```
**********************************************************
  107715 Mon Sep 21 10:48:12 2015
new/usr/src/uts/common/os/flock.c
6253 F_GETLK doesn't always return lock owner
The F_GETLK fcntl doesn't return the offending lock if there is a read lock
on the file, a waiting write lock, and a read lock is requested.
The write lock blocks the locking request, but without this patch isn't
returned by GETLK.
**********************************************************
_____unchanged_portion_omitted_

2070 /*
2071  * Finds the first lock that is mainly responsible for blocking this
2072  * request.  If there is no such lock, request->l_flock.l_type is set to
2073  * F_UNLCK.  Otherwise, request->l_flock is filled in with the particulars
2074  * of the blocking lock.
2075  *
2076  * Note: It is possible a request is blocked by a sleeping lock because
2077  * of the fairness policy used in flk_process_request() to construct the
2078  * dependencies. (see comments before flk_process_request()).
2079  */

2081 static void
2082 flk_get_first_blocking_lock(lock_descriptor_t *request)
2083 {
2084         graph_t *gp = request->l_graph;
2085         vnode_t *vp = request->l_vnode;
2086         lock_descriptor_t *lock, *blocker;

2088         ASSERT(MUTEX_HELD(&gp->gp_mutex));
2089         blocker = NULL;
2090         SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

2092         if (lock) {
2093                 do {
2094                         if (BLOCKS(lock, request)) {
2095                                 blocker = lock;
2096                                 break;
2097                         }
2098                         lock = lock->l_next;
2099                 } while (lock->l_vnode == vp);
2100         }

2102         if (blocker == NULL && request->l_flock.l_type == F_RDLCK) {
2103                 /*
2104                  * No active lock is blocking this request, but if a read
2105                  * lock is requested, it may also get blocked by a waiting
2106                  * writer. So search all sleeping locks and see if there is
2107                  * a writer waiting.
2108                  */
2109                 SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);
2110                 if (lock) {
2111                         do {
2112                                 if (BLOCKS(lock, request)) {
2113                                         blocker = lock;
2114                                         break;
2115                                 }
2116                                 lock = lock->l_next;
2117                         } while (lock->l_vnode == vp);
2118                 }
2119         }

2121 #endif /* ! codereview */
2122         if (blocker) {
2123                 report_blocker(blocker, request);
2124         } else
```



```
2125                 request->l_flock.l_type = F_UNLCK;
2126 }

2128 /*
2129  * Get the graph_t structure associated with a vnode.
2130  * If 'initialize' is non-zero, and the graph_t structure for this vnode has
2131  * not yet been initialized, then a new element is allocated and returned.
2132  */
2133 graph_t *
2134 flk_get_lock_graph(vnode_t *vp, int initialize)
2135 {
2136         graph_t *gp;
2137         graph_t *gp_alloc = NULL;
2138         int index = HASH_INDEX(vp);

2140         if (initialize == FLK_USE_GRAPH) {
2141                 mutex_enter(&flock_lock);
2142                 gp = lock_graph[index];
2143                 mutex_exit(&flock_lock);
2144                 return (gp);
2145         }

2147         ASSERT(initialize == FLK_INIT_GRAPH);

2149         if (lock_graph[index] == NULL) {

2151                 gp_alloc = kmem_zalloc(sizeof (graph_t), KM_SLEEP);

2153                 /* Initialize the graph */

2155                 gp_alloc->active_locks.l_next =
2156                     gp_alloc->active_locks.l_prev =
2157                     (lock_descriptor_t *)ACTIVE_HEAD(gp_alloc);
2158                 gp_alloc->sleeping_locks.l_next =
2159                     gp_alloc->sleeping_locks.l_prev =
2160                     (lock_descriptor_t *)SLEEPING_HEAD(gp_alloc);
2161                 gp_alloc->index = index;
2162                 mutex_init(&gp_alloc->gp_mutex, NULL, MUTEX_DEFAULT, NULL);
2163         }

2165         mutex_enter(&flock_lock);

2167         gp = lock_graph[index];

2169         /* Recheck the value within flock_lock */
2170         if (gp == NULL) {
2171                 struct flock_globals *fg;

2173                 /* We must have previously allocated the graph_t structure */
2174                 ASSERT(gp_alloc != NULL);
2175                 lock_graph[index] = gp = gp_alloc;
2176                 /*
2177                  * The lockmgr status is only needed if KLM is loaded.
2178                  */
2179                 if (flock_zone_key != ZONE_KEY_UNINITIALIZED) {
2180                         fg = flk_get_globals();
2181                         fg->lockmgr_status[index] = fg->flk_lockmgr_status;
2182                 }
2183         }

2185         mutex_exit(&flock_lock);

2187         if ((gp_alloc != NULL) && (gp != gp_alloc)) {
2188                 /* There was a race to allocate the graph_t and we lost */
2189                 mutex_destroy(&gp_alloc->gp_mutex);
2190                 kmem_free(gp_alloc, sizeof (graph_t));
```

```
2191         }

2193         return (gp);
2194 }

2196 /*
2197  * PSARC case 1997/292
2198  */
2199 int
2200 cl_flk_has_remote_locks_for_nlmid(vnode_t *vp, int nlmid)
2201 {
2202         lock_descriptor_t *lock;
2203         int result = 0;
2204         graph_t *gp;
2205         int                     lock_nlmid;

2207         /*
2208          * Check to see if node is booted as a cluster. If not, return.
2209          */
2210         if ((cluster_bootflags & CLUSTER_BOOTED) == 0) {
2211                 return (0);
2212         }

2214         gp = flk_get_lock_graph(vp, FLK_USE_GRAPH);
2215         if (gp == NULL) {
2216                 return (0);
2217         }

2219         mutex_enter(&gp->gp_mutex);

2221         SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

2223         if (lock) {
2224                 while (lock->l_vnode == vp) {
2225                         /* get NLM id from sysid */
2226                         lock_nlmid = GETNLMID(lock->l_flock.l_sysid);

2228                         /*
2229                          * If NLM server request _and_ nlmid of lock matches
2230                          * nlmid of argument, then we've found a remote lock.
2231                          */
2232                         if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
2233                                 result = 1;
2234                                 goto done;
2235                         }
2236                         lock = lock->l_next;
2237                 }
2238         }

2240         SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

2242         if (lock) {
2243                 while (lock->l_vnode == vp) {
2244                         /* get NLM id from sysid */
2245                         lock_nlmid = GETNLMID(lock->l_flock.l_sysid);

2247                         /*
2248                          * If NLM server request _and_ nlmid of lock matches
2249                          * nlmid of argument, then we've found a remote lock.
2250                          */
2251                         if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
2252                                 result = 1;
2253                                 goto done;
2254                         }
2255                         lock = lock->l_next;
2256                 }
```



```
2257         }

2259 done:
2260         mutex_exit(&gp->gp_mutex);
2261         return (result);
2262 }

2264 /*
2265  * Determine whether there are any locks for the given vnode with a remote
2266  * sysid.  Returns zero if not, non-zero if there are.
2267  *
2268  * Note that the return value from this function is potentially invalid
2269  * once it has been returned.  The caller is responsible for providing its
2270  * own synchronization mechanism to ensure that the return value is useful
2271  * (e.g., see nfs_lockcompletion()).
2272  */
2273 int
2274 flk_has_remote_locks(vnode_t *vp)
2275 {
2276         lock_descriptor_t *lock;
2277         int result = 0;
2278         graph_t *gp;

2280         gp = flk_get_lock_graph(vp, FLK_USE_GRAPH);
2281         if (gp == NULL) {
2282                 return (0);
2283         }

2285         mutex_enter(&gp->gp_mutex);

2287         SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

2289         if (lock) {
2290                 while (lock->l_vnode == vp) {
2291                         if (IS_REMOTE(lock)) {
2292                                 result = 1;
2293                                 goto done;
2294                         }
2295                         lock = lock->l_next;
2296                 }
2297         }

2299         SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

2301         if (lock) {
2302                 while (lock->l_vnode == vp) {
2303                         if (IS_REMOTE(lock)) {
2304                                 result = 1;
2305                                 goto done;
2306                         }
2307                         lock = lock->l_next;
2308                 }
2309         }

2311 done:
2312         mutex_exit(&gp->gp_mutex);
2313         return (result);
2314 }

2316 /*
2317  * Determine whether there are any locks for the given vnode with a remote
2318  * sysid matching given sysid.
2319  * Used by the new (open source) NFS Lock Manager (NLM)
2320  */
2321 int
2322 flk_has_remote_locks_for_sysid(vnode_t *vp, int sysid)
```

```
{
        lock_descriptor_t *lock;
        int result = 0;
        graph_t *gp;

        if (sysid == 0)
                return (0);

        gp = flk_get_lock_graph(vp, FLK_USE_GRAPH);
        if (gp == NULL) {
                return (0);
        }

        mutex_enter(&gp->gp_mutex);

        SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

        if (lock) {
                while (lock->l_vnode == vp) {
                        if (lock->l_flock.l_sysid == sysid) {
                                result = 1;
                                goto done;
                        }
                        lock = lock->l_next;
                }
        }

        SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

        if (lock) {
                while (lock->l_vnode == vp) {
                        if (lock->l_flock.l_sysid == sysid) {
                                result = 1;
                                goto done;
                        }
                        lock = lock->l_next;
                }
        }

done:
        mutex_exit(&gp->gp_mutex);
        return (result);
}

/*
 * Determine if there are any locks owned by the given sysid.
 * Returns zero if not, non-zero if there are.  Note that this return code
 * could be derived from flk_get_{sleeping,active}_locks, but this routine
 * avoids all the memory allocations of those routines.
 *
 * This routine has the same synchronization issues as
 * flk_has_remote_locks.
