```
**********************************************************
  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_
```

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

static void
flk_get_first_blocking_lock(lock_descriptor_t *request)
{
        graph_t *gp = request->l_graph;
        vnode_t *vp = request->l_vnode;
        lock_descriptor_t *lock, *blocker;

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

        if (lock) {
                do {
                        if (BLOCKS(lock, request)) {
                                blocker = lock;
                                break;
                        }
                        lock = lock->l_next;
                } while (lock->l_vnode == vp);
        }

        if (blocker == NULL && request->l_flock.l_type == F_RDLCK) {
                /*
                 * No active lock is blocking this request, but if a read
                 * lock is requested, it may also get blocked by a waiting
                 * writer. So search all sleeping locks and see if there is
                 * a writer waiting.
                 */
                SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);
                if (lock) {
                        do {
                                if (BLOCKS(lock, request)) {
                                        blocker = lock;
                                        break;
                                }
                                lock = lock->l_next;
                        } while (lock->l_vnode == vp);
                }
        }

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

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

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

        if (initialize == FLK_USE_GRAPH) {
                mutex_enter(&flock_lock);
                gp = lock_graph[index];
                mutex_exit(&flock_lock);
                return (gp);
        }

        ASSERT(initialize == FLK_INIT_GRAPH);

        if (lock_graph[index] == NULL) {

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

                /* Initialize the graph */

                gp_alloc->active_locks.l_next =
                    gp_alloc->active_locks.l_prev =
                    (lock_descriptor_t *)ACTIVE_HEAD(gp_alloc);
                gp_alloc->sleeping_locks.l_next =
                    gp_alloc->sleeping_locks.l_prev =
                    (lock_descriptor_t *)SLEEPING_HEAD(gp_alloc);
                gp_alloc->index = index;
                mutex_init(&gp_alloc->gp_mutex, NULL, MUTEX_DEFAULT, NULL);
        }

        mutex_enter(&flock_lock);

        gp = lock_graph[index];

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

                /* We must have previously allocated the graph_t structure */
                ASSERT(gp_alloc != NULL);
                lock_graph[index] = gp = gp_alloc;
                /*
                 * The lockmgr status is only needed if KLM is loaded.
                 */
                if (flock_zone_key != ZONE_KEY_UNINITIALIZED) {
                        fg = flk_get_globals();
                        fg->lockmgr_status[index] = fg->flk_lockmgr_status;
                }
        }

        mutex_exit(&flock_lock);

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

        return (gp);
}

/*
 * PSARC case 1997/292
 */
int
cl_flk_has_remote_locks_for_nlmid(vnode_t *vp, int nlmid)
{
        lock_descriptor_t *lock;
        int result = 0;
        graph_t *gp;
        int                     lock_nlmid;

        /*
         * Check to see if node is booted as a cluster. If not, return.
         */
        if ((cluster_bootflags & CLUSTER_BOOTED) == 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) {
                        /* get NLM id from sysid */
                        lock_nlmid = GETNLMID(lock->l_flock.l_sysid);

                        /*
                         * If NLM server request _and_ nlmid of lock matches
                         * nlmid of argument, then we’ve found a remote lock.
                         */
                        if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
                                result = 1;
                                goto done;
                        }
                        lock = lock->l_next;
                }
        }

        SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

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

                        /*
                         * If NLM server request _and_ nlmid of lock matches
                         * nlmid of argument, then we’ve found a remote lock.
                         */
                        if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
                                result = 1;
                                goto done;
                        }
                        lock = lock->l_next;
                }
```

```
        }

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

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

        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 (IS_REMOTE(lock)) {
                                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 (IS_REMOTE(lock)) {
                                result = 1;
                                goto done;
                        }
                        lock = lock->l_next;
                }
        }

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

/*
 * Determine whether there are any locks for the given vnode with a remote
 * sysid matching given sysid.
 * Used by the new (open source) NFS Lock Manager (NLM)
 */
int
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 */
                lock = SLEEPING_HEAD(gp)->l_next;
                while (lock != SLEEPING_HEAD(gp)) {
                        nlock = lock->l_next;
                        if (lock->l_flock.l_sysid == sysid) {
                                INTERRUPT_WAKEUP(lock);
                        }
                        lock = nlock;
                }

                /* delete active locks */
                lock = ACTIVE_HEAD(gp)->l_next;
                while (lock != ACTIVE_HEAD(gp)) {
                        nlock = lock->l_next;
                        if (lock->l_flock.l_sysid == sysid) {
                                flk_delete_active_lock(lock, 0);
                                flk_wakeup(lock, 1);
                                flk_free_lock(lock);
                        }
                        lock = nlock;
                }
                mutex_exit(&gp->gp_mutex);    /* release mutex on lock graph */
        }
}

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

static void
flk_delete_locks_by_sysid(lock_descriptor_t *request)
{
        int     sysid  = request->l_flock.l_sysid;
        lock_descriptor_t *lock, *nlock;
        graph_t *gp;
        int i;

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

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

        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);

                /* signal sleeping requests so that they bail out */
                lock = SLEEPING_HEAD(gp)->l_next;
                while (lock != SLEEPING_HEAD(gp)) {
                        nlock = lock->l_next;
                        if (lock->l_flock.l_sysid == sysid) {
                                INTERRUPT_WAKEUP(lock);
                        }
                        lock = nlock;
                }

