Exam Review

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Reach out to TA's to schedule PennOS Demo ASAP
  - Today and tomorrow are the last days to demo
  - You should use the version of PennOS you submitted unless you got prior approval to use one with small bug fixes.

- Exam is Thus 7-9pm in Meyerson B1
  - Exam policies and review materials will be posted after lecture.

Administrivia

- Two things due before Reading Days, will be released after the exam
  - Check-in (Short survey), done anonymously and pass/fail
  - Team Evaluation for PennOS
    - Pass/fail
    - Done individually, you will describe how much and what everyone contributed to pennos
    - We will use this to handle cases where there was a large imbalance in the work done.
    - If there are big inconsistencies between team members, we will investigate

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

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**

Lecture Outline

- Processes vs Threads
- Memory Allocation
- Caches
- Scheduling
- File System Block Allocation
- RAID
- 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

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)?

```
#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]);
}
```

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

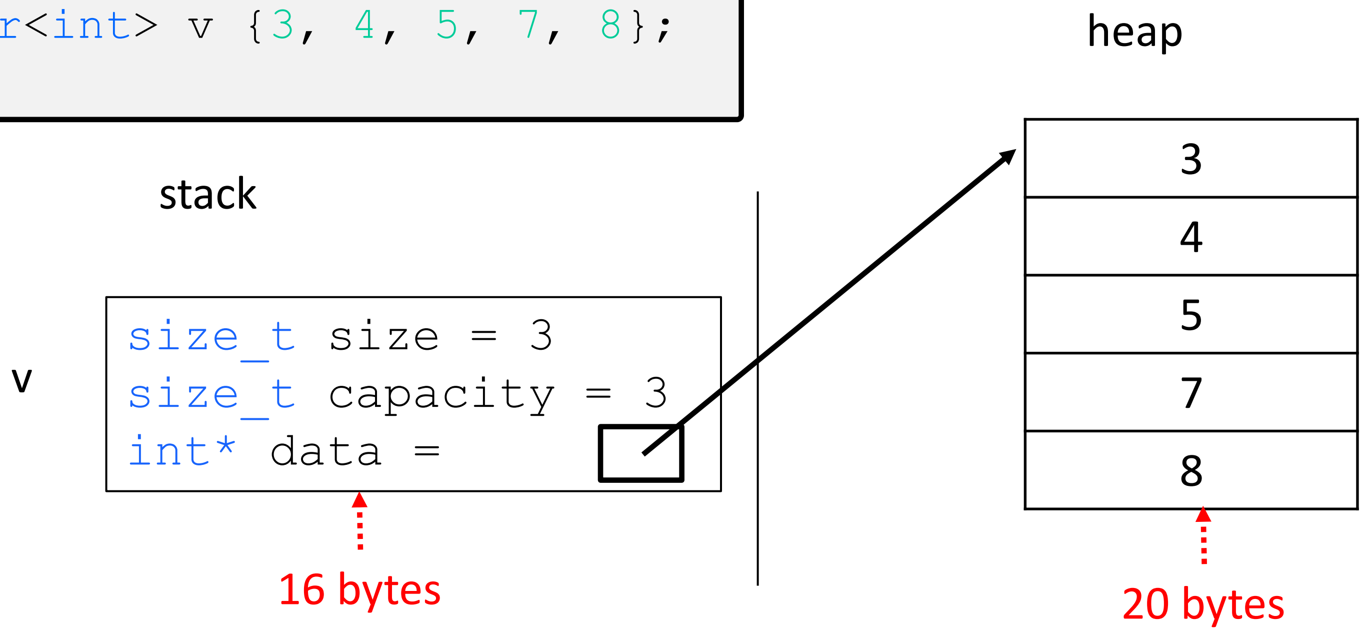
Would also accept more vague answers like (grow the stack by 10 integers)

