

Course Wrapup

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	

pollev.com/tqm

- ❖ What did you learn in this course? Is there anything you wish we talked about more? Anything you wish we talked about less?

Administrivia: Final Exam & End of Semester

- ❖ Final Exam: Monday May 5th from 9am to 11am
 - Final Exam Policies posted on course website
 - Old exams & exam questions
- ❖ TA-led Final Exam review on Saturday the 3rd from 6pm to 8pm. Towne 217
- ❖ End of Semester Survey: Due Sunday May 4th
 - Graded on completion.
- ❖ PennOS Peer Evaluation Survey: Due Sunday May 4th
 - Only submit after your PennOS Demo. Each groupmate submits individually and privately
 - You get a little PennOS Extracredit for completing the survey

Administrivia: PennOS

❖ PennOS

- PennOS Final Demo posted
 - You will have to demo FAT here even if you pass the autograder.
- Integration can be a big pain, make sure you allocate enough time to it!
- Can use 1 late token for free now to submit by EOD Sunday
- We will ask you short answer questions during the demo to check that you actually understand your code.
 - You will be able to choose the category of question.
 - More details on Ed

Administrivia: PennOS

❖ Some notes:

- **NEW**: If you are having issues with the scheduler sometimes not suspending a thread: make sure that you do NOT have interrupts enabled when you call `spthread_create`.
 - Or, you can redownload `spthread.c` from the course website
 - You do not HAVE to do either of these, though this will almost certainly cause issues.
- DO NOT mmap the entire File System. Only mmap the Allocation Table, the rest of the file system needs to be handled with `lseek/write`.
 - Do not keep the contents of the file in memory, it should be stored in the file
 - If your PennFat is killed with `kill -9`, your file contents should still be saved in disk
- Advice for using gdb to debug
 - **Handle SIGUSR1 noprint nostop**
Makes it so that gdb doesn't report every time SIGUSR1 goes and interrupts you

Administrivia: PennOS

❖ Some notes:

- Reminder, you instead of just doing:

```
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);
```

you may need to do:

```
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);  
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);
```

- With the description of `setitimer()`, it just says that sigalarm is delivered to the process, not necessarily the calling thread. To make sure siglaram goes to the scheduler, you may want to make it so that all threads (spthread or otherwise) that aren't the scheduler call something like: **`pthread_sigmask(SIG_BLOCK, SIGALARM)`**
 - Which will block SIGALARM in that thread.

Administrivia: PennOS


- ❖ If you are having issues with the scheduler not running you can try running
 - `strace -e 'trace=!all' ./bin/pennos`
 - You may have to install strace: `sudo apt install strace`
 - This will print out every time a signal is sent to your pennos
 - (Usual fix is the pthread_sigmask thing on the previous slide)

Lecture Outline

- ❖ **Course Wrap-up**
- ❖ Exam Review



What have we been up to for the last 14 weeks?

- Ideally, you would have “learned” everything in this course, but we’ll use red stars  today to highlight the ideas that we hope stick with you beyond this course

Operating Systems: The Why

- ❖ The programming skills, engineering discipline, and knowledge you need to build a system
 - 1) Understanding the “layer below” makes you a better programmer at the layer above
 - 2) Gain experience with working with and designing more complex “systems”
 - 3) Learning how to handle the unique challenges of low-level programming allows you to work directly with the countless “systems” that take advantage of it

So What is a System?

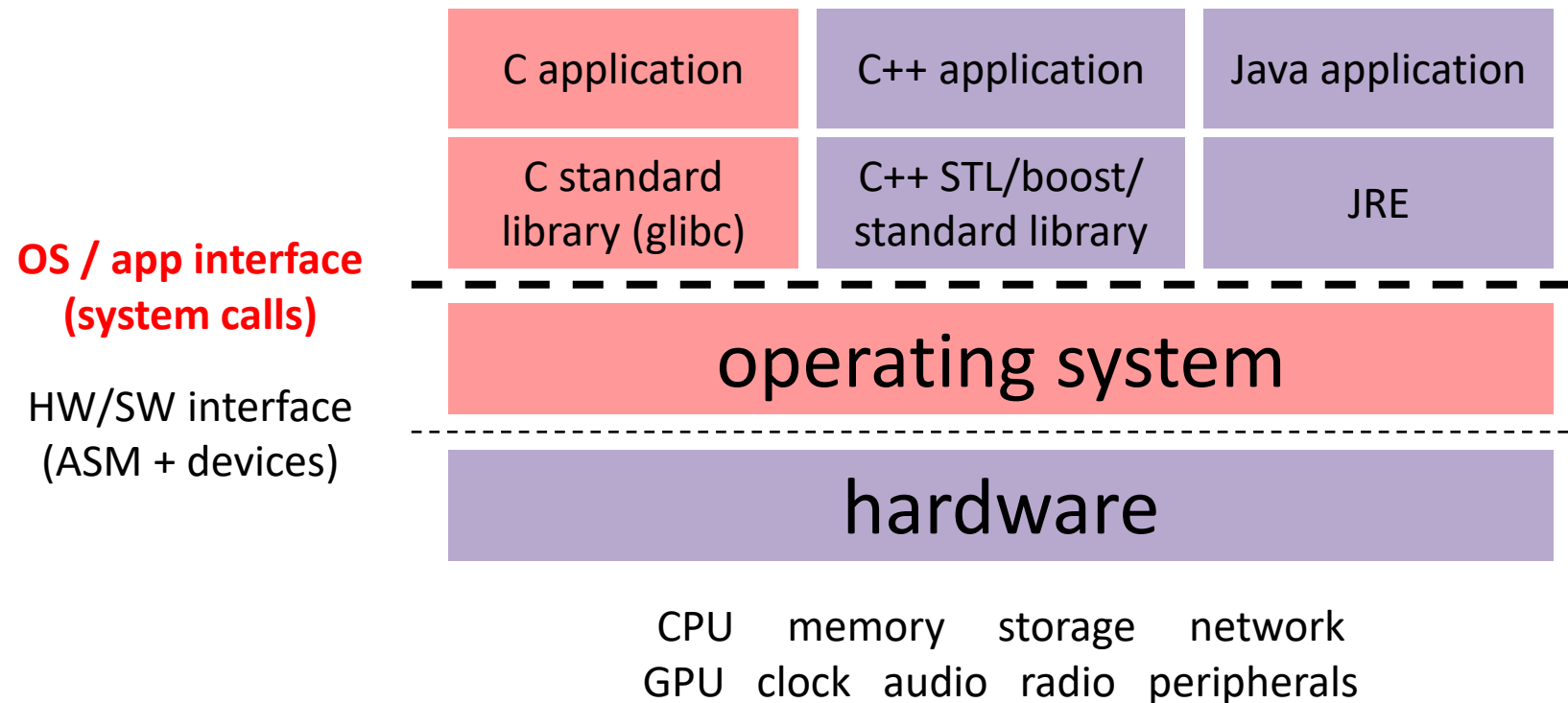
- ❖ “A **system** is a group of interacting or interrelated entities that form a unified **whole**. A system is delineated by its spatial and temporal boundaries, surrounded and influenced by its environment, **described by its structure and purpose and expressed in its functioning**.”
 - <https://en.wikipedia.org/wiki/System>
 - Still vague, maybe still confusing
- ❖ But hopefully you have a better idea of what a system in CS is now
 - What kinds of systems have we seen...?

