```
**********************************************************
   27608 Mon Aug 19 16:40:53 2013
new/usr/src/uts/common/fs/fsh.c
filesystem hook framework (August 19th)
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2013 Damian Bogel.  All rights reserved.
 */

#include <sys/debug.h>
#include <sys/errno.h>
#include <sys/fsh.h>
#include <sys/fsh_impl.h>
#include <sys/id_space.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
#include <sys/list.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/vfs.h>
#include <sys/vnode.h>

/*
 * TODO:
 * - solve the fsh_cblist_lock deadlock potential problem (fsh_callback_install
 * inside a callback) (that is also a problem for hooks)
 */

/*
 * Filesystem hook framework (fsh)
 *
 * 1. Abstract.
 * The main goal of the filesystem hook framework is to provide an easy way to
 * inject consumer-defined behaviour into vfs/vnode calls. fsh works on
 * vfs_t granularity.
 *
 *
 * 2. Overview.
 * fsh_t is the main object in the fsh. An fsh_t is a structure containing:
 *      - pointers to hooking functions (named after corresponding
 *      vnodeops/vfsops)
 *      - a pointer to an argument to pass (this is shared for all the
 *      hooks in a given fsh_t)
 *      - pointer to a callback that should be executed after a hook is removed
 *
 * The information from fsh_t is copied by the fsh and an fsh_handle_t
 * is returned. It should be used for further removing.
 *
 *
 * 3. Usage.
 * It is expected that vfs_t/vnode_t that are passed to fsh_foo() functions
 * are held by the caller when needed. fsh does no vfs_t/vnode_t locking.
 *
 * fsh_t is a structure filled out by the consumer. If a consumer does not want
```



```
 * to add/remove a hook for function foo(), he should fill the foo field of
 * fsh_t with NULL. Every hook has a type of corresponding vfsop/vnodeop with
 * two additional arguments:
 *      - fsh_int_t *fsh_int - this argument MUST be passed to
 *      hook_next_foo(). fsh wouldn't know which hook to execute next
 *      without it
 *      - void *arg - this is the argument passed with fsh_t during
 *      installation
 * fsh_t containts also two other fields:
 *      - void *arg - mentioned above, which is the argument passed to the hooks
 *      - remove_callback - it's a pointer to a callback that should execute
 *      right after the hook has been removed. It takes the arg and the handle
 *      as parameters.
 * After installation, an fsh_handle_t is returned to the caller.
 *
 * When a vfs_t is freed, all remaining hooks are being removed and the
 * remove_callbacks are being executed.
 *
 * Every hook function is responsible for passing the control to the next
 * hook associated with a particular call. In order to provide an easy way to
 * modify the behaviour of a function call both before and after the
 * underlying vfsop/vnodeop (or next hook) execution, a hook has to call
 * fsh_next_foo() at some point. This function does necessary internal
 * operations and calls the next hook, until there's no hook left, then it
 * calls the underlying vfsop/vnodeop.
 * Example:
 * my_freefs(fsh_int_t *fsh_int, void *arg, vfs_t *vfsp) {
 *      cmn_err(CE_NOTE, "freefs called!\n");
 *      return (fsh_next_freefs(fsh_int, vfsp));
 * }
 *
 * A consumer might want to fire callbacks when vfs_t's are being mounted
 * or freed. There's an fsh_callback_t structure provided to install such
 * callbacks along with the API.
 *
 *
 * 4. API
 * None of the APIs should be called during interrupt context above lock
 * level. The only exceptions are fsh_next_foo() functions, which do not use
 * locks.
 *
 * a) fsh.h
 * None of the functions listed below should be called inside of a hook
 * Doing so will cause a deadlock. The only exceptions are fsh_next_foo() and
 * fsh_callback_{install,remove}().
 *
 * fsh_callback_{install,remove}() should not be called inside of a {mount,free}
 * callback. Doing so will cause a deadlock.
 *
 * fsh_fs_enable(vfs_t *vfsp)
 * fsh_fs_disable(vfs_t *vfsp)
 *      Enables/disables fsh for a given vfs_t.
 *
 * fsh_hook_install(vfs_t *vfsp, fsh_t *hooks)
 *      Installs hooks on vfsp filesystem. It's important that hooks are
 *      executed in LIFO installation order, which means that if there are
 *      hooks A and B installed in this order, B is going to be executed
 *      before A. It returns a correct handle, or (-1) if hook/callback
 *      limit exceeded.
 *
 * fsh_hook_remove(fsh_handle_t handle)
 *      Removes hooks.
 *
 * fsh_next_foo(fsh_int_t *fsh_int, void *arg, ARGUMENTS)
 *      This is the function which should be called once in every hook. It
 *      does the necessary internal operations and passes control to the
```

```
 *      next hook or, if there's no hook left, to the underlying
 *      vfsop/vnodeop.
 *
 * fsh_callback_install(fsh_callback_t *callback)
 * fsh_callback_remove(fsh_callback_handle_t handle)
 *      Installs/removes callbacks for vfs_t mount/free. The mount callback
 *      is executed right before domount() returns. The free callback is
 *      called right before VFS_FREEVFS() is called. The
 *      fsh_callback_install() returns a correct handle, or (-1) if
 *      hook/callback limit exceeded.
 *
 * b) fsh_impl.h (for vfs.c and vnode.c only)
 * fsh_init()
 *      This call has to be done in vfsinit(). It initialises the fsh. It
 *      is absolutely necessary that this call is made before any other fsh
 *      operation.
 *
 * fsh_exec_mount_callbacks(vfs_t *vfsp)
 * fsh_exec_free_callbacks(vfs_t *vfsp)
 *      Used to execute all fsh callbacks for {mount,free} of a vfs_t.
 *
 * fsh_fsrec_destroy(struct fsh_fsrecord *fsrecp)
 *      Destroys an fsh_fsrecord structure.
 *
 * fsh_foo(ARGUMENTS)
 *      Function used to start executing the hook chain for a given call.
 *
 * ---
 *
 * Because of the fact that unremoved hooks are being removed when a vfs_t
 * is being freed, the client should not expect that remove_callback()
 * would be called only when the vfs_t is still alive.
 *
 * 5. Internals.
 * fsh_fsrecord_t is a structure which lives inside a vfs_t.
 * fsh_fsrecord_t contains:
 *      - an rw-lock that protects the structure
 *      - a list of hooks installed on this vfs_t
 *      - a flag which tells whether fsh is enabled on this vfs_t
 *
 * Unfortunately, because of unexpected behaviour of some filesystems (no
 * use of vfs_alloc()/vfs_init()) there's no good place to initialise the
 * fsh_fshrecord_t structure. The approach being used here is to check if
 * it's initialised in every call. Because of the fact that no lock could
 * be used here (the same problem with initialisation), a spinlock is used.
 * This is explained in more detail in a comment before
 * fsh_prepare_fsrec(), a function that should be used whenever a
 * vfsp->vfs_fshrecord needs to be accessed.  After doing that, it's
 * completely safe to keep this pointer locally, because it won't be
 * changed until vfs_free() is called.
 *
 * The only exception from the fsh_prepare_fsrec() rule is vfs_free(),
 * where there is expected that no other fsh calls would be made for the
 * vfs_t that's being freed.
 *
 * When there are no fsh functions that use a particular fsh_fsrecord_t
 * executing, the vfs_fshrecord pointer won't be equal to fsh_res_ptr. It
 * would be NULL or a pointer to an initialised fsh_fsrecord_t.
 *
 *
 * fsh_next_foo()
 * This function is quite simple. It takes the fsh_int_t and passes control to
 * the next hook or to the underlying vnodeop/vfsop.
 *
 *
 * Mount callbacks are executed by a call to fsh_exec_mount_callbacks() right
```



```
 * before returning from domount()@vfs.c.
 *
 * Free callbacks are executed by a call to fsh_exec_free_callbacks() right
 * before calling VFS_FREEVFS(), after vfs_t's reference count drops to 0.
 *
 *
 * 6. Concurrency
 * fsh does no vfs_t nor vnode_t locking. It is expected that whenever it is
 * needed, the consumer does that.
 *
 * An fsh_fsrecord_t of a vfs_t is read-locked (fshfsr_lock) by every
 * fsh_foo() function (with the mentioned vfs_t as a parameter, of course).
 * This means that fsh_hook_{install,remove}() must NOT be called inside of
 * a hook, because it will cause a deadlock.
 *
 * The same thing applies to callbacks. fsh_cblist is read-locked by
 * fsh_exec_{mount,free}(). This means that fsh_callback_{install,remove}
 * must not be called inside a callback, because it will cause a deadlock.
 *
 * Solution to concurrency issues involving vfs_fshrecord are explained
 * both in chapter 5th "Internals" and before fsh_prepare_fsrec() function.
 */

/* Internals */
struct fsh_int {
        fsh_handle_t    fshi_handle;
        fsh_t           fshi_hooks;
        list_node_t     fshi_next;
};

typedef struct fsh_callback_int {
        fsh_callback_t  fshci_cb;
        fsh_callback_handle_t fshci_handle;
        list_node_t     fshci_next;
} fsh_callback_int_t;

/* Used for mapping an fsh_handle_t to fsh_int_t. */
typedef struct fsh_mapping {
        fsh_handle_t    fshm_handle;
        fsh_int_t       *fshm_fshi;
        vfs_t           *fshm_vfsp;
        list_node_t     fshm_next;
} fsh_mapping_t;

/*
 * fsh_fsrecord_t is the main internal structure. It's content is protected
 * by fshfsr_lock. The fshfsr_list is a list of fsh_int_t hook entries for
 * the vfs_t that contains the fsh_fsrecord_t.
 *
 * It is guaranteed by the fsh_prepare_fsrec() that outside the fsh,
 * a pointer to fsh_fsrecord inside a vfs_t is never equal to fsh_res_ptr.
 */
struct fsh_fsrecord {
        krwlock_t       fshfsr_lock;
        int             fshfsr_enabled;
        list_t          fshfsr_list;    /* list of fsh_int_t */
};

/*
 * It's a list of fsh_mapping_t's used to map fsh_handle_t's to
 * fsh_int_t's. This is needed because of the fact that we'd like an opaque
 * handle returned to the fsh API client after a hook is successfully
 * installed. We'd like to make the handle the only thing that is needed
 * after the hooks are installed, for futher actions on them. This means,
 * that there is no easy way to search for the hooks matching a handle,
 * without having the vfs_t on which they are installed.
```

```
 * The same problem doesn't apply to callbacks, that's why fsh_map handles
 * only fsh_handle_t to fsh_int_t translation.
 */
static kmutex_t fsh_map_lock;
static list_t fsh_map;

/*
 * It's a list of fsh_callback_int_t's. Unlike hooks, there is no need to
 * keep a separate list for translating handles to fsh_callback_int_t's,
 * because a callback list is global for all the vfs_t's.
 */
static krwlock_t fsh_cblist_lock;
static list_t fsh_cblist;

/*
 * A reserved pointer for fsh purposes. It is used because of the method
 * chosen for solving concurrency issues with vfs_fshrecord. The full
 * explanation is in the big theory statement at the beginning of this
 * file. It is initialised in fsh_init().