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

```
                                flk_delete_active_lock(lock, 0);
                                flk_wakeup(lock, 1);
                                flk_free_lock(lock);
                        }
                        lock = nlock;
                }
                mutex_exit(&gp->gp_mutex);
        }

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

/*
 * Clustering: Deletes PXFS locks
 * Effects: Delete all locks on files in the given file system and with the
 *  given PXFS id.
 */
void
cl_flk_delete_pxfs_locks(struct vfs *vfsp, int pxfsid)
{
        lock_descriptor_t *lock, *nlock;
        graph_t *gp;
        int i;

        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);

                /* signal sleeping requests so that they bail out */
                lock = SLEEPING_HEAD(gp)->l_next;
                while (lock != SLEEPING_HEAD(gp)) {
                        nlock = lock->l_next;
                        if (lock->l_vnode->v_vfsp == vfsp) {
                                ASSERT(IS_PXFS(lock));
                                if (GETPXFSID(lock->l_flock.l_sysid) ==
                                    pxfsid) {
                                        flk_set_state(lock,
                                            FLK_CANCELLED_STATE);
                                        flk_cancel_sleeping_lock(lock, 1);
                                }
                        }
                        lock = nlock;
                }

                /* delete active locks */
                lock = ACTIVE_HEAD(gp)->l_next;
                while (lock != ACTIVE_HEAD(gp)) {
                        nlock = lock->l_next;
                        if (lock->l_vnode->v_vfsp == vfsp) {
                                ASSERT(IS_PXFS(lock));
                                if (GETPXFSID(lock->l_flock.l_sysid) ==
                                    pxfsid) {
                                        flk_delete_active_lock(lock, 0);
                                        flk_wakeup(lock, 1);
                                        flk_free_lock(lock);
                                }
                        }
                        lock = nlock;
                }
                mutex_exit(&gp->gp_mutex);
        }
}

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

static int
flk_canceled(lock_descriptor_t *request)
{
        lock_descriptor_t *lock, *nlock;
        graph_t *gp = request->l_graph;
        vnode_t *vp = request->l_vnode;

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

        if (lock) {
                while (lock->l_vnode == vp) {
                        nlock = lock->l_next;
                        if (SAME_OWNER(lock, request) &&
                            lock->l_start == request->l_start &&
                            lock->l_end == request->l_end) {
                                INTERRUPT_WAKEUP(lock);
                                return (1);
                        }
                        lock = nlock;
                }
        }
        return (0);
}

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

void
cleanlocks(vnode_t *vp, pid_t pid, int sysid)
{
        graph_t *gp;
        lock_descriptor_t *lock, *nlock;
        lock_descriptor_t *link_stack;

        STACK_INIT(link_stack);

        gp = flk_get_lock_graph(vp, FLK_USE_GRAPH);

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

        CHECK_SLEEPING_LOCKS(gp);
        CHECK_ACTIVE_LOCKS(gp);

        SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

        if (lock) {
                do {
                        nlock = lock->l_next;
                        if ((lock->l_flock.l_pid == pid ||
                            pid == IGN_PID) &&
                            lock->l_flock.l_sysid == sysid) {
```

```
                                CANCEL_WAKEUP(lock);
                        }
                        lock = nlock;
                } while (lock->l_vnode == vp);
        }

        SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

        if (lock) {
                do {
                        nlock = lock->l_next;
                        if ((lock->l_flock.l_pid == pid ||
                            pid == IGN_PID) &&
                            lock->l_flock.l_sysid == sysid) {
                                flk_delete_active_lock(lock, 0);
                                STACK_PUSH(link_stack, lock, l_stack);
                        }
                        lock = nlock;
                } while (lock->l_vnode == vp);
        }

        while ((lock = STACK_TOP(link_stack)) != NULL) {
                STACK_POP(link_stack, l_stack);
                flk_wakeup(lock, 1);
                flk_free_lock(lock);
        }

        CHECK_SLEEPING_LOCKS(gp);
        CHECK_ACTIVE_LOCKS(gp);
        CHECK_OWNER_LOCKS(gp, pid, sysid, vp);
        mutex_exit(&gp->gp_mutex);
}


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

int
chklock(
        struct vnode    *vp,
        int             iomode,
        u_offset_t      offset,
        ssize_t         len,
        int             fmode,
        caller_context_t *ct)
{
        register int    i;
        struct flock64  bf;
        int             error = 0;

        bf.l_type = (iomode & FWRITE) ? F_WRLCK : F_RDLCK;
        bf.l_whence = 0;
        bf.l_start = offset;
        bf.l_len = len;
        if (ct == NULL) {
                bf.l_pid = curproc->p_pid;
                bf.l_sysid = 0;
        } else {
                bf.l_pid = ct->cc_pid;
                bf.l_sysid = ct->cc_sysid;
        }
        i = (fmode & (FNDELAY|FNONBLOCK)) ? INOFLCK : INOFLCK|SLPFLCK;
        if ((i = reclock(vp, &bf, i, 0, offset, NULL)) != 0 ||
            bf.l_type != F_UNLCK)
```

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

/*
 * convoff - converts the given data (start, whence) to the
 * given whence.
 */
int
convoff(vp, lckdat, whence, offset)
        struct vnode    *vp;
        struct flock64  *lckdat;
        int             whence;
        offset_t        offset;
{
        int             error;
        struct vattr    vattr;

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

        switch (lckdat->l_whence) {
        case 1:
                lckdat->l_start += offset;
                break;
        case 2:
                lckdat->l_start += vattr.va_size;
                /* FALLTHRU */
        case 0:
                break;
        default:
                return (EINVAL);
        }

