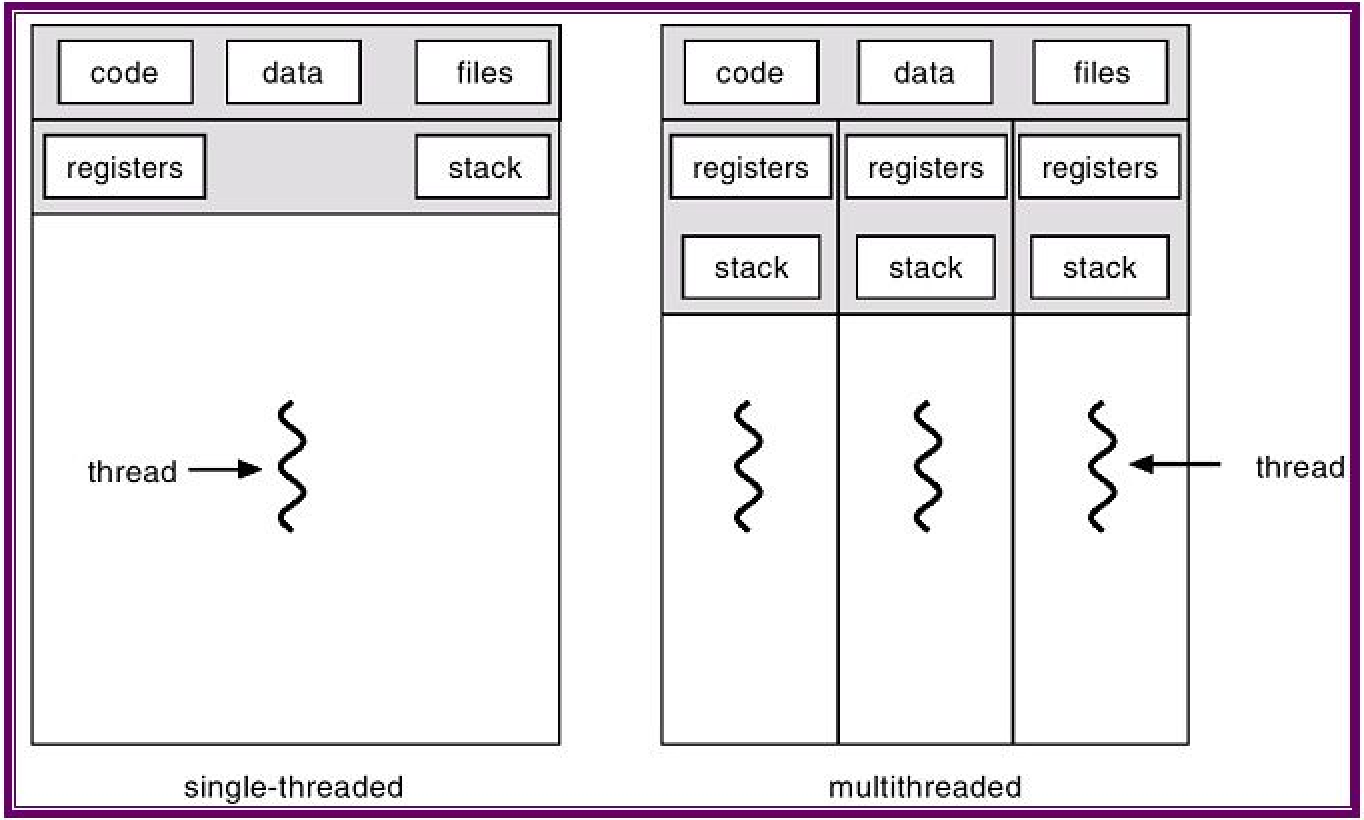
Tutorial #3

Concurrency

Forks vs Threads

- Forks create a *near* duplicate between a parent and a child, but they each have their own lives to lead and communication is like pulling teeth
    - I.e. Shared Memory
- However, *threads* share code, variables/data, and files.
    - Which means, they share information and they can *affect* each other
    - So sometimes, we need to prevent that using **semaphores and mutexes.**
- To compile a gcc program to include mutexes and semaphores:
    - gcc file.c -o prog **-lpthread -lrt**
    - **#include <pthread.h>**
    - **#include <semaphore.h>**


code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

In kindergarten they told us to share ...

