        if (e.error) {
8692                nfs_rw_exit(&drp->r_rwlock);
8693                return (e.error);
8694        }
8695        if (vp == cdir) {
8696                VN_RELE(vp);
8697                nfs_rw_exit(&drp->r_rwlock);
8698                return (EINVAL);
8699        }

8701        /*
8702         * Since nfsv4 remove op works on both files and directories,
8703         * check that the removed object is indeed a directory.
8704         */
8705        if (vp->v_type != VDIR) {
8706                VN_RELE(vp);
8707                nfs_rw_exit(&drp->r_rwlock);
8708                return (ENOTDIR);
8709        }

8711        /*
8712         * First just remove the entry from the name cache, as it
8713         * is most likely an entry for this vp.
8714         */
8715        dnlc_remove(dvp, nm);

8717        /*
8718         * If there vnode reference count is greater than one, then
8719         * there may be additional references in the DNLC which will
8720         * need to be purged.  First, trying removing the entry for
8721         * the parent directory and see if that removes the additional
8722         * reference(s).  If that doesn't do it, then use dnlc_purge_vp
8723         * to completely remove any references to the directory which
8724         * might still exist in the DNLC.
8725         */
8726        if (vp->v_count > 1) {
8727                dnlc_remove(vp, "..");
8728                if (vp->v_count > 1)
8729                        dnlc_purge_vp(vp);
```

```
8730         }

8732         mi = VTOMI4(dvp);
8733         recov_state.rs_flags = 0;
8734         recov_state.rs_num_retry_despite_err = 0;

8736 recov_retry:
8737         args.ctag = TAG_RMDIR;

8739         /*
8740          * Rmdir ops: putfh dir; remove
8741          */
8742         args.array_len = 3;
8743         args.array = argop;

8745         e.error = nfs4_start_op(VTOMI4(dvp), dvp, NULL, &recov_state);
8746         if (e.error) {
8747                 nfs_rw_exit(&drp->r_rwlock);
8748                 return (e.error);
8749         }
8750         need_end_op = TRUE;

8752         /* putfh directory */
8753         argop[0].argop = OP_CPUTFH;
8754         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

8756         /* remove */
8757         argop[1].argop = OP_CREMOVE;
8758         argop[1].nfs_argop4_u.opcremove.ctarget = nm;

8760         /* getattr (postop attrs for dir that contained removed dir) */
8761         argop[2].argop = OP_GETATTR;
8762         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8763         argop[2].nfs_argop4_u.opgetattr.mi = mi;

8765         dinfo.di_time_call = gethrtime();
8766         doqueue = 1;
8767         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

8769         PURGE_ATTRCACHE4(vp);

8771         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
8772         if (e.error) {
8773                 PURGE_ATTRCACHE4(dvp);
8774         }

8776         if (needrecov) {
8777                 if (nfs4_start_recovery(&e, VTOMI4(dvp), dvp, NULL, NULL,
8778                     NULL, OP_REMOVE, NULL, NULL, NULL) == FALSE) {
8779                         if (!e.error)
8780                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
8781                                     (caddr_t)&res);

8783                         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state,
8784                             needrecov);
8785                         need_end_op = FALSE;
8786                         goto recov_retry;
8787                 }
8788         }

8790         if (!e.error) {
8791                 resp = &res;

8793                 /*
8794                  * Only return error if first 2 ops (OP_REMOVE or earlier)
8795                  * failed.
```



```
8796                  */
8797                 if (res.status != NFS4_OK && res.array_len <= 2) {
8798                         e.error = geterrno4(res.status);
8799                         PURGE_ATTRCACHE4(dvp);
8800                         nfs4_end_op(VTOMI4(dvp), dvp, NULL,
8801                             &recov_state, needrecov);
8802                         need_end_op = FALSE;
8803                         nfs4_purge_stale_fh(e.error, dvp, cr);
8804                         /*
8805                          * System V defines rmdir to return EEXIST, not
8806                          * ENOTEMPTY if the directory is not empty.  Over
8807                          * the wire, the error is NFSERR_ENOTEMPTY which
8808                          * geterrno4 maps to ENOTEMPTY.
8809                          */
8810                         if (e.error == ENOTEMPTY)
8811                                 e.error = EEXIST;
8812                 } else {
8813                         resop = &res.array[1];  /* remove res */
8814                         rm_res = &resop->nfs_resop4_u.opremove;

8816                         if (res.status == NFS4_OK) {
8817                                 resop = &res.array[2];  /* dir attrs */
8818                                 dinfo.di_garp =
8819                                     &resop->nfs_resop4_u.opgetattr.ga_res;
8820                                 dinfo.di_cred = cr;
8821                                 dinfop = &dinfo;
8822                         } else
8823                                 dinfop = NULL;

8825                         /* Update dir attribute, readdir and dnlc caches */
8826                         nfs4_update_dircaches(&rm_res->cinfo, dvp, NULL, NULL,
8827                             dinfop);

8829                         /* destroy rddir cache for dir that was removed */
8830                         if (VTOR4(vp)->r_dir != NULL)
8831                                 nfs4_purge_rddir_cache(vp);
8832                 }
8833         }

8835         if (need_end_op)
8836                 nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);

8838         nfs_rw_exit(&drp->r_rwlock);

8840         if (resp)
8841                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

8843         if (e.error == 0) {
8844                 vnode_t *tvp;
8845                 rnode4_t *trp;
8846                 trp = VTOR4(vp);
8847                 tvp = vp;
8848                 if (IS_SHADOW(vp, trp))
8849                         tvp = RTOV4(trp);
8850                 vnevent_rmdir(tvp, dvp, nm, ct);
8851         }

8853         VN_RELE(vp);

8855         return (e.error);
8856 }

8858 /* ARGSUSED */
8859 static int
8860 nfs4_symlink(vnode_t *dvp, char *lnm, struct vattr *tva, char *tnm, cred_t *cr,
8861     caller_context_t *ct, int flags)
```

```
{
        int error;
        vnode_t *vp;
        rnode4_t *rp;
        char *contents;
        mntinfo4_t *mi = VTOMI4(dvp);

        if (nfs_zone() != mi->mi_zone)
                return (EPERM);
        if (!(mi->mi_flags & MI4_SYMLINK))
                return (EOPNOTSUPP);

        error = call_nfs4_create_req(dvp, lnm, tnm, tva, &vp, cr, NF4LNK);
        if (error)
                return (error);

        ASSERT(nfs4_consistent_type(vp));
        rp = VTOR4(vp);
        if (nfs4_do_symlink_cache && rp->r_symlink.contents == NULL) {

                contents = kmem_alloc(MAXPATHLEN, KM_SLEEP);

                if (contents != NULL) {
                        mutex_enter(&rp->r_statelock);
                        if (rp->r_symlink.contents == NULL) {
                                rp->r_symlink.len = strlen(tnm);
                                bcopy(tnm, contents, rp->r_symlink.len);
                                rp->r_symlink.contents = contents;
                                rp->r_symlink.size = MAXPATHLEN;
                                mutex_exit(&rp->r_statelock);
                        } else {
                                mutex_exit(&rp->r_statelock);
                                kmem_free((void *)contents, MAXPATHLEN);
                        }
                }
        }
        VN_RELE(vp);

        return (error);
}


/*
 * Read directory entries.
 * There are some weird things to look out for here.  The uio_loffset
 * field is either 0 or it is the offset returned from a previous
 * readdir.  It is an opaque value used by the server to find the
 * correct directory block to read. The count field is the number
 * of blocks to read on the server.  This is advisory only, the server
 * may return only one block's worth of entries.  Entries may be compressed
 * on the server.
 */
/* ARGSUSED */
static int
nfs4_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
    caller_context_t *ct, int flags)
{
        int error;
        uint_t count;
        rnode4_t *rp;
        rddir4_cache *rdc;
        rddir4_cache *rrdc;

        if (nfs_zone() != VTOMI4(vp)->mi_zone)
                return (EIO);
        rp = VTOR4(vp);
```



```
        ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));

        /*
         * Make sure that the directory cache is valid.
         */
        if (rp->r_dir != NULL) {
                if (nfs_disable_rddir_cache != 0) {
                        /*
                         * Setting nfs_disable_rddir_cache in /etc/system
                         * allows interoperability with servers that do not
                         * properly update the attributes of directories.
                         * Any cached information gets purged before an
                         * access is made to it.
                         */
                        nfs4_purge_rddir_cache(vp);
                }

                error = nfs4_validate_caches(vp, cr);
                if (error)
                        return (error);
        }

        count = MIN(uiop->uio_iov->iov_len, MAXBSIZE);

        /*
         * Short circuit last readdir which always returns 0 bytes.
         * This can be done after the directory has been read through
         * completely at least once.  This will set r_direof which
         * can be used to find the value of the last cookie.
         */
        mutex_enter(&rp->r_statelock);
        if (rp->r_direof != NULL &&
            uiop->uio_loffset == rp->r_direof->nfs4_ncookie) {
                mutex_exit(&rp->r_statelock);
#ifdef DEBUG
                nfs4_readdir_cache_shorts++;
#endif
                if (eofp)
                        *eofp = 1;
                return (0);
        }

        /*
         * Look for a cache entry.  Cache entries are identified
         * by the NFS cookie value and the byte count requested.
         */
        rdc = rddir4_cache_lookup(rp, uiop->uio_loffset, count);

        /*
         * If rdc is NULL then the lookup resulted in an unrecoverable error.
         */
        if (rdc == NULL) {
                mutex_exit(&rp->r_statelock);
                return (EINTR);
        }

        /*
         * Check to see if we need to fill this entry in.
         */
        if (rdc->flags & RDDIRREQ) {
                rdc->flags &= ~RDDIRREQ;
                rdc->flags |= RDDIR;
                mutex_exit(&rp->r_statelock);

                /*
```

```
8994                 * Do the readdir.
8995                 */
8996                nfs4readdir(vp, rdc, cr);

8998                /*
8999                 * Reacquire the lock, so that we can continue
9000                 */
9001                mutex_enter(&rp->r_statelock);
9002                /*
9003                 * The entry is now complete
9004                 */
9005                rdc->flags &= ~RDDIR;
9006        }

9008        ASSERT(!(rdc->flags & RDDIR));

9010        /*
9011         * If an error occurred while attempting
9012         * to fill the cache entry, mark the entry invalid and
9013         * just return the error.
9014         */
9015        if (rdc->error) {
9016                error = rdc->error;
9017                rdc->flags |= RDDIRREQ;
9018                rddir4_cache_rele(rp, rdc);
9019                mutex_exit(&rp->r_statelock);
9020                return (error);
9021        }

9023        /*
9024         * The cache entry is complete and good,
9025         * copyout the dirent structs to the calling
9026         * thread.
9027         */
9028        error = uiomove(rdc->entries, rdc->actlen, UIO_READ, uiop);

9030        /*
9031         * If no error occurred during the copyout,
9032         * update the offset in the uio struct to
9033         * contain the value of the next NFS 4 cookie
9034         * and set the eof value appropriately.
9035         */
9036        if (!error) {
9037                uiop->uio_loffset = rdc->nfs4_ncookie;
9038                if (eofp)
9039                        *eofp = rdc->eof;
9040        }

9042        /*
9043         * Decide whether to do readahead.  Don't if we
9044         * have already read to the end of directory.
9045         */
9046        if (rdc->eof) {
9047                /*
9048                 * Make the entry the direof only if it is cached
9049                 */
9050                if (rdc->flags & RDDIRCACHED)
9051                        rp->r_direof = rdc;
9052                rddir4_cache_rele(rp, rdc);
9053                mutex_exit(&rp->r_statelock);
9054                return (error);
9055        }

9057        /* Determine if a readdir readahead should be done */
9058        if (!(rp->r_flags & R4LOOKUP)) {
9059                rddir4_cache_rele(rp, rdc);
```



```
9060                mutex_exit(&rp->r_statelock);
9061                return (error);
9062        }

9064        /*
9065         * Now look for a readahead entry.
9066         *
9067         * Check to see whether we found an entry for the readahead.
9068         * If so, we don't need to do anything further, so free the new
9069         * entry if one was allocated.  Otherwise, allocate a new entry, add
9070         * it to the cache, and then initiate an asynchronous readdir
9071         * operation to fill it.
9072         */
9073        rrdc = rddir4_cache_lookup(rp, rdc->nfs4_ncookie, count);

9075        /*
9076         * A readdir cache entry could not be obtained for the readahead.  In
9077         * this case we skip the readahead and return.
9078         */
9079        if (rrdc == NULL) {
9080                rddir4_cache_rele(rp, rdc);
9081                mutex_exit(&rp->r_statelock);
9082                return (error);
9083        }

9085        /*
9086         * Check to see if we need to fill this entry in.
9087         */
9088        if (rrdc->flags & RDDIRREQ) {
9089                rrdc->flags &= ~RDDIRREQ;
9090                rrdc->flags |= RDDIR;
9091                rddir4_cache_rele(rp, rdc);
9092                mutex_exit(&rp->r_statelock);
9093 #ifdef DEBUG
9094                nfs4_readdir_readahead++;
9095 #endif
9096                /*
9097                 * Do the readdir.
9098                 */
9099                nfs4_async_readdir(vp, rrdc, cr, do_nfs4readdir);
9100                return (error);
9101        }

9103        rddir4_cache_rele(rp, rrdc);
9104        rddir4_cache_rele(rp, rdc);
9105        mutex_exit(&rp->r_statelock);
9106        return (error);
9107 }

9109 static int
9110 do_nfs4readdir(vnode_t *vp, rddir4_cache *rdc, cred_t *cr)
9111 {
9112        int error;
9113        rnode4_t *rp;

9115        ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

9117        rp = VTOR4(vp);

9119        /*
9120         * Obtain the readdir results for the caller.
9121         */
9122        nfs4readdir(vp, rdc, cr);

9124        mutex_enter(&rp->r_statelock);
9125        /*
```

```
9126          * The entry is now complete
9127          */
9128         rdc->flags &= ~RDDIR;

9130         error = rdc->error;
9131         if (error)
9132                 rdc->flags |= RDDIRREQ;
9133         rddir4_cache_rele(rp, rdc);
9134         mutex_exit(&rp->r_statelock);

9136         return (error);
9137 }

9139 /*
9140  * Read directory entries.
9141  * There are some weird things to look out for here.  The uio_loffset
9142  * field is either 0 or it is the offset returned from a previous
9143  * readdir.  It is an opaque value used by the server to find the
9144  * correct directory block to read. The count field is the number
9145  * of blocks to read on the server.  This is advisory only, the server
9146  * may return only one block's worth of entries.  Entries may be compressed
9147  * on the server.
9148  *
9149  * Generates the following compound request:
9150  * 1. If readdir offset is zero and no dnlc entry for parent exists,
9151  *    must include a Lookupp as well. In this case, send:
9152  *    { Putfh <fh>; Readdir; Lookupp; Getfh; Getattr }
9153  * 2. Otherwise just do: { Putfh <fh>; Readdir }
9154  *
9155  * Get complete attributes and filehandles for entries if this is the
9156  * first read of the directory. Otherwise, just get fileid's.
9157  */
9158 static void
9159 nfs4readdir(vnode_t *vp, rddir4_cache *rdc, cred_t *cr)
9160 {
9161         COMPOUND4args_clnt args;
9162         COMPOUND4res_clnt res;
9163         READDIR4args *rargs;
9164         READDIR4res_clnt *rd_res;
9165         bitmap4 rd_bitsval;
9166         nfs_argop4 argop[5];
9167         nfs_resop4 *resop;
9168         rnode4_t *rp = VTOR4(vp);
9169         mntinfo4_t *mi = VTOMI4(vp);
9170         int doqueue;
9171         u_longlong_t nodeid, pnodeid;   /* id's of dir and its parents */
9172         vnode_t *dvp;
9173         nfs_cookie4 cookie = (nfs_cookie4)rdc->nfs4_cookie;
9174         int num_ops, res_opcnt;
9175         bool_t needrecov = FALSE;
9176         nfs4_recov_state_t recov_state;
9177         hrtime_t t;
9178         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

9180         ASSERT(nfs_zone() == mi->mi_zone);
9181         ASSERT(rdc->flags & RDDIR);
9182         ASSERT(rdc->entries == NULL);

9184         /*
9185          * If rp were a stub, it should have triggered and caused
9186          * a mount for us to get this far.
9187          */
9188         ASSERT(!RP_ISSTUB(rp));

9190         num_ops = 2;
9191         if (cookie == (nfs_cookie4)0 || cookie == (nfs_cookie4)1) {
```



```
9192                 /*
9193                  * Since nfsv4 readdir may not return entries for "." and "..",
9194                  * the client must recreate them:
9195                  * To find the correct nodeid, do the following:
9196                  * For current node, get nodeid from dnlc.
9197                  * - if current node is rootvp, set pnodeid to nodeid.
9198                  * - else if parent is in the dnlc, get its nodeid from there.
9199                  * - else add LOOKUPP+GETATTR to compound.
9200                  */
9201                 nodeid = rp->r_attr.va_nodeid;
9202                 if (vp->v_flag & VROOT) {
9203                         pnodeid = nodeid;       /* root of mount point */
9204                 } else {
9205                         dvp = dnlc_lookup(vp, "..");
9206                         if (dvp != NULL && dvp != DNLC_NO_VNODE) {
9207                                 /* parent in dnlc cache - no need for otw */
9208                                 pnodeid = VTOR4(dvp)->r_attr.va_nodeid;
9209                         } else {
9210                                 /*
9211                                  * parent not in dnlc cache,
9212                                  * do lookupp to get its id
9213                                  */
9214                                 num_ops = 5;
9215                                 pnodeid = 0; /* set later by getattr parent */
9216                         }
9217                         if (dvp)
9218                                 VN_RELE(dvp);
9219                 }
9220         }
9221         recov_state.rs_flags = 0;
9222         recov_state.rs_num_retry_despite_err = 0;

9224         /* Save the original mount point security flavor */
9225         (void) save_mnt_secinfo(mi->mi_curr_serv);

9227 recov_retry:
9228         args.ctag = TAG_READDIR;

9230         args.array = argop;
9231         args.array_len = num_ops;

9233         if (e.error = nfs4_start_fop(VTOMI4(vp), vp, NULL, OH_READDIR,
9234             &recov_state, NULL)) {
9235                 /*
9236                  * If readdir a node that is a stub for a crossed mount point,
9237                  * keep the original secinfo flavor for the current file
9238                  * system, not the crossed one.
9239                  */
9240                 (void) check_mnt_secinfo(mi->mi_curr_serv, vp);
9241                 rdc->error = e.error;
9242                 return;
9243         }

9245         /*
9246          * Determine which attrs to request for dirents.  This code
9247          * must be protected by nfs4_start/end_fop because of r_server
9248          * (which will change during failover recovery).
9249          *
9250          */
9251         if (rp->r_flags & (R4LOOKUP | R4READDIRWATTR)) {
9252                 /*
9253                  * Get all vattr attrs plus filehandle and rdattr_error
9254                  */
9255                 rd_bitsval = NFS4_VATTR_MASK |
9256                     FATTR4_RDATTR_ERROR_MASK |
9257                     FATTR4_FILEHANDLE_MASK;
```

```
9259                if (rp->r_flags & R4READDIRWATTR) {
9260                        mutex_enter(&rp->r_statelock);
9261                        rp->r_flags &= ~R4READDIRWATTR;
9262                        mutex_exit(&rp->r_statelock);
9263                }
9264        } else {
9265                servinfo4_t *svp = rp->r_server;

9267                /*
9268                 * Already read directory. Use readdir with
9269                 * no attrs (except for mounted_on_fileid) for updates.
9270                 */
9271                rd_bitsval = FATTR4_RDATTR_ERROR_MASK;

9273                /*
9274                 * request mounted on fileid if supported, else request
9275                 * fileid.  maybe we should verify that fileid is supported
9276                 * and request something else if not.
9277                 */
9278                (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
9279                if (svp->sv_supp_attrs & FATTR4_MOUNTED_ON_FILEID_MASK)
9280                        rd_bitsval |= FATTR4_MOUNTED_ON_FILEID_MASK;
9281                nfs_rw_exit(&svp->sv_lock);
9282        }

9284        /* putfh directory fh */
9285        argop[0].argop = OP_CPUTFH;
9286        argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

9288        argop[1].argop = OP_READDIR;
9289        rargs = &argop[1].nfs_argop4_u.opreaddir;
9290        /*
9291         * 1 and 2 are reserved for client "." and ".." entry offset.
9292         * cookie 0 should be used over-the-wire to start reading at
9293         * the beginning of the directory excluding "." and "..".
9294         */
9295        if (rdc->nfs4_cookie == 0 ||
9296            rdc->nfs4_cookie == 1 ||
9297            rdc->nfs4_cookie == 2) {
9298                rargs->cookie = (nfs_cookie4)0;
9299                rargs->cookieverf = 0;
9300        } else {
9301                rargs->cookie = (nfs_cookie4)rdc->nfs4_cookie;
9302                mutex_enter(&rp->r_statelock);
9303                rargs->cookieverf = rp->r_cookieverf4;
9304                mutex_exit(&rp->r_statelock);
9305        }
9306        rargs->dircount = MIN(rdc->buflen, mi->mi_tsize);
9307        rargs->maxcount = mi->mi_tsize;
9308        rargs->attr_request = rd_bitsval;
9309        rargs->rdc = rdc;
9310        rargs->dvp = vp;
9311        rargs->mi = mi;
9312        rargs->cr = cr;


9315        /*
9316         * If count < than the minimum required, we return no entries
9317         * and fail with EINVAL
9318         */
9319        if (rargs->dircount < (DIRENT64_RECLEN(1) + DIRENT64_RECLEN(2))) {
9320                rdc->error = EINVAL;
9321                goto out;
9322        }
```



```
9324        if (args.array_len == 5) {
9325                /*
9326                 * Add lookupp and getattr for parent nodeid.
9327                 */
9328                argop[2].argop = OP_LOOKUPP;

9330                argop[3].argop = OP_GETFH;

9332                /* getattr parent */
9333                argop[4].argop = OP_GETATTR;
9334                argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
9335                argop[4].nfs_argop4_u.opgetattr.mi = mi;
9336        }

9338        doqueue = 1;

9340        if (mi->mi_io_kstats) {
9341                mutex_enter(&mi->mi_lock);
9342                kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
9343                mutex_exit(&mi->mi_lock);
9344        }

9346        /* capture the time of this call */
9347        rargs->t = t = gethrtime();

9349        rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

9351        if (mi->mi_io_kstats) {
9352                mutex_enter(&mi->mi_lock);
9353                kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
9354                mutex_exit(&mi->mi_lock);
9355        }

9357        needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);

9359        /*
9360         * If RPC error occurred and it isn't an error that
9361         * triggers recovery, then go ahead and fail now.
9362         */
9363        if (e.error != 0 && !needrecov) {
9364                rdc->error = e.error;
9365                goto out;
9366        }

9368        if (needrecov) {
9369                bool_t abort;

9371                NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
9372                    "nfs4readdir: initiating recovery.\n"));

9374                abort = nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
9375                    NULL, OP_READDIR, NULL, NULL, NULL);
9376                if (abort == FALSE) {
9377                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_READDIR,
9378                            &recov_state, needrecov);
9379                        if (!e.error)
9380                                (void) xdr_free(xdr_COMPOUND4res_clnt,
9381                                    (caddr_t)&res);
9382                        if (rdc->entries != NULL) {
9383                                kmem_free(rdc->entries, rdc->entlen);
9384                                rdc->entries = NULL;
9385                        }
9386                        goto recov_retry;
9387                }

9389                if (e.error != 0) {
```

```
9390                        rdc->error = e.error;
9391                        goto out;
9392                }

9394                /* fall through for res.status case */
9395        }

9397        res_opcnt = res.array_len;

9399        /*
9400         * If compound failed first 2 ops (PUTFH+READDIR), then return
9401         * failure here.  Subsequent ops are for filling out dot-dot
9402         * dirent, and if they fail, we still want to give the caller
9403         * the dirents returned by (the successful) READDIR op, so we need
9404         * to silently ignore failure for subsequent ops (LOOKUPP+GETATTR).
9405         *
9406         * One example where PUTFH+READDIR ops would succeed but
9407         * LOOKUPP+GETATTR would fail would be a dir that has r perm
9408         * but lacks x.  In this case, a POSIX server's VOP_READDIR
9409         * would succeed; however, VOP_LOOKUP(..) would fail since no
9410         * x perm.  We need to come up with a non-vendor-specific way
9411         * for a POSIX server to return d_ino from dotdot's dirent if
9412         * client only requests mounted_on_fileid, and just say the
9413         * LOOKUPP succeeded and fill out the GETATTR.  However, if
9414         * client requested any mandatory attrs, server would be required
9415         * to fail the GETATTR op because it can't call VOP_LOOKUP+VOP_GETATTR
9416         * for dotdot.
9417         */

9419        if (res.status) {
9420                if (res_opcnt <= 2) {
9421                        e.error = geterrno4(res.status);
9422                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_READDIR,
9423                            &recov_state, needrecov);
9424                        nfs4_purge_stale_fh(e.error, vp, cr);
9425                        rdc->error = e.error;
9426                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
9427                        if (rdc->entries != NULL) {
9428                                kmem_free(rdc->entries, rdc->entlen);
9429                                rdc->entries = NULL;
9430                        }
9431                        /*
9432                         * If readdir a node that is a stub for a
9433                         * crossed mount point, keep the original
9434                         * secinfo flavor for the current file system,
9435                         * not the crossed one.
9436                         */
9437                        (void) check_mnt_secinfo(mi->mi_curr_serv, vp);
9438                        return;
9439                }
9440        }

9442        resop = &res.array[1];  /* readdir res */
9443        rd_res = &resop->nfs_resop4_u.opreaddirclnt;

9445        mutex_enter(&rp->r_statelock);
9446        rp->r_cookieverf4 = rd_res->cookieverf;
9447        mutex_exit(&rp->r_statelock);

9449        /*
9450         * For "." and ".." entries
9451         * e.g.
9452         *      seek(cookie=0) -> "." entry with d_off = 1
9453         *      seek(cookie=1) -> ".." entry with d_off = 2
9454         */
9455        if (cookie == (nfs_cookie4) 0) {
```



```
9456                if (rd_res->dotp)
9457                        rd_res->dotp->d_ino = nodeid;
9458                if (rd_res->dotdotp)
9459                        rd_res->dotdotp->d_ino = pnodeid;
9460        }
9461        if (cookie == (nfs_cookie4) 1) {
9462                if (rd_res->dotdotp)
9463                        rd_res->dotdotp->d_ino = pnodeid;
9464        }


9467        /* LOOKUPP+GETATTR attemped */
9468        if (args.array_len == 5 && rd_res->dotdotp) {
9469                if (res.status == NFS4_OK && res_opcnt == 5) {
9470                        nfs_fh4 *fhp;
9471                        nfs4_sharedfh_t *sfhp;
9472                        vnode_t *pvp;
9473                        nfs4_ga_res_t *garp;

9475                        resop++;        /* lookupp */
9476                        resop++;        /* getfh   */
9477                        fhp = &resop->nfs_resop4_u.opgetfh.object;

9479                        resop++;        /* getattr of parent */

9481                        /*
9482                         * First, take care of finishing the
9483                         * readdir results.
9484                         */
9485                        garp = &resop->nfs_resop4_u.opgetattr.ga_res;
9486                        /*
9487                         * The d_ino of .. must be the inode number
9488                         * of the mounted filesystem.
9489                         */
9490                        if (garp->n4g_va.va_mask & AT_NODEID)
9491                                rd_res->dotdotp->d_ino =
9492                                    garp->n4g_va.va_nodeid;


9495                        /*
9496                         * Next, create the ".." dnlc entry
9497                         */
9498                        sfhp = sfh4_get(fhp, mi);
9499                        if (!nfs4_make_dotdot(sfhp, t, vp, cr, &pvp, 0)) {
9500                                dnlc_update(vp, "..", pvp);
9501                                VN_RELE(pvp);
9502                        }
9503                        sfh4_rele(&sfhp);
9504                }
9505        }

9507        if (mi->mi_io_kstats) {
9508                mutex_enter(&mi->mi_lock);
9509                KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
9510                KSTAT_IO_PTR(mi->mi_io_kstats)->nread += rdc->actlen;
9511                mutex_exit(&mi->mi_lock);
9512        }

9514        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

9516 out:
9517        /*
9518         * If readdir a node that is a stub for a crossed mount point,
9519         * keep the original secinfo flavor for the current file system,
9520         * not the crossed one.
9521         */
```

```
9522         (void) check_mnt_secinfo(mi->mi_curr_serv, vp);

9524         nfs4_end_fop(mi, vp, NULL, OH_READDIR, &recov_state, needrecov);
9525 }


9528 static int
9529 nfs4_bio(struct buf *bp, stable_how4 *stab_comm, cred_t *cr, bool_t readahead)
9530 {
9531         rnode4_t *rp = VTOR4(bp->b_vp);
9532         int count;
9533         int error;
9534         cred_t *cred_otw = NULL;
9535         offset_t offset;
9536         nfs4_open_stream_t *osp = NULL;
9537         bool_t first_time = TRUE;       /* first time getting otw cred */
9538         bool_t last_time = FALSE;       /* last time getting otw cred */

9540         ASSERT(nfs_zone() == VTOMI4(bp->b_vp)->mi_zone);

9542         DTRACE_IO1(start, struct buf *, bp);
9543         offset = ldbtob(bp->b_lblkno);

9545         if (bp->b_flags & B_READ) {
9546         read_again:
9547                 /*
9548                  * Releases the osp, if it is provided.
9549                  * Puts a hold on the cred_otw and the new osp (if found).
9550                  */
9551                 cred_otw = nfs4_get_otw_cred_by_osp(rp, cr, &osp,
9552                     &first_time, &last_time);
9553                 error = bp->b_error = nfs4read(bp->b_vp, bp->b_un.b_addr,
9554                     offset, bp->b_bcount, &bp->b_resid, cred_otw,
9555                     readahead, NULL);
9556                 crfree(cred_otw);
9557                 if (!error) {
9558                         if (bp->b_resid) {
9559                                 /*
9560                                  * Didn't get it all because we hit EOF,
9561                                  * zero all the memory beyond the EOF.
9562                                  */
9563                                 /* bzero(rdaddr + */
9564                                 bzero(bp->b_un.b_addr +
9565                                     bp->b_bcount - bp->b_resid, bp->b_resid);
9566                         }
9567                         mutex_enter(&rp->r_statelock);
9568                         if (bp->b_resid == bp->b_bcount &&
9569                             offset >= rp->r_size) {
9570                                 /*
9571                                  * We didn't read anything at all as we are
9572                                  * past EOF.  Return an error indicator back
9573                                  * but don't destroy the pages (yet).
9574                                  */
9575                                 error = NFS_EOF;
9576                         }
9577                         mutex_exit(&rp->r_statelock);
9578                 } else if (error == EACCES && last_time == FALSE) {
9579                         goto read_again;
9580                 }
9581         } else {
9582                 if (!(rp->r_flags & R4STALE)) {
9583 write_again:
9584                         /*
9585                          * Releases the osp, if it is provided.
9586                          * Puts a hold on the cred_otw and the new
9587                          * osp (if found).
```



```
9588                          */
9589                         cred_otw = nfs4_get_otw_cred_by_osp(rp, cr, &osp,
9590                             &first_time, &last_time);
9591                         mutex_enter(&rp->r_statelock);
9592                         count = MIN(bp->b_bcount, rp->r_size - offset);
9593                         mutex_exit(&rp->r_statelock);
9594                         if (count < 0)
9595                                 cmn_err(CE_PANIC, "nfs4_bio: write count < 0");
9596 #ifdef DEBUG
9597                         if (count == 0) {
9598                                 zoneid_t zoneid = getzoneid();

9600                                 zcmn_err(zoneid, CE_WARN,
9601                                     "nfs4_bio: zero length write at %lld",
9602                                     offset);
9603                                 zcmn_err(zoneid, CE_CONT, "flags=0x%x, "
9604                                     "b_bcount=%ld, file size=%lld",
9605                                     rp->r_flags, (long)bp->b_bcount,
9606                                     rp->r_size);
9607                                 sfh4_printfhandle(VTOR4(bp->b_vp)->r_fh);
9608                                 if (nfs4_bio_do_stop)
9609                                         debug_enter("nfs4_bio");
9610                         }
9611 #endif
9612                         error = nfs4write(bp->b_vp, bp->b_un.b_addr, offset,
9613                             count, cred_otw, stab_comm);
9614                         if (error == EACCES && last_time == FALSE) {
9615                                 crfree(cred_otw);
9616                                 goto write_again;
9617                         }
9618                         bp->b_error = error;
9619                         if (error && error != EINTR &&
9620                             !(bp->b_vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)) {
9621                                 /*
9622                                  * Don't print EDQUOT errors on the console.
9623                                  * Don't print asynchronous EACCES errors.
9624                                  * Don't print EFBIG errors.
9625                                  * Print all other write errors.
9626                                  */
9627                                 if (error != EDQUOT && error != EFBIG &&
9628                                     (error != EACCES ||
9629                                     !(bp->b_flags & B_ASYNC)))
9630                                         nfs4_write_error(bp->b_vp,
9631                                             error, cred_otw);
9632                                 /*
9633                                  * Update r_error and r_flags as appropriate.
9634                                  * If the error was ESTALE, then mark the
9635                                  * rnode as not being writeable and save
9636                                  * the error status.  Otherwise, save any
9637                                  * errors which occur from asynchronous
9638                                  * page invalidations.  Any errors occurring
9639                                  * from other operations should be saved
9640                                  * by the caller.
9641                                  */
9642                                 mutex_enter(&rp->r_statelock);
9643                                 if (error == ESTALE) {
9644                                         rp->r_flags |= R4STALE;
9645                                         if (!rp->r_error)
9646                                                 rp->r_error = error;
9647                                 } else if (!rp->r_error &&
9648                                     (bp->b_flags &
9649                                     (B_INVAL|B_FORCE|B_ASYNC)) ==
9650                                     (B_INVAL|B_FORCE|B_ASYNC)) {
9651                                         rp->r_error = error;
9652                                 }
9653                                 mutex_exit(&rp->r_statelock);
```

```
9654                        }
9655                        crfree(cred_otw);
9656                } else {
9657                        error = rp->r_error;
9658                        /*
9659                         * A close may have cleared r_error, if so,
9660                         * propagate ESTALE error return properly
9661                         */
9662                        if (error == 0)
9663                                error = ESTALE;
9664                }
9665        }

9667        if (error != 0 && error != NFS_EOF)
9668                bp->b_flags |= B_ERROR;

