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	

Poll Everywhere

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…?

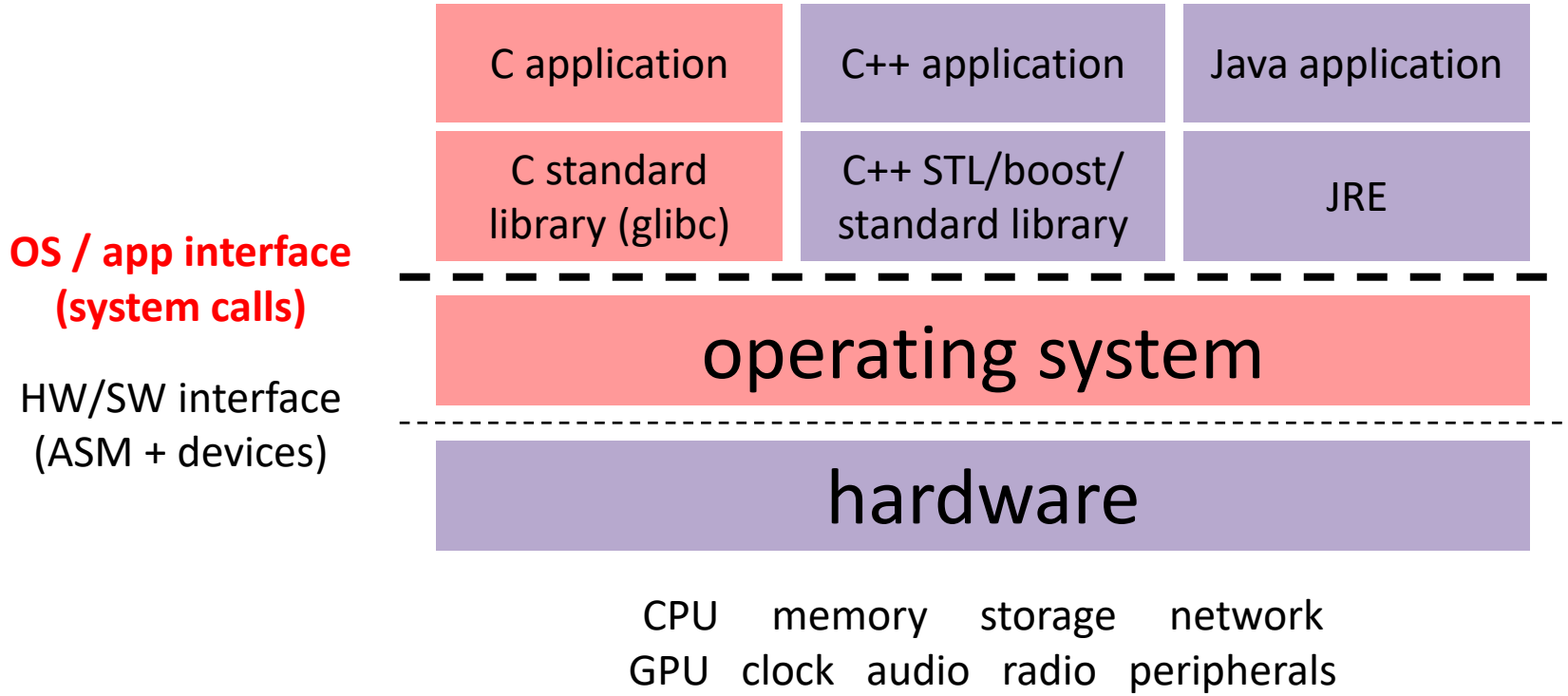
Software System

- 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 Programing
  - 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!