 */
static void *fsh_res_ptr;

static fsh_fsrecord_t *fsh_fsrec_create();

int fsh_limit = INT_MAX;
static id_space_t *fsh_idspace;

/*
 * Important note:
 * Before using this function, fsh_init() MUST be called. We do that in
 * vfsinit()@vfs.c.
 *
 * One would ask, why isn't the vfsp->vfs_fshrecord initialised when the
 * vfs_t is created. Unfortunately, some filesystems (e.g. fifofs) do not
 * call vfs_init() or even vfs_alloc(), It's possible that some unbundled
 * filesystems could do the same thing. That's why this solution is
 * introduced. It should be called before any code that needs access to
 * vfs_fshrecord.
 *
 * Locking:
 * There are no locks here, because there's no good place to initialise
 * the lock. Concurrency issues are solved by using atomic instructions
 * and a spinlock, which is spinning only once for a given vfs_t. Because
 * of that, the usage of the spinlock isn't bad at all.
 *
 * How it works:
 * a) if vfsp->vfs_fshrecord equals NULL, atomic_cas_ptr() changes it to
 *      fsh_res_ptr. That's a signal for other threads, that the structure
 *      is being initialised.
 * b) if vfsp->vfs_fshrecord equals fsh_res_ptr, that means we have to wait,
 *      because vfs_fshrecord is being initialised by another call.
 * c) other cases:
 *      vfs_fshrecord is already initialised, so we can use it. It won't change
 *      until vfs_free() is called. It can't happen when someone is holding
 *      the vfs_t, which is expected from the caller of fsh API.
 */
static void
fsh_prepare_fsrec(vfs_t *vfsp)
{
        fsh_fsrecord_t *fsrec;

        while ((fsrec = atomic_cas_ptr(&vfsp->vfs_fshrecord, NULL,
            fsh_res_ptr)) == fsh_res_ptr)
                ;

        if (fsrec == NULL)
```



```
                atomic_swap_ptr(&vfsp->vfs_fshrecord, fsh_fsrec_create());
}

/*
 * API for enabling/disabling fsh per vfs_t.
 *
 * These functions must NOT be called in a hook.
 */
void
fsh_fs_enable(vfs_t *vfsp)
{
        fsh_prepare_fsrec(vfsp);

        rw_enter(&vfsp->vfs_fshrecord->fshfsr_lock, RW_WRITER);
        vfsp->vfs_fshrecord->fshfsr_enabled = 1;
        rw_exit(&vfsp->vfs_fshrecord->fshfsr_lock);
}

void
fsh_fs_disable(vfs_t *vfsp)
{
        fsh_prepare_fsrec(vfsp);

        rw_enter(&vfsp->vfs_fshrecord->fshfsr_lock, RW_WRITER);
        vfsp->vfs_fshrecord->fshfsr_enabled = 0;
        rw_exit(&vfsp->vfs_fshrecord->fshfsr_lock);
}

/*
 * API used for installing hooks. fsh_handle_t is returned for further
 * actions (currently just removing) on this set of hooks.
 *
 * fsh_t fields:
 * - arg - argument passed to every hook
 * - remove_callback - callback executed right after a hook is removed
 * - read, write, ... - pointers to hooks for corresponding vnodeops/vfsops;
 *      if there is no hook desired for an operation, it should be set to
 *      NULL
 *
 * It's important that the hooks are executed in LIFO installation order (they
 * are added to the head of the hook list).
 *
 * This function must NOT be called in a hook.
 *
 * Returns (-1) if hook/callback limit exceeded, handle otherwise.
 */
fsh_handle_t
fsh_hook_install(vfs_t *vfsp, fsh_t *hooks)
{
        fsh_handle_t    handle;
        fsh_int_t       *fshi;
        fsh_mapping_t   *mapping;

        fsh_prepare_fsrec(vfsp);

        if ((handle = id_alloc(fsh_idspace)) == -1)
                return (-1);

        fshi = kmem_alloc(sizeof (*fshi), KM_SLEEP);
        (void) memcpy(&fshi->fshi_hooks, hooks, sizeof (fshi->fshi_hooks));
        fshi->fshi_handle = handle;

        /* If it is called inside of a hook, causes deadlock. */
        rw_enter(&vfsp->vfs_fshrecord->fshfsr_lock, RW_WRITER);
        list_insert_head(&vfsp->vfs_fshrecord->fshfsr_list, fshi);
        rw_exit(&vfsp->vfs_fshrecord->fshfsr_lock);
```

```
394         mapping = kmem_alloc(sizeof (*mapping), KM_SLEEP);
395         mapping->fshm_handle = handle;
396         mapping->fshm_vfsp = vfsp;
397         mapping->fshm_fshi = fshi;

399         mutex_enter(&fsh_map_lock);
400         list_insert_head(&fsh_map, mapping);
401         mutex_exit(&fsh_map_lock);

403         return (handle);
404 }

406 /* Finds a mapping by handle and removes it from from fsh_map. */
407 static fsh_mapping_t *
408 fsh_remove_mapping(fsh_handle_t handle)
409 {
410         fsh_mapping_t *mapping;

412         mutex_enter(&fsh_map_lock);
413         for (mapping = list_head(&fsh_map); mapping != NULL;
414             mapping = list_next(&fsh_map, mapping)) {
415                 if (mapping->fshm_handle == handle) {
416                         list_remove(&fsh_map, mapping);
417                         break;
418                 }
419         }
420         mutex_exit(&fsh_map_lock);
421         return (mapping);
422 }

424 /*
425  * Used for removing a hook set. remove_callback() is executed right after
426  * the hook is removed from the vfs_t.
427  *
428  * This function must NOT be called in a hook.
429  *
430  * Returns (-1) if hook wasn't found, 0 otherwise.
431  */
432 int
433 fsh_hook_remove(fsh_handle_t handle)
434 {
435         fsh_fsrecord_t  *fsrecp;
436         fsh_mapping_t   *mapping;
437         fsh_t           *hooks;

439         mapping = fsh_remove_mapping(handle);
440         if (mapping == NULL)
441                 return (-1);

443         ASSERT(mapping->fshm_fshi->fshi_handle == handle);

445         /*
446          * We don't have to call fsh_prepare_fsrec() here. fsh_fsrecord_t
447          * is already initialised, because we've found a mapping for the given
448          * handle. We instead make two ASSERTs.
449          */
450         fsrecp = mapping->fshm_vfsp->vfs_fshrecord;
451         ASSERT(fsrecp != NULL);
452         ASSERT(fsrecp != fsh_res_ptr);

454         /* If it is called inside of a hook, causes deadlock. */
455         rw_enter(&fsrecp->fshfsr_lock, RW_WRITER);
456         list_remove(&fsrecp->fshfsr_list, mapping->fshm_fshi);
457         rw_exit(&fsrecp->fshfsr_lock);
```



```
459         hooks = &mapping->fshm_fshi->fshi_hooks;
460         if (hooks->remove_callback != NULL)
461                 (*(hooks->remove_callback))(hooks->arg, mapping->fshm_handle);

463         id_free(fsh_idspace, handle);

465         kmem_free(mapping->fshm_fshi, sizeof (*mapping->fshm_fshi));
466         kmem_free(mapping, sizeof (*mapping));

468         return (0);
469 }

471 /*
472  * API for installing global mount/free callbacks.
473  *
474  * fsh_callback_t fields:
475  * fshc_arg - argument passed to the callbacks
476  * fshc_free - callback fired before VFS_FREEVFS() is called, after vfs_count
477  *      drops to 0
478  * fshc_mount - callback fired right before returning from domount()
479  * The first argument of these callbacks is the vfs_t that is mounted/freed.
480  * The second one is the fshc_arg.
481  *
482  * fsh_callback_handle_t is filled out by this function.
483  *
484  * This function must NOT be called in a callback, because it will cause
485  * a deadlock.
486  *
487  * Returns (-1) if hook/callback limit exceeded.
488  */
489 fsh_callback_handle_t
490 fsh_callback_install(fsh_callback_t *callback)
491 {
492         fsh_callback_int_t *fshci;
493         fsh_callback_handle_t handle;

495         if ((handle = id_alloc(fsh_idspace)) == -1)
496                 return (-1);

498         fshci = (fsh_callback_int_t *)kmem_alloc(sizeof (*fshci), KM_SLEEP);
499         (void) memcpy(&fshci->fshci_cb, callback, sizeof (fshci->fshci_cb));
500         fshci->fshci_handle = handle;

502         /* If it is called in a {mount,free} callback, causes deadlock. */
503         rw_enter(&fsh_cblist_lock, RW_WRITER);
504         list_insert_head(&fsh_cblist, fshci);
505         rw_exit(&fsh_cblist_lock);

507         return (handle);
508 }

510 /*
511  * API for removing global mount/free callbacks.
512  *
513  * This function must NOT be called in a callback, because it will cause
514  * a deadlock.
515  *
516  * Returns (-1) if callback wasn't found, 0 otherwise.
517  */
518 int
519 fsh_callback_remove(fsh_callback_handle_t handle)
520 {
521         fsh_callback_int_t *fshci;

523         /* If it is called in a {mount,free} callback, causes deadlock. */
```

```
524         rw_enter(&fsh_cblist_lock, RW_WRITER);
525         for (fshci = list_head(&fsh_cblist); fshci != NULL;
526             fshci = list_next(&fsh_cblist, fshci)) {
527                 if (fshci->fshci_handle == handle) {
528                         list_remove(&fsh_cblist, fshci);
529                         break;
530                 }
531         }
532         rw_exit(&fsh_cblist_lock);

534         if (fshci == NULL)
535                 return (-1);

537         kmem_free(fshci, sizeof (*fshci));
538         id_free(fsh_idspace, handle);

540         return (0);
541 }

543 /*
544  * This function is executed right before returning from domount()@vfs.c.
545  * We are sure that it's called only after fsh_init().
546  * It executes all the mount callbacks installed in the fsh.
547  */
548 void
549 fsh_exec_mount_callbacks(vfs_t *vfsp)
550 {
551         fsh_callback_int_t *fshci;
552         fsh_callback_t *cb;

554         rw_enter(&fsh_cblist_lock, RW_READER);
555         for (fshci = list_head(&fsh_cblist); fshci != NULL;
556             fshci = list_next(&fsh_cblist, fshci)) {
557                 cb = &fshci->fshci_cb;
558                 if (cb->fshc_mount != NULL)
559                         (*(cb->fshc_mount))(vfsp, cb->fshc_arg);
560         }
561         rw_exit(&fsh_cblist_lock);
562 }

564 /*
565  * This function is executed right before VFS_FREEVFS() is called in
566  * vfs_rele()@vfs.c. We are sure that it's called only after fsh_init().
567  * It executes all the free callbacks installed in the fsh.
568  */
569 void
570 fsh_exec_free_callbacks(vfs_t *vfsp)
571 {
572         fsh_callback_int_t *fshci;
573         fsh_callback_t *cb;

575         rw_enter(&fsh_cblist_lock, RW_READER);
576         for (fshci = list_head(&fsh_cblist); fshci != NULL;
577             fshci = list_next(&fsh_cblist, fshci)) {
578                 cb = &fshci->fshci_cb;
579                 if (cb->fshc_free != NULL)
580                         (*(cb->fshc_free))(vfsp, cb->fshc_arg);
581         }
582         rw_exit(&fsh_cblist_lock);
583 }

585 /*
586  * API for vnode.c/vfs.c to start executing the fsh for a given operation.
