Course Review

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

Administrivia

- PennOS
  - Everyone should have already contacted their group, and should get started working on it.
  - Full Thing due ~April 22nd (Yesterday)
    - Can still use late tokens, so late deadline is April 26th
    - After you submit, you need to schedule a meeting with your TA to demonstrate that it is working
  - **There will be a PennOS Team evaluation form that goes out sometime soon**
    - **Will be due on the last day of classes: wednesday (5/1) @ midnight**

Administrivia

- Post semester survey
  - To be released soon, due sometime next week
- We released stress.c and stress.h for testing your PennOS kernel
- **The full PennOS demo plan is on ed, please look at it!**
- CIS TA Application is out now!
  - 2400 is “due” April 26th @ midnight

pollev.com/tqm

- How is PennOS going? Any questions related to it?

Midterm Philosophy / Advice (pt. 1)

- I do not like midterms that ask you to memorize things
  - You will still have to memorize some critical things.
  - I will hint at some things, provide documentation or a summary of some things. (for example: I will provide parts of the man pages for various system calls)

- I am more interested in questions that ask you to:
  - Apply concepts to solve new problems
  - Analyze situations to see how concepts from lecture apply

- Will there be multiple choice?
  - If there is, you will still have to justify your choices

Midterm Philosophy / Advice (pt. 2)

- I am still trying to keep the exam fair to you, you must remember some things
  - High level concepts or fundamentals. I do not expect you to remember every minute detail.
    - E.g. how a multi level page table works should be know, but not the exact details of what is in each page table entry
    - (I know this boundary is blurry, but hopefully this statement helps)

- I am NOT trying to "trick" you (like I sometimes do in poll everywhere questions)

Midterm Philosophy / Advice (pt. 3)

- I am trying to make sure you have adequate time to stop and think about the questions.
  - You should still be wary of how much time you have
  - But also, remember that sometimes you can stop and take a deep breath.
- Remember that you can move on to another problem.
- Remember that you can still move on to the next part even if you haven't finished the current part

Midterm Philosophy / Advice (pt. 4)

- On the midterm you will have to explain things
- Your explanations should be more than just stating a topic name.
- Don't just say something like (for example) "because of threads" or just state some facts like "threads are parallel and lightweight processes".
- State how the topic(s) relate to the exam problem and answer the question being asked.

Disclaimer

- **THIS REVIEW IS NOT EXHAUSTIVE**
- **Topics not in this review are still testable**
- **Recitation after lecture is exam review**

Practice Problems

- Processes vs Threads
- Memory Allocation
- Caches
- Scheduling
- Virtual Memory
- Threads & Data Races
- Deadlock

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

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

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.		

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.		

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.		

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.		

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.		

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

Memory Allocation Q1

- Slab allocator is really fast, but it would be inconvenient to replace malloc with a slab allocator. Why is that?

- How much internal and external fragmentation does a slab allocator have?

Memory Allocation Q1

- Slab allocator is really fast, but it would be inconvenient to replace malloc with a slab allocator. Why is that?

Slab allocator only handles allocations of a specific size. If we replaced malloc with it, we could not handle allocations of all sizes. Allocation requests that are too big would not work and allocations of a small size would have a lot of internal fragmentaiont

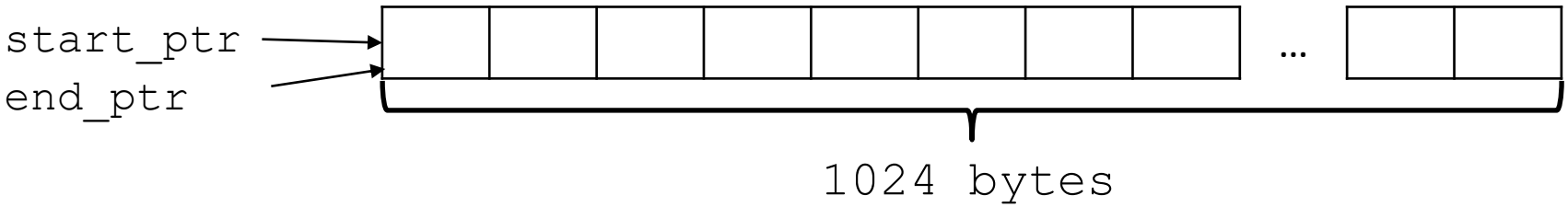
- How much internal and external fragmentation does a slab allocator have?

