Precept 4: IPC & Process Mngmt.

COS 318: Fall 2020

Project 4 Schedule

- **Precept:** Monday 10/26 & Tuesday 10/27, 7:30- 8:20pm
- **Design Review:** Monday 10/26 & Tuesday 10/27, 4 - 7pm
- **Due:** Sunday 11/1, 11:55pm

Project 4 Overview

- **Goal:** Add process management and inter-process communication to the kernel
- Read the project spec for details
- Starter code can be found on the lab machines (/u/318/code/project4)
- **Start early**

Project 4 Overview

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

Project 4 Implementation Checklist

1. `do_spawn`: creates a new process (kernel.c)
2. `do_mbox_*`: mbox functions to enable IPC (mbox.c)
   - `open`, `close`, `send`, `recv`, `is_full`
3. Handle keyboard input: `putchar`, `do_getchar` (keyboard.c)
4. `do_kill`: kills a process (kernel.c)
5. `do_wait`: waits on a process (kernel.c)

System Calls

Spawn

- Kernel has a fixed array of PCBs (`NUM_PCBS`)
- Spawn looks for the program by name in a filesystem (see `ramdisk.h`), and creates a new process if found
- What information do you need to initialize a process?
  - PID
  - New stacks (user/stack)
  - Entry point (ramdisk_find)
  - `total_ready_priority` (lottery scheduling)
- Scheduler uses lottery scheduling → Make sure you keep the sum of the priorities updated
- Tips: see `ramdisk.c` and `kernel.c` for some useful helper functions

Kill

- A process should be killed immediately
- Which queue it's in (ready, blocked, sleeping, etc.) doesn't matter – kill it!
- If blocked on any resource (e.g `locks, semaphores, conditional variables, barriers`), does not affect other processes that interact with this resource
- Do not reclaim locks (this is extra credit)
- Reclaim memory:
  - PCB
  - Stacks
  - Look at robinhood test case to determine what else needs to be reclaimed
- Update `total_ready_priority`

Wait

- Waits for a process to terminate:
  - Blocks until the process is killed or exits normally
- What do you need to add to the PCB to implement this behavior?
- Return -1 on failure, 0 on success

Message Passing + Keyboard

Message Box - Overview

- Used for inter-process communication
  - Processes can both put and consume data from the message box
- It’s a bounded buffer problem!
  - **`send`** blocks if the message box is full
  - **`recv`** blocks if there are no messages

Message Box - Implementation

- Implemented as a circular buffer
  - Array, with head and tail pointers
- Receive messages in FIFO order
- Messages can have variable length
  - But, there is a fixed max length. See constants at bottom of **`common.h`**

Message Box - Suggestions

- Use locks and CVs as shown in class
  - Probably need two CVs: **`fullBuffer`** and **`emptyBuffer`**
- Multiple producers + consumers: protect against race conditions
- Review Lecture 9 and MOS 2.3.7-8

Keyboard - How does it work?

- **`IRQ1`** interrupt generated on key press or release
- Interrupt handler gets key scan code from hardware
- Specific key handler called, based on key type:
  - Modifier Key: change internal state
  - Other Keys: convert scan code to ASCII char + post to keyboard buffer

Keyboard - Software Design

- **`kernel.c:init_idt`** sets keyboard handler to **`entry.S:irq1_entry`**
- **`irq1_entry`** saves context + calls **`keyboard.c:keyboard_interrupt`**
- **`keyboard_interrupt`** gets scan code from hardware + calls specific key handler…

Keyboard - Software Design

- Modifier keys get their own handlers
- **`normal_handler`** catches everything else:
  - Converts scan code to ASCII character
  - Calls **`putchar`** to add it to keyboard buffer
- Processes read from buffer with **`get_char`**

Keyboard - What you need to do

- Implement **`putchar`** and **`do_getchar`**
  - Use your message box API!
- Producer should not be blocked
  - If keyboard message box is full, discard the character
  - Use **`do_mbox_is_full`** to check beforehand
- What if IRQ1 occurs while a process is calling **`get_char`**?

Tips + Other Notes

- Synchronization is tricky: think carefully about when / how to use locks, CVs, and critical sections
- Look at **`util.h`** + other **`.h`** files for helpful functions
- May need to change other pieces of code - this is fine
  - Make sure you submit them!
- Only two test cases provided: write your own unit tests

Design Review

- **Process Management:**
  - How will your spawn, wait, and kill work?
  - How will you satisfy the requirement that *if a process is killed while blocked on a lock, semaphore, condition variable or barrier, the other processes which interact with that synchronization primitive will be unaffected*?
- **Mailboxes:**
  - What fields will the structs need?
  - Which synchronization primitives will you use?

Questions?