```
#if !defined(__IMAIL_H)
#define __IMAIL_H

#define IMAIL_MAX_USERS 32
#define IMAIL_NAME_LEN  32
#define IMAIL_PASS_LEN  32
#define IMAIL_SUBJ_LEN  64
#define IMAIL_BODY_LEN  384
#define IMAIL_MSG_LEN   (IMAIL_NAME_LEN + IMAIL_SUBJ_LEN + IMAIL_BODY_LEN + 20)

/*
 * For simplicity, we compare the contents of this structure exactly
 * rather than treating the two fields as strings, thus the values must
 * be zero-padded before passing them into the kernel.
 */
typedef struct Imail_auth_t Imail_auth_t;
struct Imail_auth_t {
    char name[IMAIL_NAME_LEN];
    char passphrase[IMAIL_PASS_LEN];
};

/*
 * For simplicity, we compare the contents of the recipient field of this
 * structure exactly rather than treating it as a string, thus the name
 * must be zero-padded before passing it into the kernel.
 */
typedef struct Imail_header_t Imail_header_t;
struct Imail_header_t {
    char recipient[IMAIL_NAME_LEN];
    char subject[IMAIL_SUBJ_LEN];
};

/* See asm/ioctl.h for IOCTL macro definitions. */

/* authenticate a user */
#define IMAIL_AUTH_USER   _IOW('X', 0x00, struct Imail_auth_t)
#define IMAIL_SET_PASS    _IOW('X', 0x01, struct Imail_auth_t)
#define IMAIL_WRITE_MSG   _IOW('X', 0x02, struct Imail_header_t)
#define IMAIL_DELETE_MSG  _IO( 'X', 0x03)

/* IOCTL's performed only by Imail administrator. */
#define IMAIL_ADD_USER    _IOW('X', 0x80, struct Imail_auth_t)
#define IMAIL_DELETE_USER _IOW('X', 0x81, struct Imail_auth_t)

#endif /* __IMAIL_H */
```

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/capability.h>
#include <linux/fs.h>
#include <linux/poll.h>
#include <linux/rwsem.h>
#include <linux/slab.h>

#include <asm/io.h>
#include <asm/uaccess.h>

#include <linux/Imail.h>


#define IMAIL_DEV_NAME "Imail"   /* device name to be used in registration */


/*
 * LOCK ORDERING RULES FOR THIS MODULE (when >1 needed):
 *   user list r/w semaphore (user_list_rwsem)
 *   user data structure semaphore (sem)
 *
 * NOTE: Only one user's lock should be held at a time to avoid cycles
 * in lock acquisition.
 */


/*
 * The Imail_msg_t is used internally to hold messages for Imail users.
 * Messages are dynamically allocated when a user requests to write one
 * (via the IMAIL_WRITE_MSG ioctl command), at which point the header
 * is filled in.  Subsequent writes fill the body, and when the user
 * requests that it be sent (via fsync), closes the Imail descriptor,
 * or starts a new message, the message is delivered to the recipient.
 * If a user is deleted, the pending message is freed immediately, as
 * are any messages delivered to that user.  Otherwise, message memory
 * is reclaimed when a user deletes a message (via the IMAIL_DELETE_MSG
 * ioctl command).
 */
typedef struct Imail_msg_t Imail_msg_t;
struct Imail_msg_t {
    Imail_msg_t* next;              /* next message in list  */
    int length;                     /* length of body        */
    char recipient[IMAIL_NAME_LEN];  /* name of recipient     */
    char body[IMAIL_MSG_LEN];        /* body of message       */
};


/*
 * When a message is written/read, it is formatted as shown below:

From   : Steve Lumetta
Subject: Imail message formatting
(body)

+ the name in the from line is padded with spaces to make it
  IMAIL_NAME_LEN chararacters long, and terminated with LF
+ the subject ine is padded to IMAIL_SUBJ_LEN characters, terminated with LF

*/


/*
 * Data structure for each Imail user.  A single structure is statically
 * allocated and cannot be deleted--this structure holds the data for
 * the Imail administrator.  All other user structures are dynamically
 * allocated and form a linked list.  Access to the list is protected
 * by a reader/writer semaphore, user_list_rwsem.
 */
typedef struct Imail_user_data_t Imail_user_data_t;
struct Imail_user_data_t {
    /*
     * This part of the data structure (including the list structure)
     * is synchronized with user_list_rwsem.
     */
    Imail_user_data_t*  next;        /* link to next user             */
    Imail_auth_t        user_auth;   /* user authentication data      */

    /* Wait queues contain their own synchronization variables. */
    wait_queue_head_t   read_queue;  /* wait queue for new messages   */

    /*
     * The semaphore below is used to synchronize access to the rest
     * of the data structure.  Reading the values of active_file and
     * msg_list is allowed without the semaphore (to simplify the
     * implementation of waiting for messages), but dereferencing them
     * is not.  Note that checking for user deletion (active_file is NULL
     * while the files private_data points to this structure) MUST be
     * done with the semaphore to avoid races with the deletion code.
     */
    struct semaphore    sem;         /* for user data synchronization */
    struct file*        active_file; /* non-NULL when authenticated   */
    Imail_msg_t*        msg_list;    /* mailbox (linked list)         */
    Imail_msg_t*        writing;     /* message being written         */
};


/* module entry/exit and system call declarations */
static int Imail_init (void);
static void Imail_exit (void);
static ssize_t Imail_read (struct file*, char*, size_t, loff_t*);
static ssize_t Imail_write (struct file*, const char*, size_t, loff_t*);
static unsigned int Imail_poll (struct file*, struct poll_table_struct*);
static int Imail_ioctl (struct inode*, struct file*, unsigned int,
                        unsigned long);
static int Imail_release (struct inode*, struct file*);
static int Imail_fsync (struct file*, struct dentry*, int datasync);


