Project 4 (150 points)
Assigned: February 14, 2014
Due: March 4, 2014, 11:59 PM

CS-3013, Operating Systems
C-Term 2014

Project 4: Kernel Message System

Introduction & Problem statement

In this project, you will implement a mailbox system within the Linux kernel that allows processes in separate address spaces to send to and receive messages from each other.

The purpose of this project is threefold:–

1. To gain additional experience with programming and testing synchronization among concurrent computations.
2. To gain additional experience working inside an operating system kernel on a non-trivial problem.
3. To develop and/or adapt an intensive, highly concurrent test program to exercise your kernel message system.

Traditional Unix and Linux *processes* occupy separate address spaces and have no way to communicate with each other except via pipes, files, or segments of shared virtual memory. Although the invention of *threads* has mitigated this problem substantially, there are still circumstances in which processes need to retain the benefits of separate address spaces while enjoying a more friendly form of inter-process communication.

Your mailbox system will allow processes to send *messages* to each other. These messages will be copied by the kernel into mailboxes of the receiving processes. To keep things simple, messages will be of fixed size and will be processed in FIFO (*first-in, first-out*) order. Every ordinary process (i.e., *thread group* in Linux kernel terminology) can have a mailbox. Any thread or process will be able to send messages to any mailbox (including its own), but only the threads of the mailbox "owner" will be able to receive them.

In previous years, this assignment required students to modify the **`task_struct`** to include a pointer to the mailbox of a process, to modify **`fork.c`** and **`exit.c`** to create and destroy mailboxes, to add a module to the kernel to implement the mailbox functions, and to build a new kernel. This year, students will use the kernel built during Project 0 and implement the mailbox functions in a Loadable Kernel Module (LKM). It is likely to be necessary to intercept the **`exit`** system call and possibly others.

A mailbox is a kernel-space object implement by your LKM. It should be dynamically created on the first attempt by any process to use it, and should be destroyed when the owning process exits. It is suggested that you organize a mailbox like a *monitor*. See the lecture notes about monitors, including the example, in pptx or pdf.

To test your mailbox system, you will need to write a test system *in user space* that sends messages randomly, receives replies, and keeps track of what was sent and received. This test system is intended to accomplish goals similar to the bathroom simulation of Project 3, even though the problem is dramatically different. For your convenience, several test programs by students of previous terms are provided (see below).

Note: You are strongly encouraged to work in two- or three-person teams. Teams should register with the professor by e-mail in order for them to be recognized by the *Turnin* system. This is *Project 4*. Previous teams must re-register for this project.

Mailboxes and Messages

The application-level interface (i.e., the API) to the mailbox facility is defined in the *C* header file at the mailbox.h and specified in this section. A *mailbox* is an abstract object capable of containing a bounded number of *messages* of undefined structure. Threads within the same process share the same mailbox.

In order to keep things simple, message bodies in this project are of fixed maximum size but of no defined type. The mailbox header file defines this maximum size in bytes as

```
#define MAX_MSG_SIZE 128
```

Your implementation should specify a maximum number of messages that any mailbox may hold. This should be at least 32, but any larger number would be acceptable. The maximum number of messages is not part of the message system API. Your test program should be able to discover the maximum size dynamically.

Any process or thread (i.e., any Linux task) may send a message to any process (including itself) if it knows the process ID of that process — i.e., its **`pid_t`**. It may learn this process ID when it is forked or spawned, or it may learn it from some other source of information such as another message. To send a message, the task invokes the function

```
int SendMsg(pid_t dest, void *msg, int len, bool block)
```

The argument **`dest`** is the process ID of the recipient. The argument ***`msg`** points to an uninterpreted array of bytes of length **`len`**, not exceeding **`MAX_MSG_SIZE`**. The application program must cast this pointer to its own data structure. The argument **`block`** specifies whether the sending task *blocks* if it cannot send the message. If **`block`** is **`TRUE`** and the mailbox is full, then the calling task will wait and only return when the mailbox is no longer full and the message has been sent. If **`block`** is **`FALSE`** and the mailbox is full, then the sending task will return immediately with an error indication, and the message will *not* be sent. If another task stops the mailbox (see below) while the calling task is waiting (because the mailbox is full and **`block`** is **`TRUE`**), **`SendMsg`** returns an error indicating the mailbox is stopped. Zero-length messages are allowed, but negative-length messages are errors.