Software System

- ❖ Writing complex software systems is *difficult*!
 - Modularization and encapsulation of code
 - ★ ■ Resource management
 - Documentation and specification are critical
 - ★ ■ Robustness and error handling
 - Must be user-friendly and maintained (not write-once, read-never)
- ★ **Discipline:** cultivate good habits, encourage clean code
 - Coding style conventions
 - Unit testing, code coverage testing, regression testing
 - Documentation (code comments, design docs)

The Computer as a System

- ❖ Modern computer systems are increasingly complex!
 - threads, processes, pipes, files
 - Buffered vs. unbuffered I/O, blocking calls, caches, virtual memory



Systems Programming: The What

- ❖ The programming skills, engineering discipline, and knowledge you need to build a system

- ★ **Programming:** C (& other languages)

- **Discipline:** design, testing, debugging, performance analysis

- **Knowledge:** long list of interesting topics

- Concurrency, OS interfaces and semantics, techniques for consistent data management, ...

- ★ Most important: a deep understanding of the “layer below”

Main Topics

- ❖ C
 - Low-level programming language
- ❖ Memory management & allocation
- ❖ System interfaces and services
- ❖ Concurrency basics – POSIX threads, synchronization
- ❖ Multi-processing Basics – Fork, Pipe, Exec
- ❖ Buffering, Caches, Locality
- ❖ Operating System Internals
 - File systems
 - Scheduling
 - Virtual Memory

Topic Theme: Abstraction

- ❖ C: **void*** to abstract away types for some functions (pthread_create, read, write, etc).
- ✧ abstract away details of interacting with system resources via system call interface (e.g. file descriptors and pids)
- ✧ The concept of processes and virtual memory to abstract away sharing hardware
- ✧ Read Write Locks and monitors abstract away their implementation of using a mutex & condition variable
- ❖ Nice abstractions minimize cognitive complexity and make it harder for users of the abstraction to fuck up.

Topic Theme: Data & Locality

★ I/O to send and receive data from outside of your program (*e.g.*, disk/files, network, streams)

- Linux/POSIX treats all I/O similarly

★ Takes a LONG time relative to other operations

- Blocking vs. non-blocking (and the sin that is spinning)

❖ C: Memory model (Stack vs Heap)

★ Buffers can be used to temporarily hold data

- Buffering can be used to reduce costly I/O accesses, depending on access pattern

★ Caching & Locality

- Some memory is quicker to access than others
- Hardware makes assumptions on your program's access patterns

Topic Theme: Allocating Resources

- ❖ It is often the tasks of a system to distribute/allocate a finite number of resources:
 - Scheduling algorithms allocate which threads can utilize the CPU
 - Memory allocation schemes (slab allocator, buddy algorithm)
 - Virtual Memory: allocating pages in physical memory
 - Caches: deciding what memory is in the cache.
 - File System: Allocating Blocks in file system
- ❖ These allocation schemes need to consider:
 - Efficient utilization of the resource that is being allocated
 - Fragmentation, fairness, minimize times we go to slower storage
 - Minimal overhead in the allocation scheme.
 - Time spent on the allocation is time not spent doing other things

Topic Theme: Concurrency

Processes

- Exec
- Process Groups
 - Terminal Control
- IPC
 - Pipe
 - Signals

Threads

Synchronization

- mutex
- Condition variables
- Deadlock

Concurrency vs parallelism

MISSING Topic Theme: Society

- ❖ One flaw (among others) of this course is how we don't talk ENOUGH about how this relates to the rest of the world
 - These systems we build do not have to necessarily be “evil”, but can often be used in those ways
 - We need to work and communicate with other people, even in CS.

