ECE391: Computer Systems Engineering **Lecture Notes Set 6**

I-mail: An Example of Device Driver Design (Part 1 of 2)

I-mail, or Illinois Mail, provides instant message services in the Linux kernel. Putting an instant message module into an operating system is silly in that centralizing instant messages within a single machine renders them far less useful and in that moving the functionality from user-level into the kernel serves little direct purpose. However, I-mail serves as an excellent example of driver design because it is moderately complex (around 1,000 lines of code) and raises many of the driver design issues encountered in other communication drivers (*e.g.*, the Transmission Control Protocol and Internet Protocols, TCP/IP, on which most Internet applications are built) but with a much simpler protocol. As a result, we can describe I-mail development in its entirety and illustrate the challenges in writing a correct driver without getting lost in the details of a network protocol or a hardware interface specification.

This set of notes begins with a description of a general process for designing a device driver, then follows that process in order to design I-mail.

Note: The second part of these notes will cover creation of helper functions and their synchronization, then give some examples and describe I-mail unit test. As of this writing, I have yet to port I-mail to Linux 2.6, so I have made the code available to you for reading but have not given out the sources, test code, and so forth.

The Device Driver Design Process

The steps below outline the process of designing a device driver :

0. Contemplate security. Security is hard or impossible to add in correctly as an afterthought.
1. Write descriptions of all operations in terms of the visible interface (the file operations structure).
2. Design the data structures.
3. Pick a locking strategy: organize data into sets protected by locks.
4. Determine what types of locks should be used and where they are stored, then define a lock ordering.
5. Identify blocking conditions and events that cause blocked tasks to wake up.
6. Consider dynamic allocation issues and hazards.
7. Write the code.
8. Write subfunctions and synchronization rules for them (in comments).
9. Return to Step 3 or Step 4 if Step 7 or Step 8 fails.
10. Write unit tests for the driver.

Many software engineering experts argue that tests should be written before code—just before Step 7, for example. Although doing so can be helpful in developing the code, tests should also be written so as to cover all code paths in an actual implementation, which requires that an implementation exist. Even if the implementation changes later, these tests form a fairly solid core, whereas tests written prior to implementation are more likely to overlook subtle corner cases, simply because the programmer has yet to consider these cases.

Security and I-mail

Security is important to all device drivers. Most modern operating systems do not protect themselves from driver code, thus a security vulnerability in a device driver is a vulnerability in the system as a whole. Security issues are often subtle and may constrain both implementation choices and possible functionality. For these reasons, *security should be one of the topics considered first in any system design*. Once a system has been designed and implemented, trying to eliminate vulnerabilities and information leaks or to cleanly separate the functionality available according to user roles becomes much more difficult, if not impossible without a redesign.

Historically, Unix security was built around a set of users each with a name (their login identity, akin to your NetID) and a unique numeric identifier. The system also defined groups as sets of users, and each user had a default group. The access control system was then built into the file system. Each file was marked with the owning user and an associated group as well as a set of permission bits, also called **permissions**, **privileges**, or **access rights**.[1]

[1] And sometimes capabilities, but that term usually means something more general: a right that can be transferred, withdrawn, expired, and so forth.

Permission bits encoded the abilities to read, write, and execute a file (for directories, execute is used for traversal), with separate bits for the file's owner, users in the file's group, and all other users. The system also had a few special bits to allow users to inherit the file's owner id (called "set user identity," or **setuid**) or the file's group id (called "set group identity," or **setgid**). You might have noticed that the setuid bit is set on the `mntdrives` command in your virtual machine, for example, which allows you to mount your directories with your user account. Since mounting disks requires root, the executable is marked setuid and owned by root. When your user account executes the file, the program runs as though root had executed it.

In the old model, all administrative rights were concentrated into a single superuser (root) role. The superuser could effectively do almost anything. The superuser need not have all rights, however. In most systems, for example, administrators cannot obtain user passwords directly. Passwords are typically stored as one-way hashes of the unencrypted form, and having access to the hashed result does not enable someone to reverse the hash and obtain the original. The value here is not to protect against attacks by the machine's administrator, but rather to prevent someone who compromises the machine or obtains the stored password file from easily obtaining passwords for all users.

Being able to change a user's password, on the other hand, is a useful capability to allow the administrator: when users forget their passwords, the administrator can be asked to change the user's password to something memorable, such as "changeme." The user can then change the password again so that the administrator does not retain direct login access.

