Precept 4: IPC & Process Mngmt.

COS 318: Fall 2021

Project 4 Schedule

- **Precept:** Monday 10/25, Tues 10/26, 7:30 PM
- **Assn 3 Due:** Wedn 10/27 11:55 PM
- **Design Review:** Wedn 10/27 5 - 7pm, 8-10PM, Thurs 10/28 5-7 PM*
- **Assn 4 Due:** Sunday 11/7, 11:55pm

Preemption/Scheduling Finished!

- Now: Kernel is able to handle many different processes all running at once, and can automatically preempt them to give time for each process.
- Also have synchronization primitives so processes can work together, if known which will be alive together.

- TODO: All these processes need to be set up at compile time when the kernel is being built. How can we make it so that new processes can be added dynamically at run-time?
- And, if we cannot have statically-known memory, how do processes communicate?

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

System Calls

Spawn(char* filename)

- Kernel has a fixed max amount of PCBs
- 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 (i.e. sampling)
- Need to keep the sum of the priorities updated

Kill(some pid)

- A process should be killed (immediately?) after this is invoked
- Any queue it's in (ready, blocked, sleeping, etc.) doesn't matter – kill it! Two methods of implementation, lazy or eager
- 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(other pid)

- 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
- Implementing a bounded buffer:
  - **`send`** blocks if the message box is full
  - **`recv`** blocks if there are no messages

Message Box - Implementation

- Recommend implement 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 10 and MOS 2.3.7-8

Message Boxes Outside of OS

- Erlang: “Actor Model”, many independent processes that communicate with messages
- Golang: Green Threads (i.e. userspace threads) with channels
- Common in distributed systems, networking
- https://go.dev/blog/codelab-share

Message Passing vs. Shared Memory

- In theory, if you had implemented a heap allocator, could use shared memory with sync primitives instead of message passing
- What are the potential pros/cons of using shared memory vs message passing?

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`** (same as irq0)
- **`irq1_entry`** saves context + calls **`keyboard.c:keyboard_interrupt`**
- **`keyboard_interrupt`** gets scan code from hardware + calls specific key handler

Keyboard - Software Design

- keyboard.c contains logic for handling keys
- Modifier keys (shift, ctrl, etc) w/ own handlers
- **`normal_handler`** catches character keys:
  - 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

- Keyboard stores typed characters in an mbox
- Implement **`putchar`** and **`do_getchar`** for handling typed keys and syscall for getting keys
- Producer should not block
  - 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`**?

Note: Keyboard vs Other Devices

- While the keyboard (and mouse) is special-cased, other devices do not have their own interrupt handlers.
- Other devices may adhere to different protocols, such as communicating with USB or hard-drives.
- Because of many different standards, device drivers often are the buggiest part of an OS, but keyboard can still have subtle bugs.

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!
- Be careful of pass by value in C
- 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?