- ❖ Actions:
 - Take Algorithmic Justice (CIS 7000) with Danaë Metaxa
 - Take Software Engineering (CIS 3500)
 - Join a community of people working on things that matter to you, (Unions or other organizations)
 - Join as a TA for 2400 or 54800 next year. We are trying to further integrate ethics.

Congratulations!

- ❖ Look how much we learned!
- ❖ Lots of effort and work, but lots of useful takeaways:
 - Debugging practice
 - Reading documentation
 - Tools (gdb, valgrind)
 - C familiarity
 - Concurrent Programming
 - Designing large systems
 - Working with others
- ❖ Go forth and build cool systems!

Future Courses

❖ Systems Courses

- CIS 5050: Software Systems
- CIS 5530: Networked Systems
- CIS 5521: Compilers
- CIS 5550: Internet and Web Systems
- CIS 5500: Database and Information Systems
- CIS 5470: Software Analysis

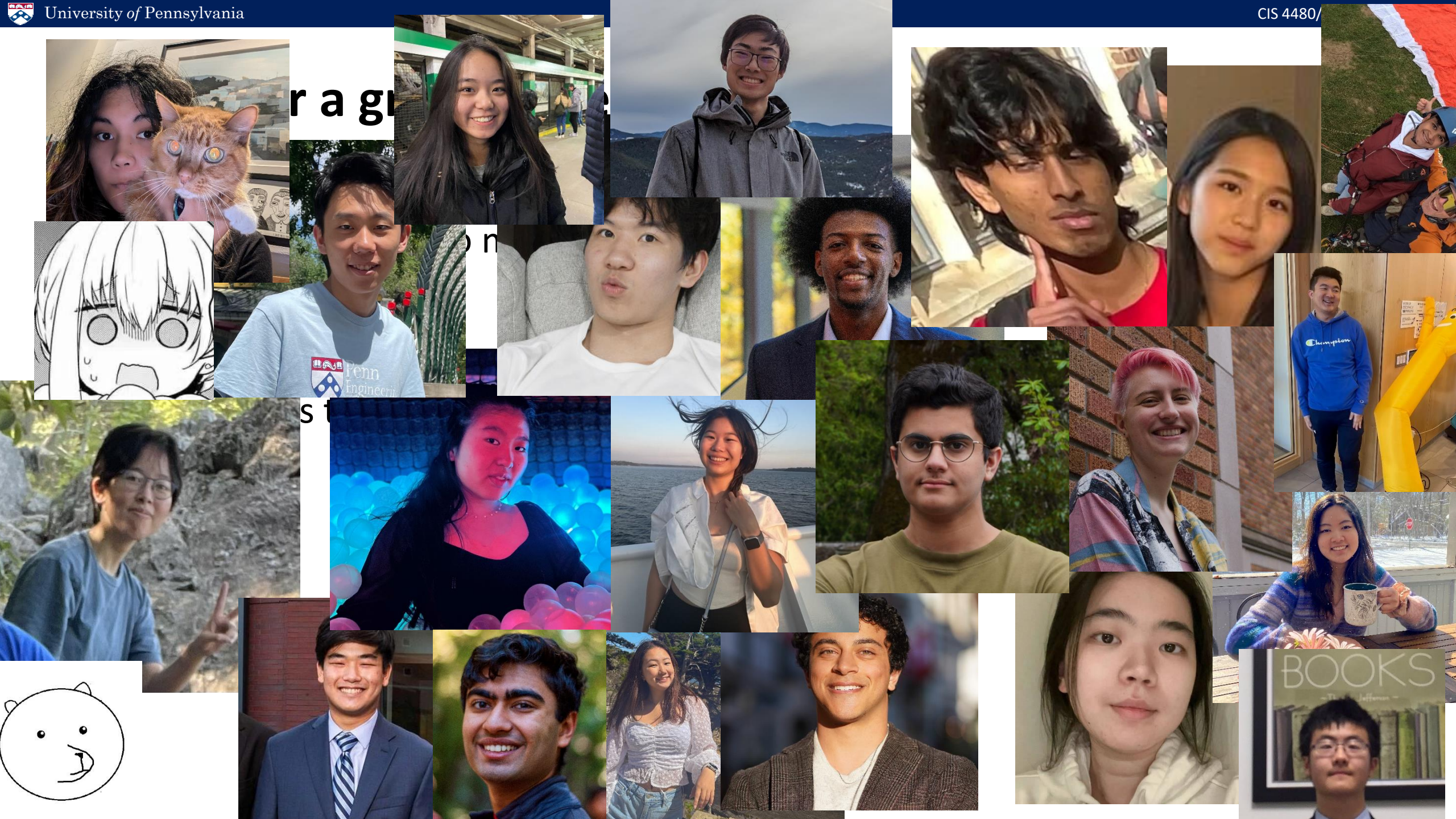
❖ Otherwise related courses

- CIS 5600 Interactive Computer Graphics
- CIS 5650 GPU Programming and Architecture
- CIS 5510 Security

Thanks for a great semester!

- ❖ Special thanks to all the instructors before me (Both at UPenn and UW) who have influenced me to make the course what it is
- ❖ Huge thanks to the course TA's for helping with the course!





Thanks for a great semester!

❖ Thanks to you!

