Agenda for Today

1. Lab Q&A
2. New assignment coming soon
3. Concurrency Bugs
   a. Non-Deadlock Bugs
   b. Deadlock
4. Wrap Up
Password Cracker Lab Q&A

Do we have to use locks in our solution? What do we need to protect with locks?
Maybe, but probably not. You need to use locks when there is a possibility that, while one thread is writing a value, another thread could try to read or write that value. You have to use some judgement here, but hopefully today’s class helps with this question.
Password Cracker Lab Q&A
Password Cracker Lab Q&A
Concurrency Bugs Assignment

*(posted later today)*
Concurrency Bugs
Types of Concurrency Bugs

Discuss with your group to come up with a list of the types of concurrency bugs we've read about. Be prepared to share your list and a definition of each.

Atomicity Violations
Two threads are running. The programmer expected a block of code to run in one thread uninterrupted, but in reality the other thread can observe state in the middle of that block.

Order Violations
The programmer expects one part of a thread to execute before another one does, but there's nothing in place to make sure this actually happens. A common cause of order violations is initialization or thread creation.

Deadlock
Multiple threads are waiting for each other, so the program can’t make progress.
Types of Concurrency Bugs

What is a race condition?

We can think about program execution as a sequence of events, some are ordered, and some are not. Within one thread, operations are totally-ordered: the first iteration of a loop runs before the second, which runs before the third, etc.

With concurrent programs, operations that run in different threads are often unordered. There’s nothing forcing those statements to run in any particular order. That’s good—we use threads specifically because we want things to happen at the same time.

A race condition occurs when two (or more) statements are unordered, but touch the same shared state, with at least one thread changing that state. At the end of these two statements, you don’t know the possible outcome.

Ordering thread creation, using locks/semaphores/condition variables allows you to impose ordering on statements that access shared state.
An Example Program
A Flag-Drawing Program

```c
int row = 0;
int color = RED;

int main() {
    pthread_t ts[3];
    for(int i=0; i<3; i++) {
        pthread_create(&ts[i], NULL, worker, NULL);
    }
    for(int i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* worker(void* p) {
    int my_row = row;
    row++;
    int my_color = color;
    if(color == RED) color = WHITE;
    else if(color == WHITE) color = BLUE;
    for(int col=0, col<4, col++) {
        put_color(my_row, col, my_color);
    }
    return NULL;
}
```
int row = 0;
int color = RED;

int main() {
    pthread_t ts[3];
    for(int i=0; i<3; i++) {
        pthread_create(&ts[i], NULL, worker, NULL);
    }

    for(int i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* worker(void* p) {
    int my_row = row;
    row++;

    int my_color = color;
    if(color == RED) color = WHITE;
    else if(color == WHITE) color = BLUE;

    for(int col=0, col<4, col++) {
        put_color(my_row, col, my_color);
    }
    return NULL;
}
What went wrong?

What was the concurrency bug caused the incorrect output?
Two threads may be assigned the same row in the output. One thread may assign itself row zero, then before updating the global row, another thread could also assign itself row zero.

This is an atomicity violation: the read of the global row and its update aren’t atomic.

How can we fix the bug?
We should acquire a lock before reading the row, and release that lock right after we increment the global row variable.

Or: instead of assigning the row with a global, pass it in with a thread parameter.
A Better Flag-Drawing Program

int color = RED;

int main() {
    pthread_t ts[3];
    int rows[3];
    for(int i=0; i<3; i++) {
        rows[i] = i;  // Assign row number here
        pthread_create(&ts[i], NULL, worker, rows[i]);
    }

    for(int i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* worker(void* arg) {
    int my_row = *(int*)arg;  // Get assigned row
    int my_color = color;
    if(color == RED) color = WHITE;
    else if(color == WHITE) color = BLUE;
    for(int col=0, col<4, col++) {
        put_color(my_row, col, my_color);
    }
    return NULL;
}
What went wrong?

```c
int color = RED;

int main() {  
    pthread_t ts[3];  
    int rows[3];  
    for(int i=0; i<3; i++) {  
        rows[i] = i;  
        pthread_create(&ts[i], NULL, worker, rows[i]); 
    }
    for(int i=0; i<3; i++) {  
        pthread_join(threads[i], NULL);  
    }
}

void* worker(void* arg) {  
    int row = *(int*)arg;  

    int my_color = color;  
    if(color == RED) color = WHITE;  
    else if(color == WHITE) color = BLUE;  
    for(int col=0, col<4, col++) {  
        put_color(my_row, col, my_color);  
    }  
    return NULL;  
}
```
What went wrong?