/* module system call jump table */
static struct file_operations Imail_fops;


/* static variables for module */
static kdev_t Imail_dev;                /* assigned major device number   */
static Imail_user_data_t user_list;     /* user list head (administrator) */
static DECLARE_RWSEM (user_list_rwsem); /* user list r/w semaphore        */


/*--------------------------------------------------------------------------*
 * The following functions are basic operations and tests on Imail user    *
 * data structures.  They appear first to allow inlining by the compiler.  *
 *--------------------------------------------------------------------------*/

/*
 * Imail_user_data_init
 *   DESCRIPTION: Initialize the data for a new user; does NOT initialize
 *                authentication data.
```

```
 *   INPUTS: udata -- an uninitialized user
 *   OUTPUTS: udata -- the newly initialized user
 *   RETURN VALUE: none
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: User must NOT be accessible from user list during
 *                    this function; link user into list AFTER
 *                    initialization.
 */
static void
Imail_user_data_init (Imail_user_data_t* udata)
{
    udata->next = NULL;
    init_MUTEX (&udata->sem);
    udata->active_file = NULL;
    udata->msg_list = NULL;
    udata->writing = NULL;
    init_waitqueue_head (&udata->read_queue);
}


/*
 * Imail_user_data_free
 *   DESCRIPTION: Free all dynamically allocated data associated with a
 *                user (but not the user structure).
 *   INPUTS: udata -- the user structure
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECTS: frees all messages from list and writing
 *   SYNCHRONIZATION: User and/or messages must NOT be accessible to other
 *                    code during this function.
 */
static void
Imail_user_data_free (Imail_user_data_t* udata)
{
    Imail_msg_t* msg;     /* message to be freed */

    /* Free all messages in message list. */
    while (NULL != (msg = udata->msg_list)) {
        udata->msg_list = msg->next;
        kfree (msg);
    }

    /*
     * Discard any pending message and issue a warning--message should
     * have been sent or deleted before this point.
     */
    if (NULL != udata->writing) {
        printk ("Imail deleting unsent message.\n");
        kfree (udata->writing);
    }
}


/*
 * Imail_user_is_administrator
 *   DESCRIPTION: Checks whether a specified user is the administrator.
 *                Any caller with system administrator privileges also
 *                has Imail administrative privileges.  Otherwise, the
 *                caller must have authenticated as the Imail administrator
 *                (and udata points to the static element at the head of
 *                the user list).
 *   INPUTS: udata -- the user structure, or NULL if not authenticated
 *   OUTPUTS: none
 *   RETURN VALUE: 1 if the user has administrative privileges, 0 otherwise
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: none necessary; udata pointer is not dereferenced
 */
static int
Imail_user_is_administrator (const Imail_user_data_t* udata)
{
    /*
     * Either the Imail administrator or the system administrator.
     * Note that we check system administrator capability REGARDLESS
     * of the state of the calling process, so a system administrator
     * can be unauthenticated or even authenticated as a user and
     * subsequently deleted and still exercise administrator privileges
     * with the same open file descriptor.
     */
    return (udata == &user_list || capable (CAP_SYS_ADMIN));
}


/*
 * Imail_user_auth_matches
 *   DESCRIPTION: Checks whether an authorization structure provided
 *                matches that of a given user.
 *   INPUTS: udata -- the user structure to be checked against
 *           auth -- the authentication data provided
 *   OUTPUTS: none
 *   RETURN VALUE: 1 if the data match, 0 otherwise
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: Acquire a lock (read or write) on the user list
 *                    before calling this function.
 */
static int
Imail_user_auth_matches (const Imail_user_data_t* udata,
                         const Imail_auth_t* auth)
{
    return (0 == memcmp (&udata->user_auth, auth, sizeof (*auth)));
}


/*
 * Imail_user_name_matches
 *   DESCRIPTION: Checks whether a name provided matches that of a
 *                given user.
 *   INPUTS: udata -- the user structure to be checked against
 *           name -- the name provided
 *   OUTPUTS: none
 *   RETURN VALUE: 1 if the name matches, 0 otherwise
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: Acquire a lock (read or write) on the user list
 *                    before calling this function.
 */
static int
Imail_user_name_matches (const Imail_user_data_t* udata,
                         const char name[IMAIL_NAME_LEN])
{
    return (0 == memcmp (&udata->user_auth.name, name, IMAIL_NAME_LEN));
}


/*---------------------------------------------------------------------*
 * The following two functions are used by the kernel to install and   *
 * uninstall the Imail module.                                         *
 *---------------------------------------------------------------------*/

/*
 * Imail_init
 *   DESCRIPTION: Initialize the Imail module.  Creates a single
```

```
 *               user, who is also the administrator.
 *   INPUTS: none
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success, error number from register_chrdev on failure
 *   SIDE EFFECTS: registers the Imail pseudo-device as a character device
 *                 with dynamically-allocated major number; prints a message
 *                 with the allocated major number
 *   SYNCHRONIZATION: called atomically by kernel, using the big kernel lock
 *                    to prevent devices from being opened before this code
 *                    finishes
 */
static int __init
Imail_init ()
{
    int rval;
    int i;

    Imail_user_data_init (&user_list);
    memset (&user_list.user_auth, 0, sizeof (user_list.user_auth));
    strcpy (user_list.user_auth.name, "administrator");
    strcpy (user_list.user_auth.passphrase, "ece398");

    if ((rval = register_chrdev (0, IMAIL_DEV_NAME, &Imail_fops)) < 0)
        return rval;