- It has been another tough semester. Look at the state of Society 😊
- Things are still a bit rough in the course as we change it.
 - Joel's first time teaching this course, Travis' 3rd
- You've made it through so far, be proud that you've made it and what you've accomplished!

❖ Please take care of yourselves, your friends, and your community

Lecture Outline

- ❖ Course Wrap-up
- ❖ **Exam Review**

Disclaimer

- ❖ **THIS REVIEW IS NOT EXHAUSTIVE**
- ❖ **Topics not in this review are still testable**
- ❖ **Exam Review tentatively during reading days. Saturday 6pm – 8pm (tentative)**

Practice Problems

- ❖ Processes vs Threads
- ❖ Signal Handlers
- ❖ Memory Allocation
- ❖ Caches
- ❖ Scheduling (Same as extra practice at end of scheduling lecture)
- ❖ File System
- ❖ Virtual Memory
- ❖ Threads & Data Races
- ❖ Deadlock

Processes vs Threads

- ❖ Let's say we had a program that did an expensive computation (like summing a 1,000,000 element array) that we wanted to parallelize, we could use either threads or processes. Which one would be faster and why?

Processes vs Threads

- ❖ Let's say we had a program that did an expensive computation (like summing a 1,000,000 element array) that 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.
- ❖ Additionally, if we were using processes, how they would synchronize their sums would become a more involved issue.

Threads and Exec

- ❖ You spawn 10 threads and assign to each a random function to execute. Some seem harmless and others not so much.
- ❖ Specifically, one of the random functions they can call is the following.

```
int random_func_a(){  
    char *argv[] = {"sleep", "0", NULL};  
    execvp(argv[0], argv);  
}
```

What happens if one of the threads is assigned this function and runs it?

Threads and Exec

- ❖ You spawn 10 threads and assign to each a random function to execute. Some seem harmless and others not so much.
- ❖ Specifically, one of the random functions they can call is the following.

```
int random_func_a(){  
    char *argv[] = {"sleep", "0", NULL};  
    execvp(argv[0], argv);  
}
```

If a thread runs exec, the entire process is scrapped and thus, so are the other threads. It's all gone. Tada.

Processes vs Threads

- ❖ Let's say you've written a program that runs really well and does everything you need to, except that once every day it crashes. Fortunately for you, it's not doing anything critical - but it's not worth the development time to find and fix the cause of the crash.
- ❖ You decide to write a program that checks the status of another program and restarts it if it crashes. You are deciding whether your two programs (the one that crashes and the one that restarts) should be two threads in the same process or in two separate processes.
- ❖ Which do you choose? Briefly explain your answer

Processes vs Threads

- ❖ Let's say you've written a program that runs really well and does everything you need to, except that once every day it crashes. Fortunately for you, it's not doing anything critical - but it's not worth the development time to find and fix the cause of the crash.
- ❖ You decide to write a program that checks the status of another program and restarts it if it crashes. You are deciding whether your two programs (the one that crashes and the one that restarts) should be two threads in the same process or in two separate processes.
- ❖ Which do you choose? Briefly explain your answer
- ❖ You need two separate processes because otherwise the two threads share a memory space and if one crashes they both will crash. If we have two processes there is some isolation and thus the program that “restarts” the failing program can keep running when the failing program

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 three concurrency models, state whether it is possible to do each of the following.
- ❖ In real exam, I 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 three concurrency models, state whether it is possible to do each of the following.
- ❖ In real exam, I 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 three concurrency models, state whether it is possible to do each of the following.
- ❖ In real exam, I 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 three concurrency models, state whether it is possible to do each of the following.
- ❖ In real exam, I 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 three concurrency models, state whether it is possible to do each of the following.
- ❖ In real exam, I 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 three concurrency models, state whether it is possible to do each of the following.
- ❖ In real exam, I 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

Kernal Signal Handlers

- ❖ You're a TA in OS and you're overseeing a group. You notice they wrote functionality in their signal handlers (`:/`). PCBs are updated within the handler and also within their waitpid implementation. They leave all signals unblocked.

```
void update_pcb(ksignal __signal){  
    //check for child updates  
    //update pcb as necessary  
}
```

This is *exactly what the function does*.

It does nothing other than check for updates and update the PCB.

They tell you that sometimes the PCB updates correctly, but other times it becomes corrupted.

What could explain this behavior?

Kernal Signal Handlers

- ❖ You're a TA in OS and you're overseeing a group. You notice they wrote functionality in their signal handlers (`:/`). PCBs are updated within the handler and also within their waitpid implementation. They leave all signals unblocked.

```
void update_pcb(ksignal __signal){  
    //check for child updates  
    //update pcb as necessary  
}
```

This is *exactly what the function does*.

It does nothing other than check for updates and update the PCB.

They tell you that sometimes the PCB updates correctly, but other times it becomes corrupted.

What could explain this behavior?

If a section of your code must run to completion without being interrupted (meaning the current thread or process must finish executing it without being paused), you should disable interrupts to prevent preemption by another thread or interruption by a signal handler. In this case, there's nothing stopping this code from being interrupted mid signal handler by something else or even within their waitpid implementation. Usually, all handler does is update a flag to indicate they need to update later, rather than doing the update within the handler. Gotta be fast to respond.

Memory Allocation

- ❖ Some memory allocators (like the internal memory allocator for the Linux kernel) allow for some options to be specified that can change the behavior of these allocators. For each of these can you explain why this feature may be useful to have as an option?
 - If there is no memory available, then the allocation call may wait for some memory to be freed up so that it can eventually succeed
 - If memory is not able to be immediately allocated, give up at once. Caller can retry later if they desire.
- ❖ Some allocators enforce a minimum size for each allocation. If you request less than the minimum size it is rounded up.
- ❖ Why may an allocator do this?
- ❖ What is a downside to doing this?