Minimal/none for both ☺ Since we know how big each allocation is, we can allocate the exact size requested (no internal) and chunk our memory so that there is minimal space between each allocated chunk

Memory Allocation Part 2

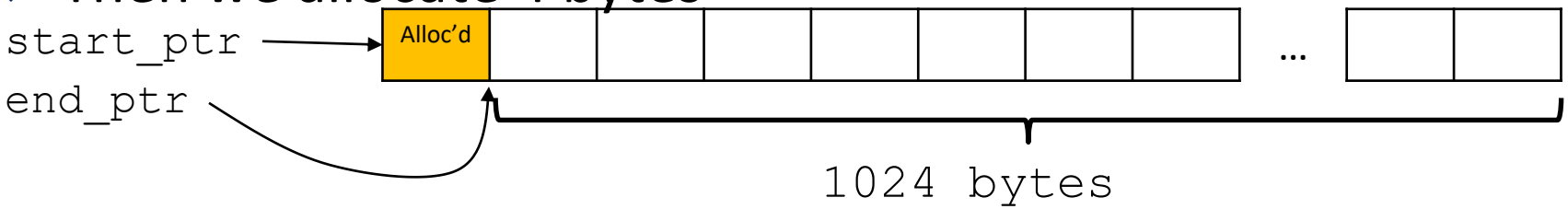
- In some instances, we want to allocate a lot of items and limit those allocations to one scope. We call our allocator a “temp_allocator” since it allocates things that are expected to be temporary to some scope.

- For example, Consider we start with:



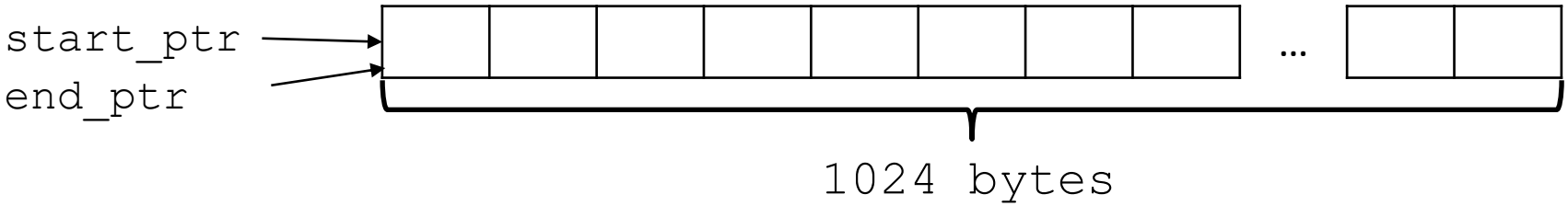
  - Note that there is no metadata, just these two pointers

- Then we allocate 4 bytes



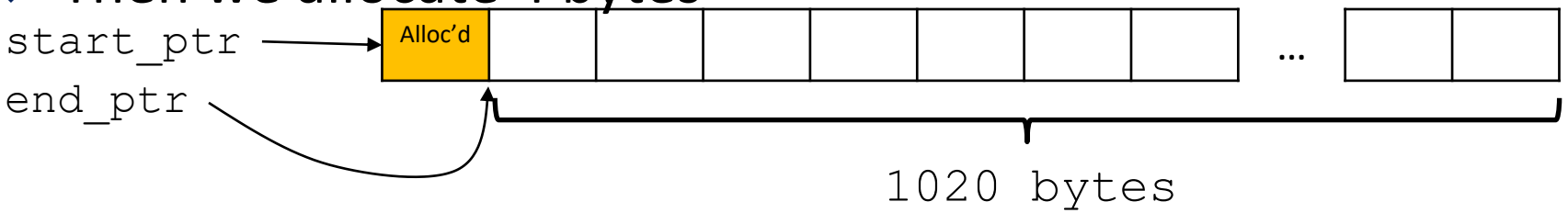
Memory Allocation Part 2

- For example, Consider we start with:

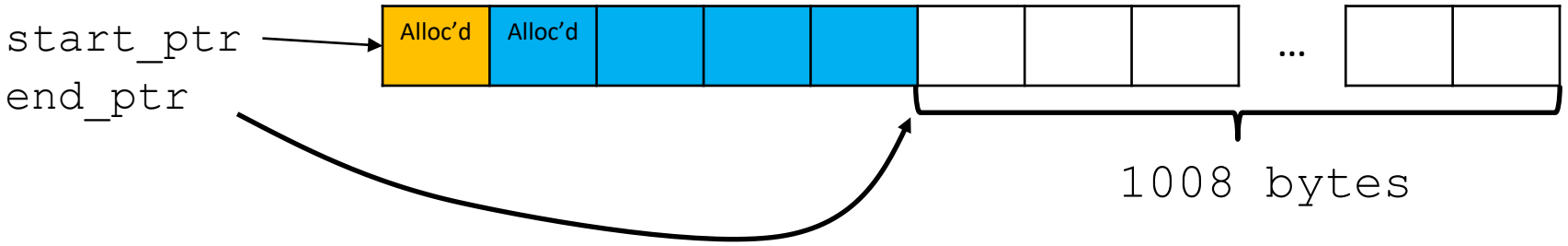


  - Note that there is no metadata, just these two pointers

- Then we allocate 4 bytes

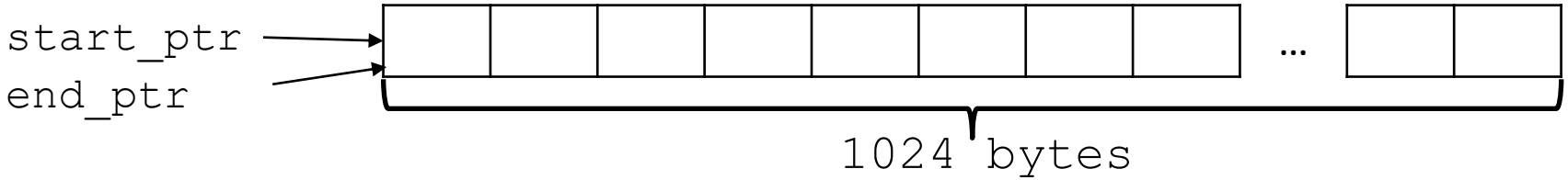


- Then we allocate 16 bytes



Memory Allocation Part 2

- Once we are done with our temporaries, we free the all allocations, and we can then use it again as if "fresh"



  - Looks the same as when we started!
- **That is the entire API**
- Example usage:

```
temp_allocator t_alloc = init_allocator();
for (many iterations) {
  int *ptr = allocate(t_alloc, 4 bytes);
  image *img = allocate(t_alloc, 1024 bytes);
  // a bunch of other allocations local
  // to this scope
  clear_allocs(t_alloc);
}
```

Memory Allocation Part 2

- How fast is our allocator at allocating things on average? At freeing things?
- What does the internal and external fragmentation look like with our allocator?
- Why can't we use this as a replacement for malloc maintaining lists of allocated & freed memory?

Memory Allocation Part 2

- How fast is our allocator at allocating things on average? At freeing things?

  Very Fast, constant time for each

- What does the internal and external fragmentation look like with our allocator?

  Minimal/none for both ☺ Since we know how big each allocation is, we can allocate the exact size requested (no internal) and chunk our memory so that there is minimal space between each allocated chunk

- Why can't we use this as a replacement for malloc maintaining lists of allocated & freed memory?