    Imail_dev = MKDEV (rval, 0);
    printk ("Registered Imail as device %d.\n", rval);

    return 0;
}


/*
 * Imail_exit
 *   DESCRIPTION: Remove the Imail module.  Frees all dynamically allocated
 *                memory associated with the module.
 *   INPUTS: none
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECTS: destroys user list and messages
 *   SYNCHRONIZATION: The kernel is responsible for ensuring that no files
 *                    are actively using the module (linux-2.4 and later).
 */
static void
Imail_exit ()
{
    Imail_user_data_t* udata;   /* user data to be freed */

    /* Free any messages in administrator's data. */
    Imail_user_data_free (&user_list);

    /* Walk through list of users, freeing each one's data. */
    while (user_list.next != NULL) {
        udata = user_list.next;
        Imail_user_data_free (udata);
        user_list.next = udata->next;
        kfree (udata);
    }

    /*
     * If the unregister call fails, we can't do anything except panic
     * and bring down the kernel, which seems like overkill.
     */
    (void)unregister_chrdev (MAJOR (Imail_dev), IMAIL_DEV_NAME);
}


/*-------------------------------------------------------------------------*
 * The functions below are helper functions for the Imail system calls.    *
 *-------------------------------------------------------------------------*/

/*
 * Imail_find_user
 *   DESCRIPTION: Find a user by name.
 *   INPUTS: name -- the name of the user sought
 *   OUTPUTS: none
 *   RETURN VALUE: the user's data, or NULL if no such user exists
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: Acquire a lock (read or write) on the user list
 *                    before calling this function.
 */
static Imail_user_data_t*
Imail_find_user (const char name[IMAIL_NAME_LEN])
{
    Imail_user_data_t* udata;   /* user data structure for search */

    /* Attempt to find the requested user in the list. */
    for (udata = &user_list; NULL != udata; udata = udata->next) {

        /* Compare the name provided with that of the user. */
        if (Imail_user_name_matches (udata, name))
            return udata;
    }

    /* No luck: return NULL. */
    return NULL;
}


/*
 * Imail_grab_pending
 *   DESCRIPTION: Extract a pending message out of a user's data structure
 *                for delivery or discard (in the case of user deletion).
 *   INPUTS: udata -- the user's data
 *   OUTPUTS: udata -- the user's data after message is removed
 *   RETURN VALUE: the message, or NULL if none was being written
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: Acquire the user data semaphore (udata->sem)
 *                    before calling this function.
 */
static Imail_msg_t*
Imail_grab_pending (Imail_user_data_t* udata)
{
    Imail_msg_t* pending;     /* the pending message */

    pending = udata->writing;
    udata->writing = NULL;
    return pending;
}


/*
 * Imail_deliver
 *   DESCRIPTION: Deliver a message to its intended recipient, appending
 *                it to the end of the recipient's message list.
 *   INPUTS: msg -- the message to be delivered
 *           have_list_lock -- non-zero if the caller has a lock (read or
 *                             write) on the user list
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success, or -ENOENT if recipient does not exist
```

```
 *   SIDE EFFECTS: message appended to recipient's message list
 *   SYNCHRONIZATION: May be called with or without a user list lock.
 *                    The second argument specifies the state on entry.
 *                    If no lock is held already, a read lock is acquired
 *                    on the user list.  If the recipient exists, the
 *                    user data semaphore for the recipient is acquired for
 *                    appending the message to the recipient's message list.
 */
static int
Imail_deliver (Imail_msg_t* msg, int have_user_list_lock)
{
    Imail_user_data_t* udata;   /* recipient's user data structure  */
    Imail_msg_t** find;         /* used to find end of message list */

    /* Lock the user list to find the recipient. */
    if (!have_user_list_lock)
        down_read (&user_list_rwsem);

    /*
     * Find the user based on the name in the message.  If no recipient
     * is found, throw it away and return an error.
     */
    if (NULL == (udata = Imail_find_user (msg->recipient))) {
        up_read (&user_list_rwsem);
        kfree (msg);
        return -ENOENT;
    }

    /*
     * Lock the user's data, add the message to the end of the user's
     * list, wake up any programs waiting for new messages for this
     * user, and unlock the data.
     */
    down (&udata->sem);
    for (find = &udata->msg_list; NULL != (*find); find = &(*find)->next);
    *find = msg;
    wake_up_interruptible (&udata->read_queue);
    up (&udata->sem);

    /* Release the user list lock if acquired by this function. */
    if (!have_user_list_lock)
        up_read (&user_list_rwsem);

    /* Return success. */
    return 0;
}


/*
 * nul_to_space_cpy
 *   DESCRIPTION: Copy bytes from one place to another, transforming ASCII
 *                NUL characters into spaces.  Used in formatting messages.
 *   INPUTS: src -- pointer to source buffer
 *           len -- number of bytes to copy
 *   OUTPUTS: dst -- destination buffer
 *   RETURN VALUE: none
 *   SIDE EFFECTS: none
 *   SYNCHRONIZATION: none required per se
 */
static void
nul_to_space_cpy (char* dst, const char* src, size_t len)
{
    char c;   /* character to copy */

    /* Copy characters one by one. */
    while (len-- > 0) {

        /* Transform NUL to SPACE */
        if ('\0' == (c = *src++))
            c = ' ';
        *dst++ = c;
    }
}


/*-------------------------------------------------------------------------*
 * The functions below are the Imail system calls.  The ioctl commands are *
 * each implemented as a separate function.                                *
 *-------------------------------------------------------------------------*/

