```
**********************************************************
   34567 Mon Sep  9 17:14:59 2013
new/usr/src/uts/common/fs/fsh.c
Update from fsd_sep3 webrev to fsd_sep9
**********************************************************
```

```
/*
 * 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>

/*
 * Filesystem hook framework (fsh)
 *
 * 1. Abstract.
 * The main goal of the filesystem hook framework is to provide an easy way to
 * inject client-defined behaviour into vfs/vnode calls. fsh works on
 * vfs_t granularity.
 *
 * Note: In this document, both an fsh_t structure and hooking function for a
 * vnodeop/vfsop is referred to as *hook*.
 *
 *
 * 2. Overview.
 * fsh_t is the main object in the fsh. An fsh_t is a structure containing:
 *      - pointers to hooking functions
 *      - an argument to pass (this is shared for all the hooks in a given
 *      fsh_t)
 *      - a pointer to the *hook remove callback*
 *      - 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)
 *      - a pointer to the *hook remove callback* - it's being fired after a
 *      hook is removed and the hook has stopped executing. It's safe to destroy
 *      any data associated with this hook.
 *
 * 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 passed to fsh_foo() functions are held by
```



```
 * the caller when needed. fsh does no vfs_t/vnode_t locking.
 * 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 client. It contains:
 *      - pointers to hooking functions
 *      - the argument passed to the hooks
 *      - the *hook remove callback*
 * fsh_t is a structure filled out by the client. If a client 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
 *      - void (*remove_cb)(void *, fsh_handle_t) - hook remove callback
 *      (mentioned earlier); it's first argument is arg, the second is the
 *      handle
 *
 * If a client does not want to add a hook for function foo(), he should fill
 * corresponding fields with NULLs. For every vfsop/vnodeop there are two
 * fields: pre_foo() and post_foo(). These are the functions called before and
 * after the next hook or underlying vfsop/vnodeop.
 *
 * Pre hooks take:
 *      - arg
 *      - pointer to a field containing void* - it should be filled whenever
 *      the client wants to have some data shared by the pre and post hooks in
 *      the same syscall execution. This is called the *instance data*.
 *      - pointers to the arguments passed to the underlying vfsop/vnodeop
 * Pre hooks return void.
 *
 * Post hooks take:
 *      - value returned by the previous post hook or underlying vfsop/vnodeop
 *      - arg
 *      - pointer to the *instance data*
 *      - arguments passed to the underlying vfsop/vnodeop
 * Post hooks return an int, which should be treated as the vfsop/vnodeop
 * return value.
 * Memory allocated by pre hook must be deallocated by the post hook.
 *
 * Execution path of hooks A, B, C is as follows:
 * foo()
 *      preA(argA, &instancepA, ...);
 *      preB(argB, &instancepB, ...);
 *      preC(argC, &instancepC, ...);
 *      ret = VOP_FOO();
 *      ret = postC(ret, argC, instancepC, ...);
 *      ret = postB(ret, argB, instancepB, ...);
 *      ret = postC(ret, argA, instancepA, ...);
 *      return (ret);
 *
 * After installation, an fsh_handle_t is returned to the caller.
 *
 * Hook remove callback - it's a function being fired after a hook is removed
 * and no thread is going to execute it anymore. It's safe to destroy all the
 * data associated with this hook inside it.
 * 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.
```

```
 79  * Example:
 80  * my_freefs(fsh_int_t *fsh_int, void *arg, vfs_t *vfsp) {
 81  *      cmn_err(CE_NOTE, "freefs called!\n");
 82  *      return (fsh_next_freefs(fsh_int, vfsp));
 83  * }
100  *
101  * It is guaranteed, that whenever a pre_hook() is called, there will be also
102  * post_hook() called within the same syscall.
103  *
104  * If a hook (HNew) is installed/removed on/from a vfs_t within execution of
105  * another hook (HExec) installed on this vfs_t, the syscall that executes
106  * HExec won't fire HNew.
107  *
108  * A client might want to fire callbacks when vfs_ts are being mounted
 86  * A client might want to fire callbacks when vfs_t's are being mounted
109  * or freed. There's an fsh_callback_t structure provided to install such
110  * callbacks along with the API.
111  * It is legal to call fsh_hook_{install,remove}() inside a mount callback
112  * WITHOUT holding the vfs_t.
113  *
114  * After vfs_t's free callback returns, all the handles associated with the
115  * hooks installed on this vfs_t are invalid and must not be used.
116  *
 95  *
117  * 4. API
118  * None of the APIs should be called during interrupt context above lock
119  * level.
 98  * level. The only exceptions are fsh_next_foo() functions, which do not use
 99  * locks.
120  *
121  * a) fsh.h
122  * Any of these functions could be called in a hook or a hook remove callback.
123  * The only functions that must not be called inside a {mount,free} callback are
124  * fsd_callback_{install,remove}. Using them will cause a deadlock.
102  * Any of these functions could be called inside a hook or a hook remove
103  * callback.
104  * fsh_callback_{install,remove}() must not be called inside a {mount,free}
105  * callback. Doing so will cause a deadlock. Other functions can be called
106  * inside {mount,free} callbacks.
125  *
126  *
127  * fsh_fs_enable(vfs_t *vfsp)
128  * fsh_fs_disable(vfs_t *vfsp)
129  *      Enables/disables fsh for a given vfs_t.
130  *
131  * fsh_hook_install(vfs_t *vfsp, fsh_t *hooks)
132  *      Installs hooks on vfsp filesystem.
133  *      It's important that hooks are executed in LIFO installation order,
134  *      which means that if there are hooks A and B installed in this order, B
135  *      is going to be executed before A.
136  *      It returns a correct handle, or (-1) if hook/callback limit exceeded.
137  *      The handle is valid until a free callback returns or an explicit call
138  *      to fsh_hook_remove().
139  *
140  * fsh_hook_remove(fsh_handle_t handle)
141  *      Removes a hook and invalidates the handle.
142  *      It is guaranteed that after this funcion returns, calls to
143  *      vnodeops/vfsops won't go through this hook, although there might be
144  *      some threads still executing this hook. When hook remove callback is
145  *      fired, it is guaranteed that the hook won't be executed anymore. It is
146  *      safe to remove all the internal data associated with this hook inside
147  *      the hook remove callback. The hook remove callback could be called
148  *      inside fsh_hook_remove().
149  *
131  * fsh_next_foo(fsh_int_t *fsh_int, void *arg, ARGUMENTS)
132  *      This is the function which should be called once in every hook. It
```



```
133  *      does the necessary internal operations and passes control to the
134  *      next hook or, if there's no hook left, to the underlying
135  *      vfsop/vnodeop.
150  *
151  * fsh_callback_install(fsh_callback_t *callback)
152  * fsh_callback_remove(fsh_callback_handle_t handle)
153  *      Installs/removes callbacks for vfs_t mount/free. The mount callback
154  *      is executed right before domount() returns. The free callback is
155  *      called right before VFS_FREEVFS() is called.
156  *      The fsh_callback_install() returns a correct handle, or (-1) if
157  *      hook/callback limit exceeded.
158  *
159  *
160  * b) fsh_impl.h (for vfs.c and vnode.c only)
161  * fsh_init()
162  *      This call has to be done in vfsinit(). It initialises the fsh. It
163  *      is absolutely necessary that this call is made before any other fsh
164  *      operation.
165  *
166  * fsh_exec_mount_callbacks(vfs_t *vfsp)
167  * fsh_exec_free_callbacks(vfs_t *vfsp)
168  *      Used to execute all fsh callbacks for {mount,free} of a vfs_t.
169  *
170  * fsh_fsrec_destroy(struct fsh_fsrecord *fsrecp)
171  *      Destroys an fsh_fsrecord structure. All the hooks installed on this
172  *      vfs_t are then destroyed. free callback is called before this function.
173  *
174  * fsh_foo(ARGUMENTS)
175  *      Function used to execute the hook chain for a given syscall.
160  *      Function used to start executing the hook chain for a given call.
176  *
177  *
178  * 5. Internals.
179  * fsh_int_t is an internal hook structure. It is reference counted.
180  * fshi_hold() and fshi_rele() should be used whenever needed.
181  * fsh_int_t entries are elements of both fsh_map (global) and fshfsr_list
182  * (local to vfs_t). All entries are unique and are identified by fshi_handle.
183  *
184  * fsh_int_t properties:
185  *      - fsh_hook_install() sets the ref. counter to 1 and adds it to both
186  *      fsh_map and fshfsr_list
187  *      - fsh_hook_remove() decreases the ref. counter by 1, removes the hook
188  *      from fsh_map and marks the hook as *doomed*
189  *      - if fsh_int_t is on the fshfsr_list, it's alive and there is a thread
190  *      executing it
191  *      - if fsh_int_t is marked as *doomed*, the reference counter is not
192  *      be increased and thus no thread can acquire this fsh_int_t
193  *      - ref. counter can drop to 0 only after an fsh_hook_remove() call; this
194  *      also means that the fsh_int_t is *doomed* and isn't a part of fsh_map
195  *      - fsh_int_t could be also destroyed without fsh_hook_remove() call,
196  *      that happens only inside fsh_fsrec_destroy() where it is guaranteed
197  *      that there is no thread executing the hook
198  *
199  *
200  * fsh_fsrecord_t is a structure which lives inside a vfs_t.
201  * fsh_fsrecord_t contains:
202  *      - an rw-lock that protects the structure
203  *      - a list of hooks installed on this vfs_t
204  *      - a flag which tells whether fsh is enabled on this vfs_t
205  *
206  *
207  * fsh_fsrec_prepare rule:
192  * fsh_prepare_fsrec rule:
208  * Every function that needs vfsp->vfs_fshrecord has to call
209  * fsh_fsrec_prepare() first. If and only if the call is made, it is safe to
194  * fsh_prepare_fsrec() first. If and only if the call is made, it is safe to
```

```
 * use vfsp->vfs_fshrecord.
 *
 * 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_fsrec_prepare(). After
 * explained in more detail in a comment before fsh_prepare_fsrec(). After
 * calling fsh_preapre_fsrec() it's completely safe to keep the vfs_fshrecord
 * pointer locally, because it won't be changed until vfs_free() is called.
 *
 * Exceptions from this rule:
 * - vfs_free() - it is expected that no other fsh calls would be made for the
 * 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. That's why vfs_fshrecord could be only NULL or a
 * valid pointer and could not be concurrently accessed.
 * - fshi_rele() - fsh_hook_install() comes before first fshi_rele() call;
 * the fsh_fsrecord_t has been initialised there
 *
 *
 * 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.
