Final Exam Review (Pt. 3)

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Sid Sannapareddy Maya Huizar

Poll Everywhere

pollev.com/tqm

- Any General questions on the course or PennOS before we begin?

Disclaimer

- **THIS REVIEW IS NOT EXHAUSTIVE**
- **Topics not in this review are still testable**

Disclaimer

- THIS IS REVIEW
- DO THE OLD EXAMS FOR THE BEST PRACTICE YOU CAN GET

New Problems Added

- New Cache Problem
- New Virtual Memory Problem
- New Memory Allocation Problem
- New File System Problem (Added now ☺)

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?

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?

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.		

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?

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?

```
#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

- 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[1-2];
  }

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

Slab Slob Slub ☺

- Sid is making a custom memory allocator that internally uses 4 different slab allocators with object sizes: 16 bytes, 64 bytes, 256 bytes, and 4096 bytes. Each of these allocators manage slabs that are 2 pages (8192 bytes) big.

- If a user makes these requests in this order, how much internal fragmentation do we get in total?
  - 18 bytes
  - 64 bytes
  - 1000 bytes
  - 2400 bytes
  - 152 bytes
  - 3990 bytes

- If we started with no slabs allocated, how many pages are being managed by the allocators after these allocations?

Slab Slob Slub ☺

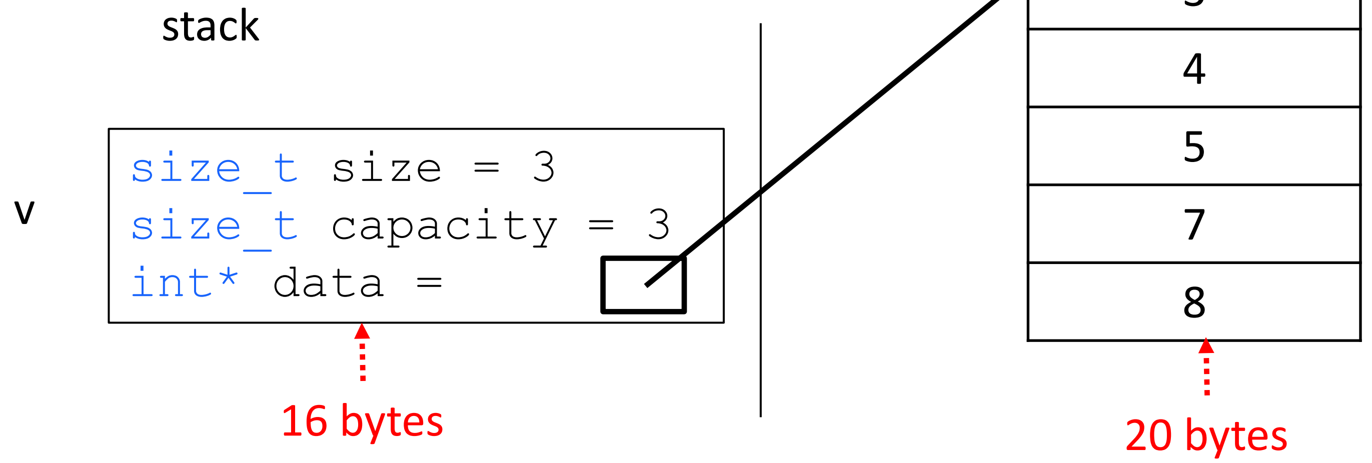
- If a Slab allocator has multiple slabs that can fulfill an allocation request, it prefers to fill up slabs that are closest to being full. This is sort of the opposite of our "worst fit" algorithm. Why does the slab allocator do this?

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:

```cpp
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

- 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 Q2

- Let's say we are making a program that simulates various particles interacting with each other. To do this we have the following structs to represent a color and a point

```
struct color {
  int red, green, blue;
};
```

```
struct point {
  double x, y;
  struct color c;
};
```