In all cases, **`SendMsg`** returns zero if it successfully queues the message onto the destination mailbox, and it returns an error code if not. See below for error codes.

To receive a message and remove it from its mailbox, a task invokes

```
int RcvMsg(pid_t *sender, void *msg, int *len, bool block)
```

The process ID of the sender is copied into the **`pid_t`** pointed to by **`sender`**. The area pointed to by **`msg`** must be an array of bytes of **`MAX_MSG_SIZE`** in length. The body of the queued message is copied this area, and its length is copied into the integer pointed to by **`len`**. The actual behavior of **`RcvMsg`** depends upon the value of **`block`**, whether or not the mailbox is empty, and whether or not the mailbox is stopped. Specifically:–

- If a mailbox is not empty, **`RcvMsg`** retrieves the *oldest* message in the mailbox and returns it, regardless of the value of **`block`** and regardless of whether the mailbox is stopped or not.
- If the mailbox is empty and not stopped, and if **`block`** is **`TRUE`**, **`RcvMsg`** waits until a new message has been received, and then it retrieves that new message.
- If the mailbox is empty and not stopped, and if the argument **`block`** is **`FALSE`**, then **`RcvMsg`** returns immediately with an error.
- If the mailbox is stopped, **`RcvMsg`** will retrieve an existing queued message from the mailbox, but it will return an error indicating that the mailbox is stopped if the mailbox is empty.
- If another task stops the mailbox while **`RcvMsg`** is waiting (because the mailbox is empty and block is **`TRUE`**), **`RcvMsg`** immediately returns an error indicating that the mailbox has been stopped.

In all cases, **`RcvMsg`** returns zero if it successfully retrieves and returns a message from the mailbox and an error code if not.

Note that any thread of a process, or even multiple threads, can attempt to receive messages from the mailbox of that process. Messages are received in FIFO order.

A task may manage its mailbox by calling

```
int ManageMailbox(bool stop, int *count)
```

The argument **`*count`** is a pointer to an integer; **`ManageMailbox`** will copy into this integer the number of queued messages in the mailbox. The Boolean argument **`stop`** specifies whether to stop this mailbox from receiving any more messages. If **`stop`** is true, then any attempt to send a future message to this mailbox results in an error to the sending task. Also, all blocked calls to **`SendMsg`** or **`RcvMsg`** for this mailbox *immediately* return with the error code indicating that the mailbox has been stopped. A task may continue to retrieve accumulated messages from its mailbox after stopping it. A mailbox may be stopped multiple times, with the same effect as stopping it once. To keep this assignment simple, there is no way to restart a mailbox that has been stopped.

A mailbox is automatically started the first time that *any* process tries to send a message to it or whenever its owning process tries to receive a message from it or to manage it. A mailbox is automatically stopped when its process is terminated. When a process terminates, all messages must be flushed and discarded, and any blocked tasks must immediately return with an error. Note that this only happens when the last remaining thread of the process owning the mailbox terminates.

Error codes

All of the message and mailbox functions return **`int`** values indicating success or error. A return value of zero represents success. The following errors have been defined: –

- **`MAILBOX_FULL`**: This is returned when **`SendMsg`** with **`block`** set to **`FALSE`** attempts to send a message to a mailbox that already contains the maximum number of messages that it can hold.

