ECE391
Computer System Engineering

Lecture 27

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2018

Lecture Topics

- I-mail design (step 5+)

Aministrivia

- **MP3.5 Code cutoff:**
  - **5:00pm Monday, December 10**
- Hand-in/demo
  - 0.5 h (all members should be there)
  - (Monday night); Tuesday; Wednesday
- Post time slots the next week
  - First come first served

- Competition
  - Thursday December 13, in the ECE391 Lab

Aministrivia

- **Final Exam:**
  - 8:00am to 11:00am, Friday, December 14
- Review session
  - During the Lecture Time om Thursday, December 6

Driver Design Process

0. contemplate security (hard/impossible to add in correctly as afterthought!)
1. write out all ops (in terms of visible interface)
2. design data structures
3. pick a locking strategy
4. determine locks needed, pick lock ordering
5. consider blocking issues & wakeup

Driver Design Process

6. consider dynamic allocation issues & hazards
7. write code
8. write subfunctions and synchronization rules
9. return to 3 or 4 if 7 or 8 fails
10. unit test

Wait Queues

- Consider a task that can't make progress immediately
  - needs a semaphore
  - needs a page currently on disk
- In these cases
  - task puts itself to sleep
  - rather than wasting CPU cycles (as with spin lock)

Wait Queues

- What actually happens when a task puts itself to sleep?
  - recall TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE states
  - task puts itself into queue and tells scheduler to run something else
  - some other task eventually wakes it up

Wait Queues

- Complexity!
  - race conditions between
    - task checking sleep condition and putting itself to sleep
    - some other task looking at queue and possibly waking it up
  - problems historically
    - led Linux to create macros
    - to keep most programmers from writing the code

- Data structure in Linux: wait queues
  - **`wait_queue_head_t`**
  - forms a doubly-linked list of tasks waiting for some event

Wait Queues

- Waiting for an event

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

  - again, a macro
  - **`condition`** may be evaluated many times
  - returns 0 when condition is true,
    -ERESTARTSYS if interrupted by signal

Wait Queues

```
int ret_val = 0;
if (!(condition)) {
    while (1) {
        // start critical section using (wq->lock)
        //     add 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_val = -ERESTARTSYS;
        break; // deliver signal, then maybe return
    }
}
return ret_val;
```

schedule() call: 1) catches "last-minute" signals and 2) removes task from runqueue

Waking Tasks up

```
void wake_up (wait_queue_head_t* wq);
```

- wake up one task waiting on a wait queue

```
void wake_up_all (wait_queue_head_t* wq);
```

- wake up all tasks waiting on a wait queue

Waking Tasks up

```
void wake_up_interruptible
                        (wait_queue_head_t* wq);
```

– wake up one interruptible task waiting on a wait queue

```
void wake_up_interruptible_all
                        (wait_queue_head_t* wq);
```

