CS 452 (Spring 23): Operating Systems

Phase 2 - Mailboxes and Devices

due at 5pm, Thu 2 Mar 2023

1 Phase 2 Overview

In this Phase, you will implement a "mailbox" mechanism. This mechanism will be critical for how we handle interrupts in our simulation, as well as for some other features that we'll be implementing.

The vast majority of your time in this Phase will be implementing and testing the mailbox mechanism - but once that is ready, you will also add some new interrupt handling. You won't be implementing the actual device drivers until Phase 4 - but in this Phase you will build the mechanism which will make it possible.

2 Interaction with Phase 1

Your code in this Phase may utilize any of the functions that were defined for Phase 1. You will **not** provide your own Phase 1 implementation; instead, you will use a library which I have provided for you; see the class website.

You will not be able to access any of the private data or helper functions of the Phase 1 library - the only thing that you can do is to call the functions that I defined in the Phase 1 spec.

To use the Phase 1 library, find the correct library for the OS that you are running on (use `linux` if you are inside Docker), and rename that file to `libphase1.a` .

3 Mailboxes Overview

In our system, a "mailbox" is a mechanism that allows you to send and receive messages; the messages will be sent in discrete packets of information, ranging from 0 to `MAX_MESSAGE` bytes long (inclusive).

A mailbox can, potentially, have any number of processes that send messages and receive messages; although it would be unusual, it's even legal for a process to send a message, and then receive its own message back. We call a process that is sending message(s) a **"producer,"** and a process that is receiving messages(s) a **"consumer."**

Messages that are sent to the mailbox can sometimes be queued up - that is, stored into slots for later delivery. Each mailbox has a limit to how many slots it is willing to consume, and there is also a global limit of how many slots exist. If we queue up a message into one of the slots, then the producer is not required to block; it may record its message and immediately move on. Later, when a consumer attempts to receive a message, the consumer will read it from the slot, and likewise will not block.

On the other hand, both producers and consumers sometimes block. Producers block when they attempt to send a message but the mailbox already has consumed its maximum number of allowable mail slots; it will block until one or more of the messages are consumed, at which time the producer will be allowed to allocate a slot and write its message. Consumers block when there are no queued messages; when a producer finally writes a message to a slot, the consumer will read the message, free the slot, and then return.

Additionally, your mailbox mechanism will also implement "conditional" send and receive functions; these work exactly like the normal send and receive, except that they refuse to block. If they determine that blocking would be necessary, they instead return an error return code to the caller.

4 Ordering Guarantees

Producers must deliver their messages in the same order that they arrived, and consumers must receive messages in the same order that they arrived. By "arrived," I mean when the function first is called, and either completes its work, or blocks.

Thus, if one or more processes block, you must arrange them into a queue, and make sure that they do their work in the proper order, as if they had not blocked. You will need a queue for the producers, another queue for the consumers, and likewise a queue of mail slots, for the pending messages.

4.1 Ordering Example #1

When waking blocked processes, you must ensure that the right consumer receives the proper message (based on the order in which the producers arrived, and the consumers arrived). However, it will not be possible for you to control the order in which the processes **report** that they have completed their work (sending or receiving). To see this problem in action, consider the following code snippets. Let's suppose that we have a first process, which happens to be running at priority 5 (the lowest):

```
// MASTER PROCESS - priority 5
fork1(receiverProc, ... 4);
fork1(receiverProc, ... 3);
fork1(receiverProc, ... 2);
```

This process creates three other processes, each running the same code; they all are higher priority than the "master," but we create them in reverse order.

Now, let's look at the code for the receiver processes:

```
// RECEIVER PROCESSES - priorities 4,3,2
recvMsg(...)
printf("Process %d: recv() completed!\n", processNum);
```

So, each receiver process blocks on the mailbox, waiting for a message to arrive. Since each of these processes is higher priority than the master, they run (and block on the mailbox) immediately when they are created. Therefore, the priority 4 process is first in the queue on the mailbox, followed by the priority 3 process and the priority 2 process.