/*
 * Imail_read
 *   DESCRIPTION: Implements the read system call for Imail.  Data are
 *                read from the first message in the user's list (only).
 *                Copies data to buffer and advances file pointer.
 *   INPUTS: f -- the file structure used for reading
 *           buf -- the user's buffer
 *           len -- the length of the buffer
 *           offp -- pointer to the file offset
 *   OUTPUTS: buf -- buffer with data copied from first message
 *            offp -- updated file pointer
 *   RETURN VALUE: number of bytes read on success
 *                 -EPERM if user not authenticated or deleted after
 *                        authentication
 *                 -EAGAIN when no message available (non-blocking I/O only)
 *                 -EINTR when signal received
 *                 -EFAULT if buffer is bad
 *   SIDE EFFECTS: advances file pointer; may sleep
 *   SYNCHRONIZATION: Obtains the associated user's user data semaphore.
 *                    May sleep on user data's read_queue waiting for a
 *                    message to arrive.
 */
static ssize_t
Imail_read (struct file* f, char* buf, size_t len, loff_t* offp)
{
    Imail_user_data_t* udata;   /* user data structure */
    Imail_msg_t* msg;           /* message to be read  */
    ssize_t n_read;             /* bytes read          */
    loff_t off;                 /* offset              */

    /* If user has not authenticated yet, permission is denied. */
    if (NULL == (udata = f->private_data))
        return -EPERM;

    /* Loop to wait for data to be available for reading. */
    while (1) {

        /* Start by obtaining user data lock and checking for user deletion. */
        down (&udata->sem);
        if (NULL == udata->active_file) {
            up (&udata->sem);
            return -EPERM;
        }

        /* Have data to read?  Break out of this loop. */
        if (NULL != (msg = udata->msg_list))
            break;

        /*
         * No data yet.  Release semaphore and either return failure (for
```

```
         * non-blocking I/O) or wait for a message or user deletion.
         */
        up (&udata->sem);
        if (0 != (f->f_flags & O_NONBLOCK))
            return -EAGAIN;
        if (wait_event_interruptible
            (udata->read_queue, (NULL == udata->active_file ||
                                 NULL != udata->msg_list)))
            return -ERESTARTSYS; /* got a signal */
        /*
         * ERESTARTSYS prevents use of signals to break out of system calls.
         * The semantics are taken from BSD.  To change this property, use
         * sigaction and turn off SA_RESTART for the signal used to kick
         * system calls out of blocking.
         */
    }

    /*
     * We have an undeleted user, a message to read, and a semaphore to
     * protect us.  Note that the default kernel seek functions do not
     * allow file offsets < 0.
     */
    n_read = 0;
    off = *offp;
    if (msg->length > off) {
        /* There are bytes left to read--read as many as fit in buffer. */
        if (len < (n_read = msg->length - off))
            n_read = len;

        /* Copy the bytes into the user's buffer. */
        if (copy_to_user (buf, msg->body + off, n_read))
            n_read = -EFAULT;
        else
            (*offp) = off + n_read;
    }

    /*
     * Release the semaphore and return the number of bytes read
     * (or error number).
     */
    up (&udata->sem);
    return n_read;
}


/*
 * Imail_write
 *   DESCRIPTION: Implements the write system call for Imail.  Data are
 *                written to a message allocated with ioctl.  Copies data
 *                to message and increases message length.
 *   INPUTS: f -- the file structure used for reading
 *           buf -- the user's buffer
 *           len -- the length of the buffer
 *           offp -- IGNORED
 *   OUTPUTS: none
 *   RETURN VALUE: number of bytes written on success
 *                 -EPERM if user not authenticated or deleted after
 *                        authentication
 *                 -EIO when no message is being written currently
 *                 -EFBIG when no space is left in the current message
 *                 -EFAULT if buffer is bad
 *   SIDE EFFECTS: updates message body length
 *   SYNCHRONIZATION: Obtains the associated user's user data semaphore.
 *                    May sleep with semaphore if user buffer is not
 *                    in memory (page fault).
 */
static ssize_t
Imail_write (struct file* f, const char* buf, size_t len, loff_t* offp)
{
    Imail_user_data_t* udata;   /* user data structure   */
    Imail_msg_t* msg;           /* message to be written */
    ssize_t n_written;          /* bytes written         */

    /* If user has not authenticated yet, permission is denied. */
    if (NULL == (udata = f->private_data))
        return -EPERM;

    /* Get the user data semaphore. */
    down (&udata->sem);

    /* Make checks, then write. */
    n_written = 0;
    if (NULL == udata->active_file) {
        /* User was deleted after authentication. */
        n_written = -EPERM;
    } else if (NULL == (msg = udata->writing)) {
        /* User has not started writing a message. */
        n_written = -EIO;
    } else if (0 >= (n_written = (IMAIL_MSG_LEN - msg->length))) {
        /* No space left to write. */
        n_written = -EFBIG;
    } else {
        /* Limit to characters provided. */
        if (len < n_written)
            n_written = len;

        /* Copy the bytes from the user's buffer. */
        if (copy_from_user (msg->body + msg->length, buf, n_written))
            n_written = -EFAULT;
        else
            msg->length += n_written;
    }

    /*
     * Release the semaphore and return the number of bytes written
     * (or error number).
     */
    up (&udata->sem);
    return n_written;
}