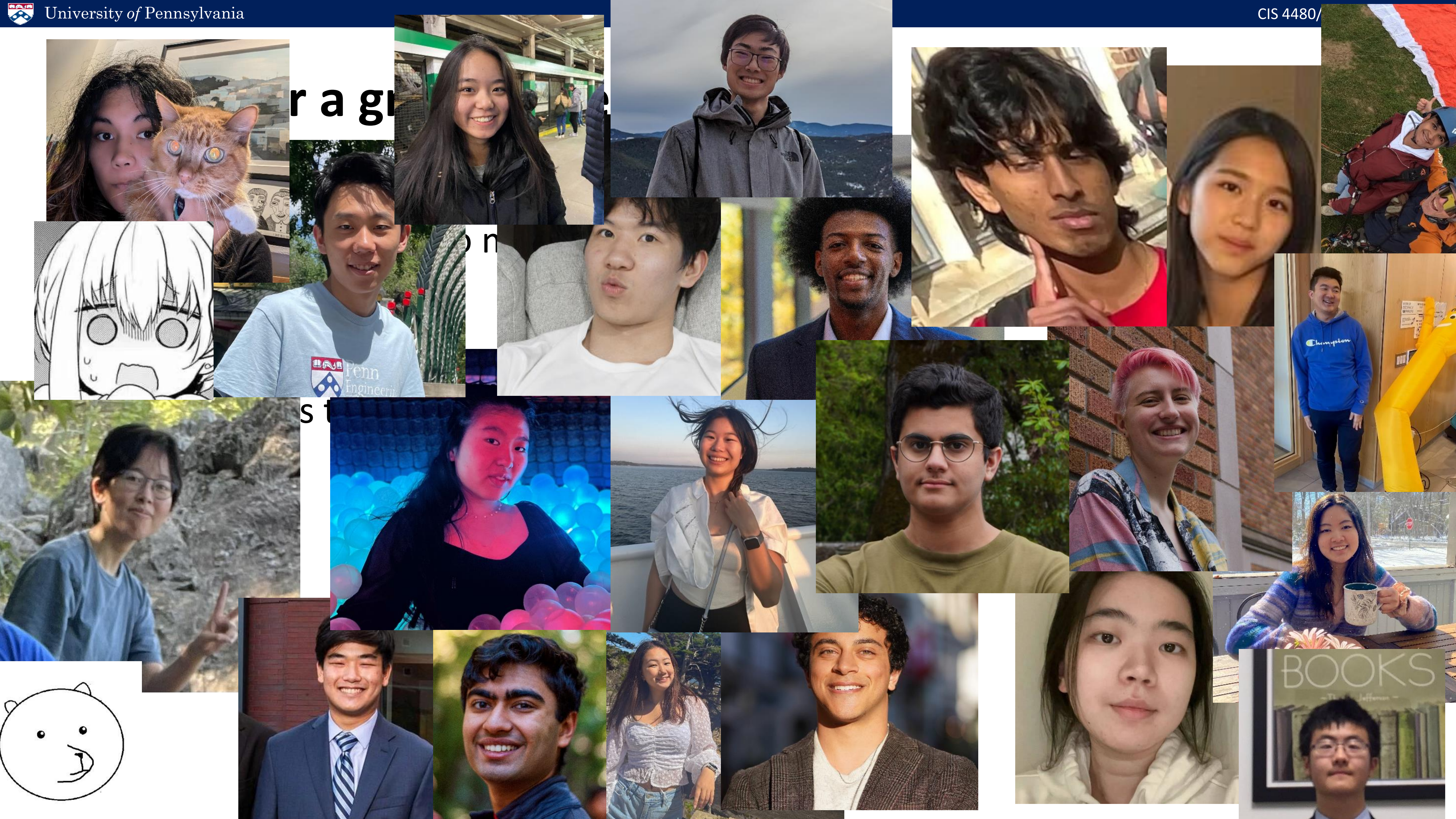
- Special thanks to all the instructors before me (Both at UPenn and UW) who have influenced me to make the course what it is

- Huge thanks to the course TA's for helping with the course!

r a gr

o n

s t



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 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

- 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.		

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

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

```c
#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);
}
```

```c
#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

- 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

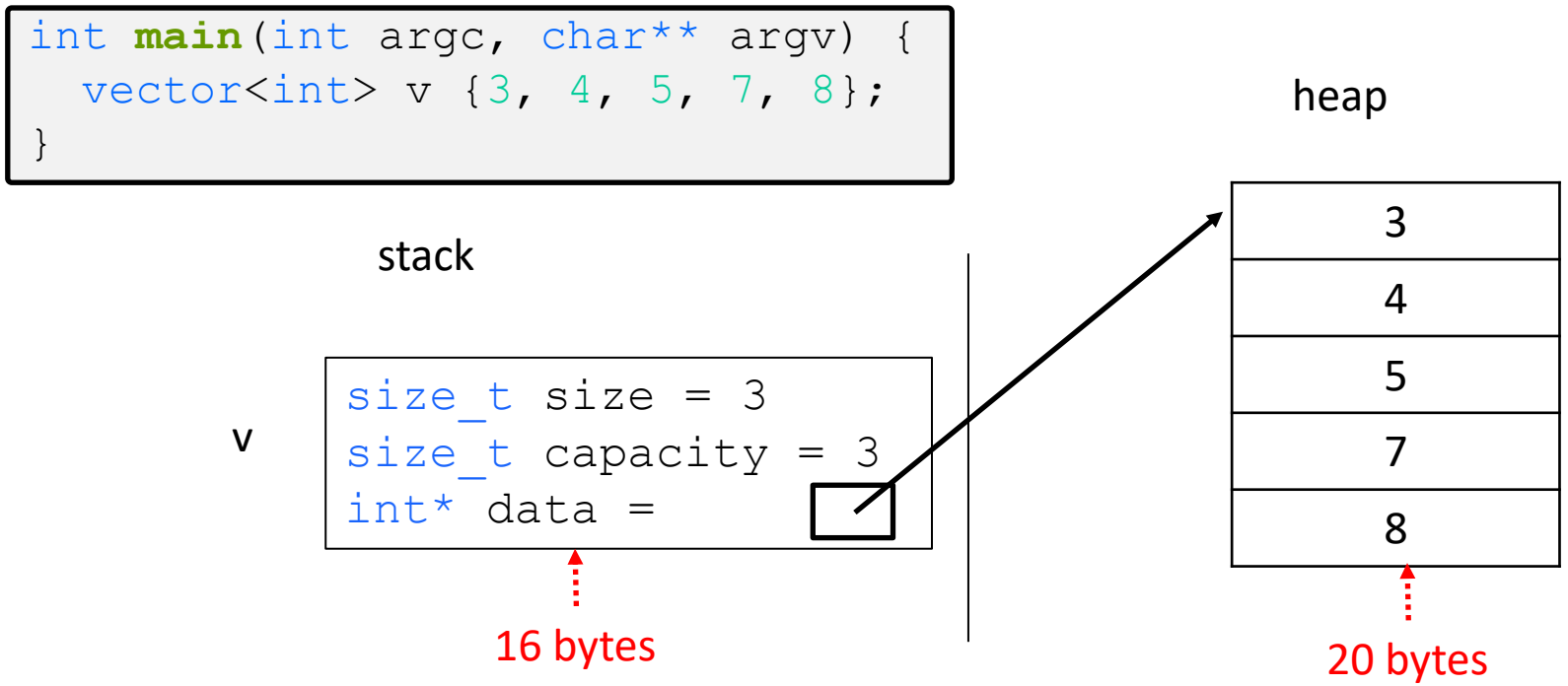
- How is the array in this snippet of code likely allocated at a low level (in assembly)?