587  *
588  * These interfaces are using fsh_res_ptr (in fsh_prepare_fsrec()), so it's
589  * absolutely necessary to call fsh_init() before using them. That's done in
```



```
590  * vfsinit().
591  *
592  * While these functions are executing, it's expected that necessary vfs_t's
593  * are held so that vfs_free() isn't called. vfs_free() expects that noone
594  * else accesses vfs_fshrecord of a given vfs_t.
595  * It's also the caller responsibility to keep vnode_t passed to fsh_foo()
596  * alive and valid.
597  * All these expectations are met because these functions are used only in
598  * correspondng {fop,fsop}_foo() functions.
599  */
600 int
601 fsh_read(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
602         caller_context_t *ct)
603 {
604         int ret;
605         fsh_fsrecord_t *fsrecp;

607         fsh_prepare_fsrec(vp->v_vfsp);
608         fsrecp = vp->v_vfsp->vfs_fshrecord;

610         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
611         if (!(fsrecp->fshfsr_enabled)) {
612                 rw_exit(&fsrecp->fshfsr_lock);
613                 return ((*(vp->v_op->vop_read))(vp, uiop, ioflag, cr, ct));
614         }

616         ret = fsh_next_read(list_head(&fsrecp->fshfsr_list), vp, uiop, ioflag,
617             cr, ct);
618         rw_exit(&fsrecp->fshfsr_lock);

620         return (ret);
621 }

623 int
624 fsh_write(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
625         caller_context_t *ct)
626 {
627         int ret;
628         fsh_fsrecord_t *fsrecp;

630         fsh_prepare_fsrec(vp->v_vfsp);
631         fsrecp = vp->v_vfsp->vfs_fshrecord;

633         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
634         if (!(vp->v_vfsp->vfs_fshrecord->fshfsr_enabled)) {
635                 rw_exit(&fsrecp->fshfsr_lock);
636                 return ((*(vp->v_op->vop_write))(vp, uiop, ioflag, cr, ct));
637         }

639         ret = fsh_next_write(list_head(&fsrecp->fshfsr_list), vp, uiop, ioflag,
640             cr, ct);
641         rw_exit(&fsrecp->fshfsr_lock);

643         return (ret);
644 }

646 int
647 fsh_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
648 {
649         fsh_fsrecord_t *fsrecp;
650         int ret;

652         fsh_prepare_fsrec(vfsp);
653         fsrecp = vfsp->vfs_fshrecord;

655         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
```

```
656         if (!(fsrecp->fshfsr_enabled)) {
657                 rw_exit(&fsrecp->fshfsr_lock);
658                 return ((*(vfsp->vfs_op->vfs_mount))(vfsp, mvp, uap, cr));
659         }

661         ret = fsh_next_mount(list_head(&fsrecp->fshfsr_list), vfsp, mvp, uap,
662             cr);
663         rw_exit(&fsrecp->fshfsr_lock);

665         return (ret);
666 }

668 int
669 fsh_unmount(vfs_t *vfsp, int flag, cred_t *cr)
670 {
671         fsh_fsrecord_t *fsrecp;
672         int ret;

674         fsh_prepare_fsrec(vfsp);
675         fsrecp = vfsp->vfs_fshrecord;

677         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
678         if (!(fsrecp->fshfsr_enabled)) {
679                 rw_exit(&fsrecp->fshfsr_lock);
680                 return ((*(vfsp->vfs_op->vfs_unmount))(vfsp, flag, cr));
681         }

683         ret = fsh_next_unmount(list_head(&fsrecp->fshfsr_list), vfsp, flag, cr);
684         rw_exit(&fsrecp->fshfsr_lock);

686         return (ret);
687 }

689 /*
690  * This is the funtion used by fsh_prepare_fsrec() to allocate a new
691  * fsh_fsrecord. This function is called by the first function which
692  * access the vfs_fshrecord and finds out it's NULL.
693  */
694 static fsh_fsrecord_t *
695 fsh_fsrec_create()
696 {
697         fsh_fsrecord_t *fsrecp;

699         fsrecp = (fsh_fsrecord_t *)kmem_zalloc(sizeof (*fsrecp), KM_SLEEP);
700         list_create(&fsrecp->fshfsr_list, sizeof (fsh_int_t),
701             offsetof(fsh_int_t, fshi_next));
702         rw_init(&fsrecp->fshfsr_lock, NULL, RW_DRIVER, NULL);
703         fsrecp->fshfsr_enabled = 1;
704         return (fsrecp);
705 }

707 /*
708  * This call can be used ONLY in vfs_free(). It's assumed that no other
709  * fsh_foo() using the vfs_t that owns the fsh_fsrecord to be destroyed
710  * is executing while call to fsh_fsrec_destroy() is made. With this
711  * assumptions, no concurrency issues occur.
712  *
713  * Before calling this function outside the fsh, it's sufficient and
714  * requited to check if the passed fsh_fsrecord * is not NULL. We don't
715  * have to check if it is not equal to fsh_res_ptr, because all the fsh API
716  * calls involving this vfs_t should end before vfs_free() is called
717  * (outside the fsh, fsh_fsrecord is never equal to fsh_res_ptr). That is
718  * guaranteed by the explicit requirement that the caller of fsh API holds
719  * the vfs_t when needed.
720  *
721  * All the remaining hooks are being removed.
```



```
722  */
723 void
724 fsh_fsrec_destroy(struct fsh_fsrecord *volatile fsrecp)
725 {
726         fsh_int_t *fshi;

728         ASSERT(fsrecp != NULL);

730         /*
731          * TODO: We could add fshi_mapping field to fsh_int_t and make the
732          * process of removing the hooks faster. fsh_remove_mapping() works in
733          * linear time, because it has to find the element that matches the
734          * handle. It doesn't have to be this way, because the fsh_int_t
735          * could have contained the fshi_mapping field pointed to the
736          * corresponding fsh_mapping_t.
737          * Since we don't really know if it's going to be a performance
738          * issue, we leave it like this for the sake of simplicity.
739          */
740         while ((fshi = list_remove_head(&fsrecp->fshfsr_list)) != NULL) {
741                 /*
742                  * It's a simpler version of the code present in
743                  * fsh_hook_remove(). There is no fsh_fshrecord locking here,
744                  * because it is not needed anymore.
745                  */
746                 fsh_mapping_t *mapping;
747                 fsh_t *hooks;

749                 mapping = fsh_remove_mapping(fshi->fshi_handle);
750                 /*
751                  * mapping == NULL was acceptable in fsh_hook_remove()
752                  * because an invalid handle could have been passed by the
753                  * client. This time, we take the handle from an internal
754                  * structure, so it definitely has to be present in the
755                  * fsh_map.
756                  */
757                 ASSERT(mapping != NULL);
758                 ASSERT(mapping->fshm_handle == fshi->fshi_handle);
759                 ASSERT(mapping->fshm_fshi == fshi);

761                 hooks = &mapping->fshm_fshi->fshi_hooks;
762                 if (hooks->remove_callback != NULL)
763                         (*(hooks->remove_callback))(hooks->arg,
764                             mapping->fshm_handle);

766                 id_free(fsh_idspace, fshi->fshi_handle);

768                 kmem_free(fshi, sizeof (*fshi));
769                 kmem_free(mapping, sizeof (*mapping));
770         }
771         list_destroy(&fsrecp->fshfsr_list);
772         rw_destroy(&fsrecp->fshfsr_lock);
773         kmem_free(fsrecp, sizeof (*fsrecp));
774 }

776 /*
777  * fsh_init() is called in vfsinit()@vfs.c. This function MUST be called
778  * before every other fsh call.
779  */
780 void
781 fsh_init(void)
782 {
783         rw_init(&fsh_cblist_lock, NULL, RW_DRIVER, NULL);
784         list_create(&fsh_cblist, sizeof (fsh_callback_int_t),
785             offsetof(fsh_callback_int_t, fshci_next));

787         mutex_init(&fsh_map_lock, NULL, MUTEX_DRIVER, NULL);
```

```
788         list_create(&fsh_map, sizeof (fsh_mapping_t),
789             offsetof(fsh_mapping_t, fshm_next));
790
791         /* See comment above fsh_prepare_fsrec() */
792         fsh_res_ptr = (void *)-1;
793
794         fsh_idspace = id_space_create("fsh", 0, fsh_limit);
795 }
796
797 /*
798  * These functions are used to pass control to the next hook or underlying
799  * vop or vfsop. It's consumer doesn't have to worry about any locking, because
800  * all the necessities are guaranteed by the fsh_foo().
801  *
802  * In fsh_next_foo() we execute the hook passed in the first argument and
803  * try to find the next one. It is guaranteed that the passed hook is still
804  * valid, because of fshfsr_lock held by fsh_foo().
805  */
806 int
807 fsh_next_read(fsh_int_t *fshi, vnode_t *vp, uio_t *uiop, int ioflag,
808         cred_t *cr, caller_context_t *ct)
809 {
810         fsh_int_t       *next;
811
812         if (fshi == NULL)
813                 return ((*(vp->v_op->vop_read))(vp, uiop, ioflag, cr, ct));
814
815         next = fshi;
816         do {
817                 next = list_next(&vp->v_vfsp->vfs_fshrecord->fshfsr_list,
818                     next);
819         } while (next != NULL && next->fshi_hooks.read == NULL);
820
821         return ((*(fshi->fshi_hooks.read))(next, fshi->fshi_hooks.arg, vp,
822             uiop, ioflag, cr, ct));
823 }
824
825 int
826 fsh_next_write(fsh_int_t *fshi, vnode_t *vp, uio_t *uiop, int ioflag,
827         cred_t *cr, caller_context_t *ct)
828 {
829         fsh_int_t       *next;
830
831         if (fshi == NULL)
832                 return ((*(vp->v_op->vop_write))(vp, uiop, ioflag, cr, ct));
833
834         next = fshi;
835         do {
836                 next = list_next(&vp->v_vfsp->vfs_fshrecord->fshfsr_list, next);
837         } while (next != NULL && next->fshi_hooks.write == NULL);
838
839         return ((*(fshi->fshi_hooks.write))(next, fshi->fshi_hooks.arg, vp,
840             uiop, ioflag, cr, ct));
841 }
842
843 int
844 fsh_next_mount(fsh_int_t *fshi, vfs_t *vfsp, vnode_t *mvp, struct mounta *uap,
845         cred_t *cr)
846 {
847         fsh_int_t       *next;
848
849         if (fshi == NULL)
850                 return ((*(vfsp->vfs_op->vfs_mount))(vfsp, mvp, uap, cr));
851
852         next = fshi;
853         do {
```



```
854                 next = list_next(&vfsp->vfs_fshrecord->fshfsr_list, next);
855         } while (next != NULL && next->fshi_hooks.mount == NULL);
856
857         return ((*(fshi->fshi_hooks.mount))(next, fshi->fshi_hooks.arg, vfsp,
858             mvp, uap, cr));
859 }
860
861 int
862 fsh_next_unmount(fsh_int_t *fshi, vfs_t *vfsp, int flag, cred_t *cr)
863 {
864         fsh_int_t       *next;
865
866         if (fshi == NULL)
867                 return ((*(vfsp->vfs_op->vfs_unmount))(vfsp, flag, cr));
868
869         next = fshi;
870         do {
871                 next = list_next(&vfsp->vfs_fshrecord->fshfsr_list, next);
872         } while (next != NULL && next->fshi_hooks.unmount == NULL);
873
874         return ((*(fshi->fshi_hooks.unmount))(next, fshi->fshi_hooks.arg,
875             vfsp, flag, cr));
876 }
```

```
**********************************************************
  118568 Mon Aug 19 16:40:53 2013
new/usr/src/uts/common/fs/vfs.c
filesystem hook framework (August 19th)
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
  23  */

  25 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
  26 /*        All Rights Reserved   */

  28 /*
  29  * University Copyright- Copyright (c) 1982, 1986, 1988
  30  * The Regents of the University of California
  31  * All Rights Reserved
  32  *
  33  * University Acknowledgment- Portions of this document are derived from
  34  * software developed by the University of California, Berkeley, and its
  35  * contributors.
