# Midterm Review

## Computer Operating Systems, Fall 2025


Instructors: Joel Ramirez

Head TAs: Maya Huizar Akash Kaukuntla
Vedansh Goenka Joy Liu

TAs:

Eric Zou Joseph Dattilo Aniket Ghorpade Shriya Sane
Zihao Zhou Eric Lee Shruti Agarwal Yemisi Jones
Connor Cummings Shreya Mukunthan Alexander Mehta Raymond Feng
Bo Sun Steven Chang Rania Souissi Rashi Agrawal
Sana Manesh

Poll Everywhere

pollev.com/cis5480

- Any questions?

# Administrivia

- Midterm is Thursday, 5:15PM – 6:45PM
  - Old exams and exam policies are posted on the course website
  - Review session today from 7-8PM in Towne 100.

- PennOS:
  - Specifications and team sign-up to be posted Friday (day after exam)
  - Done in groups of 4
  - Partner signup due by end of day on Monday, **10/20 (SIGN UP ON CANVAS!!)**
    - Those left unassigned will be randomly assigned the next morning (Tuesday the 21st)
  - Lecture dedicated to PennOS in class on Tuesday the 21st. ***Highly recommend you go.***

# Administrivia

- Midterm Review Session Tonight; 7PM – 8PM
  - Towne 100!
  - Go over an unreleased Midterm!
  - Go to get prepppppedddddd

Poll Everywhere

pollev.com/cis5480

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

# 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");
  }
}
```

# fork

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

```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");
  }
}
```

  - 380380
    - Possible, we have four processes from calling fork() twice. Pid is set from the second call to fork which is called from two different processes resulting in two new children.
    - To get 380380 we can imagine that there are two parent-child relationships created from the second call to fork. We can run one of those children and then its parent in that order. We can repeat this process for the second parent-child pair.

# fork

- Consider the following C code that uses fork()
  - Which of these outputs are possible? Please justify your answer
  - 338008
    - Impossible, we can't have an 8 last. Within a process it still executes in a specific order, a process cannot print 0 and then 8.

```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");
  }
}
```

# fork

- Consider the following C code that uses fork()
  - Which of these outputs are possible? Please justify your answer
  - 380803
    - Impossible, we can't have a 3 as last the parent waits on the child before it prints "0". This means the child waited on must finish before the parent prints "0", so "3" must come before an "0".

```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)?

```c
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: Critical Sections

- The last two lines could have an issue. Since they modify a piece of memory that could be shared, there may be an issue with signal handlers pushing onto the vector at the same time. Consider the case where we increment length and then a signal handler runs to push something onto the end of the vector.

```
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

- We want to have our program acknowledge when a user presses CTRL + Z or CTRL + C and print a message before exiting/stopping
  - **Probably need this to be done with signals to catch CTRL + Z and CTRL + C. Can't catch them otherwise**
  - **The printing may not be safe since we are modifying global state when we go through the file system to print**

# 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.

# 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.
  - **Do not do this. We can have the user input the floating point values in other ways without us having to resort to using signals to communicate this.**

# Processes

- We want to write a 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() | | |
| sigaction() | | |
| tcsetpgrp() | | |

# 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() | 2 | Need to fork compiler and penn-shredder |
| execvp() | 2 | To exec compiler and penn-shredder |
| pipe() | | |
| waitpid() | | |
| kill() | | |
| sigaction() | | |
| tcsetpgrp() | | |

# 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() | 2 | Need to fork compiler and penn-shredder |
| execvp() | 2 | To exec compiler and penn-shredder |
| pipe() | 2 | To send input and get output from penn-shredder |
| waitpid() | | |
| kill() | | |
| sigaction() | | |
| tcsetpgrp() | | |

# 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() | 2 | Need to fork compiler and penn-shredder |
| execvp() | 2 | To exec compiler and penn-shredder |
| pipe() | 2 | To send input and get output from penn-shredder |
| waitpid() | 2 | To wait for compiler and penn-shredder to terminate |
| kill() | | |
| sigaction() | | |
| tcsetpgrp() | | |