 */

int
flk_sysid_has_locks(int sysid, int lck_type)
{
        int             has_locks = 0;
        lock_descriptor_t       *lock;
        graph_t         *gp;
        int             i;

        for (i = 0; i < HASH_SIZE && !has_locks; i++) {
                mutex_enter(&flock_lock);
                gp = lock_graph[i];
                mutex_exit(&flock_lock);
```



```
                if (gp == NULL) {
                        continue;
                }

                mutex_enter(&gp->gp_mutex);

                if (lck_type & FLK_QUERY_ACTIVE) {
                        for (lock = ACTIVE_HEAD(gp)->l_next;
                            lock != ACTIVE_HEAD(gp) && !has_locks;
                            lock = lock->l_next) {
                                if (lock->l_flock.l_sysid == sysid)
                                        has_locks = 1;
                        }
                }

                if (lck_type & FLK_QUERY_SLEEPING) {
                        for (lock = SLEEPING_HEAD(gp)->l_next;
                            lock != SLEEPING_HEAD(gp) && !has_locks;
                            lock = lock->l_next) {
                                if (lock->l_flock.l_sysid == sysid)
                                        has_locks = 1;
                        }
                }
                mutex_exit(&gp->gp_mutex);
        }

        return (has_locks);
}


/*
 * PSARC case 1997/292
 *
 * Requires: "sysid" is a pair [nlmid, sysid].  The lower half is 16-bit
 *  quantity, the real sysid generated by the NLM server; the upper half
 *  identifies the node of the cluster where the NLM server ran.
 *  This routine is only called by an NLM server running in a cluster.
 * Effects: Remove all locks held on behalf of the client identified
 *  by "sysid."
 */
void
cl_flk_remove_locks_by_sysid(int sysid)
{
        graph_t *gp;
        int i;
        lock_descriptor_t *lock, *nlock;

        /*
         * Check to see if node is booted as a cluster. If not, return.
         */
        if ((cluster_bootflags & CLUSTER_BOOTED) == 0) {
                return;
        }

        ASSERT(sysid != 0);
        for (i = 0; i < HASH_SIZE; i++) {
                mutex_enter(&flock_lock);
                gp = lock_graph[i];
                mutex_exit(&flock_lock);

                if (gp == NULL)
                        continue;

                mutex_enter(&gp->gp_mutex);     /*  get mutex on lock graph */

                /* signal sleeping requests so that they bail out */
```

```
2455                 lock = SLEEPING_HEAD(gp)->l_next;
2456                 while (lock != SLEEPING_HEAD(gp)) {
2457                         nlock = lock->l_next;
2458                         if (lock->l_flock.l_sysid == sysid) {
2459                                 INTERRUPT_WAKEUP(lock);
2460                         }
2461                         lock = nlock;
2462                 }

2464                 /* delete active locks */
2465                 lock = ACTIVE_HEAD(gp)->l_next;
2466                 while (lock != ACTIVE_HEAD(gp)) {
2467                         nlock = lock->l_next;
2468                         if (lock->l_flock.l_sysid == sysid) {
2469                                 flk_delete_active_lock(lock, 0);
2470                                 flk_wakeup(lock, 1);
2471                                 flk_free_lock(lock);
2472                         }
2473                         lock = nlock;
2474                 }
2475                 mutex_exit(&gp->gp_mutex);    /* release mutex on lock graph */
2476         }
2477 }

2479 /*
2480  * Delete all locks in the system that belongs to the sysid of the request.
2481  */

2483 static void
2484 flk_delete_locks_by_sysid(lock_descriptor_t *request)
2485 {
2486         int     sysid  = request->l_flock.l_sysid;
2487         lock_descriptor_t *lock, *nlock;
2488         graph_t *gp;
2489         int i;

2491         ASSERT(MUTEX_HELD(&request->l_graph->gp_mutex));
2492         ASSERT(sysid != 0);

2494         mutex_exit(&request->l_graph->gp_mutex);

2496         for (i = 0; i < HASH_SIZE; i++) {
2497                 mutex_enter(&flock_lock);
2498                 gp = lock_graph[i];
2499                 mutex_exit(&flock_lock);

2501                 if (gp == NULL)
2502                         continue;

2504                 mutex_enter(&gp->gp_mutex);

2506                 /* signal sleeping requests so that they bail out */
2507                 lock = SLEEPING_HEAD(gp)->l_next;
2508                 while (lock != SLEEPING_HEAD(gp)) {
2509                         nlock = lock->l_next;
2510                         if (lock->l_flock.l_sysid == sysid) {
2511                                 INTERRUPT_WAKEUP(lock);
2512                         }
2513                         lock = nlock;
2514                 }

2516                 /* delete active locks */
2517                 lock = ACTIVE_HEAD(gp)->l_next;
2518                 while (lock != ACTIVE_HEAD(gp)) {
2519                         nlock = lock->l_next;
2520                         if (lock->l_flock.l_sysid == sysid) {
```



```
2521                                 flk_delete_active_lock(lock, 0);
2522                                 flk_wakeup(lock, 1);
2523                                 flk_free_lock(lock);
2524                         }
2525                         lock = nlock;
2526                 }
2527                 mutex_exit(&gp->gp_mutex);
2528         }

2530         mutex_enter(&request->l_graph->gp_mutex);
2531 }

2533 /*
2534  * Clustering: Deletes PXFS locks
2535  * Effects: Delete all locks on files in the given file system and with the
2536  *  given PXFS id.
2537  */
2538 void
2539 cl_flk_delete_pxfs_locks(struct vfs *vfsp, int pxfsid)
2540 {
2541         lock_descriptor_t *lock, *nlock;
2542         graph_t *gp;
2543         int i;

2545         for (i = 0; i < HASH_SIZE; i++) {
2546                 mutex_enter(&flock_lock);
2547                 gp = lock_graph[i];
2548                 mutex_exit(&flock_lock);

2550                 if (gp == NULL)
2551                         continue;

2553                 mutex_enter(&gp->gp_mutex);

2555                 /* signal sleeping requests so that they bail out */
2556                 lock = SLEEPING_HEAD(gp)->l_next;
2557                 while (lock != SLEEPING_HEAD(gp)) {
2558                         nlock = lock->l_next;
2559                         if (lock->l_vnode->v_vfsp == vfsp) {
2560                                 ASSERT(IS_PXFS(lock));
2561                                 if (GETPXFSID(lock->l_flock.l_sysid) ==
2562                                     pxfsid) {
2563                                         flk_set_state(lock,
2564                                             FLK_CANCELLED_STATE);
2565                                         flk_cancel_sleeping_lock(lock, 1);
2566                                 }
2567                         }
2568                         lock = nlock;
2569                 }

2571                 /* delete active locks */
2572                 lock = ACTIVE_HEAD(gp)->l_next;
2573                 while (lock != ACTIVE_HEAD(gp)) {
2574                         nlock = lock->l_next;
2575                         if (lock->l_vnode->v_vfsp == vfsp) {
2576                                 ASSERT(IS_PXFS(lock));
2577                                 if (GETPXFSID(lock->l_flock.l_sysid) ==
2578                                     pxfsid) {
2579                                         flk_delete_active_lock(lock, 0);
2580                                         flk_wakeup(lock, 1);
2581                                         flk_free_lock(lock);
2582                                 }
2583                         }
2584                         lock = nlock;
2585                 }
2586                 mutex_exit(&gp->gp_mutex);
```

```
2587         }
2588 }

2590 /*
2591  * Search for a sleeping lock manager lock which matches exactly this lock
2592  * request; if one is found, fake a signal to cancel it.
2593  *
2594  * Return 1 if a matching lock was found, 0 otherwise.
2595  */

2597 static int
2598 flk_canceled(lock_descriptor_t *request)
2599 {
2600         lock_descriptor_t *lock, *nlock;
2601         graph_t *gp = request->l_graph;
2602         vnode_t *vp = request->l_vnode;

2604         ASSERT(MUTEX_HELD(&gp->gp_mutex));
2605         ASSERT(IS_LOCKMGR(request));
2606         SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

2608         if (lock) {
2609                 while (lock->l_vnode == vp) {
2610                         nlock = lock->l_next;
2611                         if (SAME_OWNER(lock, request) &&
2612                             lock->l_start == request->l_start &&
2613                             lock->l_end == request->l_end) {
2614                                 INTERRUPT_WAKEUP(lock);
2615                                 return (1);
2616                         }
2617                         lock = nlock;
2618                 }
2619         }
2620         return (0);
2621 }

2623 /*
2624  * Remove all the locks for the vnode belonging to the given pid and sysid.
2625  */

2627 void
2628 cleanlocks(vnode_t *vp, pid_t pid, int sysid)
2629 {
2630         graph_t *gp;
2631         lock_descriptor_t *lock, *nlock;
2632         lock_descriptor_t *link_stack;

2634         STACK_INIT(link_stack);

2636         gp = flk_get_lock_graph(vp, FLK_USE_GRAPH);

2638         if (gp == NULL)
2639                 return;
2640         mutex_enter(&gp->gp_mutex);

2642         CHECK_SLEEPING_LOCKS(gp);
2643         CHECK_ACTIVE_LOCKS(gp);

2645         SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

2647         if (lock) {
2648                 do {
2649                         nlock = lock->l_next;
2650                         if ((lock->l_flock.l_pid == pid ||
2651                             pid == IGN_PID) &&
2652                             lock->l_flock.l_sysid == sysid) {
```



```
2653                                 CANCEL_WAKEUP(lock);
2654                         }
2655                         lock = nlock;
2656                 } while (lock->l_vnode == vp);
2657         }

2659         SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

2661         if (lock) {
2662                 do {
2663                         nlock = lock->l_next;
2664                         if ((lock->l_flock.l_pid == pid ||
2665                             pid == IGN_PID) &&
2666                             lock->l_flock.l_sysid == sysid) {
2667                                 flk_delete_active_lock(lock, 0);
2668                                 STACK_PUSH(link_stack, lock, l_stack);
2669                         }
2670                         lock = nlock;
2671                 } while (lock->l_vnode == vp);
2672         }

2674         while ((lock = STACK_TOP(link_stack)) != NULL) {
2675                 STACK_POP(link_stack, l_stack);
2676                 flk_wakeup(lock, 1);
2677                 flk_free_lock(lock);
2678         }

2680         CHECK_SLEEPING_LOCKS(gp);
2681         CHECK_ACTIVE_LOCKS(gp);
2682         CHECK_OWNER_LOCKS(gp, pid, sysid, vp);
2683         mutex_exit(&gp->gp_mutex);
2684 }


2687 /*
2688  * Called from 'fs' read and write routines for files that have mandatory
2689  * locking enabled.