  36  */

  38 #include <sys/types.h>
  39 #include <sys/t_lock.h>
  40 #include <sys/param.h>
  41 #include <sys/errno.h>
  42 #include <sys/user.h>
  43 #include <sys/fstyp.h>
  44 #include <sys/kmem.h>
  45 #include <sys/systm.h>
  46 #include <sys/proc.h>
  47 #include <sys/mount.h>
  48 #include <sys/vfs.h>
  49 #include <sys/vfs_opreg.h>
  50 #include <sys/fem.h>
  51 #include <sys/mntent.h>
  52 #include <sys/stat.h>
  53 #include <sys/statvfs.h>
  54 #include <sys/statfs.h>
  55 #include <sys/cred.h>
  56 #include <sys/vnode.h>
  57 #include <sys/rwstlock.h>
  58 #include <sys/dnlc.h>
  59 #include <sys/file.h>
  60 #include <sys/time.h>
  61 #include <sys/atomic.h>
```



```
  62 #include <sys/cmn_err.h>
  63 #include <sys/buf.h>
  64 #include <sys/swap.h>
  65 #include <sys/debug.h>
  66 #include <sys/vnode.h>
  67 #include <sys/modctl.h>
  68 #include <sys/ddi.h>
  69 #include <sys/pathname.h>
  70 #include <sys/bootconf.h>
  71 #include <sys/dumphdr.h>
  72 #include <sys/dc_ki.h>
  73 #include <sys/poll.h>
  74 #include <sys/sunddi.h>
  75 #include <sys/sysmacros.h>
  76 #include <sys/zone.h>
  77 #include <sys/policy.h>
  78 #include <sys/ctfs.h>
  79 #include <sys/objfs.h>
  80 #include <sys/console.h>
  81 #include <sys/reboot.h>
  82 #include <sys/attr.h>
  83 #include <sys/zio.h>
  84 #include <sys/spa.h>
  85 #include <sys/lofi.h>
  86 #include <sys/bootprops.h>
  87 #include <sys/fsh.h>
  88 #include <sys/fsh_impl.h>

  90 #include <vm/page.h>

  92 #include <fs/fs_subr.h>
  93 /* Private interfaces to create vopstats-related data structures */
  94 extern void             initialize_vopstats(vopstats_t *);
  95 extern vopstats_t       *get_fstype_vopstats(struct vfs *, struct vfssw *);
  96 extern vsk_anchor_t     *get_vskstat_anchor(struct vfs *);

  98 static void vfs_clearmntopt_nolock(mntopts_t *, const char *, int);
  99 static void vfs_setmntopt_nolock(mntopts_t *, const char *,
 100     const char *, int, int);
 101 static int  vfs_optionisset_nolock(const mntopts_t *, const char *, char **);
 102 static void vfs_freemnttab(struct vfs *);
 103 static void vfs_freeopt(mntopt_t *);
 104 static void vfs_swapopttbl_nolock(mntopts_t *, mntopts_t *);
 105 static void vfs_swapopttbl(mntopts_t *, mntopts_t *);
 106 static void vfs_copyopttbl_extend(const mntopts_t *, mntopts_t *, int);
 107 static void vfs_createopttbl_extend(mntopts_t *, const char *,
 108     const mntopts_t *);
 109 static char **vfs_copycancelopt_extend(char **const, int);
 110 static void vfs_freecancelopt(char **);
 111 static void getrootfs(char **, char **);
 112 static int getmacpath(dev_info_t *, void *);
 113 static void vfs_mnttabvp_setup(void);

 115 struct ipmnt {
 116         struct ipmnt    *mip_next;
 117         dev_t           mip_dev;
 118         struct vfs      *mip_vfsp;
 119 };
_____unchanged_portion_omitted_

 214 /*
 215  * File system operation dispatch functions.
 216  */

 218 int
 219 fsop_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
```

```
 220 {
 221         return (fsh_mount(vfsp, mvp, uap, cr));
 219         return (*(vfsp)->vfs_op->vfs_mount)(vfsp, mvp, uap, cr);
 222 }

 224 int
 225 fsop_unmount(vfs_t *vfsp, int flag, cred_t *cr)
 226 {
 227         return (fsh_unmount(vfsp, flag, cr));
 225         return (*(vfsp)->vfs_op->vfs_unmount)(vfsp, flag, cr);
 228 }
______unchanged_portion_omitted_

1085 /*
1086  * Common mount code.  Called from the system call entry point, from autofs,
1087  * nfsv4 trigger mounts, and from pxfs.
1088  *
1089  * Takes the effective file system type, mount arguments, the mount point
1090  * vnode, flags specifying whether the mount is a remount and whether it
1091  * should be entered into the vfs list, and credentials.  Fills in its vfspp
1092  * parameter with the mounted file system instance's vfs.
1093  *
1094  * Note that the effective file system type is specified as a string.  It may
1095  * be null, in which case it's determined from the mount arguments, and may
1096  * differ from the type specified in the mount arguments; this is a hook to
1097  * allow interposition when instantiating file system instances.
1098  *
1099  * The caller is responsible for releasing its own hold on the mount point
1100  * vp (this routine does its own hold when necessary).
1101  * Also note that for remounts, the mount point vp should be the vnode for
1102  * the root of the file system rather than the vnode that the file system
1103  * is mounted on top of.
1104  */
1105 int
1106 domount(char *fsname, struct mounta *uap, vnode_t *vp, struct cred *credp,
1107         struct vfs **vfspp)
1108 {
1109         struct vfssw    *vswp;
1110         vfsops_t        *vfsops;
1111         struct vfs      *vfsp;
1112         struct vnode    *bvp;
1113         dev_t           bdev = 0;
1114         mntopts_t       mnt_mntopts;
1115         int             error = 0;
1116         int             copyout_error = 0;
1117         int             ovflags;
1118         char            *opts = uap->optptr;
1119         char            *inargs = opts;
1120         int             optlen = uap->optlen;
1121         int             remount;
1122         int             rdonly;
1123         int             nbmand = 0;
1124         int             delmip = 0;
1125         int             addmip = 0;
1126         int             splice = ((uap->flags & MS_NOSPLICE) == 0);
1127         int             fromspace = (uap->flags & MS_SYSSPACE) ?
1128             UIO_SYSSPACE : UIO_USERSPACE;
1129         char            *resource = NULL, *mountpt = NULL;
1130         refstr_t        *oldresource, *oldmntpt;
1131         struct pathname pn, rpn;
1132         vsk_anchor_t    *vskap;
1133         char fstname[FSTYPSZ];

1135         /*
1136          * The v_flag value for the mount point vp is permanently set
1137          * to VVFSLOCK so that no one bypasses the vn_vfs*locks routine
```



```
1138          * for mount point locking.
1139          */
1140         mutex_enter(&vp->v_lock);
1141         vp->v_flag |= VVFSLOCK;
1142         mutex_exit(&vp->v_lock);

1144         mnt_mntopts.mo_count = 0;
1145         /*
1146          * Find the ops vector to use to invoke the file system-specific mount
1147          * method.  If the fsname argument is non-NULL, use it directly.
1148          * Otherwise, dig the file system type information out of the mount
1149          * arguments.
1150          *
1151          * A side effect is to hold the vfssw entry.
1152          *
1153          * Mount arguments can be specified in several ways, which are
1154          * distinguished by flag bit settings.  The preferred way is to set
1155          * MS_OPTIONSTR, indicating an 8 argument mount with the file system
1156          * type supplied as a character string and the last two arguments
1157          * being a pointer to a character buffer and the size of the buffer.
1158          * On entry, the buffer holds a null terminated list of options; on
1159          * return, the string is the list of options the file system
1160          * recognized. If MS_DATA is set arguments five and six point to a
1161          * block of binary data which the file system interprets.
1162          * A further wrinkle is that some callers don't set MS_FSS and MS_DATA
1163          * consistently with these conventions.  To handle them, we check to
1164          * see whether the pointer to the file system name has a numeric value
1165          * less than 256.  If so, we treat it as an index.
1166          */
1167         if (fsname != NULL) {
1168                 if ((vswp = vfs_getvfssw(fsname)) == NULL) {
1169                         return (EINVAL);
1170                 }
1171         } else if (uap->flags & (MS_OPTIONSTR | MS_DATA | MS_FSS)) {
1172                 size_t n;
1173                 uint_t fstype;

1175                 fsname = fstname;

1177                 if ((fstype = (uintptr_t)uap->fstype) < 256) {
1178                         RLOCK_VFSSW();
1179                         if (fstype == 0 || fstype >= nfstype ||
1180                             !ALLOCATED_VFSSW(&vfssw[fstype])) {
1181                                 RUNLOCK_VFSSW();
1182                                 return (EINVAL);
1183                         }
1184                         (void) strcpy(fsname, vfssw[fstype].vsw_name);
1185                         RUNLOCK_VFSSW();
1186                         if ((vswp = vfs_getvfssw(fsname)) == NULL)
1187                                 return (EINVAL);
1188                 } else {
1189                         /*
1190                          * Handle either kernel or user address space.
1191                          */
1192                         if (uap->flags & MS_SYSSPACE) {
1193                                 error = copystr(uap->fstype, fsname,
1194                                     FSTYPSZ, &n);
1195                         } else {
1196                                 error = copyinstr(uap->fstype, fsname,
1197                                     FSTYPSZ, &n);
1198                         }
1199                         if (error) {
1200                                 if (error == ENAMETOOLONG)
1201                                         return (EINVAL);
1202                                 return (error);
1203                         }
```

```
1204                        if ((vswp = vfs_getvfssw(fsname)) == NULL)
1205                                return (EINVAL);
1206                }
1207        } else {
1208                if ((vswp = vfs_getvfsswbyvfsops(vfs_getops(rootvfs))) == NULL)
1209                        return (EINVAL);
1210                fsname = vswp->vsw_name;
1211        }
1212        if (!VFS_INSTALLED(vswp))
1213                return (EINVAL);

1215        if ((error = secpolicy_fs_allowed_mount(fsname)) != 0)  {
1216                vfs_unrefvfssw(vswp);
1217                return (error);
1218        }

1220        vfsops = &vswp->vsw_vfsops;

1222        vfs_copyopttbl(&vswp->vsw_optproto, &mnt_mntopts);
1223        /*
1224         * Fetch mount options and parse them for generic vfs options
1225         */
1226        if (uap->flags & MS_OPTIONSTR) {
1227                /*
1228                 * Limit the buffer size
1229                 */
1230                if (optlen < 0 || optlen > MAX_MNTOPT_STR) {
1231                        error = EINVAL;
1232                        goto errout;
1233                }
1234                if ((uap->flags & MS_SYSSPACE) == 0) {
1235                        inargs = kmem_alloc(MAX_MNTOPT_STR, KM_SLEEP);
1236                        inargs[0] = '\0';
1237                        if (optlen) {
1238                                error = copyinstr(opts, inargs, (size_t)optlen,
1239                                    NULL);
1240                                if (error) {
1241                                        goto errout;
1242                                }
1243                        }
1244                }
1245                vfs_parsemntopts(&mnt_mntopts, inargs, 0);
1246        }
1247        /*
1248         * Flag bits override the options string.