# Processes Cont.

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

Some of these can have varying answers. If this was an exam, as long as they are reasonable and justified well, we will accept it

| System Call | Number | Justification |
|---|---|---|
| fork() | 2 | Need to fork compiler and penn-shredder |
| execvp() | 2 | To exec compiler and penn-shredder |
| pipe() | 2 | To send input and get output from penn-shredder |
| waitpid() | 2 | To wait for compiler and penn-shredder to terminate |
| kill() | 1 | Debatable, can be justified if you used it. I use it to kill the child after timeout has occurred. Better than just using alarm in child since we can handle the timeout more elegantly and print out an error |
| sigaction() | | |
| tcsetpgrp() | | |

# Processes Cont.

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

Some of these can have varying answers. If this was an exam, as long as they are reasonable and justified well, we will accept it

| System Call | Number | Justification |
|---|---|---|
| fork() | 2 | Need to fork compiler and penn-shredder |
| execvp() | 2 | To exec compiler and penn-shredder |
| pipe() | 2 | To send input and get output from penn-shredder |
| waitpid() | 2 | To wait for compiler and penn-shredder to terminate |
| kill() | 1 | … (trimmed for space see previous slides) |
| sigaction() | 1 | Debatable again. Used to register SIGALRM for timeout. Could be avoided if we register alarm in child |
| tcsetpgrp() | | |

# Processes Cont.

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

Some of these can have varying answers. If this was an exam, as long as they are reasonable and justified well, we will accept it

| System Call | Number | Justification |
|---|---|---|
| fork() | 2 | Need to fork compiler and penn-shredder |
| execvp() | 2 | To exec compiler and penn-shredder |
| pipe() | 2 | To send input and get output from penn-shredder |
| waitpid() | 2 | To wait for compiler and penn-shredder to terminate |
| kill() | 1 | … (trimmed for space see previous slides) |
| sigaction() | 1 | … (trimmed for space see previous slides) |
| tcsetpgrp() | 1 | Debatable again. used so penn-shredder has control of terminal and so it will get the keyboard signals and our program won't. Could instead register the signals in our program and use kill in handler to send to child. |

# 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

- 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?

- Probably threads. Threads and processes are both parallelizable, but processes have a larger overhead since they have separate address spaces that need to be switched between.

# Processes vs Threads

- 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?
- Part of exec is that it replaces the entire address space with the program we want to run. The address space initial state is (mostly) specified by the program executable. If we tried to load in the program into just one thread, it would affect the memory space that is being shared with other threads

# 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. | | |

# 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. | Yes | Yes |
| 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. | | |

# 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. | Yes | Yes |
| Can communicate through pipes (can't redirect w/o affecting other threads though) | Yes | Yes |
| 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. | | |

# 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. | Yes | Yes |
| Can communicate through pipes | Yes | Yes |
| Run in parallel with one another (assuming multiple CPUs/Cores) | Yes | Yes |
| 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. | | |

# 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. | Yes | Yes |
| Can communicate through pipes | Yes | Yes |
| Run in parallel with one another (assuming multiple CPUs/Cores) | Yes | Yes |
| Modify and read the same data structure that is stored in the heap | No | Yes |
| Switch to another concurrent task when one makes a blocking system call. | | |

# 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. | Yes | Yes |
| Can communicate through pipes | Yes | Yes |
| Run in parallel with one another (assuming multiple CPUs/Cores) | Yes | Yes |
| Modify and read the same data structure that is stored in the heap | No | Yes |
| Switch to another concurrent task when one makes a blocking system call. | Yes | Yes |

