Midterm Review

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Upcoming Due Dates

- HW2 (Threads)
  - Released
  - Due after spring break

- Midterm
  - In person Wednesday Evening 7-9 pm in Towne 100
  - Next lecture will be dedicated to last minute review

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 list some of the functions that may be useful and a brief summary of what the function does)

- 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**
  - **We recommend going through the course material. Lecture polls, recitation worksheets, and the previous homeworks.**

Review Topics

- C++ Programming
- Concurrency & Threads
- Scheduling
- Processes vs Threads
- Memory Hierarchy & Locality

C++ Programming

- Implement the function filter() which takes in a vector of integers and a set of integers. The function returns a new vector that contains all of the integers of the input vector, except for any elements that were in the set.

- For example, the following code should print
  - 4
  - 5

```
vector<int> v {3, 4, 5};
set<int> s {3, 6};

auto res = filter(v, s);

for (auto& num : res) {
  cout << num << endl;
}
```

C++ Programming

```
vector<int> filter(const vector<int>& numbers
                   const set<int>& omit) {

  vector<int> result{};

  for (const auto& num : numbers) {
    if (!omit.contains(num)) {
      result.push_back(num);
    }
  }
  return result;
}
```

Concurrency

- There are at least 4 bad practices/mistakes done with locks in the following code. Find them.
  - Assume `g_lock` and `k_lock` have been initialized and will be cleaned up.
  - Assume that these functions will be called by multi-threaded code.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

Concurrency

- k++ could have a data race on it
- k_lock is uncessarily used around a+=b
- g_lock is used when k_lock should be used
- cin >> c does not need to be locked, could cause significant delays.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

Threads & Mutex

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

```
1  // global variables
2  pthread_mutex_t lock;
3  int g =    0;
4  int k = 0;
5
6  void fun1() {
7    pthread_mutex_lock(&lock);
8    g += 3;
9    pthread_mutex_unlock(&lock);
10   k++;
11 }
12
13 void fun2(int a, int b) {
14   g += a;
15   a += b;
16   k = a;
17 }
18
19 void fun3() {
20   pthread_mutex_lock(&lock);
21   g = k + 2;
22   pthread_mutex_unlock(&lock);
23 }
```

- Thread-1 executes line 8 while Thread-2 executes line 21. Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 15 while Thread-2 executes line 15. Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

Threads & Mutex

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

```
1  // global variables
2  pthread_mutex_t lock;
3  int g =     0;
4  int k = 0;
5
6  void fun1() {
7    pthread_mutex_lock(&lock);
8    g += 3;
9    pthread_mutex_unlock(&lock);
10   k++;
11 }
12
13 void fun2(int a, int b) {
14   g += a;
15   a += b;
16   k = a;
17 }
18
19 void fun3() {
20   pthread_mutex_lock(&lock);
21   g = k + 2;
22   pthread_mutex_unlock(&lock);
23 }
```

- Thread-1 executes line 8 while Thread-2 executes line 21. Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 15 while Thread-2 executes line 15. Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

Threads & Mutex

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

```
1   // global variables
2   pthread_mutex_t lock;
3   int g =    0;
4   int k = 0;
5
6   void fun1() {
7     pthread_mutex_lock(&lock);
8     g += 3;
9     pthread_mutex_unlock(&lock);
10    k++;
11  }
12
13  void fun2(int a, int b) {
14    g += a;
15    a += b;
16    k = a;
17  }
18
19  void fun3() {
20    pthread_mutex_lock(&lock);
21    g = k + 2;
22    pthread_mutex_unlock(&lock);
23  }
```

- Thread-1 executes line 8 while Thread-2 executes line 14
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

Threads & Mutex

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

```
1  // global variables
2  pthread_mutex_t lock;
3  int g =    0;
4  int k = 0;
5
6  void fun1() {
7    pthread_mutex_lock(&lock);
8    g += 3;
9    pthread_mutex_unlock(&lock);
10   k++;
11 }
12
13 void fun2(int a, int b) {
14   g += a;
15   a += b;
16   k = a;
17 }
18
19 void fun3() {
20   pthread_mutex_lock(&lock);
21   g = k + 2;
22   pthread_mutex_unlock(&lock);
23 }
```

- Thread-1 executes line 8 while Thread-2 executes line 14
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