- **`MAILBOX_EMPTY`**: This is returned when **`RcvMsg`** with **`block`** set to **`FALSE`** attempts to retrieve a message from an empty mailbox.
- **`MAILBOX_STOPPED`**: This is returned when any call to **`SendMsg`** is made to a mailbox that has been stopped or when any call is made to **`RcvMsg`** to a mailbox that is both stopped and empty. It is also returned by a call to **`SendMsg`** or **`RcvMsg`** with **`block`** set to **`TRUE`** if another task stops the mailbox while the calling task is waiting.
- **`MAILBOX_INVALID`**: This is returned when **`SendMsg`** specifies a destination that is not a valid process ID. Note that kernel tasks and certain other system processes are also considered invalid. This error is also returned if a process (and its mailbox) have been or are being deleted.
- **`MSG_LENGTH_ERROR`**: This is returned when a call to **`SendMsg`** attempts to send a message longer than **`MAX_MSG_SIZE`** or of negative length.
- **`MSG_ARG_ERROR`**: This is returned when any pointer argument to any message or mailbox call is invalid, so that **`copy_to_user()`** and **`copy_from_user()`** fail.
- **`MAILBOX_ERROR`**: This is returned in the case of any other mailbox or message error.

Note: The semantics of the **`MAILBOX_STOPPED`** error code are intended to have the effect that **`SendMsg`** to a full mailbox with **`BLOCK`** = **`TRUE`** gets the same result whether the mailbox was stopped before or after invoking **`SendMsg`**. Likewise, **`RcvMsg`** with **`BLOCK`** = **`TRUE`** to an empty mailbox gets the same result whether the mailbox was stopped before or after invoking **`RcvMsg`**.

A stopped mailbox of a running process should always return **`MAILBOX_STOPPED`**, not **`MAILBOX_INVALID`**. **`MAILBOX_INVALID`** should be reserved for tasks that no longer exist or that have **`NULL`** mailbox pointers.

Implementation

There are several parts to the implementation of this message and mailbox facility: –

- Initialization of the mailbox system itself as a Loadable Kernel Module (LKM).
- Setting up a new mailbox when it is needed and deleting it during termination. You should only delete a mailbox when all tasks of a task group (i.e., process) have terminated. You must also deal with potential race conditions that can occur during deletion.
- Intercepting system calls to implement the functions of the mailbox facility. You should use **`cs3013_syscall1`**, **`cs3013_syscall2`**, and **`cs3013_syscall3`** that were added to your Linux kernel during Project 0.
- Allocating and managing space for messages and mailboxes in the kernel.
- Implementing the synchronization needed for the sending and receiving messages.
- Creating a user-space module that provides the interface to the message and mailbox facility.
- Aggressively testing the mailbox facility.

Suggested Approach

Instead of storing pointers to mailboxes in **`tast_struct`**s, as was done in previous years, it is suggested that you create a separate hash table[1] indexed by process IDs (i.e., **`pid_t`**) that contains pointers to mailboxes of the corresponding processes. Whenever any process attempts to send to, receive from, or manage a mailbox, your kernel functions must index into the hash table by the process ID (i.e., its **`pid_t`**) to find it. When a process terminates, the mailbox must be deleted. Future attempts to send messages to that mailbox must return **`MAILBOX_INVALID`** errors.

Creating and keeping track of mailboxes

The easiest approach is to create a mailbox on demand — that it, on the first attempt to use it by any of **`SendMsg`**, **`RcvMsg`**, or **`ManageMailbox`**. The process ID must be validated and the mailbox must be created if it does not exist (and the process is not a system process). A pointer to the mailbox must be entered into the hash table and must remain there so long as the process is alive (and also, so long as the mailbox LKM is loaded).

You should not create a mailbox for a kernel thread, but you should create an entry for it in your hash table with a null mailbox pointer. You can recognize a kernel thread because it has no virtual memory — i.e., its **`task_struct->mm`** pointer is null. An attempt to send a message to a task with a null **`mailbox`** pointer should return a **`MAILBOX_INVALID`** error.

Note: A useful debugging technique used by students of previous terms is to add extra functionality to one of the system calls. This helps, for example, to print out the message queue of the mailbox and to examine the values of the mailbox data structures from the pleasure of a user space program. This is easier than using **`printk()`** in many cases. However, do *not* make your test programs depend upon such extra functionality, because they won't work on other mailbox implementations.