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

        switch (whence) {
        case 1:
                lckdat->l_start -= offset;
                break;
        case 2:
                lckdat->l_start -= vattr.va_size;
                /* FALLTHRU */
        case 0:
                break;
        default:
                return (EINVAL);
        }

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


/*      proc_graph function definitions */

/*
 * Function checks for deadlock due to the new ’lock’. If deadlock found
 * edges of this lock are freed and returned.
 */

static int
```

```
flk_check_deadlock(lock_descriptor_t *lock)
{
        proc_vertex_t   *start_vertex, *pvertex;
        proc_vertex_t *dvertex;
        proc_edge_t *pep, *ppep;
        edge_t  *ep, *nep;
        proc_vertex_t *process_stack;

        STACK_INIT(process_stack);

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

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

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

                adj_proc = flk_get_proc_vertex(ep->to_vertex);
                for (pep = start_vertex->edge; pep != NULL; pep = pep->next) {
                        if (pep->to_proc == adj_proc) {
                                ASSERT(pep->refcount);
                                pep->refcount++;
                                break;
                        }
                }
                if (pep == NULL) {
                        pep = flk_get_proc_edge();
                        pep->to_proc = adj_proc;
                        pep->refcount = 1;
                        adj_proc->incount++;
                        pep->next = start_vertex->edge;
                        start_vertex->edge = pep;
                }
                ep = NEXT_ADJ(ep);
        }

        ep = FIRST_IN(lock);

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

                in_proc = flk_get_proc_vertex(ep->from_vertex);

                for (pep = in_proc->edge; pep != NULL; pep = pep->next) {
                        if (pep->to_proc == start_vertex) {
                                ASSERT(pep->refcount);
                                pep->refcount++;
                                break;
                        }
                }
                if (pep == NULL) {
                        pep = flk_get_proc_edge();
                        pep->to_proc = start_vertex;
                        pep->refcount = 1;
                        start_vertex->incount++;
                        pep->next = in_proc->edge;
                        in_proc->edge = pep;
                }
                ep = NEXT_IN(ep);
        }

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

```
                        return (0);
                }

                flk_proc_graph_uncolor();

                start_vertex->p_sedge = start_vertex->edge;

                STACK_PUSH(process_stack, start_vertex, p_stack);

                while ((pvertex = STACK_TOP(process_stack)) != NULL) {
                        for (pep = pvertex->p_sedge; pep != NULL; pep = pep->next) {
                                dvertex = pep->to_proc;
                                if (!PROC_ARRIVED(dvertex)) {
                                        STACK_PUSH(process_stack, dvertex, p_stack);
                                        dvertex->p_sedge = dvertex->edge;
                                        PROC_ARRIVE(pvertex);
                                        pvertex->p_sedge = pep->next;
                                        break;
                                }
                                if (!PROC_DEPARTED(dvertex))
                                        goto deadlock;
                        }
                        if (pep == NULL) {
                                PROC_DEPART(pvertex);
                                STACK_POP(process_stack, p_stack);
                        }
                }
                mutex_exit(&flock_lock);
                return (0);

        deadlock:

                /* we remove all lock edges and proc edges */

                ep = FIRST_ADJ(lock);
                while (ep != HEAD(lock)) {
                        proc_vertex_t *adj_proc;
                        adj_proc = flk_get_proc_vertex(ep->to_vertex);
                        nep = NEXT_ADJ(ep);
                        IN_LIST_REMOVE(ep);
                        ADJ_LIST_REMOVE(ep);
                        flk_free_edge(ep);
                        ppep = start_vertex->edge;
                        for (pep = start_vertex->edge; pep != NULL; ppep = pep,
                            pep = ppep->next) {
                                if (pep->to_proc == adj_proc) {
                                        pep->refcount--;
                                        if (pep->refcount == 0) {
                                                if (pep == ppep) {
                                                        start_vertex->edge = pep->next;
                                                } else {
                                                        ppep->next = pep->next;
                                                }
                                                adj_proc->incount--;
                                                flk_proc_release(adj_proc);
                                                flk_free_proc_edge(pep);
                                        }
                                        break;
                                }
                        }
                        ep = nep;
                }
                ep = FIRST_IN(lock);
                while (ep != HEAD(lock)) {
                        proc_vertex_t *in_proc;
                        in_proc = flk_get_proc_vertex(ep->from_vertex);
```

```
                        nep = NEXT_IN(ep);
                        IN_LIST_REMOVE(ep);
                        ADJ_LIST_REMOVE(ep);
                        flk_free_edge(ep);
                        ppep = in_proc->edge;
                        for (pep = in_proc->edge; pep != NULL; ppep = pep,
                            pep = ppep->next) {
                                if (pep->to_proc == start_vertex) {
                                        pep->refcount--;
                                        if (pep->refcount == 0) {
                                                if (pep == ppep) {
                                                        in_proc->edge = pep->next;
                                                } else {
                                                        ppep->next = pep->next;
                                                }
                                                start_vertex->incount--;
                                                flk_proc_release(in_proc);
                                                flk_free_proc_edge(pep);
                                        }
                                        break;
                                }
                        }
                        ep = nep;
                }
                flk_proc_release(start_vertex);
                mutex_exit(&flock_lock);
                return (1);
        }

        /*
         * Get a proc vertex. If lock’s pvertex value gets a correct proc vertex
         * from the list we return that, otherwise we allocate one. If necessary,
         * we grow the list of vertices also.
         */

        static proc_vertex_t *
        flk_get_proc_vertex(lock_descriptor_t *lock)
        {
                int i;
                proc_vertex_t   *pv;
                proc_vertex_t   **palloc;

                ASSERT(MUTEX_HELD(&flock_lock));
                if (lock->pvertex != -1) {
                        ASSERT(lock->pvertex >= 0);
                        pv = pgraph.proc[lock->pvertex];
                        if (pv != NULL && PROC_SAME_OWNER(lock, pv)) {
                                return (pv);
                        }
                }
                for (i = 0; i < pgraph.gcount; i++) {
                        pv = pgraph.proc[i];
                        if (pv != NULL && PROC_SAME_OWNER(lock, pv)) {
                                lock->pvertex = pv->index = i;
                                return (pv);
                        }
                }
                pv = kmem_zalloc(sizeof (struct proc_vertex), KM_SLEEP);
                pv->pid = lock->l_flock.l_pid;
                pv->sysid = lock->l_flock.l_sysid;
                flk_proc_vertex_allocs++;
                if (pgraph.free != 0) {
                        for (i = 0; i < pgraph.gcount; i++) {
                                if (pgraph.proc[i] == NULL) {
                                        pgraph.proc[i] = pv;
                                        lock->pvertex = pv->index = i;
                                pgraph.free--;
                                return (pv);
                        }
                }
        }
        palloc = kmem_zalloc((pgraph.gcount + PROC_CHUNK) *
            sizeof (proc_vertex_t *), KM_SLEEP);

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

                kmem_free(pgraph.proc,
                    pgraph.gcount * sizeof (proc_vertex_t *));
        }
        pgraph.proc = palloc;
        pgraph.free += (PROC_CHUNK - 1);
        pv->index = lock->pvertex = pgraph.gcount;
        pgraph.gcount += PROC_CHUNK;
        pgraph.proc[pv->index] = pv;
        return (pv);
}

/*
 * Allocate a proc edge.
 */

static proc_edge_t *
flk_get_proc_edge()
{
        proc_edge_t *pep;

        pep = kmem_zalloc(sizeof (proc_edge_t), KM_SLEEP);
        flk_proc_edge_allocs++;
        return (pep);
}

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

static void
flk_free_proc_edge(proc_edge_t *pep)
{
        ASSERT(pep->refcount == 0);
        kmem_free((void *)pep, sizeof (proc_edge_t));
        flk_proc_edge_frees++;
}

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

static void
flk_proc_graph_uncolor()
{
        int i;

        if (pgraph.mark == UINT_MAX) {
                for (i = 0; i < pgraph.gcount; i++)
                        if (pgraph.proc[i] != NULL) {
                                pgraph.proc[i]->atime = 0;
                                pgraph.proc[i]->dtime = 0;
                        }
                pgraph.mark = 1;
        } else {
```

```
                pgraph.mark++;
        }
}

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

static void
flk_proc_release(proc_vertex_t *proc)
{
        ASSERT(MUTEX_HELD(&flock_lock));
        if (proc->edge == NULL && proc->incount == 0) {
                pgraph.proc[proc->index] = NULL;
                pgraph.free++;
                kmem_free(proc, sizeof (proc_vertex_t));
                flk_proc_vertex_frees++;
        }
}

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

static void
flk_update_proc_graph(edge_t *ep, int delete)
{
        proc_vertex_t *toproc, *fromproc;
        proc_edge_t *pep, *prevpep;

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

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

        ASSERT(pep != NULL);
        while (pep != NULL) {
                if (pep->to_proc == toproc) {
                        ASSERT(pep->refcount > 0);
                        pep->refcount--;
                        if (pep->refcount == 0) {
                                if (pep == prevpep) {
                                        fromproc->edge = pep->next;
                                } else {
                                        prevpep->next = pep->next;
                                }
                                toproc->incount--;
                                flk_proc_release(toproc);
                                flk_free_proc_edge(pep);
                        }
                        break;
                }
                prevpep = pep;
                pep = pep->next;
        }
        flk_proc_release(fromproc);
        mutex_exit(&flock_lock);
        return;
```

```c
add:

        pep = fromproc->edge;

        while (pep != NULL) {
                if (pep->to_proc == toproc) {
                        ASSERT(pep->refcount > 0);
                        pep->refcount++;
                        break;
                }
                pep = pep->next;
        }
        if (pep == NULL) {
                pep = flk_get_proc_edge();
                pep->to_proc = toproc;
                pep->refcount = 1;
                toproc->incount++;
                pep->next = fromproc->edge;
                fromproc->edge = pep;
        }
        mutex_exit(&flock_lock);
}

/*
 * Set the control status for lock manager requests.
 *
 */

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

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

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

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

        if (FLK_REGISTRY_IS_NLM_UNKNOWN(nlm_reg_status, nlmid)) {
                /*
                 * If the NLM server "nlmid" is unknown in the NLM registry,
