Phase 2: Messages and Interrupt Handlers

Complete program due: Monday, October 5th, 9:00 pm

1.0 Overview

For this second phase of the operating system, you will implement low-level process synchronization and communication via messages, and interrupt handlers. This phase, combined with phase 1, provides the building blocks needed by later phases, which will implement more complicated process-control functions, device drivers, and virtual memory.

2.0 Kernel Protection

The functions provided by this level of the kernel may only be called by processes running in kernel mode. The kernel should confirm this and in case of execution by a user mode process, should print an error message and invoke **`halt(1)`**. Functions in this phase can disable interrupts as needed to protect shared data structures (same situation as phase 1.)

3.0 Messages

Phase 2 implements routines for passing messages between processes. The six routines are **`MboxCreate`**, **`MboxSend`**, **`MboxReceive`**, **`MboxCondSend`**, **`MboxCondReceive`**, **`MboxRelease`**. Each mailbox will have a unique id. Processes can create and release mailboxes. The mailbox is not "owned" by a particular process; that is, a process can create a mailbox and a different process can release it, possibly after the creating process has quit.

The mailboxes have slots for holding messages. The number of slots for a particular mailbox is specified when the mailbox is created, along with the maximum size for the slots of that mailbox. When no messages are available, **`MboxReceive`** will block the calling process until a message arrives. When the slots of a mailbox are full, **`MboxSend`** will block until a slot becomes available.

`MboxCondSend` and **`MboxCondReceive`** modify this behavior by not blocking, but returning a value to indicate success or failure. Zero-slot mailboxes are a special case. Such mailboxes are intended for synchronization between sender and receiver. The sender will be blocked until a receiver collects the message; or, the receiver will be blocked until the sender sends the message. The conditional operations will not block on zero-slot mailboxes, but may succeed if a (non-conditional) operation has previously blocked a sender or receiver.

You need to insure that messages are delivered in the order sent, and in the order receive's are done. In general, the slot mechanism makes this work. However, priorities can adversely affect this performance. For example, consider a low priority process that blocks on a receive, followed by a high priority process blocking on a receive on the same mailbox. When a message is sent to this mailbox, the message must be delivered to the low priority process since it is at the head of the queue, not to the high priority process. When two messages are sent to this same mailbox, the first message must be delivered to the low priority process and the second message delivered to the high priority process.

There are three constants defined in the provided *phase2.h*:

- **`MAXMBOX`** specifies the maximum number of mailboxes.
- **`MAXSLOTS`** specifies the maximum number of mailbox slots that can be in use at any one time. Note that this is the number of active messages being held in mailboxes. Thus, you can create mailboxes whose slots total more than **`MAXSLOTS`** so long as there are not more than **`MAXSLOTS`** actual messages in the system at one time.
- **`MAX_MESSAGE`** specifies the largest possible size of one message in bytes.

`int MboxCreate(int num_slots, int slot_size);`

Creates a mailbox that can be used for interprocess communication and synchronization. The constant **`MAXMBOX`**, found in *phase2.h*, defines the maximum number of mailboxes. A unique mailbox ID is returned that is used in subsequent sends and receives.

Return Values:

-1:	**`slot_size`** or **`num_slots`** is incorrect; or, no mailboxes available.
>= 0:	ID of the mailbox.

`int MboxRelease(int mailboxID);`

Releases a previously created mailbox. Any process waiting on the mailbox should be **`zap`**'d. Note, however, that **`zap`**'ing does not quite work. It would work for a high priority process releasing low priority processes from the mailbox, but not the other way around. You will need to devise a different means of handling processes that are blocked on a mailbox being released. Essentially, you will need to have a blocked process return -3 from the send or receive that caused it to block. You will need to have the process that called **`MboxRelease`** unblock all the blocked processes. When each of these processes awake from the **`block_me`** call inside send or receive, they will need to "notice" that the mailbox has been released…

Return values:

-3:	process was **`zap`**'d while releasing the mailbox.
-1:	the mailboxID is not a mailbox that is in use.
0:	successful completion.

