Agenda for Today

1. Department Events
2. Q&A
3. Concurrency Bugs
4. Wrap Up
Department Events

**CS Table** noon on *Tuesday, November 8th*
Digital Redlining (watch for an email today)

**US Midterm Election**  *Tuesday, November 8th (no extra credit)*
Tuesday, November 8th is the last day to vote in the US Midterm Election. If you are eligible to vote, please do so!

**CS Extra**  *4pm on Thursday, November 10th*
Stay tuned for an announcement
How does \texttt{__syncthreads\_count} work?

All the threads in a block will reach the \texttt{__syncthreads\_count} call before any can move on. That's the same as normal \texttt{__syncthreads()). The extra feature is that each thread passes a value (called "predicate") to \texttt{__syncthreads\_count}. When \texttt{__syncthreads\_count} returns, it returns the same value to all threads in the block. That value is the number of threads that passed a non-zero predicate to \texttt{__syncthreads\_count}. 
Concurrency Bugs
Types of Concurrency Bugs

The reading described three types of concurrency bugs. What are they, and how do they occur?

Order Violation
Operations should be in order, but we didn't ensure that would happen.

Atomicity Violation
Two or more operations should happen uninterrupted, but do not.

Deadlock
Threads are waiting indefinitely, and can never make progress.
An Example Program
A Flag-Drawing Program

```cpp
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?

```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);
    }
}
```

```c
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;
}
```
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;
        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;
    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(row, col, my_color);
  }
  return NULL;
}
```
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(row, col, my_color);
    }
    return NULL;
}
```
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(row, col, my_color);
    }
    return NULL;
}
One Possible Fix

```c
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 are the four causes of deadlock? How can we prevent each cause?

**No Preemption:** *We can't take a lock away from a thread that holds it.*

We could take locks away (after a timeout, for example)

**Cyclic Wait:** *For two threads, one holds a lock and waits for another. The second thread holds that lock, and is waiting for the one the first thread holds.*

Impose a consistent order of locks. If a thread needs multiple locks, it must acquire them in this specified order.
Deadlock

What are the four causes of deadlock? How can we prevent each cause?

*Hold and wait:* A thread can hold one lock while waiting for another. Whenever a thread is waiting, its locks are released. This is what `pthread_cond_wait` does.

*Mutual exclusion:* threads can hold locks exclusively. Just don't use locks.
Wrap Up
Reminders

The End of Daylight Saving Time
  Set your clocks back an hour on Sunday morning!

Lab
  Sudoku lab is due on Wednesday, November 9th (next week)

Reading
  Read about distributed systems and networks for Monday