2690  */

2692 int
2693 chklock(
2694         struct vnode    *vp,
2695         int             iomode,
2696         u_offset_t      offset,
2697         ssize_t         len,
2698         int             fmode,
2699         caller_context_t *ct)
2700 {
2701         register int    i;
2702         struct flock64  bf;
2703         int             error = 0;

2705         bf.l_type = (iomode & FWRITE) ? F_WRLCK : F_RDLCK;
2706         bf.l_whence = 0;
2707         bf.l_start = offset;
2708         bf.l_len = len;
2709         if (ct == NULL) {
2710                 bf.l_pid = curproc->p_pid;
2711                 bf.l_sysid = 0;
2712         } else {
2713                 bf.l_pid = ct->cc_pid;
2714                 bf.l_sysid = ct->cc_sysid;
2715         }
2716         i = (fmode & (FNDELAY|FNONBLOCK)) ? INOFLCK : INOFLCK|SLPFLCK;
2717         if ((i = reclock(vp, &bf, i, 0, offset, NULL)) != 0 ||
2718             bf.l_type != F_UNLCK)
```

```
2719                 error = i ? i : EAGAIN;
2720         return (error);
2721 }

2723 /*
2724  * convoff - converts the given data (start, whence) to the
2725  * given whence.
2726  */
2727 int
2728 convoff(vp, lckdat, whence, offset)
2729         struct vnode    *vp;
2730         struct flock64  *lckdat;
2731         int             whence;
2732         offset_t        offset;
2733 {
2734         int             error;
2735         struct vattr    vattr;

2737         if ((lckdat->l_whence == 2) || (whence == 2)) {
2738                 vattr.va_mask = AT_SIZE;
2739                 if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
2740                         return (error);
2741         }

2743         switch (lckdat->l_whence) {
2744         case 1:
2745                 lckdat->l_start += offset;
2746                 break;
2747         case 2:
2748                 lckdat->l_start += vattr.va_size;
2749                 /* FALLTHRU */
2750         case 0:
2751                 break;
2752         default:
2753                 return (EINVAL);
2754         }

2756         if (lckdat->l_start < 0)
2757                 return (EINVAL);

2759         switch (whence) {
2760         case 1:
2761                 lckdat->l_start -= offset;
2762                 break;
2763         case 2:
2764                 lckdat->l_start -= vattr.va_size;
2765                 /* FALLTHRU */
2766         case 0:
2767                 break;
2768         default:
2769                 return (EINVAL);
2770         }

2772         lckdat->l_whence = (short)whence;
2773         return (0);
2774 }


2777 /*      proc_graph function definitions */

2779 /*
2780  * Function checks for deadlock due to the new 'lock'. If deadlock found
2781  * edges of this lock are freed and returned.
2782  */

2784 static int
```



```
2785 flk_check_deadlock(lock_descriptor_t *lock)
2786 {
2787         proc_vertex_t   *start_vertex, *pvertex;
2788         proc_vertex_t *dvertex;
2789         proc_edge_t *pep, *ppep;
2790         edge_t  *ep, *nep;
2791         proc_vertex_t *process_stack;

2793         STACK_INIT(process_stack);

2795         mutex_enter(&flock_lock);
2796         start_vertex = flk_get_proc_vertex(lock);
2797         ASSERT(start_vertex != NULL);

2799         /* construct the edges from this process to other processes */

2801         ep = FIRST_ADJ(lock);
2802         while (ep != HEAD(lock)) {
2803                 proc_vertex_t *adj_proc;

2805                 adj_proc = flk_get_proc_vertex(ep->to_vertex);
2806                 for (pep = start_vertex->edge; pep != NULL; pep = pep->next) {
2807                         if (pep->to_proc == adj_proc) {
2808                                 ASSERT(pep->refcount);
2809                                 pep->refcount++;
2810                                 break;
2811                         }
2812                 }
2813                 if (pep == NULL) {
2814                         pep = flk_get_proc_edge();
2815                         pep->to_proc = adj_proc;
2816                         pep->refcount = 1;
2817                         adj_proc->incount++;
2818                         pep->next = start_vertex->edge;
2819                         start_vertex->edge = pep;
2820                 }
2821                 ep = NEXT_ADJ(ep);
2822         }

2824         ep = FIRST_IN(lock);

2826         while (ep != HEAD(lock)) {
2827                 proc_vertex_t *in_proc;

2829                 in_proc = flk_get_proc_vertex(ep->from_vertex);

2831                 for (pep = in_proc->edge; pep != NULL; pep = pep->next) {
2832                         if (pep->to_proc == start_vertex) {
2833                                 ASSERT(pep->refcount);
2834                                 pep->refcount++;
2835                                 break;
2836                         }
2837                 }
2838                 if (pep == NULL) {
2839                         pep = flk_get_proc_edge();
2840                         pep->to_proc = start_vertex;
2841                         pep->refcount = 1;
2842                         start_vertex->incount++;
2843                         pep->next = in_proc->edge;
2844                         in_proc->edge = pep;
2845                 }
2846                 ep = NEXT_IN(ep);
2847         }

2849         if (start_vertex->incount == 0) {
2850                 mutex_exit(&flock_lock);
```

```
2851                return (0);
2852        }
2854        flk_proc_graph_uncolor();
2856        start_vertex->p_sedge = start_vertex->edge;
2858        STACK_PUSH(process_stack, start_vertex, p_stack);
2860        while ((pvertex = STACK_TOP(process_stack)) != NULL) {
2861                for (pep = pvertex->p_sedge; pep != NULL; pep = pep->next) {
2862                        dvertex = pep->to_proc;
2863                        if (!PROC_ARRIVED(dvertex)) {
2864                                STACK_PUSH(process_stack, dvertex, p_stack);
2865                                dvertex->p_sedge = dvertex->edge;
2866                                PROC_ARRIVE(pvertex);
2867                                pvertex->p_sedge = pep->next;
2868                                break;
2869                        }
2870                        if (!PROC_DEPARTED(dvertex))
2871                                goto deadlock;
2872                }
2873                if (pep == NULL) {
2874                        PROC_DEPART(pvertex);
2875                        STACK_POP(process_stack, p_stack);
2876                }
2877        }
2878        mutex_exit(&flock_lock);
2879        return (0);
2881 deadlock:
2883        /* we remove all lock edges and proc edges */
2885        ep = FIRST_ADJ(lock);
2886        while (ep != HEAD(lock)) {
2887                proc_vertex_t *adj_proc;
2888                adj_proc = flk_get_proc_vertex(ep->to_vertex);
2889                nep = NEXT_ADJ(ep);
2890                IN_LIST_REMOVE(ep);
2891                ADJ_LIST_REMOVE(ep);
2892                flk_free_edge(ep);
2893                ppep = start_vertex->edge;
2894                for (pep = start_vertex->edge; pep != NULL; ppep = pep,
2895                    pep = ppep->next) {
2896                        if (pep->to_proc == adj_proc) {
2897                                pep->refcount--;
2898                                if (pep->refcount == 0) {
2899                                        if (pep == ppep) {
2900                                                start_vertex->edge = pep->next;
2901                                        } else {
2902                                                ppep->next = pep->next;
2903                                        }
2904                                        adj_proc->incount--;
2905                                        flk_proc_release(adj_proc);
2906                                        flk_free_proc_edge(pep);
2907                                }
2908                                break;
2909                        }
2910                }
2911                ep = nep;
2912        }
2913        ep = FIRST_IN(lock);
2914        while (ep != HEAD(lock)) {
2915                proc_vertex_t *in_proc;
2916                in_proc = flk_get_proc_vertex(ep->from_vertex);
```



```
2917                nep = NEXT_IN(ep);
2918                IN_LIST_REMOVE(ep);
2919                ADJ_LIST_REMOVE(ep);
2920                flk_free_edge(ep);
2921                ppep = in_proc->edge;
2922                for (pep = in_proc->edge; pep != NULL; ppep = pep,
2923                    pep = ppep->next) {
2924                        if (pep->to_proc == start_vertex) {
2925                                pep->refcount--;
2926                                if (pep->refcount == 0) {
2927                                        if (pep == ppep) {
2928                                                in_proc->edge = pep->next;
2929                                        } else {
2930                                                ppep->next = pep->next;
2931                                        }
2932                                        start_vertex->incount--;
2933                                        flk_proc_release(in_proc);
2934                                        flk_free_proc_edge(pep);
2935                                }
2936                                break;
2937                        }
2938                }
2939                ep = nep;
2940        }
2941        flk_proc_release(start_vertex);
2942        mutex_exit(&flock_lock);
2943        return (1);
2944 }
2946 /*
2947  * Get a proc vertex. If lock's pvertex value gets a correct proc vertex
2948  * from the list we return that, otherwise we allocate one. If necessary,
2949  * we grow the list of vertices also.