# Scheduling

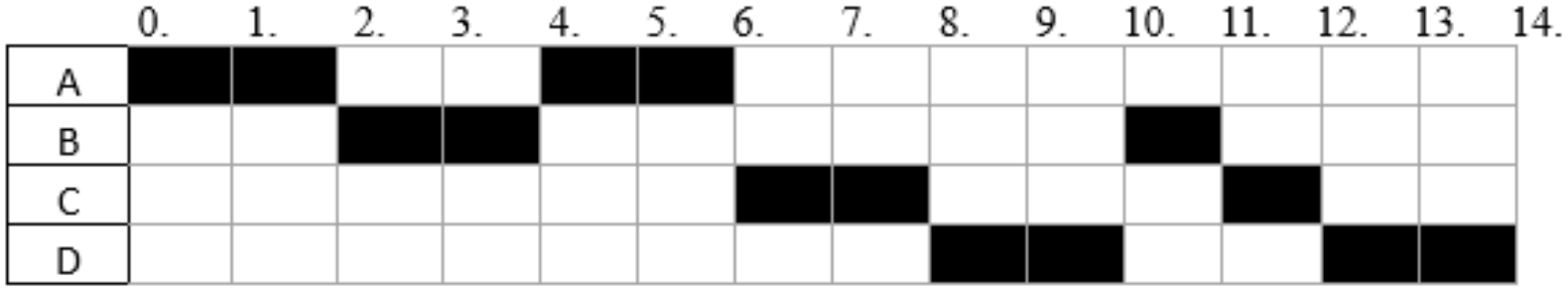
- You manage the back-end servers for an online puzzle game. Players worldwide expect fast response times when navigating mazes in real time.
- Workload:
  - A large number of short, interactive tasks: e.g., responding to player movements and chat messages.
  - Occasional long-running background tasks (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

- 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.
  - **One possible answer, (we would probably accept others or give most credit to others that are properly justified)**
  - **CFS is pretty good. CFS would ensure that both short interactive tasks and longer background tasks get fair treatment since CFS tries to balance it so that all tasks get roughly equal utilization of the CPU by always running the task that is ready to run and has run on the CPU the least.**
  - **The short interactive tasks that interact with users get high priority since they will have low total CPU usage, but when they are done background tasks will still get to run as needed. Priority can be included via nice scores.**

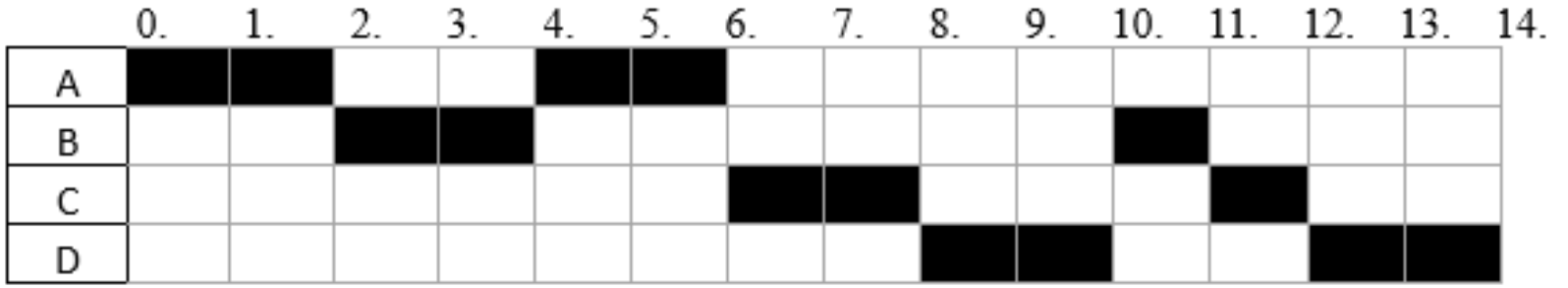
# Scheduling (cont.)

- 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.

