Phase 2: Messages and Interrupt Handlers

1 Overview

For the second phase of the operating system project, 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 advanced features such as process-control functions and device drivers.

2 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 the 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 (just as was done in phase 1.)

3 Messages

Phase 2 implements routines for passing messages between processes. The six routines are **MboxCreate, MboxSend, MboxReceive, MboxCondSend, MboxCondReceive, and 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 and by returning a value to indicate success or failure.

Zero-slot mailboxes are a special type of mailbox and are intended for synchronization between a sender and a receiver. These mailboxes have no slots and are used to pass messages directly between processes; they act much like a signal. The sender is 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 ensure that messages are delivered in the order sent and are received in the order the receive function is **called**. In general, the slot mechanism helps to ensure that this order is maintained. However, process priority must be considered here. For example, consider a low priority process that blocks on a receive, followed by a high priority process that subsequently runs and calls 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 and 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**[1]:

1. *MAXMBOX* specifies the maximum number of mailboxes.

2. *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 total slots are more than *MAXSLOTS* so long as there are not more than *MAXSLOTS* actual messages in the system at one time.

3. *MAXMESSAGE* specifies the largest possible size of one message in bytes.

[1] The header file is included in the provided USLOSS package. You should be able to find it in usloss/include with other header files.

In this phase, you need implement the following functions:

- **int MboxCreate(int num slots, int slot size)**;

  Creates a mailbox that can be used for inter-process communication and synchronization. The constant *MAXMBOX*, found in **phase2.h**, defines the maximum number of mailboxes for the system. The function returns a unique mailbox ID is returned that is used in subsequent sends and receives.

  Return Values:

  -1: no mailboxes available, or **slot size** is incorrect.

  ≥ 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 work in every case. It would work for a high priority process releasing low priority processes from the mailbox, but not the other way around [2]. 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 process. 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...[3]

  Return values:

  -3: process was zap'd while releasing the mailbox.

  -1: the mailboxID is not a mailbox that is in use.

[2] If the releasing process is a lower priority one than the blocked processes, the unblocked processes that are holding higher priorities are able to run before the releasing process finishes the mailbox releasing.

[3] Note that this approach makes blocked processes receive a return value indicating they were "zap'd", but the phase 1 code does not have them marked as zap'd. Thus, if **is zapped** is called, it will return 0 to these processes.

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. Treat the message as binary data and use the **memcpy** function to copy the data into the mailbox slot. Do not use **strcpy** to copy messages. If the system is out of mailbox slots, print an appropriate error message and call **halt(1)**.

  Return values:

  -3: process is zap'd, or the mailbox was released while the process was 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 specifies the size of the storage.

  Return values:

  -3: process is zap'd, or the mailbox was released while the process was 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 is zap'd.

  -2: mailbox full, message not sent; or no mailbox 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 in cases where there are no messages to receive. Instead, return a -2 if there are no messages in the mailbox or, in the case of a zero-slot mailbox, there is no sender blocked on the send function.

  Return values:

  -3: process is zap'd.

  -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 Interrupts

You are to write interrupt handlers for all devices supported by USLOSS. USLOSS invokes interrupt handlers with two parameters: the first is the interrupt number, and the second is a pointer that can be cast to the unit number of the device that caused the interrupt (except for the **syscall**

interrupt, in which case the second is the argument passed to syscall; see below). In phase 1, the interrupt vector was defined. In phase 2, you need to initialize the interrupt entries to handle clock interrupt, disk and terminal interrupt, and system calls[4]. (You may need to recheck USLOSS manual for more details.)

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 every **100** milliseconds. Once again, in phase 4, 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 100 milliseconds.

For your interrupt handling, you need to create zero-slot I/O mailboxes for clock, terminals, and disks. For clock, you need have one mailbox for the single unit. For terminal device, you need four mailboxes, one for each terminal unit. For disk device, you need two mailboxes, one for each disk unit. The mailboxes are used for synchronizing between processes and interrupt handlers. For your disk and terminal handlers, you need to read the device status register by using the USLOSS **device input** function[5]. After the device status is retrieved, your handler should conditionally send the content of the status register to the appropriate I/O mailbox.

Processes 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. In this manner, processes can wait for I/O to complete. To synchronize the processes, the interrupt handler conditionally sends to the device's mailbox. The interrupt handler for a device will also pass the contents of the device's status register in the message.[6]

---

[4] USLOSS treats system calls as a form of interrupt, routing them through the interrupt vector

[5] The function, just like psr get or context init, was defined in USLOSS library

[6] The status value will allow 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 phase 4, you will be implementing device drivers and 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.

- **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, except for the clock device which should only be sent a message every 100 milliseconds (every 5 interrupts). This routine will be used to synchronize with a device driver process in phase 4. **waitdevice** returns the device's status register in *status.

  Return values:

  -1: the process was zapped while waiting

  0: otherwise

4.1 System Calls

Implementation of system calls will begin in the next phase, but you need to build support for system calls in this phase. 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 **syscall** causes a **syscall** interrupt to occur. The second parameter to the interrupt handler is a pointer to a struct that contains the system call number; your handler is to use this system call 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 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 Initial Startup

First your phase 2 should implement the routine **start1** to initialize the necessary data structures. Then 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*. The **start1** process should block on a **join** to wait for **start2** to finish; **start1** will call **quit**.

7 Phase 1 Library

I will grade your phase 2 using the phase 1 library that I supply. The provided phase 1 library will be named: **libdurenphase1.a**. This library will be available at D2L web site after phase 1 due date. You may use either your phase 1 library, or the phase 1 library that I will supply while developing your phase 2. However, it will be graded using the **libdurenphase1.a**. After you download the phase1 archive file, you should save it to your phase2 working directory.

8 Submitting Phase 2 for Grading

For the turn-in, you will need to submit all the files that make up your phase 2 project. You are also expected to include a self-evaluation report to evaluate your phase 2 development based on the output of the provided test cases. Design and implementation will be considered, so make sure your code contains insightful comments, variables and functions have reasonable names, no hardcoded constants, etc. You may, if helpful, turn in a **README** file to help me understand your solution. Your **Makefile** must be arranged so that typing 'make' in your directory will create an archive named libphase2.a. Your Makefile must also have a target called 'clean' which will delete any generated files (e.g. .o files, .a files, core files, etc). You should NOT turn in any files that are part of the USLOSS package, including library files (e.g. libusloss.a libdurenphase1.a), usloss.h, phase1.h, and any derived files.