Week 11 – Friday

COMP 3400

Last time

- What did we talk about last time?
- Barriers

Questions?

Project 3

Condition Variables

Weaknesses of semaphores

- Semaphores are very general purpose concurrency tool, but they have some weaknesses:
  - Semaphores take thought to use correctly: Incrementing and decrementing values don't map clearly to synchronization problems
  - Different implementations of semaphores have different features
  - Some systems (like macOS) don't have a full implementation of semaphores
  - Semaphores can only signal to one thread: no broadcasting
  - After getting a signal, threads have to take another step (like acquiring a lock) to get mutually exclusive access, time that can allow a race condition

Condition variables

- **Condition variables** try to overcome some weaknesses of semaphores by tying themselves directly to a lock
- They also have the ability to broadcast, waking up all waiting threads
- Like semaphores, they still have a function to wait and a function to signal
- However, something sneaky happens with wait:
  - First, the thread must acquire a lock
  - Then, it calls the wait function
  - If it has to wait, it releases the lock but then reacquires it when it gets woken up
  - All of which happens atomically
- This allows a thread to safely check a condition and wait until it gets signaled
- Think of a condition variable as a queue for waiting threads

Condition variable functions

```
int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_condattr_t *attr);
```

- Initialize a condition variable

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

- Release a mutex, wait for the signal, then re-acquire the mutex

```
int pthread_cond_signal (pthread_cond_t *cond);
```

- Send a signal to one waiting thread to wake up

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

- Send a signal to all waiting threads to wake up

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

- Clean up the resources associated with a condition variable

Using a condition variable

- To use a condition variable correctly, the thread has to acquire a lock before calling **`pthread_cond_wait()`** (which releases the lock)
  - Calling **`pthread_cond_wait()`** without getting the lock has undefined behavior
- Calls to **`pthread_cond_wait()`** should be inside a while loop
  - Sometimes threads are incorrectly woken up and should check before moving on
- Calling **`pthread_cond_signal()`** or **`pthread_cond_broadcast()`** doesn't wake up threads on its own
  - Later calling **`pthread_mutex_unlock()`** is what actually allows those threads to run again

Parallels to Java

- If you have done concurrent programming in Java, these ideas of condition variables have been integrated into the syntax in a cleaner way
  - Executing code in a **`synchronized`** method or **`synchronized`** block acquires a lock
  - Calling **`wait()`** (which can only be done in **`synchronized`** code) is the same as calling **`pthread_cond_wait()`**
  - Calling **`notify()`** is the same as calling **`pthread_cond_signal()`**
  - Calling **`notifyAll()`** is the same as calling **`pthread_cond_broadcast()`**

Java example

```java
public class Buffer {
    public final static int SIZE = 10;
    private volatile Object[] objects = new Object[SIZE];
    private volatile int count = 0;

    public synchronized void addItem(Object object) throws InterruptedException {
        while(count == SIZE)
            wait();
        objects[count] = object;
        count++;
        notifyAll();
    }

    public synchronized Object removeItem() throws InterruptedException {
        while(count == 0)
            wait();
        count--;
        Object object = objects[count];
        notifyAll();
        return object;
    }
}
```

Explanation of the Java example

- Syntax is a little cleaner looking in Java
- The **`synchronized`** methods work like they have a lock at the beginning and end
- Calling **`wait()`** waits until a **`notify()`** or **`notifyAll()`** happens
- This example shows a **`Buffer`** where items can be added or removed only by acquiring the lock (implicit in calling a **`synchronized`** method)
- Because the array has fixed length, only so many things can be added before it gets full
- That's why the **`addItem()`** will repeatedly call **`wait()`** until there's room and the **`removeItem()`** will repeatedly call **`wait()`** if there's nothing there

Deadlock

Deadlock

- In order to avoid race conditions, we introduced several synchronization tools:
  - Locks (mutexes)
  - Semaphores
  - Barriers
  - Condition variables
- Each of these can be misused, failing to avoid race conditions
- Likewise, each introduces overhead, slowing the system down
- But an even worse possibility is **deadlock**

Deadlock

- Deadlock occurs when the use of synchronization primitives cause threads to get stuck so that they will never make progress again
  - A lock that never gets unlocked
  - A semaphore that never gets posted on
  - A barrier that is never reached by enough threads
  - A condition variable that is never signaled on
- Like many concurrency problems, deadlock can occur rarely or it can happen every time a program runs