THREAD 1

```
a = data;
a++;
data = a;
```

THREAD 2

```
b = data;
b--;
data = b;
```

If data = 0 before either thread starts, what is it after both threads finish?

- Who knows? It could be either -1, 0, 1 depending on the execution sequence
- This is referred to as **non-deterministic or race condition.**

You need to **protect the critical section** of code. In other words, you need to keep the data **atomic.**

Race Condition Problem

- The **race condition problem** is when an unprotected (no mutex/semaphore/monitor) shared piece of code results in a non-deterministic result.
  - Depending on type of system, a race condition is not necessarily a *bug*
  - It depends on what the programmer *intends* for the system to do
- Deterministic system
  - If you run the system a 100 times with the exact same parameters, it will return the same result all 100 times
- Non-deterministic system
  - If you run the system a 100 times it will do whatever it wants whenever it wants and there is no way to know ahead of time what the results will be.
- So how do we solve it?

… mutexes!   Using semaphores

THREAD 1

```
sem_wait(&mutex);
a = data;
a++;
data = a;

sem_post(&mutex);
```

THREAD 2

```
sem_wait(&mutex);
b = data;
b--;
data = b;

sem_post(&mutex);
```

If data = 0 before either thread starts, what is it after both threads finish?

- 2

… mutexes!   Using pthread_mutex

THREAD 1

```
_lock(&mutex);
a = data;
a++;
data = a;

_unlock(&mutex);
```

THREAD 2

```
_lock(&mutex);
b = data;
b--;
data = b;

_unlock(&mutex);
```

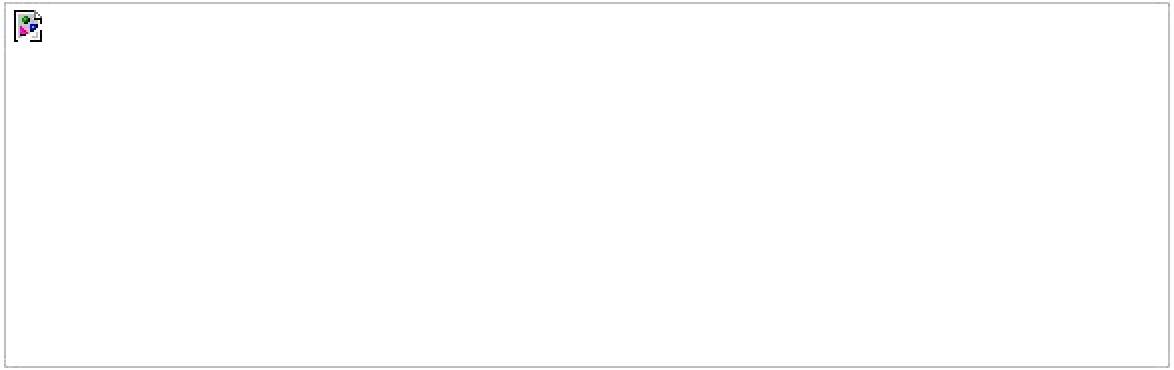
If data = 0 before either thread starts, what is it after both threads finish?

- 2

Relationships

- So, how do we know when and where to use mutexes?
- In concurrency programming, time is a bad indicator when stuff happens so we use *relationships* to describe concurrent models
    - **This can be done using state machines.**
    - These relationships must be hard-encoded into the code, so it's a good idea to model these relationships before you start coding!
        - Note: Tutorial only material will not appear on tests
- Helps avoid the perils of:
    - Starvation
    - Deadlock

This is so easy!



So, let's make a simple reader/writer for our 3 threads/processes/clients to call.

What will happen when P1 runs read()?