9670        if (osp)
9671                open_stream_rele(osp, rp);

9673        DTRACE_IO1(done, struct buf *, bp);

9675        return (error);
9676 }

9678 /* ARGSUSED */
9679 int
9680 nfs4_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
9681 {
9682        return (EREMOTE);
9683 }

9685 /* ARGSUSED2 */
9686 int
9687 nfs4_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
9688 {
9689        rnode4_t *rp = VTOR4(vp);

9691        if (!write_lock) {
9692                (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
9693                return (V_WRITELOCK_FALSE);
9694        }

9696        if ((rp->r_flags & R4DIRECTIO) ||
9697            (VTOMI4(vp)->mi_flags & MI4_DIRECTIO)) {
9698                (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
9699                if (rp->r_mapcnt == 0 && !nfs4_has_pages(vp))
9700                        return (V_WRITELOCK_FALSE);
9701                nfs_rw_exit(&rp->r_rwlock);
9702        }

9704        (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, FALSE);
9705        return (V_WRITELOCK_TRUE);
9706 }

9708 /* ARGSUSED */
9709 void
9710 nfs4_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
9711 {
9712        rnode4_t *rp = VTOR4(vp);

9714        nfs_rw_exit(&rp->r_rwlock);
9715 }

9717 /* ARGSUSED */
9718 static int
9719 nfs4_seek(vnode_t *vp, offset_t ooff, offset_t *noffp, caller_context_t *ct)
```



```
9720 {
9721        if (nfs_zone() != VTOMI4(vp)->mi_zone)
9722                return (EIO);

9724        /*
9725         * Because we stuff the readdir cookie into the offset field
9726         * someone may attempt to do an lseek with the cookie which
9727         * we want to succeed.
9728         */
9729        if (vp->v_type == VDIR)
9730                return (0);
9731        if (*noffp < 0)
9732                return (EINVAL);
9733        return (0);
9734 }


9737 /*
9738  * Return all the pages from [off..off+len) in file
9739  */
9740 /* ARGSUSED */
9741 static int
9742 nfs4_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
9743     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
9744     enum seg_rw rw, cred_t *cr, caller_context_t *ct)
9745 {
9746        rnode4_t *rp;
9747        int error;
9748        mntinfo4_t *mi;

9750        if (nfs_zone() != VTOMI4(vp)->mi_zone)
9751                return (EIO);
9752        rp = VTOR4(vp);
9753        if (IS_SHADOW(vp, rp))
9754                vp = RTOV4(rp);

9756        if (vp->v_flag & VNOMAP)
9757                return (ENOSYS);

9759        if (protp != NULL)
9760                *protp = PROT_ALL;

9762        /*
9763         * Now validate that the caches are up to date.
9764         */
9765        if (error = nfs4_validate_caches(vp, cr))
9766                return (error);

9768        mi = VTOMI4(vp);
9769 retry:
9770        mutex_enter(&rp->r_statelock);

9772        /*
9773         * Don't create dirty pages faster than they
9774         * can be cleaned so that the system doesn't
9775         * get imbalanced.  If the async queue is
9776         * maxed out, then wait for it to drain before
9777         * creating more dirty pages.  Also, wait for
9778         * any threads doing pagewalks in the vop_getattr
9779         * entry points so that they don't block for
9780         * long periods.
9781         */
9782        if (rw == S_CREATE) {
9783                while ((mi->mi_max_threads != 0 &&
9784                    rp->r_awcount > 2 * mi->mi_max_threads) ||
9785                    rp->r_gcount > 0)
```

```
9786                        cv_wait(&rp->r_cv, &rp->r_statelock);
9787        }

9789        /*
9790         * If we are getting called as a side effect of an nfs_write()
9791         * operation the local file size might not be extended yet.
9792         * In this case we want to be able to return pages of zeroes.
9793         */
9794        if (off + len > rp->r_size + PAGEOFFSET && seg != segkmap) {
9795                NFS4_DEBUG(nfs4_pageio_debug,
9796                    (CE_NOTE, "getpage beyond EOF: off=%lld, "
9797                    "len=%llu, size=%llu, attrsize =%llu", off,
9798                    (u_longlong_t)len, rp->r_size, rp->r_attr.va_size));
9799                mutex_exit(&rp->r_statelock);
9800                return (EFAULT);                /* beyond EOF */
9801        }

9803        mutex_exit(&rp->r_statelock);

9805        error = pvn_getpages(nfs4_getapage, vp, off, len, protp,
9806            pl, plsz, seg, addr, rw, cr);
9807        NFS4_DEBUG(nfs4_pageio_debug && error,
9808            (CE_NOTE, "getpages error %d; off=%lld, len=%lld",
9809            error, off, (u_longlong_t)len));

9811        switch (error) {
9812        case NFS_EOF:
9813                nfs4_purge_caches(vp, NFS4_NOPURGE_DNLC, cr, FALSE);
9814                goto retry;
9815        case ESTALE:
9816                nfs4_purge_stale_fh(error, vp, cr);
9817        }

9819        return (error);
9820 }

9822 /*
9823  * Called from pvn_getpages to get a particular page.
9824  */
9825 /* ARGSUSED */
9826 static int
9827 nfs4_getapage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protp,
9828     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
9829     enum seg_rw rw, cred_t *cr)
9830 {
9831        rnode4_t *rp;
9832        uint_t bsize;
9833        struct buf *bp;
9834        page_t *pp;
9835        u_offset_t lbn;
9836        u_offset_t io_off;
9837        u_offset_t blkoff;
9838        u_offset_t rablkoff;
9839        size_t io_len;
9840        uint_t blksize;
9841        int error;
9842        int readahead;
9843        int readahead_issued = 0;
9844        int ra_window; /* readahead window */
9845        page_t *pagefound;
9846        page_t *savepp;

9848        if (nfs_zone() != VTOMI4(vp)->mi_zone)
9849                return (EIO);

9851        rp = VTOR4(vp);
```



```
9852        ASSERT(!IS_SHADOW(vp, rp));
9853        bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

9855 reread:
9856        bp = NULL;
9857        pp = NULL;
9858        pagefound = NULL;

9860        if (pl != NULL)
9861                pl[0] = NULL;

9863        error = 0;
9864        lbn = off / bsize;
9865        blkoff = lbn * bsize;

9867        /*
9868         * Queueing up the readahead before doing the synchronous read
9869         * results in a significant increase in read throughput because
9870         * of the increased parallelism between the async threads and
9871         * the process context.
9872         */
9873        if ((off & ((vp->v_vfsp->vfs_bsize) - 1)) == 0 &&
9874            rw != S_CREATE &&
9875            !(vp->v_flag & VNOCACHE)) {
9876                mutex_enter(&rp->r_statelock);

9878                /*
9879                 * Calculate the number of readaheads to do.
9880                 * a) No readaheads at offset = 0.
9881                 * b) Do maximum(nfs4_nra) readaheads when the readahead
9882                 *    window is closed.
9883                 * c) Do readaheads between 1 to (nfs4_nra - 1) depending
9884                 *    upon how far the readahead window is open or close.
9885                 * d) No readaheads if rp->r_nextr is not within the scope
9886                 *    of the readahead window (random i/o).
9887                 */

9889                if (off == 0)
9890                        readahead = 0;
9891                else if (blkoff == rp->r_nextr)
9892                        readahead = nfs4_nra;
9893                else if (rp->r_nextr > blkoff &&
9894                    ((ra_window = (rp->r_nextr - blkoff) / bsize)
9895                    <= (nfs4_nra - 1)))
9896                        readahead = nfs4_nra - ra_window;
9897                else
9898                        readahead = 0;

9900                rablkoff = rp->r_nextr;
9901                while (readahead > 0 && rablkoff + bsize < rp->r_size) {
9902                        mutex_exit(&rp->r_statelock);
9903                        if (nfs4_async_readahead(vp, rablkoff + bsize,
9904                            addr + (rablkoff + bsize - off),
9905                            seg, cr, nfs4_readahead) < 0) {
9906                                mutex_enter(&rp->r_statelock);
9907                                break;
9908                        }
9909                        readahead--;
9910                        rablkoff += bsize;
9911                        /*
9912                         * Indicate that we did a readahead so
9913                         * readahead offset is not updated
9914                         * by the synchronous read below.
9915                         */
9916                        readahead_issued = 1;
9917                        mutex_enter(&rp->r_statelock);
```

```
9918                        /*
9919                         * set readahead offset to
9920                         * offset of last async readahead
9921                         * request.
9922                         */
9923                        rp->r_nextr = rablkoff;
9924                }
9925                mutex_exit(&rp->r_statelock);
9926        }

9928 again:
9929        if ((pagefound = page_exists(vp, off)) == NULL) {
9930                if (pl == NULL) {
9931                        (void) nfs4_async_readahead(vp, blkoff, addr, seg, cr,
9932                            nfs4_readahead);
9933                } else if (rw == S_CREATE) {
9934                        /*
9935                         * Block for this page is not allocated, or the offset
9936                         * is beyond the current allocation size, or we're
9937                         * allocating a swap slot and the page was not found,
9938                         * so allocate it and return a zero page.
9939                         */
9940                        if ((pp = page_create_va(vp, off,
9941                            PAGESIZE, PG_WAIT, seg, addr)) == NULL)
9942                                cmn_err(CE_PANIC, "nfs4_getapage: page_create");
9943                        io_len = PAGESIZE;
9944                        mutex_enter(&rp->r_statelock);
9945                        rp->r_nextr = off + PAGESIZE;
9946                        mutex_exit(&rp->r_statelock);
9947                } else {
9948                        /*
9949                         * Need to go to server to get a block
9950                         */
9951                        mutex_enter(&rp->r_statelock);
9952                        if (blkoff < rp->r_size &&
9953                            blkoff + bsize > rp->r_size) {
9954                                /*
9955                                 * If less than a block left in
9956                                 * file read less than a block.
9957                                 */
9958                                if (rp->r_size <= off) {
9959                                        /*
9960                                         * Trying to access beyond EOF,
9961                                         * set up to get at least one page.
9962                                         */
9963                                        blksize = off + PAGESIZE - blkoff;
9964                                } else
9965                                        blksize = rp->r_size - blkoff;
9966                        } else if ((off == 0) ||
9967                            (off != rp->r_nextr && !readahead_issued)) {
9968                                blksize = PAGESIZE;
9969                                blkoff = off; /* block = page here */
9970                        } else
9971                                blksize = bsize;
9972                        mutex_exit(&rp->r_statelock);

9974                        pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
9975                            &io_len, blkoff, blksize, 0);

9977                        /*
9978                         * Some other thread has entered the page,
9979                         * so just use it.
9980                         */
9981                        if (pp == NULL)
9982                                goto again;
```



```
9984                        /*
9985                         * Now round the request size up to page boundaries.
9986                         * This ensures that the entire page will be
9987                         * initialized to zeroes if EOF is encountered.
9988                         */
9989                        io_len = ptob(btopr(io_len));

9991                        bp = pageio_setup(pp, io_len, vp, B_READ);
9992                        ASSERT(bp != NULL);

9994                        /*
9995                         * pageio_setup should have set b_addr to 0.  This
9996                         * is correct since we want to do I/O on a page
9997                         * boundary.  bp_mapin will use this addr to calculate
9998                         * an offset, and then set b_addr to the kernel virtual
9999                         * address it allocated for us.
10000                         */
10001                        ASSERT(bp->b_un.b_addr == 0);

10003                        bp->b_edev = 0;
10004                        bp->b_dev = 0;
10005                        bp->b_lblkno = lbtodb(io_off);
10006                        bp->b_file = vp;
10007                        bp->b_offset = (offset_t)off;
10008                        bp_mapin(bp);

10010                        /*
10011                         * If doing a write beyond what we believe is EOF,
10012                         * don't bother trying to read the pages from the
10013                         * server, we'll just zero the pages here.  We
10014                         * don't check that the rw flag is S_WRITE here
10015                         * because some implementations may attempt a
10016                         * read access to the buffer before copying data.
10017                         */
10018                        mutex_enter(&rp->r_statelock);
10019                        if (io_off >= rp->r_size && seg == segkmap) {
10020                                mutex_exit(&rp->r_statelock);
10021                                bzero(bp->b_un.b_addr, io_len);
10022                        } else {
10023                                mutex_exit(&rp->r_statelock);
10024                                error = nfs4_bio(bp, NULL, cr, FALSE);
10025                        }

10027                        /*
10028                         * Unmap the buffer before freeing it.
10029                         */
10030                        bp_mapout(bp);
10031                        pageio_done(bp);

10033                        savepp = pp;
10034                        do {
10035                                pp->p_fsdata = C_NOCOMMIT;
10036                        } while ((pp = pp->p_next) != savepp);

10038                        if (error == NFS_EOF) {
10039                                /*
10040                                 * If doing a write system call just return
10041                                 * zeroed pages, else user tried to get pages
10042                                 * beyond EOF, return error.  We don't check
10043                                 * that the rw flag is S_WRITE here because
10044                                 * some implementations may attempt a read
10045                                 * access to the buffer before copying data.
10046                                 */
10047                                if (seg == segkmap)
10048                                        error = 0;
10049                                else
```

```
10050                                       error = EFAULT;
10051                           }
10052
10053                           if (!readahead_issued && !error) {
10054                                   mutex_enter(&rp->r_statelock);
10055                                   rp->r_nextr = io_off + io_len;
10056                                   mutex_exit(&rp->r_statelock);
10057                           }
10058                   }
10059           }
10060
10061   out:
10062           if (pl == NULL)
10063                   return (error);
10064
10065           if (error) {
10066                   if (pp != NULL)
10067                           pvn_read_done(pp, B_ERROR);
10068                   return (error);
10069           }
10070
10071           if (pagefound) {
10072                   se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);
10073
10074                   /*
10075                    * Page exists in the cache, acquire the appropriate lock.
10076                    * If this fails, start all over again.
10077                    */
10078                   if ((pp = page_lookup(vp, off, se)) == NULL) {
10079   #ifdef DEBUG
10080                           nfs4_lostpage++;
10081   #endif
10082                           goto reread;
10083                   }
10084                   pl[0] = pp;
10085                   pl[1] = NULL;
10086                   return (0);
10087           }
10088
10089           if (pp != NULL)
10090                   pvn_plist_init(pp, pl, plsz, off, io_len, rw);
10091
10092           return (error);
10093   }
10094
10095   static void
10096   nfs4_readahead(vnode_t *vp, u_offset_t blkoff, caddr_t addr, struct seg *seg,
10097       cred_t *cr)
10098   {
10099           int error;
10100           page_t *pp;
10101           u_offset_t io_off;
10102           size_t io_len;
10103           struct buf *bp;
10104           uint_t bsize, blksize;
10105           rnode4_t *rp = VTOR4(vp);
10106           page_t *savepp;
10107
10108           ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
10109
10110           bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
10111
10112           mutex_enter(&rp->r_statelock);
10113           if (blkoff < rp->r_size && blkoff + bsize > rp->r_size) {
10114                   /*
10115                    * If less than a block left in file read less
```



```
10116                    * than a block.
10117                    */
10118                   blksize = rp->r_size - blkoff;
10119           } else
10120                   blksize = bsize;
10121           mutex_exit(&rp->r_statelock);
10122
10123           pp = pvn_read_kluster(vp, blkoff, segkmap, addr,
10124               &io_off, &io_len, blkoff, blksize, 1);
10125           /*
10126            * The isra flag passed to the kluster function is 1, we may have
10127            * gotten a return value of NULL for a variety of reasons (# of free
10128            * pages < minfree, someone entered the page on the vnode etc). In all
10129            * cases, we want to punt on the readahead.
10130            */
10131           if (pp == NULL)
10132                   return;
10133
10134           /*
10135            * Now round the request size up to page boundaries.
10136            * This ensures that the entire page will be
10137            * initialized to zeroes if EOF is encountered.
10138            */
10139           io_len = ptob(btopr(io_len));
10140
10141           bp = pageio_setup(pp, io_len, vp, B_READ);
10142           ASSERT(bp != NULL);
10143
10144           /*
10145            * pageio_setup should have set b_addr to 0.  This is correct since
10146            * we want to do I/O on a page boundary. bp_mapin() will use this addr
10147            * to calculate an offset, and then set b_addr to the kernel virtual
10148            * address it allocated for us.
10149            */
10150           ASSERT(bp->b_un.b_addr == 0);
10151
10152           bp->b_edev = 0;
10153           bp->b_dev = 0;
10154           bp->b_lblkno = lbtodb(io_off);
10155           bp->b_file = vp;
10156           bp->b_offset = (offset_t)blkoff;
10157           bp_mapin(bp);
10158
10159           /*
10160            * If doing a write beyond what we believe is EOF, don't bother trying
10161            * to read the pages from the server, we'll just zero the pages here.
10162            * We don't check that the rw flag is S_WRITE here because some
10163            * implementations may attempt a read access to the buffer before
10164            * copying data.
10165            */
10166           mutex_enter(&rp->r_statelock);
10167           if (io_off >= rp->r_size && seg == segkmap) {
10168                   mutex_exit(&rp->r_statelock);
10169                   bzero(bp->b_un.b_addr, io_len);
10170                   error = 0;
10171           } else {
10172                   mutex_exit(&rp->r_statelock);
10173                   error = nfs4_bio(bp, NULL, cr, TRUE);
10174                   if (error == NFS_EOF)
10175                           error = 0;
10176           }
10177
10178           /*
10179            * Unmap the buffer before freeing it.
10180            */
10181           bp_mapout(bp);
```

```
10182         pageio_done(bp);

10184         savepp = pp;
10185         do {
10186                 pp->p_fsdata = C_NOCOMMIT;
10187         } while ((pp = pp->p_next) != savepp);

10189         pvn_read_done(pp, error ? B_READ | B_ERROR : B_READ);

10191         /*
10192          * In case of error set readahead offset
10193          * to the lowest offset.
10194          * pvn_read_done() calls VN_DISPOSE to destroy the pages
10195          */
10196         if (error && rp->r_nextr > io_off) {
10197                 mutex_enter(&rp->r_statelock);
10198                 if (rp->r_nextr > io_off)
10199                         rp->r_nextr = io_off;
10200                 mutex_exit(&rp->r_statelock);
10201         }
10202 }

10204 /*
10205  * Flags are composed of {B_INVAL, B_FREE, B_DONTNEED, B_FORCE}
10206  * If len == 0, do from off to EOF.
10207  *
10208  * The normal cases should be len == 0 && off == 0 (entire vp list) or
10209  * len == MAXBSIZE (from segmap_release actions), and len == PAGESIZE
10210  * (from pageout).
10211  */
10212 /* ARGSUSED */
10213 static int
10214 nfs4_putpage(vnode_t *vp, offset_t off, size_t len, int flags, cred_t *cr,
10215     caller_context_t *ct)
10216 {
10217         int error;
10218         rnode4_t *rp;

10220         ASSERT(cr != NULL);

10222         if (!(flags & B_ASYNC) && nfs_zone() != VTOMI4(vp)->mi_zone)
10223                 return (EIO);

10225         rp = VTOR4(vp);
10226         if (IS_SHADOW(vp, rp))
10227                 vp = RTOV4(rp);

10229         /*
10230          * XXX - Why should this check be made here?
10231          */
10232         if (vp->v_flag & VNOMAP)
10233                 return (ENOSYS);

10235         if (len == 0 && !(flags & B_INVAL) &&
10236             (vp->v_vfsp->vfs_flag & VFS_RDONLY))
10237                 return (0);

10239         mutex_enter(&rp->r_statelock);
10240         rp->r_count++;
10241         mutex_exit(&rp->r_statelock);
10242         error = nfs4_putpages(vp, off, len, flags, cr);
10243         mutex_enter(&rp->r_statelock);
10244         rp->r_count--;
10245         cv_broadcast(&rp->r_cv);
10246         mutex_exit(&rp->r_statelock);
```



```
10248         return (error);
10249 }

10251 /*
10252  * Write out a single page, possibly klustering adjacent dirty pages.
10253  */
10254 int
10255 nfs4_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *lenp,
10256     int flags, cred_t *cr)
10257 {
10258         u_offset_t io_off;
10259         u_offset_t lbn_off;
10260         u_offset_t lbn;
10261         size_t io_len;
10262         uint_t bsize;
10263         int error;
10264         rnode4_t *rp;

10266         ASSERT(!(vp->v_vfsp->vfs_flag & VFS_RDONLY));
10267         ASSERT(pp != NULL);
10268         ASSERT(cr != NULL);
10269         ASSERT((flags & B_ASYNC) || nfs_zone() == VTOMI4(vp)->mi_zone);

10271         rp = VTOR4(vp);
10272         ASSERT(rp->r_count > 0);
10273         ASSERT(!IS_SHADOW(vp, rp));

10275         bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
10276         lbn = pp->p_offset / bsize;
10277         lbn_off = lbn * bsize;

10279         /*
10280          * Find a kluster that fits in one block, or in
10281          * one page if pages are bigger than blocks.  If
10282          * there is less file space allocated than a whole
10283          * page, we'll shorten the i/o request below.
10284          */
10285         pp = pvn_write_kluster(vp, pp, &io_off, &io_len, lbn_off,
10286             roundup(bsize, PAGESIZE), flags);

10288         /*
10289          * pvn_write_kluster shouldn't have returned a page with offset
10290          * behind the original page we were given.  Verify that.
10291          */
10292         ASSERT((pp->p_offset / bsize) >= lbn);

10294         /*
10295          * Now pp will have the list of kept dirty pages marked for
10296          * write back.  It will also handle invalidation and freeing
10297          * of pages that are not dirty.  Check for page length rounding
10298          * problems.
10299          */
10300         if (io_off + io_len > lbn_off + bsize) {
10301                 ASSERT((io_off + io_len) - (lbn_off + bsize) < PAGESIZE);
10302                 io_len = lbn_off + bsize - io_off;
10303         }
10304         /*
10305          * The R4MODINPROGRESS flag makes sure that nfs4_bio() sees a
10306          * consistent value of r_size. R4MODINPROGRESS is set in writerp4().
10307          * When R4MODINPROGRESS is set it indicates that a uiomove() is in
10308          * progress and the r_size has not been made consistent with the
10309          * new size of the file. When the uiomove() completes the r_size is
10310          * updated and the R4MODINPROGRESS flag is cleared.
10311          *
10312          * The R4MODINPROGRESS flag makes sure that nfs4_bio() sees a
10313          * consistent value of r_size. Without this handshaking, it is
```

```
10314          * possible that nfs4_bio() picks  up the old value of r_size
10315          * before the uiomove() in writerp4() completes. This will result
10316          * in the write through nfs4_bio() being dropped.
10317          *
10318          * More precisely, there is a window between the time the uiomove()
10319          * completes and the time the r_size is updated. If a VOP_PUTPAGE()
10320          * operation intervenes in this window, the page will be picked up,
10321          * because it is dirty (it will be unlocked, unless it was
10322          * pagecreate'd). When the page is picked up as dirty, the dirty
10323          * bit is reset (pvn_getdirty()). In nfs4write(), r_size is
10324          * checked. This will still be the old size. Therefore the page will
10325          * not be written out. When segmap_release() calls VOP_PUTPAGE(),
10326          * the page will be found to be clean and the write will be dropped.
10327          */
10328         if (rp->r_flags & R4MODINPROGRESS) {
10329                 mutex_enter(&rp->r_statelock);
10330                 if ((rp->r_flags & R4MODINPROGRESS) &&
10331                     rp->r_modaddr + MAXBSIZE > io_off &&
10332                     rp->r_modaddr < io_off + io_len) {
10333                         page_t *plist;
10334                         /*
10335                          * A write is in progress for this region of the file.
10336                          * If we did not detect R4MODINPROGRESS here then this
10337                          * path through nfs_putapage() would eventually go to
10338                          * nfs4_bio() and may not write out all of the data
10339                          * in the pages. We end up losing data. So we decide
10340                          * to set the modified bit on each page in the page
10341                          * list and mark the rnode with R4DIRTY. This write
10342                          * will be restarted at some later time.
10343                          */
10344                         plist = pp;
10345                         while (plist != NULL) {
10346                                 pp = plist;
10347                                 page_sub(&plist, pp);
10348                                 hat_setmod(pp);
10349                                 page_io_unlock(pp);
10350                                 page_unlock(pp);
10351                         }
10352                         rp->r_flags |= R4DIRTY;
10353                         mutex_exit(&rp->r_statelock);
10354                         if (offp)
10355                                 *offp = io_off;
10356                         if (lenp)
10357                                 *lenp = io_len;
10358                         return (0);
10359                 }
10360                 mutex_exit(&rp->r_statelock);
10361         }

10363         if (flags & B_ASYNC) {
10364                 error = nfs4_async_putapage(vp, pp, io_off, io_len, flags, cr,
10365                     nfs4_sync_putapage);
10366         } else
10367                 error = nfs4_sync_putapage(vp, pp, io_off, io_len, flags, cr);

10369         if (offp)
10370                 *offp = io_off;
10371         if (lenp)
10372                 *lenp = io_len;
10373         return (error);
10374 }

10376 static int
10377 nfs4_sync_putapage(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
10378     int flags, cred_t *cr)
10379 {
```



```
10380         int error;
10381         rnode4_t *rp;

10383         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

10385         flags |= B_WRITE;

10387         error = nfs4_rdwrlbn(vp, pp, io_off, io_len, flags, cr);

10389         rp = VTOR4(vp);

10391         if ((error == ENOSPC || error == EDQUOT || error == EFBIG ||
10392             error == EACCES) &&
10393             (flags & (B_INVAL|B_FORCE)) != (B_INVAL|B_FORCE)) {
10394                 if (!(rp->r_flags & R4OUTOFSPACE)) {
10395                         mutex_enter(&rp->r_statelock);
10396                         rp->r_flags |= R4OUTOFSPACE;
10397                         mutex_exit(&rp->r_statelock);
10398                 }
10399                 flags |= B_ERROR;
10400                 pvn_write_done(pp, flags);
10401                 /*
10402                  * If this was not an async thread, then try again to
10403                  * write out the pages, but this time, also destroy
10404                  * them whether or not the write is successful.  This
10405                  * will prevent memory from filling up with these
10406                  * pages and destroying them is the only alternative
10407                  * if they can't be written out.
10408                  *
10409                  * Don't do this if this is an async thread because
10410                  * when the pages are unlocked in pvn_write_done,
10411                  * some other thread could have come along, locked
10412                  * them, and queued for an async thread.  It would be
10413                  * possible for all of the async threads to be tied
10414                  * up waiting to lock the pages again and they would
10415                  * all already be locked and waiting for an async
10416                  * thread to handle them.  Deadlock.
10417                  */
10418                 if (!(flags & B_ASYNC)) {
10419                         error = nfs4_putpage(vp, io_off, io_len,
10420                             B_INVAL | B_FORCE, cr, NULL);
10421                 }
10422         } else {
10423                 if (error)
10424                         flags |= B_ERROR;
10425                 else if (rp->r_flags & R4OUTOFSPACE) {
10426                         mutex_enter(&rp->r_statelock);
10427                         rp->r_flags &= ~R4OUTOFSPACE;
10428                         mutex_exit(&rp->r_statelock);
10429                 }
10430                 pvn_write_done(pp, flags);
10431                 if (freemem < desfree)
10432                         (void) nfs4_commit_vp(vp, (u_offset_t)0, 0, cr,
10433                             NFS4_WRITE_NOWAIT);
10434         }

10436         return (error);
10437 }

10439 #ifdef DEBUG
10440 int nfs4_force_open_before_mmap = 0;
10441 #endif

10443 /* ARGSUSED */
10444 static int
10445 nfs4_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
```

```
10446     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
10447     caller_context_t *ct)
10448 {
10449         struct segvn_crargs vn_a;
10450         int error = 0;
10451         rnode4_t *rp = VTOR4(vp);
10452         mntinfo4_t *mi = VTOMI4(vp);

10454         if (nfs_zone() != VTOMI4(vp)->mi_zone)
10455                 return (EIO);

10457         if (vp->v_flag & VNOMAP)
10458                 return (ENOSYS);

10460         if (off < 0 || (off + len) < 0)
10461                 return (ENXIO);

10463         if (vp->v_type != VREG)
10464                 return (ENODEV);

10466         /*
10467          * If the file is delegated to the client don't do anything.
10468          * If the file is not delegated, then validate the data cache.
10469          */
10470         mutex_enter(&rp->r_statev4_lock);
10471         if (rp->r_deleg_type == OPEN_DELEGATE_NONE) {
10472                 mutex_exit(&rp->r_statev4_lock);
10473                 error = nfs4_validate_caches(vp, cr);
10474                 if (error)
10475                         return (error);
10476         } else {
10477                 mutex_exit(&rp->r_statev4_lock);
10478         }

10480         /*
10481          * Check to see if the vnode is currently marked as not cachable.
10482          * This means portions of the file are locked (through VOP_FRLOCK).
10483          * In this case the map request must be refused.  We use
10484          * rp->r_lkserlock to avoid a race with concurrent lock requests.
10485          *
10486          * Atomically increment r_inmap after acquiring r_rwlock. The
10487          * idea here is to acquire r_rwlock to block read/write and
10488          * not to protect r_inmap. r_inmap will inform nfs4_read/write()
10489          * that we are in nfs4_map(). Now, r_rwlock is acquired in order
10490          * and we can prevent the deadlock that would have occurred
10491          * when nfs4_addmap() would have acquired it out of order.
10492          *
10493          * Since we are not protecting r_inmap by any lock, we do not
10494          * hold any lock when we decrement it. We atomically decrement
10495          * r_inmap after we release r_lkserlock.
10496          */

10498         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, INTR4(vp)))
10499                 return (EINTR);
10500         atomic_inc_uint(&rp->r_inmap);
10501         nfs_rw_exit(&rp->r_rwlock);

10503         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR4(vp))) {
10504                 atomic_dec_uint(&rp->r_inmap);
10505                 return (EINTR);
10506         }


10509         if (vp->v_flag & VNOCACHE) {
10510                 error = EAGAIN;
10511                 goto done;
```



```
10512         }

10514         /*
10515          * Don't allow concurrent locks and mapping if mandatory locking is
10516          * enabled.
10517          */
10518         if (flk_has_remote_locks(vp)) {
10519                 struct vattr va;
10520                 va.va_mask = AT_MODE;
10521                 error = nfs4getattr(vp, &va, cr);
10522                 if (error != 0)
10523                         goto done;
10524                 if (MANDLOCK(vp, va.va_mode)) {
10525                         error = EAGAIN;
10526                         goto done;
10527                 }
10528         }

10530         /*
10531          * It is possible that the rnode has a lost lock request that we
10532          * are still trying to recover, and that the request conflicts with
10533          * this map request.
10534          *
10535          * An alternative approach would be for nfs4_safemap() to consider
10536          * queued lock requests when deciding whether to set or clear
10537          * VNOCACHE.  This would require the frlock code path to call
10538          * nfs4_safemap() after enqueing a lost request.
10539          */
10540         if (nfs4_map_lost_lock_conflict(vp)) {
10541                 error = EAGAIN;
10542                 goto done;
10543         }

10545         as_rangelock(as);
10546         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
10547         if (error != 0) {
10548                 as_rangeunlock(as);
10549                 goto done;
10550         }

10552         if (vp->v_type == VREG) {
10553                 /*
10554                  * We need to retrieve the open stream
10555                  */
10556                 nfs4_open_stream_t      *osp = NULL;
10557                 nfs4_open_owner_t       *oop = NULL;

10559                 oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
10560                 if (oop != NULL) {
10561                         /* returns with 'os_sync_lock' held */
10562                         osp = find_open_stream(oop, rp);
10563                         open_owner_rele(oop);
10564                 }
10565                 if (osp == NULL) {
10566 #ifdef DEBUG
10567                         if (nfs4_force_open_before_mmap) {
10568                                 error = EIO;
10569                                 goto done;
10570                         }
10571 #endif
10572                         /* returns with 'os_sync_lock' held */
10573                         error = open_and_get_osp(vp, cr, &osp);
10574                         if (osp == NULL) {
10575                                 NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE,
10576                                     "nfs4_map: we tried to OPEN the file "
10577                                     "but again no osp, so fail with EIO"));
```

```
10578                                   goto done;
10579                           }
10580                   }

10582                   if (osp->os_failed_reopen) {
10583                           mutex_exit(&osp->os_sync_lock);
10584                           open_stream_rele(osp, rp);
10585                           NFS4_DEBUG(nfs4_open_stream_debug, (CE_NOTE,
10586                               "nfs4_map: os_failed_reopen set on "
10587                               "osp %p, cr %p, rp %s", (void *)osp,
10588                               (void *)cr, rnode4info(rp)));
10589                           error = EIO;
10590                           goto done;
10591                   }
10592                   mutex_exit(&osp->os_sync_lock);
10593                   open_stream_rele(osp, rp);
10594           }

10596           vn_a.vp = vp;
10597           vn_a.offset = off;
10598           vn_a.type = (flags & MAP_TYPE);
10599           vn_a.prot = (uchar_t)prot;
10600           vn_a.maxprot = (uchar_t)maxprot;
10601           vn_a.flags = (flags & ~MAP_TYPE);
10602           vn_a.cred = cr;
10603           vn_a.amp = NULL;
10604           vn_a.szc = 0;
10605           vn_a.lgrp_mem_policy_flags = 0;

10607           error = as_map(as, *addrp, len, segvn_create, &vn_a);
10608           as_rangeunlock(as);

10610   done:
10611           nfs_rw_exit(&rp->r_lkserlock);
10612           atomic_dec_uint(&rp->r_inmap);
10613           return (error);
10614   }

10616   /*
10617    * We're most likely dealing with a kernel module that likes to READ
10618    * and mmap without OPENing the file (ie: lookup/read/mmap), so lets
10619    * officially OPEN the file to create the necessary client state
10620    * for bookkeeping of os_mmap_read/write counts.
10621    *
10622    * Since VOP_MAP only passes in a pointer to the vnode rather than
10623    * a double pointer, we can't handle the case where nfs4open_otw()
10624    * returns a different vnode than the one passed into VOP_MAP (since
10625    * VOP_DELMAP will not see the vnode nfs4open_otw used).  In this case,
10626    * we return NULL and let nfs4_map() fail.  Note: the only case where
10627    * this should happen is if the file got removed and replaced with the
10628    * same name on the server (in addition to the fact that we're trying
10629    * to VOP_MAP withouth VOP_OPENing the file in the first place).
10630    */
10631   static int
10632   open_and_get_osp(vnode_t *map_vp, cred_t *cr, nfs4_open_stream_t **ospp)
10633   {
10634           rnode4_t                *rp, *drp;
10635           vnode_t                 *dvp, *open_vp;
10636           char                    file_name[MAXNAMELEN];
10637           int                     just_created;
10638           nfs4_open_stream_t      *osp;
10639           nfs4_open_owner_t       *oop;
10640           int                     error;

10642           *ospp = NULL;
10643           open_vp = map_vp;
```



```
10645           rp = VTOR4(open_vp);
10646           if ((error = vtodv(open_vp, &dvp, cr, TRUE)) != 0)
10647                   return (error);
10648           drp = VTOR4(dvp);

10650           if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp))) {
10651                   VN_RELE(dvp);
10652                   return (EINTR);
10653           }

10655           if ((error = vtoname(open_vp, file_name, MAXNAMELEN)) != 0) {
10656                   nfs_rw_exit(&drp->r_rwlock);
10657                   VN_RELE(dvp);
10658                   return (error);
10659           }

10661           mutex_enter(&rp->r_statev4_lock);
10662           if (rp->created_v4) {
10663                   rp->created_v4 = 0;
10664                   mutex_exit(&rp->r_statev4_lock);

10666                   dnlc_update(dvp, file_name, open_vp);
10667                   /* This is needed so we don't bump the open ref count */
10668                   just_created = 1;
10669           } else {
10670                   mutex_exit(&rp->r_statev4_lock);
10671                   just_created = 0;
10672           }

10674           VN_HOLD(map_vp);

10676           error = nfs4open_otw(dvp, file_name, NULL, &open_vp, cr, 0, FREAD, 0,
10677               just_created);
10678           if (error) {
10679                   nfs_rw_exit(&drp->r_rwlock);
10680                   VN_RELE(dvp);
10681                   VN_RELE(map_vp);
10682                   return (error);
10683           }

10685           nfs_rw_exit(&drp->r_rwlock);
10686           VN_RELE(dvp);

10688           /*
10689            * If nfs4open_otw() returned a different vnode then "undo"
10690            * the open and return failure to the caller.
10691            */
10692           if (!VN_CMP(open_vp, map_vp)) {
10693                   nfs4_error_t e;

10695                   NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE, "open_and_get_osp: "
10696                       "open returned a different vnode"));
10697                   /*
10698                    * If there's an error, ignore it,
10699                    * and let VOP_INACTIVE handle it.
10700                    */
10701                   (void) nfs4close_one(open_vp, NULL, cr, FREAD, NULL, &e,
10702                       CLOSE_NORM, 0, 0, 0);
10703                   VN_RELE(map_vp);
10704                   return (EIO);
10705           }

10707           VN_RELE(map_vp);

10709           oop = find_open_owner(cr, NFS4_PERM_CREATED, VTOMI4(open_vp));
```

```
10710         if (!oop) {
10711                 nfs4_error_t e;

10713                 NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE, "open_and_get_osp: "
10714                     "no open owner"));
10715                 /*
10716                  * If there's an error, ignore it,
10717                  * and let VOP_INACTIVE handle it.
10718                  */
10719                 (void) nfs4close_one(open_vp, NULL, cr, FREAD, NULL, &e,
10720                     CLOSE_NORM, 0, 0, 0);
10721                 return (EIO);
10722         }
10723         osp = find_open_stream(oop, rp);
10724         open_owner_rele(oop);
10725         *ospp = osp;
10726         return (0);
10727 }

10729 /*
10730  * Please be aware that when this function is called, the address space write
10731  * a_lock is held.  Do not put over the wire calls in this function.
10732  */
10733 /* ARGSUSED */
10734 static int
10735 nfs4_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
10736     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
10737     caller_context_t *ct)
10738 {
10739         rnode4_t                *rp;
10740         int                     error = 0;
10741         mntinfo4_t              *mi;

10743         mi = VTOMI4(vp);
10744         rp = VTOR4(vp);

10746         if (nfs_zone() != mi->mi_zone)
10747                 return (EIO);
10748         if (vp->v_flag & VNOMAP)
10749                 return (ENOSYS);

10751         /*
10752          * Don't need to update the open stream first, since this
10753          * mmap can't add any additional share access that isn't
10754          * already contained in the open stream (for the case where we
10755          * open/mmap/only update rp->r_mapcnt/server reboots/reopen doesn't
10756          * take into account os_mmap_read[write] counts).
10757          */
10758         atomic_add_long((ulong_t *)&rp->r_mapcnt, btopr(len));

10760         if (vp->v_type == VREG) {
10761                 /*
10762                  * We need to retrieve the open stream and update the counts.
10763                  * If there is no open stream here, something is wrong.
10764                  */
10765                 nfs4_open_stream_t      *osp = NULL;
10766                 nfs4_open_owner_t       *oop = NULL;

10768                 oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
10769                 if (oop != NULL) {
10770                         /* returns with 'os_sync_lock' held */
10771                         osp = find_open_stream(oop, rp);
10772                         open_owner_rele(oop);
10773                 }
10774                 if (osp == NULL) {
10775                         NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE,
```



```
10776                             "nfs4_addmap: we should have an osp"
10777                             "but we don't, so fail with EIO"));
10778                         error = EIO;
10779                         goto out;
10780                 }

10782                 NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE, "nfs4_addmap: osp %p,"
10783                     " pages %ld, prot 0x%x", (void *)osp, btopr(len), prot));

10785                 /*
10786                  * Update the map count in the open stream.
10787                  * This is necessary in the case where we
10788                  * open/mmap/close/, then the server reboots, and we
10789                  * attempt to reopen.  If the mmap doesn't add share
10790                  * access then we send an invalid reopen with
10791                  * access = NONE.
10792                  *
10793                  * We need to specifically check each PROT_* so a mmap
10794                  * call of (PROT_WRITE | PROT_EXEC) will ensure us both
10795                  * read and write access.  A simple comparison of prot
10796                  * to ~PROT_WRITE to determine read access is insufficient
10797                  * since prot can be |= with PROT_USER, etc.
10798                  */

10800                 /*
10801                  * Unless we're MAP_SHARED, no sense in adding os_mmap_write
10802                  */
10803                 if ((flags & MAP_SHARED) && (maxprot & PROT_WRITE))
10804                         osp->os_mmap_write += btopr(len);
10805                 if (maxprot & PROT_READ)
10806                         osp->os_mmap_read += btopr(len);
10807                 if (maxprot & PROT_EXEC)
10808                         osp->os_mmap_read += btopr(len);
10809                 /*
10810                  * Ensure that os_mmap_read gets incremented, even if
10811                  * maxprot were to look like PROT_NONE.
10812                  */
10813                 if (!(maxprot & PROT_READ) && !(maxprot & PROT_WRITE) &&
10814                     !(maxprot & PROT_EXEC))
10815                         osp->os_mmap_read += btopr(len);
10816                 osp->os_mapcnt += btopr(len);
10817                 mutex_exit(&osp->os_sync_lock);
10818                 open_stream_rele(osp, rp);
10819         }

10821 out:
10822         /*
10823          * If we got an error, then undo our
10824          * incrementing of 'r_mapcnt'.
10825          */

10827         if (error) {
10828                 atomic_add_long((ulong_t *)&rp->r_mapcnt, -btopr(len));
10829                 ASSERT(rp->r_mapcnt >= 0);
10830         }
10831         return (error);
10832 }

10834 /* ARGSUSED */
10835 static int
10836 nfs4_cmp(vnode_t *vp1, vnode_t *vp2, caller_context_t *ct)
10837 {

10839         return (VTOR4(vp1) == VTOR4(vp2));
10840 }
```

```
10842 /* ARGSUSED */
10843 static int
10844 nfs4_frlock(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
10845     offset_t offset, struct flk_callback *flk_cbp, cred_t *cr,
10846     caller_context_t *ct)
10847 {
10848         int rc;
10849         u_offset_t start, end;
10850         rnode4_t *rp;
10851         int error = 0, intr = INTR4(vp);
10852         nfs4_error_t e;

10854         if (nfs_zone() != VTOMI4(vp)->mi_zone)
10855                 return (EIO);

10857         /* check for valid cmd parameter */
10858         if (cmd != F_GETLK && cmd != F_SETLK && cmd != F_SETLKW)
10859                 return (EINVAL);

10861         /* Verify l_type. */
10862         switch (bfp->l_type) {
10863         case F_RDLCK:
10864                 if (cmd != F_GETLK && !(flag & FREAD))
10865                         return (EBADF);
10866                 break;
10867         case F_WRLCK:
10868                 if (cmd != F_GETLK && !(flag & FWRITE))
10869                         return (EBADF);
10870                 break;
10871         case F_UNLCK:
10872                 intr = 0;
10873                 break;

10875         default:
10876                 return (EINVAL);
10877         }

10879         /* check the validity of the lock range */
10880         if (rc = flk_convert_lock_data(vp, bfp, &start, &end, offset))
10881                 return (rc);
10882         if (rc = flk_check_lock_data(start, end, MAXEND))
10883                 return (rc);

10885         /*
10886          * If the filesystem is mounted using local locking, pass the
10887          * request off to the local locking code.
10888          */
10889         if (VTOMI4(vp)->mi_flags & MI4_LLOCK || vp->v_type != VREG) {
10890                 if (cmd == F_SETLK || cmd == F_SETLKW) {
10891                         /*
10892                          * For complete safety, we should be holding
10893                          * r_lkserlock.  However, we can't call
10894                          * nfs4_safelock and then fs_frlock while
10895                          * holding r_lkserlock, so just invoke
10896                          * nfs4_safelock and expect that this will
10897                          * catch enough of the cases.
10898                          */
10899                         if (!nfs4_safelock(vp, bfp, cr))
10900                                 return (EAGAIN);
10901                 }
10902                 return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
10903         }

10905         rp = VTOR4(vp);

10907         /*
```



```
10908          * Check whether the given lock request can proceed, given the
10909          * current file mappings.
10910          */
10911         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_WRITER, intr))
10912                 return (EINTR);
10913         if (cmd == F_SETLK || cmd == F_SETLKW) {
10914                 if (!nfs4_safelock(vp, bfp, cr)) {
10915                         rc = EAGAIN;
10916                         goto done;
10917                 }
10918         }

10920         /*
10921          * Flush the cache after waiting for async I/O to finish.  For new
10922          * locks, this is so that the process gets the latest bits from the
10923          * server.  For unlocks, this is so that other clients see the
10924          * latest bits once the file has been unlocked.  If currently dirty
10925          * pages can't be flushed, then don't allow a lock to be set.  But
10926          * allow unlocks to succeed, to avoid having orphan locks on the
10927          * server.
10928          */
10929         if (cmd != F_GETLK) {
10930                 mutex_enter(&rp->r_statelock);
10931                 while (rp->r_count > 0) {
10932                         if (intr) {
10933                                 klwp_t *lwp = ttolwp(curthread);

10935                                 if (lwp != NULL)
10936                                         lwp->lwp_nostop++;
10937                                 if (cv_wait_sig(&rp->r_cv,
10938                                     &rp->r_statelock) == 0) {
10939                                         if (lwp != NULL)
10940                                                 lwp->lwp_nostop--;
10941                                         rc = EINTR;
10942                                         break;
10943                                 }
10944                                 if (lwp != NULL)
10945                                         lwp->lwp_nostop--;
10946                                 } else
10947                                         cv_wait(&rp->r_cv, &rp->r_statelock);
10948                 }
10949                 mutex_exit(&rp->r_statelock);
10950                 if (rc != 0)
10951                         goto done;
10952                 error = nfs4_putpage(vp, (offset_t)0, 0, B_INVAL, cr, ct);
10953                 if (error) {
10954                         if (error == ENOSPC || error == EDQUOT) {
10955                                 mutex_enter(&rp->r_statelock);
10956                                 if (!rp->r_error)
10957                                         rp->r_error = error;
10958                                 mutex_exit(&rp->r_statelock);
10959                         }
10960                         if (bfp->l_type != F_UNLCK) {
10961                                 rc = ENOLCK;
10962                                 goto done;
10963                         }
10964                 }
10965         }

10967         /*
10968          * Call the lock manager to do the real work of contacting
10969          * the server and obtaining the lock.
10970          */
10971         nfs4frlock(NFS4_LCK_CTYPE_NORM, vp, cmd, bfp, flag, offset,
10972             cr, &e, NULL, NULL);
10973         rc = e.error;
```

```
10975         if (rc == 0)
10976                 nfs4_lockcompletion(vp, cmd);
10977
10978 done:
10979         nfs_rw_exit(&rp->r_lkserlock);
10980
10981         return (rc);
10982 }
10983
10984 /*
10985  * Free storage space associated with the specified vnode.  The portion
10986  * to be freed is specified by bfp->l_start and bfp->l_len (already
10987  * normalized to a "whence" of 0).
10988  *
10989  * This is an experimental facility whose continued existence is not
10990  * guaranteed.  Currently, we only support the special case
10991  * of l_len == 0, meaning free to end of file.
10992  */
10993 /* ARGSUSED */
10994 static int
10995 nfs4_space(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
10996     offset_t offset, cred_t *cr, caller_context_t *ct)
10997 {
10998         int error;
10999
11000         if (nfs_zone() != VTOMI4(vp)->mi_zone)
11001                 return (EIO);
11002         ASSERT(vp->v_type == VREG);
11003         if (cmd != F_FREESP)
11004                 return (EINVAL);
11005
11006         error = convoff(vp, bfp, 0, offset);
11007         if (!error) {
11008                 ASSERT(bfp->l_start >= 0);
11009                 if (bfp->l_len == 0) {
11010                         struct vattr va;
11011
11012                         va.va_mask = AT_SIZE;
11013                         va.va_size = bfp->l_start;
11014                         error = nfs4setattr(vp, &va, 0, cr, NULL);
11015
11016                         if (error == 0 && bfp->l_start == 0)
11017                                 vnevent_truncate(vp, ct);
11018                 } else
11019                         error = EINVAL;
11020         }
11021
11022         return (error);
11023 }
11024
11025 /* ARGSUSED */
11026 int
11027 nfs4_realvp(vnode_t *vp, vnode_t **vpp, caller_context_t *ct)
11028 {
11029         rnode4_t *rp;
11030         rp = VTOR4(vp);
11031
11032         if (vp->v_type == VREG && IS_SHADOW(vp, rp)) {
11033                 vp = RTOV4(rp);
11034         }
11035         *vpp = vp;
11036         return (0);
11037 }
11038
11039 /*
```



```
11040  * Setup and add an address space callback to do the work of the delmap call.
11041  * The callback will (and must be) deleted in the actual callback function.
11042  *
11043  * This is done in order to take care of the problem that we have with holding
11044  * the address space's a_lock for a long period of time (e.g. if the NFS server
11045  * is down).  Callbacks will be executed in the address space code while the
11046  * a_lock is not held.  Holding the address space's a_lock causes things such
11047  * as ps and fork to hang because they are trying to acquire this lock as well.
11048  */
11049 /* ARGSUSED */
11050 static int
11051 nfs4_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
11052     size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
11053     caller_context_t *ct)
11054 {
11055         int                     caller_found;
11056         int                     error;
11057         rnode4_t                *rp;
11058         nfs4_delmap_args_t      *dmapp;
11059         nfs4_delmapcall_t       *delmap_call;
11060
11061         if (vp->v_flag & VNOMAP)
11062                 return (ENOSYS);
11063
11064         /*
11065          * A process may not change zones if it has NFS pages mmap'ed
11066          * in, so we can't legitimately get here from the wrong zone.
11067          */
11068         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
11069
11070         rp = VTOR4(vp);
11071
11072         /*
11073          * The way that the address space of this process deletes its mapping
11074          * of this file is via the following call chains:
11075          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11076          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11077          *
11078          * With the use of address space callbacks we are allowed to drop the
11079          * address space lock, a_lock, while executing the NFS operations that
11080          * need to go over the wire.  Returning EAGAIN to the caller of this
11081          * function is what drives the execution of the callback that we add
11082          * below.  The callback will be executed by the address space code
11083          * after dropping the a_lock.  When the callback is finished, since
11084          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
11085          * is called again on the same segment to finish the rest of the work
11086          * that needs to happen during unmapping.
11087          *
11088          * This action of calling back into the segment driver causes
11089          * nfs4_delmap() to get called again, but since the callback was
11090          * already executed at this point, it already did the work and there
11091          * is nothing left for us to do.
11092          *
11093          * To Summarize:
11094          * - The first time nfs4_delmap is called by the current thread is when
11095          * we add the caller associated with this delmap to the delmap caller
11096          * list, add the callback, and return EAGAIN.
11097          * - The second time in this call chain when nfs4_delmap is called we
11098          * will find this caller in the delmap caller list and realize there
11099          * is no more work to do thus removing this caller from the list and
11100          * returning the error that was set in the callback execution.
11101          */
11102         caller_found = nfs4_find_and_delete_delmapcall(rp, &error);
11103         if (caller_found) {
11104                 /*
11105                  * 'error' is from the actual delmap operations.  To avoid
```

```
11106                  * hangs, we need to handle the return of EAGAIN differently
11107                  * since this is what drives the callback execution.
11108                  * In this case, we don't want to return EAGAIN and do the
11109                  * callback execution because there are none to execute.
11110                  */
11111                 if (error == EAGAIN)
11112                         return (0);
11113                 else
11114                         return (error);
11115         }

11117         /* current caller was not in the list */
11118         delmap_call = nfs4_init_delmapcall();

11120         mutex_enter(&rp->r_statelock);
11121         list_insert_tail(&rp->r_indelmap, delmap_call);
11122         mutex_exit(&rp->r_statelock);

11124         dmapp = kmem_alloc(sizeof (nfs4_delmap_args_t), KM_SLEEP);

11126         dmapp->vp = vp;
11127         dmapp->off = off;
11128         dmapp->addr = addr;
11129         dmapp->len = len;
11130         dmapp->prot = prot;
11131         dmapp->maxprot = maxprot;
11132         dmapp->flags = flags;
11133         dmapp->cr = cr;
11134         dmapp->caller = delmap_call;

11136         error = as_add_callback(as, nfs4_delmap_callback, dmapp,
11137             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

11139         return (error ? error : EAGAIN);
11140 }

11142 static nfs4_delmapcall_t *
11143 nfs4_init_delmapcall()
11144 {
11145         nfs4_delmapcall_t       *delmap_call;

11147         delmap_call = kmem_alloc(sizeof (nfs4_delmapcall_t), KM_SLEEP);
11148         delmap_call->call_id = curthread;
11149         delmap_call->error = 0;

11151         return (delmap_call);
11152 }

11154 static void
11155 nfs4_free_delmapcall(nfs4_delmapcall_t *delmap_call)
11156 {
11157         kmem_free(delmap_call, sizeof (nfs4_delmapcall_t));
11158 }

11160 /*
11161  * Searches for the current delmap caller (based on curthread) in the list of
11162  * callers.  If it is found, we remove it and free the delmap caller.
11163  * Returns:
11164  *      0 if the caller wasn't found
11165  *      1 if the caller was found, removed and freed.  *errp will be set
11166  *      to what the result of the delmap was.
11167  */
11168 static int
11169 nfs4_find_and_delete_delmapcall(rnode4_t *rp, int *errp)
11170 {
11171         nfs4_delmapcall_t       *delmap_call;
```



```
11173         /*
11174          * If the list doesn't exist yet, we create it and return
11175          * that the caller wasn't found.  No list = no callers.
11176          */
11177         mutex_enter(&rp->r_statelock);
11178         if (!(rp->r_flags & R4DELMAPLIST)) {
11179                 /* The list does not exist */
11180                 list_create(&rp->r_indelmap, sizeof (nfs4_delmapcall_t),
11181                     offsetof(nfs4_delmapcall_t, call_node));
11182                 rp->r_flags |= R4DELMAPLIST;
11183                 mutex_exit(&rp->r_statelock);
11184                 return (0);
11185         } else {
11186                 /* The list exists so search it */
11187                 for (delmap_call = list_head(&rp->r_indelmap);
11188                     delmap_call != NULL;
11189                     delmap_call = list_next(&rp->r_indelmap, delmap_call)) {
11190                         if (delmap_call->call_id == curthread) {
11191                                 /* current caller is in the list */
11192                                 *errp = delmap_call->error;
11193                                 list_remove(&rp->r_indelmap, delmap_call);
11194                                 mutex_exit(&rp->r_statelock);
11195                                 nfs4_free_delmapcall(delmap_call);
11196                                 return (1);
11197                         }
11198                 }
11199         }
11200         mutex_exit(&rp->r_statelock);
11201         return (0);
11202 }

11204 /*
11205  * Remove some pages from an mmap'd vnode.  Just update the
11206  * count of pages.  If doing close-to-open, then flush and
11207  * commit all of the pages associated with this file.
11208  * Otherwise, start an asynchronous page flush to write out
11209  * any dirty pages.  This will also associate a credential
11210  * with the rnode which can be used to write the pages.
11211  */
11212 /* ARGSUSED */
11213 static void
11214 nfs4_delmap_callback(struct as *as, void *arg, uint_t event)
11215 {
11216         nfs4_error_t            e = { 0, NFS4_OK, RPC_SUCCESS };
11217         rnode4_t                *rp;
11218         mntinfo4_t              *mi;
11219         nfs4_delmap_args_t      *dmapp = (nfs4_delmap_args_t *)arg;

11221         rp = VTOR4(dmapp->vp);
11222         mi = VTOMI4(dmapp->vp);

11224         atomic_add_long((ulong_t *)&rp->r_mapcnt, -btopr(dmapp->len));
11225         ASSERT(rp->r_mapcnt >= 0);

11227         /*
11228          * Initiate a page flush and potential commit if there are
11229          * pages, the file system was not mounted readonly, the segment
11230          * was mapped shared, and the pages themselves were writeable.
11231          */
11232         if (nfs4_has_pages(dmapp->vp) &&
11233             !(dmapp->vp->v_vfsp->vfs_flag & VFS_RDONLY) &&
11234             dmapp->flags == MAP_SHARED && (dmapp->maxprot & PROT_WRITE)) {
11235                 mutex_enter(&rp->r_statelock);
11236                 rp->r_flags |= R4DIRTY;
11237                 mutex_exit(&rp->r_statelock);
```

```
11238                 e.error = nfs4_putpage_commit(dmapp->vp, dmapp->off,
11239                     dmapp->len, dmapp->cr);
11240                 if (!e.error) {
11241                         mutex_enter(&rp->r_statelock);
11242                         e.error = rp->r_error;
11243                         rp->r_error = 0;
11244                         mutex_exit(&rp->r_statelock);
11245                 }
11246         } else
11247                 e.error = 0;

11249         if ((rp->r_flags & R4DIRECTIO) || (mi->mi_flags & MI4_DIRECTIO))
11250                 (void) nfs4_putpage(dmapp->vp, dmapp->off, dmapp->len,
11251                     B_INVAL, dmapp->cr, NULL);

11253         if (e.error) {
11254                 e.stat = puterrno4(e.error);
11255                 nfs4_queue_fact(RF_DELMAP_CB_ERR, mi, e.stat, 0,
11256                     OP_COMMIT, FALSE, NULL, 0, dmapp->vp);
11257                 dmapp->caller->error = e.error;
11258         }

11260         /* Check to see if we need to close the file */

11262         if (dmapp->vp->v_type == VREG) {
11263                 nfs4close_one(dmapp->vp, NULL, dmapp->cr, 0, NULL, &e,
11264                     CLOSE_DELMAP, dmapp->len, dmapp->maxprot, dmapp->flags);

11266                 if (e.error != 0 || e.stat != NFS4_OK) {
11267                         /*
11268                          * Since it is possible that e.error == 0 and
11269                          * e.stat != NFS4_OK (and vice versa),
11270                          * we do the proper checking in order to get both
11271                          * e.error and e.stat reporting the correct info.
11272                          */
11273                         if (e.stat == NFS4_OK)
11274                                 e.stat = puterrno4(e.error);
11275                         if (e.error == 0)
11276                                 e.error = geterrno4(e.stat);

11278                         nfs4_queue_fact(RF_DELMAP_CB_ERR, mi, e.stat, 0,
11279                             OP_CLOSE, FALSE, NULL, 0, dmapp->vp);
11280                         dmapp->caller->error = e.error;
11281                 }
11282         }

11284         (void) as_delete_callback(as, arg);
11285         kmem_free(dmapp, sizeof (nfs4_delmap_args_t));
11286 }


11289 static uint_t
11290 fattr4_maxfilesize_to_bits(uint64_t ll)
11291 {
11292         uint_t l = 1;

11294         if (ll == 0) {
11295                 return (0);
11296         }

11298         if (ll & 0xffffffff00000000) {
11299                 l += 32; ll >>= 32;
11300         }
11301         if (ll & 0xffff0000) {
11302                 l += 16; ll >>= 16;
11303         }
```



```
11304         if (ll & 0xff00) {
11305                 l += 8; ll >>= 8;
11306         }
11307         if (ll & 0xf0) {
11308                 l += 4; ll >>= 4;
11309         }
11310         if (ll & 0xc) {
11311                 l += 2; ll >>= 2;
11312         }
11313         if (ll & 0x2) {
11314                 l += 1;
11315         }
11316         return (l);
11317 }

11319 static int
11320 nfs4_have_xattrs(vnode_t *vp, ulong_t *valp, cred_t *cr)
11321 {
11322         vnode_t *avp = NULL;
11323         int error;

11325         if ((error = nfs4lookup_xattr(vp, "", &avp,
11326             LOOKUP_XATTR, cr)) == 0)
11327                 error = do_xattr_exists_check(avp, valp, cr);
11328         if (avp)
11329                 VN_RELE(avp);

11331         return (error);
11332 }

11334 /* ARGSUSED */
11335 int
11336 nfs4_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
11337     caller_context_t *ct)
11338 {
11339         int error;
11340         hrtime_t t;
11341         rnode4_t *rp;
11342         nfs4_ga_res_t gar;
11343         nfs4_ga_ext_res_t ger;

11345         gar.n4g_ext_res = &ger;

11347         if (nfs_zone() != VTOMI4(vp)->mi_zone)
11348                 return (EIO);
11349         if (cmd == _PC_PATH_MAX || cmd == _PC_SYMLINK_MAX) {
11350                 *valp = MAXPATHLEN;
11351                 return (0);
11352         }
11353         if (cmd == _PC_ACL_ENABLED) {
11354                 *valp = _ACL_ACE_ENABLED;
11355                 return (0);
11356         }

11358         rp = VTOR4(vp);
11359         if (cmd == _PC_XATTR_EXISTS) {
11360                 /*
11361                  * The existence of the xattr directory is not sufficient
11362                  * for determining whether generic user attributes exists.
11363                  * The attribute directory could only be a transient directory
11364                  * used for Solaris sysattr support.  Do a small readdir
11365                  * to verify if the only entries are sysattrs or not.
11366                  *
11367                  * pc4_xattr_valid can be only be trusted when r_xattr_dir
11368                  * is NULL.  Once the xadir vp exists, we can create xattrs,
11369                  * and we don't have any way to update the "base" object's
```

```
11370                  * pc4_xattr_exists from the xattr or xadir.  Maybe FEM
11371                  * could help out.
11372                  */
11373                 if (ATTRCACHE4_VALID(vp) && rp->r_pathconf.pc4_xattr_valid &&
11374                     rp->r_xattr_dir == NULL) {
11375                         return (nfs4_have_xattrs(vp, valp, cr));
11376                 }
11377         } else {  /* OLD CODE */
11378                 if (ATTRCACHE4_VALID(vp)) {
11379                         mutex_enter(&rp->r_statelock);
11380                         if (rp->r_pathconf.pc4_cache_valid) {
11381                                 error = 0;
11382                                 switch (cmd) {
11383                                 case _PC_FILESIZEBITS:
11384                                         *valp =
11385                                             rp->r_pathconf.pc4_filesizebits;
11386                                         break;
11387                                 case _PC_LINK_MAX:
11388                                         *valp =
11389                                             rp->r_pathconf.pc4_link_max;
11390                                         break;
11391                                 case _PC_NAME_MAX:
11392                                         *valp =
11393                                             rp->r_pathconf.pc4_name_max;
11394                                         break;
11395                                 case _PC_CHOWN_RESTRICTED:
11396                                         *valp =
11397                                             rp->r_pathconf.pc4_chown_restricted;
11398                                         break;
11399                                 case _PC_NO_TRUNC:
11400                                         *valp =
11401                                             rp->r_pathconf.pc4_no_trunc;
11402                                         break;
11403                                 default:
11404                                         error = EINVAL;
11405                                         break;
11406                                 }
11407                                 mutex_exit(&rp->r_statelock);
11408 #ifdef DEBUG
11409                                 nfs4_pathconf_cache_hits++;
11410 #endif
11411                                 return (error);
11412                         }
11413                         mutex_exit(&rp->r_statelock);
11414                 }
11415         }
11416 #ifdef DEBUG
11417         nfs4_pathconf_cache_misses++;
11418 #endif

11420         t = gethrtime();

11422         error = nfs4_attr_otw(vp, TAG_PATHCONF, &gar, NFS4_PATHCONF_MASK, cr);

11424         if (error) {
11425                 mutex_enter(&rp->r_statelock);
11426                 rp->r_pathconf.pc4_cache_valid = FALSE;
11427                 rp->r_pathconf.pc4_xattr_valid = FALSE;
11428                 mutex_exit(&rp->r_statelock);
11429                 return (error);
11430         }

11432         /* interpret the max filesize */
11433         gar.n4g_ext_res->n4g_pc4.pc4_filesizebits =
11434             fattr4_maxfilesize_to_bits(gar.n4g_ext_res->n4g_maxfilesize);
```



```
11436         /* Store the attributes we just received */
11437         nfs4_attr_cache(vp, &gar, t, cr, TRUE, NULL);

11439         switch (cmd) {
11440         case _PC_FILESIZEBITS:
11441                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_filesizebits;
11442                 break;
11443         case _PC_LINK_MAX:
11444                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_link_max;
11445                 break;
11446         case _PC_NAME_MAX:
11447                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_name_max;
11448                 break;
11449         case _PC_CHOWN_RESTRICTED:
11450                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_chown_restricted;
11451                 break;
11452         case _PC_NO_TRUNC:
11453                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_no_trunc;
11454                 break;
11455         case _PC_XATTR_EXISTS:
11456                 if (gar.n4g_ext_res->n4g_pc4.pc4_xattr_exists) {
11457                         if (error = nfs4_have_xattrs(vp, valp, cr))
11458                                 return (error);
11459                 }
11460                 break;
11461         default:
11462                 return (EINVAL);
11463         }

11465         return (0);
11466 }

11468 /*
11469  * Called by async thread to do synchronous pageio. Do the i/o, wait
11470  * for it to complete, and cleanup the page list when done.
11471  */
11472 static int
11473 nfs4_sync_pageio(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
11474     int flags, cred_t *cr)
11475 {
11476         int error;

11478         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11480         error = nfs4_rdwrlbn(vp, pp, io_off, io_len, flags, cr);
11481         if (flags & B_READ)
11482                 pvn_read_done(pp, (error ? B_ERROR : 0) | flags);
11483         else
11484                 pvn_write_done(pp, (error ? B_ERROR : 0) | flags);
11485         return (error);
11486 }

11488 /* ARGSUSED */
11489 static int
11490 nfs4_pageio(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
11491     int flags, cred_t *cr, caller_context_t *ct)
11492 {
11493         int error;
11494         rnode4_t *rp;

11496         if (!(flags & B_ASYNC) && nfs_zone() != VTOMI4(vp)->mi_zone)
11497                 return (EIO);

11499         if (pp == NULL)
11500                 return (EINVAL);
```

```
11502         rp = VTOR4(vp);
11503         mutex_enter(&rp->r_statelock);
11504         rp->r_count++;
11505         mutex_exit(&rp->r_statelock);

11507         if (flags & B_ASYNC) {
11508                 error = nfs4_async_pageio(vp, pp, io_off, io_len, flags, cr,
11509                     nfs4_sync_pageio);
11510         } else
11511                 error = nfs4_rdwrlbn(vp, pp, io_off, io_len, flags, cr);
11512         mutex_enter(&rp->r_statelock);
11513         rp->r_count--;
11514         cv_broadcast(&rp->r_cv);
11515         mutex_exit(&rp->r_statelock);
11516         return (error);
11517 }

11519 /* ARGSUSED */
11520 static void
11521 nfs4_dispose(vnode_t *vp, page_t *pp, int fl, int dn, cred_t *cr,
11522     caller_context_t *ct)
11523 {
11524         int error;
11525         rnode4_t *rp;
11526         page_t *plist;
11527         page_t *pptr;
11528         offset3 offset;
11529         count3 len;
11530         k_sigset_t smask;

11532         /*
11533          * We should get called with fl equal to either B_FREE or
11534          * B_INVAL.  Any other value is illegal.
11535          *
11536          * The page that we are either supposed to free or destroy
11537          * should be exclusive locked and its io lock should not
11538          * be held.
11539          */
11540         ASSERT(fl == B_FREE || fl == B_INVAL);
11541         ASSERT((PAGE_EXCL(pp) && !page_iolock_assert(pp)) || panicstr);

11543         rp = VTOR4(vp);

11545         /*
11546          * If the page doesn't need to be committed or we shouldn't
11547          * even bother attempting to commit it, then just make sure
11548          * that the p_fsdata byte is clear and then either free or
11549          * destroy the page as appropriate.
11550          */
11551         if (pp->p_fsdata == C_NOCOMMIT || (rp->r_flags & R4STALE)) {
11552                 pp->p_fsdata = C_NOCOMMIT;
11553                 if (fl == B_FREE)
11554                         page_free(pp, dn);
11555                 else
11556                         page_destroy(pp, dn);
11557                 return;
11558         }

11560         /*
11561          * If there is a page invalidation operation going on, then
11562          * if this is one of the pages being destroyed, then just
11563          * clear the p_fsdata byte and then either free or destroy
11564          * the page as appropriate.
11565          */
11566         mutex_enter(&rp->r_statelock);
11567         if ((rp->r_flags & R4TRUNCATE) && pp->p_offset >= rp->r_truncaddr) {
```



```
11568                 mutex_exit(&rp->r_statelock);
11569                 pp->p_fsdata = C_NOCOMMIT;
11570                 if (fl == B_FREE)
11571                         page_free(pp, dn);
11572                 else
11573                         page_destroy(pp, dn);
11574                 return;
11575         }