Let's draw a picture of what the mailbox looks like at this point in time:

```
Mail Slot Queue: -> <empty>

                   +---------+    +---------+    +---------+
Consumer Queue: -> | recv()  | -> | recv()  | -> | recv()  |
                   | pri 4   |    | pri 3   |    | pri 2   |
                   | BLOCKED |    | BLOCKED |    | BLOCKED |
                   +---------+    +---------+    +---------+

Producer Queue: -> <empty>
```

Now, suppose that the master process creates one more process, at priority 1 (that is, higher than all the rest). It will send 3 messages, each with different contents:

```
// SENDER PROCESS - priority 1
send("foo");
send("bar");
send("baz");
```

Since this process is priority 1, it will send all three messages immediately, queueing them into various mail slots.

4.1.1 Conundrum - How/When to Wake the Receviers?

Three messages are now available. How should you wake up the Receive processes? If you simply wake all three, and allow them to race for the messages, then the priority 2 process will run first, and collect the `"foo"` message. But this is **wrong** - this process was the third to queue up to receive messages, and it must therefore receive the third message that was sent (`"baz"`).

Your group must come up with a solution to this problem. I can imagine several possible solutions; I've given you two example solutions below.

4.1.2 Wakeup Strategy #1 - One At A Time

One possibility is to only wake up processes one at a time. Thus, after our sender sends three messages, the mailbox looks like this:

```
                        +-------+    +-------+    +-------+
    Mail Slot Queue: -> | "foo" | -> | "bar" | -> | "baz" |
                        +-------+    +-------+    +-------+


                       +---------+    +---------+
    Consumer Queue: -> | recv()  | -> | recv()  |
                       | pri 3   |    | pri 2   |
                       | BLOCKED |    | BLOCKED |
                       +---------+    +---------+

    Producer Queue: -> <empty>

                        +---------+
    Runnable Processes: | recv()  |
                        | pri 4   |
                        +---------+
```

(example continues on next page)

The priority 4 process receives the `"foo"` message, and is about to return - but before it does so, it looks at the Consumer Queue, and notices that there is both (a) a consumer; and (b) a message for them to read. It wakes up that process, as well. Since the newly-awoken process is higher priority, we immediately do a context switch to it:

```
    Mail Slot Queue: -> +-------+    +-------+
                        | "bar" | -> | "baz" |
                        +-------+    +-------+

                       +---------+
    Consumer Queue: -> | recv()  |
                       | pri 2   |
                       | BLOCKED |
                       +---------+

    Producer Queue: -> <empty>

                        +---------+
    Runnable Processes: | recv()  |
                        | pri 3   |
                        +---------+
                        +---------+
                        | recv()  |
                        | pri 4   |
                        | "foo"   |
                        +---------+
```

(example continues on next page)

The priority 3 process does the same, receiving `"bar"` and then waking the priority 2 process:

```
                    +-------+
Mail Slot Queue: -> | "baz" |
                    +-------+

Consumer Queue: -> WARNING WARNING RACE CONDITION HERE!

Producer Queue: -> <empty>

                    +---------+
Runnable Processes: | recv()  |
                    | pri 2   |
                    +---------+
                    +---------+
                    | recv()  |
                    | pri 3   |
                    | "bar"   |
                    +---------+
                    +---------+
                    | recv()  |
                    | pri 4   |
                    | "foo"   |
                    +---------+
```

But notice, in this scenario, that we have a **race condition,** which can corrupt how we deliver messages. We have woken up the priority 2 process, and we intend for it to receive the `"baz"` message. (And it probably will.) But what if a **priority 1** process wakes up at exactly this moment? It might read the message and remove it - thus corrupting the required order!

Thus, if you follow this strategy, you will need to have some sort of mark, which indicates "a consumer has already been queued, and has rights to the message already - any new consumers must queue themselves." Exactly how you do this is **up to you.**

(spec continues on next page)

4.1.3 Wakeup Strategy #2 - Direct Delivery

It's easy to see, from the previous examples, that we don't want to simply wake up all three of the receive processes, all at once - since they would race for the messages, and the messages would be collected by the wrong processes.