/*
 * Imail_poll
 *   DESCRIPTION: Implements the poll and select system calls for Imail.
 *                Imail pseudo-devices are always writable if write
 *                permission is held, and readable if read permission is
 *                held and a message is available.
 *   INPUTS: f -- the file structure used for reading
 *           ptab -- the poll table used to collect wait queues
 *   OUTPUTS: none
 *   RETURN VALUE: mask of permitted operations on success
 *                 POLLERR if user not authenticated or deleted after
 *                         authentication
 *   SIDE EFFECTS: adds user's read queue to the poll table
 *   SYNCHRONIZATION: The user data semaphore is acquired only to check for
 *                    user deletion, although user's msg_list is read
 *                    (not dereferenced) for read permission check.
 *                    Associating our read queue with the poll table
 *                    before the check should suffice to handle race
```

```
 *                   conditions; if message is delivered after poll_wait,
 *                   poll should not sleep; messages delivered before
 *                   poll wait (and not deleted before check) are seen
 *                   by check.
 */
static unsigned int
Imail_poll (struct file* f, struct poll_table_struct* ptab)
{
    Imail_user_data_t* udata;   /* user data structure   */
    unsigned int mask;          /* return value          */

    /* If user has not authenticated yet, return an error. */
    if (NULL == (udata = f->private_data))
        return POLLERR;

    /* Tell poll to wake up if our user's read queue is signalled. */
    poll_wait (f, &udata->read_queue, ptab);

    /*
     * Need the semaphore to check whether user was deleted.
     * If so, poll will remove our wait queue from the table
     * before returning, and udata will not be deleted before
     * poll returns.
     */
    down (&udata->sem);
    if (NULL == udata->active_file) {
        up (&udata->sem);
        return POLLERR;
    }
    up (&udata->sem);

    /* Calculate mask of available operations. */
    mask = 0;
    if (0 != (f->f_mode & FMODE_READ) && NULL != udata->msg_list)
        mask |= (POLLIN | POLLRDNORM);
    if (0 != (f->f_mode & FMODE_WRITE))
        mask |= (POLLOUT | POLLWRNORM);

    return mask;
}


/*
 * Imail_authenticate
 *   DESCRIPTION: Authenticate a user based on the name and password provided.
 *                Only unauthenticated users are allowed to authenticate.
 *                Those that have been booted off via user deletion must
 *                close all descriptors to ensure atomic deletion of the user
 *                data structure.  Reauthentication is technically possible,
 *                but introduces some complexity in the sense that any
 *                operations on the old user must be  switched over to the
 *                new user or cancelled.  It's cleaner to disallow
 *                reauthentication.
 *   INPUTS: f -- file structure on which ioctl was called
 *           arg -- a pointer to the user's authentication data
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
 *                 -EPERM if file was already authenticated, deleted after
 *                        authentication
 *                 -EFAULT if the authentication data buffer was bad
 *                 -EBUSY if the requested user is in use by another file
 *                 -EACCES if authentication data match no user in list
 *   SIDE EFFECTS: points f->private_data to the user data structure if
 *                 authentication succeeds
 *   SYNCHRONIZATION: Obtains a read lock on the user list.  If authentication
 *                    succeeds for some user, obtains that user's user data
 *                    semaphore.
 */
static int
Imail_authenticate (struct file* f, void* arg)
{
    Imail_auth_t auth;          /* authentication data         */
    Imail_user_data_t* udata;   /* user data structure         */
    int rval;                   /* return value                */

    /* Authenticated already or been booted?  Close and re-open. */
    if (NULL != (udata = f->private_data))
        return -EPERM;

    /* Read authentication structure into kernel address space. */
    if (copy_from_user (&auth, arg, sizeof (auth)))
        return -EFAULT;

    /* We need to read authentication data from the user list. */
    down_read (&user_list_rwsem);

    /* Attempt to find the requested user in the list. */
    for (udata = &user_list; NULL != udata; udata = udata->next) {
        /*
         * Compare the authentication data provided with those of
         * the user.
         */
        if (Imail_user_auth_matches (udata, &auth)) {

            /* We have a match!  Try to install this file as the active one. */
            down (&udata->sem);

            /* Check if already in use. */
            if (NULL != udata->active_file)
                rval = -EBUSY;
            else {
                /* Record the user information in both directions. */
                udata->active_file = f;
                f->private_data = udata;
                rval = 0;
            }

            /* We're done. */
            up (&udata->sem);
            up_read (&user_list_rwsem);
            return rval;
        }
    }

    /* No luck in authenticating; permission denied! */
    up_read (&user_list_rwsem);
    return -EACCES;
}


/*
 * Imail_set_pass
 *   DESCRIPTION: Set a user's password.  Changing a user's password
 *                requires that the entity making the change either
 *                authenticate first as that user or have Imail administrative
 *                privileges.
 *   INPUTS: f -- file structure on which ioctl was called
 *           arg -- a pointer to a user's name and new password
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
```

```
 *                -EFAULT if the authentication data buffer was bad
 *                -ENOENT if the named user does not exist
 *                -EPERM if the attempt to change was illegal
 *   SIDE EFFECTS: may change authentication data for a user
 *   SYNCHRONIZATION: Obtains a write lock on the user list.
 *   SECURITY: Note that the error values in this function confirm user
 *             existence.  If this type of leak is undesirable, return
 *             -EPERM instead of -ENOENT.
 */
static int
Imail_set_pass (struct file* f, void* arg)
{
    Imail_auth_t auth;          /* authentication data         */
    Imail_user_data_t* udata;   /* user data structure         */
    Imail_user_data_t* target;  /* target user data structure  */
    int rval;                   /* return value                */

    /* Read authentication structure into kernel address space. */
    if (copy_from_user (&auth, arg, sizeof (auth)))
        return -EFAULT;

    /*
     * We may need to change authentication data, so we write lock the
     * user list.
     */
    down_write (&user_list_rwsem);