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

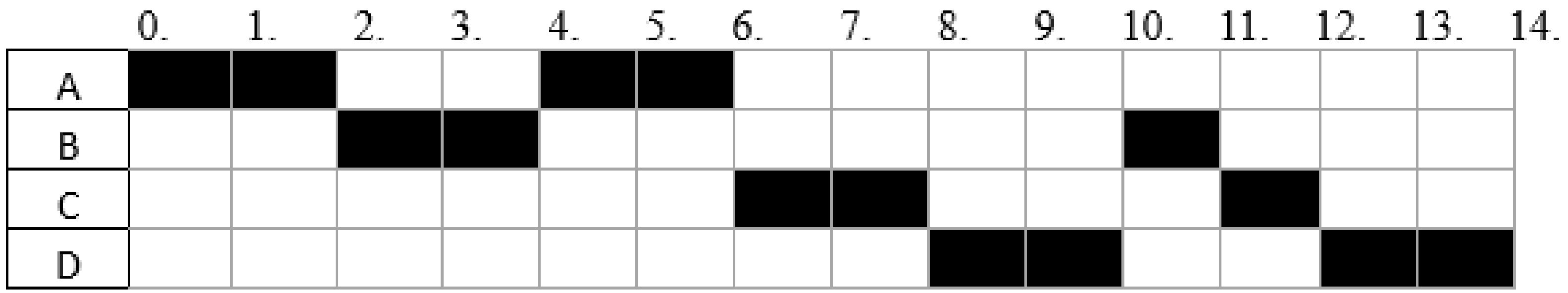
- If we stored a vector of 120 **`bool`**s, and wanted to iterate over all of them, roughly how many cache hits & misses would we have if we:
  - You can assume a cache line is 64 bytes.
  - If we used a **`vector<bool>`** that allocates the bools normally (1 byte per bool)
  - If we use a **`vector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