More modern systems implement **separation of privileges**, which refers to the use of fine-grained privileges rather than a coarse-grained or even binary scheme. For example, a system might have a web administrator who has complete control over the web server pages, active content (programs) executed by the server, and perhaps even virtual domain web hosting. But the web administrator has no rights to manage e-mail, file transfers, logins, and so forth.

The idea of a web administrator leads to **role-based access control**, which is the idea of separating roles from individuals and groups. By distinguishing roles, one allows a more flexible assignment of rights. Using the old Unix model, one can approximate this idea by creating a "webadmin" group and assigning the web server privileges to that group using a few tricks in the filesystem (for example). Users in that group then obtain the privileges of the web administrator role.

In the context of I-mail, we have two roles: user and administrator. A user can send and receive messages, but has no other rights with respect to the system nor to the messages' of others. The administrator, on the other hand, is responsible for creating (and removing) users. We separate the I-mail administrator role from the root account on the machine: the root account can always act as I-mail administrator, but this role can also be handed to another person without giving away any privileges other than I-mail management functions.

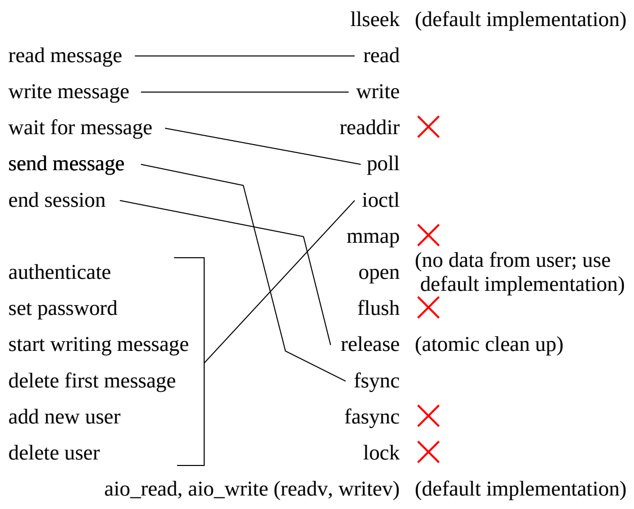
I-mail Operations

Now that we have a strategy for managing security, we can start to identify the operations supported by I-mail and think about how they will map into system calls. Abstractly, a user has authentication data, an association with a program—we allow only one file at a time, a list of messages (their mailbox), and possibly a message being written.

How should users authenticate themselves in order to access I-mail? Two general schemes are possible. As a modern engineering student, you probably use both forms almost every day. If you have a debit or credit card, for example, you know that you must authenticate yourself for each purchase. Even if you've just bought something at a store, you have to re-type your personal identification number (PIN) or re-write your signature if you want to make a second purchase. Unless, of course, you prefer to give your PIN to the store's representative, in which case you can make as many purchases as you want, and perhaps even more than you want. In contrast, when you log into a computer, you type your password once, and the computer associates your login terminal with your user identity. Programs run from your terminal are run using your identity.

Most computer systems allow a single authentication followed by many operations. As an extra level of security, we could refuse authentication to any task not owned by a specific user or group. In this manner, we could prevent users from authoring their own I-mail software, since only programs executing as the special I-mail user identity could access I-mail. Or we could use such an approach to create a strong connection between I-mail users and identities on the machine. We chose to do neither with I-mail: any user can access any I-mail account, and any user can write and execute their own programs to make use of I-mail functionality without further approval from an administrator.

The figure to the right lists the I-mail operations and illustrates their mapping to the file operations structure, which defines the system call entry points available to the driver from user code.



Let's make a list of I-mail operations (shown on the left in the diagram). The first step in using any I-mail functionality is authentication. As already discussed, we want a single operation for this purpose. A block of data containing the user's credentials—their name and password—must be provided to the kernel, and we see no natural match in the file operations, so we map the function as an `ioctl` operation.

When an I-mail user ends their session, we may have cleanup work to do, so we map this work to the `release` function, which is called after all files have been closed and all system calls using the file have returned. *The* `release` *function is thus atomic with respect to all other operations on the file.*

Next are the basic operations for reading and writing messages. A user must be able to write a message and to read a message, which we map intuitively to the `read` and `write` system calls. For convenience, we define these operations on individual messages: the read operation can read part of a message, and the write operation can write part of a message. In this manner, we allow programs more flexibility in creating and delivering messages to users. To support this type of access, however, we need additional operations. For example, we need a way for the user to tell I-mail that an outgoing message is complete and should be delivered. We map this function to `fsync`, which typically ensures that data have been given to the device, in this case the recipient's mailbox. We also need a way to start writing a new message, which becomes another `ioctl`.

Users may also want to wait until a message arrives, allowing a user program to wake up and do something when the kernel delivers a new message. The `poll` entry point supports system calls such as `poll` and `select` that provide this type of support for other devices.

How will a user see their mailbox? For simplicity, and to limit the storage space used for messages in the kernel, we expect messages to be deleted soon after they are read—a user can write user-level software to maintain an archive of their old messages, if they so desire. Rather than allowing users to read from any message, we restrict users to reading from the first message in their mailbox. To read the second or later messages, they must delete the first message. This last regular operation we also map to `ioctl`.

The remaining operations are administrative in nature, and all map into additional `ioctl`s. A user (or the I-mail administrator) may need to change a password. The I-mail administrator may need to add a new user, or may need to delete a user.

Note that we also benefit from several default implementations. Users can seek within the first message using the `llseek` entry point. They can perform vector reads and writes as well, again using default implementations that call `read` and `write` repeatedly.

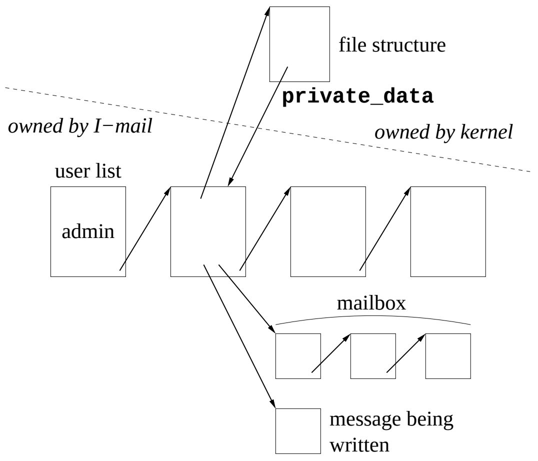
Data Structures in I-mail

The figure at the top of the next page illustrates the data structures necessary for I-mail. Our discussion initially focuses on the data owned by the I-mail driver, which appear below the dotted line in the figure.

The linked list in the center holds one data structure for each user. Recall that I-mail administration privileges should not require superuser privileges on the machine. One natural way to manage the administrator role is then to create a single user data structure for the administrator. The administrator, of course, should never be deleted. We place the admin data structure statically at the head of the user list. Other users are dynamically allocated and linked together. Favoring simplicity over scalability for the purposes of our example, we use a singly-linked list.

©2004-2011 Steven S. Lumetta. All rights reserved.

As a second simplification, we do not maintain the user list across driver installations, including system reboot. In other words, the list of users includes only the I-mail administrator; any others must be added each time the system is booted (or I-mail is reinstalled). A more complete implementation of this type of driver can mirror changes such as user creation, user deletions, and password changes to a file on disk. Then, at boot time, the driver can parse the file to rebuild the user list.



Returning to the figure, notice that each user data structure contains two pointers for message data. One points to a linked list of incoming messages, the user's mailbox. The second points to a message currently being written by the user.

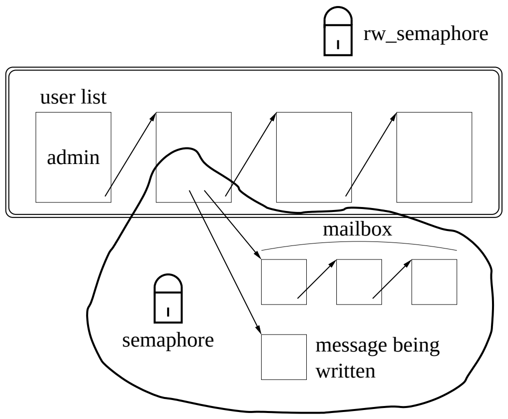
The first step in using I-mail is to authenticate as a specific user, which creates the links shown in the figure between the user data structure and a file structure. After authentication, the user data structure holds a pointer to the corresponding file structure. The file structure is owned by the Linux kernel, but we make use of the `private_data` field in the structure to point back to the corresponding user data structure, allowing the driver routines to quickly find the user data.