    /* Assume success. */
    rval = 0;
    udata = f->private_data;

    /* Find the user by name; if no match is found, return an error. */
    if (NULL == (target = Imail_find_user (auth.name))) {
        rval = -ENOENT;
    } else if (!Imail_user_is_administrator (udata) && udata != target) {
        /*
         * Changing a password for any user but yourself requires
         * administrator privileges.  Keep in mind that the file f
         * may be unauthenticated, in which case f->private_data is
         * NULL, but the caller is the system administrator.
         */
        rval = -EPERM;
    } else {
        /* Change the password by overwriting the authentication structure. */
        memcpy (&target->user_auth, &auth, sizeof (auth));
    }

    /* All done.  Release the lock and return. */
    up_write (&user_list_rwsem);
    return rval;
}


/*
 * Imail_write_msg
 *   DESCRIPTION: Prepare to write an Imail message.  Allocate a message
 *                buffer and attach it to the user's data structure.  If
 *                the user was already writing a message, send it to its
 *                recipient.
 *   INPUTS: f -- file structure on which ioctl was called
 *           arg -- a pointer to a message header
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
 *                 -EPERM if user not authenticated or deleted after
 *                        authentication
 *                -EBADF if write permission not held on file
 *                -EFAULT if the authentication data buffer was bad
 *                -ENOMEM if the kernel runs out of memory (!)
 *                -ENOENT if a previous message could not be delivered
 *   SIDE EFFECTS: delivers any message currently being written
 *   SYNCHRONIZATION: Obtains a user list read lock to check for existence
 *                    of intended recipient.  Obtains the associated user's
 *                    user data semaphore.  If message must be delivered,
 *                    obtains both a user list read lock and the recipient's
 *                    semaphore.
 */
static int
Imail_write_msg (struct file* f, void* arg)
{
    Imail_header_t header;      /* header for new message         */
    Imail_user_data_t* udata;   /* user data structure            */
    Imail_msg_t* msg;           /* message to be delivered        */
    int rval;                   /* return value                   */
    char* wpos;                 /* write position in message body */

    /* Initialize variables an check for user authentication. */
    if (NULL == (udata = f->private_data))
        return -EPERM;
    msg = NULL;
    rval = 0;

    /* Check that the file is writable. */
    if (0 == (f->f_mode & FMODE_WRITE))
        return -EBADF;

    /* Read header structure into kernel address space. */
    if (copy_from_user (&header, arg, sizeof (header)))
        return -EFAULT;

    /*
     * Check for recipient.  Recipient MAY be deleted between this check
     * and message creation, but could also be deleted at any point between
     * this call and the call that delivers the message.  This check works
     * for the more common case of writing to someone who has never existed.
     */
    down_read (&user_list_rwsem);
    if (NULL == Imail_find_user (header.recipient))
        rval = -ENOENT;
    up_read (&user_list_rwsem);

    /*
     * Lock the user data, check whether this user was deleted after
     * authentication, extract any pending messages for delivery, and
     * allocate a new message for writing.  In any case, unlock at the end.
     */
    down (&udata->sem);
    /* Check for user deletion. */
    if (NULL == udata->active_file)
        rval = -EPERM; /* overrides absent recipient error */
    else if (rval == 0) {
        /* does not create message unless recipient existed */
        msg = Imail_grab_pending (udata);
        udata->writing = kmalloc (sizeof (Imail_msg_t), GFP_KERNEL);
        if (NULL == udata->writing)
            rval = -ENOMEM;
        else {
            udata->writing->next = NULL;
            memcpy (&udata->writing->recipient, header.recipient,
                    sizeof (header.recipient));
```

```
            /* Write a header into the message body. */
            udata->writing->length = 20 + sizeof (udata->user_auth.name) +
                                     sizeof (header.subject);
            wpos = udata->writing->body;
            memcpy (wpos, "From   : ", 9);
            wpos += 9;
            nul_to_space_cpy (wpos, udata->user_auth.name,
                              sizeof (udata->user_auth.name));
            wpos += sizeof (udata->user_auth.name);
            memcpy (wpos, "\nSubject: ", 10);
            wpos += 10;
            nul_to_space_cpy (wpos, header.subject, sizeof (header.subject));
            wpos += sizeof (header.subject);
            memcpy (wpos, "\n", 1);
        }
    }
    up (&udata->sem);

    /*
     * Deliver any pending message from the user.  If we are already
     * returning an error, we ignore errors in delivery.  Otherwise,
     * we report them.
     */
    if (NULL != msg) {
        if (rval == 0)
            rval = Imail_deliver (msg, 0);
        else
            (void)Imail_deliver (msg, 0);
    }

    return rval;
}


/*
 * Imail_delete_msg
 *   DESCRIPTION: Delete the first message in a user's list of messages.
 *                Deleting incoming messages is part of reading, so this
 *                command requires read permission on the file.
 *   INPUTS: f -- file structure on which ioctl was called
 *           arg -- a pointer to a message header
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
 *                 -EPERM if user not authenticated or deleted after
 *                        authentication
 *                 -EBADF if read permission not held on file
 *   SIDE EFFECTS: frees the memory associated with the deleted message
 *   SYNCHRONIZATION: Obtains the associated user's user data semaphore.
 */
static int
Imail_delete_msg (struct file* f, void* arg)
{
    Imail_user_data_t* udata;   /* user data structure   */
    Imail_msg_t* discard;       /* message to be deleted */
    int rval;                   /* return value          */

    /* Initialize variables an check for user authentication. */
    if (NULL == (udata = f->private_data))
        return -EPERM;
    discard = NULL;
    rval = 0;