– wake up all interruptible tasks waiting on a wait queue

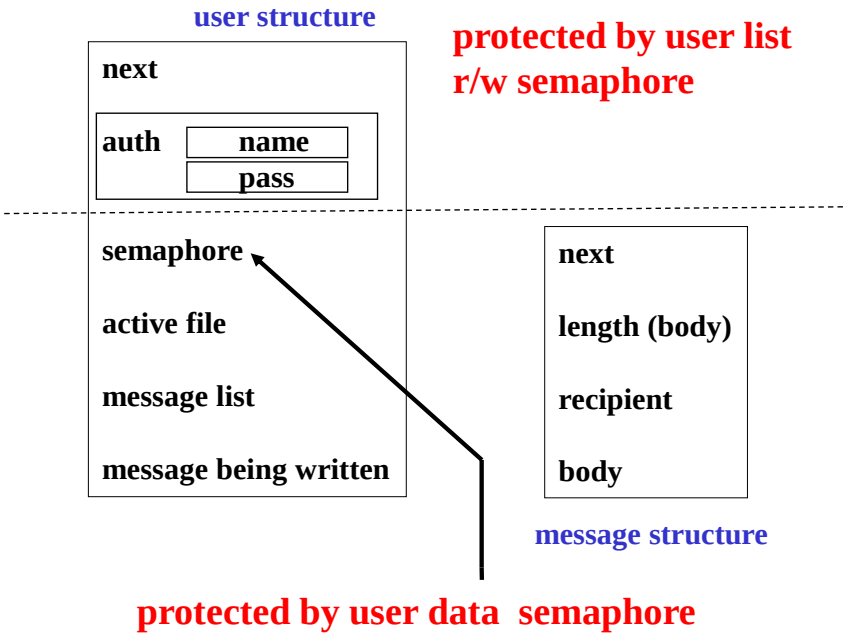
I-mail – Blocking and Wakeup

- What can block?
  - read — wait for a new message
  - poll — wait for a new message
  - write — typically needs to wait for the device, but not in I-mail
  - (so only readers/pollers block in I-mail)

- When are readers (or pollers) awoken?
  - new message delivered
  - user is deleted (nothing is coming now!)

I-mail – Blocking and Wakeup

- Data structure impact
  - add a **`wait_queue_head_t`** to the user data
  - wait queues
    - have their own synchronization
    - so use it without obtaining user data semaphore
    - (process doing the waking and sleeping processes can't all hold one semaphore simultaneously)



I-mail – Blocking and Wakeup

- Protocol for sleeping
  - release all locks
  - check conditions without locks
  - go to sleep
  - when awoken
    - reacquire locks
    - recheck all validity requirements for waking (otherwise go back to sleep; false alarm)

I-mail – Blocking and Wakeup

- Q: What implications does check conditions without locks step have?

- A: condition must be atomically safe, i.e., data structure can never be in a state in which evaluation of the condition can lead to a crash

I-mail – Blocking and Wakeup

- Important to note
  - sleeper does NOT have locks
  - safe with respect to critical sections is NOT adequate

- Example of things that are ok
  - reading an integer (and comparing it with a constant values)
  - adding two integer fields together
  - reading (but not dereferencing) a pointer and comparing it to NULL

I-mail – Blocking and Wakeup

- example of things that might not be OK
  - dereferencing a pointer
  - walking a pointer-based data structure

- in some instances, necessary to restructure
  - critical sections recalculate condition, store as integer
  - code being woken reads integer (always safe)

Example I-mail: `Imail_read()`

```
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.
         */
    }
```

no lock available in unauthenticated state

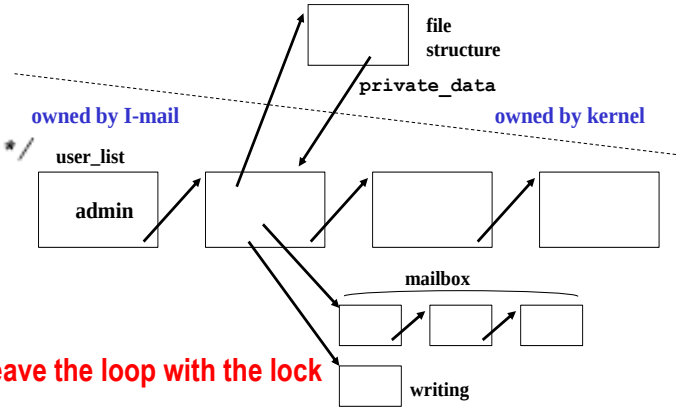
acquire the lock

if a message is found, leave the loop with the lock

release the lock and go to sleep

check asynchronous I/O flag before sleeping

conditions to wakeup



Comments on `Imail_read()` *(1)*

- Before starting, check the state of file f
  - no lock available in unauthenticated state; throw it out first
  - acquire lock before checking for deletion
- Questions
  - Q: Why not finish state check before acquiring lock?
  - A: prevent race in which user deletion occurs between check and use
  - Q: Why check in the loop?
  - A: locks must be released to sleep, thus user may be deleted between loop iterations (as desired—admin should not have to wait for user program to run!)

Comments on `Imail_read()` *(2)*

- Critical section from **`down`** to **`up`** in loop
  - if a message is found, leave the loop with the lock
  - otherwise, release lock and go to sleep
- Check asynchronous I/O flag before sleeping; if found, return error immediately

- Conditions passed to **`wait_event_interruptible`** dereference udata
  - Q: Why is this dereference safe?
  - A: User data is only actually freed in release, which can't be called while some process is in this function.