Memory Allocation

- ❖ Some memory allocators (like the internal memory allocator for the Linux kernel) allow for some options to be specified that can change the behavior of these allocators. For each of these can you explain why this feature may be useful to have as an option?
 - If there is no memory available, then the allocation call may wait for some memory to be freed up so that it can eventually succeed
 - In a multi-threaded environment we can try to avoid a catastrophic out of memory issue by just waiting till another thread releases memory. In the context of allocating kernel memory, this could also just be memory allocated to some other process's task that then gets deallocated at some point.
 - If memory is not able to be immediately allocated, give up at once. Caller can retry later if they desire.
 - Meeting tight timing requirements since doing memory allocation may take a while (especially if the heap needs to grow or a new page added to the virtual memory space). Try again later once more space is easily accessible.

Memory Allocation

- ❖ Some allocators enforce a minimum size for each allocation. If you request less than the minimum size it is rounded up.
- ❖ Why may an allocator do this?
 - For things like the buddy allocator, they do this since the larger allocation size (1 page) is core to how the system is designed. It makes it easier for the “math” to work out so that their allocation scheme is faster/easier to implement
 - Malloc may do something like this with a size of 8 as the minimum size to help make sure all allocations start at a multiple of 8 and/or to minimize external fragmentation.
- ❖ What is a downside to doing this?
 - Increased internal fragmentation, an allocation now takes up more space than it actually needs.

Memory Allocation

- ❖ Assume we have the following two pieces of code, which ones is likely faster than the other and why?

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int* arr = malloc(sizeof(int) * 10);
    arr[0] = 1;
    arr[1] = 1;
    for(int i = 2; i < 10; i++) {
        arr[i] = arr[i-1] + arr[1-2];
    }

    printf("%d\n", arr[9]);
    free(arr);
}
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int arr[10];
    arr[0] = 1;
    arr[1] = 1;
    for (int i = 2; i < 10; i++) {
        arr[i] = arr[i-1] + arr[1-2];
    }

    printf("%d\n", arr[9]);
    free(arr);
}
```

Memory Allocation

- ❖ Assume we have the following two pieces of code, which ones is likely faster than the other and why?

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int* arr = malloc(sizeof(int) * 10);
    arr[0] = 1;
    arr[1] = 1;
    for(int i = 2; i < 10; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }

    printf("%d\n", arr[9]);
    free(arr);
}
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int arr[10];
    arr[0] = 1;
    arr[1] = 1;
    for (int i = 2; i < 10; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }

    printf("%d\n", arr[9]);
}
```

Likely the one on the right. Instead of calling malloc, the array is a static size on the stack. The stack allocation is quicker to allocate and free.

Memory Allocation

- ❖ Lets say that in addition to malloc, we also had a custom slab allocator implemented that could allocate chunks of space that is 64 bytes (16 integers) large.
- ❖ What is one reason we may prefer the custom slab allocator to malloc?
- ❖ What is one reason we may prefer malloc?

Memory Allocation

- ❖ How is the array in this snippet of code likely allocated at a low level (in assembly)?

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int arr[10];
    arr[0] = 1;
    arr[1] = 1;
    for (int i = 2; i < 10; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }

    printf("%d\n", arr[9]);
}
```

Memory Allocation

- ❖ How is the array in this snippet of code likely allocated at a low level (in assembly)?

Just need to decrement the stack pointer by $10 * \text{sizeof}(\text{int})$ and there is enough space to store the array on the stack now :P

Would also accept more vague answers like (grow the stack by 10 integers)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int arr[10];
    arr[0] = 1;
    arr[1] = 1;
    for (int i = 2; i < 10; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }

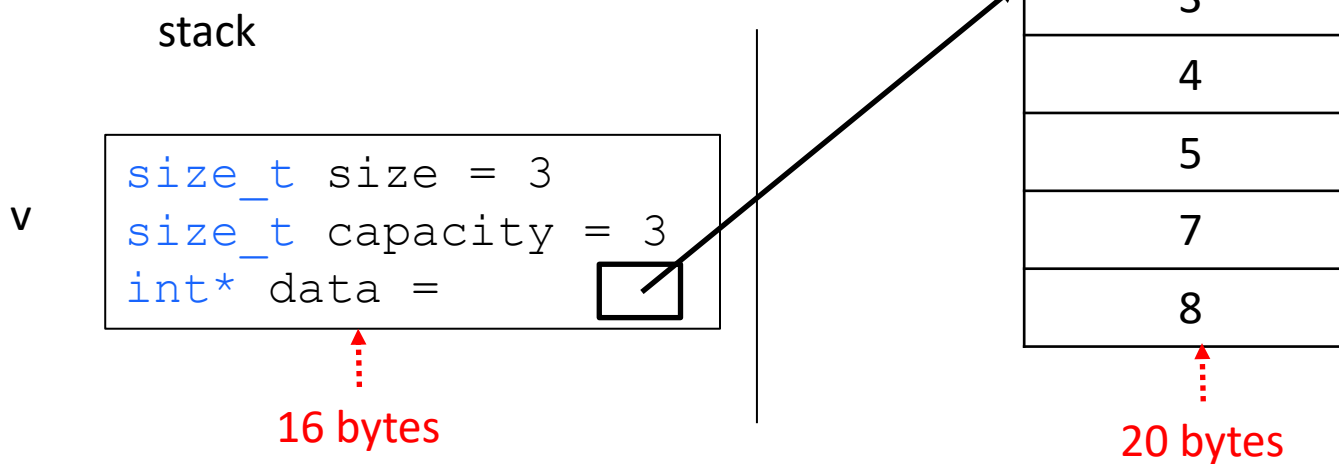
    printf("%d\n", arr[9]);
}
```

Caches

- ❖ The most common way to store a sequence of elements in C++ and most languages is a dynamically resizable array (e.g. a vector).