1249         */
1250        if (uap->flags & MS_REMOUNT)
1251                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_REMOUNT, NULL, 0, 0);
1252        if (uap->flags & MS_RDONLY)
1253                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_RO, NULL, 0, 0);
1254        if (uap->flags & MS_NOSUID)
1255                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL, 0, 0);

1257        /*
1258         * Check if this is a remount; must be set in the option string and
1259         * the file system must support a remount option.
1260         */
1261        if (remount = vfs_optionisset_nolock(&mnt_mntopts,
1262            MNTOPT_REMOUNT, NULL)) {
1263                if (!(vswp->vsw_flag & VSW_CANREMOUNT)) {
1264                        error = ENOTSUP;
1265                        goto errout;
1266                }
1267                uap->flags |= MS_REMOUNT;
1268        }
```



```
1270        /*
1271         * uap->flags and vfs_optionisset() should agree.
1272         */
1273        if (rdonly = vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_RO, NULL)) {
1274                uap->flags |= MS_RDONLY;
1275        }
1276        if (vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL)) {
1277                uap->flags |= MS_NOSUID;
1278        }
1279        nbmand = vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_NBMAND, NULL);
1280        ASSERT(splice || !remount);
1281        /*
1282         * If we are splicing the fs into the namespace,
1283         * perform mount point checks.
1284         *
1285         * We want to resolve the path for the mount point to eliminate
1286         * '.' and ".." and symlinks in mount points; we can't do the
1287         * same for the resource string, since it would turn
1288         * "/dev/dsk/c0t0d0s0" into "/devices/pci@...".  We need to do
1289         * this before grabbing vn_vfswlock(), because otherwise we
1290         * would deadlock with lookuppn().
1291         */
1292        if (splice) {
1293                ASSERT(vp->v_count > 0);

1295                /*
1296                 * Pick up mount point and device from appropriate space.
1297                 */
1298                if (pn_get(uap->spec, fromspace, &pn) == 0) {
1299                        resource = kmem_alloc(pn.pn_pathlen + 1,
1300                            KM_SLEEP);
1301                        (void) strcpy(resource, pn.pn_path);
1302                        pn_free(&pn);
1303                }
1304                /*
1305                 * Do a lookupname prior to taking the
1306                 * writelock. Mark this as completed if
1307                 * successful for later cleanup and addition to
1308                 * the mount in progress table.
1309                 */
1310                if ((uap->flags & MS_GLOBAL) == 0 &&
1311                    lookupname(uap->spec, fromspace,
1312                    FOLLOW, NULL, &bvp) == 0) {
1313                        addmip = 1;
1314                }

1316                if ((error = pn_get(uap->dir, fromspace, &pn)) == 0) {
1317                        pathname_t *pnp;

1319                        if (*pn.pn_path != '/') {
1320                                error = EINVAL;
1321                                pn_free(&pn);
1322                                goto errout;
1323                        }
1324                        pn_alloc(&rpn);
1325                        /*
1326                         * Kludge to prevent autofs from deadlocking with
1327                         * itself when it calls domount().
1328                         *
1329                         * If autofs is calling, it is because it is doing
1330                         * (autofs) mounts in the process of an NFS mount.  A
1331                         * lookuppn() here would cause us to block waiting for
1332                         * said NFS mount to complete, which can't since this
1333                         * is the thread that was supposed to doing it.
1334                         */
1335                        if (fromspace == UIO_USERSPACE) {
```

```
1336                                if ((error = lookuppn(&pn, &rpn, FOLLOW, NULL,
1337                                    NULL)) == 0) {
1338                                        pnp = &rpn;
1339                                } else {
1340                                        /*
1341                                         * The file disappeared or otherwise
1342                                         * became inaccessible since we opened
1343                                         * it; might as well fail the mount
1344                                         * since the mount point is no longer
1345                                         * accessible.
1346                                         */
1347                                        pn_free(&rpn);
1348                                        pn_free(&pn);
1349                                        goto errout;
1350                                }
1351                        } else {
1352                                pnp = &pn;
1353                        }
1354                        mountpt = kmem_alloc(pnp->pn_pathlen + 1, KM_SLEEP);
1355                        (void) strcpy(mountpt, pnp->pn_path);

1357                        /*
1358                         * If the addition of the zone's rootpath
1359                         * would push us over a total path length
1360                         * of MAXPATHLEN, we fail the mount with
1361                         * ENAMETOOLONG, which is what we would have
1362                         * gotten if we were trying to perform the same
1363                         * mount in the global zone.
1364                         *
1365                         * strlen() doesn't count the trailing
1366                         * '\0', but zone_rootpathlen counts both a
1367                         * trailing '/' and the terminating '\0'.
1368                         */
1369                        if ((curproc->p_zone->zone_rootpathlen - 1 +
1370                            strlen(mountpt)) > MAXPATHLEN ||
1371                            (resource != NULL &&
1372                            (curproc->p_zone->zone_rootpathlen - 1 +
1373                            strlen(resource)) > MAXPATHLEN)) {
1374                                error = ENAMETOOLONG;
1375                        }

1377                        pn_free(&rpn);
1378                        pn_free(&pn);
1379                }

1381                if (error)
1382                        goto errout;

1384                /*
1385                 * Prevent path name resolution from proceeding past
1386                 * the mount point.
1387                 */
1388                if (vn_vfswlock(vp) != 0) {
1389                        error = EBUSY;
1390                        goto errout;
1391                }

1393                /*
1394                 * Verify that it's legitimate to establish a mount on
1395                 * the prospective mount point.
1396                 */
1397                if (vn_mountedvfs(vp) != NULL) {
1398                        /*
1399                         * The mount point lock was obtained after some
1400                         * other thread raced through and established a mount.
1401                         */
```



```
1402                        vn_vfsunlock(vp);
1403                        error = EBUSY;
1404                        goto errout;
1405                }
1406                if (vp->v_flag & VNOMOUNT) {
1407                        vn_vfsunlock(vp);
1408                        error = EINVAL;
1409                        goto errout;
1410                }
1411        }
1412        if ((uap->flags & (MS_DATA | MS_OPTIONSTR)) == 0) {
1413                uap->dataptr = NULL;
1414                uap->datalen = 0;
1415        }

1417        /*
1418         * If this is a remount, we don't want to create a new VFS.
1419         * Instead, we pass the existing one with a remount flag.
1420         */
1421        if (remount) {
1422                /*
1423                 * Confirm that the mount point is the root vnode of the
1424                 * file system that is being remounted.
1425                 * This can happen if the user specifies a different
1426                 * mount point directory pathname in the (re)mount command.
1427                 *
1428                 * Code below can only be reached if splice is true, so it's
1429                 * safe to do vn_vfsunlock() here.
1430                 */
1431                if ((vp->v_flag & VROOT) == 0) {
1432                        vn_vfsunlock(vp);
1433                        error = ENOENT;
1434                        goto errout;
1435                }
1436                /*
1437                 * Disallow making file systems read-only unless file system
1438                 * explicitly allows it in its vfssw.  Ignore other flags.
1439                 */
1440                if (rdonly && vn_is_readonly(vp) == 0 &&
1441                    (vswp->vsw_flag & VSW_CANRWRO) == 0) {
1442                        vn_vfsunlock(vp);
1443                        error = EINVAL;
1444                        goto errout;
1445                }
1446                /*
1447                 * Disallow changing the NBMAND disposition of the file
1448                 * system on remounts.
1449                 */
1450                if ((nbmand && ((vp->v_vfsp->vfs_flag & VFS_NBMAND) == 0)) ||
1451                    (!nbmand && (vp->v_vfsp->vfs_flag & VFS_NBMAND))) {
1452                        vn_vfsunlock(vp);
1453                        error = EINVAL;
1454                        goto errout;
1455                }
1456                vfsp = vp->v_vfsp;
1457                ovflags = vfsp->vfs_flag;
1458                vfsp->vfs_flag |= VFS_REMOUNT;
1459                vfsp->vfs_flag &= ~VFS_RDONLY;
1460        } else {
1461                vfsp = vfs_alloc(KM_SLEEP);
1462                VFS_INIT(vfsp, vfsops, NULL);
1463        }

1465        VFS_HOLD(vfsp);

1467        if ((error = lofi_add(fsname, vfsp, &mnt_mntopts, uap)) != 0) {
```

```
1468                if (!remount) {
1469                        if (splice)
1470                                vn_vfsunlock(vp);
1471                        vfs_free(vfsp);
1472                } else {
1473                        vn_vfsunlock(vp);
1474                        VFS_RELE(vfsp);
1475                }
1476                goto errout;
1477        }

1479        /*
1480         * PRIV_SYS_MOUNT doesn't mean you can become root.
1481         */
1482        if (vfsp->vfs_lofi_minor != 0) {
1483                uap->flags |= MS_NOSUID;
1484                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL, 0, 0);
1485        }

1487        /*
1488         * The vfs_reflock is not used anymore the code below explicitly
1489         * holds it preventing others accesing it directly.
1490         */
1491        if ((sema_tryp(&vfsp->vfs_reflock) == 0) &&
1492            !(vfsp->vfs_flag & VFS_REMOUNT))
1493                cmn_err(CE_WARN,
1494                    "mount type %s couldn't get vfs_reflock", vswp->vsw_name);

1496        /*
1497         * Lock the vfs. If this is a remount we want to avoid spurious umount
1498         * failures that happen as a side-effect of fsflush() and other mount
1499         * and unmount operations that might be going on simultaneously and
1500         * may have locked the vfs currently. To not return EBUSY immediately
1501         * here we use vfs_lock_wait() instead vfs_lock() for the remount case.
1502         */
1503        if (!remount) {
1504                if (error = vfs_lock(vfsp)) {
1505                        vfsp->vfs_flag = ovflags;

1507                        lofi_remove(vfsp);

1509                        if (splice)
1510                                vn_vfsunlock(vp);
1511                        vfs_free(vfsp);
1512                        goto errout;
1513                }
1514        } else {
1515                vfs_lock_wait(vfsp);
1516        }

1518        /*
1519         * Add device to mount in progress table, global mounts require special
1520         * handling. It is possible that we have already done the lookupname
1521         * on a spliced, non-global fs. If so, we don't want to do it again
1522         * since we cannot do a lookupname after taking the
1523         * wlock above. This case is for a non-spliced, non-global filesystem.