`int MboxSend(int mailboxID, void *message, int message_size);`

Send a message to a mailbox. The calling process is blocked until the message has been placed in a slot in the mailbox. The message is copied to the slot using **`memcpy`**. Do not use **`strcpy`**, since the message can contain any type of data, not just string data. If the system is out of mailbox slots, print an appropriate error message and call **`halt(1)`**.

Return values:

-3:	process **`zap`**'d or the mailbox released while blocked on the mailbox.

-1:	illegal values given as arguments.
0:	message sent successfully.

`int MboxReceive(int mailboxID, void *message, int max_message_size);`

Receive a message from a mailbox. The calling process is blocked until the message has been received from the mailbox. The receiver is responsible for providing storage to hold the message. The **`max_message_size`** parameter states the size of the storage.

Return values:

-3:	process **`zap`**'d or the mailbox released while blocked on the mailbox.
-1:	illegal values given as arguments; or, message sent is too large for receiver's buffer (no data copied in this case).
>=0:	the size of the message received.

`int MboxCondSend(int mailboxID, void *message, int message_size);`

Conditionally send a message to a mailbox. Do not block the invoking process. Rather, if there is no empty slot in the mailbox in which to place the message, the value -2 is returned. Also return -2 in the case that all the mailbox slots in the system are used and none are available to allocate for this message.

Return values:

-3:	process was **`zap`**'d.
-2:	mailbox full, message not sent; or no slots available in the system.
-1:	illegal values given as arguments.
0:	message sent successfully.

`int MboxCondReceive(int mailboxID, void *message,`
`int max_message_size);`

Conditionally receive a message from a mailbox. Do not block the invoking process. Rather, if there is no message in the mailbox, the value -2 is returned.

Return values:

-3:	process was zap'd while blocked on the mailbox.
-2:	mailbox empty, no message to receive.
-1:	illegal values given as arguments; or, message sent is too large for receiver's buffer (no data copied in this case).
>=0:	the size of the message received.

4.0 Interrupts

You are to write interrupt handlers for all devices supported by USLOSS. Fortunately, processes will synchronize with interrupt handlers through the **`waitdevice`** routine (see below); this routine causes the process to block on a receive on a zero-slot mailbox associated with the device. The interrupt handler conditionally send's to the device's mailbox. In this manner, processes can wait for I/O to complete. The interrupt handler for a device will also pass the contents of the device's status register in the message; this is the I/O operation's *completion* *status* that allows the process that is waiting for the I/O to determine if the I/O completed successfully. It is only necessary to save the most recent completion status for each device. In a later phase, you will be implementing *device drivers*, processes that handle the I/O devices and therefore wait for I/O completion. There will be only one device driver process associated with a device. Thus, only one process will ever call **`waitdevice`** for a particular device. USLOSS invokes interrupt handlers with two parameters: the first is the interrupt number, and the second is the unit of the device that caused the interrupt (except for the syscall interrupt, in which case it is the argument passed to syscall; see below).

The Interval Timer of USLOSS will be used both for CPU scheduling (enforcing the time slice, from phase 1) and implementing the pseudo-clock. The pseudo-clock is nothing more than a special mailbox that has a message sent to it by the kernel on every fifth clock interrupt (about every 100 milliseconds). Once again, in a later phase, you will implement a clock device driver that will wait on the mailbox and provide a general process delay facility. For now, however, the clock mailbox will be sent a message continually by the interrupt handler every fifth clock interrupt.

`int waitdevice(int type, int unit, int *status);`

Do a receive operation on the mailbox associated with the given unit of the device type. The device types are defined in **`usloss.h`**. The appropriate device mailbox is sent a message every time an interrupt is generated by the I/O device, with the exception of the clock device which should only be sent a message every 100,000 time units (every 5 interrupts). This routine will be used to synchronize with a device driver process in the next phase. **`waitdevice`** returns the device's status register in **`*status`**.