The ownership model in the figure raises important questions about how the code can ensure that neither side ever retains pointers to data structures after those structures have been freed by the owning code. In the case of the file structure, recall that the Linux kernel invokes the `release` method in the file's operations structure between the last close (or system call completion) on the file and the actual free of the file structure. I-mail must thus provide this routine and must remove the link from the user data structure to the file whenever `release` is called. Similarly, before I-mail can free a user data structure, it must make sure that any tasks with possible references to the user data from the file structure have been allowed to finish their use and discard the pointer. We return to this topic when we discuss dynamic memory allocation issues.

Locks in I-mail

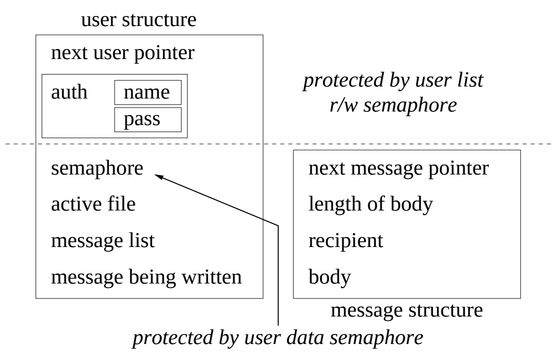
We next consider the use of locks in I-mail. As always, using a single lock for all I-mail operations is an option, but in this case we opt for a more complex design so as to illustrate a range of synchronization issues common to device drivers.

The first step is to think about the I-mail operations and how they relate to the data, then to use this information to break data into groups protected by individual synchronization variables. Many I-mail operations traverse the user list to find the data for specific users. Authentication, user creation and deletion, and message delivery must all search the user list based on a given user name. We expect that the operations that only read the list—authentication and message delivery, for example—to be much more common than those that change the list. We want to allow these operations to execute concurrently, thus we choose to protect the user list with a reader-writer lock. As a virtual device, I-mail has no need for interrupts, so we select a reader-writer semaphore (`rw_semaphore`) to protect the user list, as shown in the figure to the right.



The data for a given user, in contrast, are likely to be read and written frequently as messages are created, written, sent, and read. However, with the exception of message delivery, we expect the operations on a single user's data to be executed by a single process, so we choose simplicity over parallelism in this case by using a single lock for each user's data. As before, we choose a semaphore (`semaphore`) for this purpose rather than a spin lock.

The figure to the right shows the I-mail data structures in more detail, and is marked to indicate which data are protected by which locks. Note that part of the user structures—the linkage and the credentials—is protected by the user list reader-writer semaphore, while the remainder is protected by the user data semaphore in the structure itself. All messages associated with a user, whether in their mailbox or currently being written, are also protected by that user's user data semaphore.



What's the next step? We have more than one lock, but we have yet to specify a lock ordering! As we discuss later, when delivering a message, we need to find the recipient's user data and then deliver the message. In cases such as delivery, when both semaphores are needed, we choose to obtain the user list R/W semaphore first. *No code may ever try to obtain user list R/W semaphore while holding a user data semaphore!* Doing so can create deadlock.

But have we created a total order on our locks? We have not. We have ordered only the user list lock as our first lock. Two user data semaphores are not ordered with respect to one another. How can we order them? We don't even know how many users we have!

Several possibilities exist: we could order by the memory address of the lock; we could order according the the user list; or we could assign unique identifiers and order according to those. These schemes are complex and somewhat error-prone, and they tend to be slow. Sometimes they're the only answer. But in I-mail, we instead assert that *user data semaphores are not ordered with respect to one another*. As a result, *no code should ever acquire more than one user data semaphore*. We will need to be careful: sending a message means moving it from the sender's user data structure (the message being written) into the recipient's mailbox. We simply can't hold both semaphores simultaneously.

As an exercise, you may now want to go through the list of I-mail operations and identify which semaphores are needed for each operation. The table below provides a set of answers.

Operation	User List R/W Semaphore	User Data Semaphore	Comments
read message	no	yes	
write message	no	yes	
wait for message	no	yes	
send message/fsync	(no) read	sender recipient	for removing outgoing message for message delivery
end session/release	(no) *	yes *	for removing associated file from user data see "send message" if message delivery is necessary
authenticate	read	yes	only need user data semaphore if successful
set password	write	no	only changes authentication data—these are protected by user list R/W semaphore
start writing message	read *	yes *	only need user data semaphore if recipient exists see "send message" if message delivery is necessary
delete first message	no	yes	
add new user	write	no	no need for new user's user data semaphore: no task can authenticate while we hold write lock on user list
delete user	write	yes	user data semaphore necessary to ensure that deletion synchronizes with all other operations on the user

The first three lines of the table are fairly straightforward: when a user works with their own mailbox or outgoing message, they acquire the user data semaphore, but do not need the user list lock. The first part of sending a message (using `fsync`) is similar: the pointer to the message being written must be removed from the sender's user data.