1524         */
1525        if (!addmip) {
1526                if ((uap->flags & MS_GLOBAL) == 0 &&
1527                    lookupname(uap->spec, fromspace, FOLLOW, NULL, &bvp) == 0) {
1528                        addmip = 1;
1529                }
1530        }

1532        if (addmip) {
1533                vnode_t *lvp = NULL;
```



```
1535                error = vfs_get_lofi(vfsp, &lvp);
1536                if (error > 0) {
1537                        lofi_remove(vfsp);

1539                        if (splice)
1540                                vn_vfsunlock(vp);
1541                        vfs_unlock(vfsp);

1543                        if (remount) {
1544                                VFS_RELE(vfsp);
1545                        } else {
1546                                vfs_free(vfsp);
1547                        }

1549                        goto errout;
1550                } else if (error == -1) {
1551                        bdev = bvp->v_rdev;
1552                        VN_RELE(bvp);
1553                } else {
1554                        bdev = lvp->v_rdev;
1555                        VN_RELE(lvp);
1556                        VN_RELE(bvp);
1557                }

1559                vfs_addmip(bdev, vfsp);
1560                addmip = 0;
1561                delmip = 1;
1562        }
1563        /*
1564         * Invalidate cached entry for the mount point.
1565         */
1566        if (splice)
1567                dnlc_purge_vp(vp);

1569        /*
1570         * If have an option string but the filesystem doesn't supply a
1571         * prototype options table, create a table with the global
1572         * options and sufficient room to accept all the options in the
1573         * string.  Then parse the passed in option string
1574         * accepting all the options in the string.  This gives us an
1575         * option table with all the proper cancel properties for the
1576         * global options.
1577         *
1578         * Filesystems that supply a prototype options table are handled
1579         * earlier in this function.
1580         */
1581        if (uap->flags & MS_OPTIONSTR) {
1582                if (!(vswp->vsw_flag & VSW_HASPROTO)) {
1583                        mntopts_t tmp_mntopts;

1585                        tmp_mntopts.mo_count = 0;
1586                        vfs_createopttbl_extend(&tmp_mntopts, inargs,
1587                            &mnt_mntopts);
1588                        vfs_parsemntopts(&tmp_mntopts, inargs, 1);
1589                        vfs_swapopttbl_nolock(&mnt_mntopts, &tmp_mntopts);
1590                        vfs_freeopttbl(&tmp_mntopts);
1591                }
1592        }

1594        /*
1595         * Serialize with zone creations.
1596         */
1597        mount_in_progress();
1598        /*
1599         * Instantiate (or reinstantiate) the file system.  If appropriate,
```

```
1600          * splice it into the file system name space.
1601          *
1602          * We want VFS_MOUNT() to be able to override the vfs_resource
1603          * string if necessary (ie, mntfs), and also for a remount to
1604          * change the same (necessary when remounting '/' during boot).
1605          * So we set up vfs_mntpt and vfs_resource to what we think they
1606          * should be, then hand off control to VFS_MOUNT() which can
1607          * override this.
1608          *
1609          * For safety's sake, when changing vfs_resource or vfs_mntpt of
1610          * a vfs which is on the vfs list (i.e. during a remount), we must
1611          * never set those fields to NULL. Several bits of code make
1612          * assumptions that the fields are always valid.
1613          */
1614         vfs_swapopttbl(&mnt_mntopts, &vfsp->vfs_mntopts);
1615         if (remount) {
1616                 if ((oldresource = vfsp->vfs_resource) != NULL)
1617                         refstr_hold(oldresource);
1618                 if ((oldmntpt = vfsp->vfs_mntpt) != NULL)
1619                         refstr_hold(oldmntpt);
1620         }
1621         vfs_setresource(vfsp, resource, 0);
1622         vfs_setmntpoint(vfsp, mountpt, 0);

1624         /*
1625          * going to mount on this vnode, so notify.
1626          */
1627         vnevent_mountedover(vp, NULL);
1628         error = VFS_MOUNT(vfsp, vp, uap, credp);

1630         if (uap->flags & MS_RDONLY)
1631                 vfs_setmntopt(vfsp, MNTOPT_RO, NULL, 0);
1632         if (uap->flags & MS_NOSUID)
1633                 vfs_setmntopt(vfsp, MNTOPT_NOSUID, NULL, 0);
1634         if (uap->flags & MS_GLOBAL)
1635                 vfs_setmntopt(vfsp, MNTOPT_GLOBAL, NULL, 0);

1637         if (error) {
1638                 lofi_remove(vfsp);

1640                 if (remount) {
1641                         /* put back pre-remount options */
1642                         vfs_swapopttbl(&mnt_mntopts, &vfsp->vfs_mntopts);
1643                         vfs_setmntpoint(vfsp, refstr_value(oldmntpt),
1644                             VFSSP_VERBATIM);
1645                         if (oldmntpt)
1646                                 refstr_rele(oldmntpt);
1647                         vfs_setresource(vfsp, refstr_value(oldresource),
1648                             VFSSP_VERBATIM);
1649                         if (oldresource)
1650                                 refstr_rele(oldresource);
1651                         vfsp->vfs_flag = ovflags;
1652                         vfs_unlock(vfsp);
1653                         VFS_RELE(vfsp);
1654                 } else {
1655                         vfs_unlock(vfsp);
1656                         vfs_freemnttab(vfsp);
1657                         vfs_free(vfsp);
1658                 }
1659         } else {
1660                 /*
1661                  * Set the mount time to now
1662                  */
1663                 vfsp->vfs_mtime = ddi_get_time();
1664                 if (remount) {
1665                         vfsp->vfs_flag &= ~VFS_REMOUNT;
```



```
1666                         if (oldresource)
1667                                 refstr_rele(oldresource);
1668                         if (oldmntpt)
1669                                 refstr_rele(oldmntpt);
1670                 } else if (splice) {
1671                         /*
1672                          * Link vfsp into the name space at the mount
1673                          * point. Vfs_add() is responsible for
1674                          * holding the mount point which will be
1675                          * released when vfs_remove() is called.
1676                          */
1677                         vfs_add(vp, vfsp, uap->flags);
1678                 } else {
1679                         /*
1680                          * Hold the reference to file system which is
1681                          * not linked into the name space.
1682                          */
1683                         vfsp->vfs_zone = NULL;
1684                         VFS_HOLD(vfsp);
1685                         vfsp->vfs_vnodecovered = NULL;
1686                 }
1687                 /*
1688                  * Set flags for global options encountered
1689                  */
1690                 if (vfs_optionisset(vfsp, MNTOPT_RO, NULL))
1691                         vfsp->vfs_flag |= VFS_RDONLY;
1692                 else
1693                         vfsp->vfs_flag &= ~VFS_RDONLY;
1694                 if (vfs_optionisset(vfsp, MNTOPT_NOSUID, NULL)) {
1695                         vfsp->vfs_flag |= (VFS_NOSETUID|VFS_NODEVICES);
1696                 } else {
1697                         if (vfs_optionisset(vfsp, MNTOPT_NODEVICES, NULL))
1698                                 vfsp->vfs_flag |= VFS_NODEVICES;
1699                         else
1700                                 vfsp->vfs_flag &= ~VFS_NODEVICES;
1701                         if (vfs_optionisset(vfsp, MNTOPT_NOSETUID, NULL))
1702                                 vfsp->vfs_flag |= VFS_NOSETUID;
1703                         else
1704                                 vfsp->vfs_flag &= ~VFS_NOSETUID;
1705                 }
1706                 if (vfs_optionisset(vfsp, MNTOPT_NBMAND, NULL))
1707                         vfsp->vfs_flag |= VFS_NBMAND;
1708                 else
1709                         vfsp->vfs_flag &= ~VFS_NBMAND;

1711                 if (vfs_optionisset(vfsp, MNTOPT_XATTR, NULL))
1712                         vfsp->vfs_flag |= VFS_XATTR;
1713                 else
1714                         vfsp->vfs_flag &= ~VFS_XATTR;

1716                 if (vfs_optionisset(vfsp, MNTOPT_NOEXEC, NULL))
1717                         vfsp->vfs_flag |= VFS_NOEXEC;
1718                 else
1719                         vfsp->vfs_flag &= ~VFS_NOEXEC;

1721                 /*
1722                  * Now construct the output option string of options
1723                  * we recognized.
1724                  */
1725                 if (uap->flags & MS_OPTIONSTR) {
1726                         vfs_list_read_lock();
1727                         copyout_error = vfs_buildoptionstr(
1728                             &vfsp->vfs_mntopts, inargs, optlen);
1729                         vfs_list_unlock();
1730                         if (copyout_error == 0 &&
1731                             (uap->flags & MS_SYSSPACE) == 0) {
```

```
1732                                    copyout_error = copyoutstr(inargs, opts,
1733                                        optlen, NULL);
1734                            }
1735                    }
1736
1737                    /*
1738                     * If this isn't a remount, set up the vopstats before
1739                     * anyone can touch this. We only allow spliced file
1740                     * systems (file systems which are in the namespace) to
1741                     * have the VFS_STATS flag set.
1742                     * NOTE: PxFS mounts the underlying file system with
1743                     * MS_NOSPLICE set and copies those vfs_flags to its private
1744                     * vfs structure. As a result, PxFS should never have
1745                     * the VFS_STATS flag or else we might access the vfs
1746                     * statistics-related fields prior to them being
1747                     * properly initialized.
1748                     */
1749                    if (!remount && (vswp->vsw_flag & VSW_STATS) && splice) {
1750                            initialize_vopstats(&vfsp->vfs_vopstats);
1751                            /*
1752                             * We need to set vfs_vskap to NULL because there's
1753                             * a chance it won't be set below.  This is checked
1754                             * in teardown_vopstats() so we can't have garbage.
1755                             */
1756                            vfsp->vfs_vskap = NULL;
1757                            vfsp->vfs_flag |= VFS_STATS;
1758                            vfsp->vfs_fstypevsp = get_fstype_vopstats(vfsp, vswp);
1759                    }
1760
1761                    if (vswp->vsw_flag & VSW_XID)
1762                            vfsp->vfs_flag |= VFS_XID;
1763
1764                    vfs_unlock(vfsp);
1765            }
1766            mount_completed();
1767            if (splice)
1768                    vn_vfsunlock(vp);
1769
1770            if ((error == 0) && (copyout_error == 0)) {
1771                    if (!remount) {
1772                            /*
1773                             * Don't call get_vskstat_anchor() while holding
1774                             * locks since it allocates memory and calls
1775                             * VFS_STATVFS().  For NFS, the latter can generate
1776                             * an over-the-wire call.
1777                             */
1778                            vskap = get_vskstat_anchor(vfsp);
1779                            /* Only take the lock if we have something to do */
1780                            if (vskap != NULL) {
1781                                    vfs_lock_wait(vfsp);
1782                                    if (vfsp->vfs_flag & VFS_STATS) {
1783                                            vfsp->vfs_vskap = vskap;
1784                                    }
1785                                    vfs_unlock(vfsp);
1786                            }
1787                    }
1788                    /* Return vfsp to caller. */
1789                    *vfspp = vfsp;
1790                    fsh_exec_mount_callbacks(vfsp);
1791            }
1792    errout:
1793            vfs_freeopttbl(&mnt_mntopts);
1794            if (resource != NULL)
1795                    kmem_free(resource, strlen(resource) + 1);
1796            if (mountpt != NULL)
1797                    kmem_free(mountpt, strlen(mountpt) + 1);
```



```
1798            /*
1799             * It is possible we errored prior to adding to mount in progress
1800             * table. Must free vnode we acquired with successful lookupname.