    /* Check that the file is readable. */
    if (0 == (f->f_mode & FMODE_READ))
        return -EBADF;

    /*
     * Lock the user data and remove the first message from
     * the message list.  If the user was deleted, we return an error.
     */
    down (&udata->sem);
    if (NULL == udata->active_file)
        rval = -EPERM;
    else {
        if (NULL != (discard = udata->msg_list))
            udata->msg_list = discard->next;
        f->f_pos = 0; /* used to track read position in message */
    }
    up (&udata->sem);

    /* Free the memory outside of the critical section. */
    if (NULL != discard)
        kfree (discard);

    return rval;
}


/*
 * Imail_add_user
 *   DESCRIPTION: Add a new user to the list.   Adding a user requires
 *                that the caller have Imail administrative privileges.
 *   INPUTS: f -- file structure on which ioctl was called
 *           arg -- a pointer to the new user's name and password
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
 *                 -EPERM if no administrative privileges are held
 *                 -EFAULT if the authentication data buffer was bad
 *                 -EEXIST if the named user already exists
 *                 -ENOMEM if the kernel runs out of memory (!)
 *   SIDE EFFECTS: may allocate new user structure and insert into user list
 *   SYNCHRONIZATION: Obtains a write lock on the user list.
 */
static int
Imail_add_user (struct file* f, void* arg)
{
    Imail_user_data_t* udata;   /* user data structure            */
    int rval;                   /* return value                   */
    Imail_auth_t auth;          /* new user's authentication data */

    /* Assume success. */
    rval = 0;

    /* This command is only allowed to administrators. */
    if (!Imail_user_is_administrator (f->private_data))
        return -EPERM;

    /* Read authentication structure into kernel address space. */
    if (copy_from_user (&auth, arg, sizeof (auth)))
        return -EFAULT;

    /* Write lock the user list. */
    down_write (&user_list_rwsem);

    /* Try to add the user. */
    if (NULL != Imail_find_user (auth.name)) {
        /* User exists; we can't add another with the same name. */
        rval = -EEXIST;
    } else {
        /* Allocate a user data block. */
```

```
        udata = kmalloc (sizeof (Imail_user_data_t), GFP_KERNEL);
        if (NULL == udata)
            rval = -ENOMEM;
        else {
            /* Copy the authentication data into place. */
            memcpy (&udata->user_auth, &auth, sizeof (auth));

            /* Initialize the rest of the structure. */
            Imail_user_data_init (udata);

            /* And link the user into place after the administrator. */
            udata->next = user_list.next;
            user_list.next = udata;
        }
    }

    /* Release our lock and return the right value. */
    up_write (&user_list_rwsem);
    return rval;
}


/*
 * Imail_delete_user
 *   DESCRIPTION: Delete a user from the list.   Deleting a user requires
 *                that the caller have Imail administrative privileges.
 *   INPUTS: f -- file structure on which ioctl was called
 *           arg -- a pointer to the new user's name and password
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
 *                 -EPERM if no administrative privileges are held, or if
 *                        attempt is made to delete the Imail administrator
 *                 -EFAULT if the authentication data buffer was bad
 *                 -ENOENT if the named user does not exist
 *   SIDE EFFECTS: may unlink user data structure from list; may free
 *                 user data structure memory
 *   SYNCHRONIZATION: Obtains a write lock on the user list.  If user
 *                    exists and is active, obtains user's user data
 *                    semaphore and marks as booted from system rather
 *                    than freeing structure.  Structure is then freed
 *                    by Imail_release.
 */
static int
Imail_delete_user (struct file* f, void* arg)
{
    Imail_user_data_t* udata;   /* user data structure            */
    int rval;                   /* return value                   */
    Imail_auth_t auth;          /* new user's authentication data */
    Imail_user_data_t** find;   /* loop index for user removal    */

    /* Assume success. */
    rval = 0;

    /* This command is only allowed to administrators. */
    if (!Imail_user_is_administrator (f->private_data))
        return -EPERM;

    /* Read authentication structure into kernel address space. */
    if (copy_from_user (&auth, arg, sizeof (auth)))
        return -EFAULT;

    /* Write lock the user list. */
    down_write (&user_list_rwsem);

    /* Try to find the user. */
    if (NULL == (udata = Imail_find_user (auth.name))) {
        /* User does not exist. */
        rval = -ENOENT;
    } else if (udata == &user_list) {
        /* Deletion of administrator is not permitted. */
        rval = -EPERM;
    } else {
        /* Discard user's messages and remove them from the user list. */
        Imail_user_data_free (udata);
        for (find = &user_list.next; (*find) != udata; find = &(*find)->next);
        *find = udata->next;

        if (NULL == udata->active_file) {
            /* Not active--just delete! */
            kfree (udata);
        } else {
            /* User is active.  Boot them off indirectly. */
            udata->active_file = NULL;
            wake_up_interruptible (&udata->read_queue);
        }
    }

    /* Release our lock and return the right value. */
    up_write (&user_list_rwsem);
    return rval;
}


/*
 * Imail_ioctl
 *   DESCRIPTION: Implements the ioctl system call for Imail.  Specific
 *                effects depend on the command requested (each has its
 *                own function).
 *   INPUTS: idx -- the index node associated with the file
 *           f -- the file structure
 *           cmd -- the ioctl command
 *           arg -- operand for the command
 *   OUTPUTS: see command functions
 *   RETURN VALUE: -ENOTTY for unknown command
 *                 other values returned directly from command function
 *   SIDE EFFECTS: see command functions
 *   SYNCHRONIZATION: see command functions
 */
static int
Imail_ioctl (struct inode* idx, struct file* f, unsigned int cmd,
             unsigned long arg)
{
    /*
     * Authentication is required for all most commands, while some
     * require Imail administrator privileges.  System administrator
     * privileges suffice for the administrative commands (with or
     * without authentication as the Imail administrator).  User-level
     * commands require authentication, since they operate on the
     * user data structure.  See command functions for more detail.
     */