Deleting processes and mailboxes

When a process terminates, you need to intercept the appropriate system call in order be able to stop, flush, and remove the mailbox before the process actually exits. This requires you to study and understand the kernel file **`exit.c`**, especially the function **`do_exit()`**, which does all of the work of terminated threads and processes. This function is called by at least two system calls — **`sys_exit`** and **`sys_exit_group`**. You need to get your hands on calls to **`do_exit()`** so that you can shut down a mailbox associated with the process, if necessary. At what point during the exiting code you do this is up to you.

The kernel function **`do_exit()`** is called for both processes and threads. One of the things that this function does is to check to see whether the entire *thread group* is dead; you can see this code in **`do_exit()`**. If the group is dead, you must delete the mailbox and reclaim the memory for the mailbox data structure. However, if the group is not dead, you must not delete the mailbox.

When you are deleting a mailbox, you need to implement the specified functionality for calls that are blocked, so that they can be unblocked and return errors. The easiest way to do this is to stop the mailbox and let blocked calls return with the **`MAILBOX_STOPPED`** error. Only

[1] Hash tables are widely used for $O(1)$ access to tabular data. An example implementation can be found in §6.6 of *The C Programming Language*, 2nd edition, by Kernighan and Ritchie.

when *all blocked calls have returned* is it safe to flush and delete the mailbox and pass control **`to do_exit()`**.

AFter a process has exited, its **`pid_t`** will no longer be valid. Therefore **`SendMsg()`** should return the **`MAILBOX_INVALID`** error to any task that attempt to send to it.

Note: The function **`do_exit()`** does *not* return. Instead, to calls the scheduler to dispatch another task. The current task is never dispatched again. See the description of **`do_exit()`** in Chapter 3 of *Linux Kernel Development*, 3rd edition, by Robert Love. Therefore, when you intercept the exit system call and then invoke **`do_exit()`** after deleting the mailbox, you must not expect it to return.

Memory allocation

Messages come and go frequently, so using **`kmalloc()`** to allocate memory for each new message would lead to a lot of fragmentation of the kernel heap. Instead, memory for messages should come from the *slab allocator* in **`linux/slab.h`**. These will be in a *cache* (i.e., a pool) of identically sized chunks of kernel memory implemented by the kernel as a set of *slabs*.

Such a cache can be created at the time of loading the kernel module by

```
kmem_cache_t* kmem_cache_create(const char* name, size_t size,
    size_t align, unsigned long flags, NULL)
```

This function returns a pointer to a cache object. The first argument **`name`** points to a string containing the text name of the object, and the second argument is the **`size`** of each element in the cache (i.e., **`MAX_MSG_SIZE`** plus whatever overhead is needed for links and for storing the sender ID, the actual length of a message, and any other information you need). The next two arguments, **`align`** and **`flags`**, should normally be zero. The last argument is for a constructor function to create slabs within the cache; setting this to **`NULL`** causes the kernel to use the default constructor and destructor. See page 250 of *Linux Kernel Development,* 3rd ed., by Robert Love.

After a cache has been created, memory is allocated from the cache by

```
void* kmem_cache_alloc(kmem_cache_t* cachep, int flags)
```

This returns a pointer to a contiguous region of kernel memory of **`size`** bytes, where **`size`** was specified when the cache was created. For flags, the value **`GFP_KERNEL`** is probably what you want. To free the object, use

```
void kmem_cache_free(kmem_cache_t* cachep, void* objp)
```

This returns the object pointed to by **`objp`** to the cache for subsequent reuse.

Note: It is intended in this assignment that all messages of all mailboxes be allocated from the same **`kmem_cache_t`** pool. Do not make the mistake of some students in previous terms of creating a separate pool for each mailbox.

When the LKM is unloaded, you need to destroy the cache. This can be done by calling

```
int kmem_cache_destroy(kmem_cache_t* cachep)
```

The caller must ensure that all slabs in the cache are empty and that no one accesses the cache during and after its destruction.