1801             */
1802            if (addmip)
1803                    VN_RELE(bvp);
1804            if (delmip)
1805                    vfs_delmip(vfsp);
1806            ASSERT(vswp != NULL);
1807            vfs_unrefvfssw(vswp);
1808            if (inargs != opts)
1809                    kmem_free(inargs, MAX_MNTOPT_STR);
1810            if (copyout_error) {
1811                    lofi_remove(vfsp);
1812                    VFS_RELE(vfsp);
1813                    error = copyout_error;
1814            }
1815            return (error);
1816    }
______unchanged_portion_omitted_

4191    vfs_t EIO_vfs;
4192    vfsops_t *EIO_vfsops;
4193
4194    /*
4195     * Called from startup() to initialize all loaded vfs's
4196     */
4197    void
4198    vfsinit(void)
4199    {
4200            struct vfssw *vswp;
4201            int error;
4202            extern int vopstats_enabled;
4203            extern void vopstats_startup();
4204
4205            static const fs_operation_def_t EIO_vfsops_template[] = {
4206                    VFSNAME_MOUNT,          { .error = vfs_EIO },
4207                    VFSNAME_UNMOUNT,        { .error = vfs_EIO },
4208                    VFSNAME_ROOT,           { .error = vfs_EIO },
4209                    VFSNAME_STATVFS,        { .error = vfs_EIO },
4210                    VFSNAME_SYNC,           { .vfs_sync = vfs_EIO_sync },
4211                    VFSNAME_VGET,           { .error = vfs_EIO },
4212                    VFSNAME_MOUNTROOT,      { .error = vfs_EIO },
4213                    VFSNAME_FREEVFS,        { .error = vfs_EIO },
4214                    VFSNAME_VNSTATE,        { .error = vfs_EIO },
4215                    NULL, NULL
4216            };
4217
4218            static const fs_operation_def_t stray_vfsops_template[] = {
4219                    VFSNAME_MOUNT,          { .error = vfsstray },
4220                    VFSNAME_UNMOUNT,        { .error = vfsstray },
4221                    VFSNAME_ROOT,           { .error = vfsstray },
4222                    VFSNAME_STATVFS,        { .error = vfsstray },
4223                    VFSNAME_SYNC,           { .vfs_sync = vfsstray_sync },
4224                    VFSNAME_VGET,           { .error = vfsstray },
4225                    VFSNAME_MOUNTROOT,      { .error = vfsstray },
4226                    VFSNAME_FREEVFS,        { .error = vfsstray },
4227                    VFSNAME_VNSTATE,        { .error = vfsstray },
4228                    NULL, NULL
4229            };
4230
4231            /* Create vfs cache */
4232            vfs_cache = kmem_cache_create("vfs_cache", sizeof (struct vfs),
4233                sizeof (uintptr_t), NULL, NULL, NULL, NULL, NULL, 0);
4234
4235            /* Initialize the vnode cache (file systems may use it during init). */
```

```
4236         vn_create_cache();

4238         /* Setup event monitor framework */
4239         fem_init();

4241         /* Setup filesystem hook framework */
4242         fsh_init();

4244         /* Initialize the dummy stray file system type. */
4245         error = vfs_setfsops(0, stray_vfsops_template, NULL);

4247         /* Initialize the dummy EIO file system. */
4248         error = vfs_makefsops(EIO_vfsops_template, &EIO_vfsops);
4249         if (error != 0) {
4250                 cmn_err(CE_WARN, "vfsinit: bad EIO vfs ops template");
4251                 /* Shouldn't happen, but not bad enough to panic */
4252         }

4254         VFS_INIT(&EIO_vfs, EIO_vfsops, (caddr_t)NULL);

4256         /*
4257          * Default EIO_vfs.vfs_flag to VFS_UNMOUNTED so a lookup
4258          * on this vfs can immediately notice it's invalid.
4259          */
4260         EIO_vfs.vfs_flag |= VFS_UNMOUNTED;

4262         /*
4263          * Call the init routines of non-loadable filesystems only.
4264          * Filesystems which are loaded as separate modules will be
4265          * initialized by the module loading code instead.
4266          */

4268         for (vswp = &vfssw[1]; vswp < &vfssw[nfstype]; vswp++) {
4269                 RLOCK_VFSSW();
4270                 if (vswp->vsw_init != NULL)
4271                         (*vswp->vsw_init)(vswp - vfssw, vswp->vsw_name);
4272                 RUNLOCK_VFSSW();
4273         }

4275         vopstats_startup();

4277         if (vopstats_enabled) {
4278                 /* EIO_vfs can collect stats, but we don't retrieve them */
4279                 initialize_vopstats(&EIO_vfs.vfs_vopstats);
4280                 EIO_vfs.vfs_fstypevsp = NULL;
4281                 EIO_vfs.vfs_vskap = NULL;
4282                 EIO_vfs.vfs_flag |= VFS_STATS;
4283         }

4285         xattr_init();

4287         reparse_point_init();
4288 }
```

*unchanged_portion_omitted_*

```
4305 void
4306 vfs_free(vfs_t *vfsp)
4307 {
4308         /*
4309          * One would be tempted to assert that "vfsp->vfs_count == 0".
4310          * The problem is that this gets called out of domount() with
4311          * a partially initialized vfs and a vfs_count of 1.  This is
4312          * also called from vfs_rele() with a vfs_count of 0.  We can't
4313          * call VFS_RELE() from domount() if VFS_MOUNT() hasn't successfully
4314          * returned.  This is because VFS_MOUNT() fully initializes the
4315          * vfs structure and its associated data.  VFS_RELE() will call
```



```
4316          * VFS_FREEVFS() which may panic the system if the data structures
4317          * aren't fully initialized from a successful VFS_MOUNT()).
4318          */

4320         /* If FEM was in use, make sure everything gets cleaned up */
4321         if (vfsp->vfs_femhead) {
4322                 ASSERT(vfsp->vfs_femhead->femh_list == NULL);
4323                 mutex_destroy(&vfsp->vfs_femhead->femh_lock);
4324                 kmem_free(vfsp->vfs_femhead, sizeof (*(vfsp->vfs_femhead)));
4325                 vfsp->vfs_femhead = NULL;
4326         }

4328         /*
4329          * fsh cleanup
4330          * There's no need here to use atomic operations on vfs_fshrecord.
4331          */
4332         if (vfsp->vfs_fshrecord != NULL) {
4333                 fsh_fsrec_destroy(vfsp->vfs_fshrecord);
4334                 vfsp->vfs_fshrecord = NULL;
4335         }

4337         if (vfsp->vfs_implp)
4338                 vfsimpl_teardown(vfsp);
4339         sema_destroy(&vfsp->vfs_reflock);
4340         kmem_cache_free(vfs_cache, vfsp);
4341 }
```

*unchanged_portion_omitted_*

```
4353 /*
4354  * Decrements the vfs reference count by one atomically. When
4355  * vfs reference count becomes zero, it calls the file system
4356  * specific vfs_freevfs() to free up the resources.
4357  */
4358 void
4359 vfs_rele(vfs_t *vfsp)
4360 {
4361         ASSERT(vfsp->vfs_count != 0);
4362         if (atomic_add_32_nv(&vfsp->vfs_count, -1) == 0) {
4363                 fsh_exec_free_callbacks(vfsp);
4364                 VFS_FREEVFS(vfsp);
4365                 lofi_remove(vfsp);
4366                 if (vfsp->vfs_zone)
4367                         zone_rele_ref(&vfsp->vfs_implp->vi_zone_ref,
4368                             ZONE_REF_VFS);
4369                 vfs_freemnttab(vfsp);
4370                 vfs_free(vfsp);
4371         }
4372 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  105122 Mon Aug 19 16:40:54 2013
new/usr/src/uts/common/fs/vnode.c
filesystem hook framework (August 19th)
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/uio.h>
#include <sys/file.h>
#include <sys/pathname.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/vnode.h>
#include <sys/rwstlock.h>
#include <sys/fem.h>
#include <sys/stat.h>
#include <sys/mode.h>
#include <sys/conf.h>
#include <sys/sysmacros.h>
#include <sys/cmn_err.h>
#include <sys/systm.h>
#include <sys/kmem.h>
#include <sys/debug.h>
#include <c2/audit.h>
```



```
#include <sys/acl.h>
#include <sys/nbmlock.h>
#include <sys/fcntl.h>
#include <fs/fs_subr.h>
#include <sys/taskq.h>
#include <fs/fs_reparse.h>
#include <sys/fsh_impl.h>

/* Determine if this vnode is a file that is read-only */
#define ISROFILE(vp)    \
        ((vp)->v_type != VCHR && (vp)->v_type != VBLK && \
            (vp)->v_type != VFIFO && vn_is_readonly(vp))

/* Tunable via /etc/system; used only by admin/install */
int nfs_global_client_only;

/*
 * Array of vopstats_t for per-FS-type vopstats.  This array has the same
 * number of entries as and parallel to the vfssw table.  (Arguably, it could
 * be part of the vfssw table.)  Once it's initialized, it's accessed using
 * the same fstype index that is used to index into the vfssw table.
 */
vopstats_t **vopstats_fstype;

/* vopstats initialization template used for fast initialization via bcopy() */
static vopstats_t *vs_templatep;

/* Kmem cache handle for vsk_anchor_t allocations */
kmem_cache_t *vsk_anchor_cache;

/* file events cleanup routine */
extern void free_fopdata(vnode_t *);

/*
 * Root of AVL tree for the kstats associated with vopstats.  Lock protects
 * updates to vsktat_tree.
 */
avl_tree_t      vskstat_tree;
kmutex_t        vskstat_tree_lock;

/* Global variable which enables/disables the vopstats collection */
int vopstats_enabled = 1;

/*
 * forward declarations for internal vnode specific data (vsd)
 */
static void *vsd_realloc(void *, size_t, size_t);

/*
 * forward declarations for reparse point functions
 */
static int fs_reparse_mark(char *target, vattr_t *vap, xvattr_t *xvattr);

/*
 * VSD -- VNODE SPECIFIC DATA
 * The v_data pointer is typically used by a file system to store a
 * pointer to the file system's private node (e.g. ufs inode, nfs rnode).
 * However, there are times when additional project private data needs
 * to be stored separately from the data (node) pointed to by v_data.
 * This additional data could be stored by the file system itself or
 * by a completely different kernel entity.  VSD provides a way for
 * callers to obtain a key and store a pointer to private data associated
 * with a vnode.
 *
 * Callers are responsible for protecting the vsd by holding v_vsd_lock
 * for calls to vsd_set() and vsd_get().