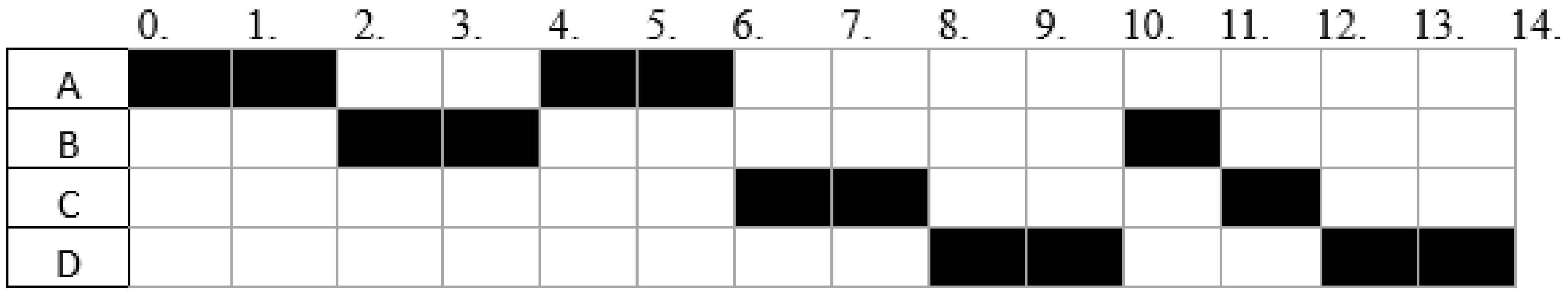
Scheduling

	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

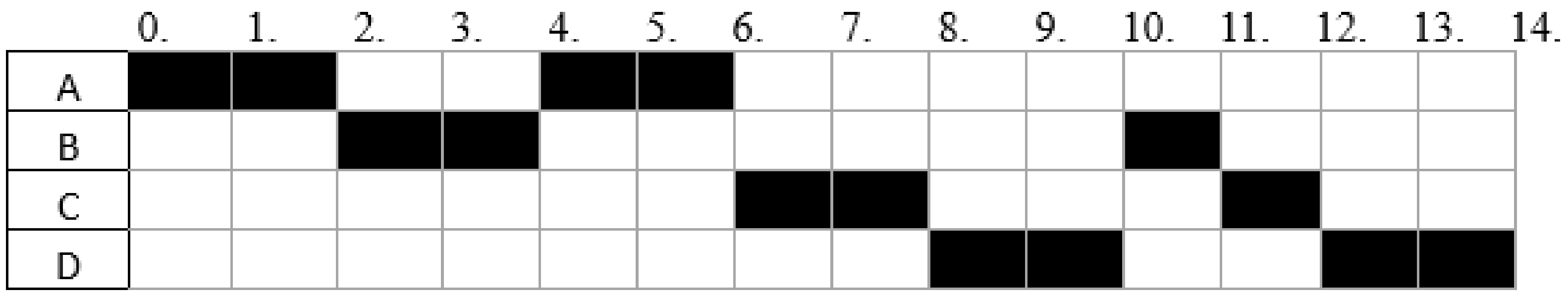


	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- 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

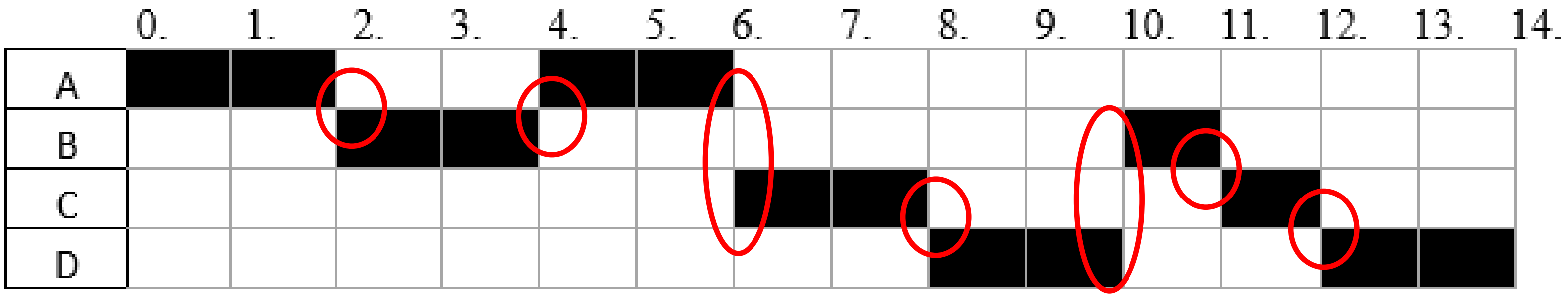


- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

❖ Which processes are in the ready queue at time 9?

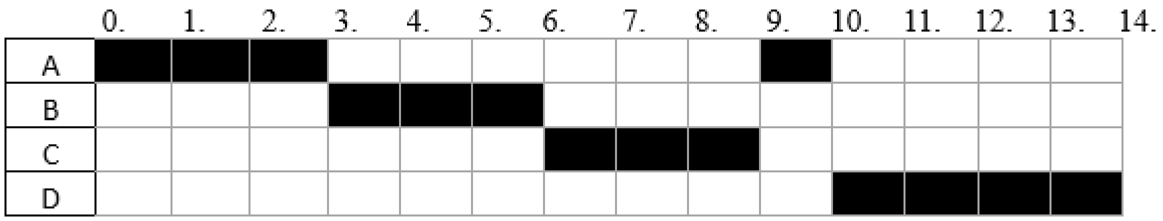
- D is running, so it is not in the queue
- A has finished
- B and C still have to finish, so they are in the queue.

Scheduling



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

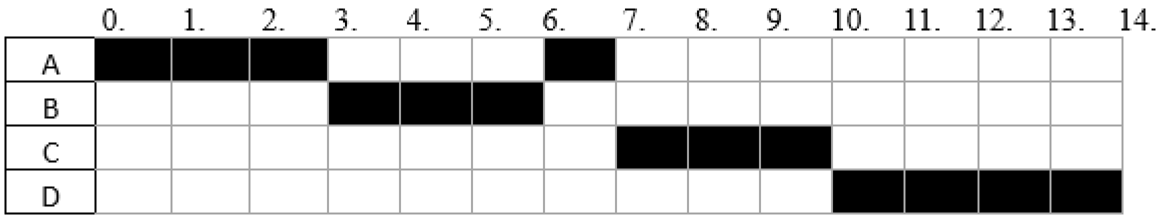
- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:



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

Depends on if C shows up at time 3 or 4

  - Or:



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

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

File System Block Allocation

- Consider that we want to read the 5th block of the file **`/home/me/.bashrc`**, what is the worst-case number of disk blocks that must be read in for each of the following:
  - You can assume a block is 4096 bytes
  - assume that directory entries we are looking for are in the firs block of each directory we search
- Linked List Allocation
  - Assume we know the block number of the first block in root dir
- Linked List Allocation via FAT
  - Assume we know where the root directory starts in the FAT.
  - You can also assume a FAT entry is 2 bytes.
- I-nodes
  - assume we know where the I Node for the root directory is

File System Block Allocation

- Consider that we want to read the 5th block of the file **`/home/me/.bashrc`**, what is the worst-case number of disk blocks that must be read in for each of the following:
  - You can assume a block is 4096 bytes
  - assume that directory entries we are looking for are in the firs block of each directory we search
- Linked List Allocation
  - Assume we know the block number of the first block in root dir
  - 1 read for the directory entry of home/ inside of /
  - 1 read for the directory entry of me/ inside of /home/
  - 1 read for the directory entry of .bashrc inside of /home/me/
  - 5 reads to get and read the 5th block of the file

File System Block Allocation

- Consider that we want to read the 5th block of the file **`/home/me/.bashrc`**, what is the worst-case number of disk blocks that must be read in for each of the following:
  - You can assume a block is 4096 bytes
  - assume that directory entries we are looking for are in the firs block of each directory we search
- Linked List Allocation via FAT
  - Assume we know where the root directory starts in the FAT.
  - You can also assume a FAT entry is 2 bytes.
  - 1 read for the directory entry of home/ inside of /
  - 1 read for the directory entry of me/ inside of /home/
  - 1 read for the directory entry of .bashrc inside of /home/me/
  - 1 read to get and read the 5th block of the file

File System Block Allocation

- I-nodes
  - assume we know where the I Node for the root directory is
  - 1 read for the directory entry of home/ inside of /
  - 1 read for the inode for /home/
  - 1 read for the directory entry of me/ inside of /home/
  - 1 read for the inode for /home/me/
  - 1 read for the directory entry of .bashrc inside of /home/me/
  - 1 read for the inode of .bashrc
  - 1 read to get and read the 5th block of the file

  - 7 disk reads

File System Block Allocation

- How does the numbers change if we instead wanted to write to the 5th block of the file?

- Despite not having the best numbers, I nodes are still chosen over FAT. Why is this the case?

File System Block Allocation

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

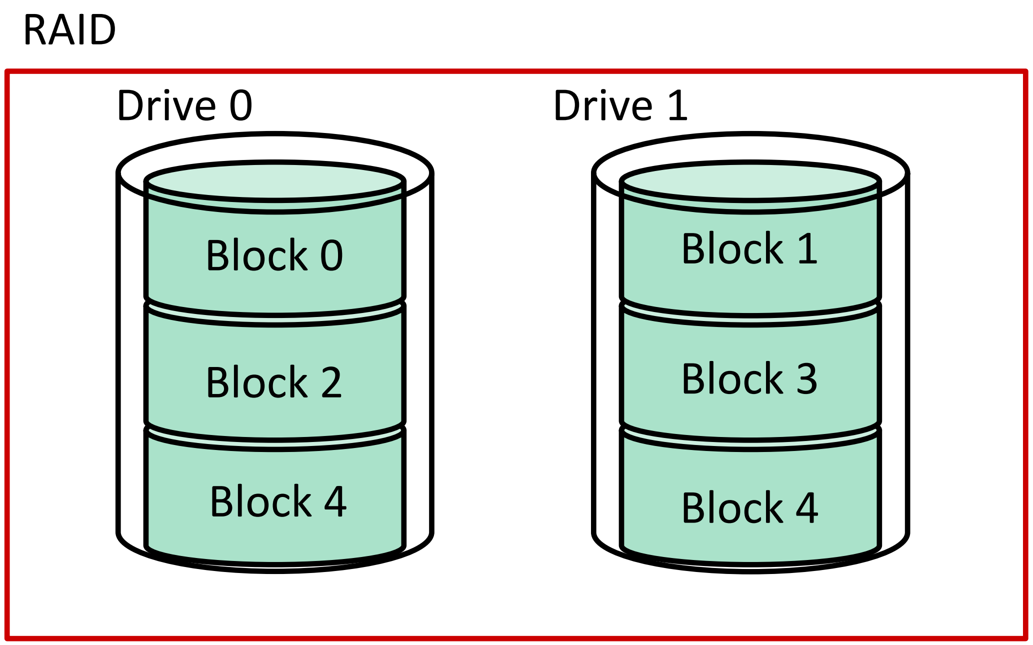
- Despite not having the best numbers, I nodes are still chosen over FAT. Why is this the case?
  - FAT takes up a lot of memory because we are caching the state of the entire filesystem in memory.
  - Inodes allow us to instead cache the information for relevant files in memory, so much lower memory consumption and similar performance for the most case.

RAID

- You are deciding between RAID 0, RAID 1, RAID 4 and RAID 5 for a system you are working on.
  - Assume we have 10 Disks available to us and a parity to data ratio of 1:4.
- Which RAID level allows for the most possible parallel reads? Which one provides the least? Why?
- What if we wanted to see which RAID level provides the most parallel writes? Which one provides the least? Why?
- **Note: We assumed all reads/writes are to 1 data block**

RAID Level 0

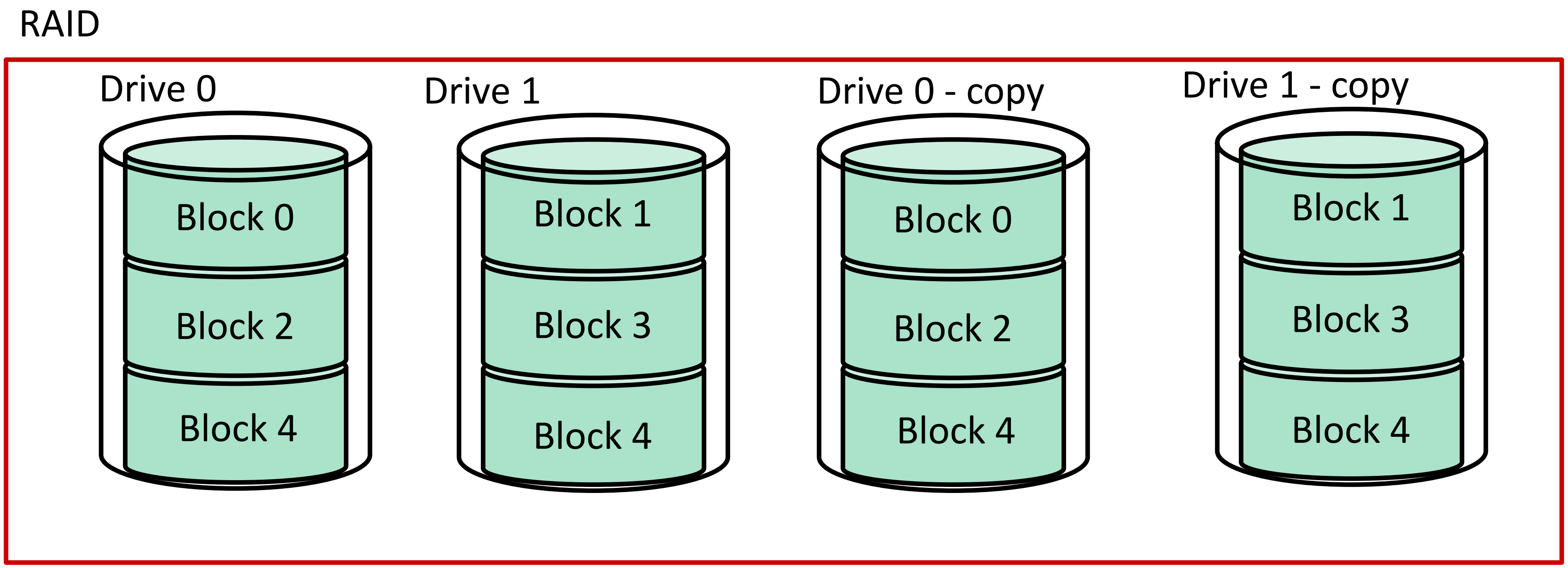
- Array of Disks with block-level striping
- If we have a file that spans multiple blocks, we can split those blocks across our array of drives



Can be more than 2 disks, just keeping it small for slides

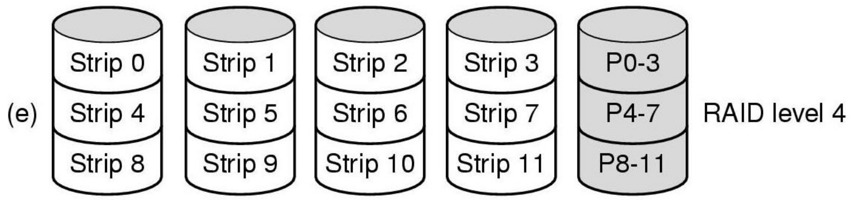
RAID Level 1

- Do the same thing as RAID 0, but each drive has a duplicate "backup" drive



RAID 4

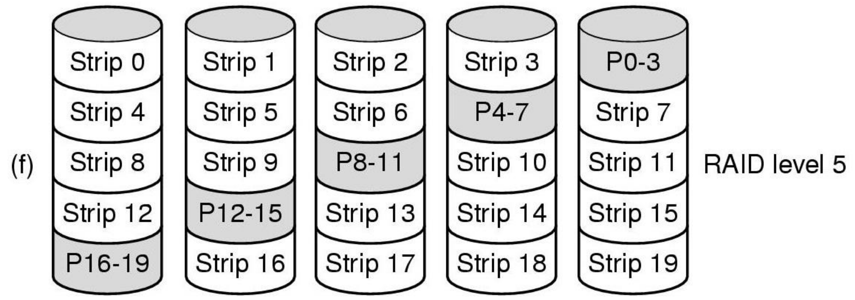
- RAID 4 is like RAID 0, but we have an extra disk dedicated to storing the parity (which you just sorta calculated!)



- In this example:
  - the blocks dedicated to storing parity are sharded.
  - P0-3 is the XOR of strip 0, 1, 2, and 3.
- We only need to have 1 disk dedicated to error recovery
- **If one disk fails, we can recalculate it by using the parity and the data of the other disks**

RAID 5

- Same as RAID 4, but we distribute the parity across different disks.



- Why?
  - Whenever we write to a block, we must also update the parity. In RAID 4, this means every write had to go also go through the parity disk, which means we can't parallelize write requests :/
  - In RAID 5, we distribute the parity so that the workload of managing parity is spread across all disks.

RAID

- You are deciding between RAID 0, RAID 1, RAID 4 and RAID 5 for a system you are working on.
  - Assume we have 10 Disks available to us and a parity to data ratio of 1:4.
- Which RAID level allows for the most possible parallel reads? Which one provides the least? Why?
  - Best: RAID 0 or RAID 1
  - Worst: RAID 4
- What if we wanted to see which RAID level provides the most parallel writes? Which one provides the least? Why?
  - Best: RAID 0
  - Worst: RAID 4

RAID

- Suppose that we took RAID 5 and had parity blocks per stripe instead of 1. Each of the two parity blocks use a different algorithm to calculate them in such a way that makes the system tolerant to two disk failures instead of 1.

  What are two downsides of this model compared to RAID level 5?

RAID

- Suppose that we took RAID 5 and had parity blocks per stripe instead of 1. Each of the two parity blocks use a different algorithm to calculate them in such a way that makes the system tolerant to two disk failures instead of 1.

  What are two downsides of this model compared to RAID level 5?

  - Increased space spent for storing parity
  - Slower writes, need to write to more blocks in the file system and need to calculate parity twice

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);
  }
}
```