# Midterm Review

## Computer Operating Systems, Spring 2025

**Instructors:** Joel Ramirez Travis McGaha

**Head TAs:** Ash Fujiyama Emily Shen Maya Huizar

**TAs:**

| | | | | |
|---|---|---|---|---|
| Ahmed Abdellah | Bo Sun | Joy Liu | Susan Zhang | Zihao Zhou |
| Akash Kaukuntla | Connor Cummings | Khush Gupta | Vedansh Goenka | |
| Alexander Cho | Eric Zou | Kyrie Dowling | Vivi Li | |
| Alicia Sun | Haoyun Qin | Rafael Sakamoto | Yousef AlRabiah | |
| August Fu | Jonathan Hong | Sarah Zhang | Yu Cao | |

# Poll Everywhere

pollev.com/tqm

- Any questions?

# Administrivia

- Penn-shell
  - Late deadline is tonight
  - You and your partner should not submit separately.
    You and your partner should have 1 submission together.
  - If you submitted already, you must add your partner to your submission.
  - Please do this so we know which submission to grade.

- Midterm Exam
  - Thursday **This** Week
    - In AGH 106 and AGH 105 (the overflow room)
  - Midterm review in Class Today
  - Makeup exam is possible, email us ASAP

# Administrivia

- PENNOS
  - Spec to be released after the midterm (Friday/Saturday)
  - Done in groups of four, partner signup will open on Friday
    - Will do random assignment after break
  - First lecture after break will be a lecture given by TA’s to talk about PennOS

# Lecture Outline

- Some Brief History on "Operating Systems"
  - Human computers
  - Eniac
  - Punch cards
  - Unix & Linux
- "Modern" Scheduling

# Poll Everywhere

pollev.com/tqm

- Which topic do you want to practice and then have us go over?
  - Fork
  - Signals
  - Processes
  - Processes vs threads
  - File System
  - Scheduling
  - Threads & data races

# fork

- Consider the following C code  that uses fork()
  - Which of these outputs are possible? Please justify your answer
  - 380380
  - 338008
  - 380803

```c
int main() {
  pid_t pid = fork();
  pid = fork();

  if (pid == 0) {
    printf("3");
  } else {
    printf("8");
    int status;
    waitpid(pid, &status, 0);
    printf("0");
  }
}
```

# Signals: Critical Sections

- A vector is data structure that represents a resizable array. For those used to Java, think of it like an ArrayList.
- Consider the following C snippet that outlines what a vector of floats is and how we would push a value to the end of it. Is there a critical section in the vec_push function? If so, what line(s)?

```
typedef struct vec_st {
  size_t length, capacity;
  float* eles;
} Vector;

void vec_push(Vector* this, float to_push) {
  // assume that we don't have to resize for simplicity
  assert(this->length < this->capacity);
  this->length += 1;  // increment length to include it
  this->eles[this->length - 1] = to_push;  // add the ele to the end
}
```

# Signals Continued

- Signals can happen at any time and thus there are issues with making signal handlers safe to avoid any critical sections. In general, it is advised to keep signal handlers as short as possible or just avoid them at all costs.

- In each of these scenarios, tell us whether it is necessary to use signals and register a signal handler. If it is necessary, how safe is it?

- We want to have our program acknowledge when a user presses CTRL + Z or CTRL + C and print a message before exiting/stopping

# Signals Continued

- The user needs to type floating point numbers to stdin, but there are some special floating point numbers like NaN, infinity, and –infinity. To avoid this, we have the user hit CTRL + C for NaN, CTRL + Z for infinity and other key combinations for other special values.

# Processes

- We want to write a in C program that will compile and evaluate some other program. The program we are grading is similar to penn-shredder. For this program we write, lets assume we are running penn-shredder once and evaluating it. We need to be able to:
  - Specify the input and get output of the shredder
  - Set a time limit so that penn-shredder doesn’t go infinite
  - Setup penn-shredder to receive signals from the keyboard (e.g. CTRL + C and CTRL + Z)

- Roughly how many times do we need to call each of these system calls? Briefly explain any system call you specify non-zero for

# Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

| System Call | Number | Justification |
|---|---|---|
| fork() | | |
| execvp() | | |
| pipe() | | |
| waitpid() | | |
| kill() | | |
| signal() | | |
| tcsetpgrp() | | |

# Processes vs Threads

- Let’s say we had a program that did an expensive computation we wanted to parallelize, we could use either threads or processes. Which one would be faster and why?

- Sometimes we want to call software that is written in another language. If it is written as a library with the proper support (e.g. TensorFlow is in C++ but callable from Python), we could use threads. If we want to invoke a program that is already compiled (isn’t a library/doesn't have a callable interface) we could not use threads. We would have to use fork & exec. Why?