Delivering a message is slightly trickier. First, we need to acquire the user list R/W semaphore in order to find the recipient's user data. Assuming that the recipient exists, we need to acquire their user data semaphore. Remember that even though I-mail checks that the message recipient exists when a user starts to write a message, the recipient can be deleted while the message is being written. But does delivery need to retain its read lock on the user list while it inserts the message into the recipient's mailbox? Or might we be able to avoid ever holding more than one lock at a time in I-mail? The answer is that delivery must hold both locks: the existence of the recipient can change as soon as the user list semaphore is released! In other words, if we release the user list semaphore before acquiring the recipient's user data semaphore, the I-mail administrator can delete the recipient before we acquire the user data semaphore. In fact, in such a case, the user data semaphore that our delivery is trying to acquire might reside in memory that has been reused for other purposes.

At this point you might reasonably wonder how we can prevent user deletion from interacting badly with other operations that we have already discussed—reading, writing, and polling, for example. The key difference is that in these cases the user being deleted is active (*i.e.*, the user data structure has an associated file structure), whereas a message recipient may not be active. We discuss safe handling of user deletion and particularly of freeing the user data structures later in these notes. For now, notice (at the bottom of the table) that we require deletion to acquire the user data semaphore for the user being deleted so as to force the task doing the deletion to synchronize with other tasks acting only on the user data.

Ending an I-mail session occurs when the `release` method is called, either immediately after the file descriptor is closed or after the last system call on the file completes. At this point, we need to remove the association between the user data structure and the file structure, which the Linux kernel frees after `release` returns. The user data semaphore suffices for this purpose. If the user has been writing a message, however, I-mail delivers it when the user ends the session, in which case the locks needed for delivery are also needed.

Authentication requires both a read lock on the user list and, assuming that authentication succeeds, a lock on the user data to link the user data to the file structure. Since the user is only active after authentication completes, the two locks must be held simultaneously for the same reason that we discussed in regard to message delivery.

Setting a password requires modifying authentication data, which are protected by the user list R/W semaphore. Thus we need a write lock on the user list, but do not actually need to read nor write any of the user data protected by the user data semaphore. Note that user deletion cannot execute concurrently with a password change, since both require a write lock on the user list.

Starting to write an I-mail message may require delivering a previous message. The operation also verifies the existence of the recipient, which requires a read lock on the user list. Finally, we need the user data semaphore to attach the new message to the user data. The actual ordering of these events can be tricky. For example, let's say that we deliver any previous message, then check the recipient, and finally create the new message. In this case, can we safely assume that the user is not already writing a message when we reach the third step (creating the new message)? No, we cannot. Let's say that our task finds a previous message to deliver. In order to deliver that message, we need the recipient's user data semaphore, and must thus release the user data semaphore for the sender. When we do so, another task executing the same code, *i.e.*, starting a new message, can grab the user data semaphore and insert a new message. Our initial task can then find the slot occupied once it again acquires the sender's user data semaphore. Rather than checking repeatedly under the protection of the semaphore, I-mail first checks the recipient, then atomically swaps the old message with a new one under the protection of the user data semaphore. The old message is then delivered.

Deleting the first message from a user's mailbox is similar to reading or writing a message, and requires only the user data semaphore.

Adding a new user requires a write lock on the user list, but does not require that the new user data semaphore be acquired. However, all user data must be initialized before the user list semaphore is released to prevent other tasks from accessing the user data before it is initialized. If the user data structure were larger or the process longer, we could simply initialize the user data semaphore, acquire it, and release the user list lock, thereby allowing other tasks to walk the user list while our task finished initializing the new user's data. Since the structure is small, I-mail uses the simpler approach.

Except for the subtleties around dynamic allocation of user data, we have already discussed user deletion.

Wait Queues

The next step in our design process requires that we identify blocking conditions and events that cause blocked tasks to wake up. Before doing so, however, we need to understand the mechanisms needed for correctly putting a task to sleep and to examine the mechanisms provided by Linux for this purpose.

When a task goes to sleep, it puts itself into either an **interruptible** (`TASK_INTERRUPTIBLE`) or **non-interruptible** (`TASK_UNINTERRUPTIBLE`) state. The difference refers to whether the task can be woken by a user-level interrupt (a signal) or not. Interruptible tasks can be woken by signals; the signal handler executes in user space, then the task either goes back to sleep or returns from the system call with an error indicating that a signal was received. The behavior can be defined by the program in Linux—see the notes on signals for details.