11577         /*
11578          * If we are freeing this page and someone else is already
11579          * waiting to do a commit, then just unlock the page and
11580          * return.  That other thread will take care of commiting
11581          * this page.  The page can be freed sometime after the
11582          * commit has finished.  Otherwise, if the page is marked
11583          * as delay commit, then we may be getting called from
11584          * pvn_write_done, one page at a time.   This could result
11585          * in one commit per page, so we end up doing lots of small
11586          * commits instead of fewer larger commits.  This is bad,
11587          * we want do as few commits as possible.
11588          */
11589         if (fl == B_FREE) {
11590                 if (rp->r_flags & R4COMMITWAIT) {
11591                         page_unlock(pp);
11592                         mutex_exit(&rp->r_statelock);
11593                         return;
11594                 }
11595                 if (pp->p_fsdata == C_DELAYCOMMIT) {
11596                         pp->p_fsdata = C_COMMIT;
11597                         page_unlock(pp);
11598                         mutex_exit(&rp->r_statelock);
11599                         return;
11600                 }
11601         }

11603         /*
11604          * Check to see if there is a signal which would prevent an
11605          * attempt to commit the pages from being successful.  If so,
11606          * then don't bother with all of the work to gather pages and
11607          * generate the unsuccessful RPC.  Just return from here and
11608          * let the page be committed at some later time.
11609          */
11610         sigintr(&smask, VTOMI4(vp)->mi_flags & MI4_INT);
11611         if (ttolwp(curthread) != NULL && ISSIG(curthread, JUSTLOOKING)) {
11612                 sigunintr(&smask);
11613                 page_unlock(pp);
11614                 mutex_exit(&rp->r_statelock);
11615                 return;
11616         }
11617         sigunintr(&smask);

11619         /*
11620          * We are starting to need to commit pages, so let's try
11621          * to commit as many as possible at once to reduce the
11622          * overhead.
11623          *
11624          * Set the 'commit inprogress' state bit.  We must
11625          * first wait until any current one finishes.  Then
11626          * we initialize the c_pages list with this page.
11627          */
11628         while (rp->r_flags & R4COMMIT) {
11629                 rp->r_flags |= R4COMMITWAIT;
11630                 cv_wait(&rp->r_commit.c_cv, &rp->r_statelock);
11631                 rp->r_flags &= ~R4COMMITWAIT;
11632         }
11633         rp->r_flags |= R4COMMIT;
```

```
11634         mutex_exit(&rp->r_statelock);
11635         ASSERT(rp->r_commit.c_pages == NULL);
11636         rp->r_commit.c_pages = pp;
11637         rp->r_commit.c_commbase = (offset3)pp->p_offset;
11638         rp->r_commit.c_commlen = PAGESIZE;

11640         /*
11641          * Gather together all other pages which can be committed.
11642          * They will all be chained off r_commit.c_pages.
11643          */
11644         nfs4_get_commit(vp);

11646         /*
11647          * Clear the 'commit inprogress' status and disconnect
11648          * the list of pages to be committed from the rnode.
11649          * At this same time, we also save the starting offset
11650          * and length of data to be committed on the server.
11651          */
11652         plist = rp->r_commit.c_pages;
11653         rp->r_commit.c_pages = NULL;
11654         offset = rp->r_commit.c_commbase;
11655         len = rp->r_commit.c_commlen;
11656         mutex_enter(&rp->r_statelock);
11657         rp->r_flags &= ~R4COMMIT;
11658         cv_broadcast(&rp->r_commit.c_cv);
11659         mutex_exit(&rp->r_statelock);

11661         if (curproc == proc_pageout || curproc == proc_fsflush ||
11662             nfs_zone() != VTOMI4(vp)->mi_zone) {
11663                 nfs4_async_commit(vp, plist, offset, len,
11664                     cr, do_nfs4_async_commit);
11665                 return;
11666         }

11668         /*
11669          * Actually generate the COMMIT op over the wire operation.
11670          */
11671         error = nfs4_commit(vp, (offset4)offset, (count4)len, cr);

11673         /*
11674          * If we got an error during the commit, just unlock all
11675          * of the pages.  The pages will get retransmitted to the
11676          * server during a putpage operation.
11677          */
11678         if (error) {
11679                 while (plist != NULL) {
11680                         pptr = plist;
11681                         page_sub(&plist, pptr);
11682                         page_unlock(pptr);
11683                 }
11684                 return;
11685         }

11687         /*
11688          * We've tried as hard as we can to commit the data to stable
11689          * storage on the server.  We just unlock the rest of the pages
11690          * and clear the commit required state.  They will be put
11691          * onto the tail of the cachelist if they are nolonger
11692          * mapped.
11693          */
11694         while (plist != pp) {
11695                 pptr = plist;
11696                 page_sub(&plist, pptr);
11697                 pptr->p_fsdata = C_NOCOMMIT;
11698                 page_unlock(pptr);
11699         }
```



```
11701         /*
11702          * It is possible that nfs4_commit didn't return error but
11703          * some other thread has modified the page we are going
11704          * to free/destroy.
11705          *    In this case we need to rewrite the page. Do an explicit check
11706          * before attempting to free/destroy the page. If modified, needs to
11707          * be rewritten so unlock the page and return.
11708          */
11709         if (hat_ismod(pp)) {
11710                 pp->p_fsdata = C_NOCOMMIT;
11711                 page_unlock(pp);
11712                 return;
11713         }

11715         /*
11716          * Now, as appropriate, either free or destroy the page
11717          * that we were called with.
11718          */
11719         pp->p_fsdata = C_NOCOMMIT;
11720         if (fl == B_FREE)
11721                 page_free(pp, dn);
11722         else
11723                 page_destroy(pp, dn);
11724 }

11726 /*
11727  * Commit requires that the current fh be the file written to.
11728  * The compound op structure is:
11729  *      PUTFH(file), COMMIT
11730  */
11731 static int
11732 nfs4_commit(vnode_t *vp, offset4 offset, count4 count, cred_t *cr)
11733 {
11734         COMPOUND4args_clnt args;
11735         COMPOUND4res_clnt res;
11736         COMMIT4res *cm_res;
11737         nfs_argop4 argop[2];
11738         nfs_resop4 *resop;
11739         int doqueue;
11740         mntinfo4_t *mi;
11741         rnode4_t *rp;
11742         cred_t *cred_otw = NULL;
11743         bool_t needrecov = FALSE;
11744         nfs4_recov_state_t recov_state;
11745         nfs4_open_stream_t *osp = NULL;
11746         bool_t first_time = TRUE;       /* first time getting OTW cred */
11747         bool_t last_time = FALSE;       /* last time getting OTW cred */
11748         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

11750         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11752         rp = VTOR4(vp);

11754         mi = VTOMI4(vp);
11755         recov_state.rs_flags = 0;
11756         recov_state.rs_num_retry_despite_err = 0;
11757 get_commit_cred:
11758         /*
11759          * Releases the osp, if a valid open stream is provided.
11760          * Puts a hold on the cred_otw and the new osp (if found).
11761          */
11762         cred_otw = nfs4_get_otw_cred_by_osp(rp, cr, &osp,
11763             &first_time, &last_time);
11764         args.ctag = TAG_COMMIT;
11765 recov_retry:
```

```
11766         /*
11767          * Commit ops: putfh file; commit
11768          */
11769         args.array_len = 2;
11770         args.array = argop;

11772         e.error = nfs4_start_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11773             &recov_state, NULL);
11774         if (e.error) {
11775                 crfree(cred_otw);
11776                 if (osp != NULL)
11777                         open_stream_rele(osp, rp);
11778                 return (e.error);
11779         }

11781         /* putfh directory */
11782         argop[0].argop = OP_CPUTFH;
11783         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

11785         /* commit */
11786         argop[1].argop = OP_COMMIT;
11787         argop[1].nfs_argop4_u.opcommit.offset = offset;
11788         argop[1].nfs_argop4_u.opcommit.count = count;

11790         doqueue = 1;
11791         rfs4call(mi, &args, &res, cred_otw, &doqueue, 0, &e);

11793         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
11794         if (!needrecov && e.error) {
11795                 nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT, &recov_state,
11796                     needrecov);
11797                 crfree(cred_otw);
11798                 if (e.error == EACCES && last_time == FALSE)
11799                         goto get_commit_cred;
11800                 if (osp != NULL)
11801                         open_stream_rele(osp, rp);
11802                 return (e.error);
11803         }

11805         if (needrecov) {
11806                 if (nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
11807                     NULL, OP_COMMIT, NULL, NULL, NULL) == FALSE) {
11808                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11809                             &recov_state, needrecov);
11810                         if (!e.error)
11811                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
11812                                     (caddr_t)&res);
11813                         goto recov_retry;
11814                 }
11815                 if (e.error) {
11816                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11817                             &recov_state, needrecov);
11818                         crfree(cred_otw);
11819                         if (osp != NULL)
11820                                 open_stream_rele(osp, rp);
11821                         return (e.error);
11822                 }
11823                 /* fall through for res.status case */
11824         }

11826         if (res.status) {
11827                 e.error = geterrno4(res.status);
11828                 if (e.error == EACCES && last_time == FALSE) {
11829                         crfree(cred_otw);
11830                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11831                             &recov_state, needrecov);
```



```
11832                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
11833                         goto get_commit_cred;
11834                 }
11835                 /*
11836                  * Can't do a nfs4_purge_stale_fh here because this
11837                  * can cause a deadlock.  nfs4_commit can
11838                  * be called from nfs4_dispose which can be called
11839                  * indirectly via pvn_vplist_dirty.  nfs4_purge_stale_fh
11840                  * can call back to pvn_vplist_dirty.
11841                  */
11842                 if (e.error == ESTALE) {
11843                         mutex_enter(&rp->r_statelock);
11844                         rp->r_flags |= R4STALE;
11845                         if (!rp->r_error)
11846                                 rp->r_error = e.error;
11847                         mutex_exit(&rp->r_statelock);
11848                         PURGE_ATTRCACHE4(vp);
11849                 } else {
11850                         mutex_enter(&rp->r_statelock);
11851                         if (!rp->r_error)
11852                                 rp->r_error = e.error;
11853                         mutex_exit(&rp->r_statelock);
11854                 }
11855         } else {
11856                 ASSERT(rp->r_flags & R4HAVEVERF);
11857                 resop = &res.array[1];  /* commit res */
11858                 cm_res = &resop->nfs_resop4_u.opcommit;
11859                 mutex_enter(&rp->r_statelock);
11860                 if (cm_res->writeverf == rp->r_writeverf) {
11861                         mutex_exit(&rp->r_statelock);
11862                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
11863                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11864                             &recov_state, needrecov);
11865                         crfree(cred_otw);
11866                         if (osp != NULL)
11867                                 open_stream_rele(osp, rp);
11868                         return (0);
11869                 }
11870                 nfs4_set_mod(vp);
11871                 rp->r_writeverf = cm_res->writeverf;
11872                 mutex_exit(&rp->r_statelock);
11873                 e.error = NFS_VERF_MISMATCH;
11874         }

11876         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
11877         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT, &recov_state, needrecov);
11878         crfree(cred_otw);
11879         if (osp != NULL)
11880                 open_stream_rele(osp, rp);

11882         return (e.error);
11883 }

11885 static void
11886 nfs4_set_mod(vnode_t *vp)
11887 {
11888         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11890         /* make sure we're looking at the master vnode, not a shadow */
11891         pvn_vplist_setdirty(RTOV4(VTOR4(vp)), nfs_setmod_check);
11892 }

11894 /*
11895  * This function is used to gather a page list of the pages which
11896  * can be committed on the server.
11897  *
```

```
11898  * The calling thread must have set R4COMMIT.  This bit is used to
11899  * serialize access to the commit structure in the rnode.  As long
11900  * as the thread has set R4COMMIT, then it can manipulate the commit
11901  * structure without requiring any other locks.
11902  *
11903  * When this function is called from nfs4_dispose() the page passed
11904  * into nfs4_dispose() will be SE_EXCL locked, and so this function
11905  * will skip it. This is not a problem since we initially add the
11906  * page to the r_commit page list.
11907  *
11908  */
11909 static void
11910 nfs4_get_commit(vnode_t *vp)
11911 {
11912         rnode4_t *rp;
11913         page_t *pp;
11914         kmutex_t *vphm;

11916         rp = VTOR4(vp);

11918         ASSERT(rp->r_flags & R4COMMIT);

11920         /* make sure we're looking at the master vnode, not a shadow */

11922         if (IS_SHADOW(vp, rp))
11923                 vp = RTOV4(rp);

11925         vphm = page_vnode_mutex(vp);
11926         mutex_enter(vphm);

11928         /*
11929          * If there are no pages associated with this vnode, then
11930          * just return.
11931          */
11932         if ((pp = vp->v_pages) == NULL) {
11933                 mutex_exit(vphm);
11934                 return;
11935         }

11937         /*
11938          * Step through all of the pages associated with this vnode
11939          * looking for pages which need to be committed.
11940          */
11941         do {
11942                 /* Skip marker pages. */
11943                 if (pp->p_hash == PVN_VPLIST_HASH_TAG)
11944                         continue;

11946                 /*
11947                  * First short-cut everything (without the page_lock)
11948                  * and see if this page does not need to be committed
11949                  * or is modified if so then we'll just skip it.
11950                  */
11951                 if (pp->p_fsdata == C_NOCOMMIT || hat_ismod(pp))
11952                         continue;

11954                 /*
11955                  * Attempt to lock the page.  If we can't, then
11956                  * someone else is messing with it or we have been
11957                  * called from nfs4_dispose and this is the page that
11958                  * nfs4_dispose was called with.. anyway just skip it.
11959                  */
11960                 if (!page_trylock(pp, SE_EXCL))
11961                         continue;

11963                 /*
```



```
11964                  * Lets check again now that we have the page lock.
11965                  */
11966                 if (pp->p_fsdata == C_NOCOMMIT || hat_ismod(pp)) {
11967                         page_unlock(pp);
11968                         continue;
11969                 }

11971                 /* this had better not be a free page */
11972                 ASSERT(PP_ISFREE(pp) == 0);

11974                 /*
11975                  * The page needs to be committed and we locked it.
11976                  * Update the base and length parameters and add it
11977                  * to r_pages.
11978                  */
11979                 if (rp->r_commit.c_pages == NULL) {
11980                         rp->r_commit.c_commbase = (offset3)pp->p_offset;
11981                         rp->r_commit.c_commlen = PAGESIZE;
11982                 } else if (pp->p_offset < rp->r_commit.c_commbase) {
11983                         rp->r_commit.c_commlen = rp->r_commit.c_commbase -
11984                             (offset3)pp->p_offset + rp->r_commit.c_commlen;
11985                         rp->r_commit.c_commbase = (offset3)pp->p_offset;
11986                 } else if ((rp->r_commit.c_commbase + rp->r_commit.c_commlen)
11987                     <= pp->p_offset) {
11988                         rp->r_commit.c_commlen = (offset3)pp->p_offset -
11989                             rp->r_commit.c_commbase + PAGESIZE;
11990                 }
11991                 page_add(&rp->r_commit.c_pages, pp);
11992         } while ((pp = pp->p_vpnext) != vp->v_pages);

11994         mutex_exit(vphm);
11995 }

11997 /*
11998  * This routine is used to gather together a page list of the pages
11999  * which are to be committed on the server.  This routine must not
12000  * be called if the calling thread holds any locked pages.
12001  *
12002  * The calling thread must have set R4COMMIT.  This bit is used to
12003  * serialize access to the commit structure in the rnode.  As long
12004  * as the thread has set R4COMMIT, then it can manipulate the commit
12005  * structure without requiring any other locks.
12006  */
12007 static void
12008 nfs4_get_commit_range(vnode_t *vp, u_offset_t soff, size_t len)
12009 {

12011         rnode4_t *rp;
12012         page_t *pp;
12013         u_offset_t end;
12014         u_offset_t off;
12015         ASSERT(len != 0);
12016         rp = VTOR4(vp);
12017         ASSERT(rp->r_flags & R4COMMIT);

12019         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12021         /* make sure we're looking at the master vnode, not a shadow */

12023         if (IS_SHADOW(vp, rp))
12024                 vp = RTOV4(rp);

12026         /*
12027          * If there are no pages associated with this vnode, then
12028          * just return.
12029          */
```

```
12030         if ((pp = vp->v_pages) == NULL)
12031                 return;
12032         /*
12033          * Calculate the ending offset.
12034          */
12035         end = soff + len;
12036         for (off = soff; off < end; off += PAGESIZE) {
12037                 /*
12038                  * Lookup each page by vp, offset.
12039                  */
12040                 if ((pp = page_lookup_nowait(vp, off, SE_EXCL)) == NULL)
12041                         continue;
12042                 /*
12043                  * If this page does not need to be committed or is
12044                  * modified, then just skip it.
12045                  */
12046                 if (pp->p_fsdata == C_NOCOMMIT || hat_ismod(pp)) {
12047                         page_unlock(pp);
12048                         continue;
12049                 }

12051                 ASSERT(PP_ISFREE(pp) == 0);
12052                 /*
12053                  * The page needs to be committed and we locked it.
12054                  * Update the base and length parameters and add it
12055                  * to r_pages.
12056                  */
12057                 if (rp->r_commit.c_pages == NULL) {
12058                         rp->r_commit.c_commbase = (offset3)pp->p_offset;
12059                         rp->r_commit.c_commlen = PAGESIZE;
12060                 } else {
12061                         rp->r_commit.c_commlen = (offset3)pp->p_offset -
12062                             rp->r_commit.c_commbase + PAGESIZE;
12063                 }
12064                 page_add(&rp->r_commit.c_pages, pp);
12065         }
12066 }

12068 /*
12069  * Called from nfs4_close(), nfs4_fsync() and nfs4_delmap().
12070  * Flushes and commits data to the server.
12071  */
12072 static int
12073 nfs4_putpage_commit(vnode_t *vp, offset_t poff, size_t plen, cred_t *cr)
12074 {
12075         int error;
12076         verifier4 write_verf;
12077         rnode4_t *rp = VTOR4(vp);

12079         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12081         /*
12082          * Flush the data portion of the file and then commit any
12083          * portions which need to be committed.  This may need to
12084          * be done twice if the server has changed state since
12085          * data was last written.  The data will need to be
12086          * rewritten to the server and then a new commit done.
12087          *
12088          * In fact, this may need to be done several times if the
12089          * server is having problems and crashing while we are
12090          * attempting to do this.
12091          */

12093 top:
12094         /*
12095          * Do a flush based on the poff and plen arguments.  This
```



```
12096          * will synchronously write out any modified pages in the
12097          * range specified by (poff, plen). This starts all of the
12098          * i/o operations which will be waited for in the next
12099          * call to nfs4_putpage
12100          */

12102         mutex_enter(&rp->r_statelock);
12103         write_verf = rp->r_writeverf;
12104         mutex_exit(&rp->r_statelock);

12106         error = nfs4_putpage(vp, poff, plen, B_ASYNC, cr, NULL);
12107         if (error == EAGAIN)
12108                 error = 0;

12110         /*
12111          * Do a flush based on the poff and plen arguments.  This
12112          * will synchronously write out any modified pages in the
12113          * range specified by (poff, plen) and wait until all of
12114          * the asynchronous i/o's in that range are done as well.
12115          */
12116         if (!error)
12117                 error = nfs4_putpage(vp, poff, plen, 0, cr, NULL);

12119         if (error)
12120                 return (error);

12122         mutex_enter(&rp->r_statelock);
12123         if (rp->r_writeverf != write_verf) {
12124                 mutex_exit(&rp->r_statelock);
12125                 goto top;
12126         }
12127         mutex_exit(&rp->r_statelock);

12129         /*
12130          * Now commit any pages which might need to be committed.
12131          * If the error, NFS_VERF_MISMATCH, is returned, then
12132          * start over with the flush operation.
12133          */
12134         error = nfs4_commit_vp(vp, poff, plen, cr, NFS4_WRITE_WAIT);

12136         if (error == NFS_VERF_MISMATCH)
12137                 goto top;

12139         return (error);
12140 }

12142 /*
12143  * nfs4_commit_vp()  will wait for other pending commits and
12144  * will either commit the whole file or a range, plen dictates
12145  * if we commit whole file. a value of zero indicates the whole
12146  * file. Called from nfs4_putpage_commit() or nfs4_sync_putapage()
12147  */
12148 static int
12149 nfs4_commit_vp(vnode_t *vp, u_offset_t poff, size_t plen,
12150     cred_t *cr, int wait_on_writes)
12151 {
12152         rnode4_t *rp;
12153         page_t *plist;
12154         offset3 offset;
12155         count3 len;

12157         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12159         rp = VTOR4(vp);

12161         /*
```

```
12162          *  before we gather commitable pages make
12163          *  sure there are no outstanding async writes
12164          */
12165         if (rp->r_count && wait_on_writes == NFS4_WRITE_WAIT) {
12166                 mutex_enter(&rp->r_statelock);
12167                 while (rp->r_count > 0) {
12168                         cv_wait(&rp->r_cv, &rp->r_statelock);
12169                 }
12170                 mutex_exit(&rp->r_statelock);
12171         }

12173         /*
12174          * Set the 'commit inprogress' state bit.  We must
12175          * first wait until any current one finishes.
12176          */
12177         mutex_enter(&rp->r_statelock);
12178         while (rp->r_flags & R4COMMIT) {
12179                 rp->r_flags |= R4COMMITWAIT;
12180                 cv_wait(&rp->r_commit.c_cv, &rp->r_statelock);
12181                 rp->r_flags &= ~R4COMMITWAIT;
12182         }
12183         rp->r_flags |= R4COMMIT;
12184         mutex_exit(&rp->r_statelock);

12186         /*
12187          * Gather all of the pages which need to be
12188          * committed.
12189          */
12190         if (plen == 0)
12191                 nfs4_get_commit(vp);
12192         else
12193                 nfs4_get_commit_range(vp, poff, plen);

12195         /*
12196          * Clear the 'commit inprogress' bit and disconnect the
12197          * page list which was gathered by nfs4_get_commit.
12198          */
12199         plist = rp->r_commit.c_pages;
12200         rp->r_commit.c_pages = NULL;
12201         offset = rp->r_commit.c_commbase;
12202         len = rp->r_commit.c_commlen;
12203         mutex_enter(&rp->r_statelock);
12204         rp->r_flags &= ~R4COMMIT;
12205         cv_broadcast(&rp->r_commit.c_cv);
12206         mutex_exit(&rp->r_statelock);

12208         /*
12209          * If any pages need to be committed, commit them and
12210          * then unlock them so that they can be freed some
12211          * time later.
12212          */
12213         if (plist == NULL)
12214                 return (0);

12216         /*
12217          * No error occurred during the flush portion
12218          * of this operation, so now attempt to commit
12219          * the data to stable storage on the server.
12220          *
12221          * This will unlock all of the pages on the list.
12222          */
12223         return (nfs4_sync_commit(vp, plist, offset, len, cr));
12224 }

12226 static int
12227 nfs4_sync_commit(vnode_t *vp, page_t *plist, offset3 offset, count3 count,
```



```
12228     cred_t *cr)
12229 {
12230         int error;
12231         page_t *pp;

12233         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12235         error = nfs4_commit(vp, (offset4)offset, (count3)count, cr);

12237         /*
12238          * If we got an error, then just unlock all of the pages
12239          * on the list.
12240          */
12241         if (error) {
12242                 while (plist != NULL) {
12243                         pp = plist;
12244                         page_sub(&plist, pp);
12245                         page_unlock(pp);
12246                 }
12247                 return (error);
12248         }
12249         /*
12250          * We've tried as hard as we can to commit the data to stable
12251          * storage on the server.  We just unlock the pages and clear
12252          * the commit required state.  They will get freed later.
12253          */
12254         while (plist != NULL) {
12255                 pp = plist;
12256                 page_sub(&plist, pp);
12257                 pp->p_fsdata = C_NOCOMMIT;
12258                 page_unlock(pp);
12259         }

12261         return (error);
12262 }

12264 static void
12265 do_nfs4_async_commit(vnode_t *vp, page_t *plist, offset3 offset, count3 count,
12266     cred_t *cr)
12267 {

12269         (void) nfs4_sync_commit(vp, plist, offset, count, cr);
12270 }

12272 /*ARGSUSED*/
12273 static int
12274 nfs4_setsecattr(vnode_t *vp, vsecattr_t *vsecattr, int flag, cred_t *cr,
12275     caller_context_t *ct)
12276 {
12277         int             error = 0;
12278         mntinfo4_t      *mi;
12279         vattr_t         va;
12280         vsecattr_t      nfsace4_vsap;

12282         mi = VTOMI4(vp);
12283         if (nfs_zone() != mi->mi_zone)
12284                 return (EIO);
12285         if (mi->mi_flags & MI4_ACL) {
12286                 /* if we have a delegation, return it */
12287                 if (VTOR4(vp)->r_deleg_type != OPEN_DELEGATE_NONE)
12288                         (void) nfs4delegreturn(VTOR4(vp),
12289                             NFS4_DR_REOPEN|NFS4_DR_PUSH);

12291                 error = nfs4_is_acl_mask_valid(vsecattr->vsa_mask,
12292                     NFS4_ACL_SET);
12293                 if (error) /* EINVAL */
```

```
12294                           return (error);
12295
12296                   if (vsecattr->vsa_mask & (VSA_ACL | VSA_DFACL)) {
12297                           /*
12298                            * These are aclent_t type entries.
12299                            */
12300                           error = vs_aent_to_ace4(vsecattr, &nfsace4_vsap,
12301                               vp->v_type == VDIR, FALSE);
12302                           if (error)
12303                                   return (error);
12304                   } else {
12305                           /*
12306                            * These are ace_t type entries.
12307                            */
12308                           error = vs_acet_to_ace4(vsecattr, &nfsace4_vsap,
12309                               FALSE);
12310                           if (error)
12311                                   return (error);
12312                   }
12313                   bzero(&va, sizeof (va));
12314                   error = nfs4setattr(vp, &va, flag, cr, &nfsace4_vsap);
12315                   vs_ace4_destroy(&nfsace4_vsap);
12316                   return (error);
12317           }
12318           return (ENOSYS);
12319   }

12321   /* ARGSUSED */
12322   int
12323   nfs4_getsecattr(vnode_t *vp, vsecattr_t *vsecattr, int flag, cred_t *cr,
12324       caller_context_t *ct)
12325   {
12326           int             error;
12327           mntinfo4_t      *mi;
12328           nfs4_ga_res_t   gar;
12329           rnode4_t        *rp = VTOR4(vp);

12331           mi = VTOMI4(vp);
12332           if (nfs_zone() != mi->mi_zone)
12333                   return (EIO);

12335           bzero(&gar, sizeof (gar));
12336           gar.n4g_vsa.vsa_mask = vsecattr->vsa_mask;

12338           /*
12339            * vsecattr->vsa_mask holds the original acl request mask.
12340            * This is needed when determining what to return.
12341            * (See: nfs4_create_getsecattr_return())
12342            */
12343           error = nfs4_is_acl_mask_valid(vsecattr->vsa_mask, NFS4_ACL_GET);
12344           if (error) /* EINVAL */
12345                   return (error);

12347           /*
12348            * If this is a referral stub, don't try to go OTW for an ACL
12349            */
12350           if (RP_ISSTUB_REFERRAL(VTOR4(vp)))
12351                   return (fs_fab_acl(vp, vsecattr, flag, cr, ct));

12353           if (mi->mi_flags & MI4_ACL) {
12354                   /*
12355                    * Check if the data is cached and the cache is valid.  If it
12356                    * is we don't go over the wire.
12357                    */
12358                   if (rp->r_secattr != NULL && ATTRCACHE4_VALID(vp)) {
12359                           mutex_enter(&rp->r_statelock);
```



```
12360                           if (rp->r_secattr != NULL) {
12361                                   error = nfs4_create_getsecattr_return(
12362                                       rp->r_secattr, vsecattr, rp->r_attr.va_uid,
12363                                       rp->r_attr.va_gid,
12364                                       vp->v_type == VDIR);
12365                                   if (!error) { /* error == 0 - Success! */
12366                                           mutex_exit(&rp->r_statelock);
12367                                           return (error);
12368                                   }
12369                           }
12370                           mutex_exit(&rp->r_statelock);
12371                   }

12373                   /*
12374                    * The getattr otw call will always get both the acl, in
12375                    * the form of a list of nfsace4's, and the number of acl
12376                    * entries; independent of the value of gar.n4g_vsa.vsa_mask.
12377                    */
12378                   gar.n4g_va.va_mask = AT_ALL;
12379                   error =  nfs4_getattr_otw(vp, &gar, cr, 1);
12380                   if (error) {
12381                           vs_ace4_destroy(&gar.n4g_vsa);
12382                           if (error == ENOTSUP || error == EOPNOTSUPP)
12383                                   error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12384                           return (error);
12385                   }

12387                   if (!(gar.n4g_resbmap & FATTR4_ACL_MASK)) {
12388                           /*
12389                            * No error was returned, but according to the response
12390                            * bitmap, neither was an acl.
12391                            */
12392                           vs_ace4_destroy(&gar.n4g_vsa);
12393                           error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12394                           return (error);
12395                   }

12397                   /*
12398                    * Update the cache with the ACL.
12399                    */
12400                   nfs4_acl_fill_cache(rp, &gar.n4g_vsa);

12402                   error = nfs4_create_getsecattr_return(&gar.n4g_vsa,
12403                       vsecattr, gar.n4g_va.va_uid, gar.n4g_va.va_gid,
12404                       vp->v_type == VDIR);
12405                   vs_ace4_destroy(&gar.n4g_vsa);
12406                   if ((error) && (vsecattr->vsa_mask &
12407                       (VSA_ACL | VSA_ACLCNT | VSA_DFACL | VSA_DFACLCNT)) &&
12408                       (error != EACCES)) {
12409                           error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12410                   }
12411                   return (error);
12412           }
12413           error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12414           return (error);
12415   }

12417   /*
12418    * The function returns:
12419    *      - 0 (zero) if the passed in "acl_mask" is a valid request.
12420    *      - EINVAL if the passed in "acl_mask" is an invalid request.
12421    *
12422    * In the case of getting an acl (op == NFS4_ACL_GET) the mask is invalid if:
12423    * - We have a mixture of ACE and ACL requests (e.g. VSA_ACL | VSA_ACE)
12424    *
12425    * In the case of setting an acl (op == NFS4_ACL_SET) the mask is invalid if:
```

```
12426  * - We have a mixture of ACE and ACL requests (e.g. VSA_ACL | VSA_ACE)
12427  * - We have a count field set without the corresponding acl field set. (e.g. -
12428  * VSA_ACECNT is set, but VSA_ACE is not)
12429  */
12430 static int
12431 nfs4_is_acl_mask_valid(uint_t acl_mask, nfs4_acl_op_t op)
12432 {
12433         /* Shortcut the masks that are always valid. */
12434         if (acl_mask == (VSA_ACE | VSA_ACECNT))
12435                 return (0);
12436         if (acl_mask == (VSA_ACL | VSA_ACLCNT | VSA_DFACL | VSA_DFACLCNT))
12437                 return (0);

12439         if (acl_mask & (VSA_ACE | VSA_ACECNT)) {
12440                 /*
12441                  * We can't have any VSA_ACL type stuff in the mask now.
12442                  */
12443                 if (acl_mask & (VSA_ACL | VSA_ACLCNT | VSA_DFACL |
12444                     VSA_DFACLCNT))
12445                         return (EINVAL);

12447                 if (op == NFS4_ACL_SET) {
12448                         if ((acl_mask & VSA_ACECNT) && !(acl_mask & VSA_ACE))
12449                                 return (EINVAL);
12450                 }
12451         }

12453         if (acl_mask & (VSA_ACL | VSA_ACLCNT | VSA_DFACL | VSA_DFACLCNT)) {
12454                 /*
12455                  * We can't have any VSA_ACE type stuff in the mask now.
12456                  */
12457                 if (acl_mask & (VSA_ACE | VSA_ACECNT))
12458                         return (EINVAL);

12460                 if (op == NFS4_ACL_SET) {
12461                         if ((acl_mask & VSA_ACLCNT) && !(acl_mask & VSA_ACL))
12462                                 return (EINVAL);

12464                         if ((acl_mask & VSA_DFACLCNT) &&
12465                             !(acl_mask & VSA_DFACL))
12466                                 return (EINVAL);
12467                 }
12468         }
12469         return (0);
12470 }

12472 /*
12473  * The theory behind creating the correct getsecattr return is simply this:
12474  * "Don't return anything that the caller is not expecting to have to free."
12475  */
12476 static int
12477 nfs4_create_getsecattr_return(vsecattr_t *filled_vsap, vsecattr_t *vsap,
12478     uid_t uid, gid_t gid, int isdir)
12479 {
12480         int error = 0;
12481         /* Save the mask since the translators modify it. */
12482         uint_t  orig_mask = vsap->vsa_mask;

12484         if (orig_mask & (VSA_ACE | VSA_ACECNT)) {
12485                 error = vs_ace4_to_acet(filled_vsap, vsap, uid, gid, FALSE);

12487                 if (error)
12488                         return (error);

12490                 /*
12491                  * If the caller only asked for the ace count (VSA_ACECNT)
```



```
12492                  * don't give them the full acl (VSA_ACE), free it.
12493                  */
12494                 if (!orig_mask & VSA_ACE) {
12495                         if (vsap->vsa_aclentp != NULL) {
12496                                 kmem_free(vsap->vsa_aclentp,
12497                                     vsap->vsa_aclcnt * sizeof (ace_t));
12498                                 vsap->vsa_aclentp = NULL;
12499                         }
12500                 }
12501                 vsap->vsa_mask = orig_mask;

12503         } else if (orig_mask & (VSA_ACL | VSA_ACLCNT | VSA_DFACL |
12504             VSA_DFACLCNT)) {
12505                 error = vs_ace4_to_aent(filled_vsap, vsap, uid, gid,
12506                     isdir, FALSE);

12508                 if (error)
12509                         return (error);

12511                 /*
12512                  * If the caller only asked for the acl count (VSA_ACLCNT)
12513                  * and/or the default acl count (VSA_DFACLCNT) don't give them
12514                  * the acl (VSA_ACL) or default acl (VSA_DFACL), free it.
12515                  */
12516                 if (!orig_mask & VSA_ACL) {
12517                         if (vsap->vsa_aclentp != NULL) {
12518                                 kmem_free(vsap->vsa_aclentp,
12519                                     vsap->vsa_aclcnt * sizeof (aclent_t));
12520                                 vsap->vsa_aclentp = NULL;
12521                         }
12522                 }

12524                 if (!orig_mask & VSA_DFACL) {
12525                         if (vsap->vsa_dfaclentp != NULL) {
12526                                 kmem_free(vsap->vsa_dfaclentp,
12527                                     vsap->vsa_dfaclcnt * sizeof (aclent_t));
12528                                 vsap->vsa_dfaclentp = NULL;
12529                         }
12530                 }
12531                 vsap->vsa_mask = orig_mask;
12532         }
12533         return (0);
12534 }