Deadlock example

- In the following code, deadlock is possible

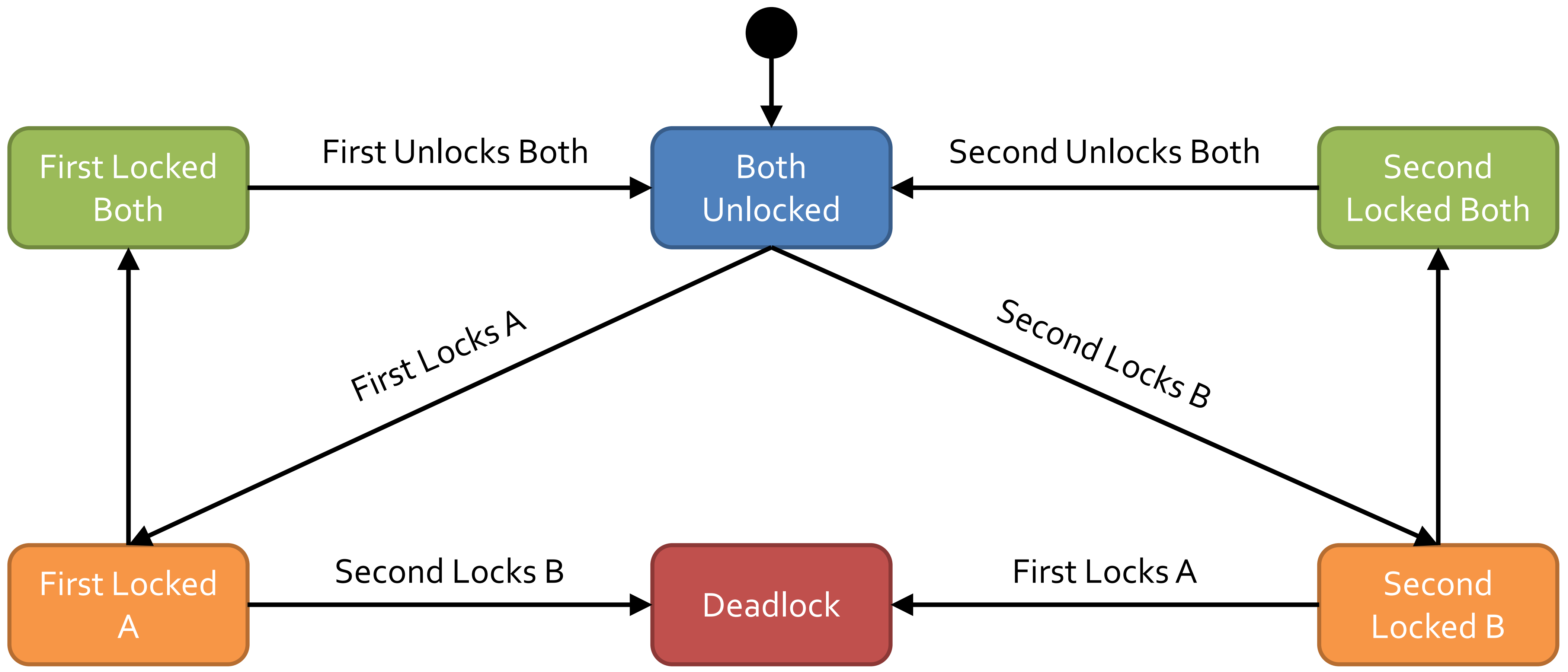
```
struct args {
  pthread_mutex_t lock_a;
  pthread_mutex_t lock_b;
};

void * first (void * args)
{
  struct args *data = (struct args *) args;
  pthread_mutex_lock (&data->lock_a); // Lock A
  pthread_mutex_lock (&data->lock_b); // Then lock B
  // Mode code (that would eventually unlock A and B)
}

void * second (void * args)
{
  struct args *data = (struct args *) args;
  pthread_mutex_lock (&data->lock_b); // Lock B
  pthread_mutex_lock (&data->lock_a); // Then lock A
  // Mode code (that would eventually unlock A and B)
}
```

Possible states

- The following state diagram shows the states the threads can be in:



Why does this happen?

- The two threads try to acquire locks in different orders:
  - First tries to get lock A followed by lock B
  - Second tries to get lock B followed by lock A
- If they tried to get the locks in the same order, we would never have this problem
- Even so, real situations are more complex
- Threads might need to acquire a number of locks for a number of resources
- The order might be hard to predict ahead of time

Necessary conditions

- Four conditions are needed for deadlock to be possible:
  1. **Mutual exclusion:** Once a resource has been acquired, no other thread gets it
  2. **No preemption:** Threads can't be made to give up their resources
  3. **Hold and wait:** Threads can get one resource and keep it while trying to get others
  4. **Circular wait:** Thread A needs a resource held by Thread B, and Thread B needs a resource held by Thread A (or extend to a chain of threads)
- Conditions 1 through 3 are unavoidable, so solutions often focus on avoiding circular wait

Livelock

- **Livelock** is similar to deadlock
- It's a condition where, due to bad timing, processes continue executing code, but they never make progress beyond a certain point
    - They're acquiring resources, giving them up, and acquiring them again, but still blocking each other
- If the system is set up in a certain way or is very unlucky, livelock could continue indefinitely
- Livelock can also sometimes resolve

Livelock example

```
struct args {
  pthread_mutex_t lock_a;
  pthread_mutex_t lock_b;
};

void * first (void * args)
{
  struct args *data = (struct args *) args;
  while (1)
  {
    pthread_mutex_lock (&data->lock_a);        // Lock A
    if (pthread_mutex_trylock (&data->lock_b)) // Try to lock B
      break;
    pthread_mutex_unlock (&data->lock_a);      // Unlock A
  }
  // Mode code (that would eventually unlock A and B)
}

void * second (void * args)
{
  struct args *data = (struct args *) args;
  while (1)
  {
    pthread_mutex_lock (&data->lock_b);        // Lock B
    if (pthread_mutex_trylock (&data->lock_a)) // Then lock A
      break;
    pthread_mutex_unlock (&data->lock_b);      // Unlock B
  }
  // Mode code (that would eventually unlock A and B)
}
```

Livelock on the previous slide

- In theory, each thread could acquire the first lock at a very similar time, making the other one fail to get the second one
- In practice, it's unlikely that this system will stay in livelock for very long
- However, real systems are more complicated and could have long chains of resources that get partially lock and unlocked but never finish

Avoiding deadlock

- As mentioned before, we usually concentrate on the circular wait condition of deadlock:
  - Order the resources and always acquire them in the same order
  - Use timed or non-blocking versions of functions that acquire resources, potentially causing livelock
  - Limit the number of threads that can access the resources, insuring that there's always enough resources to go around
  - Use strategies that we'll talk about next time
- It's a hard problem: The Java **`Thread`** class has methods that were deprecated because they can cause deadlocks

Upcoming

Next time…

- Synchronization design patterns
- Producer/consumer

Reminders

- Work on Project 3
- Read sections 8.1, 8.2, and 8.3