Note that before destroying the cache, you must stop and flush all mailboxes and allow blocked processes to leave the mailbox system calls. *This is probably the trickiest part of the assignment.*

Memory allocation for mailbox data structures

In previous terms, it was suggested that students use **`kmalloc()`** and **`kfree()`** to allocate and free memory for their kernel mailbox data structures. That was an expedient. It would be better to create a separate slab for mailbox data structures. This way, they can be easily and quickly allocated and freed without creating a lot of fragmentation in the kernel's heap.

You should also decide whether it is better to use a cache for allocating hash table entries or not. Be sure to describe your decision and reasoning in your project write-up.

Loading and Unloading your LKM

Prior to loading your LKM, user space test programs will not work. Their system calls will invoke the functionality of the **`cs3013-syscalln`** calls that you installed in Project 0.

When your LKM is loaded and initialized, you must set up and initialize the kernel messaging system from scratch. This includes intercepting the appropriate system calls, creating the kernel memory caches for allocating messages and possibly mailboxes, initializing the hash table, etc.

When your LKM is unloaded, you must clean up and restore the world to the way it was before you started. This means stopping all mailboxes and making sure that waiting processes have exited, restoring kernel system calls to their state before your LKM was loaded, flushing all mailboxes, deleting all outstanding messages, deleting the hash table, destroying the caches, restoring system calls to their original state, and generally getting the world back in order.

Synchronization and Design of the Mailbox

The mailbox facility of this assignment falls mostly under *task parallelism* as discussed in class. In previous terms, it has been suggested that each individual mailbox be structured as a producer-consumer, because there was no synchronization operation in the Linux kernel equivalent to an atomic *release-monitor-lock-and-wait-for-condition*. In recent versions of the Linux kernel, this feature has been added. This was researched and described by a student from a previous term and can be found at

http://web.cs.wpi.edu/~cs3013/c14/Resources/LockingWaitQueues.docx and .pdf

For reference about synchronization mechanisms in the kernel, please consult Chapters 9 and 10 of *Linux Kernel Development,* 3rd ed. See also pages 58-59 for more information about *wait queues* in the Linux kernel

A useful synchronization primitive for this project is the *spinlock*. Spinlocks are defined in **`linux/spinlock.h`**. A spinlock can be used as a mutex (mutual exclusion control) or "monitor lock" for each individual mailbox. Note that each mailbox in the kernel should contain one or more spinlocks — initialized to **`SPIN_LOCK_UNLOCKED`** — to manage that particular mailbox, independent of all other mailboxes.

The kernel can hold a particular spinlock on behalf of only one task at a time, so that at most one thread can be "in" a particular mailbox data structure at any given instant. A spinlock protects against the issues of multiprocessors, even if two tasks are running on separate processors or cores. Recall that spinlocks are most appropriate for code that executes quickly and is guaranteed not to become blocked while holding a spinlock.

Spinlocks *must not* be used in the Linux kernel for code that might sleep, take page faults, execute **`copy_to_user()`** or **`copy_from_user()`**, or otherwise wait for a long time. If a task needs to block itself — for example, to await an event such as the sending of a message by some other task — then other some mechanism should be used. For this assignment, the best and simplest approach is the *locking wait_queue*, described in the URL above.

Complications

If the previous paragraphs sound too easy, that's because they are. There are multiple complications that get in the way of the simple monitor model.

Stopping a mailbox

First, the interface allows a mailbox to be stopped and specifies what happens if there are tasks waiting to send or receive. The task stopping the mailbox must atomically set a flag somewhere indicating that mailbox is stopped.[2] It then must wake up any tasks that are waiting to send.

In order to wake up waiting tasks, there must be a counter somewhere indicating how many tasks might be waiting or about to wait. Each sending task must atomically increment this counter before attempting to wait on a locking wait queue or a semaphore. When a sending task wakes up, it must determine whether it awakened because (a) a message slot opened up and it can continue sending, (b) the mailbox has been stopped by another task, in which case it must return a **`MAILBOX_STOPPED`** error, or (c) it (i.e., the sending task) has been interrupted and must abort the send operation completely. Note that the sending task must atomically decrement the counter when it wakes up, no matter which case applies.