- Any idea what is in the buffer?
- Nothing we can consume :(

So, we have to make sure that a write has happened first!

```
int read() {
	return buffer;
}
```

```
void write(int val) {
	buffer = val;
}
```

Example State Machine

That was easy!

In our model of the system, we are going to ensure that we write() before a read()

- Assumption is made that buffer_mutex is locked at startup

What will happen when P1 runs read()?

- We get a 5

```
int read() {
   pthread_mutex_lock(&buffer_mutex);
   return buffer;
}


void write(int val) {
   buffer = val;
   pthread_mutex_unlock(&buffer_mutex);
}
```

Not so simple ...

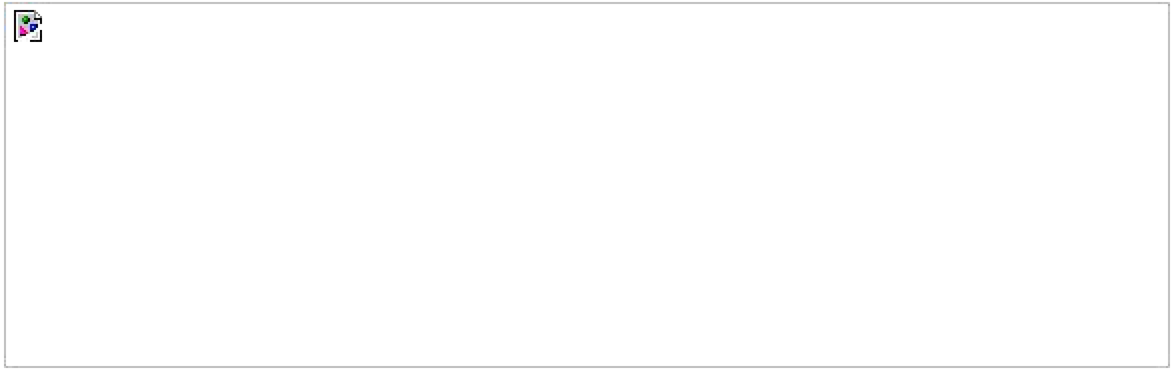
There, now the read has to wait for a write to unlock the mutex!

What happens if I do a write, read, read?

- No problem, I just wait!

Be careful when using locks that there aren’t conditions where valid sequence of operations returns a locked state.

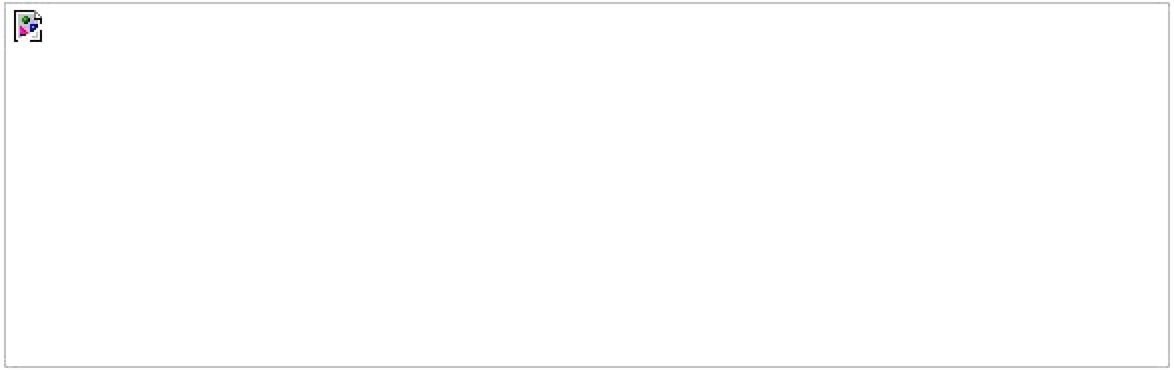
Let’s check if the value is null then ...



```
int read() {
   pthread_mutex_lock(&buffer_mutex);
   return buffer;
}
```

```
void write(int val) {
   buffer = val;
   pthread_mutex_unlock(&buffer_mutex);
}
```

Yet another wrinkle



What about now?