2950  */
2952 static proc_vertex_t *
2953 flk_get_proc_vertex(lock_descriptor_t *lock)
2954 {
2955        int i;
2956        proc_vertex_t   *pv;
2957        proc_vertex_t   **palloc;
2959        ASSERT(MUTEX_HELD(&flock_lock));
2960        if (lock->pvertex != -1) {
2961                ASSERT(lock->pvertex >= 0);
2962                pv = pgraph.proc[lock->pvertex];
2963                if (pv != NULL && PROC_SAME_OWNER(lock, pv)) {
2964                        return (pv);
2965                }
2966        }
2967        for (i = 0; i < pgraph.gcount; i++) {
2968                pv = pgraph.proc[i];
2969                if (pv != NULL && PROC_SAME_OWNER(lock, pv)) {
2970                        lock->pvertex = pv->index = i;
2971                        return (pv);
2972                }
2973        }
2974        pv = kmem_zalloc(sizeof (struct proc_vertex), KM_SLEEP);
2975        pv->pid = lock->l_flock.l_pid;
2976        pv->sysid = lock->l_flock.l_sysid;
2977        flk_proc_vertex_allocs++;
2978        if (pgraph.free != 0) {
2979                for (i = 0; i < pgraph.gcount; i++) {
2980                        if (pgraph.proc[i] == NULL) {
2981                                pgraph.proc[i] = pv;
2982                                lock->pvertex = pv->index = i;
```

```
2983                                pgraph.free--;
2984                                return (pv);
2985                        }
2986                }
2987        }
2988        palloc = kmem_zalloc((pgraph.gcount + PROC_CHUNK) *
2989            sizeof (proc_vertex_t *), KM_SLEEP);

2991        if (pgraph.proc) {
2992                bcopy(pgraph.proc, palloc,
2993                    pgraph.gcount * sizeof (proc_vertex_t *));

2995                kmem_free(pgraph.proc,
2996                    pgraph.gcount * sizeof (proc_vertex_t *));
2997        }
2998        pgraph.proc = palloc;
2999        pgraph.free += (PROC_CHUNK - 1);
3000        pv->index = lock->pvertex = pgraph.gcount;
3001        pgraph.gcount += PROC_CHUNK;
3002        pgraph.proc[pv->index] = pv;
3003        return (pv);
3004 }

3006 /*
3007  * Allocate a proc edge.
3008  */

3010 static proc_edge_t *
3011 flk_get_proc_edge()
3012 {
3013        proc_edge_t *pep;

3015        pep = kmem_zalloc(sizeof (proc_edge_t), KM_SLEEP);
3016        flk_proc_edge_allocs++;
3017        return (pep);
3018 }

3020 /*
3021  * Free the proc edge. Called whenever its reference count goes to zero.
3022  */

3024 static void
3025 flk_free_proc_edge(proc_edge_t *pep)
3026 {
3027        ASSERT(pep->refcount == 0);
3028        kmem_free((void *)pep, sizeof (proc_edge_t));
3029        flk_proc_edge_frees++;
3030 }

3032 /*
3033  * Color the graph explicitly done only when the mark value hits max value.
3034  */

3036 static void
3037 flk_proc_graph_uncolor()
3038 {
3039        int i;

3041        if (pgraph.mark == UINT_MAX) {
3042                for (i = 0; i < pgraph.gcount; i++)
3043                        if (pgraph.proc[i] != NULL) {
3044                                pgraph.proc[i]->atime = 0;
3045                                pgraph.proc[i]->dtime = 0;
3046                        }
3047                pgraph.mark = 1;
3048        } else {
```



```
3049                pgraph.mark++;
3050        }
3051 }

3053 /*
3054  * Release the proc vertex iff both there are no in edges and out edges
3055  */

3057 static void
3058 flk_proc_release(proc_vertex_t *proc)
3059 {
3060        ASSERT(MUTEX_HELD(&flock_lock));
3061        if (proc->edge == NULL && proc->incount == 0) {
3062                pgraph.proc[proc->index] = NULL;
3063                pgraph.free++;
3064                kmem_free(proc, sizeof (proc_vertex_t));
3065                flk_proc_vertex_frees++;
3066        }
3067 }

3069 /*
3070  * Updates process graph to reflect change in a lock_graph.
3071  * Note: We should call this function only after we have a correctly
3072  * recomputed lock graph. Otherwise we might miss a deadlock detection.
3073  * eg: in function flk_relation() we call this function after flk_recompute_
3074  * dependencies() otherwise if a process tries to lock a vnode hashed
3075  * into another graph it might sleep for ever.
3076  */

3078 static void
3079 flk_update_proc_graph(edge_t *ep, int delete)
3080 {
3081        proc_vertex_t *toproc, *fromproc;
3082        proc_edge_t *pep, *prevpep;

3084        mutex_enter(&flock_lock);
3085        toproc = flk_get_proc_vertex(ep->to_vertex);
3086        fromproc = flk_get_proc_vertex(ep->from_vertex);

3088        if (!delete)
3089                goto add;
3090        pep = prevpep = fromproc->edge;

3092        ASSERT(pep != NULL);
3093        while (pep != NULL) {
3094                if (pep->to_proc == toproc) {
3095                        ASSERT(pep->refcount > 0);
3096                        pep->refcount--;
3097                        if (pep->refcount == 0) {
3098                                if (pep == prevpep) {
3099                                        fromproc->edge = pep->next;
3100                                } else {
3101                                        prevpep->next = pep->next;
3102                                }
3103                                toproc->incount--;
3104                                flk_proc_release(toproc);
3105                                flk_free_proc_edge(pep);
3106                        }
3107                        break;
3108                }
3109                prevpep = pep;
3110                pep = pep->next;
3111        }
3112        flk_proc_release(fromproc);
3113        mutex_exit(&flock_lock);
3114        return;
```

```
3115 add:

3117         pep = fromproc->edge;

3119         while (pep != NULL) {
3120                 if (pep->to_proc == toproc) {
3121                         ASSERT(pep->refcount > 0);
3122                         pep->refcount++;
3123                         break;
3124                 }
3125                 pep = pep->next;
3126         }
3127         if (pep == NULL) {
3128                 pep = flk_get_proc_edge();
3129                 pep->to_proc = toproc;
3130                 pep->refcount = 1;
3131                 toproc->incount++;
3132                 pep->next = fromproc->edge;
3133                 fromproc->edge = pep;
3134         }
3135         mutex_exit(&flock_lock);
3136 }

3138 /*
3139  * Set the control status for lock manager requests.
3140  *
3141  */

3143 /*
3144  * PSARC case 1997/292
3145  *
3146  * Requires: "nlmid" must be >= 1 and <= clconf_maximum_nodeid().
3147  * Effects: Set the state of the NLM server identified by "nlmid"
3148  *   in the NLM registry to state "nlm_state."
3149  *   Raises exception no_such_nlm if "nlmid" doesn't identify a known
3150  *   NLM server to this LLM.
3151  *   Note that when this routine is called with NLM_SHUTTING_DOWN there
3152  *   may be locks requests that have gotten started but not finished.  In
3153  *   particular, there may be blocking requests that are in the callback code
3154  *   before sleeping (so they're not holding the lock for the graph).  If
3155  *   such a thread reacquires the graph's lock (to go to sleep) after
3156  *   NLM state in the NLM registry  is set to a non-up value,
3157  *   it will notice the status and bail out.  If the request gets
3158  *   granted before the thread can check the NLM registry, let it
3159  *   continue normally.  It will get flushed when we are called with NLM_DOWN.
3160  *
3161  * Modifies: nlm_reg_obj (global)
3162  * Arguments:
3163  *    nlmid     (IN):    id uniquely identifying an NLM server
3164  *    nlm_state (IN):    NLM server state to change "nlmid" to
3165  */
3166 void
3167 cl_flk_set_nlm_status(int nlmid, flk_nlm_status_t nlm_state)
3168 {
3169         /*
3170          * Check to see if node is booted as a cluster. If not, return.
3171          */
3172         if ((cluster_bootflags & CLUSTER_BOOTED) == 0) {
3173                 return;
3174         }

3176         /*
3177          * Check for development/debugging.  It is possible to boot a node
3178          * in non-cluster mode, and then run a special script, currently
3179          * available only to developers, to bring up the node as part of a
3180          * cluster.  The problem is that running such a script does not
```



```
3181          * result in the routine flk_init() being called and hence global array
3182          * nlm_reg_status is NULL.  The NLM thinks it's in cluster mode,
3183          * but the LLM needs to do an additional check to see if the global
3184          * array has been created or not. If nlm_reg_status is NULL, then
3185          * return, else continue.
3186          */
3187         if (nlm_reg_status == NULL) {
3188                 return;
3189         }

3191         ASSERT(nlmid <= nlm_status_size && nlmid >= 0);
3192         mutex_enter(&nlm_reg_lock);

3194         if (FLK_REGISTRY_IS_NLM_UNKNOWN(nlm_reg_status, nlmid)) {
3195                 /*
3196                  * If the NLM server "nlmid" is unknown in the NLM registry,
3197                  * add it to the registry in the nlm shutting down state.
3198                  */
3199                 FLK_REGISTRY_CHANGE_NLM_STATE(nlm_reg_status, nlmid,
3200                     FLK_NLM_SHUTTING_DOWN);
3201         } else {
3202                 /*
3203                  * Change the state of the NLM server identified by "nlmid"
3204                  * in the NLM registry to the argument "nlm_state."
3205                  */
3206                 FLK_REGISTRY_CHANGE_NLM_STATE(nlm_reg_status, nlmid,
3207                     nlm_state);
3208         }

3210         /*
3211          *  The reason we must register the NLM server that is shutting down
3212          *  with an LLM that doesn't already know about it (never sent a lock
3213          *  request) is to handle correctly a race between shutdown and a new
3214          *  lock request.  Suppose that a shutdown request from the NLM server
3215          *  invokes this routine at the LLM, and a thread is spawned to
3216          *  service the request. Now suppose a new lock request is in
3217          *  progress and has already passed the first line of defense in
3218          *  reclock(), which denies new locks requests from NLM servers
3219          *  that are not in the NLM_UP state.  After the current routine
3220          *  is invoked for both phases of shutdown, the routine will return,
3221          *  having done nothing, and the lock request will proceed and
3222          *  probably be granted.  The problem is that the shutdown was ignored
3223          *  by the lock request because there was no record of that NLM server
3224          *  shutting down.   We will be in the peculiar position of thinking
3225          *  that we've shutdown the NLM server and all locks at all LLMs have
3226          *  been discarded, but in fact there's still one lock held.
