**Mailbox Implementation**

The mailbox implementation is split into 4 main parts:

message.c – This handles memory allocation and structure initialization/destruction for individual Messages.

mailbox.c – This handles memory allocation and structure initialization/destruction for individual mailboxes. Also handles all per-mailbox synchronization and adding/removing (already initialized) messages to/from the mailbox.

mailbox\_manager.c – This handles process validation, storing mailboxes in a hash table, and synchronization of our hash table. Also handles delayed destruction for mailboxes (due to race condition in the override of sys\_exit, noted later). Also handles mailbox pointer “dereferencing” do handle the race condition noted in the hand out, although I think due to the deleted destruction of mailboxes, that particular condition is near impossible.

module.c – hooks in and overrides methods, calls methods in mailbox\_manager

We’ll start with message.c because it’s the one I listed first.

**message.c**

This is a relatively uninteresting file. We use a kmem\_cache (slab) to allocate and de-allocate messages. message\_create calls copy\_from\_user to copy a provided pointer into our newly allocated memory, and initializes. Messages are stored in a linked-list per mailbox, so we initialize a list\_head. Due to synchronization in mailboxes, it’s impossible for a Message struct to be used by more than one process at a time – this is because when a Message is created, it’s added to the linked list of the mailbox and never referenced again by that process. When the message is pulled from the linked list, this is done in a synchronized fashion, so once it’s out of the linked list, there is no way for any other process to get a reference to it.

This also does message validation – if it’s too long or negative length, it returns MSG\_LENGTH\_ERROR, and if it can’t copy\_from\_user it returns MSG\_ARG\_ERROR; if it can’t allocate memory for whatever reason, MAILBOX\_ERROR is a nice catch all. Although if it can’t allocate memory I think there are other problems.

**mailbox.c**

This is the heart of this project, synchronizing mailboxes. The mailbox structure itself is protected by a wait\_queue. Whenever you read or write to the structure, you grab this lock and only then do things. (The exception to this is mailbox->dereferences is protected by a separate wait\_queue – explained later).

The reason for using wait\_queue is simple: wait\_queue provides a mechanism to sleep the process until a condition is met, wait\_event, which is exactly what we need to do to handle blocking sends and receives. Wait\_queue also uses spin locks, which means that we can make the handler code “atomic” – while a spin is locked the code won’t be interrupted and no other code than handle that mailbox. Also, processes waiting for a mailbox will busy wait, and continue at the earliest possible moment – this is preferable to mutexes as we don’t have to wake up processes manually. One issue with this is of course the possibility of deadlocks, however this is avoided by only ever locking once in the function call, and always unlocking at the end. The mailbox has 2 spin locks (in each wait\_queue), and they are never held by the same caller at the same time, or ever requested to be.

Whenever a condition for the wait\_queue is changed, (mailbox->stopped, mailbox->waiting, mailbox->message\_count) we have to call wake\_up\_locked() to wake up all waiting processes. (wake\_up\_locked must be called with the spin lock held, which is why we use it)

For locking and unlocking we use spin\_lock\_irqsave and spin\_unlock\_irqrestore – this disables local (cpu) interrupts, preventing us from sleeping with a lock held, and saves/restores irq flags set by other processes for this processor.

For waiting on the wait queue we use wait\_event\_interruptible\_exclusive\_locked\_irq, which is a very long name for a function that releases the spin lock, enables interrupts and suspends. Every time it’s woken up by wake\_up\_locked, it checks the condition, and if it’s true, it grabs the spin lock (disabling interrupts), and continues along. (If it receives a signal such as SIGINT, it returns with an error, which must be handled properly, as the condition may/may not be true). The wait\_queue is a linked list and goes FIFO.

A blocking send or receive function waits on this queue until there is space or are messages in the mailbox. Stopping the mailbox waits on the same queue, but waits until there is no one left on the queue. This is done by a mailbox->waiting condition variable. It might be possible to just count the members in the wait\_queue as well, but mailbox->waiting seemed like it’d be faster and easier to implement. Reading a number vs traversing a linked list, I’ll let you make the conclusion.