12536 /* ARGSUSED */
12537 int
12538 nfs4_shrlock(vnode_t *vp, int cmd, struct shrlock *shr, int flag, cred_t *cr,
12539     caller_context_t *ct)
12540 {
12541         int error;

12543         if (nfs_zone() != VTOMI4(vp)->mi_zone)
12544                 return (EIO);
12545         /*
12546          * check for valid cmd parameter
12547          */
12548         if (cmd != F_SHARE && cmd != F_UNSHARE && cmd != F_HASREMOTELOCKS)
12549                 return (EINVAL);

12551         /*
12552          * Check access permissions
12553          */
12554         if ((cmd & F_SHARE) &&
12555             (((shr->s_access & F_RDACC) && (flag & FREAD) == 0) ||
12556             (shr->s_access == F_WRACC && (flag & FWRITE) == 0)))
12557                 return (EBADF);
```

```
12559         /*
12560          * If the filesystem is mounted using local locking, pass the
12561          * request off to the local share code.
12562          */
12563         if (VTOMI4(vp)->mi_flags & MI4_LLOCK)
12564                 return (fs_shrlock(vp, cmd, shr, flag, cr, ct));

12566         switch (cmd) {
12567         case F_SHARE:
12568         case F_UNSHARE:
12569                 /*
12570                  * This will be properly implemented later,
12571                  * see RFE: 4823948 .
12572                  */
12573                 error = EAGAIN;
12574                 break;

12576         case F_HASREMOTELOCKS:
12577                 /*
12578                  * NFS client can't store remote locks itself
12579                  */
12580                 shr->s_access = 0;
12581                 error = 0;
12582                 break;

12584         default:
12585                 error = EINVAL;
12586                 break;
12587         }

12589         return (error);
12590 }

12592 /*
12593  * Common code called by directory ops to update the attrcache
12594  */
12595 static int
12596 nfs4_update_attrcache(nfsstat4 status, nfs4_ga_res_t *garp,
12597     hrtime_t t, vnode_t *vp, cred_t *cr)
12598 {
12599         int error = 0;

12601         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12603         if (status != NFS4_OK) {
12604                 /* getattr not done or failed */
12605                 PURGE_ATTRCACHE4(vp);
12606                 return (error);
12607         }

12609         if (garp) {
12610                 nfs4_attr_cache(vp, garp, t, cr, FALSE, NULL);
12611         } else {
12612                 PURGE_ATTRCACHE4(vp);
12613         }
12614         return (error);
12615 }

12617 /*
12618  * Update directory caches for directory modification ops (link, rename, etc.)
12619  * When dinfo is NULL, manage dircaches in the old way.
12620  */
12621 static void
12622 nfs4_update_dircaches(change_info4 *cinfo, vnode_t *dvp, vnode_t *vp, char *nm,
12623     dirattr_info_t *dinfo)
```



```
12624 {
12625         rnode4_t        *drp = VTOR4(dvp);

12627         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

12629         /* Purge rddir cache for dir since it changed */
12630         if (drp->r_dir != NULL)
12631                 nfs4_purge_rddir_cache(dvp);

12633         /*
12634          * If caller provided dinfo, then use it to manage dir caches.
12635          */
12636         if (dinfo != NULL) {
12637                 if (vp != NULL) {
12638                         mutex_enter(&VTOR4(vp)->r_statev4_lock);
12639                         if (!VTOR4(vp)->created_v4) {
12640                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12641                                 dnlc_update(dvp, nm, vp);
12642                         } else {
12643                                 /*
12644                                  * XXX don't update if the created_v4 flag is
12645                                  * set
12646                                  */
12647                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12648                                 NFS4_DEBUG(nfs4_client_state_debug,
12649                                     (CE_NOTE, "nfs4_update_dircaches: "
12650                                     "don't update dnlc: created_v4 flag"));
12651                         }
12652                 }

12654                 nfs4_attr_cache(dvp, dinfo->di_garp, dinfo->di_time_call,
12655                     dinfo->di_cred, FALSE, cinfo);

12657                 return;
12658         }

12660         /*
12661          * Caller didn't provide dinfo, then check change_info4 to update DNLC.
12662          * Since caller modified dir but didn't receive post-dirmod-op dir
12663          * attrs, the dir's attrs must be purged.
12664          *
12665          * XXX this check and dnlc update/purge should really be atomic,
12666          * XXX but can't use rnode statelock because it'll deadlock in
12667          * XXX dnlc_purge_vp, however, the risk is minimal even if a race
12668          * XXX does occur.
12669          *
12670          * XXX We also may want to check that atomic is true in the
12671          * XXX change_info struct. If it is not, the change_info may
12672          * XXX reflect changes by more than one clients which means that
12673          * XXX our cache may not be valid.
12674          */
12675         PURGE_ATTRCACHE4(dvp);
12676         if (drp->r_change == cinfo->before) {
12677                 /* no changes took place in the directory prior to our link */
12678                 if (vp != NULL) {
12679                         mutex_enter(&VTOR4(vp)->r_statev4_lock);
12680                         if (!VTOR4(vp)->created_v4) {
12681                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12682                                 dnlc_update(dvp, nm, vp);
12683                         } else {
12684                                 /*
12685                                  * XXX dont' update if the created_v4 flag
12686                                  * is set
12687                                  */
12688                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12689                                 NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE,
```

```
12690                                   "nfs4_update_dircaches: don't"
12691                                   " update dnlc: created_v4 flag"));
12692                           }
12693                   }
12694           } else {
12695                   /* Another client modified directory - purge its dnlc cache */
12696                   dnlc_purge_vp(dvp);
12697           }
12698 }

12700 /*
12701  * The OPEN_CONFIRM operation confirms the sequence number used in OPENing a
12702  * file.
12703  *
12704  * The 'reopening_file' boolean should be set to TRUE if we are reopening this
12705  * file (ie: client recovery) and otherwise set to FALSE.
12706  *
12707  * 'nfs4_start/end_op' should have been called by the proper (ie: not recovery
12708  * initiated) calling functions.
12709  *
12710  * 'resend' is set to TRUE if this is a OPEN_CONFIRM issued as a result
12711  * of resending a 'lost' open request.
12712  *
12713  * 'num_bseqid_retryp' makes sure we don't loop forever on a broken
12714  * server that hands out BAD_SEQID on open confirm.
12715  *
12716  * Errors are returned via the nfs4_error_t parameter.
12717  */
12718 void
12719 nfs4open_confirm(vnode_t *vp, seqid4 *seqid, stateid4 *stateid, cred_t *cr,
12720     bool_t reopening_file, bool_t *retry_open, nfs4_open_owner_t *oop,
12721     bool_t resend, nfs4_error_t *ep, int *num_bseqid_retryp)
12722 {
12723         COMPOUND4args_clnt args;
12724         COMPOUND4res_clnt res;
12725         nfs_argop4 argop[2];
12726         nfs_resop4 *resop;
12727         int doqueue = 1;
12728         mntinfo4_t *mi;
12729         OPEN_CONFIRM4args *open_confirm_args;
12730         int needrecov;

12732         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
12733 #if DEBUG
12734         mutex_enter(&oop->oo_lock);
12735         ASSERT(oop->oo_seqid_inuse);
12736         mutex_exit(&oop->oo_lock);
12737 #endif

12739 recov_retry_confirm:
12740         nfs4_error_zinit(ep);
12741         *retry_open = FALSE;

12743         if (resend)
12744                 args.ctag = TAG_OPEN_CONFIRM_LOST;
12745         else
12746                 args.ctag = TAG_OPEN_CONFIRM;

12748         args.array_len = 2;
12749         args.array = argop;

12751         /* putfh target fh */
12752         argop[0].argop = OP_CPUTFH;
12753         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(vp)->r_fh;

12755         argop[1].argop = OP_OPEN_CONFIRM;
```



```
12756         open_confirm_args = &argop[1].nfs_argop4_u.opopen_confirm;

12758         (*seqid) += 1;
12759         open_confirm_args->seqid = *seqid;
12760         open_confirm_args->open_stateid = *stateid;

12762         mi = VTOMI4(vp);

12764         rfs4call(mi, &args, &res, cr, &doqueue, 0, ep);

12766         if (!ep->error && nfs4_need_to_bump_seqid(&res)) {
12767                 nfs4_set_open_seqid((*seqid), oop, args.ctag);
12768         }

12770         needrecov = nfs4_needs_recovery(ep, FALSE, mi->mi_vfsp);
12771         if (!needrecov && ep->error)
12772                 return;

12774         if (needrecov) {
12775                 bool_t abort = FALSE;

12777                 if (reopening_file == FALSE) {
12778                         nfs4_bseqid_entry_t *bsep = NULL;

12780                         if (!ep->error && res.status == NFS4ERR_BAD_SEQID)
12781                                 bsep = nfs4_create_bseqid_entry(oop, NULL,
12782                                     vp, 0, args.ctag,
12783                                     open_confirm_args->seqid);

12785                         abort = nfs4_start_recovery(ep, VTOMI4(vp), vp, NULL,
12786                             NULL, NULL, OP_OPEN_CONFIRM, bsep, NULL, NULL);
12787                         if (bsep) {
12788                                 kmem_free(bsep, sizeof (*bsep));
12789                                 if (num_bseqid_retryp &&
12790                                     --(*num_bseqid_retryp) == 0)
12791                                         abort = TRUE;
12792                         }
12793                 }
12794                 if ((ep->error == ETIMEDOUT ||
12795                     res.status == NFS4ERR_RESOURCE) &&
12796                     abort == FALSE && resend == FALSE) {
12797                         if (!ep->error)
12798                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
12799                                     (caddr_t)&res);

12801                         delay(SEC_TO_TICK(confirm_retry_sec));
12802                         goto recov_retry_confirm;
12803                 }
12804                 /* State may have changed so retry the entire OPEN op */
12805                 if (abort == FALSE)
12806                         *retry_open = TRUE;
12807                 else
12808                         *retry_open = FALSE;
12809                 if (!ep->error)
12810                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
12811                 return;
12812         }

12814         if (res.status) {
12815                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
12816                 return;
12817         }

12819         resop = &res.array[1];  /* open confirm res */
12820         bcopy(&resop->nfs_resop4_u.opopen_confirm.open_stateid,
12821             stateid, sizeof (*stateid));
```

```
12823         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
12824 }

12826 /*
12827  * Return the credentials associated with a client state object.  The
12828  * caller is responsible for freeing the credentials.
12829  */

12831 static cred_t *
12832 state_to_cred(nfs4_open_stream_t *osp)
12833 {
12834         cred_t *cr;

12836         /*
12837          * It's ok to not lock the open stream and open owner to get
12838          * the oo_cred since this is only written once (upon creation)
12839          * and will not change.
12840          */
12841         cr = osp->os_open_owner->oo_cred;
12842         crhold(cr);

12844         return (cr);
12845 }

12847 /*
12848  * nfs4_find_sysid
12849  *
12850  * Find the sysid for the knetconfig associated with the given mi.
12851  */
12852 static struct lm_sysid *
12853 nfs4_find_sysid(mntinfo4_t *mi)
12854 {
12855         ASSERT(nfs_zone() == mi->mi_zone);

12857         /*
12858          * Switch from RDMA knconf to original mount knconf
12859          */
12860         return (lm_get_sysid(ORIG_KNCONF(mi), &mi->mi_curr_serv->sv_addr,
12861             mi->mi_curr_serv->sv_hostname, NULL));
12862 }

12864 #ifdef DEBUG
12865 /*
12866  * Return a string version of the call type for easy reading.
12867  */
12868 static char *
12869 nfs4frlock_get_call_type(nfs4_lock_call_type_t ctype)
12870 {
12871         switch (ctype) {
12872         case NFS4_LCK_CTYPE_NORM:
12873                 return ("NORMAL");
12874         case NFS4_LCK_CTYPE_RECLAIM:
12875                 return ("RECLAIM");
12876         case NFS4_LCK_CTYPE_RESEND:
12877                 return ("RESEND");
12878         case NFS4_LCK_CTYPE_REINSTATE:
12879                 return ("REINSTATE");
12880         default:
12881                 cmn_err(CE_PANIC, "nfs4frlock_get_call_type: got illegal "
12882                     "type %d", ctype);
12883                 return ("");
12884         }
12885 }
12886 #endif
```



```
12888 /*
12889  * Map the frlock cmd and lock type to the NFSv4 over-the-wire lock type
12890  * Unlock requests don't have an over-the-wire locktype, so we just return
12891  * something non-threatening.
12892  */

12894 static nfs_lock_type4
12895 flk_to_locktype(int cmd, int l_type)
12896 {
12897         ASSERT(l_type == F_RDLCK || l_type == F_WRLCK || l_type == F_UNLCK);

12899         switch (l_type) {
12900         case F_UNLCK:
12901                 return (READ_LT);
12902         case F_RDLCK:
12903                 if (cmd == F_SETLK)
12904                         return (READ_LT);
12905                 else
12906                         return (READW_LT);
12907         case F_WRLCK:
12908                 if (cmd == F_SETLK)
12909                         return (WRITE_LT);
12910                 else
12911                         return (WRITEW_LT);
12912         }
12913         panic("flk_to_locktype");
12914         /*NOTREACHED*/
12915 }

12917 /*
12918  * Do some preliminary checks for nfs4frlock.
12919  */
12920 static int
12921 nfs4frlock_validate_args(int cmd, flock64_t *flk, int flag, vnode_t *vp,
12922     u_offset_t offset)
12923 {
12924         int error = 0;

12926         /*
12927          * If we are setting a lock, check that the file is opened
12928          * with the correct mode.
12929          */
12930         if (cmd == F_SETLK || cmd == F_SETLKW) {
12931                 if ((flk->l_type == F_RDLCK && (flag & FREAD) == 0) ||
12932                     (flk->l_type == F_WRLCK && (flag & FWRITE) == 0)) {
12933                         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
12934                             "nfs4frlock_validate_args: file was opened with "
12935                             "incorrect mode"));
12936                         return (EBADF);
12937                 }
12938         }

12940         /* Convert the offset. It may need to be restored before returning. */
12941         if (error = convoff(vp, flk, 0, offset)) {
12942                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
12943                     "nfs4frlock_validate_args: convoff  =>  error= %d\n",
12944                     error));
12945                 return (error);
12946         }

12948         return (error);
12949 }

12951 /*
12952  * Set the flock64's lm_sysid for nfs4frlock.
12953  */
```

```
12954 static int
12955 nfs4frlock_get_sysid(struct lm_sysid **lspp, vnode_t *vp, flock64_t *flk)
12956 {
12957         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12959         /* Find the lm_sysid */
12960         *lspp = nfs4_find_sysid(VTOMI4(vp));

12962         if (*lspp == NULL) {
12963                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
12964                     "nfs4frlock_get_sysid: no sysid, return ENOLCK"));
12965                 return (ENOLCK);
12966         }

12968         flk->l_sysid = lm_sysidt(*lspp);

12970         return (0);
12971 }

12973 /*
12974  * Do the remaining preliminary setup for nfs4frlock.
12975  */
12976 static void
12977 nfs4frlock_pre_setup(clock_t *tick_delayp, nfs4_recov_state_t *recov_statep,
12978     flock64_t *flk, short *whencep, vnode_t *vp, cred_t *search_cr,
12979     cred_t **cred_otw)
12980 {
12981         /*
12982          * set tick_delay to the base delay time.
12983          * (NFS4_BASE_WAIT_TIME is in secs)
12984          */

12986         *tick_delayp = drv_usectohz(NFS4_BASE_WAIT_TIME * 1000 * 1000);

12988         /*
12989          * If lock is relative to EOF, we need the newest length of the
12990          * file. Therefore invalidate the ATTR_CACHE.
12991          */

12993         *whencep = flk->l_whence;

12995         if (*whencep == 2)              /* SEEK_END */
12996                 PURGE_ATTRCACHE4(vp);

12998         recov_statep->rs_flags = 0;
12999         recov_statep->rs_num_retry_despite_err = 0;
13000         *cred_otw = nfs4_get_otw_cred(search_cr, VTOMI4(vp), NULL);
13001 }

13003 /*
13004  * Initialize and allocate the data structures necessary for
13005  * the nfs4frlock call.
13006  * Allocates argsp's op array, frees up the saved_rqstpp if there is one.
13007  */
13008 static void
13009 nfs4frlock_call_init(COMPOUND4args_clnt *argsp, COMPOUND4args_clnt **argspp,
13010     nfs_argop4 **argopp, nfs4_op_hint_t *op_hintp, flock64_t *flk, int cmd,
13011     bool_t *retry, bool_t *did_start_fop, COMPOUND4res_clnt **respp,
13012     bool_t *skip_get_err, nfs4_lost_rqst_t *lost_rqstp)
13013 {
13014         int             argoplist_size;
13015         int             num_ops = 2;

13017         *retry = FALSE;
13018         *did_start_fop = FALSE;
13019         *skip_get_err = FALSE;
```



```
13020         lost_rqstp->lr_op = 0;
13021         argoplist_size  = num_ops * sizeof (nfs_argop4);
13022         /* fill array with zero */
13023         *argopp = kmem_zalloc(argoplist_size, KM_SLEEP);

13025         *argspp = argsp;
13026         *respp = NULL;

13028         argsp->array_len = num_ops;
13029         argsp->array = *argopp;

13031         /* initialize in case of error; will get real value down below */
13032         argsp->ctag = TAG_NONE;

13034         if ((cmd == F_SETLK || cmd == F_SETLKW) && flk->l_type == F_UNLCK)
13035                 *op_hintp = OH_LOCKU;
13036         else
13037                 *op_hintp = OH_OTHER;
13038 }

13040 /*
13041  * Call the nfs4_start_fop() for nfs4frlock, if necessary.  Assign
13042  * the proper nfs4_server_t for this instance of nfs4frlock.
13043  * Returns 0 (success) or an errno value.
13044  */
13045 static int
13046 nfs4frlock_start_call(nfs4_lock_call_type_t ctype, vnode_t *vp,
13047     nfs4_op_hint_t op_hint, nfs4_recov_state_t *recov_statep,
13048     bool_t *did_start_fop, bool_t *startrecovp)
13049 {
13050         int error = 0;
13051         rnode4_t *rp;

13053         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

13055         if (ctype == NFS4_LCK_CTYPE_NORM) {
13056                 error = nfs4_start_fop(VTOMI4(vp), vp, NULL, op_hint,
13057                     recov_statep, startrecovp);
13058                 if (error)
13059                         return (error);
13060                 *did_start_fop = TRUE;
13061         } else {
13062                 *did_start_fop = FALSE;
13063                 *startrecovp = FALSE;
13064         }

13066         if (!error) {
13067                 rp = VTOR4(vp);

13069                 /* If the file failed recovery, just quit. */
13070                 mutex_enter(&rp->r_statelock);
13071                 if (rp->r_flags & R4RECOVERR) {
13072                         error = EIO;
13073                 }
13074                 mutex_exit(&rp->r_statelock);
13075         }

13077         return (error);
13078 }

13080 /*
13081  * Setup the LOCK4/LOCKU4 arguments for resending a lost lock request.  A
13082  * resend nfs4frlock call is initiated by the recovery framework.
13083  * Acquires the lop and oop seqid synchronization.
13084  */
13085 static void
```

```
13086 nfs4frlock_setup_resend_lock_args(nfs4_lost_rqst_t *resend_rqstp,
13087     COMPOUND4args_clnt *argsp, nfs_argop4 *argop, nfs4_lock_owner_t **lopp,
13088     nfs4_open_owner_t **oopp, nfs4_open_stream_t **ospp,
13089     LOCK4args **lock_argsp, LOCKU4args **locku_argsp)
13090 {
13091         mntinfo4_t *mi = VTOMI4(resend_rqstp->lr_vp);
13092         int error;

13094         NFS4_DEBUG((nfs4_lost_rqst_debug || nfs4_client_lock_debug),
13095             (CE_NOTE,
13096             "nfs4frlock_setup_resend_lock_args: have lost lock to resend"));
13097         ASSERT(resend_rqstp != NULL);
13098         ASSERT(resend_rqstp->lr_op == OP_LOCK ||
13099             resend_rqstp->lr_op == OP_LOCKU);

13101         *oopp = resend_rqstp->lr_oop;
13102         if (resend_rqstp->lr_oop) {
13103                 open_owner_hold(resend_rqstp->lr_oop);
13104                 error = nfs4_start_open_seqid_sync(resend_rqstp->lr_oop, mi);
13105                 ASSERT(error == 0);     /* recov thread always succeeds */
13106         }

13108         /* Must resend this lost lock/locku request. */
13109         ASSERT(resend_rqstp->lr_lop != NULL);
13110         *lopp = resend_rqstp->lr_lop;
13111         lock_owner_hold(resend_rqstp->lr_lop);
13112         error = nfs4_start_lock_seqid_sync(resend_rqstp->lr_lop, mi);
13113         ASSERT(error == 0);     /* recov thread always succeeds */

13115         *ospp = resend_rqstp->lr_osp;
13116         if (*ospp)
13117                 open_stream_hold(resend_rqstp->lr_osp);

13119         if (resend_rqstp->lr_op == OP_LOCK) {
13120                 LOCK4args *lock_args;

13122                 argop->argop = OP_LOCK;
13123                 *lock_argsp = lock_args = &argop->nfs_argop4_u.oplock;
13124                 lock_args->locktype = resend_rqstp->lr_locktype;
13125                 lock_args->reclaim =
13126                     (resend_rqstp->lr_ctype == NFS4_LCK_CTYPE_RECLAIM);
13127                 lock_args->offset = resend_rqstp->lr_flk->l_start;
13128                 lock_args->length = resend_rqstp->lr_flk->l_len;
13129                 if (lock_args->length == 0)
13130                         lock_args->length = ~lock_args->length;
13131                 nfs4_setup_lock_args(*lopp, *oopp, *ospp,
13132                     mi2clientid(mi), &lock_args->locker);

13134                 switch (resend_rqstp->lr_ctype) {
13135                 case NFS4_LCK_CTYPE_RESEND:
13136                         argsp->ctag = TAG_LOCK_RESEND;
13137                         break;
13138                 case NFS4_LCK_CTYPE_REINSTATE:
13139                         argsp->ctag = TAG_LOCK_REINSTATE;
13140                         break;
13141                 case NFS4_LCK_CTYPE_RECLAIM:
13142                         argsp->ctag = TAG_LOCK_RECLAIM;
13143                         break;
13144                 default:
13145                         argsp->ctag = TAG_LOCK_UNKNOWN;
13146                         break;
13147                 }
13148         } else {
13149                 LOCKU4args *locku_args;
13150                 nfs4_lock_owner_t *lop = resend_rqstp->lr_lop;
```



```
13152                 argop->argop = OP_LOCKU;
13153                 *locku_argsp = locku_args = &argop->nfs_argop4_u.oplocku;
13154                 locku_args->locktype = READ_LT;
13155                 locku_args->seqid = lop->lock_seqid + 1;
13156                 mutex_enter(&lop->lo_lock);
13157                 locku_args->lock_stateid = lop->lock_stateid;
13158                 mutex_exit(&lop->lo_lock);
13159                 locku_args->offset = resend_rqstp->lr_flk->l_start;
13160                 locku_args->length = resend_rqstp->lr_flk->l_len;
13161                 if (locku_args->length == 0)
13162                         locku_args->length = ~locku_args->length;

13164                 switch (resend_rqstp->lr_ctype) {
13165                 case NFS4_LCK_CTYPE_RESEND:
13166                         argsp->ctag = TAG_LOCKU_RESEND;
13167                         break;
13168                 case NFS4_LCK_CTYPE_REINSTATE:
13169                         argsp->ctag = TAG_LOCKU_REINSTATE;
13170                         break;
13171                 default:
13172                         argsp->ctag = TAG_LOCK_UNKNOWN;
13173                         break;
13174                 }
13175         }
13176 }

13178 /*
13179  * Setup the LOCKT4 arguments.
13180  */
13181 static void
13182 nfs4frlock_setup_lockt_args(nfs4_lock_call_type_t ctype, nfs_argop4 *argop,
13183     LOCKT4args **lockt_argsp, COMPOUND4args_clnt *argsp, flock64_t *flk,
13184     rnode4_t *rp)
13185 {
13186         LOCKT4args *lockt_args;

13188         ASSERT(nfs_zone() == VTOMI4(RTOV4(rp))->mi_zone);
13189         ASSERT(ctype == NFS4_LCK_CTYPE_NORM);
13190         argop->argop = OP_LOCKT;
13191         argsp->ctag = TAG_LOCKT;
13192         lockt_args = &argop->nfs_argop4_u.oplockt;

13194         /*
13195          * The locktype will be READ_LT unless it's
13196          * a write lock. We do this because the Solaris
13197          * system call allows the combination of
13198          * F_UNLCK and F_GETLK* and so in that case the
13199          * unlock is mapped to a read.
13200          */
13201         if (flk->l_type == F_WRLCK)
13202                 lockt_args->locktype = WRITE_LT;
13203         else
13204                 lockt_args->locktype = READ_LT;

13206         lockt_args->owner.clientid = mi2clientid(VTOMI4(RTOV4(rp)));
13207         /* set the lock owner4 args */
13208         nfs4_setlockowner_args(&lockt_args->owner, rp,
13209             ctype == NFS4_LCK_CTYPE_NORM ? curproc->p_pidp->pid_id :
13210             flk->l_pid);
13211         lockt_args->offset = flk->l_start;
13212         lockt_args->length = flk->l_len;
13213         if (flk->l_len == 0)
13214                 lockt_args->length = ~lockt_args->length;

13216         *lockt_argsp = lockt_args;
13217 }
```

```
13219 /*
13220  * If the client is holding a delegation, and the open stream to be used
13221  * with this lock request is a delegation open stream, then re-open the stream.
13222  * Sets the nfs4_error_t to all zeros unless the open stream has already
13223  * failed a reopen or we couldn't find the open stream.  NFS4ERR_DELAY
13224  * means the caller should retry (like a recovery retry).
13225  */
13226 static void
13227 nfs4frlock_check_deleg(vnode_t *vp, nfs4_error_t *ep, cred_t *cr, int lt)
13228 {
13229         open_delegation_type4   dt;
13230         bool_t                  reopen_needed, force;
13231         nfs4_open_stream_t      *osp;
13232         open_claim_type4        oclaim;
13233         rnode4_t                *rp = VTOR4(vp);
13234         mntinfo4_t              *mi = VTOMI4(vp);

13236         ASSERT(nfs_zone() == mi->mi_zone);

13238         nfs4_error_zinit(ep);

13240         mutex_enter(&rp->r_statev4_lock);
13241         dt = rp->r_deleg_type;
13242         mutex_exit(&rp->r_statev4_lock);

13244         if (dt != OPEN_DELEGATE_NONE) {
13245                 nfs4_open_owner_t       *oop;

13247                 oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
13248                 if (!oop) {
13249                         ep->stat = NFS4ERR_IO;
13250                         return;
13251                 }
13252                 /* returns with 'os_sync_lock' held */
13253                 osp = find_open_stream(oop, rp);
13254                 if (!osp) {
13255                         open_owner_rele(oop);
13256                         ep->stat = NFS4ERR_IO;
13257                         return;
13258                 }

13260                 if (osp->os_failed_reopen) {
13261                         NFS4_DEBUG((nfs4_open_stream_debug ||
13262                             nfs4_client_lock_debug), (CE_NOTE,
13263                             "nfs4frlock_check_deleg: os_failed_reopen set "
13264                             "for osp %p, cr %p, rp %s", (void *)osp,
13265                             (void *)cr, rnode4info(rp)));
13266                         mutex_exit(&osp->os_sync_lock);
13267                         open_stream_rele(osp, rp);
13268                         open_owner_rele(oop);
13269                         ep->stat = NFS4ERR_IO;
13270                         return;
13271                 }

13273                 /*
13274                  * Determine whether a reopen is needed.  If this
13275                  * is a delegation open stream, then send the open
13276                  * to the server to give visibility to the open owner.
13277                  * Even if it isn't a delegation open stream, we need
13278                  * to check if the previous open CLAIM_DELEGATE_CUR
13279                  * was sufficient.
13280                  */

13282                 reopen_needed = osp->os_delegation ||
13283                     ((lt == F_RDLCK &&
```



```
13284                       !(osp->os_dc_openacc & OPEN4_SHARE_ACCESS_READ)) ||
13285                       (lt == F_WRLCK &&
13286                       !(osp->os_dc_openacc & OPEN4_SHARE_ACCESS_WRITE)));

13288                 mutex_exit(&osp->os_sync_lock);
13289                 open_owner_rele(oop);

13291                 if (reopen_needed) {
13292                         /*
13293                          * Always use CLAIM_PREVIOUS after server reboot.
13294                          * The server will reject CLAIM_DELEGATE_CUR if
13295                          * it is used during the grace period.
13296                          */
13297                         mutex_enter(&mi->mi_lock);
13298                         if (mi->mi_recovflags & MI4R_SRV_REBOOT) {
13299                                 oclaim = CLAIM_PREVIOUS;
13300                                 force = TRUE;
13301                         } else {
13302                                 oclaim = CLAIM_DELEGATE_CUR;
13303                                 force = FALSE;
13304                         }
13305                         mutex_exit(&mi->mi_lock);

13307                         nfs4_reopen(vp, osp, ep, oclaim, force, FALSE);
13308                         if (ep->error == EAGAIN) {
13309                                 nfs4_error_zinit(ep);
13310                                 ep->stat = NFS4ERR_DELAY;
13311                         }
13312                 }
13313                 open_stream_rele(osp, rp);
13314                 osp = NULL;
13315         }
13316 }

13318 /*
13319  * Setup the LOCKU4 arguments.
13320  * Returns errors via the nfs4_error_t.
13321  * NFS4_OK              no problems.  *go_otwp is TRUE if call should go
13322  *                      over-the-wire.  The caller must release the
13323  *                      reference on *lopp.
13324  * NFS4ERR_DELAY        caller should retry (like recovery retry)
13325  * (other)              unrecoverable error.
13326  */
13327 static void
13328 nfs4frlock_setup_locku_args(nfs4_lock_call_type_t ctype, nfs_argop4 *argop,
13329     LOCKU4args **locku_argsp, flock64_t *flk,
13330     nfs4_lock_owner_t **lopp, nfs4_error_t *ep, COMPOUND4args_clnt *argsp,
13331     vnode_t *vp, int flag, u_offset_t offset, cred_t *cr,
13332     bool_t *skip_get_err, bool_t *go_otwp)
13333 {
13334         nfs4_lock_owner_t       *lop = NULL;
13335         LOCKU4args              *locku_args;
13336         pid_t                   pid;
13337         bool_t                  is_spec = FALSE;
13338         rnode4_t                *rp = VTOR4(vp);

13340         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
13341         ASSERT(ctype == NFS4_LCK_CTYPE_NORM);

13343         nfs4frlock_check_deleg(vp, ep, cr, F_UNLCK);
13344         if (ep->error || ep->stat)
13345                 return;

13347         argop->argop = OP_LOCKU;
13348         if (ctype == NFS4_LCK_CTYPE_REINSTATE)
13349                 argsp->ctag = TAG_LOCKU_REINSTATE;
```

```
13350         else
13351                 argsp->ctag = TAG_LOCKU;
13352         locku_args = &argop->nfs_argop4_u.oplocku;
13353         *locku_argsp = locku_args;

13355         /*
13356          * XXX what should locku_args->locktype be?
13357          * setting to ALWAYS be READ_LT so at least
13358          * it is a valid locktype.
13359          */

13361         locku_args->locktype = READ_LT;

13363         pid = ctype == NFS4_LCK_CTYPE_NORM ? curproc->p_pidp->pid_id :
13364             flk->l_pid;

13366         /*
13367          * Get the lock owner stateid.  If no lock owner
13368          * exists, return success.
13369          */
13370         lop = find_lock_owner(rp, pid, LOWN_ANY);
13371         *lopp = lop;
13372         if (lop && CLNT_ISSPECIAL(&lop->lock_stateid))
13373                 is_spec = TRUE;
13374         if (!lop || is_spec) {
13375                 /*
13376                  * No lock owner so no locks to unlock.
13377                  * Return success.  If there was a failed
13378                  * reclaim earlier, the lock might still be
13379                  * registered with the local locking code,
13380                  * so notify it of the unlock.
13381                  *
13382                  * If the lockowner is using a special stateid,
13383                  * then the original lock request (that created
13384                  * this lockowner) was never successful, so we
13385                  * have no lock to undo OTW.
13386                  */
13387                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
13388                     "nfs4frlock_setup_locku_args: LOCKU: no lock owner "
13389                     "(%ld) so return success", (long)pid));

13391                 if (ctype == NFS4_LCK_CTYPE_NORM)
13392                         flk->l_pid = curproc->p_pid;
13393                 nfs4_register_lock_locally(vp, flk, flag, offset);
13394                 /*
13395                  * Release our hold and NULL out so final_cleanup
13396                  * doesn't try to end a lock seqid sync we
13397                  * never started.
13398                  */
13399                 if (is_spec) {
13400                         lock_owner_rele(lop);
13401                         *lopp = NULL;
13402                 }
13403                 *skip_get_err = TRUE;
13404                 *go_otwp = FALSE;
13405                 return;
13406         }

13408         ep->error = nfs4_start_lock_seqid_sync(lop, VTOMI4(vp));
13409         if (ep->error == EAGAIN) {
13410                 lock_owner_rele(lop);
13411                 *lopp = NULL;
13412                 return;
13413         }

13415         mutex_enter(&lop->lo_lock);
```



```
13416         locku_args->lock_stateid = lop->lock_stateid;
13417         mutex_exit(&lop->lo_lock);
13418         locku_args->seqid = lop->lock_seqid + 1;

13420         /* leave the ref count on lop, rele after RPC call */

13422         locku_args->offset = flk->l_start;
13423         locku_args->length = flk->l_len;
13424         if (flk->l_len == 0)
13425                 locku_args->length = ~locku_args->length;