 *
 * It is required and sufficient to check if fsh_fsrecord_t is not NULL before
 * passing it to fsh_fsrec_destroy. We don't have to check if it is not equal
 * to fsh_res_ptr, because all the fsh API calls involving this vfs_t should
 * end before vfs_free() is called (outside the fsh, fsh_fsrecord is never
 * equal to fsh_res_ptr). That is guaranteed by the explicit requirement that
 * the caller of fsh API holds the vfs_t when needed. fsh_hook_remove() must not
 * be called either, because the handles are invalidated after free callback has
 * fired.
 *
 *
 * Callbacks:
 * 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.
 *
 *
 * fsh_next_foo(fsh_int_t *fshi, ARGUMENTS)
 *      This function is quite simple. It takes the fsh_int_t and passes control
 *      to the next hook or to the underlying vnodeop/vfsop.
 *
 *
 * 6. Locking
 * a) public
 * fsh does no vfs_t nor vnode_t locking. It is expected that whenever it is
 * needed, the client does that.
 *
 * No locks are held across hooks or hook remove callbacks execution. It is
 * safe to use fsh API inside hooks and hook remove callbacks.
 * fsh_callback_{install,remove} must not be called inside a callback, because
 * it will cause a deadlock.
 *
 * fsh_cb_lock is held across {mount,free} callbacks. Calling
 * fsh_callback_{install,remove} inside of a callback will cause a deadlock.
 *
 * b) internals
 * b) internal
 * Locking diagram:
 *
```



```
 *     fsh_hook_remove()          fsh_hook_install()   fsh_fsrec_destroy()
 *     fsh_hook_install()    fsh_hook_remove()   fsh_fsrec_destroy()
 *           |                                |                |
 *           |                                |                |
 *           +------------------+             |   +------------+
 *           |                  |             |   |
 *           |                  V             |   |
 *           V                +---------------|---|-+
 *      fshi_rele()           |  fsh_lock     |   | |
 *      (sometimes)           +---------------|---|-+
 *                                  |         |   |
 *                                  |         +---+-- fshfsr_lock, RW_WRITER -+
 *                                V  V   V
 *                                fsh_lock
 *                                  |                                          |
 *                                  |   +----- fshfsr_lock, RW_WRITER ---+     |
 *                                  |                                    |     |
 *                                  V                                    |     |
 *               +--------------------------------------+                |     |
 *               |                fsh_map               |                |     |
 *               |                                      |                |     |
 *          +----|-> vfsp->vfs_fshrecord->fshfsr_list <--|----------------+     |
 *          +----|-> vfsp->vfs_fshrecord->fshfsr_list <--|--------------+       |
 *          |    +--------------------------------^-------+              |       |
 *          |                                     |                      |       |
 *          |                                     |                      |       |
 * fshfsr_lock, RW_READER                  fshfsr_lock, RW_WRITER        |       |
 *          |                                     |                      |       |
 *          |                                     |                      |       |
 *   fsh_read(),                                 fshi_rele()             |       |
 *   fsh_write(),                                                        |       |
 *   ...                                      Might be called from:      |       |
 *                                              fsh_hook_remove()        |       |
 *                                              fsh_read(), fsh_write(), ...     |
 *   ...,                                     Might be called from:      |       |
 *   fsh_next_read(),                          fsh_hook_remove()         |       |
 *   fsh_next_write(),                         fsh_read(), fsh_write(), ...      |
 *   ...                                       fsh_next_read(), fsh_next_write(), ...
 *
 *
 * fsh_lock is a global lock for adminsitrative path (fsh_hook_install,
 * fsh_hook_remove) and fsh_fsrec_destroy() (which is semi-administrative, since
 * it destroys the unremoved hooks). It is used only when fsh_map needs to be
 * locked. The usage of this lock guarantees that the data in fsh_map and
 * fshfsr_lists is consistent.
 *
 * In order to make calling callbacks inside callbacks possible, fsh_cb_owner is
 * set by fsh_exec_{mount,free} callbacks to the thread that owns the
 * fsh_cb_lock.  It's always checked if we are owners of the mutex before
 * entering it.
 *
 */


/* Internals */
typedef struct fsh_int {
struct fsh_int {
        fsh_handle_t    fshi_handle;
        fsh_t           fshi_hooks;
        vfs_t           *fshi_vfsp;

        kmutex_t        fshi_lock;
        uint64_t        fshi_ref;
        uint64_t        fshi_doomed;    /* changed inside fsh_lock */

        /* next node in fshfsr_list */
```

```
320         list_node_t     fshi_node;
287         list_node_t     fshi_next;

322         /* next node in fsh_map */
323         list_node_t     fshi_global;
324 } fsh_int_t;
291 };

326 typedef struct fsh_callback_int {
327         fsh_callback_t  fshci_cb;
328         fsh_callback_handle_t fshci_handle;
329         list_node_t     fshci_node;
296         list_node_t     fshci_next;
330 } fsh_callback_int_t;


333 typedef struct fsh_exec {
334         fsh_int_t       *fshe_fshi;
335         void            *fshe_instance;
336         list_node_t     fshe_node;
337 } fsh_exec_t;



340 static kmutex_t fsh_lock;

342 /*
343  * fsh_fsrecord_t is the main internal structure. It's content is protected
344  * by fshfsr_lock. The fshfsr_list is a list of fsh_int_t hook entries for
345  * the vfs_t that contains the fsh_fsrecord_t.
346  */
347 struct fsh_fsrecord {
348         krwlock_t       fshfsr_lock;
349         int             fshfsr_enabled;
350         list_t          fshfsr_list;
351 };

353 /*
354  * Global list of fsh_int_t. Protected by fsh_lock.
355  */
356 static list_t fsh_map;

358 /*
359  * Global list of fsh_callback_int_t.
360  */
361 static kmutex_t fsh_cb_lock;
362 static kmutex_t fsh_cb_owner_lock;
363 static kthread_t *fsh_cb_owner;
321 static krwlock_t fsh_cblist_lock;
364 static list_t fsh_cblist;

366 /*
367  * A reserved pointer for fsh purposes. It is used because of the method
368  * chosen for solving concurrency issues with vfs_fshrecord. The full
369  * explanation is in the big theory statement at the beginning of this
370  * file and above fsh_fsrec_prepare(). It is initialised in fsh_init().
371  */
372 static void *fsh_res_ptr;

374 static fsh_fsrecord_t *fsh_fsrec_create();

376 int fsh_limit = INT_MAX;
377 static id_space_t *fsh_idspace;

379 /*
380  * fsh_fsrec_prepare()
338  * fsh_prepare_fsrec()
```



```
381  *
382  * Important note:
383  * Before using this function, fsh_init() MUST be called. We do that in
384  * vfsinit()@vfs.c.
385  *
386  * One would ask, why isn't the vfsp->vfs_fshrecord initialised when the
387  * vfs_t is created. Unfortunately, some filesystems (e.g. fifofs) do not
388  * call vfs_init() or even vfs_alloc(), It's possible that some unbundled
389  * filesystems could do the same thing. That's why this solution is
390  * introduced. It should be called before any code that needs access to
391  * vfs_fshrecord.
392  *
393  * Locking:
394  * There are no locks here, because there's no good place to initialise
395  * the lock. Concurrency issues are solved by using atomic instructions
396  * and a spinlock, which is spinning only once for a given vfs_t. Because
397  * of that, the usage of the spinlock isn't bad at all.
398  *
399  * How it works:
400  * a) if vfsp->vfs_fshrecord equals NULL, atomic_cas_ptr() changes it to
401  *      fsh_res_ptr. That's a signal for other threads, that the structure
402  *      is being initialised.
403  * b) if vfsp->vfs_fshrecord equals fsh_res_ptr, that means we have to wait,
404  *      because vfs_fshrecord is being initialised by another call.
405  * c) other cases:
406  *      vfs_fshrecord is already initialised, so we can use it. It won't change
407  *      until vfs_free() is called. It can't happen when someone is holding
408  *      the vfs_t, which is expected from the caller of fsh API.
409  */
410 static void
411 fsh_fsrec_prepare(vfs_t *vfsp)
369 fsh_prepare_fsrec(vfs_t *vfsp)
412 {
413         fsh_fsrecord_t *fsrec;

415         while ((fsrec = atomic_cas_ptr(&vfsp->vfs_fshrecord, NULL,
416             fsh_res_ptr)) == fsh_res_ptr)
417                 ;

419         if (fsrec == NULL)
420                 atomic_swap_ptr(&vfsp->vfs_fshrecord, fsh_fsrec_create());
421 }

423 /*
424  * API for enabling/disabling fsh per vfs_t.
425  *
426  * A newly created vfs_t has fsh enabled by default. If one would want to change
427  * this behaviour, mount callbacks could be used.
428  *
429  * The caller is expected to hold the vfs_t.
430  *
431  * These functions must NOT be called in a hook.
432  */
433 void
434 fsh_fs_enable(vfs_t *vfsp)
435 {
436         fsh_fsrec_prepare(vfsp);
394         fsh_prepare_fsrec(vfsp);

438         rw_enter(&vfsp->vfs_fshrecord->fshfsr_lock, RW_WRITER);
439         vfsp->vfs_fshrecord->fshfsr_enabled = 1;
440         rw_exit(&vfsp->vfs_fshrecord->fshfsr_lock);
441 }

443 void
444 fsh_fs_disable(vfs_t *vfsp)
```

```
 445 {
 446         fsh_fsrec_prepare(vfsp);
 404         fsh_prepare_fsrec(vfsp);

 448         rw_enter(&vfsp->vfs_fshrecord->fshfsr_lock, RW_WRITER);
 449         vfsp->vfs_fshrecord->fshfsr_enabled = 0;
 450         rw_exit(&vfsp->vfs_fshrecord->fshfsr_lock);
 451 }

 453 /*
 454  * API used for installing hooks. fsh_handle_t is returned for further
 455  * actions (currently just removing) on this set of hooks.
 456  *
 415  * fsh_t fields:
 416  * - arg - argument passed to every hook
 417  * - remove_cb - remove callback, called after a hook is removed and all the
 418  *      threads stops executing it
 419  * - read, write, ... - pointers to hooks for corresponding vnodeops/vfsops;
 420  *      if there is no hook desired for an operation, it should be set to
 421  *      NULL
 422  *
 457  * It's important that the hooks are executed in LIFO installation order (they
 458  * are added to the head of the hook list).
 459  *
 460  * The caller is expected to hold the vfs_t.
 461  *
 462  * Returns (-1) if hook/callback limit exceeded, handle otherwise.