Meanwhile, the stopping task must wake up all waiting tasks by calling the **`wake_up_all()`** function.

Since sending and receiving are symmetric, the same has to happen for receiving tasks. That is, a counter has to be incremented atomically before a task tries to wait, and it has to be decremented after the receiving task has awakened and completed whatever actions are necessary. The stopping task has to wake up all tasks by calling the **`wake_up_all()`** function.

To complicate matters even further, the counters and tests must be structured so that tasks can continue to receive messages from the non-empty queue of a stopped mailbox.

Flushing a mailbox

Second, when a mailbox is being deleted as a result of its process being terminated, it needs to be flushed of any remaining messages. To be safe, the task doing the flushing has to wait until all other tasks are completely out of the mailbox. This is indicated when the counters

[2] For atomic increment and decrement operations, see Table 10.1 on page 178 of Robert Love. Alternatively, you can protect your counters and "stopped" flag within another spinlock per mailbox.

used for stopping the mailbox are both zero. At this point, the task doing the flushing can simply delete any remaining messages.

Waiting for the counters to go to zero is a little like waiting on a condition variable in a monitor. In a loop, the counters need to be tested and if either is non-zero in a loop and, if so, the flushing task must wait. For this purpose, another wait queue may be appropriate.

Race conditions

Third, a much more subtle race condition occurs when mailboxes are deleted as part of the process exit code in **`do_exit()`**. That is, when all of the threads of a process have terminated and the process itself is about to go away, its mailbox **`struct`** must be properly cleaned up and deleted. In particular, the mailbox must be stopped so that no other processes can continue to send messages to it, the queued messages must be flushed, and any waiting threads must have the opportunity to wake up and exit their **`SendMsg`** or **`RcvMsg`** calls.

Suppose that a process is exiting and that its mailbox is being deleted. Suppose that at the same time, a thread of another process is trying to send a message to that that mailbox. The sending thread calls **`SendMsg`**, which first obtains the pointer to the mailbox from the hash table. Suppose now that the sending process takes a page fault (which is possible, even while it is in the kernel waiting during a **`SendMsg`** call). Suppose also that the destination process exits, and its mailbox is deleted. Finally, the sending thread gets back to the head of the ready queue, is dispatched, and attempts to complete the **`SendMsg`** call using the pointer to the mailbox that it had previously obtained. But that mailbox has since disappeared, and so the pointer is no longer valid. Big trouble!

You must address this race condition. *It is strongly suggested that you get the rest of the mailbox code working before tackling this problem.*

A suggested solution is that you maintain a reference count in the mailbox data structure that indicates how many tasks hold copies of the pointer to that mailbox data structure. The reference count itself can be an *atomic integer* such as described on p. 176 of *Linux Kernel Development*. However, it is also necessary to prevent any other task from deleting the mailbox, even while updating the reference count.

In previous terms, it was suggest that students use the reader-write spinlock **`rwlock_t tasklist_lock`** — defined near line 256 of **`include/linux/sched.h`** — might be necessary. This protects the entire set of all **`task_struct`**s. You can learn more about reader-write locks in §2.5.2 of Tanenbaum and pages 188-190 of the 3rd edition of *Linux Kernel Development*.[3] While it may not be necessary to use **`tasklist_lock`** in this implementation, using a reader-writer lock to protect the hash table may still be useful.

User-space support

No application program will call the kernel directly. Instead, you must implement a user-space support module that can be bound with an application program to invoke the message facility. This module must implement the interface mailbox.h as specified earlier in this doc-

[3] The basic point of using a reader-writer lock to protect the **`tasklist`** is that many tasks on many processors can be manipulating various **`task_structs`**. It would be a bad idea to restrict the level of concurrency among such tasks, except in the narrow instance of a task being deleted.

ument. Your implementation should be compiled into a file **`mailbox.o`** that can be linked with any test program, either yours, the graders', or another student's.