```

```
 128  */

 130 /*
 131  * vsd_lock protects:
 132  *   vsd_nkeys - creation and deletion of vsd keys
 133  *   vsd_list - insertion and deletion of vsd_node in the vsd_list
 134  *   vsd_destructor - adding and removing destructors to the list
 135  */
 136 static kmutex_t         vsd_lock;
 137 static uint_t           vsd_nkeys;       /* size of destructor array */
 138 /* list of vsd_node's */
 139 static list_t *vsd_list = NULL;
 140 /* per-key destructor funcs */
 141 static void             (**vsd_destructor)(void *);

 143 /*
 144  * The following is the common set of actions needed to update the
 145  * vopstats structure from a vnode op.  Both VOPSTATS_UPDATE() and
 146  * VOPSTATS_UPDATE_IO() do almost the same thing, except for the
 147  * recording of the bytes transferred.  Since the code is similar
 148  * but small, it is nearly a duplicate.  Consequently any changes
 149  * to one may need to be reflected in the other.
 150  * Rundown of the variables:
 151  * vp - Pointer to the vnode
 152  * counter - Partial name structure member to update in vopstats for counts
 153  * bytecounter - Partial name structure member to update in vopstats for bytes
 154  * bytesval - Value to update in vopstats for bytes
 155  * fstype - Index into vsanchor_fstype[], same as index into vfssw[]
 156  * vsp - Pointer to vopstats structure (either in vfs or vsanchor_fstype[i])
 157  */

 159 #define VOPSTATS_UPDATE(vp, counter) {                                  \
 160         vfs_t *vfsp = (vp)->v_vfsp;                                    \
 161         if (vfsp && vfsp->vfs_implp &&                                 \
 162             (vfsp->vfs_flag & VFS_STATS) && (vp)->v_type != VBAD) {    \
 163                 vopstats_t *vsp = &vfsp->vfs_vopstats;                 \
 164                 uint64_t *stataddr = &(vsp->n##counter.value.ui64);    \
 165                 extern void __dtrace_probe___fsinfo_##counter(vnode_t *, \
 166                     size_t, uint64_t *);                               \
 167                 __dtrace_probe___fsinfo_##counter(vp, 0, stataddr);    \
 168                 (*stataddr)++;                                         \
 169                 if ((vsp = vfsp->vfs_fstypevsp) != NULL) {             \
 170                         vsp->n##counter.value.ui64++;                  \
 171                 }                                                      \
 172         }                                                              \
 173 }
______unchanged_portion_omitted_

3216 int
3217 fop_read(
3218         vnode_t *vp,
3219         uio_t *uiop,
3220         int ioflag,
3221         cred_t *cr,
3222         caller_context_t *ct)
3223 {
3224         int     err;
3225         ssize_t resid_start = uiop->uio_resid;

3227         VOPXID_MAP_CR(vp, cr);

3229         err = fsh_read(vp, uiop, ioflag, cr, ct);
3228         err = (*(vp)->v_op->vop_read)(vp, uiop, ioflag, cr, ct);
3230         VOPSTATS_UPDATE_IO(vp, read,
3231             read_bytes, (resid_start - uiop->uio_resid));
3232         return (err);
```



```
3233 }

3235 int
3236 fop_write(
3237         vnode_t *vp,
3238         uio_t *uiop,
3239         int ioflag,
3240         cred_t *cr,
3241         caller_context_t *ct)
3242 {
3243         int     err;
3244         ssize_t resid_start = uiop->uio_resid;

3246         VOPXID_MAP_CR(vp, cr);

3248         err = fsh_write(vp, uiop, ioflag, cr, ct);
3247         err = (*(vp)->v_op->vop_write)(vp, uiop, ioflag, cr, ct);
3249         VOPSTATS_UPDATE_IO(vp, write,
3250             write_bytes, (resid_start - uiop->uio_resid));
3251         return (err);
3252 }
______unchanged_portion_omitted_
```

```
**********************************************************
    2082 Mon Aug 19 16:40:54 2013
new/usr/src/uts/common/sys/fsh.h
filesystem hook framework (August 19th)
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2013 Damian Bogel.  All rights reserved.
 */

#ifndef _FSH_H
#define _FSH_H

#include <sys/id_space.h>
#include <sys/types.h>
#include <sys/vfs.h>
#include <sys/vnode.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef id_t fsh_handle_t;
typedef id_t fsh_callback_handle_t;

struct fsh_int;
typedef struct fsh_int fsh_int_t;

typedef struct fsh {
        void *arg;
        void (*remove_callback)(void *, fsh_handle_t);

        /* vnode */
        int (*read)(fsh_int_t *, void *, vnode_t *, uio_t *, int, cred_t *,
                caller_context_t *);
        int (*write)(fsh_int_t *, void *, vnode_t *, uio_t *, int, cred_t *,
                caller_context_t *);

        /* vfs */
        int (*mount)(fsh_int_t *, void *, vfs_t *, vnode_t *, struct mounta *,
                cred_t *);
        int (*unmount)(fsh_int_t *, void *, vfs_t *, int, cred_t *);
} fsh_t;

typedef struct fsh_callback {
        void    *fshc_arg;
        void    (*fshc_free)(vfs_t *, void *);
        void    (*fshc_mount)(vfs_t *, void *);
} fsh_callback_t;

/* API */
extern fsh_handle_t fsh_hook_install(vfs_t *, fsh_t *);
extern int fsh_hook_remove(fsh_handle_t);

extern fsh_callback_handle_t fsh_callback_install(fsh_callback_t *);
extern int fsh_callback_remove(fsh_callback_handle_t);
```



```
extern void fsh_fs_enable(vfs_t *);
extern void fsh_fs_disable(vfs_t *);

/* fsh control passing */
extern int fsh_next_read(fsh_int_t *, vnode_t *, uio_t *, int, cred_t *,
                caller_context_t *);
extern int fsh_next_write(fsh_int_t *, vnode_t *, uio_t *, int, cred_t *,
                caller_context_t *);

extern int fsh_next_mount(fsh_int_t *, vfs_t *, vnode_t *, struct mounta *uap,
                cred_t *);
extern int fsh_next_unmount(fsh_int_t *, vfs_t *, int, cred_t *);

#ifdef __cplusplus
}
#endif

#endif /* _FSH_H */
```

```
**********************************************************
    1413 Mon Aug 19 16:40:55 2013
new/usr/src/uts/common/sys/fsh_impl.h
filesystem hook framework (August 19th)
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2013 Damian Bogel.  All rights reserved.
 */

/*
 * This file includes all the necessary declarations for vfs.c and vnode.c.
 */

#ifndef _FSH_IMPL_H
#define _FSH_IMPL_H

#include <sys/fsh.h>
#include <sys/list.h>
#include <sys/pathname.h>
#include <sys/types.h>
#include <sys/vfs.h>
#include <sys/vnode.h>

#ifdef __cplusplus
extern "C" {
#endif

struct fsh_fshrecord;
typedef struct fsh_fsrecord fsh_fsrecord_t;

/* API for vnode.c and vfs.c only */
/* vnode.c */
extern int fsh_read(vnode_t *, uio_t *, int, cred_t *, caller_context_t *);
extern int fsh_write(vnode_t *, uio_t *, int, cred_t *, caller_context_t *);

/* vfs.c */
extern void fsh_init(void);
extern void fsh_exec_mount_callbacks(vfs_t *);
extern void fsh_exec_free_callbacks(vfs_t *);
extern void fsh_fsrec_destroy(fsh_fsrecord_t *volatile);

extern int fsh_mount(vfs_t *, vnode_t *, struct mounta *, cred_t *);
extern int fsh_unmount(vfs_t *, int, cred_t *);

#ifdef __cplusplus
}
#endif

#endif /* _FSH_IMPL_H */
```

```
**********************************************************
   21197 Mon Aug 19 16:40:55 2013
new/usr/src/uts/common/sys/vfs.h
filesystem hook framework (August 19th)
**********************************************************
______unchanged_portion_omitted_

 173 extern avl_tree_t       vskstat_tree;
 174 extern kmutex_t         vskstat_tree_lock;

 176 /*
 177  * Structure per mounted file system.  Each mounted file system has
 178  * an array of operations and an instance record.
 179  *
 180  * The file systems are kept on a doubly linked circular list headed by
 181  * "rootvfs".
 182  * File system implementations should not access this list;
 183  * it's intended for use only in the kernel's vfs layer.
 184  *
 185  * Each zone also has its own list of mounts, containing filesystems mounted
 186  * somewhere within the filesystem tree rooted at the zone's rootpath.  The
 187  * list is doubly linked to match the global list.
 188  *
 189  * mnttab locking: the in-kernel mnttab uses the vfs_mntpt, vfs_resource and
 190  * vfs_mntopts fields in the vfs_t. mntpt and resource are refstr_ts that
 191  * are set at mount time and can only be modified during a remount.
 192  * It is safe to read these fields if you can prevent a remount on the vfs,
 193  * or through the convenience funcs vfs_getmntpoint() and vfs_getresource().
 194  * The mntopts field may only be accessed through the provided convenience
 195  * functions, as it is protected by the vfs list lock. Modifying a mount
 196  * option requires grabbing the vfs list write lock, which can be a very
 197  * high latency lock.
 198  */
 199 struct zone;            /* from zone.h */
 200 struct fem_head;        /* from fem.h */
 201 struct fsh_fsrecord;    /* from fsh_impl.h */

 203 typedef struct vfs {
 204         struct vfs      *vfs_next;              /* next VFS in VFS list */
 205         struct vfs      *vfs_prev;              /* prev VFS in VFS list */

 207 /* vfs_op should not be used directly.  Accessor functions are provided */
 208         vfsops_t        *vfs_op;                /* operations on VFS */

 210         struct vnode    *vfs_vnodecovered;      /* vnode mounted on */
 211         uint_t          vfs_flag;               /* flags */
 212         uint_t          vfs_bsize;              /* native block size */
 213         int             vfs_fstype;             /* file system type index */
 214         fsid_t          vfs_fsid;               /* file system id */
 215         void            *vfs_data;              /* private data */
 216         dev_t           vfs_dev;                /* device of mounted VFS */
 217         ulong_t         vfs_bcount;             /* I/O count (accounting) */
 218         struct vfs      *vfs_list;              /* sync list pointer */
 219         struct vfs      *vfs_hash;              /* hash list pointer */
 220         ksema_t         vfs_reflock;            /* mount/unmount/sync lock */
 221         uint_t          vfs_count;              /* vfs reference count */
 222         mntopts_t       vfs_mntopts;            /* options mounted with */
 223         refstr_t        *vfs_resource;          /* mounted resource name */
 224         refstr_t        *vfs_mntpt;             /* mount point name */
 225         time_t          vfs_mtime;              /* time we were mounted */
 226         struct vfs_impl *vfs_implp;             /* impl specific data */
 227         /*
 228          * Zones support.  Note that the zone that "owns" the mount isn't
 229          * necessarily the same as the zone in which the zone is visible.
 230          * That is, vfs_zone and (vfs_zone_next|vfs_zone_prev) may refer to
 231          * different zones.
```



```
 232          */
 233         struct zone     *vfs_zone;              /* zone that owns the mount */
 234         struct vfs      *vfs_zone_next;         /* next VFS visible in zone */
 235         struct vfs      *vfs_zone_prev;         /* prev VFS visible in zone */

 237         struct fem_head *vfs_femhead;           /* fs monitoring */
 238         minor_t         vfs_lofi_minor;         /* minor if lofi mount */

 240         struct fsh_fsrecord *volatile
 241                 vfs_fshrecord;                  /* fs hooking */
 242 } vfs_t;
______unchanged_portion_omitted_
```