```c
#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]);
}
```

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:



Caches

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

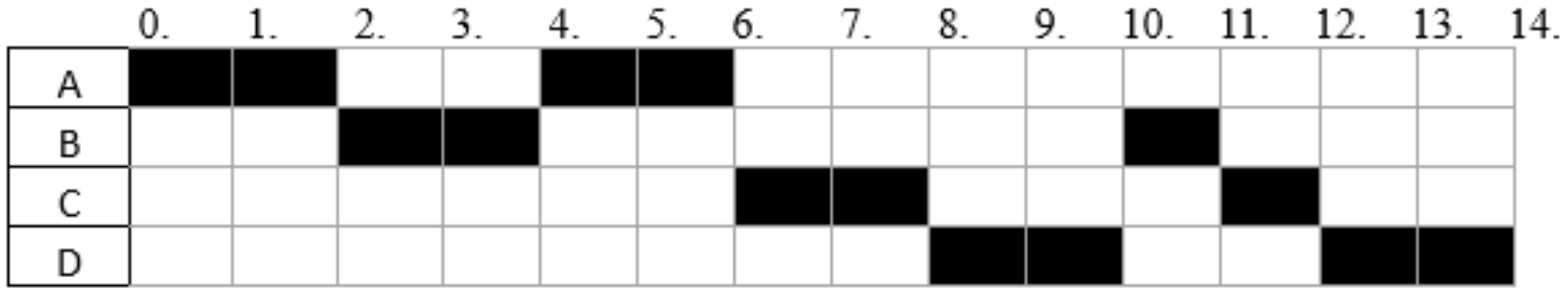
- C++ goes against the standard implementation of a vector for the bool type, and instead has each bool stored as a bit instead of the type a stand-a-lone Boolean variable would be stored as.
  - Travis thinks this was a horrible design decision, but there is a reason why they did this. What are those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

Caches

- 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

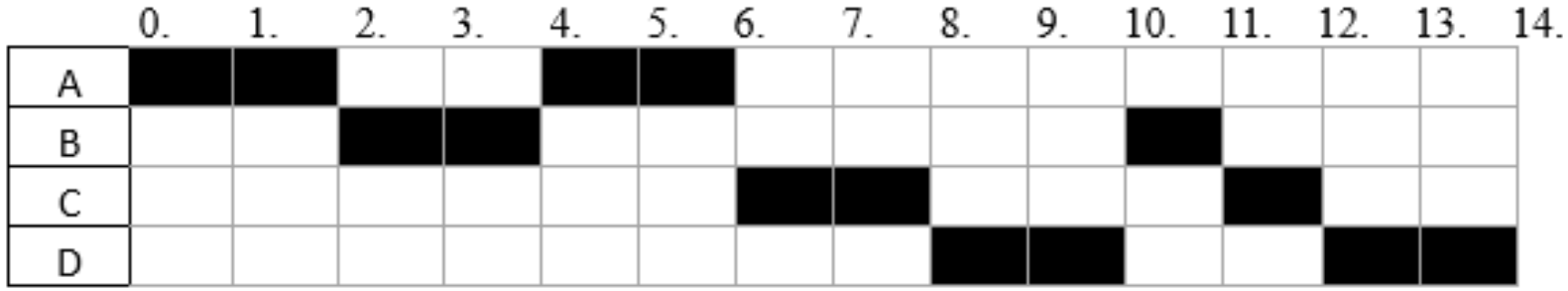
Scheduling

- 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



- 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 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.

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 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.

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'.

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?

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 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 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.

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

- 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 = "";
  }
}
```

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);
  }
}
```