Comments on `Imail_read()` *(3)*

- ERESTARTSYS
  - What should happen if a signal is delivered while the program sleeps?
  - Different operating systems chose different answers (keep going, stop with a specific error, EINTR,).
  - Linux allows you to choose.
  - ERESTARTSYS options implemented in system call assembly linkage

Example I-mail: Imail_read()

```
    /*
     * 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) {                    <-- check whether there are bytes to read
        /* 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);                           <-- release the semaphore and return
    return n_read;
}
```

Comments on `Imail_read()` *(4)*

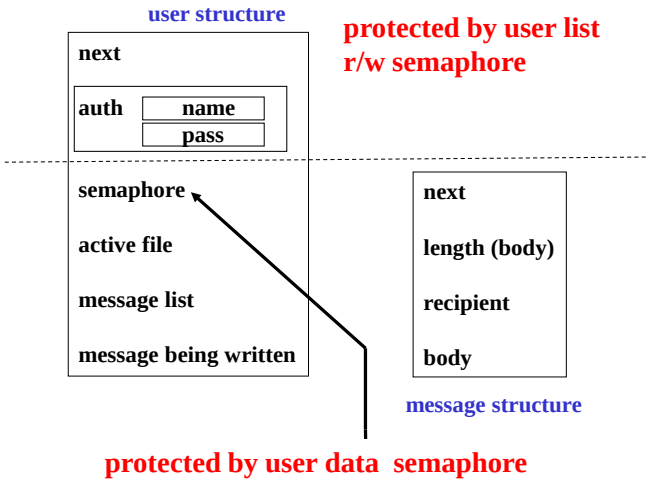
- At this point, we're ready to read some data
  - user is not deleted
  - user has a message
  - we hold the user data semaphore
- Initialization
  - the default number of bytes read is 0
  - means EOF for read semantics
- Check whether bytes remain in the message (offset vs. length)

Comments on `Imail_read()` *(5)*

- Make sure that we don't overflow the user's buffer
- Try to read (**`copy_to_user`**)
  - copying can cause page faults
    - ok to call while holding semaphores (as in I-mail)
    - not ok to call with spin locks
  - on failure, return EFAULT
  - on success
    - return number of bytes read
    - update the file offset
- In either case, release the semaphore and return

I-mail – Dynamic Allocation

- Questions that we consider
  - does allocation occur inside or outside of critical sections?
  - should we use GFP_KERNEL or GFP_ATOMIC?
  - no spin locks in I-mail, so GFP_KERNEL is adequate
- Messages
  - allocated by write message operation (ioctl)
  - deleted by
    - delete message
    - I-mail shutdown
    - delete user
    - failed delivery




```
udata->writing =
kmalloc (sizeof (Imail_msg_t), GFP_KERNEL);
```

```
kfree (udata->msg_list)
```

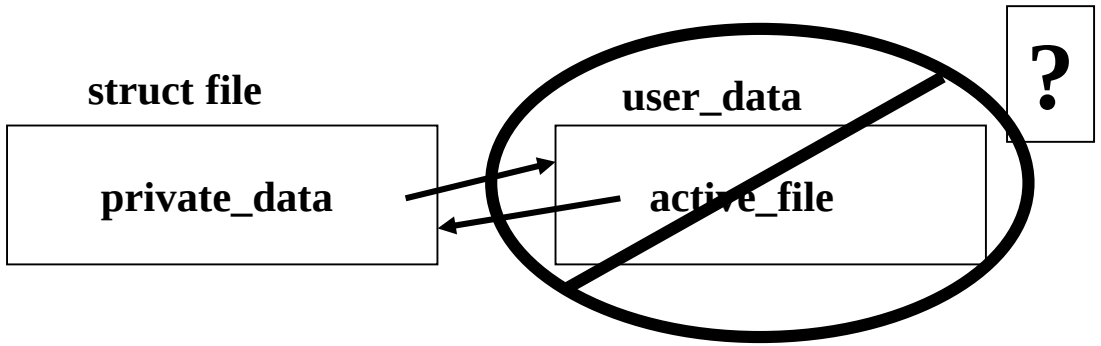
I-mail – Dynamic Allocation

- Users (other than the admin, which is unique static data in the driver)
  - allocated by add user
  - deleted by
    - I-mail shutdown
    - delete user

```
udata = kmalloc (sizeof (Imail_user_data_t), GFP_KERNEL);
kfree (udata)
```