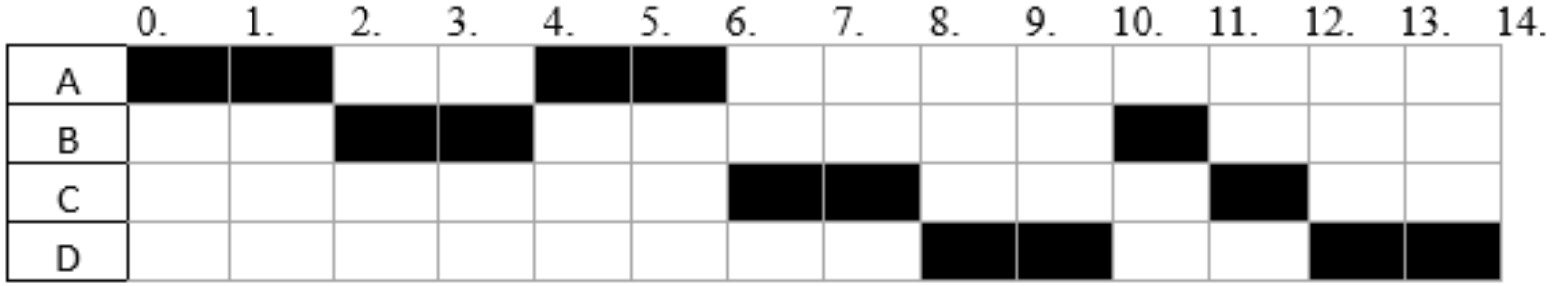
# Scheduling (cont.)




- 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?

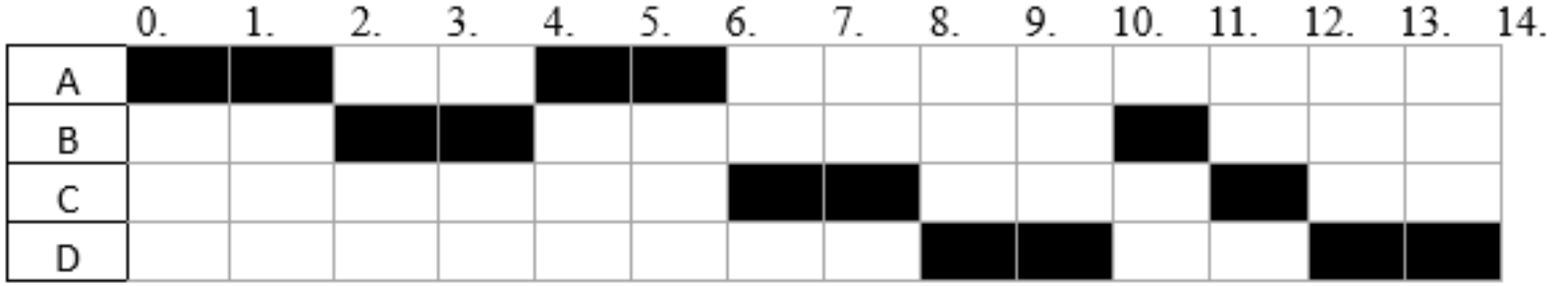
# Scheduling (cont.)




- 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?
  - If C arrived at time 0, 1, or 2, it would have run at time 4
  - C could have shown up at time 3 and come after A in the queue
  - C showed up at time 3 at earliest

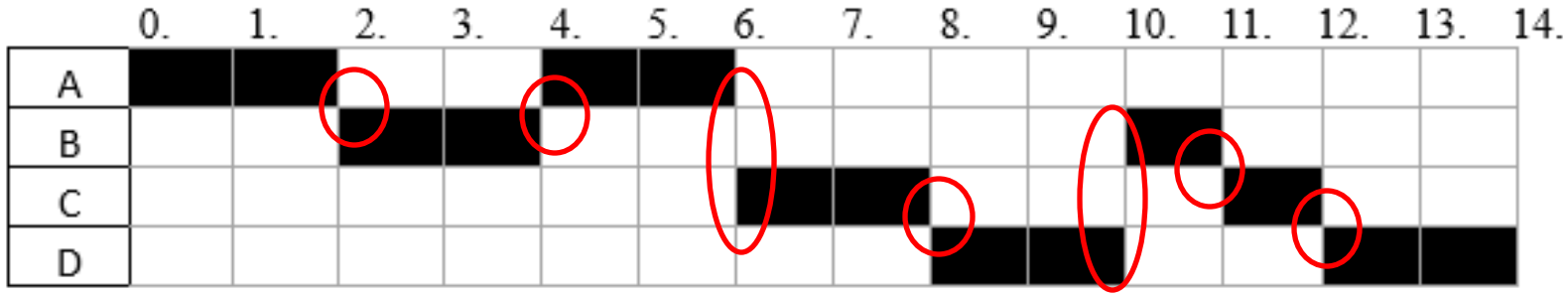
# Scheduling (cont.)