The uninterruptible task state is needed to ensure that devices that must eventually get attention from the sleeping task cannot be prevented from getting that attention by user code. If, for example, a device is asked to perform a command, and the device requires an acknowledgement to the response that it eventually generates, some task must be responsible for delivering that acknowledgement.

As you probably recall from earlier in the class, one role of an operating system is to provide the illusion of synchronous interactions between programs and devices. Putting a task to sleep while waiting for a device is thus a fairly common operation in device drivers, but the steps necessary to avoid race conditions in this code are somewhat error-prone. Bugs due to such races are unlikely to show up during testing, but can lead to drivers that leave programs permanently asleep (the programs seem to hang).

Linux avoids requiring programmers to write this code at all by providing a **wait queue** abstraction and a set of C preprocessor macros that expand into correct code for putting tasks to sleep. The `wait_queue_head_t` structure forms a doubly-linked lists of tasks waiting for some event. The functions for putting tasks to sleep are written as macros because the test condition for sleeping must be evaluated repeatedly inside the code (passing the value of the condition as an argument won't work).[2] When waiting for an interruptible event to occur, a task should call:

```
int wait_event_interruptible (wait_queue_head_t* wq, <C expression> condition);
```

The code below shows roughly what the `wait_event_interruptible` macro does—some of the code has been replaced with pseudo-code comments for clarity. A similar call is available for uninterruptible waiting.

```
int ret_val = 0;
if (!(condition)) {
    while (1) {
        // start critical section using wq->lock
        //     add current task to wait queue (protected by lock)
        //     (memory barrier)
        //     set task state to TASK_INTERRUPTIBLE
        // end critical section using wq->lock
        if (condition) {
            break;
        }
        if (!signal_pending (current)) {
            schedule (); // sleep
            continue;
        }
        ret = -ERESTARTSYS;
        break; // deliver signal, then maybe return
    }
}
```

Recall that Linux' `schedule` function catches last-minute signals and removes the task from the run queue if no such signal is received. Note that the condition passed into `wait_event_interruptible` is evaluated repeatedly within the code generated by the macro. For this reason, you should *never use expressions that have side effects, such as increment and decrement operators, with macros.* Function calls are also somewhat unsafe if the functions are not idempotent, *i.e.*, calling them repeatedly is no different than calling them once.

[2]Callbacks could be used, but are arguably too expensive and may lead to more error-prone style than do the macros.

When a task changes the condition on which tasks are sleeping in a wait queue, the task should wake the sleeping tasks by calling one of the functions in the table below. Many other variants are available in newer Linux kernels—see `linux/wait.h` for details.[3]

`void wake_up (wait_queue_head_t* wq);`	Wake up all tasks waiting in a wait queue.
`void wake_up_interruptible (wait_queue_head_t* wq);`	Wake up all interruptible tasks (with task state TASK_INTERRUPTIBLE) waiting in a wait queue.

Blocking and Wakeup in I-mail

The main question that we must address when considering blocking issues for a device driver is simply: what operations can block?

For I-mail, the answer is relatively simple. When a user tries to read a message from an empty mailbox, we want to put the task performing the read to sleep until a message is available. This behavior is analogous to that provided by almost all other input devices, and supports the illusion that the devices interact synchronously with the program. When your program asks for user input from a keyboard, for example, the program typically sleeps until the user actually types a full line. The kernel provides a line buffering discipline and handles things like backspace, only delivering a line's worth of data after a user presses the ENTER key. From the point of view of program execution, the user types a response when the program asks for it.

The `poll` operation, which can be used either to check whether a message is available or to wait for one to become available, may also need to go to sleep on an empty mailbox. The condition here is equivalent to reading a message.

What about writing messages? Most real devices do sometimes need to be able to block processes when writing. For most devices, the relative speed of the processor to the device implies that the processor can easily overload the device with data. Status bits from the device or interrupts can be used to inform the processor that the device is ready for more data, but in the meantime a task that wants to write more data can be put to sleep.

I-mail is not a real device, and while tasks executing its operations might be put to sleep when trying to acquire a semaphore or trying to allocate memory, there is no further need to wait for a recipient's mailbox to be ready for more messages, and the write operation need not block explicitly in the driver code.