                 * add it to the registry in the nlm shutting down state.
                 */
                FLK_REGISTRY_CHANGE_NLM_STATE(nlm_reg_status, nlmid,
                    FLK_NLM_SHUTTING_DOWN);
        } else {
                /*
                 * Change the state of the NLM server identified by "nlmid"
                 * in the NLM registry to the argument "nlm_state."
                 */
                FLK_REGISTRY_CHANGE_NLM_STATE(nlm_reg_status, nlmid,
                    nlm_state);
        }

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

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

        case FLK_NLM_SHUTTING_DOWN:
                /*
                 * Wake up all sleeping locks for the NLM server identified
                 * by "nlmid." Note that eventually all woken threads will
                 * have their lock requests cancelled and descriptors
                 * removed from the sleeping lock list.  Note that the NLM
                 * server state associated with each lock descriptor is
                 * changed to FLK_NLM_SHUTTING_DOWN.
                 */
                cl_flk_wakeup_sleeping_nlm_locks(nlmid);
                break;

        case FLK_NLM_DOWN:
                /*
                 * Discard all active, granted locks for this NLM server
                 * identified by "nlmid."
                 */
                cl_flk_unlock_nlm_granted(nlmid);
                break;

        default:
                panic("cl_set_nlm_status: bad status (%d)", nlm_state);
        }
}

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

void
flk_set_lockmgr_status(flk_lockmgr_status_t status)
{
        int i;
        graph_t *gp;
        struct flock_globals *fg;

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

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

        /*
         * If the lock manager is coming back up, all that's needed is to
         * propagate this information to the graphs.  If the lock manager
         * is going down, additional action is required, and each graph's
         * copy of the state is updated atomically with this other action.
         */
        switch (status) {
        case FLK_LOCKMGR_UP:
                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);
                        fg->lockmgr_status[i] = status;
                        mutex_exit(&gp->gp_mutex);
```

```
                }
                break;
        case FLK_WAKEUP_SLEEPERS:
                wakeup_sleeping_lockmgr_locks(fg);
                break;
        case FLK_LOCKMGR_DOWN:
                unlock_lockmgr_granted(fg);
                break;
        default:
                panic("flk_set_lockmgr_status: bad status (%d)", status);
                break;
        }
}

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

locklist_t *
get_lock_list(int list_type, int lock_state, int sysid, boolean_t use_sysid,
                pid_t pid, const vnode_t *vp, zoneid_t zoneid)
{
        lock_descriptor_t       *lock;
        lock_descriptor_t       *graph_head;
        locklist_t              listhead;
        locklist_t              *llheadp;
        locklist_t              *llp;
        locklist_t              *lltp;
        graph_t                 *gp;
        int                     i;
        int                     first_index; /* graph index */
        int                     num_indexes; /* graph index */

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

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

        /* Figure out which graphs we want to look at. */
        if (vp == NULL) {
                first_index = 0;
                num_indexes = HASH_SIZE;
        } else {
                first_index = HASH_INDEX(vp);
                num_indexes = 1;
        }

        for (i = first_index; i < first_index + num_indexes; i++) {
                mutex_enter(&flock_lock);
                gp = lock_graph[i];
                mutex_exit(&flock_lock);
                if (gp == NULL) {
                        continue;
                }

                mutex_enter(&gp->gp_mutex);
                graph_head = (list_type == FLK_ACTIVE_STATE) ?
                    ACTIVE_HEAD(gp) : SLEEPING_HEAD(gp);
                for (lock = graph_head->l_next;
                    lock != graph_head;
                    lock = lock->l_next) {
                        if (use_sysid && lock->l_flock.l_sysid != sysid)
                                continue;
                        if (pid != NOPID && lock->l_flock.l_pid != pid)
                                continue;
                        if (vp != NULL && lock->l_vnode != vp)
                                continue;
                        if (lock_state && !(lock_state & lock->l_state))
                                continue;
                        if (zoneid != lock->l_zoneid && zoneid != ALL_ZONES)
                                continue;
                        /*
                         * A matching lock was found.  Allocate
                         * space for a new locklist entry and fill
                         * it in.
                         */
                        llp = kmem_alloc(sizeof (locklist_t), KM_SLEEP);
                        lltp->ll_next = llp;
                        VN_HOLD(lock->l_vnode);
                        llp->ll_vp = lock->l_vnode;
                        create_flock(lock, &(llp->ll_flock));
                        llp->ll_next = (locklist_t *)NULL;
                        lltp = llp;
                }
                mutex_exit(&gp->gp_mutex);
        }

        llp = llheadp->ll_next;
        return (llp);
}

/*
 * These two functions are simply interfaces to get_lock_list.  They return
 * a list of sleeping or active locks for the given sysid and pid.  See
 * get_lock_list for details.
 *
 * In either case we don’t particularly care to specify the zone of interest;
 * the sysid-space is global across zones, so the sysid will map to exactly one
 * zone, and we’ll return information for that zone.
 */

locklist_t *
flk_get_sleeping_locks(int sysid, pid_t pid)
{
        return (get_lock_list(FLK_SLEEPING_STATE, 0, sysid, B_TRUE, pid, NULL,
            ALL_ZONES));
}

locklist_t *
flk_get_active_locks(int sysid, pid_t pid)
```

```
{
        return (get_lock_list(FLK_ACTIVE_STATE, 0, sysid, B_TRUE, pid, NULL,
            ALL_ZONES));
}

/*
 * Another interface to get_lock_list.  This one returns all the active
 * locks for a given vnode.  Again, see get_lock_list for details.
 *
 * We don’t need to specify which zone’s locks we’re interested in.  The matter
 * would only be interesting if the vnode belonged to NFS, and NFS vnodes can’t
 * be used by multiple zones, so the list of locks will all be from the right
 * zone.
 */

locklist_t *
flk_active_locks_for_vp(const vnode_t *vp)
{
        return (get_lock_list(FLK_ACTIVE_STATE, 0, 0, B_FALSE, NOPID, vp,
            ALL_ZONES));
}

/*
 * Another interface to get_lock_list.  This one returns all the active
 * nbmand locks for a given vnode.  Again, see get_lock_list for details.
 *
 * See the comment for flk_active_locks_for_vp() for why we don’t care to
 * specify the particular zone of interest.
 */
locklist_t *
flk_active_nbmand_locks_for_vp(const vnode_t *vp)
{
        return (get_lock_list(FLK_ACTIVE_STATE, NBMAND_LOCK, 0, B_FALSE,
            NOPID, vp, ALL_ZONES));
}

/*
 * Another interface to get_lock_list.  This one returns all the active
 * nbmand locks for a given pid.  Again, see get_lock_list for details.
 *
 * The zone doesn’t need to be specified here; the locks held by a
 * particular process will either be local (ie, non-NFS) or from the zone
 * the process is executing in.  This is because other parts of the system
 * ensure that an NFS vnode can’t be used in a zone other than that in
 * which it was opened.
 */
locklist_t *
flk_active_nbmand_locks(pid_t pid)
{
        return (get_lock_list(FLK_ACTIVE_STATE, NBMAND_LOCK, 0, B_FALSE,
            pid, NULL, ALL_ZONES));
}

/*
 * Free up all entries in the locklist.
 */
void
flk_free_locklist(locklist_t *llp)
{
        locklist_t *next_llp;

        while (llp) {
                next_llp = llp->ll_next;
                VN_RELE(llp->ll_vp);
                kmem_free(llp, sizeof (*llp));
                llp = next_llp;
        }
}

static void
cl_flk_change_nlm_state_all_locks(int nlmid, flk_nlm_status_t nlm_state)
{
        /*
         * For each graph "lg" in the hash table lock_graph do
         * a.  Get the list of sleeping locks
         * b.  For each lock descriptor in the list do
         *      i.   If the requested lock is an NLM server request AND
         *              the nlmid is the same as the routine argument then
         *              change the lock descriptor’s state field to
         *              "nlm_state."
         * c.  Get the list of active locks
         * d.  For each lock descriptor in the list do
         *      i.   If the requested lock is an NLM server request AND
         *              the nlmid is the same as the routine argument then
         *              change the lock descriptor’s state field to
         *              "nlm_state."
         */

        int                     i;
        graph_t                 *gp;                    /* lock graph */
        lock_descriptor_t       *lock;                  /* lock */
        lock_descriptor_t       *nlock = NULL;          /* next lock */
        int                     lock_nlmid;

        for (i = 0; i < HASH_SIZE; i++) {
                mutex_enter(&flock_lock);
                gp = lock_graph[i];
                mutex_exit(&flock_lock);
                if (gp == NULL) {
                        continue;
                }

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

                        /*
                         * If NLM server request AND nlmid of lock matches
                         * nlmid of argument, then set the NLM state of the
                         * lock to "nlm_state."
                         */
                        if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
                                SET_NLM_STATE(lock, nlm_state);
                        }
                }

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

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

```
                         * lock to "nlm_state."
                         */
                        if (IS_LOCKMGR(lock) && nlmid == lock_nlmid) {
                                ASSERT(IS_ACTIVE(lock));
                                SET_NLM_STATE(lock, nlm_state);
                        }
                }
                mutex_exit(&gp->gp_mutex);
        }
}

/*
 * Requires: "nlmid" >= 1 and <= clconf_maximum_nodeid().
 * Effects: Find all sleeping lock manager requests _only_ for the NLM server
 *   identified by "nlmid." Poke those lock requests.
 */
static void
cl_flk_wakeup_sleeping_nlm_locks(int nlmid)
{
        lock_descriptor_t *lock;
        lock_descriptor_t *nlock = NULL; /* next lock */
        int i;
        graph_t *gp;
        int     lock_nlmid;

        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);
                for (lock = SLEEPING_HEAD(gp)->l_next;
                    lock != SLEEPING_HEAD(gp);
                    lock = nlock) {
                        nlock = lock->l_next;
                        /*
                         * If NLM server request _and_ nlmid of lock matches
                         * nlmid of argument, then set the NLM state of the
                         * lock to NLM_SHUTTING_DOWN, and wake up sleeping
                         * request.
                         */
                        if (IS_LOCKMGR(lock)) {
                                /* get NLM id */
                                lock_nlmid =
                                    GETNLMID(lock->l_flock.l_sysid);
                                if (nlmid == lock_nlmid) {
                                        SET_NLM_STATE(lock,
                                            FLK_NLM_SHUTTING_DOWN);
                                        INTERRUPT_WAKEUP(lock);
                                }
                        }
                }
                mutex_exit(&gp->gp_mutex);
        }
}