# Processes vs Threads

- We have seen two concurrency models so far
  - Forking processes (fork)
    - Creates a new process, but each process will have 1 thread inside it
  - Kernel Level Threads (pthread_create)
    - User level library, but each thread we create is known by the kernel
    - 1:1 threading model

# Processes vs Threads

- For each of the concurrency models, state whether it is possible to do each of the following.
- In a real exam, we would ask you to briefly explain why

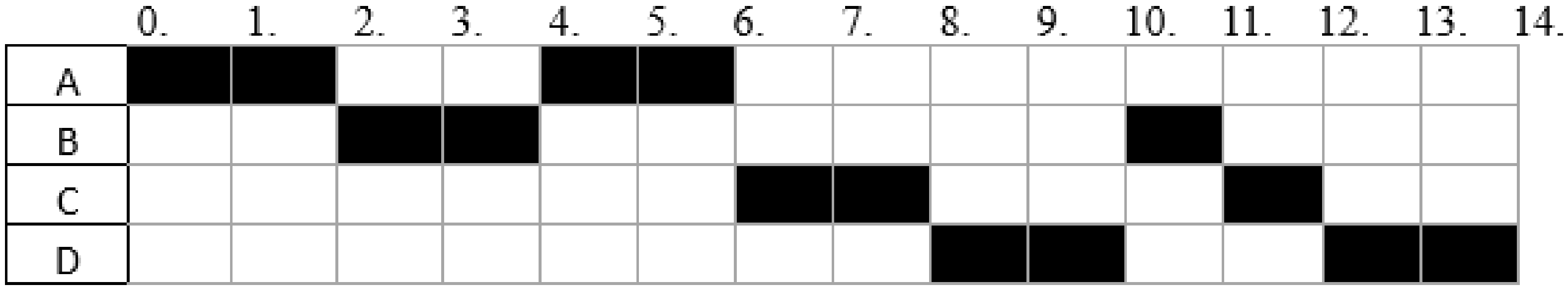
| | Processes | pthread |
|---|---|---|
| Can share files and concurrently access those files. | | |
| Can communicate through pipes | | |
| Run in parallel with one another (assuming multiple CPUs/Cores) | | |
| Modify and read the same data structure that is stored in the heap | | |
| Switch to another concurrent task when one makes a blocking system call. | | |

# Scheduling

- You manage the back-end servers for an online puzzle game. Players worldwide expect fast response times when navigating mazes in real time.
- Workload:
  - A large number of short, interactive tasks: e.g., responding to player movements and chat messages.
  - Occasional long-running background tasks (e.g., map generation, analytics).
- Constraints/Goals:
  - Fast response for interactive players to keep them engaged.
  - No single player (or background job) should monopolize the CPU.
- Which single scheduling algorithm or hybrid approach would you use, and how would it ensure both short interactive tasks and longer background processes get fair treatment? Consider context-switch overhead, time quantum size, and priority adjustments.

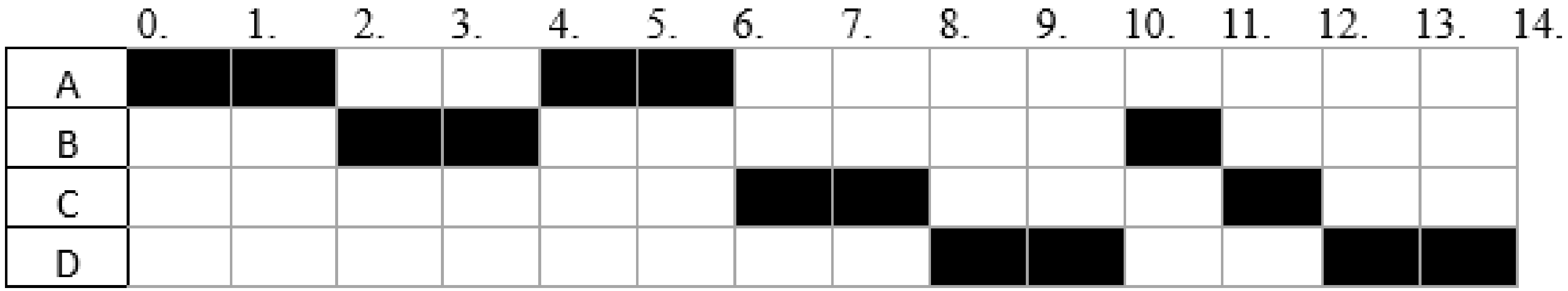
# Scheduling (cont.)

- Four processes are executing on one CPU following round robin scheduling:




- You can assume:
  - All processes do not block for I/O or any resource.
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

# Scheduling (cont.)




- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

- What is the earliest time that process C could have arrived?
- Which processes are in the ready queue at time 9?
- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?

# File System Block Allocation

- Consider that we want to read the 5th block of the file **`/home/me/script.txt`**, what is the worst-case number of physical blocks that must be read (including the 5th block) given the following :
  - Blocks are 4096 bytes
  - Each directory we are looking for is within the first block of the directory.
  - We are using **a Linked List Allocation (Implicit) file system**.
    - Assume we know the *physical block number* for the root directory.

# File System Block Allocation

- Consider that we want to read the 5th block of the file **`/home/me/script.txt`**, what is the worst-case number of physical blocks that must be read (including the $5^{th}$ block) given the following :
  - Each directory we are looking for is within the first block of the directory.
  - We are using a **Linked List Allocation via FAT**
    - Assume we know where the root directory starts in the FAT.
    - The FAT is only one block.
    - Assume we know the *physical block number* for the root directory.