 463  */
 464 fsh_handle_t
 465 fsh_hook_install(vfs_t *vfsp, fsh_t *hooks)
 466 {
 467         fsh_handle_t    handle;
 468         fsh_int_t       *fshi;

 470         fsh_fsrec_prepare(vfsp);
 436         fsh_prepare_fsrec(vfsp);

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

 475         fshi = kmem_alloc(sizeof (*fshi), KM_SLEEP);
 476         mutex_init(&fshi->fshi_lock, NULL, MUTEX_DRIVER, NULL);
 477         (void) memcpy(&fshi->fshi_hooks, hooks, sizeof (fshi->fshi_hooks));
 478         fshi->fshi_handle = handle;
 479         fshi->fshi_doomed = 0;
 480         fshi->fshi_ref = 1;
 481         fshi->fshi_vfsp = vfsp;

 483         mutex_enter(&fsh_lock);
 484         rw_enter(&vfsp->vfs_fshrecord->fshfsr_lock, RW_WRITER);
 485         list_insert_head(&vfsp->vfs_fshrecord->fshfsr_list, fshi);
 486         rw_exit(&vfsp->vfs_fshrecord->fshfsr_lock);

 488         list_insert_head(&fsh_map, fshi);
 489         mutex_exit(&fsh_lock);

 491         return (handle);
 492 }
_____unchanged_portion_omitted_

 511 /*
 512  * This function must not be called while fshfsr_lock is held. Doing so could
 513  * cause a deadlock.
 514  */
 515 static void
 516 fshi_rele(fsh_int_t *fshi)
```



```
 517 {
 518         int destroy;

 520         mutex_enter(&fshi->fshi_lock);
 521         ASSERT(fshi->fshi_ref > 0);
 522         fshi->fshi_ref--;
 523         if (fshi->fshi_ref == 0) {
 524                 ASSERT(fshi->fshi_doomed == 1);
 525                 destroy = 1;
 526         } else {
 527                 destroy = 0;
 528         }
 529         mutex_exit(&fshi->fshi_lock);

 531         if (destroy) {
 532                 /*
 533                  * At this point, we are sure that fsh_hook_remove() has been
 534                  * called, that's why we don't remove the fshi from fsh_map.
 535                  * fsh_hook_remove() did that already.
 536                  * There is also no need to call fsh_fsrec_prepare() here.
 537                  */
 538                 fsh_fsrecord_t *fsrecp;

 505                 if (fshi->fshi_hooks.remove_cb != NULL)
 506                         (*fshi->fshi_hooks.remove_cb)(
 507                             fshi->fshi_hooks.arg, fshi->fshi_handle);
 540                 /*
 541                  * We don't have to call fsh_fsrec_prepare() here.
 509                  * We don't have to call fsh_prepare_fsrec() here.
 542                  * fsh_fsrecord_t is already initialised, because we've found a
 543                  * mapping for the given handle.
 544                  */
 545                 fsrecp = fshi->fshi_vfsp->vfs_fshrecord;
 546                 ASSERT(fsrecp != NULL);
 547                 ASSERT(fsrecp != fsh_res_ptr);

 549                 rw_enter(&fsrecp->fshfsr_lock, RW_WRITER);
 550                 list_remove(&fsrecp->fshfsr_list, fshi);
 551                 rw_exit(&fsrecp->fshfsr_lock);

 553                 if (fshi->fshi_hooks.remove_cb != NULL)
 554                         (*fshi->fshi_hooks.remove_cb)(
 555                             fshi->fshi_hooks.arg, fshi->fshi_handle);

 557                 id_free(fsh_idspace, fshi->fshi_handle);
 558                 mutex_destroy(&fshi->fshi_lock);
 559                 kmem_free(fshi, sizeof (*fshi));
 560         }
 561 }
_____unchanged_portion_omitted_

 611 /*
 612  * API for installing global mount/free callbacks.
 613  *
 614  * fsh_callback_t fields:
 615  * fshc_arg - argument passed to the callbacks
 616  * fshc_free - callback fired before VFS_FREEVFS() is called, after vfs_count
 617  *      drops to 0
 618  * fshc_mount - callback fired right before returning from domount()
 619  * The first argument of these callbacks is the vfs_t that is mounted/freed.
 620  * The second one is the fshc_arg.
 621  *
 622  * fsh_callback_handle_t is filled out by this function.
 623  *
 624  * Returns (-1) if hook/callback limit exceeded.
 588  * This function must NOT be called in a callback, because it will cause
```

```
 * a deadlock.
 *
 * Calling this function in a {mount,free} callback will cause a deadlock.
 * Returns (-1) if hook/callback limit exceeded.
 */
fsh_callback_handle_t
fsh_callback_install(fsh_callback_t *callback)
{
        fsh_callback_int_t *fshci;
        fsh_callback_handle_t handle;

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

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

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

        return (handle);
}

/*
 * API for removing global mount/free callbacks.
 *
 * Returns (-1) if callback wasn't found, 0 otherwise.
 * This function must NOT be called in a callback, because it will cause
 * a deadlock.
 *
 * Calling this function in a {mount,free} callback will cause a deadlock.
 * Returns (-1) if callback wasn't found, 0 otherwise.
 */
int
fsh_callback_remove(fsh_callback_handle_t handle)
{
        fsh_callback_int_t *fshci;

        mutex_enter(&fsh_cb_lock);

        /* If it is called in a {mount,free} callback, causes deadlock. */
        rw_enter(&fsh_cblist_lock, RW_WRITER);
        for (fshci = list_head(&fsh_cblist); fshci != NULL;
            fshci = list_next(&fsh_cblist, fshci)) {
                if (fshci->fshci_handle == handle) {
                        list_remove(&fsh_cblist, fshci);
                        break;
                }
        }
        rw_exit(&fsh_cblist_lock);

        mutex_exit(&fsh_cb_lock);

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

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

        return (0);
}
```



```
/*
 * This function is executed right before returning from domount()@vfs.c.
 * We are sure that it's called only after fsh_init().
 * It executes all the mount callbacks installed in the fsh.
 *
 * Since fsh_exec_mount_callbacks() is called only inside domount(), it is legal
 * to call fsh_hook_{install,remove}() inside a mount callback WITHOUT holding
 * this vfs_t. This guarantee should be preserved, because it's in the "Usage"
 * section in the big theory statement at the top of this file.
 */
void
fsh_exec_mount_callbacks(vfs_t *vfsp)
{
        fsh_callback_int_t *fshci;
        fsh_callback_t *cb;
        int fsh_context;

        mutex_enter(&fsh_cb_owner_lock);
        fsh_context = fsh_cb_owner == curthread;
        mutex_exit(&fsh_cb_owner_lock);

        if (!fsh_context) {
                mutex_enter(&fsh_cb_lock);
                mutex_enter(&fsh_cb_owner_lock);
                fsh_cb_owner = curthread;
                mutex_exit(&fsh_cb_owner_lock);
        }

        ASSERT(MUTEX_HELD(&fsh_cb_lock));

        rw_enter(&fsh_cblist_lock, RW_READER);
        for (fshci = list_head(&fsh_cblist); fshci != NULL;
            fshci = list_next(&fsh_cblist, fshci)) {
                cb = &fshci->fshci_cb;
                if (cb->fshc_mount != NULL)
                        (*(cb->fshc_mount))(vfsp, cb->fshc_arg);
        }

        if (!fsh_context) {
                mutex_enter(&fsh_cb_owner_lock);
                fsh_cb_owner = NULL;
                mutex_exit(&fsh_cb_owner_lock);
                mutex_exit(&fsh_cb_lock);
        }
        rw_exit(&fsh_cblist_lock);
}

/*
 * This function is executed right before VFS_FREEVFS() is called in
 * vfs_rele()@vfs.c. We are sure that it's called only after fsh_init().
 * It executes all the free callbacks installed in the fsh.
 *
 * free() callback is the point after the handles associated with the hooks
 * installed on this vfs_t become invalid
 */
void
fsh_exec_free_callbacks(vfs_t *vfsp)
{
        fsh_callback_int_t *fshci;
        fsh_callback_t *cb;
        int fsh_context;

        mutex_enter(&fsh_cb_owner_lock);
        fsh_context = fsh_cb_owner == curthread;
        mutex_exit(&fsh_cb_owner_lock);
```

```
745         if (!fsh_context) {
746                 mutex_enter(&fsh_cb_lock);
747                 mutex_enter(&fsh_cb_owner_lock);
748                 fsh_cb_owner = curthread;
749                 mutex_exit(&fsh_cb_owner_lock);
750         }

752         ASSERT(MUTEX_HELD(&fsh_cb_lock));

687         rw_enter(&fsh_cblist_lock, RW_READER);
754         for (fshci = list_head(&fsh_cblist); fshci != NULL;
755             fshci = list_next(&fsh_cblist, fshci)) {
756                 cb = &fshci->fshci_cb;
757                 if (cb->fshc_free != NULL)
758                         (*(cb->fshc_free))(vfsp, cb->fshc_arg);
759         }

761         if (!fsh_context) {
762                 mutex_enter(&fsh_cb_owner_lock);
763                 fsh_cb_owner = NULL;
764                 mutex_exit(&fsh_cb_owner_lock);
765                 mutex_exit(&fsh_cb_lock);
766         }
694         rw_exit(&fsh_cblist_lock);
767 }

769 /*
770  * API for vnode.c/vfs.c to start executing the fsh for a given operation.
771  *
772  * fsh_xxx() tries to find the first non-NULL xxx hook on the fshfsr_list. If it
773  * does, it executes it. If not, underlying vnodeop/vfsop is called.
774  *
775  * These interfaces are using fsh_res_ptr (in fsh_fsrec_prepare()), so it's
703  * These interfaces are using fsh_res_ptr (in fsh_prepare_fsrec()), so it's
776  * absolutely necessary to call fsh_init() before using them. That's done in
777  * vfsinit().
778  *
779  * While these functions are executing, it's expected that necessary vfs_t's
780  * are held so that vfs_free() isn't called. vfs_free() expects that noone
781  * accesses vfs_fshrecord of a given vfs_t.
782  * It's also the caller's responsibility to keep vnode_t passed to fsh_foo()
783  * alive and valid.
784  * All these expectations are met because these functions are used only in
785  * correspondng {fop,fsop}_foo() functions.