Return values:

-1:	the process was **`zap'`**d while waiting
0:	otherwise

4.1 System Calls

System calls will be implemented starting in the next phase, but you need to implement support for them now. Each system call is numbered in the range **`[0..MAXSYSCALLS-1]`**. You are to implement a system call vector that contains the handlers for the different types of system calls, indexed by the system call number. Calling **`usyscall`** causes a syscall interrupt to occur. The second parameter to the interrupt handler is a pointer to the system call number; your handler is to use this number to index into the system call array and invoke the handler. For now, set all system call handlers to a function named **`nullsys`** that prints an error message and calls **`halt(1)`**. The next phase will begin filling in the vector with more meaningful handlers. If the system call number is invalid your interrupt handler prints an error message and calls **`halt(1)`**.

5.0 Phase 1 Functions

In general, in a layered design such as this one, you should not have to refer to global variables or data structures from a previous phase. The phase 1 functions: **`fork1`**, **`join`**, **`quit`**, **`zap`**, **`is_zapped`**, **`getpid`**, **`dump_processes`**, **`block_me`**, **`unblock_proc`**,

`read_cur_start_time`, and **`time_slice`** are available for you to use. No other phase 1 functions or variables are accessible. For example, you cannot call the **`dispatcher`** directly, reference **`Current`**, etc.

6.0 Initial Startup

After your phase 2 has completed initializing the necessary data structures, it should create a single process running the function **`start2`** in kernel mode with interrupts enabled. This initial process should be allocated **`4 * USLOSS_MIN_STACK`** of stack space, and should run at priority one. **`start1`** should then block on a join to wait for **`start2`** to finish; **`start1`** will then call **`quit`**.

7.0 Phase 1 Library

We will grade your phase 2 using the phase 1 library that we supply. The provided phase 1 library will be named: *libpatrickphase1.a*. This library will be available in */home/cs452/fall15/phase2*.

You may use either your phase 1 library, or the phase 1 library that we will supply while developing your phase 2. However, it will be graded using the phase 1 library that we will supply.

8.0 Submitting Phase 2 for Grading

Note: Programs are submitted via D2L only.

Before submitting to D2L, you will need to create a **`phase2.tgz`** file. There is a target in the provided phase1 **`Makefile`** (*/home/cs452/fall15/phase2/Makefile*) named **`submit`**. It is at the very end of the **`Makefile`**.

Doing make submit will create a **`phase2.tgz`** file that contains:

- The .c files for your phase2. This will include **`phase2.c`**, **`p1.c`**, and any other **`.c`** files that you have created. If you have created additional **`.c`** files, you should have already modified **`COBJS`** to include these additional **`.o`** files.
- The **`.h`** files for your phase1. This will include **`message.h`**. If you have created other **`.h`** files, you should have already modified **`HDRS`** to include the additional **`.h`** file(s).
- The **`Makefile`**.

You will *not* submit the usloss library or directory. You will *not* submit the phase 1 library. We will put a link for usloss and the phase1 libraries into the grading directory after we extract your files. The same is true for the testcases directory; do not submit the testcases. We will add a link for the testcases directory.

You are encouraged to create the **`phase2.tgz`** file, then copy it to a different area of your home directory, extract the file using **`tar xvfz phase2.tgz`** and do a compile to see if everything has been included.

Any file(s) that are missing from your submission that we have to request from you will result in a reduction of your grade!

To submit the **`phase2.tgz`** file to D2L

- Log on to D2L, and select CSc 452.
- Select “Dropbox”.
- You will find *Phase2*, and *Phase2-late*. Only one of these will be active. Click on the active one — will be *Phase2* if you are turning in the work on time.
- You should now be at the page: “Submit Files - Phase2”. Click on the “Add a File” button. A pop-up window will appear. Click on the “Choose File” button. Use the file browser that will appear to select your file(s). Click on the “Upload” button in the lower-right corner of the pop-up window.
- You are now back at “Submit Files - Phase2”. Click on “Upload” in the lower-right of this window. You should now be at the “File Upload Results” page, and should see the message **`File Submission Successful`**.

D2L will send you a confirmation email after each item is placed in the Dropbox. This email is sent to your UA email (which is <yourUANetId>@email.arizona.edu). Retain these emails at least until you have received a grade for the assignment; it is your confirmation that you did submit the program(s).

You may submit phase2 more than once. We will grade the last one that you put in the Dropbox.