But an alternative is to have the **sender deliver the messages directly to the receivers.** In this design, we still have a mail slot queue (because sometimes senders act before receivers), but if a process attempts to send a message while receivers are already queued up, it delivers the message directly to the receiver, and wakes it up.

The following picture shows what the mailbox would look like, under this strategy, after only the **first message** has been sent. The sender process is still running, and the priority 4 receiver has been woken up - but there is nothing in the mail slot queue, because the message was written directly to the receiver:

```
Mail Slot Queue: -> <empty>

                   +---------+    +---------+
Consumer Queue: -> | recv()  | -> | recv()  |
                   | pri 3   |    | pri 2   |
                   | BLOCKED |    | BLOCKED |
                   +---------+    +---------+

Producer Queue: -> <empty>

                    +-------------+
Runnable Processes: | send()      |
                    | pri 1       |
                    | in 1st send |
                    +-------------+
                    +-------------+
                    | recv()      |
                    | pri 4       |
                    | "foo"       |
                    +-------------+
```

(example continues on next page)

After the send process sends two more messages, it has woken up two more blocked processes:

```
Mail Slot Queue: -> <empty>

Consumer Queue: -> <empty>

Producer Queue: -> <empty>

                    +-------------+
Runnable Processes: | send()      |
                    | pri 1       |
                    | in 3rd send |
                    +-------------+
                    +-------------+
                    | recv()      |
                    | pri 2       |
                    | "baz"       |
                    +-------------+
                    +-------------+
                    | recv()      |
                    | pri 3       |
                    | "bar"       |
                    +-------------+
                    +-------------+
                    | recv()      |
                    | pri 4       |
                    | "foo"       |
                    +-------------+
```

Once the 3rd send completes, the send process will die - at which point, the priority 2 process will report that it received the "bar" message. The other processes will eventually report that they received the message later, when the priority 2 process blocks or dies.

5 Zero-Slot Mailboxes

Your system must include support for zero-slot mailboxes. These are mailboxes where you will **never** consume any slots for any messages; instead, your producers will wait, blocking, until a consumer has arrived (or vice-versa).

While the testcases will often create zero-slot mailboxes which have nonzero slot lengths (meaning that, in theory, they could send or receive messages that have nonzero length), you may assume that they will not actually send or receive any such messages. That is, all messages sent to zero-slot mailboxes will be zero length.

6 Zero-Message-Length Mailboxes

Your system must include support for mailboxes where the maximum message size is zero. Treat these no differently than other mailboxes, which use longer length messages. (In particular, these must consume mail slots to “queue” them, and you must enforce the mail-slot-limit for the mailbox.)

7 Interrupt Handling - Overview

In this Phase, you will add new mechanisms to handle more interrupt types. First, you will implement the functions `phase2_clockHandler()` and `phase2_check_io()`, which are called by Phase 1; the first is called by the clock interrupt handler, and the second is called by `sentinel`.

Second, you will add interrupt handlers for the disk and terminal devices. Unlike the clock (which only has one) each of these interrupts handles multiple devices; the parameters to the interrupt handler will tell you which device generated the interrupt. You will add several mailboxes, and will send messages on these mailboxes to indicate when interrupts have arrived on certain devices; you will then provide functions which will allow other processes (which will be implemented later) to block waiting for interrupts to arrive.

Finally, you will implement the syscall handler mechanism. You will implement the syscall interrupt handler, as well as an array of function pointers to handle the various syscalls. You will **not** implement any syscalls in this phase (we’ll do it in the next), but you will implement all the mechanisms which allow user-mode programs to send requests to kernel-mode code.

You will find important constants, such as the correct type numbers for interrupts and devices, in `usloss.h`.

8 Disk and Terminal Interrupt Handlers

The declaration for an interrupt handler has the following type:

```
void abc(int,void*);
```

The first parameter is the interrupt type. Since (I presume) you will be using different interrupt handler functions for each type of interrupt, you may ignore this parameter.