What was the concurrency bug caused the incorrect output?

This is an atomicity violation. The thread reads and changes the color, but a thread could be preempted (or interrupted) right after reading the color but before assigning a new.

How can we fix the bug?

Pretty much the same as the last one. Make sure each thread gets the appropriate color by wrapping that code in a lock. That will prevent other threads from jumping in and assigning a duplicate color before the global has been updated.
A Better Flag-Drawing Program

```c
int color = RED;
pthread_mutex_t color_lock = PTHREAD_MUTEX_INITIALIZER;

int main() {
    pthread_t ts[3];
    int rows[3];
    for(int i=0; i<3; i++) {
        rows[i] = i;
        pthread_create(&ts[i], NULL, worker, rows[i]);
    }

    for(int i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* worker(void* arg) {
    int row = *(int*)arg;

    pthread_mutex_lock(&color_lock);
    int my_color = color;
    if(color == RED) color = WHITE;
    else if(color == WHITE) color = BLUE;
    pthread_mutex_unlock(&color_lock);

    for(int col=0; col<4; col++) {
        put_color(my_row, col, my_color);
    }
    return NULL;
}
```
What went wrong?

```c
int color = RED;
pthread_mutex_t color_lock = PTHREAD_MUTEX_INITIALIZER;

int main() {
    pthread_t ts[3];
    int rows[3];
    for(int i=0; i<3; i++) {
        rows[i] = i;
        pthread_create(&ts[i], NULL, worker, rows[i]);
    }
    for(int i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* worker(void* arg) {
    int row = *(int*)arg;
    int my_color = color;
    pthread_mutex_lock(&color_lock);
    if(color == RED) color = WHITE;
    else if(color == WHITE) color = BLUE;
    pthread_mutex_unlock(&color_lock);
    for(int col=0; col<4; col++) {
        put_color(my_row, col, my_color);
    }
    return NULL;
}
```
What went wrong?

What was the concurrency bug caused the incorrect output?

This is an order violation. Each thread assigns itself a color, but that order is not fixed. The code explicitly assigns a row to each thread, but there’s nothing in the code that forces row 0 to be red, row 1 to be white, and row 2 to be blue.

How can we fix the bug?

We could modify the thread argument to specify both a color and row. Lots of possible encoding schemes. The main idea is to link the row number with the color instead of relying on that link happening by chance.
int main() {
    pthread_t ts[3];
    int rows[3];
    for(int i=0; i<3; i++) {
        rows[i] = i;
        pthread_create(&ts[i], NULL, worker, rows[i]);
    }
    for(int i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* worker(void* arg) {
    int row = *(int*)arg;
    int my_color;
    if (row == 0) my_color = RED;
    else if (row == 1) my_color = WHITE;
    else my_color = BLUE;
    for(int col=0, col<4, col++) {
        put_color(my_row, col, my_color);
    }
    return NULL;
}
Deadlock
Deadlock

What is deadlock?

Two (or more) threads are waiting for resources held by other waiting threads. The threads are all waiting and can never make progress.

What are the four causes of deadlock? *For each cause, how could we prevent it?*

**Mutual Exclusion**

A thread can claim exclusive control over some resource (like a lock). We could avoid this by not using locks. An alternative fix is to use optimistic concurrency control (transactions) instead of locks.
Deadlock

Circular Waiting
There is a chain of threads holding resources, and waiting for each other in a cycle.
Fix: impose a strict order on locking. For example, if there are two locks A and B, we can only have a circular wait if one thread locks A then B, while the other locks B then A. If we say that lock A must be acquired before lock B, a circular wait is impossible.

No Preemption
Locks can’t be revoked from threads that hold them. If we could do that, we could detect and escape a deadlock situation by taking back some thread’s lock.

Hold and Wait
A thread can hold one resource while it waits for another.
Fix: Acquire all locks at once instead of one at a time.
Another fix: Use condition variables. When you wait for a condition variable, you give it a lock to release while you’re waiting.
Wrap Up
Reminders

MAP Applications Due Today
If you're applying for a MAP, submit the form today!
Some MAPs require additional forms, so read the descriptions carefully.

CS Advisor Preferences Due Today
If you are declaring a CS major this spring, fill out this form before 5pm today
https://grinnell.co1.qualtrics.com/jfe/form/SV_5t0rLdZPOfbzk8e

Read for class
Learn about CUDA and GPU programming to prepare for in-class exercises