13427         *go_otwp = TRUE;
13428 }

13430 /*
13431  * Setup the LOCK4 arguments.
13432  *
13433  * Returns errors via the nfs4_error_t.
13434  * NFS4_OK              no problems
13435  * NFS4ERR_DELAY        caller should retry (like recovery retry)
13436  * (other)              unrecoverable error
13437  */
13438 static void
13439 nfs4frlock_setup_lock_args(nfs4_lock_call_type_t ctype, LOCK4args **lock_argsp,
13440     nfs4_open_owner_t **oopp, nfs4_open_stream_t **ospp,
13441     nfs4_lock_owner_t **lopp, nfs_argop4 *argop, COMPOUND4args_clnt *argsp,
13442     flock64_t *flk, int cmd, vnode_t *vp, cred_t *cr, nfs4_error_t *ep)
13443 {
13444         LOCK4args               *lock_args;
13445         nfs4_open_owner_t       *oop = NULL;
13446         nfs4_open_stream_t      *osp = NULL;
13447         nfs4_lock_owner_t       *lop = NULL;
13448         pid_t                   pid;
13449         rnode4_t                *rp = VTOR4(vp);

13451         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

13453         nfs4frlock_check_deleg(vp, ep, cr, flk->l_type);
13454         if (ep->error || ep->stat != NFS4_OK)
13455                 return;

13457         argop->argop = OP_LOCK;
13458         if (ctype == NFS4_LCK_CTYPE_NORM)
13459                 argsp->ctag = TAG_LOCK;
13460         else if (ctype == NFS4_LCK_CTYPE_RECLAIM)
13461                 argsp->ctag = TAG_RELOCK;
13462         else
13463                 argsp->ctag = TAG_LOCK_REINSTATE;
13464         lock_args = &argop->nfs_argop4_u.oplock;
13465         lock_args->locktype = flk_to_locktype(cmd, flk->l_type);
13466         lock_args->reclaim = ctype == NFS4_LCK_CTYPE_RECLAIM ? 1 : 0;
13467         /*
13468          * Get the lock owner.  If no lock owner exists,
13469          * create a 'temporary' one and grab the open seqid
13470          * synchronization (which puts a hold on the open
13471          * owner and open stream).
13472          * This also grabs the lock seqid synchronization.
13473          */
13474         pid = ctype == NFS4_LCK_CTYPE_NORM ? curproc->p_pid : flk->l_pid;
13475         ep->stat =
13476             nfs4_find_or_create_lock_owner(pid, rp, cr, &oop, &osp, &lop);

13478         if (ep->stat != NFS4_OK)
13479                 goto out;

13481         nfs4_setup_lock_args(lop, oop, osp, mi2clientid(VTOMI4(vp)),
```

```
13482             &lock_args->locker);

13484         lock_args->offset = flk->l_start;
13485         lock_args->length = flk->l_len;
13486         if (flk->l_len == 0)
13487                 lock_args->length = ~lock_args->length;
13488         *lock_argsp = lock_args;
13489 out:
13490         *oopp = oop;
13491         *ospp = osp;
13492         *lopp = lop;
13493 }

13495 /*
13496  * After we get the reply from the server, record the proper information
13497  * for possible resend lock requests.
13498  *
13499  * Allocates memory for the saved_rqstp if we have a lost lock to save.
13500  */
13501 static void
13502 nfs4frlock_save_lost_rqst(nfs4_lock_call_type_t ctype, int error,
13503     nfs_lock_type4 locktype, nfs4_open_owner_t *oop,
13504     nfs4_open_stream_t *osp, nfs4_lock_owner_t *lop, flock64_t *flk,
13505     nfs4_lost_rqst_t *lost_rqstp, cred_t *cr, vnode_t *vp)
13506 {
13507         bool_t unlock = (flk->l_type == F_UNLCK);

13509         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
13510         ASSERT(ctype == NFS4_LCK_CTYPE_NORM ||
13511             ctype == NFS4_LCK_CTYPE_REINSTATE);

13513         if (error != 0 && !unlock) {
13514                 NFS4_DEBUG((nfs4_lost_rqst_debug ||
13515                     nfs4_client_lock_debug), (CE_NOTE,
13516                     "nfs4frlock_save_lost_rqst: set lo_pending_rqsts to 1 "
13517                     " for lop %p", (void *)lop));
13518                 ASSERT(lop != NULL);
13519                 mutex_enter(&lop->lo_lock);
13520                 lop->lo_pending_rqsts = 1;
13521                 mutex_exit(&lop->lo_lock);
13522         }

13524         lost_rqstp->lr_putfirst = FALSE;
13525         lost_rqstp->lr_op = 0;

13527         /*
13528          * For lock/locku requests, we treat EINTR as ETIMEDOUT for
13529          * recovery purposes so that the lock request that was sent
13530          * can be saved and re-issued later.  Ditto for EIO from a forced
13531          * unmount.  This is done to have the client's local locking state
13532          * match the v4 server's state; that is, the request was
13533          * potentially received and accepted by the server but the client
13534          * thinks it was not.
13535          */
13536         if (error == ETIMEDOUT || error == EINTR ||
13537             NFS4_FRC_UNMT_ERR(error, vp->v_vfsp)) {
13538                 NFS4_DEBUG((nfs4_lost_rqst_debug ||
13539                     nfs4_client_lock_debug), (CE_NOTE,
13540                     "nfs4frlock_save_lost_rqst: got a lost %s lock for "
13541                     "lop %p oop %p osp %p", unlock ? "LOCKU" : "LOCK",
13542                     (void *)lop, (void *)oop, (void *)osp));
13543                 if (unlock)
13544                         lost_rqstp->lr_op = OP_LOCKU;
13545                 else {
13546                         lost_rqstp->lr_op = OP_LOCK;
13547                         lost_rqstp->lr_locktype = locktype;
```



```
13548                 }
13549                 /*
13550                  * Objects are held and rele'd via the recovery code.
13551                  * See nfs4_save_lost_rqst.
13552                  */
13553                 lost_rqstp->lr_vp = vp;
13554                 lost_rqstp->lr_dvp = NULL;
13555                 lost_rqstp->lr_oop = oop;
13556                 lost_rqstp->lr_osp = osp;
13557                 lost_rqstp->lr_lop = lop;
13558                 lost_rqstp->lr_cr = cr;
13559                 switch (ctype) {
13560                 case NFS4_LCK_CTYPE_NORM:
13561                         flk->l_pid = ttoproc(curthread)->p_pid;
13562                         lost_rqstp->lr_ctype = NFS4_LCK_CTYPE_RESEND;
13563                         break;
13564                 case NFS4_LCK_CTYPE_REINSTATE:
13565                         lost_rqstp->lr_putfirst = TRUE;
13566                         lost_rqstp->lr_ctype = ctype;
13567                         break;
13568                 default:
13569                         break;
13570                 }
13571                 lost_rqstp->lr_flk = flk;
13572         }
13573 }

13575 /*
13576  * Update lop's seqid.  Also update the seqid stored in a resend request,
13577  * if any.  (Some recovery errors increment the seqid, and we may have to
13578  * send the resend request again.)
13579  */

13581 static void
13582 nfs4frlock_bump_seqid(LOCK4args *lock_args, LOCKU4args *locku_args,
13583     nfs4_open_owner_t *oop, nfs4_lock_owner_t *lop, nfs4_tag_type_t tag_type)
13584 {
13585         if (lock_args) {
13586                 if (lock_args->locker.new_lock_owner == TRUE)
13587                         nfs4_get_and_set_next_open_seqid(oop, tag_type);
13588                 else {
13589                         ASSERT(lop->lo_flags & NFS4_LOCK_SEQID_INUSE);
13590                         nfs4_set_lock_seqid(lop->lock_seqid + 1, lop);
13591                 }
13592         } else if (locku_args) {
13593                 ASSERT(lop->lo_flags & NFS4_LOCK_SEQID_INUSE);
13594                 nfs4_set_lock_seqid(lop->lock_seqid +1, lop);
13595         }
13596 }

13598 /*
13599  * Calls nfs4_end_fop, drops the seqid syncs, and frees up the
13600  * COMPOUND4 args/res for calls that need to retry.
13601  * Switches the *cred_otwp to base_cr.
13602  */
13603 static void
13604 nfs4frlock_check_access(vnode_t *vp, nfs4_op_hint_t op_hint,
13605     nfs4_recov_state_t *recov_statep, int needrecov, bool_t *did_start_fop,
13606     COMPOUND4args_clnt **argspp, COMPOUND4res_clnt **respp, int error,
13607     nfs4_lock_owner_t **lopp, nfs4_open_owner_t **oopp,
13608     nfs4_open_stream_t **ospp, cred_t *base_cr, cred_t **cred_otwp)
13609 {
13610         nfs4_open_owner_t       *oop = *oopp;
13611         nfs4_open_stream_t      *osp = *ospp;
13612         nfs4_lock_owner_t       *lop = *lopp;
13613         nfs_argop4              *argop = (*argspp)->array;
```

```
13615         if (*did_start_fop) {
13616                 nfs4_end_fop(VTOMI4(vp), vp, NULL, op_hint, recov_statep,
13617                     needrecov);
13618                 *did_start_fop = FALSE;
13619         }
13620         ASSERT((*argspp)->array_len == 2);
13621         if (argop[1].argop == OP_LOCK)
13622                 nfs4args_lock_free(&argop[1]);
13623         else if (argop[1].argop == OP_LOCKT)
13624                 nfs4args_lockt_free(&argop[1]);
13625         kmem_free(argop, 2 * sizeof (nfs_argop4));
13626         if (!error)
13627                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)*respp);
13628         *argspp = NULL;
13629         *respp = NULL;

13631         if (lop) {
13632                 nfs4_end_lock_seqid_sync(lop);
13633                 lock_owner_rele(lop);
13634                 *lopp = NULL;
13635         }

13637         /* need to free up the reference on osp for lock args */
13638         if (osp != NULL) {
13639                 open_stream_rele(osp, VTOR4(vp));
13640                 *ospp = NULL;
13641         }

13643         /* need to free up the reference on oop for lock args */
13644         if (oop != NULL) {
13645                 nfs4_end_open_seqid_sync(oop);
13646                 open_owner_rele(oop);
13647                 *oopp = NULL;
13648         }

13650         crfree(*cred_otwp);
13651         *cred_otwp = base_cr;
13652         crhold(*cred_otwp);
13653 }

13655 /*
13656  * Function to process the client's recovery for nfs4frlock.
13657  * Returns TRUE if we should retry the lock request; FALSE otherwise.
13658  *
13659  * Calls nfs4_end_fop, drops the seqid syncs, and frees up the
13660  * COMPOUND4 args/res for calls that need to retry.
13661  *
13662  * Note: the rp's r_lkserlock is *not* dropped during this path.
13663  */
13664 static bool_t
13665 nfs4frlock_recovery(int needrecov, nfs4_error_t *ep,
13666     COMPOUND4args_clnt **argspp, COMPOUND4res_clnt **respp,
13667     LOCK4args *lock_args, LOCKU4args *locku_args,
13668     nfs4_open_owner_t **oopp, nfs4_open_stream_t **ospp,
13669     nfs4_lock_owner_t **lopp, rnode4_t *rp, vnode_t *vp,
13670     nfs4_recov_state_t *recov_statep, nfs4_op_hint_t op_hint,
13671     bool_t *did_start_fop, nfs4_lost_rqst_t *lost_rqstp, flock64_t *flk)
13672 {
13673         nfs4_open_owner_t       *oop = *oopp;
13674         nfs4_open_stream_t      *osp = *ospp;
13675         nfs4_lock_owner_t       *lop = *lopp;

13677         bool_t abort, retry;

13679         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
```



```
13680         ASSERT((*argspp) != NULL);
13681         ASSERT((*respp) != NULL);
13682         if (lock_args || locku_args)
13683                 ASSERT(lop != NULL);

13685         NFS4_DEBUG((nfs4_client_lock_debug || nfs4_client_recov_debug),
13686             (CE_NOTE, "nfs4frlock_recovery: initiating recovery\n"));

13688         retry = TRUE;
13689         abort = FALSE;
13690         if (needrecov) {
13691                 nfs4_bseqid_entry_t *bsep = NULL;
13692                 nfs_opnum4 op;

13694                 op = lock_args ? OP_LOCK : locku_args ? OP_LOCKU : OP_LOCKT;

13696                 if (!ep->error && ep->stat == NFS4ERR_BAD_SEQID) {
13697                         seqid4 seqid;

13699                         if (lock_args) {
13700                                 if (lock_args->locker.new_lock_owner == TRUE)
13701                                         seqid = lock_args->locker.locker4_u.
13702                                             open_owner.open_seqid;
13703                                 else
13704                                         seqid = lock_args->locker.locker4_u.
13705                                             lock_owner.lock_seqid;
13706                         } else if (locku_args) {
13707                                 seqid = locku_args->seqid;
13708                         } else {
13709                                 seqid = 0;
13710                         }

13712                         bsep = nfs4_create_bseqid_entry(oop, lop, vp,
13713                             flk->l_pid, (*argspp)->ctag, seqid);
13714                 }

13716                 abort = nfs4_start_recovery(ep, VTOMI4(vp), vp, NULL, NULL,
13717                     (lost_rqstp && (lost_rqstp->lr_op == OP_LOCK ||
13718                     lost_rqstp->lr_op == OP_LOCKU)) ? lost_rqstp :
13719                     NULL, op, bsep, NULL, NULL);

13721                 if (bsep)
13722                         kmem_free(bsep, sizeof (*bsep));
13723         }

13725         /*
13726          * Return that we do not want to retry the request for 3 cases:
13727          * 1. If we received EINTR or are bailing out because of a forced
13728          *    unmount, we came into this code path just for the sake of
13729          *    initiating recovery, we now need to return the error.
13730          * 2. If we have aborted recovery.
13731          * 3. We received NFS4ERR_BAD_SEQID.
13732          */
13733         if (ep->error == EINTR || NFS4_FRC_UNMT_ERR(ep->error, vp->v_vfsp) ||
13734             abort == TRUE || (ep->error == 0 && ep->stat == NFS4ERR_BAD_SEQID))
13735                 retry = FALSE;

13737         if (*did_start_fop == TRUE) {
13738                 nfs4_end_fop(VTOMI4(vp), vp, NULL, op_hint, recov_statep,
13739                     needrecov);
13740                 *did_start_fop = FALSE;
13741         }

13743         if (retry == TRUE) {
13744                 nfs_argop4      *argop;
```

```
13746                 argop = (*argspp)->array;
13747                 ASSERT((*argspp)->array_len == 2);

13749                 if (argop[1].argop == OP_LOCK)
13750                         nfs4args_lock_free(&argop[1]);
13751                 else if (argop[1].argop == OP_LOCKT)
13752                         nfs4args_lockt_free(&argop[1]);
13753                 kmem_free(argop, 2 * sizeof (nfs_argop4));
13754                 if (!ep->error)
13755                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)*respp);
13756                 *respp = NULL;
13757                 *argspp = NULL;
13758         }

13760         if (lop != NULL) {
13761                 nfs4_end_lock_seqid_sync(lop);
13762                 lock_owner_rele(lop);
13763         }

13765         *lopp = NULL;

13767         /* need to free up the reference on osp for lock args */
13768         if (osp != NULL) {
13769                 open_stream_rele(osp, rp);
13770                 *ospp = NULL;
13771         }

13773         /* need to free up the reference on oop for lock args */
13774         if (oop != NULL) {
13775                 nfs4_end_open_seqid_sync(oop);
13776                 open_owner_rele(oop);
13777                 *oopp = NULL;
13778         }

13780         return (retry);
13781 }

13783 /*
13784  * Handles the successful reply from the server for nfs4frlock.
13785  */
13786 static void
13787 nfs4frlock_results_ok(nfs4_lock_call_type_t ctype, int cmd, flock64_t *flk,
13788     vnode_t *vp, int flag, u_offset_t offset,
13789     nfs4_lost_rqst_t *resend_rqstp)
13790 {
13791         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
13792         if ((cmd == F_SETLK || cmd == F_SETLKW) &&
13793             (flk->l_type == F_RDLCK || flk->l_type == F_WRLCK)) {
13794                 if (ctype == NFS4_LCK_CTYPE_NORM) {
13795                         flk->l_pid = ttoproc(curthread)->p_pid;
13796                         /*
13797                          * We do not register lost locks locally in
13798                          * the 'resend' case since the user/application
13799                          * doesn't think we have the lock.
13800                          */
13801                         ASSERT(!resend_rqstp);
13802                         nfs4_register_lock_locally(vp, flk, flag, offset);
13803                 }
13804         }
13805 }

13807 /*
13808  * Handle the DENIED reply from the server for nfs4frlock.
13809  * Returns TRUE if we should retry the request; FALSE otherwise.
13810  *
13811  * Calls nfs4_end_fop, drops the seqid syncs, and frees up the
```



```
13812  * COMPOUND4 args/res for calls that need to retry.  Can also
13813  * drop and regrab the r_lkserlock.
13814  */
13815 static bool_t
13816 nfs4frlock_results_denied(nfs4_lock_call_type_t ctype, LOCK4args *lock_args,
13817     LOCKT4args *lockt_args, nfs4_open_owner_t **oopp,
13818     nfs4_open_stream_t **ospp, nfs4_lock_owner_t **lopp, int cmd,
13819     vnode_t *vp, flock64_t *flk, nfs4_op_hint_t op_hint,
13820     nfs4_recov_state_t *recov_statep, int needrecov,
13821     COMPOUND4args_clnt **argspp, COMPOUND4res_clnt **respp,
13822     clock_t *tick_delayp, short *whencep, int *errorp,
13823     nfs_resop4 *resop, cred_t *cr, bool_t *did_start_fop,
13824     bool_t *skip_get_err)
13825 {
13826         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

13828         if (lock_args) {
13829                 nfs4_open_owner_t       *oop = *oopp;
13830                 nfs4_open_stream_t      *osp = *ospp;
13831                 nfs4_lock_owner_t       *lop = *lopp;
13832                 int                     intr;

13834                 /*
13835                  * Blocking lock needs to sleep and retry from the request.
13836                  *
13837                  * Do not block and wait for 'resend' or 'reinstate'
13838                  * lock requests, just return the error.
13839                  *
13840                  * Note: reclaim requests have cmd == F_SETLK, not F_SETLKW.
13841                  */
13842                 if (cmd == F_SETLKW) {
13843                         rnode4_t *rp = VTOR4(vp);
13844                         nfs_argop4 *argop = (*argspp)->array;

13846                         ASSERT(ctype == NFS4_LCK_CTYPE_NORM);

13848                         nfs4_end_fop(VTOMI4(vp), vp, NULL, op_hint,
13849                             recov_statep, needrecov);
13850                         *did_start_fop = FALSE;
13851                         ASSERT((*argspp)->array_len == 2);
13852                         if (argop[1].argop == OP_LOCK)
13853                                 nfs4args_lock_free(&argop[1]);
13854                         else if (argop[1].argop == OP_LOCKT)
13855                                 nfs4args_lockt_free(&argop[1]);
13856                         kmem_free(argop, 2 * sizeof (nfs_argop4));
13857                         if (*respp)
13858                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
13859                                     (caddr_t)*respp);
13860                         *argspp = NULL;
13861                         *respp = NULL;
13862                         nfs4_end_lock_seqid_sync(lop);
13863                         lock_owner_rele(lop);
13864                         *lopp = NULL;
13865                         if (osp != NULL) {
13866                                 open_stream_rele(osp, rp);
13867                                 *ospp = NULL;
13868                         }
13869                         if (oop != NULL) {
13870                                 nfs4_end_open_seqid_sync(oop);
13871                                 open_owner_rele(oop);
13872                                 *oopp = NULL;
13873                         }

13875                         nfs_rw_exit(&rp->r_lkserlock);

13877                         intr = nfs4_block_and_wait(tick_delayp, rp);
```

```
13879                         if (intr) {
13880                                 (void) nfs_rw_enter_sig(&rp->r_lkserlock,
13881                                     RW_WRITER, FALSE);
13882                                 *errorp = EINTR;
13883                                 return (FALSE);
13884                         }

13886                         (void) nfs_rw_enter_sig(&rp->r_lkserlock,
13887                             RW_WRITER, FALSE);

13889                         /*
13890                          * Make sure we are still safe to lock with
13891                          * regards to mmapping.
13892                          */
13893                         if (!nfs4_safelock(vp, flk, cr)) {
13894                                 *errorp = EAGAIN;
13895                                 return (FALSE);
13896                         }

13898                         return (TRUE);
13899                 }
13900                 if (ctype == NFS4_LCK_CTYPE_NORM)
13901                         *errorp = EAGAIN;
13902                 *skip_get_err = TRUE;
13903                 flk->l_whence = 0;
13904                 *whencep = 0;
13905                 return (FALSE);
13906         } else if (lockt_args) {
13907                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
13908                     "nfs4frlock_results_denied: OP_LOCKT DENIED"));

13910                 denied_to_flk(&resop->nfs_resop4_u.oplockt.denied,
13911                     flk, lockt_args);

13913                 /* according to NLM code */
13914                 *errorp = 0;
13915                 *whencep = 0;
13916                 *skip_get_err = TRUE;
13917                 return (FALSE);
13918         }
13919         return (FALSE);
13920 }

13922 /*
13923  * Handles all NFS4 errors besides NFS4_OK and NFS4ERR_DENIED for nfs4frlock.
13924  */
13925 static void
13926 nfs4frlock_results_default(COMPOUND4res_clnt *resp, int *errorp)
13927 {
13928         switch (resp->status) {
13929         case NFS4ERR_ACCESS:
13930         case NFS4ERR_ADMIN_REVOKED:
13931         case NFS4ERR_BADHANDLE:
13932         case NFS4ERR_BAD_RANGE:
13933         case NFS4ERR_BAD_SEQID:
13934         case NFS4ERR_BAD_STATEID:
13935         case NFS4ERR_BADXDR:
13936         case NFS4ERR_DEADLOCK:
13937         case NFS4ERR_DELAY:
13938         case NFS4ERR_EXPIRED:
13939         case NFS4ERR_FHEXPIRED:
13940         case NFS4ERR_GRACE:
13941         case NFS4ERR_INVAL:
13942         case NFS4ERR_ISDIR:
13943         case NFS4ERR_LEASE_MOVED:
```



```
13944         case NFS4ERR_LOCK_NOTSUPP:
13945         case NFS4ERR_LOCK_RANGE:
13946         case NFS4ERR_MOVED:
13947         case NFS4ERR_NOFILEHANDLE:
13948         case NFS4ERR_NO_GRACE:
13949         case NFS4ERR_OLD_STATEID:
13950         case NFS4ERR_OPENMODE:
13951         case NFS4ERR_RECLAIM_BAD:
13952         case NFS4ERR_RECLAIM_CONFLICT:
13953         case NFS4ERR_RESOURCE:
13954         case NFS4ERR_SERVERFAULT:
13955         case NFS4ERR_STALE:
13956         case NFS4ERR_STALE_CLIENTID:
13957         case NFS4ERR_STALE_STATEID:
13958                 return;
13959         default:
13960                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
13961                     "nfs4frlock_results_default: got unrecognizable "
13962                     "res.status %d", resp->status));
13963                 *errorp = NFS4ERR_INVAL;
13964         }
13965 }

13967 /*
13968  * The lock request was successful, so update the client's state.
13969  */
13970 static void
13971 nfs4frlock_update_state(LOCK4args *lock_args, LOCKU4args *locku_args,
13972     LOCKT4args *lockt_args, nfs_resop4 *resop, nfs4_lock_owner_t *lop,
13973     vnode_t *vp, flock64_t *flk, cred_t *cr,
13974     nfs4_lost_rqst_t *resend_rqstp)
13975 {
13976         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

13978         if (lock_args) {
13979                 LOCK4res *lock_res;

13981                 lock_res = &resop->nfs_resop4_u.oplock;
13982                 /* update the stateid with server's response */

13984                 if (lock_args->locker.new_lock_owner == TRUE) {
13985                         mutex_enter(&lop->lo_lock);
13986                         lop->lo_just_created = NFS4_PERM_CREATED;
13987                         mutex_exit(&lop->lo_lock);
13988                 }

13990                 nfs4_set_lock_stateid(lop, lock_res->LOCK4res_u.lock_stateid);

13992                 /*
13993                  * If the lock was the result of a resending a lost
13994                  * request, we've synched up the stateid and seqid
13995                  * with the server, but now the server might be out of sync
13996                  * with what the application thinks it has for locks.
13997                  * Clean that up here.  It's unclear whether we should do
13998                  * this even if the filesystem has been forcibly unmounted.
13999                  * For most servers, it's probably wasted effort, but
14000                  * RFC3530 lets servers require that unlocks exactly match
14001                  * the locks that are held.
14002                  */
14003                 if (resend_rqstp != NULL &&
14004                     resend_rqstp->lr_ctype != NFS4_LCK_CTYPE_REINSTATE) {
14005                         nfs4_reinstitute_local_lock_state(vp, flk, cr, lop);
14006                 } else {
14007                         flk->l_whence = 0;
14008                 }
14009         } else if (locku_args) {
```

```
14010                 LOCKU4res *locku_res;

14012                 locku_res = &resop->nfs_resop4_u.oplocku;

14014                 /* Update the stateid with the server's response */
14015                 nfs4_set_lock_stateid(lop, locku_res->lock_stateid);
14016         } else if (lockt_args) {
14017                 /* Switch the lock type to express success, see fcntl */
14018                 flk->l_type = F_UNLCK;
14019                 flk->l_whence = 0;
14020         }
14021 }

14023 /*
14024  * Do final cleanup before exiting nfs4frlock.
14025  * Calls nfs4_end_fop, drops the seqid syncs, and frees up the
14026  * COMPOUND4 args/res for calls that haven't already.
14027  */
14028 static void
14029 nfs4frlock_final_cleanup(nfs4_lock_call_type_t ctype, COMPOUND4args_clnt *argsp,
14030     COMPOUND4res_clnt *resp, vnode_t *vp, nfs4_op_hint_t op_hint,
14031     nfs4_recov_state_t *recov_statep, int needrecov, nfs4_open_owner_t *oop,
14032     nfs4_open_stream_t *osp, nfs4_lock_owner_t *lop, flock64_t *flk,
14033     short whence, u_offset_t offset, struct lm_sysid *ls,
14034     int *errorp, LOCK4args *lock_args, LOCKU4args *locku_args,
14035     bool_t did_start_fop, bool_t skip_get_err,
14036     cred_t *cred_otw, cred_t *cred)
14037 {
14038         mntinfo4_t      *mi = VTOMI4(vp);
14039         rnode4_t        *rp = VTOR4(vp);
14040         int             error = *errorp;
14041         nfs_argop4      *argop;
14042         int     do_flush_pages = 0;

14044         ASSERT(nfs_zone() == mi->mi_zone);
14045         /*
14046          * The client recovery code wants the raw status information,
14047          * so don't map the NFS status code to an errno value for
14048          * non-normal call types.
14049          */
14050         if (ctype == NFS4_LCK_CTYPE_NORM) {
14051                 if (*errorp == 0 && resp != NULL && skip_get_err == FALSE)
14052                         *errorp = geterrno4(resp->status);
14053                 if (did_start_fop == TRUE)
14054                         nfs4_end_fop(mi, vp, NULL, op_hint, recov_statep,
14055                             needrecov);

14057                 /*
14058                  * We've established a new lock on the server, so invalidate
14059                  * the pages associated with the vnode to get the most up to
14060                  * date pages from the server after acquiring the lock. We
14061                  * want to be sure that the read operation gets the newest data.
14062                  * N.B.
14063                  * We used to do this in nfs4frlock_results_ok but that doesn't
14064                  * work since VOP_PUTPAGE can call nfs4_commit which calls
14065                  * nfs4_start_fop. We flush the pages below after calling
14066                  * nfs4_end_fop above
14067                  * The flush of the page cache must be done after
14068                  * nfs4_end_open_seqid_sync() to avoid a 4-way hang.
14069                  */
14070                 if (!error && resp && resp->status == NFS4_OK)
14071                         do_flush_pages = 1;
14072         }
14073         if (argsp) {
14074                 ASSERT(argsp->array_len == 2);
14075                 argop = argsp->array;
```



```
14076                 if (argop[1].argop == OP_LOCK)
14077                         nfs4args_lock_free(&argop[1]);
14078                 else if (argop[1].argop == OP_LOCKT)
14079                         nfs4args_lockt_free(&argop[1]);
14080                 kmem_free(argop, 2 * sizeof (nfs_argop4));
14081                 if (resp)
14082                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
14083         }

14085         /* free the reference on the lock owner */
14086         if (lop != NULL) {
14087                 nfs4_end_lock_seqid_sync(lop);
14088                 lock_owner_rele(lop);
14089         }

14091         /* need to free up the reference on osp for lock args */
14092         if (osp != NULL)
14093                 open_stream_rele(osp, rp);

14095         /* need to free up the reference on oop for lock args */
14096         if (oop != NULL) {
14097                 nfs4_end_open_seqid_sync(oop);
14098                 open_owner_rele(oop);
14099         }

14101         if (do_flush_pages)
14102                 nfs4_flush_pages(vp, cred);

14104         (void) convoff(vp, flk, whence, offset);

14106         lm_rel_sysid(ls);

14108         /*
14109          * Record debug information in the event we get EINVAL.
14110          */
14111         mutex_enter(&mi->mi_lock);
14112         if (*errorp == EINVAL && (lock_args || locku_args) &&
14113             (!(mi->mi_flags & MI4_POSIX_LOCK))) {
14114                 if (!(mi->mi_flags & MI4_LOCK_DEBUG)) {
14115                         zcmn_err(getzoneid(), CE_NOTE,
14116                             "%s operation failed with "
14117                             "EINVAL probably since the server, %s,"
14118                             " doesn't support POSIX style locking",
14119                             lock_args ? "LOCK" : "LOCKU",
14120                             mi->mi_curr_serv->sv_hostname);
14121                         mi->mi_flags |= MI4_LOCK_DEBUG;
14122                 }
14123         }
14124         mutex_exit(&mi->mi_lock);

14126         if (cred_otw)
14127                 crfree(cred_otw);
14128 }

14130 /*
14131  * This calls the server and the local locking code.
14132  *
14133  * Client locks are registerred locally by oring the sysid with
14134  * LM_SYSID_CLIENT. The server registers locks locally using just the sysid.
14135  * We need to distinguish between the two to avoid collision in case one
14136  * machine is used as both client and server.
14137  *
14138  * Blocking lock requests will continually retry to acquire the lock
14139  * forever.
14140  *
14141  * The ctype is defined as follows:
```

```
14142  * NFS4_LCK_CTYPE_NORM: normal lock request.
14143  *
14144  * NFS4_LCK_CTYPE_RECLAIM:  bypass the usual calls for synchronizing with client
14145  * recovery, get the pid from flk instead of curproc, and don't reregister
14146  * the lock locally.
14147  *
14148  * NFS4_LCK_CTYPE_RESEND: same as NFS4_LCK_CTYPE_RECLAIM, with the addition
14149  * that we will use the information passed in via resend_rqstp to setup the
14150  * lock/locku request.  This resend is the exact same request as the 'lost
14151  * lock', and is initiated by the recovery framework. A successful resend
14152  * request can initiate one or more reinstate requests.
14153  *
14154  * NFS4_LCK_CTYPE_REINSTATE: same as NFS4_LCK_CTYPE_RESEND, except that it
14155  * does not trigger additional reinstate requests.  This lock call type is
14156  * set for setting the v4 server's locking state back to match what the
14157  * client's local locking state is in the event of a received 'lost lock'.
14158  *
14159  * Errors are returned via the nfs4_error_t parameter.
14160  */
14161 void
14162 nfs4frlock(nfs4_lock_call_type_t ctype, vnode_t *vp, int cmd, flock64_t *flk,
14163     int flag, u_offset_t offset, cred_t *cr, nfs4_error_t *ep,
14164     nfs4_lost_rqst_t *resend_rqstp, int *did_reclaimp)
14165 {
14166         COMPOUND4args_clnt      args, *argsp = NULL;
14167         COMPOUND4res_clnt       res, *resp = NULL;
14168         nfs_argop4      *argop;
14169         nfs_resop4      *resop;
14170         rnode4_t        *rp;
14171         int             doqueue = 1;
14172         clock_t         tick_delay;  /* delay in clock ticks */
14173         struct lm_sysid *ls;
14174         LOCK4args       *lock_args = NULL;
14175         LOCKU4args      *locku_args = NULL;
14176         LOCKT4args      *lockt_args = NULL;
14177         nfs4_open_owner_t *oop = NULL;
14178         nfs4_open_stream_t *osp = NULL;
14179         nfs4_lock_owner_t *lop = NULL;
14180         bool_t          needrecov = FALSE;
14181         nfs4_recov_state_t recov_state;
14182         short           whence;
14183         nfs4_op_hint_t  op_hint;
14184         nfs4_lost_rqst_t lost_rqst;
14185         bool_t          retry = FALSE;
14186         bool_t          did_start_fop = FALSE;
14187         bool_t          skip_get_err = FALSE;
14188         cred_t          *cred_otw = NULL;
14189         bool_t          recovonly;      /* just queue request */
14190         int             frc_no_reclaim = 0;
14191 #ifdef DEBUG
14192         char *name;
14193 #endif

14195         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

14197 #ifdef DEBUG
14198         name = fn_name(VTOSV(vp)->sv_name);
14199         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4frlock: "
14200             "%s: cmd %d, type %d, offset %llu, start %"PRIx64", "
14201             "length %"PRIu64", pid %d, sysid %d, call type %s, "
14202             "resend request %s", name, cmd, flk->l_type, offset, flk->l_start,
14203             flk->l_len, ctype == NFS4_LCK_CTYPE_NORM ? curproc->p_pid :
14204             flk->l_pid, flk->l_sysid, nfs4frlock_get_call_type(ctype),
14205             resend_rqstp ? "TRUE" : "FALSE"));
14206         kmem_free(name, MAXNAMELEN);
14207 #endif
```



```
14209         nfs4_error_zinit(ep);
14210         ep->error = nfs4frlock_validate_args(cmd, flk, flag, vp, offset);
14211         if (ep->error)
14212                 return;
14213         ep->error = nfs4frlock_get_sysid(&ls, vp, flk);
14214         if (ep->error)
14215                 return;
14216         nfs4frlock_pre_setup(&tick_delay, &recov_state, flk, &whence,
14217             vp, cr, &cred_otw);

14219 recov_retry:
14220         nfs4frlock_call_init(&args, &argsp, &argop, &op_hint, flk, cmd,
14221             &retry, &did_start_fop, &resp, &skip_get_err, &lost_rqst);
14222         rp = VTOR4(vp);

14224         ep->error = nfs4frlock_start_call(ctype, vp, op_hint, &recov_state,
14225             &did_start_fop, &recovonly);

14227         if (ep->error)
14228                 goto out;

14230         if (recovonly) {
14231                 /*
14232                  * Leave the request for the recovery system to deal with.
14233                  */
14234                 ASSERT(ctype == NFS4_LCK_CTYPE_NORM);
14235                 ASSERT(cmd != F_GETLK);
14236                 ASSERT(flk->l_type == F_UNLCK);