Note: It is a requirement of this assignment that you implement the mailbox.h interface exactly. You may not alter it or add to it. The reason is that the graders will compile their own test cases and link with your **`mailbox.o`** file in order to test your kernel.

Make sure that it works. They may also replace other students' **`mailbox.o`** files with yours and run your tests on other kernels. Kernel logs are for your debugging only, not for demonstrating correct operation.

Testing your Message System

Testing any program involving inter-process synchronization is difficult. Essentially, you need a multi-process, multi-threaded test program that randomly beats up on the message system, sending and receiving lots and lots of messages at random intervals. Your test program should fork a number of processes, passing its own process ID to each one. Each child should fork a number of additional processes, passing on as many process IDs as it knows. The number of processes and the depth of the tree of processes should be specified on the command line.

Each process (parents and children) should the spawn a random number of threads, each of which randomly sleeps, sends, and reads messages. The average number of threads should be specified on the command line. Every thread should forward a random subset of the process IDs that it knows about in the messages that it sends. When sending a message, a process or thread should make a record of that message so that it can determine whether it has received a reply. When any thread receives a message, it should sleep a small, random amount of time, and then reply back to the sender to acknowledge the message.

Threads and processes should stop sending messages after a random amount of time, and then later they should print information on **`stdout`** indicating a summary before exiting. The general idea is to get lots of messages going back and forth to stress the message system.

Note: This test program should be as highly multi-threaded as the Bathroom simulation of Project 3 (.docx, pdf). The difference is that this test program must have many more processes, each of which has multiple threads.

Note: Every test needs to print a summary statement saying whether the test PASSED or FAILED. "PASSED" means that it got the expected results. Otherwise it FAILED.[4]

In order to get you started, a suite of sample test programs is provided at :–

http://web.cs.wpi.edu/~cs3013/c14/Resources/Project4_SampleTests.zip

This file includes seven test programs that were prepared by a student of a previous term. You may use them directly or adapt them to test particular aspects of your mailbox implementation. *No warranty is expressed or implied regarding the completeness or correctness of these tests.* In particular, only one of these tests is multi-threaded. You are still responsible for testing your own mailbox implementation adequately. These are only provided to help.

[4] Please do not make the mistake of one of the authors of the earlier test programs. When that program was testing for correct response to error conditions, it printed **`FAILED`** when it meant that the program was detecting the error correctly!

A second, more comprehensive set of tests is provided at:–

http://web.cs.wpi.edu/~cs3013/c14/Resources/Project4AdditionalTests.zip

These are more aggressive tests, but not all work correctly. In particular, some of them return the wrong error codes. In others, the authors assumed secret special arguments to the message system to probe its internal state.

You are responsible for the correct operation of your kernel message system. Passing these tests is not a guarantee of correctness, only an aid to you.

Your tests

You may modify and/or add to these tests any way you like. You must submit the modifications with your project.

Your tests should comprise one or more programs built by a **`makefile`** (or possibly more than one **`makefile`**). Your makefile should take the **`mailbox.o`** file as a parameter, so that you can use your test program with someone else's LKM and so that others can use their **`mailbox.o`** files with your test suite. The instructor retains the most useful test files from year to year to use in future instances of CS-3013.

Debugging Resources

The following documents are from students of a previous term. They may help in debugging the Linux kernel.

http://web.cs.wpi.edu/~cs3013/c14/Resources/How-to_read-an-Oops.doc, .pdf

http://web.cs.wpi.edu/~cs3013/c14/Resources/GDB_Cookbook.docx, .pdf

Pacing Yourselves

This project is due in 2½ weeks, but it cannot be accomplished in less than that after any procrastination. Therefore, you should adopt a work plan such as the following:–