A mailbox\_destroy stops the mailbox, calling mailbox\_exiting to also mark this mailbox as exiting for future processes trying to access it, then grabs the lock and waits for the wait\_queue to flush. Once the queue is flushed, it keeps the lock, destroys remaining messages, waits until the number of dereferences is 0, and then releases the memory. Usually by the time this is reached, the mailbox usually has no one waiting on it and no references, due to the implementation of remove\_mailbox\_for\_pid. (The mailbox is stopped, then mailbox\_destroy is scheduled for after the task exits)

Each mailbox has a member “dereferences” and a dereference\_queue – this is used mostly by mailbox manager. The reason for using a queue rather than a spin lock to protect dereferences is so that we can wait until it is zero – waiting until all references to the mailbox are gone.

**mailbox\_manger.c**

I guess since we were already talking about it, let’s talk about mailbox referencing.

Whenever we get a mailbox, we are inside a spinlock and uninterruptible. As soon as we get or create the mailbox, we unclaim\_mailbox it, which grabs the dereference\_queue lock, and increments dereferences, and releases the lock. As we do this all within another spin lock, it’s impossible to have a condition where our pointer becomes changed between getting it and claiming the mailbox. (We later have to unclaim\_mailbox when we’re done with it, which is handled mostly in module.c) If a mailbox is invalid (such as if it’s marked as stopping,) we immediately unclaim it and don’t allow access to it.

The main part of this file is the hashtable and protecting that. Originally we thought to use a rwlock to protect our hashtable, as we could get from the hashtable, and put into it. However, once it was implemented, only two main functions exited: get\_mailbox\_for\_pid, and remove\_mailbox\_for\_pid. Where’s the creation function? It’s right in get\_mailbox – making it a writing function. With no read-only functions, there’s no use for a rwlock, so a spinlock was used.

get\_mailbox\_for\_pid first checks if the process is valid, then if it is, grabs the spin lock, tries to get the mailbox from the hashtable, and if it doesn’t exist creates it. If it does exist but is marked as exiting, it returns an error. If there is an error when creating the mailbox, it returns an error. In all cases the spin is unlocked immediately before exiting – the whole function references the hash table so there’s no other way.

The reason get\_mailbox also creates the mailbox is due to the nature of the assignment – the mailbox is created on first use, therefor if the mailbox doesn’t exist, it must be the first use. Since the calling program can never know if it’s the first use, there’s always a possibility of creating a mailbox. A possible optimization would be to try getting without creating, then get with creating, allowing read-write style protection.

remove\_mailbox\_for\_pid checks if the mailbox exists in the hashtable. If it does, it stops it and marks it as exiting – this stops the mailbox, having blocked calls exit with MAILBOX\_STOPPED, and new calls exit with MAILBOX\_INVALID (since the process is stopping). After the mailbox is stopped, we schedule and dispatch a thread to wait until the task exits, and then deletes the mailbox.

mailbox\_deletion\_thread waits until the task is invalid (by the same is\_process\_valid that other callers would use), then removes the mailbox from the hash table, and calls mailbox\_destroy. Then it exits. The thread schedules itself to go every 50 ms, because 50 ms sounds like a good number.

**module.c**

This file handles injection, and overarching logic. Send message grabs the mailbox, creates the message, and sends the message. (It unclaims the mailbox pointer when it’s done with it). Receive message grabs the mailbox, removes the message from the mailbox, and tries to copy the message to the user. If it fails, it reads the message to the mailbox. If successful, it destroys the message. Manage grabs the mailbox, sets the count, and then if there is a call to stop, it stops the mailbox. It’s undefined behavior to pass a bad pointer to count, in this case passing a bad pointer to count while trying to stop the mailbox will result in not stopping the mailbox.

On exit, we check if the task group is dead, and if it is kill the mailbox.

On group exit, we just kill the mailbox. Both call remove\_mailbox\_for\_pid

**About sigint**

So SIGINT seems to not call sys\_exit, which is awesome because that means that our mailbox cleanup is never called. A simple way around this is to spawn a “garage collection” thread that can wake up every once and a while and check a linked list of pid\_t so see if mailboxes are still alive. When we create the mailbox add a copy of the pid\_t to the linked list. This could be wasteful though if there are a lot of active mailboxes.

You could also install a signal handler to all of the processes you make a mailbox for that destroys the mailbox before handing it off to the default handler. There’s a member of task\_struct for that.