The next question to consider is when a task that is asleep waiting to read from an empty mailbox should be woken up? Although we have two operations that may block, they are waiting on the same condition: a non-empty mailbox.

One answer is fairly obvious: tasks waiting for a message should be woken up when a message is delivered. We can optimize slightly by waking up tasks only when a message is delivered to an empty mailbox, but the benefit is minor.

A second answer as to when a task must be woken up is perhaps less obvious. If a user is deleted, any tasks waiting for message arrival for that user must be woken up. As we discuss later, freeing the user data as part of deletion must wait for all outstanding operations on the user data to finish, including half-completed operations by sleeping tasks. Given that no messages are delivered to a deleted user, allowing tasks to sleep through deletion of the corresponding user introduces the possibility of their never waking up. User deletion must wake these sleeping tasks.

We add a `wait_queue_head_t` to the user data structure to handle the tasks waiting on an empty mailbox. Wait queues have their own synchronization, a spin lock embedded in the head, so I-mail can make use of this queue without obtaining either type of I-mail semaphore. In fact, we can not allow a task to hold an I-mail semaphore while it sleeps if a task that must eventually wake the first task requires the same semaphore. Delivering a message, for example, requires the user data semaphore. Hence holding that semaphore while sleeping introduces deadlock any time a task goes to sleep.

[3]The semantics and rationale behind the options are not obvious from a quick perusal of the code, however, so be careful about drawing conclusions prematurely. For example, `wake_up` in recent kernels stops waking tasks when it finds one marked as exclusive (`WQ_FLAG_EXCLUSIVE`), but if you mix calls that set the exclusive flag with those that do not, you may also need to carefully control placement of exclusive tasks in the wait queue so that only the correct non-exclusive tasks are woken by the call. Alternatively, the `wake_up_all` call wakes all tasks, including all exclusive tasks.

The pseudo-code below illustrates the general protocol for using a wait queue to put a task to sleep:

```
1. Release all locks.
2. Check sleeping conditions without locks.
3. Go to sleep by calling wait_event_interruptible.
4. When awoken, reacquire necessary locks.
5. Recheck all validity requirements.
6. If the sleeping condition still holds, it was a false alarm: start this process over.
```

The implications of Step 2 are subtle. The condition that you check before going to sleep *must be atomically safe to check*. That is, the data structure can never be in a state in which evaluation of the condition can lead to a crash.

This requirement is substantially more stringent than anything we have seen previously in this class, because the task deciding whether or not to sleep does not have any locks. Using a condition that is safe to check with respect to critical sections is *not adequate*, because the condition does not wait for critical sections to finish. All possible reorderings are also relevant: for any possible interleaving of any code that touches any of the data involved in calculating the sleep condition, including any reordering by the compiler as well as any reordering by the microarchitecture, must not be able to cause the calculation to cause a crash.

The requirement is not impossible to achieve. For example, one can safely read an integer from a memory location (in a data structure that is pinned in physical memory, as is most kernel data, or that can be paged in as needed). Such an integer can then be compared safely with constant values. Adding integer fields together is also safe. Reading a pointer and comparing it to a constant such as NULL is also safe, but dereferencing the same pointer may not be.

Unsafe actions might include dereferencing any pointer, or in general walking across a pointer-based data structure. Unless care is taken to guarantee that all instructions in all critical sections that update such a structure are interleaved with any memory barrier instructions necessary in the corresponding ISA, unexpected intermediate states in the pointers may lead to problems with the condition.

If you do need to perform a complex calculation for you condition, you can restructure your code safely. Let's say, for example, that you need to walk a linked list and calculate something based on the walk to decide whether or not to sleep. Such a condition is unlikely to be safe. Instead, one can add a variable that records the condition's value (as a boolean or an integer) and then, in every critical section that might change the value of the condition, add code to update the condition. The sleeping condition can thus be transformed into integer comparison, which is safe. The critical sections that need to be modified in this way are the same as the ones that must wake the sleeping task, so the process is not really more onerous than is already necessary for correct blocking and wakeup. However, you might still need compiler and memory barriers between structure changes (the list in our example) and the updates of the variable recording the condition.

Dynamic Allocation in I-mail

We are now ready to think about dynamic allocation issues for I-mail. Consider the following questions. What structures are allocated and deallocated dynamically? What interface shall we use for each structure (raw pages, a slab cache, or `kmalloc`)? When are structures allocated? What flags should be used when allocating? When can a structure be deallocated safely, *i.e.*, when can we be sure that no other processor nor data structure is using the structure?