14238                 nfs4_error_init(ep, EINTR);
14239                 needrecov = TRUE;
14240                 lop = find_lock_owner(rp, curproc->p_pid, LOWN_ANY);
14241                 if (lop != NULL) {
14242                         nfs4frlock_save_lost_rqst(ctype, ep->error, READ_LT,
14243                             NULL, NULL, lop, flk, &lost_rqst, cr, vp);
14244                         (void) nfs4_start_recovery(ep,
14245                             VTOMI4(vp), vp, NULL, NULL,
14246                             (lost_rqst.lr_op == OP_LOCK ||
14247                             lost_rqst.lr_op == OP_LOCKU) ?
14248                             &lost_rqst : NULL, OP_LOCKU, NULL, NULL, NULL);
14249                         lock_owner_rele(lop);
14250                         lop = NULL;
14251                 }
14252                 flk->l_pid = curproc->p_pid;
14253                 nfs4_register_lock_locally(vp, flk, flag, offset);
14254                 goto out;
14255         }

14257         /* putfh directory fh */
14258         argop[0].argop = OP_CPUTFH;
14259         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

14261         /*
14262          * Set up the over-the-wire arguments and get references to the
14263          * open owner, etc.
14264          */

14266         if (ctype == NFS4_LCK_CTYPE_RESEND ||
14267             ctype == NFS4_LCK_CTYPE_REINSTATE) {
14268                 nfs4frlock_setup_resend_lock_args(resend_rqstp, argsp,
14269                     &argop[1], &lop, &oop, &osp, &lock_args, &locku_args);
14270         } else {
14271                 bool_t go_otw = TRUE;

14273                 ASSERT(resend_rqstp == NULL);
```

```
14275                 switch (cmd) {
14276                 case F_GETLK:
14277                 case F_O_GETLK:
14278                         nfs4frlock_setup_lockt_args(ctype, &argop[1],
14279                             &lockt_args, argsp, flk, rp);
14280                         break;
14281                 case F_SETLKW:
14282                 case F_SETLK:
14283                         if (flk->l_type == F_UNLCK)
14284                                 nfs4frlock_setup_locku_args(ctype,
14285                                     &argop[1], &locku_args, flk,
14286                                     &lop, ep, argsp,
14287                                     vp, flag, offset, cr,
14288                                     &skip_get_err, &go_otw);
14289                         else
14290                                 nfs4frlock_setup_lock_args(ctype,
14291                                     &lock_args, &oop, &osp, &lop, &argop[1],
14292                                     argsp, flk, cmd, vp, cr, ep);

14294                         if (ep->error)
14295                                 goto out;

14297                         switch (ep->stat) {
14298                         case NFS4_OK:
14299                                 break;
14300                         case NFS4ERR_DELAY:
14301                                 /* recov thread never gets this error */
14302                                 ASSERT(resend_rqstp == NULL);
14303                                 ASSERT(did_start_fop);

14305                                 nfs4_end_fop(VTOMI4(vp), vp, NULL, op_hint,
14306                                     &recov_state, TRUE);
14307                                 did_start_fop = FALSE;
14308                                 if (argop[1].argop == OP_LOCK)
14309                                         nfs4args_lock_free(&argop[1]);
14310                                 else if (argop[1].argop == OP_LOCKT)
14311                                         nfs4args_lockt_free(&argop[1]);
14312                                 kmem_free(argop, 2 * sizeof (nfs_argop4));
14313                                 argsp = NULL;
14314                                 goto recov_retry;
14315                         default:
14316                                 ep->error = EIO;
14317                                 goto out;
14318                         }
14319                         break;
14320                 default:
14321                         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14322                             "nfs4_frlock: invalid cmd %d", cmd));
14323                         ep->error = EINVAL;
14324                         goto out;
14325                 }

14327                 if (!go_otw)
14328                         goto out;
14329         }

14331         /* XXX should we use the local reclock as a cache ? */
14332         /*
14333          * Unregister the lock with the local locking code before
14334          * contacting the server.  This avoids a potential race where
14335          * another process gets notified that it has been granted a lock
14336          * before we can unregister ourselves locally.
14337          */
14338         if ((cmd == F_SETLK || cmd == F_SETLKW) && flk->l_type == F_UNLCK) {
14339                 if (ctype == NFS4_LCK_CTYPE_NORM)
```



```
14340                         flk->l_pid = ttoproc(curthread)->p_pid;
14341                 nfs4_register_lock_locally(vp, flk, flag, offset);
14342         }

14344         /*
14345          * Send the server the lock request.  Continually loop with a delay
14346          * if get error NFS4ERR_DENIED (for blocking locks) or NFS4ERR_GRACE.
14347          */
14348         resp = &res;

14350         NFS4_DEBUG((nfs4_client_call_debug || nfs4_client_lock_debug),
14351             (CE_NOTE,
14352             "nfs4frlock: %s call, rp %s", needrecov ? "recov" : "first",
14353             rnode4info(rp)));

14355         if (lock_args && frc_no_reclaim) {
14356                 ASSERT(ctype == NFS4_LCK_CTYPE_RECLAIM);
14357                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14358                     "nfs4frlock: frc_no_reclaim: clearing reclaim"));
14359                 lock_args->reclaim = FALSE;
14360                 if (did_reclaimp)
14361                         *did_reclaimp = 0;
14362         }

14364         /*
14365          * Do the OTW call.
14366          */
14367         rfs4call(VTOMI4(vp), argsp, resp, cred_otw, &doqueue, 0, ep);

14369         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14370             "nfs4frlock: error %d, status %d", ep->error, resp->status));

14372         needrecov = nfs4_needs_recovery(ep, TRUE, vp->v_vfsp);
14373         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14374             "nfs4frlock: needrecov %d", needrecov));

14376         if (ep->error == 0 && nfs4_need_to_bump_seqid(resp))
14377                 nfs4frlock_bump_seqid(lock_args, locku_args, oop, lop,
14378                     args.ctag);

14380         /*
14381          * Check if one of these mutually exclusive error cases has
14382          * happened:
14383          *   need to swap credentials due to access error
14384          *   recovery is needed
14385          *   different error (only known case is missing Kerberos ticket)
14386          */

14388         if ((ep->error == EACCES ||
14389             (ep->error == 0 && resp->status == NFS4ERR_ACCESS)) &&
14390             cred_otw != cr) {
14391                 nfs4frlock_check_access(vp, op_hint, &recov_state, needrecov,
14392                     &did_start_fop, &argsp, &resp, ep->error, &lop, &oop, &osp,
14393                     cr, &cred_otw);
14394                 goto recov_retry;
14395         }

14397         if (needrecov) {
14398                 /*
14399                  * LOCKT requests don't need to recover from lost
14400                  * requests since they don't create/modify state.
14401                  */
14402                 if ((ep->error == EINTR ||
14403                     NFS4_FRC_UNMT_ERR(ep->error, vp->v_vfsp)) &&
14404                     lockt_args)
14405                         goto out;
```

```
14406                 /*
14407                  * Do not attempt recovery for requests initiated by
14408                  * the recovery framework.  Let the framework redrive them.
14409                  */
14410                 if (ctype != NFS4_LCK_CTYPE_NORM)
14411                         goto out;
14412                 else {
14413                         ASSERT(resend_rqstp == NULL);
14414                 }

14416                 nfs4frlock_save_lost_rqst(ctype, ep->error,
14417                     flk_to_locktype(cmd, flk->l_type),
14418                     oop, osp, lop, flk, &lost_rqst, cred_otw, vp);

14420                 retry = nfs4frlock_recovery(needrecov, ep, &argsp,
14421                     &resp, lock_args, locku_args, &oop, &osp, &lop,
14422                     rp, vp, &recov_state, op_hint, &did_start_fop,
14423                     cmd != F_GETLK ? &lost_rqst : NULL, flk);

14425                 if (retry) {
14426                         ASSERT(oop == NULL);
14427                         ASSERT(osp == NULL);
14428                         ASSERT(lop == NULL);
14429                         goto recov_retry;
14430                 }
14431                 goto out;
14432         }

14434         /*
14435          * Bail out if have reached this point with ep->error set. Can
14436          * happen if (ep->error == EACCES && !needrecov && cred_otw == cr).
14437          * This happens if Kerberos ticket has expired or has been
14438          * destroyed.
14439          */
14440         if (ep->error != 0)
14441                 goto out;

14443         /*
14444          * Process the reply.
14445          */
14446         switch (resp->status) {
14447         case NFS4_OK:
14448                 resop = &resp->array[1];
14449                 nfs4frlock_results_ok(ctype, cmd, flk, vp, flag, offset,
14450                     resend_rqstp);
14451                 /*
14452                  * Have a successful lock operation, now update state.
14453                  */
14454                 nfs4frlock_update_state(lock_args, locku_args, lockt_args,
14455                     resop, lop, vp, flk, cr, resend_rqstp);
14456                 break;

14458         case NFS4ERR_DENIED:
14459                 resop = &resp->array[1];
14460                 retry = nfs4frlock_results_denied(ctype, lock_args, lockt_args,
14461                     &oop, &osp, &lop, cmd, vp, flk, op_hint,
14462                     &recov_state, needrecov, &argsp, &resp,
14463                     &tick_delay, &whence, &ep->error, resop, cr,
14464                     &did_start_fop, &skip_get_err);

14466                 if (retry) {
14467                         ASSERT(oop == NULL);
14468                         ASSERT(osp == NULL);
14469                         ASSERT(lop == NULL);
14470                         goto recov_retry;
14471                 }
```



```
14472                 break;
14473         /*
14474          * If the server won't let us reclaim, fall-back to trying to lock
14475          * the file from scratch. Code elsewhere will check the changeinfo
14476          * to ensure the file hasn't been changed.
14477          */
14478         case NFS4ERR_NO_GRACE:
14479                 if (lock_args && lock_args->reclaim == TRUE) {
14480                         ASSERT(ctype == NFS4_LCK_CTYPE_RECLAIM);
14481                         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14482                             "nfs4frlock: reclaim: NFS4ERR_NO_GRACE"));
14483                         frc_no_reclaim = 1;
14484                         /* clean up before retrying */
14485                         needrecov = 0;
14486                         (void) nfs4frlock_recovery(needrecov, ep, &argsp, &resp,
14487                             lock_args, locku_args, &oop, &osp, &lop, rp, vp,
14488                             &recov_state, op_hint, &did_start_fop, NULL, flk);
14489                         goto recov_retry;
14490                 }
14491                 /* FALLTHROUGH */

14493         default:
14494                 nfs4frlock_results_default(resp, &ep->error);
14495                 break;
14496         }
14497 out:
14498         /*
14499          * Process and cleanup from error.  Make interrupted unlock
14500          * requests look successful, since they will be handled by the
14501          * client recovery code.
14502          */
14503         nfs4frlock_final_cleanup(ctype, argsp, resp, vp, op_hint, &recov_state,
14504             needrecov, oop, osp, lop, flk, whence, offset, ls, &ep->error,
14505             lock_args, locku_args, did_start_fop,
14506             skip_get_err, cred_otw, cr);

14508         if (ep->error == EINTR && flk->l_type == F_UNLCK &&
14509             (cmd == F_SETLK || cmd == F_SETLKW))
14510                 ep->error = 0;
14511 }

14513 /*
14514  * nfs4_safelock:
14515  *
14516  * Return non-zero if the given lock request can be handled without
14517  * violating the constraints on concurrent mapping and locking.
14518  */

14520 static int
14521 nfs4_safelock(vnode_t *vp, const struct flock64 *bfp, cred_t *cr)
14522 {
14523         rnode4_t *rp = VTOR4(vp);
14524         struct vattr va;
14525         int error;

14527         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
14528         ASSERT(rp->r_mapcnt >= 0);
14529         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4_safelock %s: "
14530             "(%"PRIx64", %"PRIx64"); mapcnt = %ld", bfp->l_type == F_WRLCK ?
14531             "write" : bfp->l_type == F_RDLCK ? "read" : "unlock",
14532             bfp->l_start, bfp->l_len, rp->r_mapcnt));

14534         if (rp->r_mapcnt == 0)
14535                 return (1);             /* always safe if not mapped */

14537         /*
```

```
14538          * If the file is already mapped and there are locks, then they
14539          * should be all safe locks.  So adding or removing a lock is safe
14540          * as long as the new request is safe (i.e., whole-file, meaning
14541          * length and starting offset are both zero).
14542          */

14544         if (bfp->l_start != 0 || bfp->l_len != 0) {
14545                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4_safelock: "
14546                     "cannot lock a memory mapped file unless locking the "
14547                     "entire file: start %"PRIx64", len %"PRIx64,
14548                     bfp->l_start, bfp->l_len));
14549                 return (0);
14550         }

14552         /* mandatory locking and mapping don't mix */
14553         va.va_mask = AT_MODE;
14554         error = VOP_GETATTR(vp, &va, 0, cr, NULL);
14555         if (error != 0) {
14556                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4_safelock: "
14557                     "getattr error %d", error));
14558                 return (0);             /* treat errors conservatively */
14559         }
14560         if (MANDLOCK(vp, va.va_mode)) {
14561                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4_safelock: "
14562                     "cannot mandatory lock and mmap a file"));
14563                 return (0);
14564         }

14566         return (1);
14567 }


14570 /*
14571  * Register the lock locally within Solaris.
14572  * As the client, we "or" the sysid with LM_SYSID_CLIENT when
14573  * recording locks locally.
14574  *
14575  * This should handle conflicts/cooperation with NFS v2/v3 since all locks
14576  * are registered locally.
14577  */
14578 void
14579 nfs4_register_lock_locally(vnode_t *vp, struct flock64 *flk, int flag,
14580     u_offset_t offset)
14581 {
14582         int oldsysid;
14583         int error;
14584 #ifdef DEBUG
14585         char *name;
14586 #endif

14588         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

14590 #ifdef DEBUG
14591         name = fn_name(VTOSV(vp)->sv_name);
14592         NFS4_DEBUG(nfs4_client_lock_debug,
14593             (CE_NOTE, "nfs4_register_lock_locally: %s: type %d, "
14594             "start %"PRIx64", length %"PRIx64", pid %ld, sysid %d",
14595             name, flk->l_type, flk->l_start, flk->l_len, (long)flk->l_pid,
14596             flk->l_sysid));
14597         kmem_free(name, MAXNAMELEN);
14598 #endif

14600         /* register the lock with local locking */
14601         oldsysid = flk->l_sysid;
14602         flk->l_sysid |= LM_SYSID_CLIENT;
14603         error = reclock(vp, flk, SETFLCK, flag, offset, NULL);
```



```
14604 #ifdef DEBUG
14605         if (error != 0) {
14606                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14607                     "nfs4_register_lock_locally: could not register with"
14608                     " local locking"));
14609                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_CONT,
14610                     "error %d, vp 0x%p, pid %d, sysid 0x%x",
14611                     error, (void *)vp, flk->l_pid, flk->l_sysid));
14612                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_CONT,
14613                     "type %d off 0x%" PRIx64 " len 0x%" PRIx64,
14614                     flk->l_type, flk->l_start, flk->l_len));
14615                 (void) reclock(vp, flk, 0, flag, offset, NULL);
14616                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_CONT,
14617                     "blocked by pid %d sysid 0x%x type %d "
14618                     "off 0x%" PRIx64 " len 0x%" PRIx64,
14619                     flk->l_pid, flk->l_sysid, flk->l_type, flk->l_start,
14620                     flk->l_len));
14621         }
14622 #endif
14623         flk->l_sysid = oldsysid;
14624 }

14626 /*
14627  * nfs4_lockrelease:
14628  *
14629  * Release any locks on the given vnode that are held by the current
14630  * process.  Also removes the lock owner (if one exists) from the rnode's
14631  * list.
14632  */
14633 static int
14634 nfs4_lockrelease(vnode_t *vp, int flag, offset_t offset, cred_t *cr)
14635 {
14636         flock64_t ld;
14637         int ret, error;
14638         rnode4_t *rp;
14639         nfs4_lock_owner_t *lop;
14640         nfs4_recov_state_t recov_state;
14641         mntinfo4_t *mi;
14642         bool_t possible_orphan = FALSE;
14643         bool_t recovonly;

14645         ASSERT((uintptr_t)vp > KERNELBASE);
14646         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

14648         rp = VTOR4(vp);
14649         mi = VTOMI4(vp);

14651         /*
14652          * If we have not locked anything then we can
14653          * just return since we have no work to do.
14654          */
14655         if (rp->r_lo_head.lo_next_rnode == &rp->r_lo_head) {
14656                 return (0);
14657         }

14659         /*
14660          * We need to comprehend that another thread may
14661          * kick off recovery and the lock_owner we have stashed
14662          * in lop might be invalid so we should NOT cache it
14663          * locally!
14664          */
14665         recov_state.rs_flags = 0;
14666         recov_state.rs_num_retry_despite_err = 0;
14667         error = nfs4_start_fop(mi, vp, NULL, OH_LOCKU, &recov_state,
14668             &recovonly);
14669         if (error) {
```

```
14670                 mutex_enter(&rp->r_statelock);
14671                 rp->r_flags |= R4LODANGLERS;
14672                 mutex_exit(&rp->r_statelock);
14673                 return (error);
14674         }

14676         lop = find_lock_owner(rp, curproc->p_pid, LOWN_ANY);

14678         /*
14679          * Check if the lock owner might have a lock (request was sent but
14680          * no response was received).  Also check if there are any remote
14681          * locks on the file.  (In theory we shouldn't have to make this
14682          * second check if there's no lock owner, but for now we'll be
14683          * conservative and do it anyway.)  If either condition is true,
14684          * send an unlock for the entire file to the server.
14685          *
14686          * Note that no explicit synchronization is needed here.  At worst,
14687          * flk_has_remote_locks() will return a false positive, in which case
14688          * the unlock call wastes time but doesn't harm correctness.
14689          */

14691         if (lop) {
14692                 mutex_enter(&lop->lo_lock);
14693                 possible_orphan = lop->lo_pending_rqsts;
14694                 mutex_exit(&lop->lo_lock);
14695                 lock_owner_rele(lop);
14696         }

14698         nfs4_end_fop(mi, vp, NULL, OH_LOCKU, &recov_state, 0);

14700         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14701             "nfs4_lockrelease: possible orphan %d, remote locks %d, for "
14702             "lop %p.", possible_orphan, flk_has_remote_locks(vp),
14703             (void *)lop));

14705         if (possible_orphan || flk_has_remote_locks(vp)) {
14706                 ld.l_type = F_UNLCK;    /* set to unlock entire file */
14707                 ld.l_whence = 0;        /* unlock from start of file */
14708                 ld.l_start = 0;
14709                 ld.l_len = 0;           /* do entire file */

14711                 ret = VOP_FRLOCK(vp, F_SETLK, &ld, flag, offset, NULL,
14712                     cr, NULL);

14714                 if (ret != 0) {
14715                         /*
14716                          * If VOP_FRLOCK fails, make sure we unregister
14717                          * local locks before we continue.
14718                          */
14719                         ld.l_pid = ttoproc(curthread)->p_pid;
14720                         nfs4_register_lock_locally(vp, &ld, flag, offset);
14721                         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
14722                             "nfs4_lockrelease: lock release error on vp"
14723                             " %p: error %d.\n", (void *)vp, ret));
14724                 }
14725         }

14727         recov_state.rs_flags = 0;
14728         recov_state.rs_num_retry_despite_err = 0;
14729         error = nfs4_start_fop(mi, vp, NULL, OH_LOCKU, &recov_state,
14730             &recovonly);
14731         if (error) {
14732                 mutex_enter(&rp->r_statelock);
14733                 rp->r_flags |= R4LODANGLERS;
14734                 mutex_exit(&rp->r_statelock);
14735                 return (error);
```



```
14736         }

14738         /*
14739          * So, here we're going to need to retrieve the lock-owner
14740          * again (in case recovery has done a switch-a-roo) and
14741          * remove it because we can.
14742          */
14743         lop = find_lock_owner(rp, curproc->p_pid, LOWN_ANY);

14745         if (lop) {
14746                 nfs4_rnode_remove_lock_owner(rp, lop);
14747                 lock_owner_rele(lop);
14748         }

14750         nfs4_end_fop(mi, vp, NULL, OH_LOCKU, &recov_state, 0);
14751         return (0);
14752 }

14754 /*
14755  * Wait for 'tick_delay' clock ticks.
14756  * Implement exponential backoff until hit the lease_time of this nfs4_server.
14757  * NOTE: lock_lease_time is in seconds.
14758  *
14759  * XXX For future improvements, should implement a waiting queue scheme.
14760  */
14761 static int
14762 nfs4_block_and_wait(clock_t *tick_delay, rnode4_t *rp)
14763 {
14764         long milliseconds_delay;
14765         time_t lock_lease_time;

14767         /* wait tick_delay clock ticks or siginteruptus */
14768         if (delay_sig(*tick_delay)) {
14769                 return (EINTR);
14770         }
14771         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4_block_and_wait: "
14772             "reissue the lock request: blocked for %ld clock ticks: %ld "
14773             "milliseconds", *tick_delay, drv_hztousec(*tick_delay) / 1000));

14775         /* get the lease time */
14776         lock_lease_time = r2lease_time(rp);

14778         /* drv_hztousec converts ticks to microseconds */
14779         milliseconds_delay = drv_hztousec(*tick_delay) / 1000;
14780         if (milliseconds_delay < lock_lease_time * 1000) {
14781                 *tick_delay = 2 * *tick_delay;
14782                 if (drv_hztousec(*tick_delay) > lock_lease_time * 1000 * 1000)
14783                         *tick_delay = drv_usectohz(lock_lease_time*1000*1000);
14784         }
14785         return (0);
14786 }


14789 void
14790 nfs4_vnops_init(void)
14791 {
14792 }

14794 void
14795 nfs4_vnops_fini(void)
14796 {
14797 }

14799 /*
14800  * Return a reference to the directory (parent) vnode for a given vnode,
14801  * using the saved pathname information and the directory file handle.  The
```

```
14802  * caller is responsible for disposing of the reference.
14803  * Returns zero or an errno value.
14804  *
14805  * Caller should set need_start_op to FALSE if it is the recovery
14806  * thread, or if a start_fop has already been done.  Otherwise, TRUE.
14807  */
14808 int
14809 vtodv(vnode_t *vp, vnode_t **dvpp, cred_t *cr, bool_t need_start_op)
14810 {
14811         svnode_t *svnp;
14812         vnode_t *dvp = NULL;
14813         servinfo4_t *svp;
14814         nfs4_fname_t *mfname;
14815         int error;

14817         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

14819         if (vp->v_flag & VROOT) {
14820                 nfs4_sharedfh_t *sfh;
14821                 nfs_fh4 fh;
14822                 mntinfo4_t *mi;

14824                 ASSERT(vp->v_type == VREG);

14826                 mi = VTOMI4(vp);
14827                 svp = mi->mi_curr_serv;
14828                 (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
14829                 fh.nfs_fh4_len = svp->sv_pfhandle.fh_len;
14830                 fh.nfs_fh4_val = svp->sv_pfhandle.fh_buf;
14831                 sfh = sfh4_get(&fh, VTOMI4(vp));
14832                 nfs_rw_exit(&svp->sv_lock);
14833                 mfname = mi->mi_fname;
14834                 fn_hold(mfname);
14835                 dvp = makenfs4node_by_fh(sfh, NULL, &mfname, NULL, mi, cr, 0);
14836                 sfh4_rele(&sfh);

14838                 if (dvp->v_type == VNON)
14839                         dvp->v_type = VDIR;
14840                 *dvpp = dvp;
14841                 return (0);
14842         }

14844         svnp = VTOSV(vp);

14846         if (svnp == NULL) {
14847                 NFS4_DEBUG(nfs4_client_shadow_debug, (CE_NOTE, "vtodv: "
14848                     "shadow node is NULL"));
14849                 return (EINVAL);
14850         }

14852         if (svnp->sv_name == NULL || svnp->sv_dfh == NULL) {
14853                 NFS4_DEBUG(nfs4_client_shadow_debug, (CE_NOTE, "vtodv: "
14854                     "shadow node name or dfh val == NULL"));
14855                 return (EINVAL);
14856         }

14858         error = nfs4_make_dotdot(svnp->sv_dfh, 0, vp, cr, &dvp,
14859             (int)need_start_op);
14860         if (error != 0) {
14861                 NFS4_DEBUG(nfs4_client_shadow_debug, (CE_NOTE, "vtodv: "
14862                     "nfs4_make_dotdot returned %d", error));
14863                 return (error);
14864         }
14865         if (!dvp) {
14866                 NFS4_DEBUG(nfs4_client_shadow_debug, (CE_NOTE, "vtodv: "
14867                     "nfs4_make_dotdot returned a NULL dvp"));
```



```
14868                 return (EIO);
14869         }
14870         if (dvp->v_type == VNON)
14871                 dvp->v_type = VDIR;
14872         ASSERT(dvp->v_type == VDIR);
14873         if (VTOR4(vp)->r_flags & R4ISXATTR) {
14874                 mutex_enter(&dvp->v_lock);
14875                 dvp->v_flag |= V_XATTRDIR;
14876                 mutex_exit(&dvp->v_lock);
14877         }
14878         *dvpp = dvp;
14879         return (0);
14880 }

14882 /*
14883  * Copy the (final) component name of vp to fnamep.  maxlen is the maximum
14884  * length that fnamep can accept, including the trailing null.
14885  * Returns 0 if okay, returns an errno value if there was a problem.
14886  */

14888 int
14889 vtoname(vnode_t *vp, char *fnamep, ssize_t maxlen)
14890 {
14891         char *fn;
14892         int err = 0;
14893         servinfo4_t *svp;
14894         svnode_t *shvp;

14896         /*
14897          * If the file being opened has VROOT set, then this is
14898          * a "file" mount.  sv_name will not be interesting, so
14899          * go back to the servinfo4 to get the original mount
14900          * path and strip off all but the final edge.  Otherwise
14901          * just return the name from the shadow vnode.
14902          */

14904         if (vp->v_flag & VROOT) {

14906                 svp = VTOMI4(vp)->mi_curr_serv;
14907                 (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);

14909                 fn = strrchr(svp->sv_path, '/');
14910                 if (fn == NULL)
14911                         err = EINVAL;
14912                 else
14913                         fn++;
14914         } else {
14915                 shvp = VTOSV(vp);
14916                 fn = fn_name(shvp->sv_name);
14917         }

14919         if (err == 0)
14920                 if (strlen(fn) < maxlen)
14921                         (void) strcpy(fnamep, fn);
14922                 else
14923                         err = ENAMETOOLONG;

14925         if (vp->v_flag & VROOT)
14926                 nfs_rw_exit(&svp->sv_lock);
14927         else
14928                 kmem_free(fn, MAXNAMELEN);

14930         return (err);
14931 }

14933 /*
```

```
14934  * Bookkeeping for a close that doesn't need to go over the wire.
14935  * *have_lockp is set to 0 if 'os_sync_lock' is released; otherwise
14936  * it is left at 1.
14937  */
14938 void
14939 nfs4close_notw(vnode_t *vp, nfs4_open_stream_t *osp, int *have_lockp)
14940 {
14941         rnode4_t                *rp;
14942         mntinfo4_t              *mi;

14944         mi = VTOMI4(vp);
14945         rp = VTOR4(vp);

14947         NFS4_DEBUG(nfs4close_notw_debug, (CE_NOTE, "nfs4close_notw: "
14948             "rp=%p osp=%p", (void *)rp, (void *)osp));
14949         ASSERT(nfs_zone() == mi->mi_zone);
14950         ASSERT(mutex_owned(&osp->os_sync_lock));
14951         ASSERT(*have_lockp);

14953         if (!osp->os_valid ||
14954             osp->os_open_ref_count > 0 || osp->os_mapcnt > 0) {
14955                 return;
14956         }

14958         /*
14959          * This removes the reference obtained at OPEN; ie,
14960          * when the open stream structure was created.
14961          *
14962          * We don't have to worry about calling 'open_stream_rele'
14963          * since we our currently holding a reference to this
14964          * open stream which means the count can not go to 0 with
14965          * this decrement.
14966          */
14967         ASSERT(osp->os_ref_count >= 2);
14968         osp->os_ref_count--;
14969         osp->os_valid = 0;
14970         mutex_exit(&osp->os_sync_lock);
14971         *have_lockp = 0;

14973         nfs4_dec_state_ref_count(mi);
14974 }

14976 /*
14977  * Close all remaining open streams on the rnode.  These open streams
14978  * could be here because:
14979  * - The close attempted at either close or delmap failed
14980  * - Some kernel entity did VOP_OPEN but never did VOP_CLOSE
14981  * - Someone did mknod on a regular file but never opened it
14982  */
14983 int
14984 nfs4close_all(vnode_t *vp, cred_t *cr)
14985 {
14986         nfs4_open_stream_t *osp;
14987         int error;
14988         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
14989         rnode4_t *rp;

14991         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

14993         error = 0;
14994         rp = VTOR4(vp);

14996         /*
14997          * At this point, all we know is that the last time
14998          * someone called vn_rele, the count was 1.  Since then,
14999          * the vnode could have been re-activated.  We want to
```



```
15000          * loop through the open streams and close each one, but
15001          * we have to be careful since once we release the rnode
15002          * hash bucket lock, someone else is free to come in and
15003          * re-activate the rnode and add new open streams.  The
15004          * strategy is take the rnode hash bucket lock, verify that
15005          * the count is still 1, grab the open stream off the
15006          * head of the list and mark it invalid, then release the
15007          * rnode hash bucket lock and proceed with that open stream.
15008          * This is ok because nfs4close_one() will acquire the proper
15009          * open/create to close/destroy synchronization for open
15010          * streams, and will ensure that if someone has reopened
15011          * the open stream after we've dropped the hash bucket lock
15012          * then we'll just simply return without destroying the
15013          * open stream.
15014          * Repeat until the list is empty.
15015          */

15017         for (;;) {

15019                 /* make sure vnode hasn't been reactivated */
15020                 rw_enter(&rp->r_hashq->r_lock, RW_READER);
15021                 mutex_enter(&vp->v_lock);
15022                 if (vp->v_count > 1) {
15023                         mutex_exit(&vp->v_lock);
15024                         rw_exit(&rp->r_hashq->r_lock);
15025                         break;
15026                 }
15027                 /*
15028                  * Grabbing r_os_lock before releasing v_lock prevents
15029                  * a window where the rnode/open stream could get
15030                  * reactivated (and os_force_close set to 0) before we
15031                  * had a chance to set os_force_close to 1.
15032                  */
15033                 mutex_enter(&rp->r_os_lock);
15034                 mutex_exit(&vp->v_lock);

15036                 osp = list_head(&rp->r_open_streams);
15037                 if (!osp) {
15038                         /* nothing left to CLOSE OTW, so return */
15039                         mutex_exit(&rp->r_os_lock);
15040                         rw_exit(&rp->r_hashq->r_lock);
15041                         break;
15042                 }

15044                 mutex_enter(&rp->r_statev4_lock);
15045                 /* the file can't still be mem mapped */
15046                 ASSERT(rp->r_mapcnt == 0);
15047                 if (rp->created_v4)
15048                         rp->created_v4 = 0;
15049                 mutex_exit(&rp->r_statev4_lock);

15051                 /*
15052                  * Grab a ref on this open stream; nfs4close_one
15053                  * will mark it as invalid
15054                  */
15055                 mutex_enter(&osp->os_sync_lock);
15056                 osp->os_ref_count++;
15057                 osp->os_force_close = 1;
15058                 mutex_exit(&osp->os_sync_lock);
15059                 mutex_exit(&rp->r_os_lock);
15060                 rw_exit(&rp->r_hashq->r_lock);

15062                 nfs4close_one(vp, osp, cr, 0, NULL, &e, CLOSE_FORCE, 0, 0, 0);

15064                 /* Update error if it isn't already non-zero */
15065                 if (error == 0) {
```

```
15066                         if (e.error)
15067                                 error = e.error;
15068                         else if (e.stat)
15069                                 error = geterrno4(e.stat);
15070                 }

15072 #ifdef  DEBUG
15073                 nfs4close_all_cnt++;
15074 #endif
15075                 /* Release the ref on osp acquired above. */
15076                 open_stream_rele(osp, rp);

15078                 /* Proceed to the next open stream, if any */
15079         }
15080         return (error);
15081 }

15083 /*
15084  * nfs4close_one - close one open stream for a file if needed.
15085  *
15086  * "close_type" indicates which close path this is:
15087  * CLOSE_NORM: close initiated via VOP_CLOSE.
15088  * CLOSE_DELMAP: close initiated via VOP_DELMAP.
15089  * CLOSE_FORCE: close initiated via VOP_INACTIVE.  This path forces
15090  *      the close and release of client state for this open stream
15091  *      (unless someone else has the open stream open).
15092  * CLOSE_RESEND: indicates the request is a replay of an earlier request
15093  *      (e.g., due to abort because of a signal).
15094  * CLOSE_AFTER_RESEND: close initiated to "undo" a successful resent OPEN.
15095  *
15096  * CLOSE_RESEND and CLOSE_AFTER_RESEND will not attempt to retry after client
15097  * recovery.  Instead, the caller is expected to deal with retries.
15098  *
15099  * The caller can either pass in the osp ('provided_osp') or not.
15100  *
15101  * 'access_bits' represents the access we are closing/downgrading.
15102  *
15103  * 'len', 'prot', and 'mmap_flags' are used for CLOSE_DELMAP.  'len' is the
15104  * number of bytes we are unmapping, 'maxprot' is the mmap protection, and
15105  * 'mmap_flags' tells us the type of sharing (MAP_PRIVATE or MAP_SHARED).
15106  *
15107  * Errors are returned via the nfs4_error_t.
15108  */
15109 void
15110 nfs4close_one(vnode_t *vp, nfs4_open_stream_t *provided_osp, cred_t *cr,
15111     int access_bits, nfs4_lost_rqst_t *lrp, nfs4_error_t *ep,
15112     nfs4_close_type_t close_type, size_t len, uint_t maxprot,
15113     uint_t mmap_flags)
15114 {
15115         nfs4_open_owner_t *oop;
15116         nfs4_open_stream_t *osp = NULL;
15117         int retry = 0;
15118         int num_retries = NFS4_NUM_RECOV_RETRIES;
15119         rnode4_t *rp;
15120         mntinfo4_t *mi;
15121         nfs4_recov_state_t recov_state;
15122         cred_t *cred_otw = NULL;
15123         bool_t recovonly = FALSE;
15124         int isrecov;
15125         int force_close;
15126         int close_failed = 0;
15127         int did_dec_count = 0;
15128         int did_start_op = 0;
15129         int did_force_recovlock = 0;
15130         int did_start_seqid_sync = 0;
15131         int have_sync_lock = 0;
```



```
15133         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

15135         NFS4_DEBUG(nfs4close_one_debug, (CE_NOTE, "closing vp %p osp %p, "
15136             "lrp %p, close type %d len %ld prot %x mmap flags %x bits %x",
15137             (void *)vp, (void *)provided_osp, (void *)lrp, close_type,
15138             len, maxprot, mmap_flags, access_bits));

15140         nfs4_error_zinit(ep);
15141         rp = VTOR4(vp);
15142         mi = VTOMI4(vp);
15143         isrecov = (close_type == CLOSE_RESEND ||
15144             close_type == CLOSE_AFTER_RESEND);

15146         /*
15147          * First get the open owner.
15148          */
15149         if (!provided_osp) {
15150                 oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
15151         } else {
15152                 oop = provided_osp->os_open_owner;
15153                 ASSERT(oop != NULL);
15154                 open_owner_hold(oop);
15155         }

15157         if (!oop) {
15158                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
15159                     "nfs4close_one: no oop, rp %p, mi %p, cr %p, osp %p, "
15160                     "close type %d", (void *)rp, (void *)mi, (void *)cr,
15161                     (void *)provided_osp, close_type));
15162                 ep->error = EIO;
15163                 goto out;
15164         }

15166         cred_otw = nfs4_get_otw_cred(cr, mi, oop);
15167 recov_retry:
15168         osp = NULL;
15169         close_failed = 0;
15170         force_close = (close_type == CLOSE_FORCE);
15171         retry = 0;
15172         did_start_op = 0;
15173         did_force_recovlock = 0;
15174         did_start_seqid_sync = 0;
15175         have_sync_lock = 0;
15176         recovonly = FALSE;
15177         recov_state.rs_flags = 0;
15178         recov_state.rs_num_retry_despite_err = 0;

15180         /*
15181          * Second synchronize with recovery.
15182          */
15183         if (!isrecov) {
15184                 ep->error = nfs4_start_fop(mi, vp, NULL, OH_CLOSE,
15185                     &recov_state, &recovonly);
15186                 if (!ep->error) {
15187                         did_start_op = 1;
15188                 } else {
15189                         close_failed = 1;
15190                         /*
15191                          * If we couldn't get start_fop, but have to
15192                          * cleanup state, then at least acquire the
15193                          * mi_recovlock so we can synchronize with
15194                          * recovery.
15195                          */
15196                         if (close_type == CLOSE_FORCE) {
15197                                 (void) nfs_rw_enter_sig(&mi->mi_recovlock,
```

```
15198                                   RW_READER, FALSE);
15199                           did_force_recovlock = 1;
15200                   } else
15201                           goto out;
15202           }
15203   }

15205   /*
15206    * We cannot attempt to get the open seqid sync if nfs4_start_fop
15207    * set 'recovonly' to TRUE since most likely this is due to
15208    * reovery being active (MI4_RECOV_ACTIV).  If recovery is active,
15209    * nfs4_start_open_seqid_sync() will fail with EAGAIN asking us
15210    * to retry, causing us to loop until recovery finishes.  Plus we
15211    * don't need protection over the open seqid since we're not going
15212    * OTW, hence don't need to use the seqid.
15213    */
15214   if (recovonly == FALSE) {
15215           /* need to grab the open owner sync before 'os_sync_lock' */
15216           ep->error = nfs4_start_open_seqid_sync(oop, mi);
15217           if (ep->error == EAGAIN) {
15218                   ASSERT(!isrecov);
15219                   if (did_start_op)
15220                           nfs4_end_fop(mi, vp, NULL, OH_CLOSE,
15221                               &recov_state, TRUE);
15222                   if (did_force_recovlock)
15223                           nfs_rw_exit(&mi->mi_recovlock);
15224                   goto recov_retry;
15225           }
15226           did_start_seqid_sync = 1;
15227   }

15229   /*
15230    * Third get an open stream and acquire 'os_sync_lock' to
15231    * sychronize the opening/creating of an open stream with the
15232    * closing/destroying of an open stream.
15233    */
15234   if (!provided_osp) {
15235           /* returns with 'os_sync_lock' held */
15236           osp = find_open_stream(oop, rp);
15237           if (!osp) {
15238                   ep->error = EIO;
15239                   goto out;
15240           }
15241   } else {
15242           osp = provided_osp;
15243           open_stream_hold(osp);
15244           mutex_enter(&osp->os_sync_lock);
15245   }
15246   have_sync_lock = 1;

15248   ASSERT(oop == osp->os_open_owner);

15250   /*
15251    * Fourth, do any special pre-OTW CLOSE processing
15252    * based on the specific close type.
15253    */
15254   if ((close_type == CLOSE_NORM || close_type == CLOSE_AFTER_RESEND) &&
15255       !did_dec_count) {
15256           ASSERT(osp->os_open_ref_count > 0);
15257           osp->os_open_ref_count--;
15258           did_dec_count = 1;
15259           if (osp->os_open_ref_count == 0)
15260                   osp->os_final_close = 1;
15261   }

15263   if (close_type == CLOSE_FORCE) {
```



```
15264           /* see if somebody reopened the open stream. */
15265           if (!osp->os_force_close) {
15266                   NFS4_DEBUG(nfs4close_one_debug, (CE_NOTE,
15267                       "nfs4close_one: skip CLOSE_FORCE as osp %p "
15268                       "was reopened, vp %p", (void *)osp, (void *)vp));
15269                   ep->error = 0;
15270                   ep->stat = NFS4_OK;
15271                   goto out;
15272           }

15274           if (!osp->os_final_close && !did_dec_count) {
15275                   osp->os_open_ref_count--;
15276                   did_dec_count = 1;
15277           }

15279           /*
15280            * We can't depend on os_open_ref_count being 0 due to the
15281            * way executables are opened (VN_RELE to match a VOP_OPEN).
15282            */
15283 #ifdef  NOTYET
15284           ASSERT(osp->os_open_ref_count == 0);
15285 #endif
15286           if (osp->os_open_ref_count != 0) {
15287                   NFS4_DEBUG(nfs4close_one_debug, (CE_NOTE,
15288                       "nfs4close_one: should panic here on an "
15289                       "ASSERT(osp->os_open_ref_count == 0). Ignoring "
15290                       "since this is probably the exec problem."));

15292                   osp->os_open_ref_count = 0;
15293           }

15295           /*
15296            * There is the possibility that nfs4close_one()
15297            * for close_type == CLOSE_DELMAP couldn't find the
15298            * open stream, thus couldn't decrement its os_mapcnt;
15299            * therefore we can't use this ASSERT yet.
15300            */
15301 #ifdef  NOTYET
15302           ASSERT(osp->os_mapcnt == 0);
15303 #endif
15304           osp->os_mapcnt = 0;
15305   }

15307   if (close_type == CLOSE_DELMAP && !did_dec_count) {
15308           ASSERT(osp->os_mapcnt >= btopr(len));

15310           if ((mmap_flags & MAP_SHARED) && (maxprot & PROT_WRITE))
15311                   osp->os_mmap_write -= btopr(len);
15312           if (maxprot & PROT_READ)
15313                   osp->os_mmap_read -= btopr(len);
15314           if (maxprot & PROT_EXEC)
15315                   osp->os_mmap_read -= btopr(len);
15316           /* mirror the PROT_NONE check in nfs4_addmap() */
15317           if (!(maxprot & PROT_READ) && !(maxprot & PROT_WRITE) &&
15318               !(maxprot & PROT_EXEC))
15319                   osp->os_mmap_read -= btopr(len);
15320           osp->os_mapcnt -= btopr(len);
15321           did_dec_count = 1;
15322   }

15324   if (recovonly) {
15325           nfs4_lost_rqst_t lost_rqst;

15327           /* request should not already be in recovery queue */
15328           ASSERT(lrp == NULL);
15329           nfs4_error_init(ep, EINTR);
```

```
15330                   nfs4close_save_lost_rqst(ep->error, &lost_rqst, oop,
15331                       osp, cred_otw, vp);
15332                   mutex_exit(&osp->os_sync_lock);
15333                   have_sync_lock = 0;
15334                   (void) nfs4_start_recovery(ep, mi, vp, NULL, NULL,
15335                       lost_rqst.lr_op == OP_CLOSE ?
15336                       &lost_rqst : NULL, OP_CLOSE, NULL, NULL, NULL);
15337                   close_failed = 1;
15338                   force_close = 0;
15339                   goto close_cleanup;
15340           }

15342           /*
15343            * If a previous OTW call got NFS4ERR_BAD_SEQID, then
15344            * we stopped operating on the open owner's <old oo_name, old seqid>
15345            * space, which means we stopped operating on the open stream
15346            * too.  So don't go OTW (as the seqid is likely bad, and the
15347            * stateid could be stale, potentially triggering a false
15348            * setclientid), and just clean up the client's internal state.
15349            */
15350           if (osp->os_orig_oo_name != oop->oo_name) {
15351                   NFS4_DEBUG(nfs4close_one_debug || nfs4_client_recov_debug,
15352                       (CE_NOTE, "nfs4close_one: skip OTW close for osp %p "
15353                       "oop %p due to bad seqid (orig oo_name %" PRIx64 " current "
15354                       "oo_name %" PRIx64")",
15355                       (void *)osp, (void *)oop, osp->os_orig_oo_name,
15356                       oop->oo_name));
15357                   close_failed = 1;
15358           }

15360           /* If the file failed recovery, just quit. */
15361           mutex_enter(&rp->r_statelock);
15362           if (rp->r_flags & R4RECOVERR) {
15363                   close_failed = 1;
15364           }
15365           mutex_exit(&rp->r_statelock);

15367           /*
15368            * If the force close path failed to obtain start_fop
15369            * then skip the OTW close and just remove the state.
15370            */
15371           if (close_failed)
15372                   goto close_cleanup;

15374           /*
15375            * Fifth, check to see if there are still mapped pages or other
15376            * opens using this open stream.  If there are then we can't
15377            * close yet but we can see if an OPEN_DOWNGRADE is necessary.
15378            */
15379           if (osp->os_open_ref_count > 0 || osp->os_mapcnt > 0) {
15380                   nfs4_lost_rqst_t        new_lost_rqst;
15381                   bool_t                  needrecov = FALSE;
15382                   cred_t                  *odg_cred_otw = NULL;
15383                   seqid4                  open_dg_seqid = 0;

15385                   if (osp->os_delegation) {
15386                           /*
15387                            * If this open stream was never OPENed OTW then we
15388                            * surely can't DOWNGRADE it (especially since the
15389                            * osp->open_stateid is really a delegation stateid
15390                            * when os_delegation is 1).
15391                            */
15392                           if (access_bits & FREAD)
15393                                   osp->os_share_acc_read--;
15394                           if (access_bits & FWRITE)
15395                                   osp->os_share_acc_write--;
```



```
15396                           osp->os_share_deny_none--;
15397                           nfs4_error_zinit(ep);
15398                           goto out;
15399                   }
15400                   nfs4_open_downgrade(access_bits, 0, oop, osp, vp, cr,
15401                       lrp, ep, &odg_cred_otw, &open_dg_seqid);
15402                   needrecov = nfs4_needs_recovery(ep, TRUE, mi->mi_vfsp);
15403                   if (needrecov && !isrecov) {
15404                           bool_t abort;
15405                           nfs4_bseqid_entry_t *bsep = NULL;

15407                           if (!ep->error && ep->stat == NFS4ERR_BAD_SEQID)
15408                                   bsep = nfs4_create_bseqid_entry(oop, NULL,
15409                                       vp, 0,
15410                                       lrp ? TAG_OPEN_DG_LOST : TAG_OPEN_DG,
15411                                       open_dg_seqid);

15413                           nfs4open_dg_save_lost_rqst(ep->error, &new_lost_rqst,
15414                               oop, osp, odg_cred_otw, vp, access_bits, 0);
15415                           mutex_exit(&osp->os_sync_lock);
15416                           have_sync_lock = 0;
15417                           abort = nfs4_start_recovery(ep, mi, vp, NULL, NULL,
15418                               new_lost_rqst.lr_op == OP_OPEN_DOWNGRADE ?
15419                               &new_lost_rqst : NULL, OP_OPEN_DOWNGRADE,
15420                               bsep, NULL, NULL);
15421                           if (odg_cred_otw)
15422                                   crfree(odg_cred_otw);
15423                           if (bsep)
15424                                   kmem_free(bsep, sizeof (*bsep));

15426                           if (abort == TRUE)
15427                                   goto out;

15429                           if (did_start_seqid_sync) {
15430                                   nfs4_end_open_seqid_sync(oop);
15431                                   did_start_seqid_sync = 0;
15432                           }
15433                           open_stream_rele(osp, rp);

15435                           if (did_start_op)
15436                                   nfs4_end_fop(mi, vp, NULL, OH_CLOSE,
15437                                       &recov_state, FALSE);
15438                           if (did_force_recovlock)
15439                                   nfs_rw_exit(&mi->mi_recovlock);

15441                           goto recov_retry;
15442                   } else {
15443                           if (odg_cred_otw)
15444                                   crfree(odg_cred_otw);
15445                   }
15446                   goto out;
15447           }

15449           /*
15450            * If this open stream was created as the results of an open
15451            * while holding a delegation, then just release it; no need
15452            * to do an OTW close.  Otherwise do a "normal" OTW close.
15453            */
15454           if (osp->os_delegation) {
15455                   nfs4close_notw(vp, osp, &have_sync_lock);
15456                   nfs4_error_zinit(ep);
15457                   goto out;
15458           }

15460           /*
15461            * If this stream is not valid, we're done.
```

```
15462          */
15463         if (!osp->os_valid) {
15464                 nfs4_error_zinit(ep);
15465                 goto out;
15466         }

15468         /*
15469          * Last open or mmap ref has vanished, need to do an OTW close.
15470          * First check to see if a close is still necessary.
15471          */
15472         if (osp->os_failed_reopen) {
15473                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
15474                     "don't close OTW osp %p since reopen failed.",
15475                     (void *)osp));
15476                 /*
15477                  * Reopen of the open stream failed, hence the
15478                  * stateid of the open stream is invalid/stale, and
15479                  * sending this OTW would incorrectly cause another
15480                  * round of recovery.  In this case, we need to set
15481                  * the 'os_valid' bit to 0 so another thread doesn't
15482                  * come in and re-open this open stream before
15483                  * this "closing" thread cleans up state (decrementing
15484                  * the nfs4_server_t's state_ref_count and decrementing
15485                  * the os_ref_count).
15486                  */
15487                 osp->os_valid = 0;
15488                 /*
15489                  * This removes the reference obtained at OPEN; ie,
15490                  * when the open stream structure was created.
15491                  *
15492                  * We don't have to worry about calling 'open_stream_rele'
15493                  * since we our currently holding a reference to this
15494                  * open stream which means the count can not go to 0 with
15495                  * this decrement.
15496                  */
15497                 ASSERT(osp->os_ref_count >= 2);
15498                 osp->os_ref_count--;
15499                 nfs4_error_zinit(ep);
15500                 close_failed = 0;
15501                 goto close_cleanup;
15502         }

15504         ASSERT(osp->os_ref_count > 1);

15506         /*
15507          * Sixth, try the CLOSE OTW.
15508          */
15509         nfs4close_otw(rp, cred_otw, oop, osp, &retry, &did_start_seqid_sync,
15510             close_type, ep, &have_sync_lock);

15512         if (ep->error == EINTR || NFS4_FRC_UNMT_ERR(ep->error, vp->v_vfsp)) {
15513                 /*
15514                  * Let the recovery thread be responsible for
15515                  * removing the state for CLOSE.
15516                  */
15517                 close_failed = 1;
15518                 force_close = 0;
15519                 retry = 0;
15520         }

15522         /* See if we need to retry with a different cred */
15523         if ((ep->error == EACCES ||
15524             (ep->error == 0 && ep->stat == NFS4ERR_ACCESS)) &&
15525             cred_otw != cr) {
15526                 crfree(cred_otw);
15527                 cred_otw = cr;
```



```
15528                 crhold(cred_otw);
15529                 retry = 1;
15530         }

15532         if (ep->error || ep->stat)
15533                 close_failed = 1;

15535         if (retry && !isrecov && num_retries-- > 0) {
15536                 if (have_sync_lock) {
15537                         mutex_exit(&osp->os_sync_lock);
15538                         have_sync_lock = 0;
15539                 }
15540                 if (did_start_seqid_sync) {
15541                         nfs4_end_open_seqid_sync(oop);
15542                         did_start_seqid_sync = 0;
15543                 }
15544                 open_stream_rele(osp, rp);

15546                 if (did_start_op)
15547                         nfs4_end_fop(mi, vp, NULL, OH_CLOSE,
15548                             &recov_state, FALSE);
15549                 if (did_force_recovlock)
15550                         nfs_rw_exit(&mi->mi_recovlock);
15551                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
15552                     "nfs4close_one: need to retry the close "
15553                     "operation"));
15554                 goto recov_retry;
15555         }
15556 close_cleanup:
15557         /*
15558          * Seventh and lastly, process our results.
15559          */
15560         if (close_failed && force_close) {
15561                 /*
15562                  * It's ok to drop and regrab the 'os_sync_lock' since
15563                  * nfs4close_notw() will recheck to make sure the
15564                  * "close"/removal of state should happen.
15565                  */
15566                 if (!have_sync_lock) {
15567                         mutex_enter(&osp->os_sync_lock);
15568                         have_sync_lock = 1;
15569                 }
15570                 /*
15571                  * This is last call, remove the ref on the open
15572                  * stream created by open and clean everything up.
15573                  */
15574                 osp->os_pending_close = 0;
15575                 nfs4close_notw(vp, osp, &have_sync_lock);
15576                 nfs4_error_zinit(ep);
15577         }

15579         if (!close_failed) {
15580                 if (have_sync_lock) {
15581                         osp->os_pending_close = 0;
15582                         mutex_exit(&osp->os_sync_lock);
15583                         have_sync_lock = 0;
15584                 } else {
15585                         mutex_enter(&osp->os_sync_lock);
15586                         osp->os_pending_close = 0;
15587                         mutex_exit(&osp->os_sync_lock);
15588                 }
15589                 if (did_start_op && recov_state.rs_sp != NULL) {
15590                         mutex_enter(&recov_state.rs_sp->s_lock);
15591                         nfs4_dec_state_ref_count_nolock(recov_state.rs_sp, mi);
15592                         mutex_exit(&recov_state.rs_sp->s_lock);
15593                 } else {
```

```
15594                           nfs4_dec_state_ref_count(mi);
15595                   }
15596                   nfs4_error_zinit(ep);
15597           }

15599   out:
15600           if (have_sync_lock)
15601                   mutex_exit(&osp->os_sync_lock);
15602           if (did_start_op)
15603                   nfs4_end_fop(mi, vp, NULL, OH_CLOSE, &recov_state,
15604                       recovonly ? TRUE : FALSE);
15605           if (did_force_recovlock)
15606                   nfs_rw_exit(&mi->mi_recovlock);
15607           if (cred_otw)
15608                   crfree(cred_otw);
15609           if (osp)
15610                   open_stream_rele(osp, rp);
15611           if (oop) {
15612                   if (did_start_seqid_sync)
15613                           nfs4_end_open_seqid_sync(oop);
15614                   open_owner_rele(oop);
15615           }
15616   }

15618   /*
15619    * Convert information returned by the server in the LOCK4denied
15620    * structure to the form required by fcntl.
15621    */
15622   static void
15623   denied_to_flk(LOCK4denied *lockt_denied, flock64_t *flk, LOCKT4args *lockt_args)
15624   {
15625           nfs4_lo_name_t *lo;

15627   #ifdef  DEBUG
15628           if (denied_to_flk_debug) {
15629                   lockt_denied_debug = lockt_denied;
15630                   debug_enter("lockt_denied");
15631           }
15632   #endif

15634           flk->l_type = lockt_denied->locktype == READ_LT ? F_RDLCK : F_WRLCK;
15635           flk->l_whence = 0;      /* aka SEEK_SET */
15636           flk->l_start = lockt_denied->offset;
15637           flk->l_len = lockt_denied->length;

15639           /*
15640            * If the blocking clientid matches our client id, then we can
15641            * interpret the lockowner (since we built it).  If not, then
15642            * fabricate a sysid and pid.  Note that the l_sysid field
15643            * in *flk already has the local sysid.
15644            */

15646           if (lockt_denied->owner.clientid == lockt_args->owner.clientid) {

15648                   if (lockt_denied->owner.owner_len == sizeof (*lo)) {
15649                           lo = (nfs4_lo_name_t *)
15650                               lockt_denied->owner.owner_val;

15652                           flk->l_pid = lo->ln_pid;
15653                   } else {
15654                           NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
15655                               "denied_to_flk: bad lock owner length\n"));

15657                           flk->l_pid = lo_to_pid(&lockt_denied->owner);
15658                   }
15659           } else {
```



```
15660                   NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
15661                   "denied_to_flk: foreign clientid\n"));

15663                   /*
15664                    * Construct a new sysid which should be different from
15665                    * sysids of other systems.
15666                    */

15668                   flk->l_sysid++;
15669                   flk->l_pid = lo_to_pid(&lockt_denied->owner);
15670           }
15671   }

15673   static pid_t
15674   lo_to_pid(lock_owner4 *lop)
15675   {
15676           pid_t pid = 0;
15677           uchar_t *cp;
15678           int i;

15680           cp = (uchar_t *)&lop->clientid;

15682           for (i = 0; i < sizeof (lop->clientid); i++)
15683                   pid += (pid_t)*cp++;

15685           cp = (uchar_t *)lop->owner_val;

15687           for (i = 0; i < lop->owner_len; i++)
15688                   pid += (pid_t)*cp++;

15690           return (pid);
15691   }

15693   /*
15694    * Given a lock pointer, returns the length of that lock.
15695    * "end" is the last locked offset the "l_len" covers from
15696    * the start of the lock.
15697    */
15698   static off64_t
15699   lock_to_end(flock64_t *lock)
15700   {
15701           off64_t lock_end;

15703           if (lock->l_len == 0)
15704                   lock_end = (off64_t)MAXEND;
15705           else
15706                   lock_end = lock->l_start + lock->l_len - 1;

15708           return (lock_end);
15709   }

15711   /*
15712    * Given the end of a lock, it will return you the length "l_len" for that lock.
15713    */
15714   static off64_t
15715   end_to_len(off64_t start, off64_t end)
15716   {
15717           off64_t lock_len;

15719           ASSERT(end >= start);
15720           if (end == MAXEND)
15721                   lock_len = 0;
15722           else
15723                   lock_len = end - start + 1;

15725           return (lock_len);
```

```
15726 }

15728 /*
15729  * On given end for a lock it determines if it is the last locked offset
15730  * or not, if so keeps it as is, else adds one to return the length for
15731  * valid start.
15732  */
15733 static off64_t
15734 start_check(off64_t x)
15735 {
15736         if (x == MAXEND)
15737                 return (x);
15738         else
15739                 return (x + 1);
15740 }

15742 /*
15743  * See if these two locks overlap, and if so return 1;
15744  * otherwise, return 0.
15745  */
15746 static int
15747 locks_intersect(flock64_t *llfp, flock64_t *curfp)
15748 {
15749         off64_t llfp_end, curfp_end;

15751         llfp_end = lock_to_end(llfp);
15752         curfp_end = lock_to_end(curfp);

15754         if (((llfp_end >= curfp->l_start) &&
15755             (llfp->l_start <= curfp->l_start)) ||
15756             ((curfp->l_start <= llfp->l_start) && (curfp_end >= llfp->l_start)))
15757                 return (1);
15758         return (0);
15759 }

15761 /*
15762  * Determine what the intersecting lock region is, and add that to the
15763  * 'nl_llpp' locklist in increasing order (by l_start).
15764  */
15765 static void
15766 nfs4_add_lock_range(flock64_t *lost_flp, flock64_t *local_flp,
15767     locklist_t **nl_llpp, vnode_t *vp)
15768 {
15769         locklist_t *intersect_llp, *tmp_fllp, *cur_fllp;
15770         off64_t lost_flp_end, local_flp_end, len, start;

15772         NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE, "nfs4_add_lock_range:"));

15774         if (!locks_intersect(lost_flp, local_flp))
15775                 return;

15777         NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE, "nfs4_add_lock_range: "
15778             "locks intersect"));

15780         lost_flp_end = lock_to_end(lost_flp);
15781         local_flp_end = lock_to_end(local_flp);

15783         /* Find the starting point of the intersecting region */
15784         if (local_flp->l_start > lost_flp->l_start)
15785                 start = local_flp->l_start;
15786         else
15787                 start = lost_flp->l_start;

15789         /* Find the lenght of the intersecting region */
15790         if (lost_flp_end < local_flp_end)
15791                 len = end_to_len(start, lost_flp_end);
```



```
15792         else
15793                 len = end_to_len(start, local_flp_end);

15795         /*
15796          * Prepare the flock structure for the intersection found and insert
15797          * it into the new list in increasing l_start order. This list contains
15798          * intersections of locks registered by the client with the local host
15799          * and the lost lock.
15800          * The lock type of this lock is the same as that of the local_flp.
15801          */
15802         intersect_llp = (locklist_t *)kmem_alloc(sizeof (locklist_t), KM_SLEEP);
15803         intersect_llp->ll_flock.l_start = start;
15804         intersect_llp->ll_flock.l_len = len;
15805         intersect_llp->ll_flock.l_type = local_flp->l_type;
15806         intersect_llp->ll_flock.l_pid = local_flp->l_pid;
15807         intersect_llp->ll_flock.l_sysid = local_flp->l_sysid;
15808         intersect_llp->ll_flock.l_whence = 0;   /* aka SEEK_SET */
15809         intersect_llp->ll_vp = vp;

15811         tmp_fllp = *nl_llpp;
15812         cur_fllp = NULL;
15813         while (tmp_fllp != NULL && tmp_fllp->ll_flock.l_start <
15814             intersect_llp->ll_flock.l_start) {
15815                 cur_fllp = tmp_fllp;
15816                 tmp_fllp = tmp_fllp->ll_next;
15817         }
15818         if (cur_fllp == NULL) {
15819                 /* first on the list */
15820                 intersect_llp->ll_next = *nl_llpp;
15821                 *nl_llpp = intersect_llp;
15822         } else {
15823                 intersect_llp->ll_next = cur_fllp->ll_next;
15824                 cur_fllp->ll_next = intersect_llp;
15825         }

15827         NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE, "nfs4_add_lock_range: "
15828             "created lock region: start %"PRIx64" end %"PRIx64" : %s\n",
15829             intersect_llp->ll_flock.l_start,
15830             intersect_llp->ll_flock.l_start + intersect_llp->ll_flock.l_len,
15831             intersect_llp->ll_flock.l_type == F_RDLCK ? "READ" : "WRITE"));
15832 }

15834 /*
15835  * Our local locking current state is potentially different than
15836  * what the NFSv4 server thinks we have due to a lost lock that was
15837  * resent and then received.  We need to reset our "NFSv4" locking
15838  * state to match the current local locking state for this pid since
15839  * that is what the user/application sees as what the world is.
15840  *
15841  * We cannot afford to drop the open/lock seqid sync since then we can
15842  * get confused about what the current local locking state "is" versus
15843  * "was".
15844  *
15845  * If we are unable to fix up the locks, we send SIGLOST to the affected
15846  * process.  This is not done if the filesystem has been forcibly
15847  * unmounted, in case the process has already exited and a new process
15848  * exists with the same pid.
15849  */
15850 static void
15851 nfs4_reinstitute_local_lock_state(vnode_t *vp, flock64_t *lost_flp, cred_t *cr,
15852     nfs4_lock_owner_t *lop)
15853 {
15854         locklist_t *locks, *llp, *ri_llp, *tmp_llp;
15855         mntinfo4_t *mi = VTOMI4(vp);
15856         const int cmd = F_SETLK;
15857         off64_t cur_start, llp_ll_flock_end, lost_flp_end;
```

```
15858         flock64_t ul_fl;
15859
15860         NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE,
15861             "nfs4_reinstitute_local_lock_state"));
15862
15863         /*
15864          * Find active locks for this vp from the local locking code.
15865          * Scan through this list and find out the locks that intersect with
15866          * the lost lock. Once we find the lock that intersects, add the
15867          * intersection area as a new lock to a new list "ri_llp". The lock
15868          * type of the intersection region lock added to ri_llp is the same
15869          * as that found in the active lock list, "list". The intersecting
15870          * region locks are added to ri_llp in increasing l_start order.
15871          */
15872         ASSERT(nfs_zone() == mi->mi_zone);
15873
15874         locks = flk_active_locks_for_vp(vp);
15875         ri_llp = NULL;
15876
15877         for (llp = locks; llp != NULL; llp = llp->ll_next) {
15878                 ASSERT(llp->ll_vp == vp);
15879                 /*
15880                  * Pick locks that belong to this pid/lockowner
15881                  */
15882                 if (llp->ll_flock.l_pid != lost_flp->l_pid)
15883                         continue;
15884
15885                 nfs4_add_lock_range(lost_flp, &llp->ll_flock, &ri_llp, vp);
15886         }
15887
15888         /*
15889          * Now we have the list of intersections with the lost lock. These are
15890          * the locks that were/are active before the server replied to the
15891          * last/lost lock. Issue these locks to the server here. Playing these
15892          * locks to the server will re-establish aur current local locking state
15893          * with the v4 server.
15894          * If we get an error, send SIGLOST to the application for that lock.
15895          */
15896
15897         for (llp = ri_llp; llp != NULL; llp = llp->ll_next) {
15898                 NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE,
15899                     "nfs4_reinstitute_local_lock_state: need to issue "
15900                     "flock: [%"PRIx64" - %"PRIx64"] : %s",
15901                     llp->ll_flock.l_start,
15902                     llp->ll_flock.l_start + llp->ll_flock.l_len,
15903                     llp->ll_flock.l_type == F_RDLCK ? "READ" :
15904                     llp->ll_flock.l_type == F_WRLCK ? "WRITE" : "INVALID"));
15905                 /*
15906                  * No need to relock what we already have
15907                  */
15908                 if (llp->ll_flock.l_type == lost_flp->l_type)
15909                         continue;
15910
15911                 push_reinstate(vp, cmd, &llp->ll_flock, cr, lop);
15912         }
15913
15914         /*
15915          * Now keeping the start of the lost lock as our reference parse the
15916          * newly created ri_llp locklist to find the ranges that we have locked
15917          * with the v4 server but not in the current local locking. We need
15918          * to unlock these ranges.
15919          * These ranges can also be reffered to as those ranges, where the lost
15920          * lock does not overlap with the locks in the ri_llp but are locked
15921          * since the server replied to the lost lock.
15922          */
15923         cur_start = lost_flp->l_start;
```



```
15924         lost_flp_end = lock_to_end(lost_flp);
15925
15926         ul_fl.l_type = F_UNLCK;
15927         ul_fl.l_whence = 0;     /* aka SEEK_SET */
15928         ul_fl.l_sysid = lost_flp->l_sysid;
15929         ul_fl.l_pid = lost_flp->l_pid;
15930
15931         for (llp = ri_llp; llp != NULL; llp = llp->ll_next) {
15932                 llp_ll_flock_end = lock_to_end(&llp->ll_flock);
15933
15934                 if (llp->ll_flock.l_start <= cur_start) {
15935                         cur_start = start_check(llp_ll_flock_end);
15936                         continue;
15937                 }
15938                 NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE,
15939                     "nfs4_reinstitute_local_lock_state: "
15940                     "UNLOCK [%"PRIx64" - %"PRIx64"]",
15941                     cur_start, llp->ll_flock.l_start));
15942
15943                 ul_fl.l_start = cur_start;
15944                 ul_fl.l_len = end_to_len(cur_start,
15945                     (llp->ll_flock.l_start - 1));
15946
15947                 push_reinstate(vp, cmd, &ul_fl, cr, lop);
15948                 cur_start = start_check(llp_ll_flock_end);
15949         }
15950
15951         /*
15952          * In the case where the lost lock ends after all intersecting locks,
15953          * unlock the last part of the lost lock range.
15954          */
15955         if (cur_start != start_check(lost_flp_end)) {
15956                 NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE,
15957                     "nfs4_reinstitute_local_lock_state: UNLOCK end of the "
15958                     "lost lock region [%"PRIx64" - %"PRIx64"]",
15959                     cur_start, lost_flp->l_start + lost_flp->l_len));
15960
15961                 ul_fl.l_start = cur_start;
15962                 /*
15963                  * Is it an to-EOF lock? if so unlock till the end
15964                  */
15965                 if (lost_flp->l_len == 0)
15966                         ul_fl.l_len = 0;
15967                 else
15968                         ul_fl.l_len = start_check(lost_flp_end) - cur_start;
15969
15970                 push_reinstate(vp, cmd, &ul_fl, cr, lop);
15971         }
15972
15973         if (locks != NULL)
15974                 flk_free_locklist(locks);
15975
15976         /* Free up our newly created locklist */
15977         for (llp = ri_llp; llp != NULL; ) {
15978                 tmp_llp = llp->ll_next;
15979                 kmem_free(llp, sizeof (locklist_t));
15980                 llp = tmp_llp;
15981         }
15982
15983         /*
15984          * Now return back to the original calling nfs4frlock()
15985          * and let us naturally drop our seqid syncs.
15986          */
15987 }
15988
15989 /*
```

```
15990  * Create a lost state record for the given lock reinstantiation request
15991  * and push it onto the lost state queue.
15992  */
15993 static void
15994 push_reinstate(vnode_t *vp, int cmd, flock64_t *flk, cred_t *cr,
15995     nfs4_lock_owner_t *lop)
15996 {
15997         nfs4_lost_rqst_t req;
15998         nfs_lock_type4 locktype;
15999         nfs4_error_t e = { EINTR, NFS4_OK, RPC_SUCCESS };

16001         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

16003         locktype = flk_to_locktype(cmd, flk->l_type);
16004         nfs4frlock_save_lost_rqst(NFS4_LCK_CTYPE_REINSTATE, EINTR, locktype,
16005             NULL, NULL, lop, flk, &req, cr, vp);
16006         (void) nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
16007             (req.lr_op == OP_LOCK || req.lr_op == OP_LOCKU) ?
16008             &req : NULL, flk->l_type == F_UNLCK ? OP_LOCKU : OP_LOCK,
16009             NULL, NULL, NULL);
16010 }
```