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

```
{
        lock_descriptor_t *lock;
        lock_descriptor_t *nlock = NULL; /* next lock */
        int i;
        graph_t *gp;
        int     lock_nlmid;

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

                        /*
                         * If it’s an  NLM server request _and_ nlmid of
                         * the lock matches nlmid of argument, then
                         * remove the active lock the list, wakup blocked
                         * threads, and free the storage for the lock.
                         * Note that there’s no need to mark the NLM state
                         * of this lock to NLM_DOWN because the lock will
                         * be deleted anyway and its storage freed.
                         */
                        if (IS_LOCKMGR(lock)) {
                                /* get NLM id */
                                lock_nlmid = GETNLMID(lock->l_flock.l_sysid);
                                if (nlmid == lock_nlmid) {
                                        flk_delete_active_lock(lock, 0);
                                        flk_wakeup(lock, 1);
                                        flk_free_lock(lock);
                                }
                        }
                }
                mutex_exit(&gp->gp_mutex);
        }
}

/*
 * Find all sleeping lock manager requests and poke them.
 */
static void
wakeup_sleeping_lockmgr_locks(struct flock_globals *fg)
{
        lock_descriptor_t *lock;
        lock_descriptor_t *nlock = NULL; /* next lock */
        int i;
        graph_t *gp;
        zoneid_t zoneid = getzoneid();

        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);
```

```
                fg->lockmgr_status[i] = FLK_WAKEUP_SLEEPERS;
                for (lock = SLEEPING_HEAD(gp)->l_next;
                    lock != SLEEPING_HEAD(gp);
                    lock = nlock) {
                        nlock = lock->l_next;
                        if (IS_LOCKMGR(lock) && lock->l_zoneid == zoneid) {
                                INTERRUPT_WAKEUP(lock);
                        }
                }
                mutex_exit(&gp->gp_mutex);
        }
}


/*
 * Find all active (granted) lock manager locks and release them.
 */
static void
unlock_lockmgr_granted(struct flock_globals *fg)
{
        lock_descriptor_t *lock;
        lock_descriptor_t *nlock = NULL; /* next lock */
        int i;
        graph_t *gp;
        zoneid_t zoneid = getzoneid();

        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);
                fg->lockmgr_status[i] = FLK_LOCKMGR_DOWN;
                for (lock = ACTIVE_HEAD(gp)->l_next;
                    lock != ACTIVE_HEAD(gp);
                    lock = nlock) {
                        nlock = lock->l_next;
                        if (IS_LOCKMGR(lock) && lock->l_zoneid == zoneid) {
                                ASSERT(IS_ACTIVE(lock));
                                flk_delete_active_lock(lock, 0);
                                flk_wakeup(lock, 1);
                                flk_free_lock(lock);
                        }
                }
                mutex_exit(&gp->gp_mutex);
        }
}


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

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

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

        while (!(IS_GRANTED(request)) && !(IS_CANCELLED(request)) &&
            !(IS_INTERRUPTED(request))) {
                if (!cv_wait_sig(&request->l_cv, &gp->gp_mutex)) {
                        flk_set_state(request, FLK_INTERRUPTED_STATE);
                        request->l_state |= INTERRUPTED_LOCK;
                }
        }
}

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

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

        flp->l_type = lp->l_type;
        flp->l_whence = 0;
        flp->l_start = lp->l_start;
        flp->l_len = (lp->l_end == MAX_U_OFFSET_T) ? 0 :
            (lp->l_end - lp->l_start + 1);
        flp->l_sysid = lp->l_flock.l_sysid;
        flp->l_pid = lp->l_flock.l_pid;
}

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

int
flk_convert_lock_data(vnode_t *vp, flock64_t *flp,
    u_offset_t *start, u_offset_t *end, offset_t offset)
{
        struct vattr    vattr;
        int     error;

        /*
         * Determine the starting point of the request
         */
        switch (flp->l_whence) {
        case 0:         /* SEEK_SET */
                *start = (u_offset_t)flp->l_start;
                break;
        case 1:         /* SEEK_CUR */
                *start = (u_offset_t)(flp->l_start + offset);
                break;
        case 2:         /* SEEK_END */
                vattr.va_mask = AT_SIZE;
                if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
                        return (error);
                *start = (u_offset_t)(flp->l_start + vattr.va_size);
                break;
        default:
                return (EINVAL);
        }

        /*
```

```
         * Determine the range covered by the request.
         */
        if (flp->l_len == 0)
                *end = MAX_U_OFFSET_T;
        else if ((offset_t)flp->l_len > 0) {
                *end = (u_offset_t)(*start + (flp->l_len - 1));
        } else {
                /*
                 * Negative length; why do we even allow this ?
                 * Because this allows easy specification of
                 * the last n bytes of the file.
                 */
                *end = *start;
                *start += (u_offset_t)flp->l_len;
                (*start)++;
        }
        return (0);
}

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

int
flk_check_lock_data(u_offset_t start, u_offset_t end, offset_t max)
{
        /*
         * The end (length) for local locking should never be greater
         * than MAXEND. However, the representation for
         * the entire file is MAX_U_OFFSET_T.
         */
        if ((start > max) ||
            ((end > max) && (end != MAX_U_OFFSET_T))) {
                return (EINVAL);
        }
        if (start > end) {
                return (EINVAL);
        }
        return (0);
}

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

static void
report_blocker(lock_descriptor_t *blocker, lock_descriptor_t *request)
{
        flock64_t *flrp;                        /* l_flock portion of request */
        ASSERT(blocker != NULL);

        flrp = &request->l_flock;
        flrp->l_whence = 0;
        flrp->l_type = blocker->l_type;
        flrp->l_pid = blocker->l_flock.l_pid;
        flrp->l_sysid = blocker->l_flock.l_sysid;

        if (IS_LOCKMGR(request)) {
                flrp->l_start = blocker->l_start;
                if (blocker->l_end == MAX_U_OFFSET_T)
                        flrp->l_len = 0;
                else
                        flrp->l_len = blocker->l_end - blocker->l_start + 1;
        } else {
                if (blocker->l_start > MAXEND) {
                        flrp->l_start = MAXEND;
                        flrp->l_len = 0;
                } else {
                        flrp->l_start = blocker->l_start;
                        if (blocker->l_end == MAX_U_OFFSET_T)
                                flrp->l_len = 0;
                        else
                                flrp->l_len = blocker->l_end -
                                    blocker->l_start + 1;
                }
        }
}

/*
 * PSARC case 1997/292
 */
/*
 * This is the public routine exported by flock.h.
 */
void
cl_flk_change_nlm_state_to_unknown(int nlmid)
{
        /*
         * Check to see if node is booted as a cluster. If not, return.
         */
        if ((cluster_bootflags & CLUSTER_BOOTED) == 0) {
                return;
        }

        /*
         * See comment in cl_flk_set_nlm_status().
         */
        if (nlm_reg_status == NULL) {
                return;
        }

        /*
         * protect NLM registry state with a mutex.
         */
        ASSERT(nlmid <= nlm_status_size && nlmid >= 0);
        mutex_enter(&nlm_reg_lock);
        FLK_REGISTRY_CHANGE_NLM_STATE(nlm_reg_status, nlmid, FLK_NLM_UNKNOWN);
        mutex_exit(&nlm_reg_lock);
}

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