- By February 21, you should have the initialization and mailbox creation working and tested by your test program, along with the simple sending and receiving of messages. You should get this part working before addressing the synchronization issues and race conditions identified above. At the very minimum, the programs *testmailbox1* and *testmailbox2* from the suite of sample tests should pass.
- By February 28, you should be able synchronize access to a mailbox by multiple tasks, and you should be able to stop and flush a mailbox and handle blocked calls. These would be indicated by the programs *testmailbox3* and *testmailbox4* from the suite of sample tests, plus any extra conditions that you identify of your own.
- By March 4, all parts of the project should be working to the best of your ability, including the flushing and deletion of mailboxes and associated race conditions. Your implementation should be able to pass the sample tests *testprogram5, testprogram6*, and *testprogram7*, along with any other tests that show that it works under an aggressive load

This project is intended to be completed by the end of term. Due to the short deadline for submitting grades, there is no scope for extensions.

Write-up

In your write-up, you must describe in clear and cogent detail the following:–

- the synchronization of your message sending and receiving facility, including the *programming invariants* of your solution;
- how you deal with the race condition and waiting tasks at the time of destroying your mailbox, including the programming invariant pertaining to task exits; and
- how you test your mailbox facility — in particular, comment on the output of each test of the mailbox implementation as listed in the Grading Rubric below.
- summary output of your tests.

Submitting this Assignment

Your final submission should include a zip file containing two folders and a write-up. The zip file should be named

`<name>-kernelMailbox.zip`

where **`<name>`** is either your WPI userID (i.e., e-mail ID) or your team name.

One folder should be named **`Message_LKM`** and should contain

- the **`.c`** and **`.h`** file(s) for your Loadable Kernel Module (LKM)
- The **`makefile`** to build your LKM.
- Any other files and headers necessary to build your LKM.

The second folder should be named **`Test`** and should contain

- The source code for the user-space interface module and all test programs that you used to test your system, including any adapted from the suite of sample tests.
- You should also include a copy of mailbox.h. so that the **`makefile`** (next bullet) works properly in the graders' environment.
- A **`makefile`** to compile the user-space interface module and all test programs.
- Output of your test runs demonstrating the various aspects of your implementation.

The write-up should be in either MS Word (**`.doc`** or **`.docx`**) format or **`PDF`** format.

Submit your zip file via *Turnin*. This assignment is *Project 4*. *Be sure to put your name at the top of every file you submit and/or at the top of every file that you edit!*

Grading

This is the largest programming project of the term, representing nearly half of the total project component of the grade for this course. Grading will be allocated approximately as follows: –

Loadable Kernel Module (80 points)

- Correctly structured LKM that compiles without warnings on the graders' kernel — 10 points
- Hash table and functions for validating processes, adding and finding mailboxes, and recognizing messages to defunct processes — 10 points
- Correct use of slab allocators for messages and mailboxes — 10 points

  (Additional three points for using the slab-allocator for hash table entries.)
- Correct sending and receiving of messages in simple situations (i.e., not multi-threaded or multiple processes), including correct use of error codes — 10 points
- Correctly synchronizing the sending and receiving messages to and from one mailbox by multiple processes and/or threads, including test output showing the interleaved sending and receiving — 10 points
- Stopping mailboxes and correctly handling blocked calls when a mailbox is stopped, including test output showing the flushing of stopped mailboxes — 10 points
- Correctly handling defunct mailboxes (i.e., mailboxes of processes that have exited) and messages to those mailboxes — 10 points
- Correctly deleting mailboxes (including non-empty mailboxes) and solving the race condition pertaining to deletion, including test output showing correct even with blocked calls to the mailbox — 10 points.

Test Program (50 points)

- Correctly compiled test suite structured as one or more programs that can be compiled and linked with any other **`mailbox.o`** — 10 points
- Tests showing that your mailbox system works correctly in simple (i.e., non-multi-threaded or non-multi-process) situations — 10 points
- Tests of creating and deleting mailboxes, including race conditions upon deletion — 10 points
- Aggressive multi-process and/or multi-threaded test program to show that mailbox system works correctly under stress — 20 points

Write-up (20 points)

- Description of algorithms, code, and programming invariants for correct operation of the mailbox — 10 points
- Description of test results and output showing that your kernel message system works correctly — 10 points