- 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.

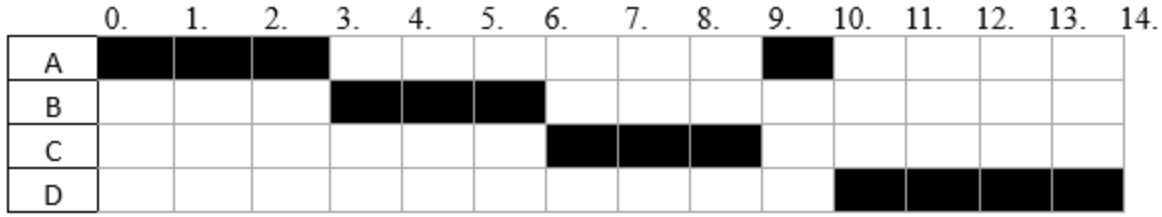
- Which processes are in the ready queue at time 9?
  - D is running, so it is not in the queue
  - A has finished
  - B and C still have to finish, so they are in the queue.

# Scheduling (cont.)

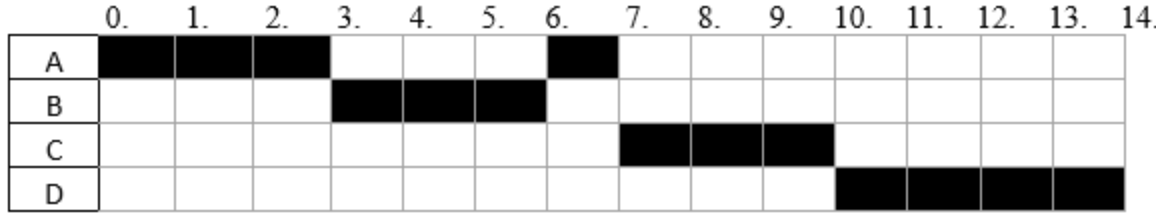



- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:

Depends on if C shows up at time 3 or 4




  - Or:




Either way, only 4 context switches, so 3 less than quantum = 2

# 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 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.

1. Need to read the block for the root directory. + 1
2. Need to then read the block for the 'home' directory. + 1
3. Need to then read the block for the 'me' directory. + 1
   a) We know the physical block number, A, for the script.txt file now.
4. We follow the implicit linked list starting from the physical block number A . This requires 5 reads to reach the 5th block of the file.

You can assume directories are just arrays of `struct dirents` in all designs unless told other wise.

In total, 8 blocks!

# 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 :
  - 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 :
  - 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.

1. Need to read the block for the root directory. + 1
2. Need to then read the block for the 'home' directory. + 1
3. Need to then read the block for the 'me' directory. + 1
   a) We know the physical block number, A, for the script.txt file now.
4. We can find the 5th block number by traversing the FAT; requires 1 read of the entire FAT
   ***a) (Doesn't count, already in memory (RAM) BIG PART OF FAT!!!! IMPERITIVE!!!!!***
5. Finally, we access the 5th block.

4 blocks read!

# 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

- I-nodes
  - assume we know where the I Node for the root directory is
  - 1 read to read in the inode of the root directory (This is a portion of the inode table)
  - 1 read for the directory entry of home/ inside of /
  - 1 read for the inode for /home/
  - 1 read for the directory entry of me/ inside of /home/
  - 1 read for the inode for /home/me/
  - 1 read for the directory entry of script.txt inside of /home/me/
  - 1 read for the inode of script.txt
  - 1 read to get and read the 5$^{th}$ block of the file
  - 8 block reads

Worst case scenario: Each inode we need is in a separate block within the Inode table.