The second parameter is a generic payload variable. How it is used depends on the interrupt type. In disk and terminal interrupts, it is simply an integer; cast the pointer to an integer directly. (Since `int` is fewer bytes than a pointer on most modern architectures, gcc may complain if you cast it directly. If you prefer, you may do a two-step cast, which will eliminate the gcc warning.)

```
int unitNo = (int)(long)arg2;
```

In USLOSS, there are two different disks, and four terminals. When a disk or terminal interrupt arrives, you will need to determine which "unit" each arrived on.

In some operating systems, the interrupt handler would directly handle a device (or at least, directly call a function which handled the device). However, in our simulation, we will do all of the device handling in specialized processes. Thus, your interrupt handler must **wake up** a given process, and notify it that there is work to do. To do this, you will use a mailbox. For each unit (that is, 2 disks, 4 terminals, and a clock that I will describe later), define a mailbox which will only buffer a single message in a slot, and will only send messages large enough to hold a single `int`; you will send messages to it to wake up the device driver, when an interrupt occurs.

Specifically, when an interrupt occurs, you will first determine which unit sent the interrupt. You will then use `USLOSS_DeviceInput` to read the current status from the device; you will send this status as the payload for a message. When sending a message, you will always use `MboxCondSend()`, because it is imperative that you never block a process inside of an interrupt handler. In normal operation, we expect that the device driver will be reading messages from the queue extremely rapidly, and thus you should never fail to send a message; however, if the device driver falls behind, your interrupt handler **must** allow the message to be lost, instead of blocking.

9 Delay Mechanism

The USLOSS clock device sends interrupts (roughly) every 20 milliseconds. You will implement a mechanism, similar to the disk and terminal device interrupt handler mechanisms, but which sends a message on a mailbox (roughly) every 100 milliseconds. Specifically, each time that you are notified about a clock interrupt, check the current wall-clock time. If it has been 100 ms (or more) since you last sent a message on the mailbox, send another.

Create a mailbox for this stream of messages, just like you did for the disk and terminal interrupts. The process (part of a later Phase) which wishes to receive messages from this mailbox will call `waitDevice()` with `USLOSS_CLOCK_DEV`.

10 Sentinel Helper

In Phase 1, the `sentinel` process calls `phase2_check_io()` to see if any processes are currently blocked waiting for interrupts. You will keep track of how many processes are currently blocked inside of `waitDevice()` (see below), and you will return 1 from `phase2_check_io()` if one or more processes are blocked.

11 Syscall Handler

Syscalls allow user-mode programs to make a request of the kernel. In most OSes (and likewise in our system), we assign different integer values to various syscalls; the syscall mechanism also allows for several (typically a half-dozen or so) parameters. Depending on the OS, these parameters might be passed in registers, or they might be passed in a structure that is stored in memory.

In our system, syscalls use a struct named `USLOSS_Sysargs`, which is defined in the USLOSS header `usyscall.h` . The first field, `number`, indicates which system call is being requested; it also has 5 arguments, which are all `void*`. (The argument fields will also be used for returning results to the caller.)

When the user triggers a system call, they will pass a pointer to a struct of this type. This will be passed as the 2nd parameter to your interrupt handler (recall that its type is `void*`). You can cast this pointer from `void*` to `USLOSS_Sysargs*`; you can then access the fields of that struct.

In this Phase, you will not be implementing any system calls - however, you will be implementing the mechanism which dispatches them. Just like interrupts require interrupt handlers to be installed on the `USLOSS_IntVec[]` array, system calls require that syscall handler to be installed on the `systemCallVec[]` array.

`systemCallVec[]` has an `extern` declaration in `phase2.h`; your Phase 2 code must include a non-`extern` declaration of it. The size of this array must have exactly `MAXSYSCALLS` elements.

Your `phase2_init()` function must set all of the elements of `systemCallVec[]` to `nullsys`, a function that you will implement.