786  */
787 int
788 fsh_read(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
789         caller_context_t *ct)
790 {
791         int ret;
792         fsh_fsrecord_t *fsrecp;
793         fsh_int_t *fshi;
794         fsh_exec_t *fshe;
795         list_t exec_list;

797         fsh_fsrec_prepare(vp->v_vfsp);
723         fsh_prepare_fsrec(vp->v_vfsp);
798         fsrecp = vp->v_vfsp->vfs_fshrecord;

800         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
801         if (!(fsrecp->fshfsr_enabled)) {
802                 rw_exit(&fsrecp->fshfsr_lock);
803                 return ((*vp->v_op->vop_read)(vp, uiop, ioflag, cr, ct));
729                 return ((*(vp->v_op->vop_read))(vp, uiop, ioflag, cr, ct));
804         }
```



```
806         list_create(&exec_list, sizeof (fsh_exec_t),
807             offsetof(fsh_exec_t, fshe_node));

809         for (fshi = list_head(&fsrecp->fshfsr_list); fshi != NULL;
810             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
811                 if (fshi->fshi_hooks.pre_read != NULL ||
812                     fshi->fshi_hooks.post_read != NULL) {
813                         if (fshi_hold(fshi)) {
814                                 fshe = kmem_alloc(sizeof (*fshe), KM_SLEEP);
815                                 fshe->fshe_fshi = fshi;
816                                 list_insert_tail(&exec_list, fshe);
734                 if (fshi->fshi_hooks.read != NULL)
735                         if (fshi_hold(fshi))
736                                 break;
817                         }
818                 }
819         }
820         rw_exit(&fsrecp->fshfsr_lock);

822         /* Execute pre hooks */
823         for (fshe = list_head(&exec_list); fshe != NULL;
824             fshe = list_next(&exec_list, fshe)) {
825                 if (fshe->fshe_fshi->fshi_hooks.pre_read != NULL)
826                         (*fshe->fshe_fshi->fshi_hooks.pre_read)(
827                             fshe->fshe_fshi->fshi_hooks.arg,
828                             &fshe->fshe_instance,
829                             &vp, &uiop, &ioflag, &cr, &ct);
830         }
740         if (fshi == NULL)
741                 return ((*(vp->v_op->vop_read))(vp, uiop, ioflag, cr, ct));

832         ret = (*vp->v_op->vop_read)(vp, uiop, ioflag, cr, ct);

834         /* Execute post hooks */
835         while ((fshe = list_remove_tail(&exec_list)) != NULL) {
836                 if (fshe->fshe_fshi->fshi_hooks.post_read != NULL)
837                         ret = (*fshe->fshe_fshi->fshi_hooks.post_read)(
838                             ret, fshe->fshe_fshi->fshi_hooks.arg,
839                             fshe->fshe_instance,
743         ret = (*fshi->fshi_hooks.read)(fshi, fshi->fshi_hooks.arg,
840                             vp, uiop, ioflag, cr, ct);
841                 fshi_rele(fshe->fshe_fshi);
842                 kmem_free(fshe, sizeof (*fshe));
843         }
844         list_destroy(&exec_list);

745         fshi_rele(fshi);
846         return (ret);
847 }

849 int
850 fsh_write(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
851         caller_context_t *ct)
852 {
753         fsh_int_t *fshi;
853         int ret;
854         fsh_fsrecord_t *fsrecp;
855         fsh_int_t *fshi;
856         fsh_exec_t *fshe;
857         list_t exec_list;

859         fsh_fsrec_prepare(vp->v_vfsp);
757         fsh_prepare_fsrec(vp->v_vfsp);
860         fsrecp = vp->v_vfsp->vfs_fshrecord;
```

```
862         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
863         if (!(fsrecp->fshfsr_enabled)) {
761         if (!(vp->v_vfsp->vfs_fshrecord->fshfsr_enabled)) {
864                 rw_exit(&fsrecp->fshfsr_lock);
865                 return ((*vp->v_op->vop_write)(vp, uiop, ioflag, cr, ct));
763                 return ((*(vp->v_op->vop_write))(vp, uiop, ioflag, cr, ct));
866         }

868         list_create(&exec_list, sizeof (fsh_exec_t),
869             offsetof(fsh_exec_t, fshe_node));

871         for (fshi = list_head(&fsrecp->fshfsr_list); fshi != NULL;
872             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
873                 if (fshi->fshi_hooks.pre_write != NULL ||
874                     fshi->fshi_hooks.post_write != NULL) {
875                         if (fshi_hold(fshi)) {
876                                 fshe = kmem_alloc(sizeof (*fshe), KM_SLEEP);
877                                 fshe->fshe_fshi = fshi;
878                                 list_insert_tail(&exec_list, fshe);
768                 if (fshi->fshi_hooks.write != NULL)
769                         if (fshi_hold(fshi))
770                                 break;
879                         }
880                 }
881         }
882         rw_exit(&fsrecp->fshfsr_lock);

884         /* Execute pre hooks */
885         for (fshe = list_head(&exec_list); fshe != NULL;
886             fshe = list_next(&exec_list, fshe)) {
887                 if (fshe->fshe_fshi->fshi_hooks.pre_write != NULL)
888                         (*fshe->fshe_fshi->fshi_hooks.pre_write)(
889                             fshe->fshe_fshi->fshi_hooks.arg,
890                             &fshe->fshe_instance,
891                             &vp, &uiop, &ioflag, &cr, &ct);
892         }
774         if (fshi == NULL)
775                 return ((*(vp->v_op->vop_write))(vp, uiop, ioflag, cr, ct));

894         ret = (*vp->v_op->vop_write)(vp, uiop, ioflag, cr, ct);

896         /* Execute post hooks */
897         while ((fshe = list_remove_tail(&exec_list)) != NULL) {
898                 if (fshe->fshe_fshi->fshi_hooks.post_write != NULL)
899                         ret = (*fshe->fshe_fshi->fshi_hooks.post_write)(
900                             ret, fshe->fshe_fshi->fshi_hooks.arg,
901                             fshe->fshe_instance,
777         ret = (*fshi->fshi_hooks.write)(fshi, fshi->fshi_hooks.arg,
902                             vp, uiop, ioflag, cr, ct);
903                 fshi_rele(fshe->fshe_fshi);
904                 kmem_free(fshe, sizeof (*fshe));
905         }
906         list_destroy(&exec_list);

779         fshi_rele(fshi);
908         return (ret);
909 }

911 int
912 fsh_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
913 {
914         int ret;
915         fsh_fsrecord_t *fsrecp;
916         fsh_int_t *fshi;
917         fsh_exec_t *fshe;
918         list_t exec_list;
```



```
788         int ret;

920         fsh_fsrec_prepare(vfsp);
790         fsh_prepare_fsrec(vfsp);
921         fsrecp = vfsp->vfs_fshrecord;

923         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
924         if (!(fsrecp->fshfsr_enabled)) {
925                 rw_exit(&fsrecp->fshfsr_lock);
926                 return ((*vfsp->vfs_op->vfs_mount)(vfsp, mvp, uap, cr));
796                 return ((*(vfsp->vfs_op->vfs_mount))(vfsp, mvp, uap, cr));
927         }

929         list_create(&exec_list, sizeof (fsh_exec_t),
930             offsetof(fsh_exec_t, fshe_node));

932         for (fshi = list_head(&fsrecp->fshfsr_list); fshi != NULL;
933             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
934                 if (fshi->fshi_hooks.pre_mount != NULL ||
935                     fshi->fshi_hooks.post_mount != NULL) {
936                         if (fshi_hold(fshi)) {
937                                 fshe = kmem_alloc(sizeof (*fshe), KM_SLEEP);
938                                 fshe->fshe_fshi = fshi;
939                                 list_insert_tail(&exec_list, fshe);
801                 if (fshi->fshi_hooks.mount != NULL)
802                         if (fshi_hold(fshi))
803                                 break;
940                         }
941                 }
942         }
943         rw_exit(&fsrecp->fshfsr_lock);

945         /* Execute pre hooks */
946         for (fshe = list_head(&exec_list); fshe != NULL;
947             fshe = list_next(&exec_list, fshe)) {
948                 if (fshe->fshe_fshi->fshi_hooks.pre_mount != NULL)
949                         (*fshe->fshe_fshi->fshi_hooks.pre_mount)(
950                             &fshe->fshe_fshi->fshi_hooks.arg,
951                             &fshe->fshe_instance,
952                             &vfsp, &mvp, &uap, &cr);
953         }
807         if (fshi == NULL)
808                 return ((*(vfsp->vfs_op->vfs_mount))(vfsp, mvp, uap, cr));

955         ret = (*vfsp->vfs_op->vfs_mount)(vfsp, mvp, uap, cr);

957         /* Execute post hooks */
958         while ((fshe = list_remove_tail(&exec_list)) != NULL) {
959                 if (fshe->fshe_fshi->fshi_hooks.post_mount != NULL)
960                         ret = (*fshe->fshe_fshi->fshi_hooks.post_mount)(
961                             ret, fshe->fshe_fshi->fshi_hooks.arg,
962                             fshe->fshe_instance,
810         ret = (*fshi->fshi_hooks.mount)(fshi, fshi->fshi_hooks.arg,
963                             vfsp, mvp, uap, cr);
964                 fshi_rele(fshe->fshe_fshi);
965                 kmem_free(fshe, sizeof (*fshe));
966         }
967         list_destroy(&exec_list);

812         fshi_rele(fshi);
969         return (ret);
970 }

972 int
973 fsh_unmount(vfs_t *vfsp, int flag, cred_t *cr)
974 {
```

```
 975         int ret;
 976         fsh_fsrecord_t *fsrecp;
 977         fsh_int_t *fshi;
 978         fsh_exec_t *fshe;
 979         list_t exec_list;
 821         int ret;

 981         fsh_fsrec_prepare(vfsp);
 823         fsh_prepare_fsrec(vfsp);
 982         fsrecp = vfsp->vfs_fshrecord;

 984         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
 985         if (!(fsrecp->fshfsr_enabled)) {
 986                 rw_exit(&fsrecp->fshfsr_lock);
 987                 return ((*vfsp->vfs_op->vfs_unmount)(vfsp, flag, cr));
 829                 return ((*(vfsp->vfs_op->vfs_unmount))(vfsp, flag, cr));
 988         }

 990         list_create(&exec_list, sizeof (fsh_exec_t),
 991             offsetof(fsh_exec_t, fshe_node));

 993         for (fshi = list_head(&fsrecp->fshfsr_list); fshi != NULL;
 994             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
 995                 if (fshi->fshi_hooks.pre_unmount != NULL ||
 996                     fshi->fshi_hooks.post_unmount != NULL) {
 997                         if (fshi_hold(fshi)) {
 998                                 fshe = kmem_alloc(sizeof (*fshe), KM_SLEEP);
 999                                 fshe->fshe_fshi = fshi;
1000                                 list_insert_tail(&exec_list, fshe);
 834                 if (fshi->fshi_hooks.unmount != NULL)
 835                         if (fshi_hold(fshi))
 836                                 break;
1001                         }
1002                 }
1003         }
1004         rw_exit(&fsrecp->fshfsr_lock);

1006         /* Execute pre hooks */
1007         for (fshe = list_head(&exec_list); fshe != NULL;
1008             fshe = list_next(&exec_list, fshe)) {
1009                 if (fshe->fshe_fshi->fshi_hooks.pre_unmount != NULL)
1010                         (*fshe->fshe_fshi->fshi_hooks.pre_unmount)(
1011                             fshe->fshe_fshi->fshi_hooks.arg,
1012                             &fshe->fshe_instance,
1013                             &vfsp, &flag, &cr);
1014         }
 840         if (fshi == NULL)
 841                 return ((*(vfsp->vfs_op->vfs_unmount))(vfsp, flag, cr));

1016         ret = (*vfsp->vfs_op->vfs_unmount)(vfsp, flag, cr);

1018         /* Execute post hooks */
1019         while ((fshe = list_remove_tail(&exec_list)) != NULL) {
1020                 if (fshe->fshe_fshi->fshi_hooks.post_unmount != NULL)
1021                         ret = (*fshe->fshe_fshi->fshi_hooks.post_unmount)(
1022                             ret, fshe->fshe_fshi->fshi_hooks.arg,
1023                             fshe->fshe_instance,
 843         ret = (*fshi->fshi_hooks.unmount)(fshi, fshi->fshi_hooks.arg,
1024                             vfsp, flag, cr);
1025                 fshi_rele(fshe->fshe_fshi);
1026                 kmem_free(fshe, sizeof (*fshe));
1027         }
1028         list_destroy(&exec_list);

 845         fshi_rele(fshi);
1030         return (ret);
```



```
1031 }

1033 /*
1034  * This is the funtion used by fsh_fsrec_prepare() to allocate a new
 850  * This is the funtion used by fsh_prepare_fsrec() to allocate a new
1035  * fsh_fsrecord. This function is called by the first function which
1036  * access the vfs_fshrecord and finds out it's NULL.
1037  */
1038 static fsh_fsrecord_t *
1039 fsh_fsrec_create()
1040 {
1041         fsh_fsrecord_t *fsrecp;

1043         fsrecp = (fsh_fsrecord_t *)kmem_zalloc(sizeof (*fsrecp), KM_SLEEP);
1044         list_create(&fsrecp->fshfsr_list, sizeof (fsh_int_t),
1045             offsetof(fsh_int_t, fshi_node));
 861             offsetof(fsh_int_t, fshi_next));
1046         rw_init(&fsrecp->fshfsr_lock, NULL, RW_DRIVER, NULL);
1047         fsrecp->fshfsr_enabled = 1;
1048         return (fsrecp);
1049 }


1052 /*
1053  * This call must be used ONLY in vfs_free().
 869  * This call can be used ONLY in vfs_free(). It's assumed that no other
 870  * fsh calls using the vfs_t that owns the fsh_fsrecord to be destroyed
 871  * are executing while a call to fsh_fsrec_destroy() is made. With this
 872  * assumptions, no concurrency issues occur.
1054  *
1055  * It is required and sufficient to check if fsh_fsrecord_t is not NULL before
1056  * passing it to fsh_fsrec_destroy.
 874  * Before calling this function outside the fsh, it's sufficient and
 875  * required to check if the passed fsh_fsrecord * is not NULL. We don't
 876  * have to check if it is not equal to fsh_res_ptr, because all the fsh API
 877  * calls involving this vfs_t should end before vfs_free() is called
 878  * (outside the fsh, fsh_fsrecord is never equal to fsh_res_ptr). That is
 879  * guaranteed by the explicit requirement that the caller of fsh API holds
 880  * the vfs_t when needed.
1057  *
1058  * All the remaining hooks are being removed here.
 882  * All the remaining hooks are being removed.
1059  */
1060 void
1061 fsh_fsrec_destroy(struct fsh_fsrecord *volatile fsrecp)
1062 {
1063         fsh_int_t *fshi;

1065         VERIFY(fsrecp != NULL);

1067         _NOTE(CONSTCOND)
1068         while (1) {
1069                 mutex_enter(&fsh_lock);
1070                 rw_enter(&fsrecp->fshfsr_lock, RW_WRITER);
 894                 /* No need here to hold fshfsr_lock */
1071                 fshi = list_remove_head(&fsrecp->fshfsr_list);
1072                 rw_exit(&fsrecp->fshfsr_lock);
1073                 if (fshi == NULL) {
1074                         mutex_exit(&fsh_lock);
1075                         break;
1076                 }
1077                 ASSERT(fshi->fshi_doomed == 0);
1078                 list_remove(&fsh_map, fshi);
1079                 mutex_exit(&fsh_lock);

1081                 if (fshi->fshi_hooks.remove_cb != NULL)
```

```
1082                        (*fshi->fshi_hooks.remove_cb)(fshi->fshi_hooks.arg,
1083                            fshi->fshi_handle);

1085                id_free(fsh_idspace, fshi->fshi_handle);
1086                mutex_destroy(&fshi->fshi_lock);
1087                kmem_free(fshi, sizeof (*fshi));

1089        }

1091        list_destroy(&fsrecp->fshfsr_list);
1092        rw_destroy(&fsrecp->fshfsr_lock);
1093        kmem_free(fsrecp, sizeof (*fsrecp));
1094 }

1096 /*
1097  * fsh_init() is called in vfsinit()@vfs.c. This function MUST be called
1098  * before every other fsh call.
1099  */
1100 void
1101 fsh_init(void)
1102 {
1103         mutex_init(&fsh_cb_lock, NULL, MUTEX_DRIVER, NULL);
1104         mutex_init(&fsh_cb_owner_lock, NULL, MUTEX_DRIVER, NULL);
 925         rw_init(&fsh_cblist_lock, NULL, RW_DRIVER, NULL);
1105         list_create(&fsh_cblist, sizeof (fsh_callback_int_t),
1106             offsetof(fsh_callback_int_t, fshci_node));
 927             offsetof(fsh_callback_int_t, fshci_next));

1108         mutex_init(&fsh_lock, NULL, MUTEX_DRIVER, NULL);

1110         list_create(&fsh_map, sizeof (fsh_int_t), offsetof(fsh_int_t,
1111             fshi_global));

1113         /* See comment above fsh_fsrec_prepare() */
 934         /* See comment above fsh_prepare_fsrec() */
1114         fsh_res_ptr = (void *)-1;

1116         fsh_idspace = id_space_create("fsh", 0, fsh_limit);
 938 }

 940 /*
 941  * These functions are used to pass control to the next hook or underlying
 942  * vop or vfsop. It's client doesn't have to worry about any locking.
 943  */
 944 int
 945 fsh_next_read(fsh_int_t *fshi, vnode_t *vp, uio_t *uiop, int ioflag,
 946         cred_t *cr, caller_context_t *ct)
 947 {
 948         int ret;
 949         fsh_fsrecord_t *fsrecp = vp->v_vfsp->vfs_fshrecord;

 951         /*
 952          * The passed fshi is the previous hook (the one from which we've been
 953          * called). We need to find the next one.
 954          */
 955         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
 956         for (fshi = list_next(&fsrecp->fshfsr_list, fshi); fshi != NULL;
 957             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
 958                 if (fshi->fshi_hooks.read != NULL)
 959                         if (fshi_hold(fshi))
 960                                 break;
 961         }
 962         rw_exit(&fsrecp->fshfsr_lock);

 964         if (fshi == NULL)
 965                 return ((*vp->v_op->vop_read)(vp, uiop, ioflag, cr, ct));
```



```
 967         ret = (*fshi->fshi_hooks.read)(fshi, fshi->fshi_hooks.arg,
 968             vp, uiop, ioflag, cr, ct);
 969         fshi_rele(fshi);
 970         return (ret);
 971 }

 973 int
 974 fsh_next_write(fsh_int_t *fshi, vnode_t *vp, uio_t *uiop, int ioflag,
 975         cred_t *cr, caller_context_t *ct)
 976 {
 977         fsh_fsrecord_t *fsrecp = vp->v_vfsp->vfs_fshrecord;
 978         int ret;

 980         /*
 981          * The passed fshi is the previous hook (the one from which we've been
 982          * called). We need to find the next one.
 983          */
 984         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
 985         for (fshi = list_next(&fsrecp->fshfsr_list, fshi); fshi != NULL;
 986             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
 987                 if (fshi->fshi_hooks.write != NULL)
 988                         if (fshi_hold(fshi))
 989                                 break;
 990         }
 991         rw_exit(&fsrecp->fshfsr_lock);

 993         if (fshi == NULL)
 994                 return ((*vp->v_op->vop_write)(vp, uiop, ioflag, cr, ct));

 996         ret = (*fshi->fshi_hooks.write)(fshi, fshi->fshi_hooks.arg,
 997             vp, uiop, ioflag, cr, ct);
 998         fshi_rele(fshi);
 999         return (ret);
1000 }

1002 int
1003 fsh_next_mount(fsh_int_t *fshi, vfs_t *vfsp, vnode_t *mvp, struct mounta *uap,
1004         cred_t *cr)
1005 {
1006         fsh_fsrecord_t *fsrecp = vfsp->vfs_fshrecord;
1007         int ret;

1009         /*
1010          * The passed fshi is the previous hook (the one from which we've been
1011          * called). We need to find the next one.
1012          */
1013         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
1014         for (fshi = list_next(&fsrecp->fshfsr_list, fshi); fshi != NULL;
1015             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
1016                 if (fshi->fshi_hooks.mount != NULL)
1017                         if (fshi_hold(fshi))
1018                                 break;
1019         }
1020         rw_exit(&fsrecp->fshfsr_lock);

1022         if (fshi == NULL)
1023                 return ((*(vfsp->vfs_op->vfs_mount))(vfsp, mvp, uap, cr));

1025         ret = (*fshi->fshi_hooks.mount)(fshi, fshi->fshi_hooks.arg,
1026             vfsp, mvp, uap, cr);
1027         fshi_rele(fshi);
1028         return (ret);
1029 }

1031 int
```

```
1032 fsh_next_unmount(fsh_int_t *fshi, vfs_t *vfsp, int flag, cred_t *cr)
1033 {
1034         fsh_fsrecord_t *fsrecp = vfsp->vfs_fshrecord;
1035         int ret;

1037         /*
1038          * The passed fshi is the previous hook (the one from which we've been
1039          * called). We need to find the next one.
1040          */
1041         rw_enter(&fsrecp->fshfsr_lock, RW_READER);
1042         for (fshi = list_next(&fsrecp->fshfsr_list, fshi); fshi != NULL;
1043             fshi = list_next(&fsrecp->fshfsr_list, fshi)) {
1044                 if (fshi->fshi_hooks.unmount != NULL)
1045                         if (fshi_hold(fshi))
1046                                 break;
1047         }
1048         rw_exit(&fsrecp->fshfsr_lock);

1050         if (fshi == NULL)
1051                 return ((*vfsp->vfs_op->vfs_unmount)(vfsp, flag, cr));

1053         ret = (*fshi->fshi_hooks.unmount)(fshi, fshi->fshi_hooks.arg,
1054             vfsp, flag, cr);
1055         fshi_rele(fshi);
1056         return (ret);
1117 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   25828 Mon Sep  9 17:14:59 2013
new/usr/src/uts/common/io/fsd/fsd.c
Update from fsd_sep3 webrev to fsd_sep9
**********************************************************
```

```
/*
 * 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.
 */

/*
 * Filesystem disturber pseudo-device driver.
 */

#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/file.h>
#include <sys/fsd.h>
#include <sys/fsh.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
#include <sys/list.h>
#include <sys/mkdev.h>
#include <sys/refstr.h>
#include <sys/stat.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
#include <sys/types.h>

/*
 * TODO:
 * - add checking if a file descriptor passed by the client is indeed
 *      a mountpoint (we'd like to avoid disturbing / instead of an
 *      unmounted filesystem)
 */
/*
 * fsd - filesystem disturber
 *
 * 1. Abstract
 * Filesystem disturber is a pseudo-device driver used to inject pathological
 * behaviour into vfs calls. It is NOT a fuzzer. That kind of behaviour
 * should be expected and correctly handled by software. A simple example of
 * such behaviour is read() reading less bytes than it was requested. It's
 * well documented and every read() caller should check the return value of
 * this function before proceeding.
 *
 * 2. Features
 * * per-vfs injections
 * * injection installing on every newly mounted vfs (that's called an
 *   omnipresent disturber)
 *
 * 3. Usage
 * fsd_t is a structure which contains all the parameters for the disturbers.
 * This structure is shared by all hooks on a vfs_t.
 *
 * fsd_info_t is filled out by a call to ioctl() and it provides basic
```



```
 * information about fsd's current status.
 *
 * fsd_dis_t is passed to ioctl() when a request to disturb a filesystem is
 * made. It's just a descriptor of a representative file and an fsd_t structure.
 *
 * fsd_fs_t is a structure filled out by ioctl() call when the client requests a
 * full list of disturbers installed in the system.
 *
 * fsd_ioc_t is an union for different ioctl() commands.
 *
 * ioctl() commands:
 * FSD_ENABLE:
 *      ioctl(fd, FSD_ENABLE);
 *      Enables the fsd. When fsd is enabled, any attemps to detach the driver
 *      will fail.
 *
 * FSD_DISABLE:
 *      ioctl(fd, FSD_DISABLE);
 *      Disables the fsd.
 *
 * FSD_GET_PARAM:
 *      ioctl(fd, FSD_GET_PARAM, ioc);
 *      Get's fsd_t associated with a given filesystem. ioc is fsdioc_mnt when
 *      passed to ioctl(). fsdioc_param is the output.
 *      Errors:
 *              ENOENT - the filesystem is not being disturbed
 *
 * FSD_DISTURB:
 *      ioctl(fd, FSD_DISTURB, ioc);
 *      Installs a disturber on a given filesystem. If a disturber is already
 *      installed on this filesystem, it overwrites it. ioc is fsdioc_dis.
 *      Errors:
 *              EAGAIN - hook limit exceeded
 *              EBADFD - cannot open the file descriptor
 *              EINVAL - parameters are invalid
 *
 * FSD_DISTURB_OFF:
 *      ioctl(fd, FSD_DISTURB_OFF, ioc);
 *      Removes a disturber from a given filesystem. ioc is fsdioc_mnt
 *      Errors:
 *              EBADFD - cannot open the file descriptor
 *              ENOENT - the filesystem is not being disturbed
 *
 * FSD_DISTURB_OMNI:
 *      ioctl(fd, FSD_DISTURB_OMNI, ioc);
 *      Install an omnipresent disturber. It means that whenever a new vfs_t is
 *      being created, this disturber is installed on it. If an omnipresent
 *      disturber is already installed, it overwrites it. ioc is fsdioc_param
 *      Errors:
 *              EINVAL - parameters are invalid
 *
 * FSD_DISTURB_OMNI_OFF:
 *      ioctl(fd, FSD_DISTURB_OMNI_OFF);
 *      Removes the omnipresent disturber. That does NOT mean that filesystems
 *      which are disturbed because of the omnipresent disturber presence in the
 *      past are going to stop being disturbed after this call.
 *
 * FSD_GET_LIST:
 *      ioctl(fd, FSD_GET_LIST, ioc);
 *      Get's a full list of disturbers installed in the system. ioc is
 *      fsdioc_list here. This is a structure with two fields, count and listp.
 *      The count is the number of fsd_fs_t's allocated on the address that
 *      listp is pointing to. There would be at most count fsd_fs_t entries
 *      copied out to the caller. Also, count is set to the number of entries
 *      copied out.
 *
```

```
 128  * FSD_GET_INFO:
 129  *      ioctl(fd, FSD_GET_INFO, ioc);
 130  *      Get's current information about fsd. ioc is fsdioc_info here.
 131  *
 132  * At most one hook is installed per vfs_t, and fsd_t describes all possible
 133  * disturbance methods. Multiple commands using the fsd should somehow cooperate
 134  * in order not to destroy each other efforts in installing disturbers.
 135  *
 136  * 4. Internals
 137  * When fsd_enabled is nonzero, fsd_detach() fails.
 138  *
 139  * These mount callback is used for installing injections on newly mounted
 140  * vfs_t's (omnipresent). The free callback is used for cleaning up.
 140  * vfs_t's (omnipresent).
 141  *
 142  * The list of currently installed hooks is kept in fsd_list.
 143  *
 144  * fsd installs at most one hook on a vfs_t.
 145  *
 146  * Inside fsd_detach, we go through fsd_hooks list. There is no guarantee that
 147  * a hook remove callback (fsd_remove_cb) wouldn't execute inside
 148  * fsh_hook_remove(), thus we can't assume that while walking through fsd_hooks,
 149  * our iterator will be valid, because fsh_hook_remove() could invalidate it.
 150  * That's why fsd_detaching flag is introduced.
 151  *
 152  * 5. Locking
 153  * Every modification of fsd_enable, fsd_hooks, fsd_omni_param and fsd_list is
 154  * protected by fsd_lock.
 155  *
 156  * Hooks use only the elements of fsd_list, nothing else. Before an element of
 157  * fsd_list is destroyed, a hook which uses it is removed. Elements from
 158  * fsd_lists are removed and destroyed in the hook remove callback
 159  * (fsd_remove_cb).
 160  *
 161  * Because of the fact that fsd_remove_cb() could be called both in the context
 162  * of the thread that executes fsh_hook_remove() or outside the fsd, we need to
 163  * use fsd_rem_thread in order not to cause a deadlock. fsh_hook_remove() could
 164  * be called by at most one thread inside fsd (fsd_disturber_remove() holds
 164  * be called by at most one thread inside fsd (fsd_remove_disturber() holds
 165  * fsd_lock). We just have to check inside fsd_remove_cb() if it was called
 166  * from fsh_hook_remove() or not. We use fsd_rem_thread to determine that.
 167  *
 168  * fsd_int_t.fsdi_param is protected by fsd_int_t.fsdi_lock which is an rwlock.
 169  */

 171 /*
 172  * Once a set of hooks is installed on a filesystem, there's no need
 173  * to bother fsh if we want to change the parameters of disturbance.
 174  * Intead, we use fsd_lock to protect the fsd_int_t when it's being
 175  * used or changed.
 176  */
 177 typedef struct fsd_int {
 178         krwlock_t       fsdi_lock;      /* protects fsd_param */
 179         fsd_t           fsdi_param;
 180         fsh_handle_t    fsdi_handle;    /* we use fsh's handle in fsd */
 181         vfs_t           *fsdi_vfsp;
 182         int             fsdi_doomed;
 183         list_node_t     fsdi_node;
 183         list_node_t     fsdi_next;
 184 } fsd_int_t;
_____unchanged_portion_omitted_

 229 /* vnode hooks */
 230 /*
 231  * A pointer to a given fsd_int_t is valid always inside fsh_hook_xxx()
 232  * call, because it's valid until the hooks associated with it are removed.
```



```
 233  * If a hook is removed, it cannot be executing.
 234  */
 235 static void
 236 fsd_hook_pre_read(void *arg, void **instancep, vnode_t **vpp, uio_t **uiopp,
 237         int *ioflagp, cred_t **crp, caller_context_t **ctp)
 235 static int
 236 fsd_hook_read(fsh_int_t *fshi, void *arg, vnode_t *vp, uio_t *uiop,
 237         int ioflag, cred_t *cr, caller_context_t *ct)
 238 {
 239         _NOTE(ARGUNUSED(ioflagp));
 240         _NOTE(ARGUNUSED(crp));
 241         _NOTE(ARGUNUSED(ctp));

 243         fsd_int_t *fsdi = (fsd_int_t *)arg;
 244         uint64_t less_chance;
 240         uint64_t count, less, less_chance;

 246         /*
 247          * It is used to keep an odd number of fsd_rand() calls in every
 248          * fsd_hook_pre_read() call. That is desired because when a range of
 249          * width 2 is set as a parameter, we don't want to make it a constant.
 244          * fsd_hook_read() call. That is desired because when a range of width
 245          * 2 is set as a parameter, we don't want to make it a constant.
 250          * The pseudo-random number generator returns a number with different
 251          * parity with every call. If this function is called in every
 252          * fsd_hook_pre_read() execution even number of times, it would always
 253          * be the same % 2.
 248          * fsd_hook_read() execution even number of times, it would always be
 249          * the same % 2.