I-mail – Dynamic Allocation

- Dilemma: What if a user is using I-mail when the user data is deleted?
- Can't just delete the user data
  - file structure has pointer
  - may be actively using (changing to NULL may not solve)



Solution to Dilemma

- Our solution
  - on deletion, remove from user list, but don't free the structure
  - give ownership (i.e., responsibility for deletion) to file structure
  - it can delete atomically on **`release`** call
  - signal by changing **`active_file`** to NULL

- Thus
  - if user is not authorized, simply delete
  - if user is authorized, set **`active_file`** to NULL

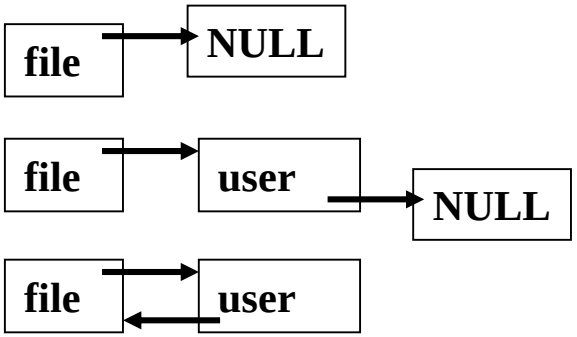
Few Questions

- Q: How do we know that user doesn't authorize after our check?
- A: auth needs read lock, but delete user op holds write lock on user list
- Q: What about after the deletion operation?
- A: user is not in the user list any more!

Few Questions

- Q: Why not be more aggressive? We could try to delete the user data any time that we found the **`active_file`** field to be NULL?
- A: Needs to be atomic; basically the same problem as before. Release is atomic.

- so…



unauthenticated (NOT_AUTH)

deleted (DELETED)

authenticated (GOOD)

Sub-operations in I-mail

- Find by name (need user list lock of either flavor)

Used by

  - set password
  - write message (check for recipient in user list)
  - add user
  - delete user
  - deliver mail

```
Imail_find_user ();    //(down_write (rwsem))
```

Sub-operations in I-mail

- Extract message being written (need user data semaphore)
  - release
  - fsync
  - write message

```
Imail_grab_pending ();    //(down (sem))
```

- Deliver mail (need user list lock, recipient's user data semaphore)
  - release
  - fsync
  - write message

Sub-function Synchronization

- For each sub-function
  - think about interactions with all callers/users
  - select synchronization details on that basis
  - document in function header to avoid misuse

Sub-function Synchronization

- Find by name (**`Imail_find_user`**)
  - if user list lock is acquired/released inside of function
  - user may be deleted between search and return
  - thus need to acquire before calling this subfunction

- Extract message being written (**`Imail_grab_pending`**)
  - callers must check deletion first
  - deletion might occur if lock is released and reacquired in function
  - thus need to acquire user data sem. before calling this subfunction

Sub-function Synchronization

- Deliver mail (**`Imail_deliver`**)
  - allows caller to optionally hold user list lock on entry
  - if not held (as specified by parameter), acquires read user list lock
  - acquires recipient's user data semaphore internally
  - releases user list lock if it was acquired within the function

Example I-mail:

`Imail_delete_user()`

```
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;
}
```

check permissions

acquire the lock

check whether such a user exists

check whether this is administrator

remove the user from the user list
(but don't free it!)

release the lock

Comments on `Imail_delete_user()` *(1)*

- Start by checking permissions
  - function checks both admin authentication and
  - sysadmin privileges (capability)
- Read the deletion identification before entering the critical section
- Mark the critical section (the rest of the function)

- Check whether such a user exists
- Check whether it's the admin
  - who cannot be deleted

Comments on `Imail_delete_user()` *(2)*

- Clean up a bit
  - throw away the mailbox and any message in progress
  - remove the user from the user list (but don't free it!)
    - note that the removal code does not allow for missing users
  - Q: How do we know that another program didn't remove first?
  - A: We have held a write lock on the user list since we located the user.

Comments on `Imail_delete_user()` *(3)*

- If the user is not online (authenticated)
  - delete right away
  - no file structure release call to which to delegate deletion anyway
  - again, write lock prevents race with authentication

- If the user is online
  - clear the active file pointer
  - wake up any sleeping user programs