The `nullsys()` function (which may be static if you wish - no code outside your Phase will call it directly) will simply print out an error message and terminate the simulation.

In later Phases, you will write code that adds handlers for various system calls (similar to how you have added interrupt handlers, for various interrupt types).

12 Required Data Structures

You will implement a (non-static, non-extern) array of function pointers, named `systemCallVec[]` (see above for details).

You will also need to implement the following arrays; these will not be accessed by other Phases, and thus should be declared `static`:

- An array of mailboxes, with exactly `MAXMBOX` elements. (Just like the Phase 1 process table, you will define whatever fields you want for this.)
- An array of mail slots, with space for exactly `MAXSLOTS` messages. Each slot must be able to handle messages up to `MAX_MESSAGE` bytes.

  These slots must be **shared** across all of the mailboxes; do not have per-mailbox slots. Again, you can define the fields of each mail slot however you want.

- A "shadow" process table. Since your Phase 2 cannot access the Phase 1 process table, it is **impossible** to add fields to the Phase 1 PCB. Instead, you will declare a new process table, that exists only for Phase 2, which will include any fields you need for Phase 2 - such as the mechanisms for queueing blocked processes.

  Remember how you can use the `%` operator to map PID to a slot in the array - it will be trivial for you to figure out the proper slot to store your data, for any running process.

  NOTE: You will not be warned if a process dies and the slot is re-used. Design your shadow process table in such a way that this won't be a problem.

13 Required Functions - Detailed Specifications

Just like in Phase 1, all of the required functions must be called in kernel mode; you must check this, and terminate the simulation if they are called from user mode.

In this Phase **nearly all** functions must be called in process context; the only exceptions are `MboxCondSend()`, `MboxCondRecv()`, and `phase2_clockHandler()`.

13.1 `int MboxCreate(int numSlots, int slotSize)`

May Block: no

May Context Switch: no

Args:

- `numSlots` - the maximum number of slots that may be used to queue up messages from this mailbox
- `slotSize` - the largest allowable message that can be sent through this mailbox

Return Value:

- -1 : `numSlots` or `slotSize` is negative, or larger than allowed (see constants in `phase2.h`). Or, there are no mailboxes available.
- >= 0 : ID of allocated mailbox

Creates a new mailbox. You may choose any way to assign IDs for your created mailboxes (I simply return the index into the array of mailboxes).

If you destroy a mailbox, and then later create a new one, it is permissible to re-use an old mailbox ID.

13.2 `int MboxRelease(int mailboxID)`

May Block: no

May Context Switch: yes

Args:

- `mailboxID` - the ID of the mailbox

Return Value:

- -1 : The ID is not a mailbox that is currently in use
- 0 : Success

Destroys a mailbox. All slots consumed by the mailbox will be freed. All blocked producers and consumers will be unblocked, and return -3.

Once the mailbox has been marked as destroyed, no more processes will be allowed to block on it; any attempt to `Send()` or `Recv()` on it will return -3.

You must destroy the mailbox (and wake its blocked processes) **promptly** but not necessarily **instantly.** By "promptly," I mean that the various blocked processes should be awoken, removed from any pending queues, etc. as soon as is practical. However, we do **not** guarantee that all of these processes will be awake when this function returns - and as such, it might not be possible to re-create the mailbox immediately after this function returns.

Why might this happen? It depends on the mailbox implementation. In your implementation, this might not be a problem - but in my implementation, only one producer, and one consumer, can be waking up at a time - meaning that it takes a while to "flush" any blocked producers and consumers from the various queues.

13.3 `int MboxSend(int mailboxID, void *message, int messageSize)`

May Block: yes

May Context Switch: yes

Args:

- `mailboxID` - the ID of the mailbox
- `message` - pointer to a buffer. If `messageSize` is nonzero, then this must be non-NULL. Otherwise, a NULL pointer is permissible.
- `messageSize` - the length of the message to send

Return Value:

- -3 : The mailbox was released (but is not completely invalid yet)
- -2 : The system has run out of global mailbox slots, so the message could not be queued
- -1 : Illegal values given as arguments (including invalid mailbox ID)
- 0 : Success

Sends a message through a mailbox. If the message is delivered directly to a consumer or queued up in a mail slot, then this function will not block (although it might context switch, even then, if it wakes up a higher-priority process).