 254          */
 255         (void) fsd_rand();

 257         ASSERT((*vpp)->v_vfsp == fsdi->fsdi_vfsp);
 253         ASSERT(vp->v_vfsp == fsdi->fsdi_vfsp);

 259         rw_enter(&fsdi->fsdi_lock, RW_READER);
 260         less_chance = fsdi->fsdi_param.read_less_chance;
 257         less = (uint64_t)fsd_rand() %
 258             (fsdi->fsdi_param.read_less_r[1] + 1 -
 259             fsdi->fsdi_param.read_less_r[0]) + fsdi->fsdi_param.read_less_r[0];
 261         rw_exit(&fsdi->fsdi_lock);

 262         count = uiop->uio_iov->iov_len;
 263         if ((uint64_t)fsd_rand() % 100 < less_chance) {
 264                 extern size_t copyout_max_cached;
 265                 uint64_t r[2];
 266                 uint64_t count, less;
 265                 int ret;

 268                 count = (*uiopp)->uio_iov->iov_len;
 269                 r[0] = fsdi->fsdi_param.read_less_r[0];
 270                 r[1] = fsdi->fsdi_param.read_less_r[1];
 271                 less = (uint64_t)fsd_rand() % (r[1] + 1 - r[0]) + r[0];

 273                 if (count > less) {
 267                 if (count > less)
 274                         count -= less;
 275                         *instancep = kmem_alloc(sizeof (uint64_t), KM_SLEEP);
 276                         *(*(uint64_t **)instancep) = less;
 277                 } else {
 278                         *instancep = NULL;
 279                         return;
 280                 }
 269                 else
 270                         less = 0;
```

```
282                 (*uiopp)->uio_iov->iov_len = count;
283                 (*uiopp)->uio_resid = count;
272                 uiop->uio_iov->iov_len = count;
273                 uiop->uio_resid = count;
284                 if (count <= copyout_max_cached)
285                         (*uiopp)->uio_extflg = UIO_COPY_CACHED;
275                         uiop->uio_extflg = UIO_COPY_CACHED;
286                 else
287                         (*uiopp)->uio_extflg = UIO_COPY_DEFAULT;
288         } else {
289                 *instancep = NULL;
290         }
291 }
277                         uiop->uio_extflg = UIO_COPY_DEFAULT;

293 static int
294 fsd_hook_post_read(int ret, void *arg, void *instance, vnode_t *vp,
295         uio_t *uiop, int oflag, cred_t *cr, caller_context_t *ct)
296 {
297         _NOTE(ARGUNUSED(arg));
298         _NOTE(ARGUNUSED(vp));
299         _NOTE(ARGUNUSED(oflag));
300         _NOTE(ARGUNUSED(cr));
301         _NOTE(ARGUNUSED(ct));

303         if (instance != NULL) {
304                 uint64_t *lessp = instance;
305                 uiop->uio_resid += *lessp;
306                 kmem_free(lessp, sizeof (*lessp));
307         }
279                 ret = fsh_next_read(fshi, vp, uiop, ioflag, cr, ct);
280                 uiop->uio_resid += less;
308         return (ret);
282         }

284         return (fsh_next_read(fshi, vp, uiop, ioflag, cr, ct));
309 }


311 static void
312 fsd_remove_cb(void *arg, fsh_handle_t handle)
313 {
314         _NOTE(ARGUNUSED(handle));

316         fsd_int_t *fsdi = (fsd_int_t *)arg;
317         int fsd_context;

319         mutex_enter(&fsd_rem_thread_lock);
320         fsd_context = fsd_rem_thread == curthread;
321         mutex_exit(&fsd_rem_thread_lock);

323         if (!fsd_context)
324                 mutex_enter(&fsd_lock);

326         ASSERT(MUTEX_HELD(&fsd_lock));

328         if (!fsd_detaching)
329                 list_remove(&fsd_list, fsdi);

331         rw_destroy(&fsdi->fsdi_lock);
332         kmem_free(fsdi, sizeof (*fsdi));

334         fsd_list_count--;
335         if (fsd_list_count == 0)
336                 cv_signal(&fsd_cv_empty);
```



```
338         if (!fsd_context)
339                 mutex_exit(&fsd_lock);
340 }

342 /*
343  * Installs a set of hook with given parameters on a vfs_t.
344  *
345  * It is expected that fsd_lock is being held.
346  *
347  * Returns 0 on success and non-zero if hook limit exceeded.
348  */
349 static int
350 fsd_disturber_install(vfs_t *vfsp, fsd_t *fsd)
327 fsd_install_disturber(vfs_t *vfsp, fsd_t *fsd)
351 {
352         fsd_int_t *fsdi;

354         ASSERT(MUTEX_HELD(&fsd_lock));

356         for (fsdi = list_head(&fsd_list); fsdi != NULL;
357             fsdi = list_next(&fsd_list, fsdi)) {
358                 if (fsdi->fsdi_vfsp == vfsp)
359                         break;
360         }

362         if (fsdi != NULL) {
363                 /* Just change the existing fsd_int_t */
364                 rw_enter(&fsdi->fsdi_lock, RW_WRITER);
365                 (void) memcpy(&fsdi->fsdi_param, fsd,
366                     sizeof (fsdi->fsdi_param));
367                 rw_exit(&fsdi->fsdi_lock);
368         } else {
369                 fsh_t hook = { 0 };

371                 fsdi = kmem_zalloc(sizeof (*fsdi), KM_SLEEP);
372                 fsdi->fsdi_vfsp = vfsp;
373                 (void) memcpy(&fsdi->fsdi_param, fsd,
374                     sizeof (fsdi->fsdi_param));
375                 rw_init(&fsdi->fsdi_lock, NULL, RW_DRIVER, NULL);

377                 hook.arg = fsdi;
378                 hook.pre_read = fsd_hook_pre_read;
379                 hook.post_read = fsd_hook_post_read;
355                 hook.read = fsd_hook_read;
380                 hook.remove_cb = fsd_remove_cb;

382                 /*
383                  * It is safe to do so, because none of the hooks installed
384                  * by fsd uses fsdi_handle nor the fsd_list.
385                  */
386                 fsdi->fsdi_handle = fsh_hook_install(vfsp, &hook);
387                 if (fsdi->fsdi_handle == -1) {
388                         kmem_free(fsdi, sizeof (*fsdi));
389                         rw_destroy(&fsdi->fsdi_lock);
390                         return (-1);
391                 }
392                 list_insert_head(&fsd_list, fsdi);
393                 fsd_list_count++;
394         }
395         return (0);
396 }

398 static int
399 fsd_disturber_remove(vfs_t *vfsp)
375 fsd_remove_disturber(vfs_t *vfsp)
400 {
```

```
401         fsd_int_t *fsdi;

403         ASSERT(MUTEX_HELD(&fsd_lock));

405         for (fsdi = list_head(&fsd_list); fsdi != NULL;
406             fsdi = list_next(&fsd_list, fsdi)) {
407                 if (fsdi->fsdi_vfsp == vfsp)
408                         break;
409         }
410         if (fsdi == NULL || fsdi->fsdi_doomed)
411                 return (ENOENT);

413         fsdi->fsdi_doomed = 1;

415         mutex_enter(&fsd_rem_thread_lock);
416         fsd_rem_thread = curthread;
417         mutex_exit(&fsd_rem_thread_lock);

419         ASSERT(fsh_hook_remove(fsdi->fsdi_handle) == 0);

421         mutex_enter(&fsd_rem_thread_lock);
422         fsd_rem_thread = NULL;
423         mutex_exit(&fsd_rem_thread_lock);

425         return (0);
426 }

428 static void
429 fsd_mount_callback(vfs_t *vfsp, void *arg)
405 fsd_callback_mount(vfs_t *vfsp, void *arg)
430 {
431         _NOTE(ARGUNUSED(arg));

433         int error = 0;

435         mutex_enter(&fsd_lock);
436         if (fsd_omni_param != NULL)
437                 error = fsd_disturber_install(vfsp, fsd_omni_param);
413                 error = fsd_install_disturber(vfsp, fsd_omni_param);
438         mutex_exit(&fsd_lock);

440         if (error != 0) {
441                 refstr_t *mntref;

443                 mntref = vfs_getmntpoint(vfsp);
444                 (void) cmn_err(CE_NOTE, "Installing disturber for %s failed.\n",
445                     refstr_value(mntref));
446                 refstr_rele(mntref);
447         }
448 }

450 /*
451  * Although, we might delete the fsd_free_callback(), it would make the whole
452  * proces less clear. There's a time window between firing free callbacks and
453  * freeing the vfs_t in fsd_disturber_remove() could be called. fsh can
454  * deal with invalid handles (until there is no collision), but we'd like to
455  * have a nice assertion instead.
456  */
457 static void
458 fsd_free_callback(vfs_t *vfsp, void *arg)
459 {
460         _NOTE(ARGUNUSED(arg));

462         fsd_int_t *fsdi;

464         mutex_enter(&fsd_lock);
```



```
465         for (fsdi = list_head(&fsd_list); fsdi != NULL;
466             fsdi = list_next(&fsd_list, fsdi)) {
467                 if (fsdi->fsdi_vfsp == vfsp) {
468                         if (fsdi->fsdi_doomed)
469                                 continue;

471                         fsdi->fsdi_doomed = 1;
472                         /*
473                          * We make such assertion, because fsd_lock is held
474                          * and that means that neither fsd_disturber_remove()
475                          * nor fsd_remove_cb() has removed this hook in
476                          * different thread.
477                          */
478                         mutex_enter(&fsd_rem_thread_lock);
479                         fsd_rem_thread = curthread;
480                         mutex_exit(&fsd_rem_thread_lock);

482                         ASSERT(fsh_hook_remove(fsdi->fsdi_handle) == 0);

484                         mutex_enter(&fsd_rem_thread_lock);
485                         fsd_rem_thread = NULL;
486                         mutex_exit(&fsd_rem_thread_lock);

488                         /*
489                          * Since there is at most one hook installed by fsd,
490                          * we break.
491                          */
492                         break;
493                 }
494         }
495         /*
496          * We can't write ASSERT(fsdi != NULL) because it is possible that
497          * there was a concurrent call to fsd_disturber_remove() or
498          * fsd_detach().