  Malloc manages things that are freed individually that may be allocated for varying lengths of time. This allocator assumes everything can be allocated together.

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

- 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]);
}
```

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)?

```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]);
}
```

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 * sizeof(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)

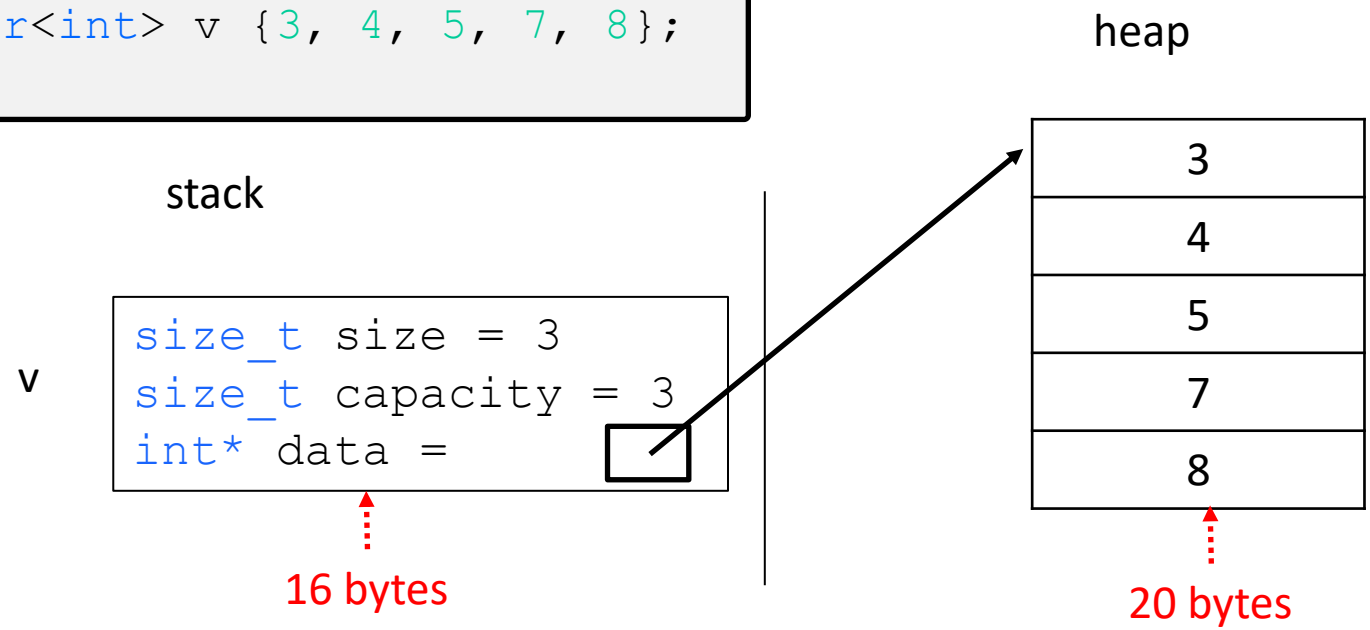
```c
#include <stdio.h>
#include <stdlib.h>