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

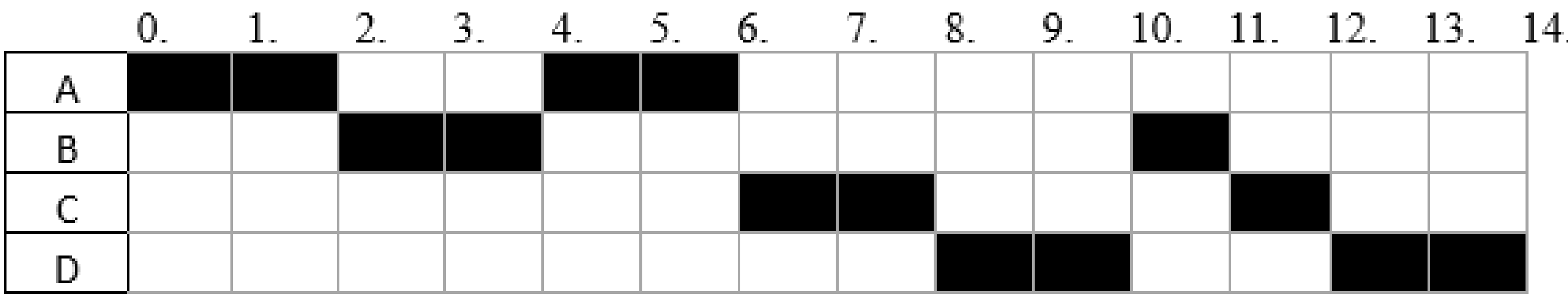
- There is an issue of inefficient CPU utilization going on in this code. What is it and how can we fix it?

```
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 = "";
  }
}
```

- (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

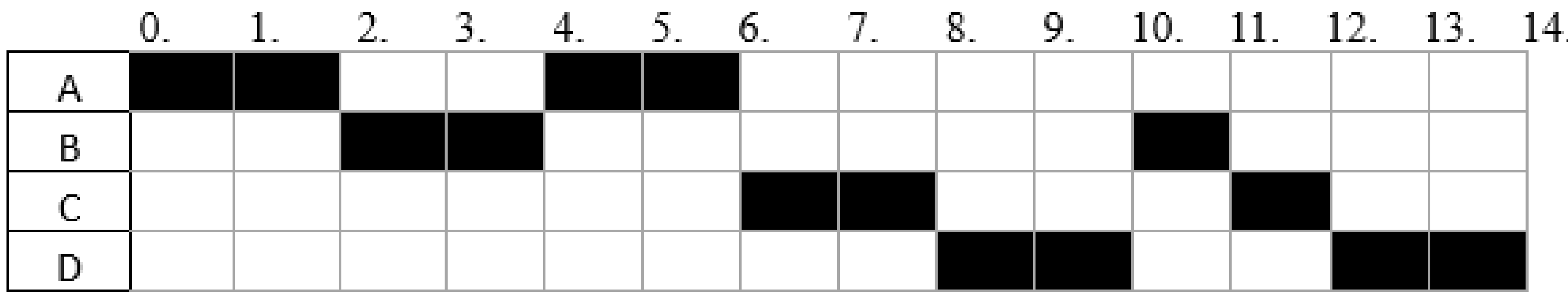
Scheduling

- 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.

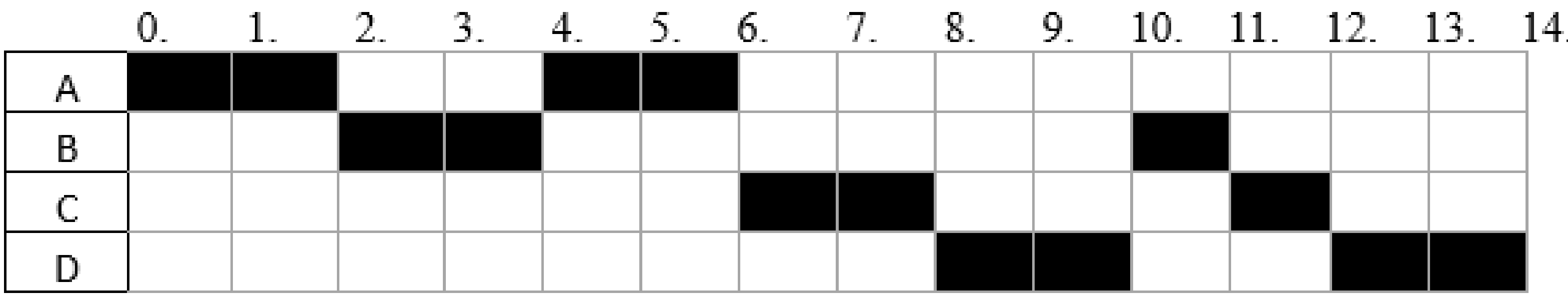
Scheduling



- 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?