3227          *  The solution is to record the existence of NLM server and change
3228          *  its state immediately to NLM_SHUTTING_DOWN.  The lock request in
3229          *  progress may proceed because the next phase NLM_DOWN will catch
3230          *  this lock and discard it.
3231          */
3232         mutex_exit(&nlm_reg_lock);

3234         switch (nlm_state) {
3235         case FLK_NLM_UP:
3236                 /*
3237                  * Change the NLM state of all locks still held on behalf of
3238                  * the NLM server identified by "nlmid" to NLM_UP.
3239                  */
3240                 cl_flk_change_nlm_state_all_locks(nlmid, FLK_NLM_UP);
3241                 break;

3243         case FLK_NLM_SHUTTING_DOWN:
3244                 /*
3245                  * Wake up all sleeping locks for the NLM server identified
3246                  * by "nlmid." Note that eventually all woken threads will
```

```
3247                 * have their lock requests cancelled and descriptors
3248                 * removed from the sleeping lock list.  Note that the NLM
3249                 * server state associated with each lock descriptor is
3250                 * changed to FLK_NLM_SHUTTING_DOWN.
3251                 */
3252                cl_flk_wakeup_sleeping_nlm_locks(nlmid);
3253                break;

3255        case FLK_NLM_DOWN:
3256                /*
3257                 * Discard all active, granted locks for this NLM server
3258                 * identified by "nlmid."
3259                 */
3260                cl_flk_unlock_nlm_granted(nlmid);
3261                break;

3263        default:
3264                panic("cl_set_nlm_status: bad status (%d)", nlm_state);
3265        }
3266 }

3268 /*
3269  * Set the control status for lock manager requests.
3270  *
3271  * Note that when this routine is called with FLK_WAKEUP_SLEEPERS, there
3272  * may be locks requests that have gotten started but not finished.  In
3273  * particular, there may be blocking requests that are in the callback code
3274  * before sleeping (so they're not holding the lock for the graph).  If
3275  * such a thread reacquires the graph's lock (to go to sleep) after
3276  * flk_lockmgr_status is set to a non-up value, it will notice the status
3277  * and bail out.  If the request gets granted before the thread can check
3278  * flk_lockmgr_status, let it continue normally.  It will get flushed when
3279  * we are called with FLK_LOCKMGR_DOWN.
3280  */

3282 void
3283 flk_set_lockmgr_status(flk_lockmgr_status_t status)
3284 {
3285        int i;
3286        graph_t *gp;
3287        struct flock_globals *fg;

3289        fg = flk_get_globals();
3290        ASSERT(fg != NULL);

3292        mutex_enter(&flock_lock);
3293        fg->flk_lockmgr_status = status;
3294        mutex_exit(&flock_lock);

3296        /*
3297         * If the lock manager is coming back up, all that's needed is to
3298         * propagate this information to the graphs.  If the lock manager
3299         * is going down, additional action is required, and each graph's
3300         * copy of the state is updated atomically with this other action.
3301         */
3302        switch (status) {
3303        case FLK_LOCKMGR_UP:
3304                for (i = 0; i < HASH_SIZE; i++) {
3305                        mutex_enter(&flock_lock);
3306                        gp = lock_graph[i];
3307                        mutex_exit(&flock_lock);
3308                        if (gp == NULL)
3309                                continue;
3310                        mutex_enter(&gp->gp_mutex);
3311                        fg->lockmgr_status[i] = status;
3312                        mutex_exit(&gp->gp_mutex);
```



```
3313                }
3314                break;
3315        case FLK_WAKEUP_SLEEPERS:
3316                wakeup_sleeping_lockmgr_locks(fg);
3317                break;
3318        case FLK_LOCKMGR_DOWN:
3319                unlock_lockmgr_granted(fg);
3320                break;
3321        default:
3322                panic("flk_set_lockmgr_status: bad status (%d)", status);
3323                break;
3324        }
3325 }

3327 /*
3328  * This routine returns all the locks that are active or sleeping and are
3329  * associated with a particular set of identifiers.  If lock_state != 0, then
3330  * only locks that match the lock_state are returned. If lock_state == 0, then
3331  * all locks are returned. If pid == NOPID, the pid is ignored.  If
3332  * use_sysid is FALSE, then the sysid is ignored.  If vp is NULL, then the
3333  * vnode pointer is ignored.
3334  *
3335  * A list containing the vnode pointer and an flock structure
3336  * describing the lock is returned.  Each element in the list is
3337  * dynamically allocated and must be freed by the caller.  The
3338  * last item in the list is denoted by a NULL value in the ll_next
3339  * field.
3340  *
3341  * The vnode pointers returned are held.  The caller is responsible
3342  * for releasing these.  Note that the returned list is only a snapshot of
3343  * the current lock information, and that it is a snapshot of a moving
3344  * target (only one graph is locked at a time).
3345  */

3347 locklist_t *
3348 get_lock_list(int list_type, int lock_state, int sysid, boolean_t use_sysid,
3349                pid_t pid, const vnode_t *vp, zoneid_t zoneid)
3350 {
3351        lock_descriptor_t       *lock;
3352        lock_descriptor_t       *graph_head;
3353        locklist_t              listhead;
3354        locklist_t              *llheadp;
3355        locklist_t              *llp;
3356        locklist_t              *lltp;
3357        graph_t                 *gp;
3358        int                     i;
3359        int                     first_index; /* graph index */
3360        int                     num_indexes; /* graph index */

3362        ASSERT((list_type == FLK_ACTIVE_STATE) ||
3363            (list_type == FLK_SLEEPING_STATE));

3365        /*
3366         * Get a pointer to something to use as a list head while building
3367         * the rest of the list.
3368         */
3369        llheadp = &listhead;
3370        lltp = llheadp;
3371        llheadp->ll_next = (locklist_t *)NULL;

3373        /* Figure out which graphs we want to look at. */
3374        if (vp == NULL) {
3375                first_index = 0;
3376                num_indexes = HASH_SIZE;
3377        } else {
3378                first_index = HASH_INDEX(vp);
```

```
3379                    num_indexes = 1;
3380            }
3381
3382            for (i = first_index; i < first_index + num_indexes; i++) {
3383                    mutex_enter(&flock_lock);
3384                    gp = lock_graph[i];
3385                    mutex_exit(&flock_lock);
3386                    if (gp == NULL) {
3387                            continue;
3388                    }
3389
3390                    mutex_enter(&gp->gp_mutex);
3391                    graph_head = (list_type == FLK_ACTIVE_STATE) ?
3392                        ACTIVE_HEAD(gp) : SLEEPING_HEAD(gp);
3393                    for (lock = graph_head->l_next;
3394                        lock != graph_head;
3395                        lock = lock->l_next) {
3396                            if (use_sysid && lock->l_flock.l_sysid != sysid)
3397                                    continue;
3398                            if (pid != NOPID && lock->l_flock.l_pid != pid)
3399                                    continue;
3400                            if (vp != NULL && lock->l_vnode != vp)
3401                                    continue;
3402                            if (lock_state && !(lock_state & lock->l_state))
3403                                    continue;
3404                            if (zoneid != lock->l_zoneid && zoneid != ALL_ZONES)
3405                                    continue;
3406                            /*
3407                             * A matching lock was found.  Allocate
3408                             * space for a new locklist entry and fill
3409                             * it in.
3410                             */
3411                            llp = kmem_alloc(sizeof (locklist_t), KM_SLEEP);
3412                            lltp->ll_next = llp;
3413                            VN_HOLD(lock->l_vnode);
3414                            llp->ll_vp = lock->l_vnode;
3415                            create_flock(lock, &(llp->ll_flock));
3416                            llp->ll_next = (locklist_t *)NULL;
3417                            lltp = llp;
3418                    }
3419                    mutex_exit(&gp->gp_mutex);
3420            }
3421
3422            llp = llheadp->ll_next;
3423            return (llp);
3424    }
3425
3426    /*
3427     * These two functions are simply interfaces to get_lock_list.  They return
3428     * a list of sleeping or active locks for the given sysid and pid.  See
3429     * get_lock_list for details.
3430     *
3431     * In either case we don't particularly care to specify the zone of interest;
3432     * the sysid-space is global across zones, so the sysid will map to exactly one
3433     * zone, and we'll return information for that zone.
3434     */
3435
3436    locklist_t *
3437    flk_get_sleeping_locks(int sysid, pid_t pid)
3438    {
3439            return (get_lock_list(FLK_SLEEPING_STATE, 0, sysid, B_TRUE, pid, NULL,
3440                ALL_ZONES));
3441    }
3442
3443    locklist_t *
3444    flk_get_active_locks(int sysid, pid_t pid)
```



```
3445    {
3446            return (get_lock_list(FLK_ACTIVE_STATE, 0, sysid, B_TRUE, pid, NULL,
3447                ALL_ZONES));
3448    }
3449
3450    /*
3451     * Another interface to get_lock_list.  This one returns all the active
3452     * locks for a given vnode.  Again, see get_lock_list for details.
3453     *
3454     * We don't need to specify which zone's locks we're interested in.  The matter
3455     * would only be interesting if the vnode belonged to NFS, and NFS vnodes can't
3456     * be used by multiple zones, so the list of locks will all be from the right
3457     * zone.
3458     */
3459
3460    locklist_t *
3461    flk_active_locks_for_vp(const vnode_t *vp)
3462    {
3463            return (get_lock_list(FLK_ACTIVE_STATE, 0, 0, B_FALSE, NOPID, vp,
3464                ALL_ZONES));
3465    }
3466
3467    /*
3468     * Another interface to get_lock_list.  This one returns all the active
3469     * nbmand locks for a given vnode.  Again, see get_lock_list for details.
3470     *
3471     * See the comment for flk_active_locks_for_vp() for why we don't care to
3472     * specify the particular zone of interest.
3473     */
3474    locklist_t *
3475    flk_active_nbmand_locks_for_vp(const vnode_t *vp)
3476    {
3477            return (get_lock_list(FLK_ACTIVE_STATE, NBMAND_LOCK, 0, B_FALSE,
3478                NOPID, vp, ALL_ZONES));
3479    }
3480
3481    /*
3482     * Another interface to get_lock_list.  This one returns all the active
3483     * nbmand locks for a given pid.  Again, see get_lock_list for details.
3484     *
3485     * The zone doesn't need to be specified here; the locks held by a
3486     * particular process will either be local (ie, non-NFS) or from the zone
3487     * the process is executing in.  This is because other parts of the system
3488     * ensure that an NFS vnode can't be used in a zone other than that in
3489     * which it was opened.
3490     */
3491    locklist_t *
3492    flk_active_nbmand_locks(pid_t pid)
3493    {
3494            return (get_lock_list(FLK_ACTIVE_STATE, NBMAND_LOCK, 0, B_FALSE,
3495                pid, NULL, ALL_ZONES));
3496    }
3497
3498    /*
3499     * Free up all entries in the locklist.
3500     */
3501    void
3502    flk_free_locklist(locklist_t *llp)
3503    {
3504            locklist_t *next_llp;
3505
3506            while (llp) {
3507                    next_llp = llp->ll_next;
3508                    VN_RELE(llp->ll_vp);
3509                    kmem_free(llp, sizeof (*llp));
3510                    llp = next_llp;
```

```
3511         }
3512 }

3514 static void
3515 cl_flk_change_nlm_state_all_locks(int nlmid, flk_nlm_status_t nlm_state)
3516 {
3517         /*
3518          * For each graph "lg" in the hash table lock_graph do
3519          * a.  Get the list of sleeping locks
3520          * b.  For each lock descriptor in the list do
3521          *      i.   If the requested lock is an NLM server request AND
3522          *           the nlmid is the same as the routine argument then
3523          *           change the lock descriptor's state field to
3524          *           "nlm_state."
3525          * c.  Get the list of active locks
3526          * d.  For each lock descriptor in the list do
3527          *      i.   If the requested lock is an NLM server request AND
3528          *           the nlmid is the same as the routine argument then
3529          *           change the lock descriptor's state field to
3530          *           "nlm_state."
3531          */

3533         int                     i;
3534         graph_t                 *gp;                    /* lock graph */
3535         lock_descriptor_t       *lock;                  /* lock */
3536         lock_descriptor_t       *nlock = NULL;          /* next lock */
3537         int                     lock_nlmid;

3539         for (i = 0; i < HASH_SIZE; i++) {
3540                 mutex_enter(&flock_lock);
3541                 gp = lock_graph[i];
3542                 mutex_exit(&flock_lock);
3543                 if (gp == NULL) {
3544                         continue;
3545                 }

3547                 /* Get list of sleeping locks in current lock graph. */
3548                 mutex_enter(&gp->gp_mutex);
3549                 for (lock = SLEEPING_HEAD(gp)->l_next;
3550                     lock != SLEEPING_HEAD(gp);
3551                     lock = nlock) {
3552                         nlock = lock->l_next;
3553                         /* get NLM id */
3554                         lock_nlmid = GETNLMID(lock->l_flock.l_sysid);

3556                         /*
3557                          * If NLM server request AND nlmid of lock matches
3558                          * nlmid of argument, then set the NLM state of the
3559                          * lock to "nlm_state."
3560                          */
3561                         if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
3562                                 SET_NLM_STATE(lock, nlm_state);
3563                         }
3564                 }

3566                 /* Get list of active locks in current lock graph. */
3567                 for (lock = ACTIVE_HEAD(gp)->l_next;
3568                     lock != ACTIVE_HEAD(gp);
3569                     lock = nlock) {
3570                         nlock = lock->l_next;
3571                         /* get NLM id */
3572                         lock_nlmid = GETNLMID(lock->l_flock.l_sysid);

3574                         /*
3575                          * If NLM server request AND nlmid of lock matches
3576                          * nlmid of argument, then set the NLM state of the
```



```
3577                          * lock to "nlm_state."
3578                          */
3579                         if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
3580                                 ASSERT(IS_ACTIVE(lock));
3581                                 SET_NLM_STATE(lock, nlm_state);
3582                         }
3583                 }
3584                 mutex_exit(&gp->gp_mutex);
3585         }
3586 }

3588 /*
3589  * Requires: "nlmid" >= 1 and <= clconf_maximum_nodeid().
3590  * Effects: Find all sleeping lock manager requests _only_ for the NLM server
3591  *   identified by "nlmid." Poke those lock requests.
3592  */
3593 static void
3594 cl_flk_wakeup_sleeping_nlm_locks(int nlmid)
3595 {
3596         lock_descriptor_t *lock;
3597         lock_descriptor_t *nlock = NULL; /* next lock */
3598         int i;
3599         graph_t *gp;
3600         int     lock_nlmid;

3602         for (i = 0; i < HASH_SIZE; i++) {
3603                 mutex_enter(&flock_lock);
3604                 gp = lock_graph[i];
3605                 mutex_exit(&flock_lock);
3606                 if (gp == NULL) {
3607                         continue;
3608                 }

3610                 mutex_enter(&gp->gp_mutex);
3611                 for (lock = SLEEPING_HEAD(gp)->l_next;
3612                     lock != SLEEPING_HEAD(gp);
3613                     lock = nlock) {
3614                         nlock = lock->l_next;
3615                         /*
3616                          * If NLM server request _and_ nlmid of lock matches
3617                          * nlmid of argument, then set the NLM state of the
3618                          * lock to NLM_SHUTTING_DOWN, and wake up sleeping
3619                          * request.
3620                          */
3621                         if (IS_LOCKMGR(lock)) {
3622                                 /* get NLM id */
3623                                 lock_nlmid =
3624                                     GETNLMID(lock->l_flock.l_sysid);
3625                                 if (nlmid == lock_nlmid) {
3626                                         SET_NLM_STATE(lock,
3627                                             FLK_NLM_SHUTTING_DOWN);
3628                                         INTERRUPT_WAKEUP(lock);
3629                                 }
3630                         }
3631                 }
3632                 mutex_exit(&gp->gp_mutex);
3633         }
3634 }

3636 /*
3637  * Requires: "nlmid" >= 1 and <= clconf_maximum_nodeid()
3638  * Effects:  Find all active (granted) lock manager locks _only_ for the
3639  *   NLM server identified by "nlmid" and release them.
3640  */
3641 static void
3642 cl_flk_unlock_nlm_granted(int nlmid)
```

```
3643 {
3644         lock_descriptor_t *lock;
3645         lock_descriptor_t *nlock = NULL; /* next lock */
3646         int i;
3647         graph_t *gp;
3648         int     lock_nlmid;

3650         for (i = 0; i < HASH_SIZE; i++) {
3651                 mutex_enter(&flock_lock);
3652                 gp = lock_graph[i];
3653                 mutex_exit(&flock_lock);
3654                 if (gp == NULL) {
3655                         continue;
3656                 }

3658                 mutex_enter(&gp->gp_mutex);
3659                 for (lock = ACTIVE_HEAD(gp)->l_next;
3660                     lock != ACTIVE_HEAD(gp);
3661                     lock = nlock) {
3662                         nlock = lock->l_next;
3663                         ASSERT(IS_ACTIVE(lock));

3665                         /*
3666                          * If it's an  NLM server request _and_ nlmid of
3667                          * the lock matches nlmid of argument, then
3668                          * remove the active lock the list, wakup blocked
3669                          * threads, and free the storage for the lock.
3670                          * Note that there's no need to mark the NLM state
3671                          * of this lock to NLM_DOWN because the lock will
3672                          * be deleted anyway and its storage freed.
3673                          */
3674                         if (IS_LOCKMGR(lock)) {
3675                                 /* get NLM id */
3676                                 lock_nlmid = GETNLMID(lock->l_flock.l_sysid);
3677                                 if (nlmid == lock_nlmid) {
3678                                         flk_delete_active_lock(lock, 0);
3679                                         flk_wakeup(lock, 1);
3680                                         flk_free_lock(lock);
3681                                 }
3682                         }
3683                 }
3684                 mutex_exit(&gp->gp_mutex);
3685         }
3686 }

3688 /*
3689  * Find all sleeping lock manager requests and poke them.
3690  */
3691 static void
3692 wakeup_sleeping_lockmgr_locks(struct flock_globals *fg)
3693 {
3694         lock_descriptor_t *lock;
3695         lock_descriptor_t *nlock = NULL; /* next lock */
3696         int i;
3697         graph_t *gp;
3698         zoneid_t zoneid = getzoneid();

3700         for (i = 0; i < HASH_SIZE; i++) {
3701                 mutex_enter(&flock_lock);
3702                 gp = lock_graph[i];
3703                 mutex_exit(&flock_lock);
3704                 if (gp == NULL) {
3705                         continue;
3706                 }

3708                 mutex_enter(&gp->gp_mutex);
```



```
3709                 fg->lockmgr_status[i] = FLK_WAKEUP_SLEEPERS;
3710                 for (lock = SLEEPING_HEAD(gp)->l_next;
3711                     lock != SLEEPING_HEAD(gp);
3712                     lock = nlock) {
3713                         nlock = lock->l_next;
3714                         if (IS_LOCKMGR(lock) && lock->l_zoneid == zoneid) {
3715                                 INTERRUPT_WAKEUP(lock);
3716                         }
3717                 }
3718                 mutex_exit(&gp->gp_mutex);
3719         }
3720 }


3723 /*
3724  * Find all active (granted) lock manager locks and release them.