    /* Switch on command. */
    switch (cmd) {
        case IMAIL_AUTH_USER:   return Imail_authenticate (f, (void*)arg);
        case IMAIL_SET_PASS:    return Imail_set_pass     (f, (void*)arg);
        case IMAIL_WRITE_MSG:   return Imail_write_msg    (f, (void*)arg);
        case IMAIL_DELETE_MSG:  return Imail_delete_msg   (f, (void*)arg);
        case IMAIL_ADD_USER:    return Imail_add_user     (f, (void*)arg);
        case IMAIL_DELETE_USER: return Imail_delete_user  (f, (void*)arg);
        default:                return -ENOTTY; /* invalid command */
```

```
    }
}


/*
 * Imail_release
 *   DESCRIPTION: Closes an Imail file for the last time.  Release is only
 *                called when the last operation finishes on a file, so we
 *                know that nothing else is still working with this
 *                particular file.
 *   INPUTS: idx -- the index node associated with the file
 *           f -- the file structure
 *   OUTPUTS: none
 *   RETURN VALUE: 0 always; ignored by kernel
 *   SIDE EFFECTS: delivers an pending message to its recipient; frees
 *                 user data structure if user was deleted after
 *                 authentication
 *   SYNCHRONIZATION: Obtains the associated user's user data semaphore
 *                    in order to extract the message.  Obtains both a
 *                    user list read lock and the recipient's semaphore
 *                    in order to deliver a message.
 */
static int
Imail_release (struct inode* idx, struct file* f)
{
    Imail_user_data_t* udata;   /* user data structure             */
    int i;                      /* loop index for finding user     */
    Imail_msg_t* msg;           /* message pending from user       */

    /*
     * Initialize variables.  If file's owner had not authenticated yet,
     * we do nothing.
     */
    if (NULL == (udata = f->private_data))
        return 0;
    msg = NULL;

    /*
     * We need a lock on the user data to prevent another file from
     * slipping in and authenticating for this user while we're
     * cleaning up.
     */
    down (&udata->sem);

    /* Were we properly authenticated, or booted via user deletion? */
    if (NULL == udata->active_file) {
        /* Booted: delete user data.  The lock is being deleted, too. */
        kfree (udata);
    } else {
        /*
         * Properly authenticated: capture any pending message, then
         * release user data for next file.
         */
        msg = Imail_grab_pending (udata);
        udata->active_file = NULL;
        up (&udata->sem);
    }

    /*
     * Release is atomic for this file.  After returning from our
     * function, f will be added to the kernel's free list of files, but
     * we don't want pointers to our data floating around in the system.
     */
    f->private_data = NULL;

    /* Deliver any pending message, ignoring failure. */
    if (NULL != msg)
        (void)Imail_deliver (msg, 0);

    /*
     * Always succeeds; the return value is ignored by the kernel, anyway.
     * If we want to return errors, we need to use the flush method rather
     * than only catching release.
     */
    return 0;
}


/*
 * Imail_fsync
 *   DESCRIPTION: Implements the fsync and fdatasync system calls for Imail.
 *                Extracts any pending message from the user and delivers it
 *                to its recipient.
 *   INPUTS: f -- the file structure used for reading
 *           dent -- the directory entry
 *           datasync -- 1 for datasync call, 0 for sync call
 *   OUTPUTS: none
 *   RETURN VALUE: 0 on success
 *                 -EPERM if user is not authenticated or has been deleted
 *                        after authentication
 *                 -ENODATA if no message was being written
 *                 -ENOENT if recipient of message does not exist
 *   SIDE EFFECTS: removes message from user and delivers to recipient
 *   SYNCHRONIZATION: Obtains the associated user's user data semaphore
 *                    in order to extract the message.  Obtains both a
 *                    user list read lock and the recipient's semaphore
 *                    in order to deliver the message.
 */
static int
Imail_fsync (struct file* f, struct dentry* dent, int datasync)
{
    Imail_user_data_t* udata;   /* user data structure             */
    Imail_msg_t* msg;           /* message pending from user       */

    /* If user has not authenticated yet, permission is denied. */
    if (NULL == (udata = f->private_data))
        return -EPERM;

    /* Capture any pending message for delivery. */
    down (&udata->sem);
    if (NULL == udata->active_file) {
        /* User was deleted after authentication. */
        up (&udata->sem);
        return -EPERM;
    }
    msg = Imail_grab_pending (udata);
    up (&udata->sem);

    /* Try to deliver; return failure or success to caller. */
    if (NULL != msg)
        return Imail_deliver (msg, 0);

    /* No message was being written! */
    return -ENODATA;
}


/*---------------------------------------------------------------------*
 * The remainder of the file is module linkage.                        *
 *---------------------------------------------------------------------*/
```

```
/*
 * Use GCC's tagged field initialization to select only those functions
 * supported.
 */
static struct file_operations Imail_fops = {
    owner:   THIS_MODULE,
    read:    Imail_read,
    write:   Imail_write,
    poll:    Imail_poll,
    ioctl:   Imail_ioctl,
    release: Imail_release,
    fsync:   Imail_fsync,
};

module_init (Imail_init);
module_exit (Imail_exit);
EXPORT_NO_SYMBOLS;

MODULE_AUTHOR ("Steven S. Lumetta");
MODULE_LICENSE ("see file header");
```