If there are no consumers queued and no space available to queue a message, then this process will block until the message can be delivered - either to a consumer, or into a mail slot.

13.4 `int MboxRecv(int mailboxID, void *message, int maxMessageSize)`

May Block: yes

May Context Switch: yes

Args:

- `mailboxID` - the ID of the mailbox
- `message` - pointer to a buffer. If `messageSize` is nonzero, then this must be non-NULL. Otherwise, a NULL pointer is permissible.
- `maxMessageSize` - the size of the buffer. The receiver can receive **`up to`** this size, but might receive less.

Return Value:

- -3 : The mailbox was released (but is not completely invalid yet)
- -1 : Illegal values given as arguments (including invalid mailbox ID). Or, the message received was too large for the receiver's buffer.
- >= 0 : The size of the message received

Waits to receive a message through a mailbox. If there is a message already queued in a mail slot, it may read it directly and return (but be careful to obey the ordering rules we discussed earlier in the spec). Otherwise it will block until a message is available. (But note the special rules for zero-slot mailboxes, see above.)

13.5 int MboxCondSend(...), int MboxCondRecv(...)

These functions work exactly like their non-`Cond` versions, except that they refuse to block. If, at any point, they would normally have to block, they will return -2 instead.

While these functions will never block, they might context switch, if they wake up a process that was higher priority than the one on which the interrupt handler is running. This is **not a problem.**

Note that you may find it useful, instead of implementing two different copies of `Send()` and `Recv()`, to instead create (private) helper functions, which both the `Cond` and non-`Cond` versions of your functions can call. But remember: you **must not** change the declaration of any function called by the testcases!

13.6 int waitDevice(int type, int unit, int *status)

May Block: yes

May Context Switch: yes

Args:

- `type` - the device type: clock, disk, or terminal. (See `usloss.h`)
- `unit` - which "unit" of a given type you are accessing a NULL pointer is permissible.
- `status` - an **out** parameter, which will be used to deliver the device status once an interrupt arrives.

Return Value:

- 0 : Always

Waits for an interrupt to fire, on a given device. Only three device types are valid: the clock, disk, and terminal devices. The unit field must be a valid value (0 for clock; 0,1 for disk; 0,1,2,3 for terminal); if it is not, report an error and halt the simulation.

This function will `Recv()` from the proper mailbox for this device; when the message arrives, it will store the status (remember, the status was sent, as the message payload, from the interrupt handler) into the out parameter and then return.

13.7 void phase2_init(void)

May Block: no

May Context Switch: no

Very similar to phase1_init(), this function is called by the testcase during bootstrap, before any processes are running. Use it to initialize any data structures that you plan to use. You **must not** attempt to fork1() any processes, or use any other process-specific functions, since the processes are not yet running.

13.8 void phase2_start_service_processes(void)

May Block: no

May Context Switch: yes

Called by Phase 1 from init, once processes are running but before the testcase begins. If your implementation requires any service processes to be running for Phase 2 (I don't expect that it will), then this is the place to call fork1() to create them.

13.9 int phase2_check_io(void)

May Block: no

May Context Switch: no

Called by Phase 1 from sentinel, if that process ever runs. Return nonzero if any process is currently blocked inside waitDevice(), or zero if no processes are blocked.

13.10 void phase2_clockHandler(void)

May Block: no

May Context Switch: no

Called by Phase 1 from the **clock interrupt.** Use it to implement any logic that you want to run every time that the clock interrupt occurs.

14 Turning in Your Solution

You must turn in your code using GradeScope. While the autograder cannot actually assign you a grade (because we don't expect you to match the "correct" output with perfect accuracy), it will allow you to compare your output, to the expected output. Use this report to see if you are missing features, crashing when you should be running, etc.