3725  */
3726 static void
3727 unlock_lockmgr_granted(struct flock_globals *fg)
3728 {
3729         lock_descriptor_t *lock;
3730         lock_descriptor_t *nlock = NULL; /* next lock */
3731         int i;
3732         graph_t *gp;
3733         zoneid_t zoneid = getzoneid();

3735         for (i = 0; i < HASH_SIZE; i++) {
3736                 mutex_enter(&flock_lock);
3737                 gp = lock_graph[i];
3738                 mutex_exit(&flock_lock);
3739                 if (gp == NULL) {
3740                         continue;
3741                 }

3743                 mutex_enter(&gp->gp_mutex);
3744                 fg->lockmgr_status[i] = FLK_LOCKMGR_DOWN;
3745                 for (lock = ACTIVE_HEAD(gp)->l_next;
3746                     lock != ACTIVE_HEAD(gp);
3747                     lock = nlock) {
3748                         nlock = lock->l_next;
3749                         if (IS_LOCKMGR(lock) && lock->l_zoneid == zoneid) {
3750                                 ASSERT(IS_ACTIVE(lock));
3751                                 flk_delete_active_lock(lock, 0);
3752                                 flk_wakeup(lock, 1);
3753                                 flk_free_lock(lock);
3754                         }
3755                 }
3756                 mutex_exit(&gp->gp_mutex);
3757         }
3758 }


3761 /*
3762  * Wait until a lock is granted, cancelled, or interrupted.
3763  */

3765 static void
3766 wait_for_lock(lock_descriptor_t *request)
3767 {
3768         graph_t *gp = request->l_graph;

3770         ASSERT(MUTEX_HELD(&gp->gp_mutex));

3772         while (!(IS_GRANTED(request)) && !(IS_CANCELLED(request)) &&
3773             !(IS_INTERRUPTED(request))) {
3774                 if (!cv_wait_sig(&request->l_cv, &gp->gp_mutex)) {
```

```
3775                        flk_set_state(request, FLK_INTERRUPTED_STATE);
3776                        request->l_state |= INTERRUPTED_LOCK;
3777                }
3778        }
3779 }

3781 /*
3782  * Create an flock structure from the existing lock information
3783  *
3784  * This routine is used to create flock structures for the lock manager
3785  * to use in a reclaim request.  Since the lock was originated on this
3786  * host, it must be conforming to UNIX semantics, so no checking is
3787  * done to make sure it falls within the lower half of the 32-bit range.
3788  */

3790 static void
3791 create_flock(lock_descriptor_t *lp, flock64_t *flp)
3792 {
3793        ASSERT(lp->l_end == MAX_U_OFFSET_T || lp->l_end <= MAXEND);
3794        ASSERT(lp->l_end >= lp->l_start);

3796        flp->l_type = lp->l_type;
3797        flp->l_whence = 0;
3798        flp->l_start = lp->l_start;
3799        flp->l_len = (lp->l_end == MAX_U_OFFSET_T) ? 0 :
3800            (lp->l_end - lp->l_start + 1);
3801        flp->l_sysid = lp->l_flock.l_sysid;
3802        flp->l_pid = lp->l_flock.l_pid;
3803 }

3805 /*
3806  * Convert flock_t data describing a lock range into unsigned long starting
3807  * and ending points, which are put into lock_request.  Returns 0 or an
3808  * errno value.
3809  * Large Files: max is passed by the caller and we return EOVERFLOW
3810  * as defined by LFS API.
3811  */

3813 int
3814 flk_convert_lock_data(vnode_t *vp, flock64_t *flp,
3815     u_offset_t *start, u_offset_t *end, offset_t offset)
3816 {
3817        struct vattr    vattr;
3818        int     error;

3820        /*
3821         * Determine the starting point of the request
3822         */
3823        switch (flp->l_whence) {
3824        case 0:         /* SEEK_SET */
3825                *start = (u_offset_t)flp->l_start;
3826                break;
3827        case 1:         /* SEEK_CUR */
3828                *start = (u_offset_t)(flp->l_start + offset);
3829                break;
3830        case 2:         /* SEEK_END */
3831                vattr.va_mask = AT_SIZE;
3832                if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
3833                        return (error);
3834                *start = (u_offset_t)(flp->l_start + vattr.va_size);
3835                break;
3836        default:
3837                return (EINVAL);
3838        }

3840        /*
```



```
3841         * Determine the range covered by the request.
3842         */
3843        if (flp->l_len == 0)
3844                *end = MAX_U_OFFSET_T;
3845        else if ((offset_t)flp->l_len > 0) {
3846                *end = (u_offset_t)(*start + (flp->l_len - 1));
3847        } else {
3848                /*
3849                 * Negative length; why do we even allow this ?
3850                 * Because this allows easy specification of
3851                 * the last n bytes of the file.
3852                 */
3853                *end = *start;
3854                *start += (u_offset_t)flp->l_len;
3855                (*start)++;
3856        }
3857        return (0);
3858 }

3860 /*
3861  * Check the validity of lock data.  This can used by the NFS
3862  * frlock routines to check data before contacting the server.  The
3863  * server must support semantics that aren't as restrictive as
3864  * the UNIX API, so the NFS client is required to check.
3865  * The maximum is now passed in by the caller.
3866  */

3868 int
3869 flk_check_lock_data(u_offset_t start, u_offset_t end, offset_t max)
3870 {
3871        /*
3872         * The end (length) for local locking should never be greater
3873         * than MAXEND. However, the representation for
3874         * the entire file is MAX_U_OFFSET_T.
3875         */
3876        if ((start > max) ||
3877            ((end > max) && (end != MAX_U_OFFSET_T))) {
3878                return (EINVAL);
3879        }
3880        if (start > end) {
3881                return (EINVAL);
3882        }
3883        return (0);
3884 }

3886 /*
3887  * Fill in request->l_flock with information about the lock blocking the
3888  * request.  The complexity here is that lock manager requests are allowed
3889  * to see into the upper part of the 32-bit address range, whereas local
3890  * requests are only allowed to see signed values.
3891  *
3892  * What should be done when "blocker" is a lock manager lock that uses the
3893  * upper portion of the 32-bit range, but "request" is local?  Since the
3894  * request has already been determined to have been blocked by the blocker,
3895  * at least some portion of "blocker" must be in the range of the request,
3896  * or the request extends to the end of file.  For the first case, the
3897  * portion in the lower range is returned with the indication that it goes
3898  * "to EOF."  For the second case, the last byte of the lower range is
3899  * returned with the indication that it goes "to EOF."
3900  */

3902 static void
3903 report_blocker(lock_descriptor_t *blocker, lock_descriptor_t *request)
3904 {
3905        flock64_t *flrp;                        /* l_flock portion of request */
```

```
3907         ASSERT(blocker != NULL);

3909         flrp = &request->l_flock;
3910         flrp->l_whence = 0;
3911         flrp->l_type = blocker->l_type;
3912         flrp->l_pid = blocker->l_flock.l_pid;
3913         flrp->l_sysid = blocker->l_flock.l_sysid;

3915         if (IS_LOCKMGR(request)) {
3916                 flrp->l_start = blocker->l_start;
3917                 if (blocker->l_end == MAX_U_OFFSET_T)
3918                         flrp->l_len = 0;
3919                 else
3920                         flrp->l_len = blocker->l_end - blocker->l_start + 1;
3921         } else {
3922                 if (blocker->l_start > MAXEND) {
3923                         flrp->l_start = MAXEND;
3924                         flrp->l_len = 0;
3925                 } else {
3926                         flrp->l_start = blocker->l_start;
3927                         if (blocker->l_end == MAX_U_OFFSET_T)
3928                                 flrp->l_len = 0;
3929                         else
3930                                 flrp->l_len = blocker->l_end -
3931                                     blocker->l_start + 1;
3932                 }
3933         }
3934 }

3936 /*
3937  * PSARC case 1997/292
3938  */
3939 /*
3940  * This is the public routine exported by flock.h.
3941  */
3942 void
3943 cl_flk_change_nlm_state_to_unknown(int nlmid)
3944 {
3945         /*
3946          * Check to see if node is booted as a cluster. If not, return.
3947          */
3948         if ((cluster_bootflags & CLUSTER_BOOTED) == 0) {
3949                 return;
3950         }

3952         /*
3953          * See comment in cl_flk_set_nlm_status().
3954          */
3955         if (nlm_reg_status == NULL) {
3956                 return;
3957         }

3959         /*
3960          * protect NLM registry state with a mutex.
3961          */
3962         ASSERT(nlmid <= nlm_status_size && nlmid >= 0);
3963         mutex_enter(&nlm_reg_lock);
3964         FLK_REGISTRY_CHANGE_NLM_STATE(nlm_reg_status, nlmid, FLK_NLM_UNKNOWN);
3965         mutex_exit(&nlm_reg_lock);
3966 }

3968 /*
3969  * Return non-zero if the given I/O request conflicts with an active NBMAND
3970  * lock.
3971  * If svmand is non-zero, it means look at all active locks, not just NBMAND
3972  * locks.
```



```
3973  */

3975 int
3976 nbl_lock_conflict(vnode_t *vp, nbl_op_t op, u_offset_t offset,
3977                 ssize_t length, int svmand, caller_context_t *ct)
3978 {
3979         int conflict = 0;
3980         graph_t                 *gp;
3981         lock_descriptor_t       *lock;
3982         pid_t pid;
3983         int sysid;

3985         if (ct == NULL) {
3986                 pid = curproc->p_pid;
3987                 sysid = 0;
3988         } else {
3989                 pid = ct->cc_pid;
3990                 sysid = ct->cc_sysid;
3991         }

3993         mutex_enter(&flock_lock);
3994         gp = lock_graph[HASH_INDEX(vp)];
3995         mutex_exit(&flock_lock);
3996         if (gp == NULL)
3997                 return (0);

3999         mutex_enter(&gp->gp_mutex);
4000         SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

4002         for (; lock && lock->l_vnode == vp; lock = lock->l_next) {
4003                 if ((svmand || (lock->l_state & NBMAND_LOCK)) &&
4004                     (lock->l_flock.l_sysid != sysid ||
4005                     lock->l_flock.l_pid != pid) &&
4006                     lock_blocks_io(op, offset, length,
4007                     lock->l_type, lock->l_start, lock->l_end)) {
4008                         conflict = 1;
4009                         break;
4010                 }
4011         }
4012         mutex_exit(&gp->gp_mutex);

4014         return (conflict);
4015 }

4017 /*
4018  * Return non-zero if the given I/O request conflicts with the given lock.