A vector of `<int>` looks something like this in memory:

```
int main(int argc, char** argv) {  
    vector<int> v {3, 4, 5, 7, 8};  
}
```



Caches

- ❖ Typically, a `bool` variable is 1 byte. How much space does a `bool` strictly *need* though?
 - 1 bit

- ❖ C++ goes against the standard implementation of a vector for the `bool` type, and instead has each `bool` stored as a bit instead of the type a stand-a-lone Boolean variable would be stored as.
 - Travis thinks this was a horrible design decision, but there is a reason why they did this. What are those reasons?

Caches

- ❖ Typically, a `bool` variable is 1 byte. How much space does a `bool` strictly *need* though?
 - 1 bit
- ❖ C++ goes against the standard implementation of a vector for the `bool` type, and instead has each `bool` stored as a bit instead of the type a stand-a-lone Boolean variable would be stored as.
 - Travis thinks this was a horrible design decision, but there is a reason why they did this. What are those reasons?
 - **A lot less space is taken up, and as a side effect of that, you probably don't have to call `malloc` as often and will have better cache performance**

Caches

- ❖ If we stored a vector of 120 `bool`s, and wanted to iterate over all of them, roughly how many cache hits & misses would we have if we:
 - You can assume a cache line is 64 bytes.
 - If we used a `vector<bool>` that allocates the bools normally (1 byte per bool)
 - If we use a `vector<bool>` that represents each bool with a single bit

Caches

- ❖ If we stored a vector of 120 `bool`s, and wanted to iterate over all of them, roughly how many cache hits & misses would we have if we:
 - You can assume a cache line is 64 bytes.
 - If we used a `vector<bool>` that allocates the bools normally (1 byte per bool)
 - 2 cache misses, 118 cache hits
 - If we use a `vector<bool>` that represents each bool with a single bit
 - 1 cache miss, 119 cache hits

Scheduling

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

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A															
B															
C															
D															

- ❖ 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

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- 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

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- 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

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- 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

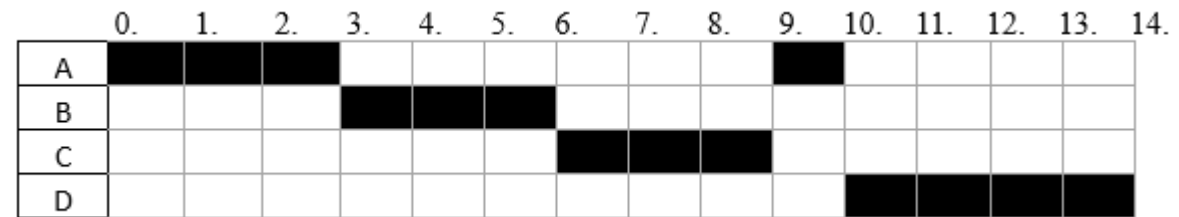


❖ If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?

■ Currently there are 7 context switches

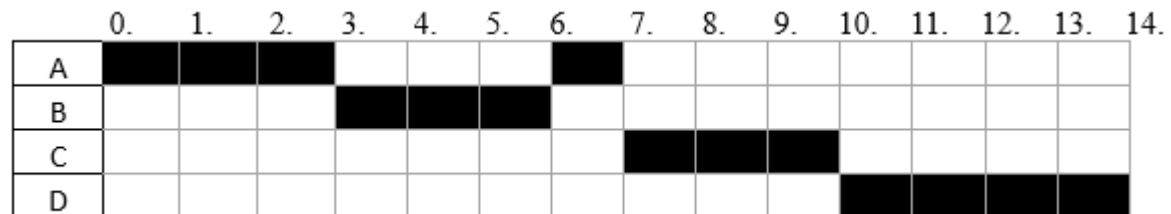
■ If quantum was 3:

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 Navigation

In a traditional Linux file system (like ext2), navigating a path like `/dir1/dir2/file.txt` involves multiple steps.

Describe what the file system must do to locate the inode for ***file.txt***, starting from the root directory.

File System Navigation

In a traditional Linux file system (like ext2), navigating a path like `/dir1/dir2/file.txt` involves multiple steps.

Describe what the file system must do to locate the inode for ***file.txt***, starting from the root directory.

1. First, we need to load in the blocks containing the directory entries for the root directory, `/`, in inode 2.
 - After looping through the blocks containing the dirents, we find the entry for `“dir1”` and its inode X.
2. We need to load in the blocks containing the directory entries for the directory, `“dir1”`, in inode X.
 - After looping through the dirents, we find the associated entry for `“dir2”` and its inode Y.
3. Finally, we load in the blocks containing the directory entries for the directory, `“dir2”`, in inode Y.
 - We can finally loop through the directory entries for `“dir2”` and find `file.txt`’s entry and thus corresponding inode. And we are done!