We have two data structures in I-mail: user data and messages. Neither occupies nearly a full page, so we really only need to choose between slab caches and `kmalloc`. Walking the user list is a common operation in I-mail, so a slab cache is perhaps the best choice for user data, as it reduces the number of pages occupied by the users in the list. The same property does not hold for message data: we only walk over all users' mailboxes when shutting down. For simplicity, however, I chose to use `kmalloc` in both cases.

When are structures allocated? A new user is allocated when the I-mail administrator executes an add user operation. A new message is allocated when a user starts writing a new message.

What flags should we use when allocating? I-mail uses no real device, so DMA-accessible memory is not a concern. Is `GFP_KERNEL` or `GFP_ATOMIC` the right choice? Recall that `GFP_ATOMIC` *specifies that the caller can not be put to sleep*. This property is necessary if allocation occurs, for example, while holding a spin lock, or with interrupts blocked, or in an interrupt handler. I-mail uses no spin locks and has no interrupt handlers nor need to block interrupts, so we are safe using `GFP_KERNEL` in all cases. In fact, `GFP_USER` is probably a better choice for message allocation, since we allow normal users to create an unlimited number of messages, which poses a potential denial of service attack if we aren't careful (the code actually uses `GFP_KERNEL`).

What about deallocation? The simple answer is deletion by a user: a message is freed when a user deletes it from their mailbox. But what happens when a user is deleted? That user will never clean out their mailbox, so I-mail must delete all messages in the mailbox before deleting the user. Similarly, when we want to shut I-mail down, we need to clean up all of the dynamically allocated data, including all users and messages. There is one last case: when a message is sent but can't be delivered, we choose to delete the message (and return an error if possible) rather than returning the undeliverable message to the sender.

Are these actions safe? A user holds the user data semaphore when deleting a message, so no other task should have access to the message at that time. Deleting a user also requires the user data semaphore. Linux (2.4 and later) is responsible for ensuring that no files are using a driver when the driver's exit routine is invoked, so we can assume that shutting down is safe. Finally, message delivery removes the message atomically from the sender under the protection of the sender's user data semaphore, at which point the task performing the delivery is the only one that has access to the message.

Now consider user deletion. The simple answer in this case is that we free a user data structure when the I-mail administrator deletes the user. Shutdown is again another case, but a simpler one given the guarantees made by Linux when invoking the driver's shutdown function.

What happens if a user is using I-mail when the user data is deleted? The file structure in this case holds a pointer to the user data. *Changing this pointer to NULL does not help*, since the task may have loaded the user data structure pointer into a register, or may even be waiting to acquire the user data semaphore (for example). We can't just delete the user data! What can we do?

The answer is that we have to wait until we know that no other task can be using the structure, although we can speed these tasks along to completion by indicating that the user has been deleted. The first step is to remove the user from the user list, which prevents tasks from authenticating as the deleted user. If the user is not active at that point, they can be safely deleted and the structure freed.

If the user is active, we defer the deallocation until all user activity finishes. We do not change the pointer in the file structure to avoid data races with other code. We have no lock protecting the file structure's `private_data` field, so changing the field to NULL to remove the user data reference implies that we write any I-mail code that makes use of this pointer with extreme care. In particular, every time we load from the field to obtain the pointer to the user data, we must check for NULL again. Rather than deal with such complexity across the entire driver, user deletion leaves the pointer in the file structure unchanged.

Instead, we assign ownership of the user data (*i.e.*, responsibility for deletion) to the file structure. We know that Linux calls `release` on the file structure after the last use finishes, so we can safely follow the pointer in that routine and perform the deletion.

How can we let other tasks know that the user has been deleted? The user data structure's pointer to the file structure is under the protection of the user data semaphore, so user deletion can safely change it to NULL. After acquiring the user data semaphore, an I-mail operation can check for user deletion by simply comparing the user data structure's file pointer with NULL.

Let's consider races between user deletion and authentication by the deleted user. The two operations require incompatible user list locks, so they can't occur at the same time. The case in which the user has already authenticated when deletion starts has already been discussed. After deletion finishes, the user data is no longer in the user list, so authentication using that structure is impossible.

Can we be more aggressive? Perhaps any task that wakes up and finds that its user data has been marked for deletion can proceed with `kfree`? Unfortunately, no. We can't guarantee that such a task is the last, and we must wait for the last before deallocating the structure. However, that guarantee is provided by `release`.