4019  */

4021 static int
4022 lock_blocks_io(nbl_op_t op, u_offset_t offset, ssize_t length,
4023             int lock_type, u_offset_t lock_start, u_offset_t lock_end)
4024 {
4025         ASSERT(op == NBL_READ || op == NBL_WRITE || op == NBL_READWRITE);
4026         ASSERT(lock_type == F_RDLCK || lock_type == F_WRLCK);

4028         if (op == NBL_READ && lock_type == F_RDLCK)
4029                 return (0);

4031         if (offset <= lock_start && lock_start < offset + length)
4032                 return (1);
4033         if (lock_start <= offset && offset <= lock_end)
4034                 return (1);

4036         return (0);
4037 }
```

```
#ifdef DEBUG
static void
check_active_locks(graph_t *gp)
{
        lock_descriptor_t *lock, *lock1;
        edge_t  *ep;

        for (lock = ACTIVE_HEAD(gp)->l_next; lock != ACTIVE_HEAD(gp);
            lock = lock->l_next) {
                ASSERT(IS_ACTIVE(lock));
                ASSERT(NOT_BLOCKED(lock));
                ASSERT(!IS_BARRIER(lock));

                ep = FIRST_IN(lock);

                while (ep != HEAD(lock)) {
                        ASSERT(IS_SLEEPING(ep->from_vertex));
                        ASSERT(!NOT_BLOCKED(ep->from_vertex));
                        ep = NEXT_IN(ep);
                }

                for (lock1 = lock->l_next; lock1 != ACTIVE_HEAD(gp);
                    lock1 = lock1->l_next) {
                        if (lock1->l_vnode == lock->l_vnode) {
                        if (BLOCKS(lock1, lock)) {
                                cmn_err(CE_PANIC,
                                    "active lock %p blocks %p",
                                    (void *)lock1, (void *)lock);
                        } else if (BLOCKS(lock, lock1)) {
                                cmn_err(CE_PANIC,
                                    "active lock %p blocks %p",
                                    (void *)lock, (void *)lock1);
                        }
                        }
                }
        }
}

/*
 * Effect: This functions checks to see if the transition from 'old_state' to
 *      'new_state' is a valid one.  It returns 0 if the transition is valid
 *      and 1 if it is not.
 *      For a map of valid transitions, see sys/flock_impl.h
 */
static int
check_lock_transition(int old_state, int new_state)
{
        switch (old_state) {
        case FLK_INITIAL_STATE:
                if ((new_state == FLK_START_STATE) ||
                    (new_state == FLK_SLEEPING_STATE) ||
                    (new_state == FLK_ACTIVE_STATE) ||
                    (new_state == FLK_DEAD_STATE)) {
                        return (0);
                } else {
                        return (1);
                }
        case FLK_START_STATE:
                if ((new_state == FLK_ACTIVE_STATE) ||
                    (new_state == FLK_DEAD_STATE)) {
                        return (0);
                } else {
                        return (1);
                }
        case FLK_ACTIVE_STATE:
                if (new_state == FLK_DEAD_STATE) {
```



```
                        return (0);
                } else {
                        return (1);
                }
        case FLK_SLEEPING_STATE:
                if ((new_state == FLK_GRANTED_STATE) ||
                    (new_state == FLK_INTERRUPTED_STATE) ||
                    (new_state == FLK_CANCELLED_STATE)) {
                        return (0);
                } else {
                        return (1);
                }
        case FLK_GRANTED_STATE:
                if ((new_state == FLK_START_STATE) ||
                    (new_state == FLK_INTERRUPTED_STATE) ||
                    (new_state == FLK_CANCELLED_STATE)) {
                        return (0);
                } else {
                        return (1);
                }
        case FLK_CANCELLED_STATE:
                if ((new_state == FLK_INTERRUPTED_STATE) ||
                    (new_state == FLK_DEAD_STATE)) {
                        return (0);
                } else {
                        return (1);
                }
        case FLK_INTERRUPTED_STATE:
                if (new_state == FLK_DEAD_STATE) {
                        return (0);
                } else {
                        return (1);
                }
        case FLK_DEAD_STATE:
                /* May be set more than once */
                if (new_state == FLK_DEAD_STATE) {
                        return (0);
                } else {
                        return (1);
                }
        default:
                return (1);
        }
}

static void
check_sleeping_locks(graph_t *gp)
{
        lock_descriptor_t *lock1, *lock2;
        edge_t *ep;
        for (lock1 = SLEEPING_HEAD(gp)->l_next; lock1 != SLEEPING_HEAD(gp);
            lock1 = lock1->l_next) {
                                ASSERT(!IS_BARRIER(lock1));
        for (lock2 = lock1->l_next; lock2 != SLEEPING_HEAD(gp);
            lock2 = lock2->l_next) {
                if (lock1->l_vnode == lock2->l_vnode) {
                        if (BLOCKS(lock2, lock1)) {
                                ASSERT(!IS_GRANTED(lock1));
                                ASSERT(!NOT_BLOCKED(lock1));
                                path(lock1, lock2);
                        }
                }
        }

        for (lock2 = ACTIVE_HEAD(gp)->l_next; lock2 != ACTIVE_HEAD(gp);
            lock2 = lock2->l_next) {
```

```
4171                                    ASSERT(!IS_BARRIER(lock1));
4172                    if (lock1->l_vnode == lock2->l_vnode) {
4173                            if (BLOCKS(lock2, lock1)) {
4174                                    ASSERT(!IS_GRANTED(lock1));
4175                                    ASSERT(!NOT_BLOCKED(lock1));
4176                                    path(lock1, lock2);
4177                            }
4178                    }
4179            }
4180            ep = FIRST_ADJ(lock1);
4181            while (ep != HEAD(lock1)) {
4182                    ASSERT(BLOCKS(ep->to_vertex, lock1));
4183                    ep = NEXT_ADJ(ep);
4184            }
4185            }
4186    }

4188    static int
4189    level_two_path(lock_descriptor_t *lock1, lock_descriptor_t *lock2, int no_path)
4190    {
4191            edge_t  *ep;
4192            lock_descriptor_t       *vertex;
4193            lock_descriptor_t *vertex_stack;

4195            STACK_INIT(vertex_stack);

4197            flk_graph_uncolor(lock1->l_graph);
4198            ep = FIRST_ADJ(lock1);
4199            ASSERT(ep != HEAD(lock1));
4200            while (ep != HEAD(lock1)) {
4201                    if (no_path)
4202                            ASSERT(ep->to_vertex != lock2);
4203                    STACK_PUSH(vertex_stack, ep->to_vertex, l_dstack);
4204                    COLOR(ep->to_vertex);
4205                    ep = NEXT_ADJ(ep);
4206            }

4208            while ((vertex = STACK_TOP(vertex_stack)) != NULL) {
4209                    STACK_POP(vertex_stack, l_dstack);
4210                    for (ep = FIRST_ADJ(vertex); ep != HEAD(vertex);
4211                        ep = NEXT_ADJ(ep)) {
4212                            if (COLORED(ep->to_vertex))
4213                                    continue;
4214                            COLOR(ep->to_vertex);
4215                            if (ep->to_vertex == lock2)
4216                                    return (1);

4218                            STACK_PUSH(vertex_stack, ep->to_vertex, l_dstack);
4219                    }
4220            }
4221            return (0);
4222    }

4224    static void
4225    check_owner_locks(graph_t *gp, pid_t pid, int sysid, vnode_t *vp)
4226    {
4227            lock_descriptor_t *lock;

4229            SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

4231            if (lock) {
4232                    while (lock != ACTIVE_HEAD(gp) && (lock->l_vnode == vp)) {
4233                            if (lock->l_flock.l_pid == pid &&
4234                                lock->l_flock.l_sysid == sysid)
4235                                    cmn_err(CE_PANIC,
4236                                        "owner pid %d's lock %p in active queue",
```



```
4237                                        pid, (void *)lock);
4238                            lock = lock->l_next;
4239                    }
4240            }
4241            SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

4243            if (lock) {
4244                    while (lock != SLEEPING_HEAD(gp) && (lock->l_vnode == vp)) {
4245                            if (lock->l_flock.l_pid == pid &&
4246                                lock->l_flock.l_sysid == sysid)
4247                                    cmn_err(CE_PANIC,
4248                                        "owner pid %d's lock %p in sleep queue",
4249                                        pid, (void *)lock);
4250                            lock = lock->l_next;
4251                    }
4252            }
4253    }

4255    static int
4256    level_one_path(lock_descriptor_t *lock1, lock_descriptor_t *lock2)
4257    {
4258            edge_t *ep = FIRST_ADJ(lock1);

4260            while (ep != HEAD(lock1)) {
4261                    if (ep->to_vertex == lock2)
4262                            return (1);
4263                    else
4264                            ep = NEXT_ADJ(ep);
4265            }
4266            return (0);
4267    }

4269    static int
4270    no_path(lock_descriptor_t *lock1, lock_descriptor_t *lock2)
4271    {
4272            return (!level_two_path(lock1, lock2, 1));
4273    }

4275    static void
4276    path(lock_descriptor_t *lock1, lock_descriptor_t *lock2)
4277    {
4278            if (level_one_path(lock1, lock2)) {
4279                    if (level_two_path(lock1, lock2, 0) != 0) {
4280                            cmn_err(CE_WARN,
4281                                "one edge one path from lock1 %p lock2 %p",
4282                                (void *)lock1, (void *)lock2);
4283                    }
4284            } else if (no_path(lock1, lock2)) {
4285                    cmn_err(CE_PANIC,
4286                        "No path from  lock1 %p to lock2 %p",
4287                        (void *)lock1, (void *)lock2);
4288            }
4289    }
4290    #endif /* DEBUG */
```