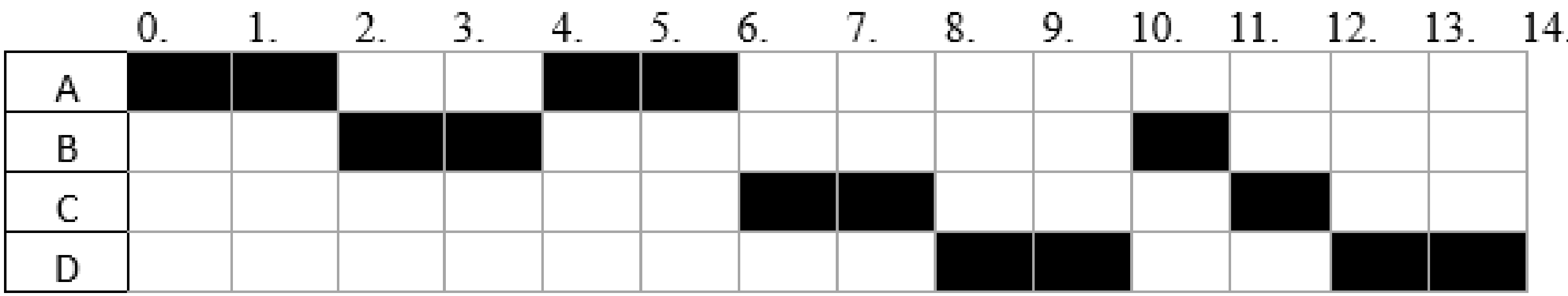
Scheduling



- 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

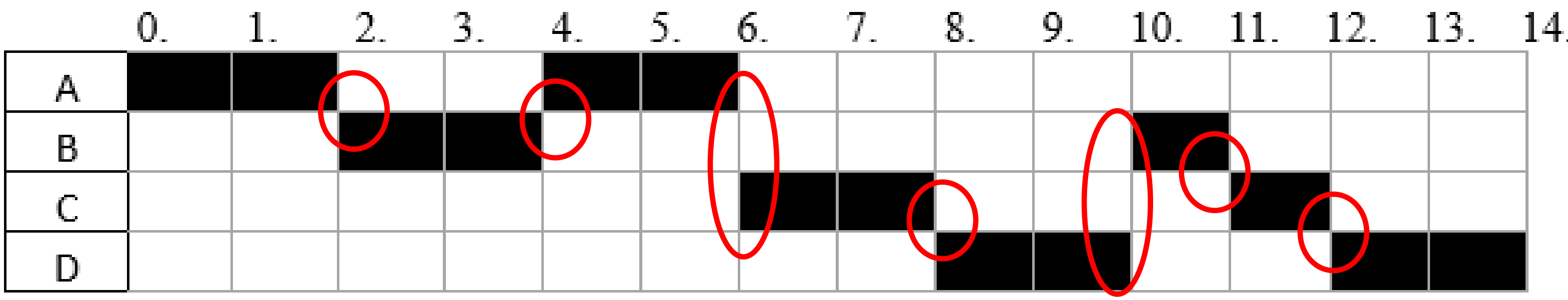


- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

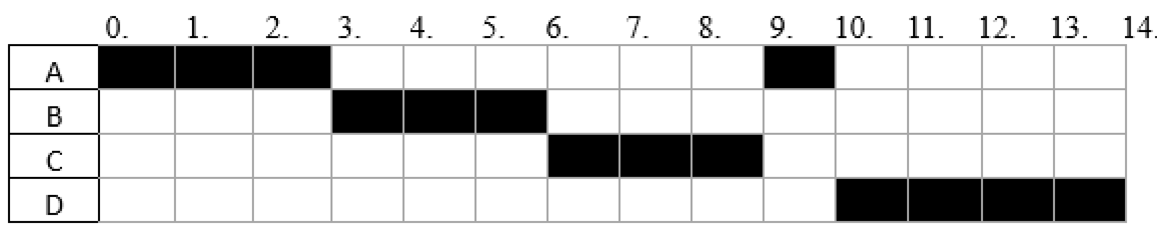
❖ Which processes are in the ready queue at time 9?

- 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

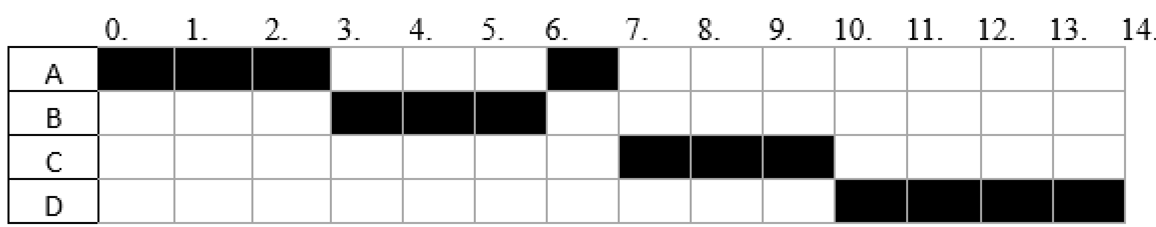


- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:



Depends on if C shows up at time 3 or 4

  - Or:



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

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?

- Let’s say that the code we wanted to parallelize was faulty and sometimes had the chance to crash. If we wanted to parallelize still but minimize the effects of program crashes, which would we choose and 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

- Let’s say that the code we wanted to parallelize was faulty and sometimes had the chance to crash. If we wanted to parallelize still but minimize the effects of program crashes, which would we choose and why?
- We would choose fork since processes have more isolation between them. If one process crashes, the others will continue to run (unless something really really bad happens). If one thread crashes, all the threads in that process will crash.

Caches Q1

- 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 Q1

- 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

Roughly every other time we access a point struct, it will already be in the cache. The other 50% of the time, it needs to be fetched from memory

Caches Q2

- 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?

Caches Q2

- 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?

Change point to just be:

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

Then Store two arrays:

```
point arr1[100];
color arr2[100];
// point at index I
// has color arr2[i]
```

Each time we access a point, we can now load 4 points into the cache. We now get ~25 cache misses and 75 hits