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:

```
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

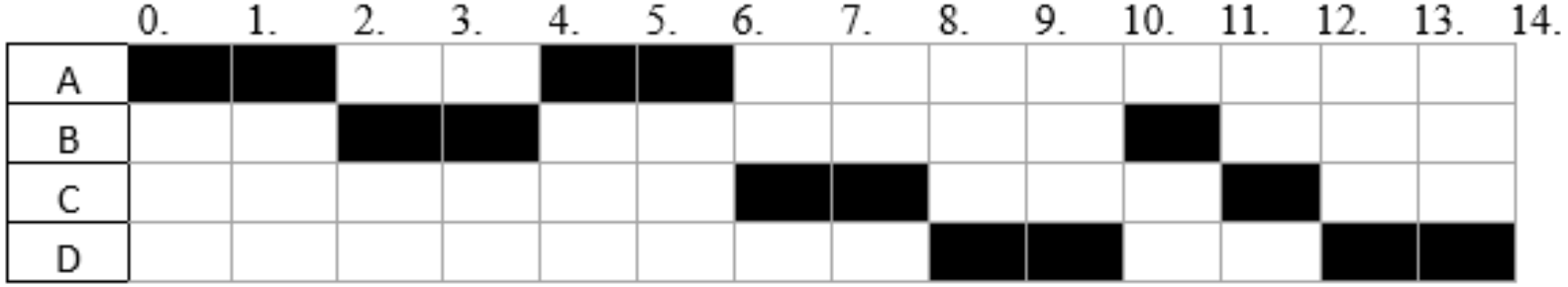
- If we stored a vector of 120 **`bool`**s, and wanted to iterate over all of them, roughly how many cache hits & misses would we have if we:
  - You can assume a cache line is 64 bytes.
  - If we used a **`vector<bool>`** that allocates the bools normally (1 byte per bool)
  - If we use a **`vector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

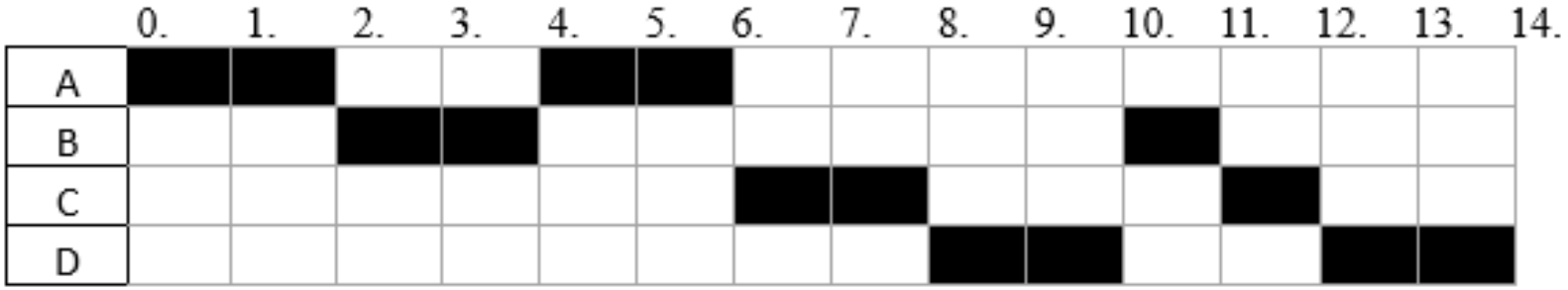
Scheduling

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



- 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.

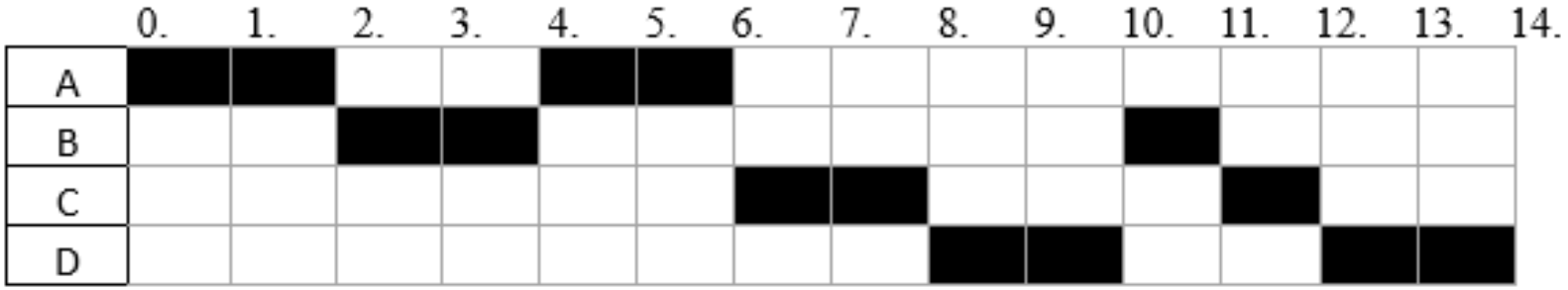
Scheduling



- 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?

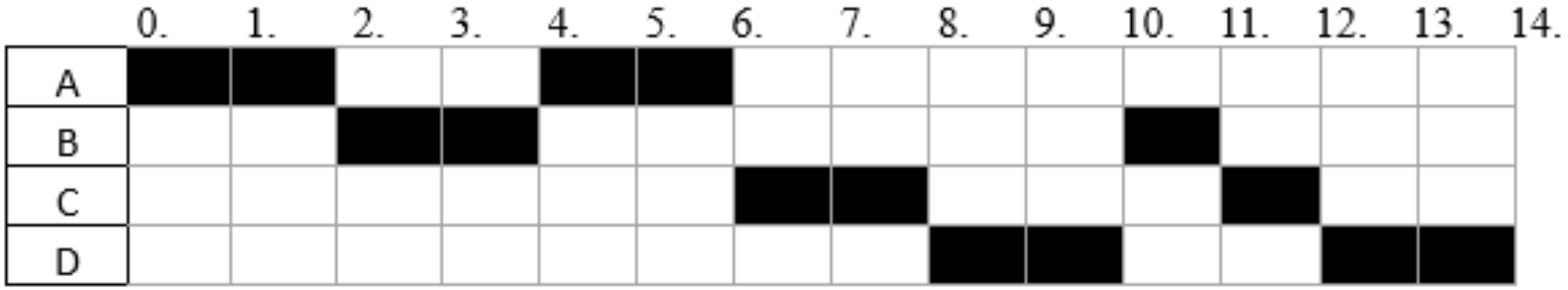
Scheduling



- 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?
  - 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

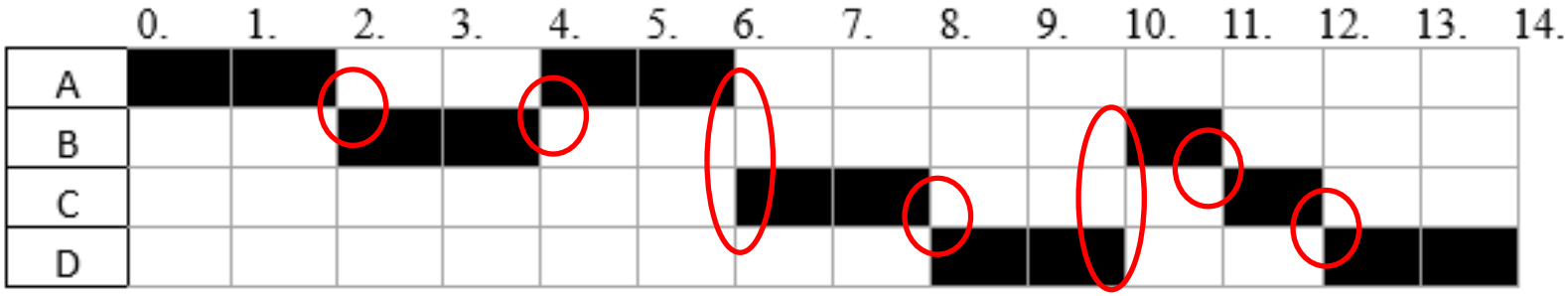
Scheduling



- 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.

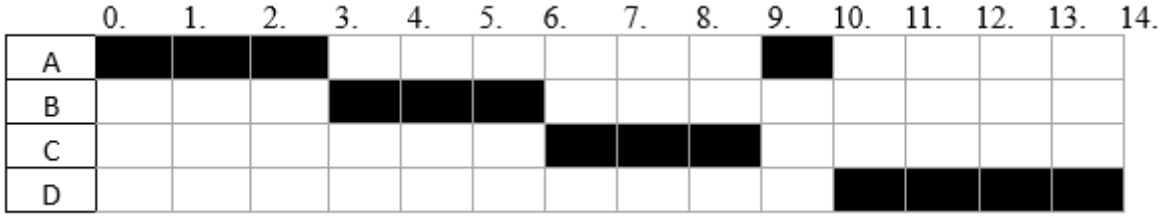
- Which processes are in the ready queue at time 9?
  - D is running, so it is not in the queue
  - A has finished
  - B and C still have to finish, so they are in the queue.

Scheduling

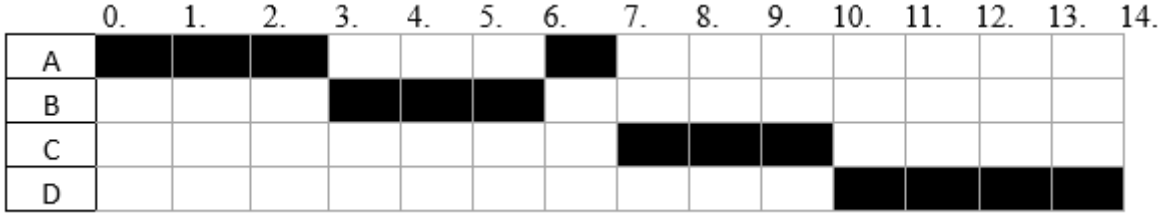


- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:

Depends on if C shows up at time 3 or 4



  - Or:



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

Page Tables Q1

- One oddity about page tables is that the page table itself exists in memory. However, the memory that is used to store some page tables are usually "pinned" into memory, meaning that those page tables cannot be evicted/removed from physical memory.

- Why is it important that some of the memory representing these page tables remain "pinned"? Please explain your answer.

Page Tables Q1

- One oddity about page tables is that the page table itself exists in memory. However, the memory that is used to store some page tables are usually "pinned" into memory, meaning that those page tables cannot be evicted/removed from physical memory.

- Why is it important that some of the memory representing these page tables remain "pinned"? Please explain your answer.

Page tables exist in virtual memory, meaning we may need to do a lookup of the address of nodes in the page table. If we don't have some addresses pinned or specially handled, we could not do translations since we wouldn't know what physical memory contains the page table entries we need

Page Tables Q2

- When we first brought up the idea of page tables, we imagined the page table as one giant array containing one page table entry for each page. We investigated other page table implementations (inverted and multi-level) since this "big array" model uses up A LOT of space for entries that may never be used.

- Let's say we had a virtual page number that we wanted to translate to a physical page number. How would the lookup speed of our original "big array" page table model compare to the more space efficient page tables implementations?

Page Tables Q2

- When we first brought up the idea of page tables, we imagined the page table as one giant array containing one page table entry for each page. We investigated other page table implementations (inverted and multi-level) since this "big array" model uses up A LOT of space for entries that may never be used.

- Let's say we had a virtual page number that we wanted to translate to a physical page number. How would the lookup speed of our original "big array" page table model compare to the more space efficient page tables implementations?

It would be constant time lookup and only one memory access. We can index into the page table using the virtual page number we want to ranslate

Page Tables Q3

- One thing that is different about inverted page tables is that the page table has one entry per physical page instead of per virtual page.

- Because of this, a page table can be shared across all processes instead of being per process. This is since all processes share physical memory.

- If a page table is shared across all processes, what issues could this cause? How does an inverted page table handle this issue?

Page Tables Q3

- One thing that is different about inverted page tables is that the page table has one entry per physical page instead of per virtual page.
- Because of this, a page table can be shared across all processes instead of being per process. This is since all processes share physical memory.
- If a page table is shared across all processes, what issues could this cause? How does an inverted page table handle this issue?

We need to make sure processes only use memory allocated to that process. Inverted page table also stores the process ID with each entry and uses it in the hash to make sure only processes with the specified ID accesses that entry

Page Replacement Policy

- Seungmin and Nate 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

- Seungmin and Nate 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()**

```
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?
- (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);
  }
}
```