499          */
500         mutex_exit(&fsd_lock);
501 }

503 static void
504 fsd_enable()
505 {
506         mutex_enter(&fsd_lock);
507         fsd_enabled = 1;
508         mutex_exit(&fsd_lock);
509 }
_____unchanged_portion_omitted_


520 /* Entry points */
521 static int
522 fsd_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
523 {
524         minor_t instance;
525         fsh_callback_t cb = { 0 };

527         if (cmd != DDI_ATTACH)
528                 return (DDI_FAILURE);

530         if (fsd_devi != NULL)
531                 return (DDI_FAILURE);

533         instance = ddi_get_instance(dip);
534         if (ddi_create_minor_node(dip, "fsd", S_IFCHR, instance,
535             DDI_PSEUDO, 0) == DDI_FAILURE)
536                 return (DDI_FAILURE);
537         fsd_devi = dip;
```

```
538         ddi_report_dev(fsd_devi);

540         list_create(&fsd_list, sizeof (fsd_int_t),
541             offsetof(fsd_int_t, fsdi_node));
464             offsetof(fsd_int_t, fsdi_next));

543         fsd_rand_seed = gethrtime();

545         mutex_init(&fsd_lock, NULL, MUTEX_DRIVER, NULL);
546         mutex_init(&fsd_rem_thread_lock, NULL, MUTEX_DRIVER, NULL);
547         cv_init(&fsd_cv_empty, NULL, CV_DRIVER, NULL);

549         cb.fshc_mount = fsd_mount_callback;
550         cb.fshc_free = fsd_free_callback;
472         cb.fshc_mount = fsd_callback_mount;
551         cb.fshc_arg = fsd_omni_param;
552         fsd_cb_handle = fsh_callback_install(&cb);
553         if (fsd_cb_handle == -1) {
554                 /* Cleanup */
555                 list_destroy(&fsd_list);
556                 cv_destroy(&fsd_cv_empty);
557                 mutex_destroy(&fsd_rem_thread_lock);
558                 mutex_destroy(&fsd_lock);
559                 ddi_remove_minor_node(fsd_devi, NULL);
560                 fsd_devi = NULL;
561                 return (DDI_FAILURE);
562         }

564         return (DDI_SUCCESS);
565 }

567 /*
568  * If fsd_enable() was called and there was no subsequent fsd_disable() call,
569  * detach will fail.
570  */
571 static int
572 fsd_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
573 {
574         fsd_int_t *fsdi;

576         if (cmd != DDI_DETACH)
577                 return (DDI_FAILURE);

579         ASSERT(dip == fsd_devi);

581         /*
582          * No need to hold fsd_lock here. Since only the hooks and callbacks
583          * might be running at this point.
584          */
585         if (fsd_enabled)
586                 return (DDI_FAILURE);

588         ddi_remove_minor_node(dip, NULL);
589         fsd_devi = NULL;

591         /*
592          * 1. Remove the hooks.
593          * 2. Remove the callbacks.
594          *
595          * This order has to be preserved, because of the fact that
596          * fsd_free_callback() is the last stop before a vfs_t is destroyed.
597          * Without it, this might happen:
598          *                vfs_free()                        fsd_detach()
599          * 1.   Handle for the hook is
600          *      invalidated.
601          * 2.   Fired fsd_remove_cb().
```



```
602          * 3.   fsd_remove_cb() hasn't yet    fsd_lock is acquired.
603          *      acquired the fsd_lock.
604          * 4    Waiting for fsd_lock. That    ASSERT(fsh_hook_remove(..) == 0);
605          *      means that the hook hasn't    failed, because the handle is
606          *      been removed from fsd_hooks   already invalid.
607          *      fsd_hooks yet.
608          *
609          * The ASSERT() here is nice and without a good reason, we don't want
610          * to get rid of it.
611          */
612         mutex_enter(&fsd_lock);
613         /*
614          * After we set fsd_detaching to 1, hook remove callback (fsd_remove_cb)
615          * won't try to remove entries from fsd_list.
616          */
617         fsd_detaching = 1;
618         while ((fsdi = list_remove_head(&fsd_list)) != NULL) {
515         while ((fsdi = list_remove_head(&fsd_list)) != NULL)
619                 if (fsdi->fsdi_doomed == 0) {
620                         fsdi->fsdi_doomed = 1;

622                         mutex_enter(&fsd_rem_thread_lock);
623                         fsd_rem_thread = curthread;
624                         mutex_exit(&fsd_rem_thread_lock);

626                         /*
627                          * fsd_lock is held, so no other thread could have
628                          * removed this hook.
629                          */
630                         ASSERT(fsh_hook_remove(fsdi->fsdi_handle) == 0);

632                         mutex_enter(&fsd_rem_thread_lock);
633                         fsd_rem_thread = NULL;
634                         mutex_exit(&fsd_rem_thread_lock);
635                 }
636         }

638         while (fsd_list_count > 0)
639                 cv_wait(&fsd_cv_empty, &fsd_lock);
640         mutex_exit(&fsd_lock);
641         cv_destroy(&fsd_cv_empty);

643         ASSERT(fsh_callback_remove(fsd_cb_handle) == 0);
644         if (fsd_omni_param != NULL) {
645                 kmem_free(fsd_omni_param, sizeof (*fsd_omni_param));
646                 fsd_omni_param = NULL;
647         }

649         /* After removing the callback and hooks, it is safe to remove these */
650         list_destroy(&fsd_list);
651         mutex_destroy(&fsd_rem_thread_lock);
652         mutex_destroy(&fsd_lock);

654         return (DDI_SUCCESS);
655 }
_____unchanged_portion_omitted_

716 static int
717 fsd_ioctl_disturb(fsd_ioc_t *ioc, int mode, int *rvalp)
718 {
719         file_t *file;
720         fsd_dis_t dis;
721         int rv;

723         if (ddi_copyin(&ioc->fsdioc_dis, &dis, sizeof (dis), mode))
724                 return (EFAULT);
```

```
726         if ((rv = fsd_check_param(&dis.fsdd_param)) != 0) {
727                 *rvalp = rv;
728                 return (0);
729         }

731         if ((file = getf((int)dis.fsdd_mnt)) == NULL) {
732                 *rvalp = EBADFD;
733                 return (0);
734         }

736         mutex_enter(&fsd_lock);
737         rv = fsd_disturber_install(file->f_vnode->v_vfsp, &dis.fsdd_param);
633         rv = fsd_install_disturber(file->f_vnode->v_vfsp, &dis.fsdd_param);
738         mutex_exit(&fsd_lock);

740         releasef((int)dis.fsdd_mnt);

742         if (rv != 0)
743                 *rvalp = EAGAIN;
744         else
745                 *rvalp = 0;

747         return (0);
748 }
_____unchanged_portion_omitted_

875 static int
876 fsd_ioctl_disturb_off(fsd_ioc_t *ioc, int mode, int *rvalp)
877 {
878         file_t *file;
879         int64_t fd;

881         if (ddi_copyin(&ioc->fsdioc_mnt, &fd, sizeof (fd), mode))
882                 return (EFAULT);

884         if ((file = getf((int)fd)) == NULL) {
885                 *rvalp = EBADFD;
886                 return (0);
887         }

889         mutex_enter(&fsd_lock);
890         *rvalp = fsd_disturber_remove(file->f_vnode->v_vfsp);
786         *rvalp = fsd_remove_disturber(file->f_vnode->v_vfsp);
891         releasef((int)fd);
892         mutex_exit(&fsd_lock);

894         return (0);
895 }
_____unchanged_portion_omitted_


923 static int
924 fsd_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp,
925         int *rvalp)
926 {
927         _NOTE(ARGUNUSED(dev));
928         _NOTE(ARGUNUSED(credp));

930         int enabled;

932         mutex_enter(&fsd_lock);
933         enabled = fsd_enabled;
934         mutex_exit(&fsd_lock);

936         if (!enabled && cmd != FSD_ENABLE) {
```



```
826         if (!fsd_enabled && cmd != FSD_ENABLE) {
937                 *rvalp = ENOTACTIVE;
938                 return (0);
939         }

941         switch (cmd) {
942         case FSD_ENABLE:
943                 fsd_enable();
944                 *rvalp = 0;
945                 return (0);

947         case FSD_DISABLE:
948                 fsd_disable();
949                 *rvalp = 0;
950                 return (0);

952         case FSD_GET_PARAM:
953                 return (fsd_ioctl_get_param((fsd_ioc_t *)arg, mode, rvalp));

955         case FSD_DISTURB:
956                 return (fsd_ioctl_disturb((fsd_ioc_t *)arg, mode, rvalp));

958         case FSD_DISTURB_OFF:
959                 return (fsd_ioctl_disturb_off((fsd_ioc_t *)arg, mode, rvalp));

961         case FSD_DISTURB_OMNI:
962                 return (fsd_ioctl_disturb_omni((fsd_ioc_t *)arg, mode, rvalp));

964         case FSD_DISTURB_OMNI_OFF:
965                 mutex_enter(&fsd_lock);
966                 if (fsd_omni_param != NULL)
967                         kmem_free(fsd_omni_param, sizeof (*fsd_omni_param));
968                 fsd_omni_param = NULL;
969                 mutex_exit(&fsd_lock);

971                 *rvalp = 0;
972                 return (0);

974         case FSD_GET_LIST:
975                 return (fsd_ioctl_get_list((fsd_ioc_t *)arg, mode, rvalp));

977         case FSD_GET_INFO:
978                 return (fsd_ioctl_get_info((fsd_ioc_t *)arg, mode, rvalp));

980         default:
981                 return (ENOTTY);
982         }
983 }
_____unchanged_portion_omitted_
```

```
**********************************************************
    2022 Mon Sep  9 17:14:59 2013
new/usr/src/uts/common/sys/fsh.h
Update from fsd_sep3 webrev to fsd_sep9
**********************************************************
```

```
/*
 * 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_cb)(void *, fsh_handle_t);

        /* vnode */
        void (*pre_read)(void *, void **, vnode_t **, uio_t **, int *,
                cred_t **, caller_context_t **);
        int (*post_read)(int, void *, void *, vnode_t *, uio_t *, int, cred_t *,
        int (*read)(fsh_int_t *, void *, vnode_t *, uio_t *, int, cred_t *,
                caller_context_t *);
        void (*pre_write)(void *, void **, vnode_t **, uio_t **, int *,
                cred_t **, caller_context_t **);
        int (*post_write)(int, void *, 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 */
        void (*pre_mount)(void *, void **, vfs_t **, vnode_t **,
                struct mounta **, cred_t **);
        int (*post_mount)(int, void *, void *, vfs_t *, vnode_t *,
                struct mounta *, cred_t *);
        void (*pre_unmount)(void *, void **, vfs_t **, int *, cred_t **);
        int (*post_unmount)(int, void *, void *, vfs_t *, int, cred_t *);
        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;
```

_____unchanged_portion_omitted_


```
/* 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
}
```

_____unchanged_portion_omitted_