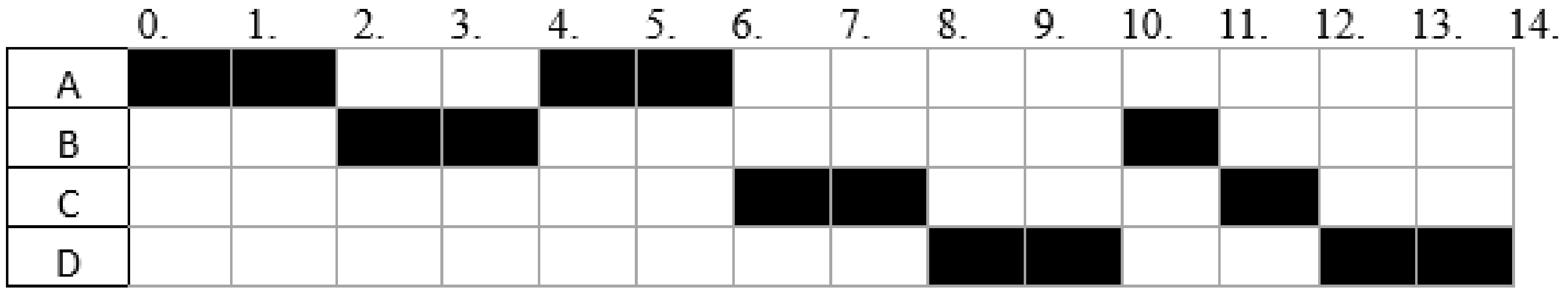
- If we were to store 100 point structs in an array, and iterate over all of them, accessing them in order, roughly how many cache hits and cache misses would we have?
  - Assume:
    - a cache line is 64 bytes
    - the cache starts empty
    - `sizeof(point)` is 32 bytes, `sizeof(color)` is 16 bytes

Caches Q3

- Consider the previous problem with point and color structs.
- In our simulator, it turns out a VERY common operation is to iterate over all points and do calculations with their X and Y values.
- How else can we store/represent the point objects to make this operation faster while still maintaining the same data? Roughly how many cache hits would we get from this updated code?

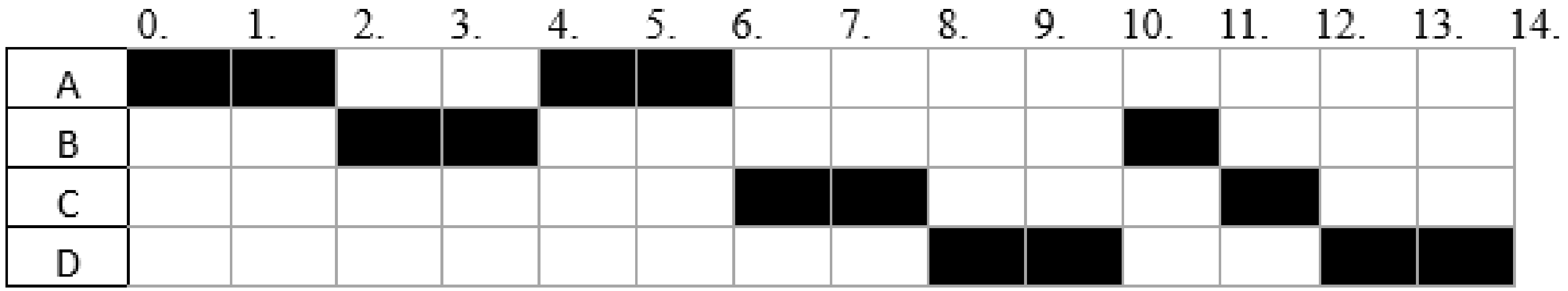
Scheduling

- 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



	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?

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'.

Cow Files

- Some old systems would do COW (copy-on-write) for files. Meaning that when we copied a file, we didn't make a copy, instead we would have two directory entries that refer to the same I-node, and a real copy was only made once one file had been written to.

- Assume we are on an I-node based system.

- What are the benefits of implementing file copying this way?

- If we wanted to support COW for files, how do we do that?
  - What (if anything) would need to change in the process-level file descriptor table, system open file table, directory entries, etc.
  - What would need to happen when a copy is made and when a copy is written to?

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.