Largest File Possible

You are tasked with designing **MinimalFS**, where each file is represented by an inode.

- ❖ Each inode can operate in one of two modes: **small** or **large** mode.
- ❖ In small mode, the inode directly stores up to **5 block numbers** that point to file data.
- ❖ Each block is **1024 bytes** in size.
- ❖ Assuming a file contains at least some data (i.e., it's not empty), what is the **smallest amount of space** that would be allocated for a file?

Largest File Possible

You are tasked with designing **MinimalFS**, where each file is represented by an inode.

- ❖ Each inode can operate in one of two modes: **small** or **large** mode.
- ❖ In small mode, the inode directly stores up to **5 block numbers** that point to file data.
- ❖ Each block is **1024 bytes** in size.
- ❖ Assuming a file contains at least some data (i.e., it's not empty), what is the **smallest amount of space** that would be allocated for a file?

THE SMALLEST AMOUNT OF SPACE ALLOCATED TO A FILE IS ONE BLOCK SO 1024 BYTES!

Largest File Possible

You are tasked with designing **MinimalFS**, where each file is represented by an inode.

- ❖ Each inode can operate in one of two modes: **small** or **large** mode.
- ❖ In **large mode**, the inode directly stores up to **10 block numbers**. The **1st** is **singly indirect**, the next **7** are **double indirect**, and the last **2** are **triply indirect**.
- ❖ Each block is **1024 bytes** in size.
- ❖ And block numbers are 4 bytes large.
- ❖ Assuming a file contains at least some data (i.e., it's not empty), what is the **largest amount of space** that would be allocated for a file? Feel free to leave your answer as an expression.

Largest File Possible

You are tasked with designing **MinimalFS**, where each file is represented by an inode.

- ❖ In **large mode**, the inode directly stores up to **10 block numbers**. The **1st is singly indirect, the next 7 are double indirect, and the last 2 are triply indirect**.
- ❖ Each block is **1024 bytes** in size And block numbers are **4 bytes**.
- ❖ What is the **largest amount of space** that would be allocated for a file? Feel free to leave your answer as an expression.

$X = \# \text{ of Block Nums for singly indirect} = 1024/4$

$Y = \# \text{ of Block Nums for doubly indirect} = 7 * (1024/4 * 1024/4)$

$Z = \# \text{ of Block Nums for triple indirect} = 2 * (1024/4 * 1024/4 * 1024/4)$

$(X + Y + Z) * 1024 \text{ bytes}$ or just $x + y + x$ blocks would be fine to say or write on an exam

File System Block Allocation

- ❖ When you move (mv) a file from one directory to another **on the same Linux file system**, does the file's inode number have to change?

In other words, can the file keep the same inode number after the move?

What needs to happen for this to work correctly?

```
$ mv myfile ./dir/
```

- ❖ Here, in this command, we are moving the file 'myfile' to directory './dir'.

File System Block Allocation

- ❖ When you move (mv) a file from one directory to another **on the same Linux file system**, does the file's inode number have to change?

In other words, can the file keep the same inode number after the move?

What needs to happen for this to work correctly?

```
$ mv myfile ./dir/
```

- ❖ Here, in this command, we are moving the file 'myfile' to directory './dir'.

Yes, the inode number can stay the same!

To move the file, the system only needs to **update the directory entries**: it adds an entry for 'myfile' in the target directory (./dir) that points to the same inode, and then **removes** the old entry from the original directory (.).

The inode itself, and all the information stored in the inode, do not change (other than last accessed/modified time stamps if so).

Processes and Virtual Memory

❖ Take a look at the following program:

```
int main(){
    pid_t child = fork();
    if(child == 0){
        printf("I'm the child!:\n");
        return;
    }
    printf("Just exec'd a child!\n");
    waitpid(child, NULL, 0);
    return 0;
}
```

Suppose a kernel is unable to create new virtual memory mappings after a fork operation (you have an old computer what can I say). This means all address map to identical physical memory locations in each process here.

Could this program function correctly without requiring new mappings?

Processes and Virtual Memory

❖ Take a look at the following program:

```
int main(){
    pid_t child = fork();
    if(child == 0){
        printf("I'm the child!:\n");
        return;
    }
    printf("Just exec'd a child!\n");
    waitpid(child, NULL, 0);
    return 0;
}
```

Suppose a kernel is unable to create new virtual memory mappings after a fork operation (you have an old computer what can I say). This means all address map to identical physical memory locations in each process here.

Could this program function correctly without requiring new mappings?

REMEMBER: PRINTF MAINTAINS A BUFFER (GLOBAL STATE) AND IF THEY BOTH SHARE THE SAME BUFFER THEN NO BUENO! THEY NEED TO WRITE TO SEPARATE BUFFERS.

THERE IS ALSO pid_t child WHICH IS SHARED. THOUGH THIS COULD POSSIBLY BE KEPT IN SEPARATE REGISTERS.

Page Tables Q1

- ❖ One oddity is that page tables exist in memory themselves. However, the memory that is used to store *some (not all)* page tables are usually “pinned” in memory, meaning that those pages cannot be evicted/removed from physical memory even if we need more space.
- ❖ Why is it important that some of the pages containing these page tables remain “pinned”? Please explain your answer.

Page Tables Q1

- ❖ One oddity is that page tables exist in memory themselves. However, the memory that is used to store *some (not all)* page tables are usually “pinned” in memory, meaning that those pages cannot be evicted/removed from physical memory even if we need more space.
- ❖ Why is it important that some of the pages containing these page tables remain “pinned”? Please explain your answer.