```
 */

int
nbl_lock_conflict(vnode_t *vp, nbl_op_t op, u_offset_t offset,
                ssize_t length, int svmand, caller_context_t *ct)
{
        int conflict = 0;
        graph_t                 *gp;
        lock_descriptor_t       *lock;
        pid_t pid;
        int sysid;

        if (ct == NULL) {
                pid = curproc->p_pid;
                sysid = 0;
        } else {
                pid = ct->cc_pid;
                sysid = ct->cc_sysid;
        }

        mutex_enter(&flock_lock);
        gp = lock_graph[HASH_INDEX(vp)];
        mutex_exit(&flock_lock);
        if (gp == NULL)
                return (0);

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

        for (; lock && lock->l_vnode == vp; lock = lock->l_next) {
                if ((svmand || (lock->l_state & NBMAND_LOCK)) &&
                    (lock->l_flock.l_sysid != sysid ||
                    lock->l_flock.l_pid != pid) &&
                    lock_blocks_io(op, offset, length,
                    lock->l_type, lock->l_start, lock->l_end)) {
                        conflict = 1;
                        break;
                }
        }
        mutex_exit(&gp->gp_mutex);

        return (conflict);
}

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

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

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

        if (offset <= lock_start && lock_start < offset + length)
                return (1);
        if (lock_start <= offset && offset <= lock_end)
                return (1);

        return (0);
}
```

```
#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) {
```

```
                                ASSERT(!IS_BARRIER(lock1));
                if (lock1->l_vnode == lock2->l_vnode) {
                        if (BLOCKS(lock2, lock1)) {
                                ASSERT(!IS_GRANTED(lock1));
                                ASSERT(!NOT_BLOCKED(lock1));
                                path(lock1, lock2);
                        }
                }
        }
        ep = FIRST_ADJ(lock1);
        while (ep != HEAD(lock1)) {
                ASSERT(BLOCKS(ep->to_vertex, lock1));
                ep = NEXT_ADJ(ep);
        }
        }
}

static int
level_two_path(lock_descriptor_t *lock1, lock_descriptor_t *lock2, int no_path)
{
        edge_t  *ep;
        lock_descriptor_t       *vertex;
        lock_descriptor_t *vertex_stack;

        STACK_INIT(vertex_stack);

        flk_graph_uncolor(lock1->l_graph);
        ep = FIRST_ADJ(lock1);
        ASSERT(ep != HEAD(lock1));
        while (ep != HEAD(lock1)) {
                if (no_path)
                        ASSERT(ep->to_vertex != lock2);
                STACK_PUSH(vertex_stack, ep->to_vertex, l_dstack);
                COLOR(ep->to_vertex);
                ep = NEXT_ADJ(ep);
        }

        while ((vertex = STACK_TOP(vertex_stack)) != NULL) {
                STACK_POP(vertex_stack, l_dstack);
                for (ep = FIRST_ADJ(vertex); ep != HEAD(vertex);
                    ep = NEXT_ADJ(ep)) {
                        if (COLORED(ep->to_vertex))
                                continue;
                        COLOR(ep->to_vertex);
                        if (ep->to_vertex == lock2)
                                return (1);

                        STACK_PUSH(vertex_stack, ep->to_vertex, l_dstack);
                }
        }
        return (0);
}

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

        SET_LOCK_TO_FIRST_ACTIVE_VP(gp, lock, vp);

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

```
                                    pid, (void *)lock);
                        lock = lock->l_next;
                }
        }
        SET_LOCK_TO_FIRST_SLEEP_VP(gp, lock, vp);

        if (lock) {
                while (lock != SLEEPING_HEAD(gp) && (lock->l_vnode == vp)) {
                        if (lock->l_flock.l_pid == pid &&
                            lock->l_flock.l_sysid == sysid)
                                cmn_err(CE_PANIC,
                                    "owner pid %d’s lock %p in sleep queue",
                                    pid, (void *)lock);
                        lock = lock->l_next;
                }
        }
}

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

        while (ep != HEAD(lock1)) {
                if (ep->to_vertex == lock2)
                        return (1);
                else
                        ep = NEXT_ADJ(ep);
        }
        return (0);
}

static int
no_path(lock_descriptor_t *lock1, lock_descriptor_t *lock2)
{
        return (!level_two_path(lock1, lock2, 1));
}

static void
path(lock_descriptor_t *lock1, lock_descriptor_t *lock2)
{
        if (level_one_path(lock1, lock2)) {
                if (level_two_path(lock1, lock2, 0) != 0) {
                        cmn_err(CE_WARN,
                            "one edge one path from lock1 %p lock2 %p",
                            (void *)lock1, (void *)lock2);
                }
        } else if (no_path(lock1, lock2)) {
                cmn_err(CE_PANIC,
                    "No path from  lock1 %p to lock2 %p",
                    (void *)lock1, (void *)lock2);
        }
}
#endif /* DEBUG */
```