- Um ….. ?

This is a **race condition.** Get used to them, they happen a lot in concurrent systems.

Note: RAM is atomic which means that simultaneous writes can’t happen (the RAM controller stalls one CPU until the other is done).

```
int read() {
   pthread_mutex_lock(&buffer_mutex);
   return buffer;
}
```

```
void write(int val) {
   buffer = val;
   pthread_mutex_unlock(&buffer_mutex);
}
```

Let’s use a mutex!

So let’s just add in a mutex so we can’t P2 and P3 can’t write at the same time

Is the system now deterministic?

- Nope!

There is no real way to know if P2 or P3 will grab the lock first. In this example, P2 was arbitrarily chosen.

Note: The order of the unlocks() is very important!

```
int read() {
   pthread_mutex_lock(&buffer_mutex);
   pthread_mutex_unlock(&write_mutex);
   return buffer;
}


void write(int val) {
   pthread_mutex_lock(&write_mutex);
   buffer = val;
   pthread_mutex_unlock(&buffer_mutex);
}
```

Let’s use a mutex!

What about simultaneous read/writes?

Race condition! Let’s stick in another lock!

- Not so fast, remember what we did earlier?

In our system, because we have the relationship *write->read* this is deterministic.

So both P1 will get back 5, and P3 will wait for the next write.

```
int read() {
   pthread_mutex_lock(&buffer_mutex);
   pthread_mutex_unlock(&write_mutex);
   return buffer;
}


void write(int val) {
   pthread_mutex_lock(&write_mutex);
   buffer = val;
   pthread_mutex_unlock(&buffer_mutex);
}
```

What else could go wrong?

What happens if I want to do something with a value I read?

- Current code does not prevent this from happening
- This is called ***stale data.***

This is part of the **critical-section problem.**

```
int read() {
   pthread_mutex_lock(&buffer_mutex);
   pthread_mutex_unlock(&write_mutex);
   return buffer;
}


void write(int val) {
   pthread_mutex_lock(&write_mutex);
   buffer = val;
   pthread_mutex_unlock(&buffer_mutex);
}
```

Critical Section Problem

The critical-section problem:

- Code section accessing **shared data**
- Only **one thread executing** in critical section
  - Only one thread accessing the shared data: serialize
- **Choose the right (size of) critical section**

Properties of the critical-section problem:

- Mutual exclusion
  - No more than one process in the critical section
- Making progress
  - If no process in the critical section, one can come in
- Bounded waiting
  - For processes that want to get in the critical section, their waiting time is bounded

I hate concurrent systems ...

Remember before what I said about modelling concurrent systems?

- Make sure you know how your system is supposed to behave
- **Be careful selecting your critical sections**

One possible solution is to create another mutex for the **add** operation.

In more complicated programs (like Task 2), better way is to use **conditional variables.**

```
void add(int inc_by) {
        pthread_mutex_lock(&count_mutex);
        /* start critical section */
        int val = read();
        write(val + inc_by);
        /* end critical section */
        pthread_mutex_unlock(&count_mutex);
}
```

Condition Variables

- Synchronization mechanisms need more than just mutual exclusion; also need a way to wait for another thread to do something (e.g., wait for a character to be added to the buffer)
- *Condition variables*: used to wait for a particular condition to become true (e.g. characters in buffer).
  - `pthread_cond_wait(condition, lock)`: release lock, put thread to sleep until `condition` is signaled; when thread wakes up again, re-acquire lock before returning.
  - `pthread_cond_signal(condition, lock)`: if any threads are waiting on `condition`, wake up one of them. Caller must hold `lock`, which must be the same as the `lock` used in the `wait` call.
  - `pthread_cond_broadcast(condition, lock)`: same as `signal`, except wake up all waiting threads.
  - Note: after `signal`, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.
  - Warning: when a thread wakes up after `condition_wait` there is no guarantee that the desired condition still exists: another thread might have snuck in.

A very good step-by-step breakdown of threads and condition variables can be found here:

- http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf