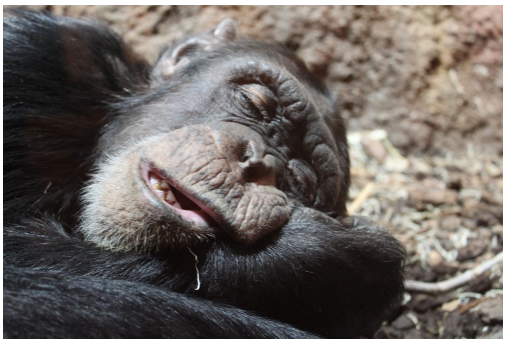
Lab 4 - Inter Process Communication, Process Management & Keyboard Handling

Operating Systems, CS-UH 3010

Assigned:	Oct 31, 2022	Due:	Nov 14, 2022



Do not share the start codes or virtual image provided in this lab, as they were kindly provided by Princeton without permission to redistribute.

1 Overview

In this project, you will add the following functionality to the previous assignment's kernel. We recommend you tackle these problems in this order:

1. Implement a **spawn** system call
2. Implement inter-process communication using message boxes
3. Implement a handler for the keyboard interrupt
4. Implement a **kill** system call
5. Implement a **wait** system call

or

1. Implement message-boxes
2. Keyboard input (`do_getchar()`)
3. Process management (**spawn, kill, wait**)

2 Test Cases

The source code is distributed with two test cases:

- `test_given` implements a simple shell.

- `robinhoodandlittlejohn` features three processes that interact using spawn, kill and wait, and which communicate using IPC.

To select a test case, use the tasks script just like in project 3.

3 IPC

Processes will be able to communicate via first-in, first-out message box queues. These queues should be an efficient implementation of the bounded-buffer problem. Message boxes support four operations, each with a corresponding system call:

- **mbox_open(name)** will create an IPC queue named name. If a queue already exists under that name, then instead return the existing queue and increment its usage count.

  This allows multiple processes to access the same message box if they use the same name. This syscall should return an integer that uniquely specifies the message box.

  You may assume that

  - There will never be more than `MAX_MBOXEN` message boxes open at any time o The name of a message box is not more than `MBOX_NAME_LENGTH` characters long.
  - The bounded buffer will hold no more than `MAX_MBOX_LENGTH` messages.
- **mbox_close(mbox)** will decrease the usage count of the specified message box. If no one is using that message box, it will deallocate this message box. If the message box has not been initialized, then the behavior is undefined.
- **mbox_send(mbox,buf,size)** will add the message buf to the specified message box. size is the length of the message, in bytes. If the message box is full, then the process will block until it can enqueue the message. You may assume that the message is no longer `MAX_MESSAGE_LENGTH` bytes. If the message box has not been initialized, then the behavior is undefined.
- **mbox_recv(mbox,buf,size)** will receive the next message from the message box. It will copy the contents of the message into buf. It will not copy more than size bytes, as to avoid buffer overflows. If the message box has not been initialized, then the behavior is undefined.

4 Handling the Keyboard

You will write a handler for **irq1**. This handler will receive bytes from the keyboard and buffer them using a message box. If the keyboard buffer is full, the handler must instead discard the character.

You must also implement the `get_char` system call. This system call will try to read a character from the keyboard buffer or block until one is available.

To aid in your debugging, the initial code distribution contains a dummy implementation of `get_char`. This implementation repeatedly types out the strings:

```
help
plane
count
```

These strings are commands for the shell in the `test_given` test case.

5 Process Management

5.1 Spawning Processes

The **spawn()** system call should create a new running process.

First, it must look up a process by name. Since we have not yet implemented file systems, you are provided a dummy filesystem defined in ramdisk.h. The test cases each define their own files.

You may assume a finite number of running processes (`NUM_PCBS`).

Return the pid on success, -1 if unable to find the process, -2 if there are too many processes.

5.2 Killing Processes

The **kill()** system call should change the state of a process such that it will die (soon.)

Special care must be taken in certain circumstances; for instance, there may be difficulties if this process is not in the ready queue. Think about this problem, and discuss your solution at design review.

When a process is killed, no effort should be made to release any resources that it holds (such as locks).

The kill system call should immediately kill a process, even if it is sleeping or blocked (even on a wait() call.). If a process is killed while sleeping, other sleeping processes should still wake on schedule. If a process is killed while blocked on a lock, semaphore, condition variable or barrier, the other processes which interact with that synchronization primitive should be unaffected.

If a process is killed while it is in the ready queue, lottery scheduling should still give proportional scheduling to the surviving processes.

5.3 Waiting on Processes

The wait() system call should block the caller until a given process has terminated. You implementation must be efficient.

6 Hints

- During system intialization, calls to `leave_critical()` will (re-)enable interrupts prematurely. To conquer this, you may want to define `_helper()` variants of several system calls to avoid this problem (compare `tolock_acquire_helper()`).
- It might be helpful to define a new queue method that, given a `node_t*` will remove that element from whatever list it is a member of.
- The calls `mbox_send()` and `mbox_recv()` can be implemented without a critical section. In fact, you should implement these without a critical section.
- You should reclaim stacks.

7 Design Review

The design review will be worth 5 points, distributed as follows:

1. **Message Data Structures**: Explain what fields you will need in the MessageBox and Message structs in mbox.c. *1 point*

2. **Keyboard Interrrupt**: How will you use inter-process communications to make your keyboard interrupts work? *1 point*

3. **Process Management**: Describe your design for an efficient implementation of wait. Describe your designs for spawn and kill. Under what conditions should each of the spawn, kill and wait calls fail? *2 points*

4. **Inter-Process Communication**: Explain how two processes can both access the same message box, even though their address spaces are (conceptually) disjoint. *1 point*

8 Grading Rubric

The project is worth 15 points plus one for extra credit. The breakdown is as follows:

Task	**Deliverables**	**Points**
Design review	Correct descriptions of the main concepts of the project.	2
Send & receive	Mbox send and receive correctly.	2
Keyboard	Keyboard input handled correctly.	1
Spawn	Spawn correctly schedules a new process.	2
Wait	Wait correctly blocks a task and wakes up on kill or exit.	1
Kill	Kill correctly removes tasks without affecting others.	2
PCB reclaim	Correctly reclaim pcbs and mboxen.	1
Stacks reclaim	Correctly reclaim stacks.	1
Release locks (Bonus)	Correctly release the locks held by the killed task.	+1

9 Acknowledgments

Once again, we gratefully acknowledge the kindness of the staff of Princeton's COS 318 course for providing us with the material for (and most of the text of) this assignment.

Like always, best wishes and good luck!