# File System Block Allocation

- Consider that we want to read the 5th block of the file **`/home/me/script.txt`**, what is the worst-case number of physical blocks that must be read (including the 5th block) given the following :
  - assume that directory entries we are looking for are in the first block of each directory we search
- I-nodes
  - assume we know where the I Node for the root directory is

# File System Block Allocation

- Consider that we want to read the 5th block of the file `/home/me/script.txt`, what is the best-case number of physical blocks that must be read (including the 5th block) given the following :
  - assume that directory entries we are looking for are in the first block of each directory we search
- I-nodes
  - assume we know where the I Node for the root directory is

# File System Block Allocation

- How does the numbers change if we instead wanted to write to the 5th block of the file?

- Despite not having the best numbers, I nodes are still chosen over FAT. Why is this the case?

# I-Node Design

- Assume that blocks are 4,096 bytes and an inode is 128 bytes large.
- Inode numbers are uint32_t (that is, unsigned integers).
- How many blocks do we need in this file system configuration to create an inode table for each possible inode number? Feel free to write an expression, not a definite value. (It's a somewhat big value)

# Wait, where do we know how large the Inode Table is?

- The previous question alluded to the fact the number of inodes in the table is capped. Where would we need to look for to know how many total inodes there are?

# Fat Design

- Assume that blocks are 4,096 bytes.
- FAT numbers are 16 bits.
- How many blocks do we need in this file system configuration to create an fat table for each possible fat number? Feel free to write an expression, not a definite value.

# Has your friend been misled?

- You missed a super important lecture on filesystems and you think your friend has gone mad. They say that on a single disk (Hard Drive), you can have multiple different file systems! Is your friend correct? Why or why not?

# Has your friend been misled again?

- You missed yet another super important lecture on filesystems and you think your friend has gone mad (again!). They say that the operating system on your machine exists on the CPU and RAM prior to the first time turning on the machine. Have they been misled? Why or why not?

# Are we in the same directory?

- You're given two inodes, A and B. And you're tasked with writing a program that tells us whether or not they share a common directory.
  - ***(excluding the root directory).***
    - .e.g. **/usr/bin/echo** and **/usr/huh** share the **/usr/** directory
    - .e.g. **/usr/bin/echo** and **/dev/tty06** do not share a directory.
    - .e.g. **/usr/local/lib/stdio.h** and **/usr/local/lib/stdlib.h** share a directory, the **/usr/local/lib/** directory.
  - In other words, if somewhere up the path they share a directory, they have a common directory!
  - You are also given the Inode number for the directory that A and B are stored in.
  - Describe at a high level, what you would need to do to accomplish this. And what critical aspects of the file system structure would you need? Hint: *What about the structure of the directory blocks is imperative here?*

# Threads & Data Races

- Consider the following pseudocode that uses threads. Assume that file.txt is large file containing the contents of a book. Assume that there is a **main()** that creates one thread running **first_thread()** and one thread for **second_thread()**

- There is a data race. How do we fix it using just a mutex? (where do we add calls to lock and unlock?)

```
string data = "";   // global

void* first_thread(void* arg) {
  f = open("file.txt", O_RDONLY);
  while (!f.eof()) {
      string data_read = f.read(10 chars);
      data = data_read;
  }
}
void* second_thread(void* arg) {
  while (true) {
    if (data.size() != 0) {
      print(data);
    }
    data = "";
  }
}
```

# Threads & Data Races

- There is a data race. How do we fix it using just a mutex? (where do we add calls to lock and unlock?)

```
string data = "";   // global

void* first_thread(void* arg) {
  f = open("file.txt", O_RDONLY);
  while (!f.eof()) {
     string data_read = f.read(10 chars);
     data = data_read;
  }
}
void* second_thread(void* arg) {
  while (true) {
    if (data.size() != 0) {
      print(data);
    }
    data = "";
  }
}
```

# Threads & Data Races

- After we remove the data race on the global string, do we have deterministic output? (Assuming the contents of the file stays the same).

```
string data = "";  // global

void* first_thread(void* arg) {
  f = open("file.txt", O_RDONLY);
  while (!f.eof()) {
     string data_read = f.read(10 chars);
     data = data_read;
  }
}

void* second_thread(void* arg) {
  while (true) {
    if (data.size() != 0) {
      print(data);
    }
    data = "";
  }
}
```

# Threads & Data Races

- There is an issue of inefficient CPU utilization going on in this code. What is it and how can we fix it?
- (You can describe the fix at a high level, no need to write code)

```
string data = "";  // global

void* first_thread(void* arg) {
  f = open("file.txt", O_RDONLY);
  while (!f.eof()) {
     string data_read = f.read(10 chars);
     data = data_read;
  }
}

void* second_thread(void* arg) {
  while (true) {
    if (data.size() != 0) {
      print(data);
    }
    data = "";
  }
}
```

# That’s all!

- See you next time!