# 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
  - Assume 32 inodes in a block

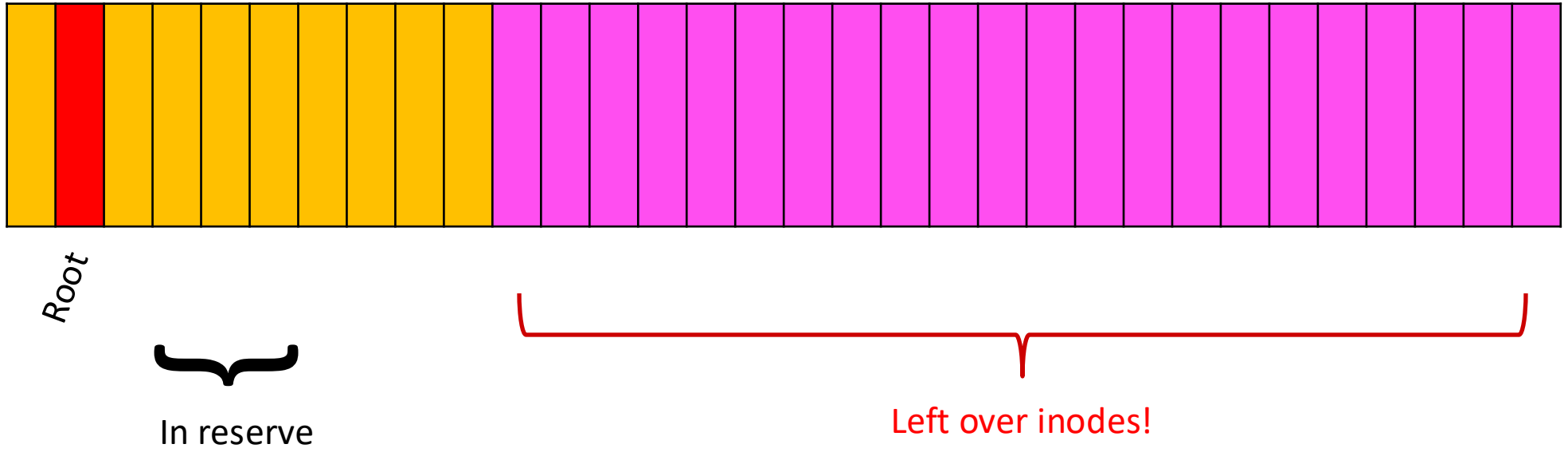
# File System Block Allocation

- I-nodes
  - assume we know where the I Node for the root directory is
  - 1 read to read in the inode of the root directory (This is a portion of the inode table)
  - 1 read for the directory entry of home/ inside of /
  - 0 reads for the inode for /home/ (In same block as root inode)
  - 1 read for the directory entry of me/ inside of /home/
  - 0 read for the inode for /home/me/ (In same block as root inode)
  - 1 read for the directory entry of script.txt inside of /home/me/
  - 0 read for the inode of script.txt (In same block as root inode)
  - 1 read to get and read the 5$^{th}$ block of the file
  - 5 disk reads

Best case scenario: Each inode we need is in the same block as the root inode. *(Totally possible with larger block sizes)*

# File System Block Allocation

All one block.




In a 4096 Block Size FS with Inodes from ext2 (128 bytes), you can fit 32 inodes in one block.

If we're following the ext2 design, the first 10 inodes are in reserve;
leaves 22 inodes for /home, /home/me, & /home/me/script.txt (totally feasible)

Best case scenario: Each inode we need is in the same block as the root inode. *(Totally possible with larger block sizes)*

# 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?

# File System Block Allocation

- How does the numbers change if we instead wanted to write to the 5th block of the file?
  - Nothing changes, we would just write to the 5th block of the file instead of reading it.