A page table walk (resolving a physical address) might be required for **any** virtual address at **any** time — whether valid or invalid. To perform the walk, the system **must** be able to access the relevant page table entries. But if those entries themselves require translation (and we don't know where the page tables are in physical memory), we'd be stuck in a loop.

That's why some addresses — such as the ones containing the page tables — must be **pinned in physical memory** so the hardware can always find and use them without needing to translate further.

Page Tables Q2

- ❖ At the beginning, we imagined the page table as one giant array containing one page table entry for each page (where the page number was the index into the table). However, we saw that this design is pretty wasteful (do you remember why?)
- ❖ Let's say we had a virtual page number that we wanted to translate to a physical page number. What would the look up speed be of the "big array" page table be? What about one with 4 page table levels?

Page Tables Q2

- ❖ At the beginning, we imagined the page table as one giant array containing one page table entry for each page (where the page number was the index into the table). However, we saw that this design is pretty wasteful (do you remember why?)

We would then need one PTE for every virtual page, but most virtual pages won't have a mapping (or "exist") .

In multilevel we can allocate PTE's & create mappings later (when the page gets accessed for the first time and thus the mapping is needed)

Page Tables Q2

- ❖ Let's say we had a virtual page number that we wanted to translate to a physical page number. What would the look up speed be of the “big array” page table be? What about one with 4 page table levels?

The large array model provides **constant-time lookup** with just **one memory access**, since the specific section of the table we need can be **directly indexed** using the virtual page number (VPN). Once we have the VPN, we index into the page table, and the translation is complete. Tada.

In contrast, a 4-level page table requires us to **traverse 4 separate memory locations**, one for each level of the hierarchy. Although this may seem like a minor increase, the overhead can quickly add up — especially if any of those accesses trigger a **page fault**, causing the system to load different page table levels from disk...no bueno (but we hope this doesn't happen. ;))

Page Replacement Policy

- ❖ Eric and Akash are debating the best page replacement policy. One of them says that LRU is strictly better (e.g. better in all cases) than FIFO page replacement and always leads to less page faults.
- ❖ Is this true or false? Please explain your answer. If it is not true, provide an example of page accesses that counters this claim.

Page Replacement Policy

- ❖ Eric and Akash are debating the best page replacement policy. One of them says that LRU is strictly better (e.g. better in all cases) than FIFO page replacement and always leads to less page faults.
- ❖ Is this true or false? Please explain your answer. If it is not true, provide an example of page accesses that counters this claim.

False: consider we have 4 physical pages and have the reference string:

0 1 2 3 0 4 1 2 3

In LRU we get 8 page faults

In FIFO we get 5 page faults

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

- ❖ 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?
- ❖ (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 = "";
    }
}
```

Deadlock

- ❖ Consider we are working with a data base that has N numbered blocks. Multiple threads can access the data base and before they perform an operation, the thread first acquires the lock for the blocks it needs.
 - Example: Thread1 accesses B3, B5 and B1. Thread2 may want to access B3, B9, B6. Here is some example pseudo code:

```
void transaction(list<int> block_numbers) {  
    for (every block_num in block_numbers) {  
        acquire_lock(block_num)  
    }  
  
    operation(block_numbers);  
  
    for (every block_num in block_numbers) {  
        release_lock(block_num);  
    }  
}
```

Deadlock

- This code has the possibility to deadlock. Give an example of this happening. You can assume no thread tries to acquire the same lock twice
- Someone proposes we fix this by locking the whole database instead of locking at the block level. What downsides does this have? Does it even avoid deadlocks?
- How can we fix this (without locking the whole database if that even works)?

```
void transaction(list<int> block_numbers) {  
    for (every block_num in block_numbers) {  
        acquire_lock(block_num)  
    }  
  
    operation(block_numbers);  
  
    for (every block_num in block_numbers) {  
        release_lock(block_num);  
    }  
}
```


Deadlock

- This code has the possibility to deadlock. Give an example of this happening. You can assume no thread tries to acquire the same lock twice
 - **Thread 1 wants B2 and B4. Thread 2 also wants B2 and B4, but lists them in a different order. Thread 1 gets B2, Thread 2 get B4, and we deadlock.**

```
void transaction(list<int> block_numbers) {  
    for (every block_num in block_numbers) {  
        acquire_lock(block_num)  
    }  
  
    operation(block_numbers);  
  
    for (every block_num in block_numbers) {  
        release_lock(block_num);  
    }  
}
```

Deadlock

- Someone proposes we fix this by locking the whole database instead of locking at the block level. What downsides does this have? Does it even avoid deadlocks?
 - **This works, but now our data base is run entirely sequentially for these transactions even if two thread have completely separate blocks they operate on, they cannot run in parallel.**

```
void transaction(list<int> block_numbers) {  
    for (every block_num in block_numbers) {  
        acquire_lock(block_num)  
    }  
  
    operation(block_numbers);  
  
    for (every block_num in block_numbers) {  
        release_lock(block_num);  
    }  
}
```

Deadlock

- How can we fix this (without locking the whole database if that even works)?
- **Have each thread acquire the locks in a strict increasing numerical order. This prevents any cycles from happening**

```
void transaction(list<int> block_numbers) {  
    for (every block_num in block_numbers) {  
        acquire_lock(block_num)  
    }  
  
    operation(block_numbers);  
  
    for (every block_num in block_numbers) {  
        release_lock(block_num);  
    }  
}
```