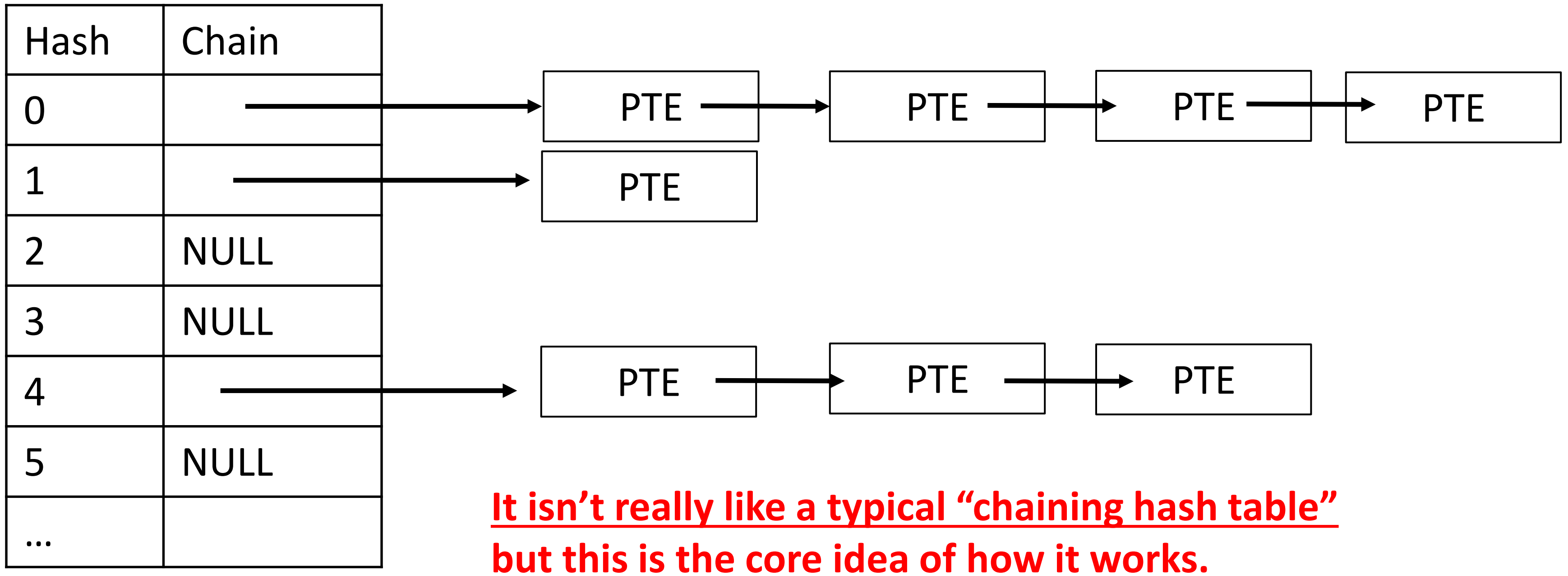
- Is this true or false? Please explain your answer. If it is not true, provide an example of page accesses that counters this claim.

Inverted Page Table

- Some systems used something different than a multi-level page table. They realized that there are a lot more virtual pages than there are physical pages… So why not just have one entry per physical page?
- Would be one global page table since it is based on physical memory
- Implemented essentially as a chaining hash table

Inverted Page Table

- Chaining Hash Table
  - Hash: If a process wants to lookup to see if a page is in physical memory, it hashes the virtual page number and iterates through that chain



Hash	Chain
0	
1	
2	NULL
3	NULL
4	
5	NULL
…	

It isn't really like a typical "chaining hash table" but this is the core idea of how it works.

Inverted Page Table

- How can we enforce process isolation in the table if it is shared across all processes?
- What is at least one advantage this has over the multi-level page tables?
- What is at least one disadvantage? (Other than the one in the question below)
- It turns out there is a benefit to having an entry for mappings that still exist, but point to the swap file and not physical memory. Why do you think this is?

Threads & Data Races

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

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

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

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

Threads & Data Races

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

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

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

Threads & Data Races

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

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

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

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

Threads & Data Races

- There is an issue of inefficient CPU utilization going on in this code. What is it and how can we fix it?

- (You can describe the fix at a high level, no need to write code)

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

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

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

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);
  }
}
```