- Despite not having the best numbers, I nodes are still chosen over FAT. Why might this the case?
  - FAT takes up a lot of memory because we are storing the state of the entire filesystem in memory when we load the FAT into RAM.
  - Inodes allow us to instead cache the information for relevant files in memory, so much lower memory consumption and similar performance for the most case.
    - Additionally, they not only store data about the data blocks of a file but also other metadata.

# 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)

# 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)

Number of possible inodes: **2^32 (this is number of distinct values UINT32_T can take on)**
Each inode is **128 bytes**, so total storage required for all inodes is: **2^32 * 128 = 2^39 bytes**
Since each block is 4,096 (2^12) bytes, the total number of blocks needed is: (2^39) / (2^12) = 2^27 blocks…

This is only about 550 gigabytes! (Totally feasible on most large file systems like the 1TB on my machine)
However, you usually don't just create the table to have the maximum number of inodes as a computer usually will not need 4294967296 different files at a time.

# 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?

# 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?

**The Superblock** – The superblock contains ***critical*** metadata, including the total number of inodes, total number of data blocks, inodes in usage, and etc. Every filesystem has a superblock, and it is usually the block right before the inode table (in Linux and UNIX).

# 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.

# 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.

32 blocks….

**Number of FAT entries possible**: 2^16

**Number of bytes for all FAT entires: 2^16 * 2 = 2^17**

Since each block is 4096 (2^12) the **total number of blocks** required is: 2^17/2^12 = 2^5 = 32…

so small!

# 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?

- 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?

Totally possible, the ***boot block (or sector)*** contains information about the different partitions on a disk.

Each partition can contain any file system. Of course, as long as it is configured correctly. On more antiquated boot blocks (Master Boot Records) you can have at most 4 partitions (or 3 partitions and an extended)

# 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?

# 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?

Your friend has misunderstood the lecture completely! (They must have slept through it)

Before you turn on your computer, the operating system (OS) isn't hanging out in RAM or the CPU. It's just sitting there on your SSD, HDD, or whatever non-volatile storage you've got. The OS doesn't actually do anything until you power on the machine, at which point it gets bootstrapped into RAM and—boom! You're running DOOM.

# 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?*

# Are we in the same directory? (A Hard Question)

- 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).***
  - This is a common example of where a algorithms problem, Common Ancestor in a Tree, applies to file systems!
  - Starting from the two inodes, A and B, you would need to use the '..' reference to the parent directories to travel up the path. The '..' entry of a directory will give us the inode number for the parent directory and we can go and grab that directories data blocks by going in reverse.
  - For both Inodes A and B, we would need to keep track of the inode numbers as we travel up the file system. If at any point we see that they share a common Inode Number, then they must share a directory.
  - The ".." entry within every directory block is what is absolutely necessary to make this possible. If not, we wouldn't be able to "backtrack" up the file system.

# Threads & Data Races

- Consider the following pseudocode that uses threads. Assume that file.txt is large file containing a multiple of 10 characters. Assume that there is a **main()** that creates one thread running **first_thread()** and one thread for **second_thread()**. Assume that accesses to the string data are done safely by the threads.

```
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 = "";
  }
}
```

- Do we have deterministic output? If so how? If not, what are the min and max number of characters printed

# Threads & Data Races

- Do we have deterministic output?
  - No, we could still have a difference in output depending on when threads are run. It is possible a the first thread overwrites the global before second thread reads it
  - Min: 0 characters are printed
  - Max: everything works out fine and all characters in the book are printed

```
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

- 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()**

```
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 = "";
  }
}
```

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

# 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

- 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
pthread_mutex_t mutex;

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

# 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
pthread_mutex_t mutex;

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

# 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

- After we remove the data race on the global string, do we have deterministic output? (Assuming the contents of the file stays the same).
  - No, we could still have a difference in output depending on when threads are run. It is possible a the first thread overwrites the global before second thread reads it

    This is the distinction between a data race and a race condition

```
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? (Not on exam)
- (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 = "";
  }
}
```

# 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)
  - Busy waiting possible in second_thread. We could have the threads use a condition variable to wait for data to be updated and thread1 to signal thread2 once ready

```
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!