Week 12 – Monday

COMP 3400

Last time

- What did we talk about last time?
- Condition variables
- Deadlock
- Livelock

Questions?

Project 3

Deadlock

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
  }
  // More 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
  }
  // More code (that would eventually unlock A and B)
}
```

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 today
- It's a hard problem: The Java `Thread` class has methods that were deprecated because they can cause deadlocks

Synchronization Design Patterns

Synchronization design patterns

- As with non-concurrent code, certain practices and patterns for synchronization have a good track record of working
- Using these synchronization design patterns
  - Gives a set of choices to pick from when doing synchronization
  - Provides frameworks that are more likely to work correctly
- Design patterns we'll discuss:
  - Signaling
  - Turnstiles
  - Rendezvous
  - Multiplexing
  - Lightswitches

Signaling

- **Signaling** is a design pattern we've already discussed
  - One thread needs to wait until a event has occurred
  - A second thread signals the first
- POSIX thread implementation:
  - Initialize a semaphore to `0`
  - Have the first thread call **`sem_wait()`** on the semaphore when it needs to wait
  - Have a second thread call **`sem_post()`** when the event has occurred
- Because semaphores have an integer value, the scheduling of the threads doesn't matter
  - If the second thread has already signaled, the first thread will immediately return from **`sem_wait()`**

Turnstiles

- Unlike signaling, which unblocks a *single* thread, the **turnstile** design pattern is used to unblock *many* threads when an event occurs
- POSIX implementation:
  - Initialize a semaphore to **`0`**
  - Have a thread call **`sem_post()`** on the semaphore when the event occurs
  - All threads that need to wait call **`sem_wait()`** followed by **`sem_post()`**
  - Each thread waking up will wake up one more
- Turnstiles work similarly to the broadcast function for condition variables
  - But broadcasts will only wake up those threads that are currently waiting
  - Turnstiles let all threads pass through, even if they reach the **`sem_wait()`** after the event has already happened

Rendezvous

- In the **rendezvous** pattern, two threads signal that they have both reached a specific point in execution
- POSIX implementation:
  - Initialize semaphore A and semaphore B to **`0`**
  - Thread 1 calls **`sem_post()`** on semaphore A and **`sem_wait()`** on semaphore B
  - Thread 2 calls **`sem_post()`** on semaphore B and **`sem_wait()`** on semaphore A
- Each thread will only get blocked until the other one signals
  - Order matters! Flip the waits with the posts and you'll have deadlock
- For larger numbers of threads, using a barrier might be a better approach

Multiplexing

- **Multiplexing** is another design pattern we've already mentioned
- Multiplexing is useful when mutual exclusion is more restrictive than you need, but you still want to limit the total number of threads able to execute a section of code
- POSIX implementation:
  - Initialize a semaphore to ***n***, where ***n*** is the maximum number of concurrent accesses you want to allow
  - Each thread calls **`sem_wait()`** on the semaphore before executing the code
  - Each thread calls **`sem_post()`** on the semaphore after executing the code
- This design pattern can be useful when spawning threads on a server to handle requests
  - We want to prevent too many threads from being created in order to avoid bogging down the server

Lightswitches

- We sometimes want to allow multiple threads of a certain kind to run code concurrently but force others to use mutual exclusion
  - Many threads that only read memory, for example, could access the memory at the same time
  - But only one thread that writes memory should be allowed in
- The **lightswitch** design pattern allows this kind of access
  - The name comes from the idea that the first person into a room turns on a lightswitch and the last person turns it off
- POSIX implementation:
  - Initialize a semaphore to **`1`**
  - Initialize a counter variable to **`0`**
  - Create a lock
  - Whenever a reader thread wants to read:
    - It acquires the lock
    - Increments the counter
    - If the counter is **`1`**, call **`sem_wait()`** on the semaphore
    - Unlock the lock
  - Whenever a reader thread is done reading:
    - It acquires the lock
    - It decrements the counter
    - If the counter is **`0`**, it calls **`sem_post()`** on the semaphore
    - Unlock the lock
  - Writers simply call **`sem_wait()`** to start writing and **`sem_post()`** when done

Producer-Consumer

Producer-consumer

- The **producer-consumer problem** comes up all the time in concurrent systems
  - One or more threads is producing elements that go into a buffer
  - One or more threads is consuming elements from the buffer
- A producer can't put an item into a full buffer and must block
- A consumer can't remove an item from an empty buffer and must block
- Example:
  - An OS thread is putting data into a buffer that's coming across the network
  - A user thread is trying to read data out of that buffer

Unsafe producer-consumer with an unbounded queue

- Buffers are usually finite in size, but to make the problem simplest, we can use a linked list where producers enqueue elements and consumers dequeue elements
- The following code does so in a way that's **unsafe** for a multi-threaded application:

```
void enqueue_unsafe (queue_t *queue, data_t *data)
{
  // Create a new node and put it on the back of the queue
  queue->back->next = calloc (1, sizeof (queue_node_t));
  queue->back = queue->back->next;
  queue->back->data = data;
}

data_t *dequeue_unsafe (queue_t *queue)
{
  // If back = front, the queue is empty
  if (queue->back == queue->front)
    return NULL;
  data_t * data = queue->front->data;
  queue_node_t * next = queue->front->next;
  free (queue->front);
  queue->front = next;
  return data;
}
```

Safe producer-consumer with an unbounded queue

- We can make these operations safe by putting a lock around each one

```
void enqueue (queue_t *queue, data_t *data, pthread_mutex_t *lock)
{
  pthread_mutex_lock (lock);
  enqueue_unsafe (queue, data);
  pthread_mutex_unlock (lock);
}

data_t * dequeue (queue_t *queue, pthread_mutex_t *lock)
{
  pthread_mutex_lock (lock);
  data_t * data = dequeue_unsafe (queue);
  pthread_mutex_unlock (lock);
  return data;
}
```

Unsafe producer-consumer with a bounded queue

- We can move on to a version with a bounded buffer
- Our implementation uses a circular array (that wraps back around to the beginning)
- The following code is unsafe for two reasons:
  - It doesn't check to see if the buffer is full when enqueuing or empty when dequeuing
  - Changing queue data is **unsafe** for a multi-threaded application

```
void enqueue_unsafe (queue_t *queue, data_t *data)
{
  // Store the data in the array and advance the index
  queue->contents[queue->back++] = data;
  queue->back %= queue->capacity;
}

data_t * dequeue_unsafe (queue_t *queue)
{
  data_t * data = queue->contents[queue->front++];
  queue->front %= queue->capacity;
  return data;
}
```

Safe producer-consumer with a bounded queue and a single producer and consumer

- We could use locks and check a variable giving the total number of elements in the queue
- However, semaphores have this feature built in
- We initialize the **space** semaphore to the maximum size of the queue
- We initialize the **items** semaphore to 0

```
void enqueue (queue_t *queue, data_t *data, sem_t *space, sem_t *items)
{
  sem_wait (space);
  enqueue_unsafe (queue, data);
  sem_post (items);
}

data_t * dequeue (queue_t * queue, sem_t *space, sem_t *items)
{
  sem_wait (items);
  data_t * data = dequeue_unsafe (queue);
  sem_post (space);
  return data;
}
```

Safe producer-consumer with a bounded queue and multiple producers and consumers

- Unfortunately, the two semaphores aren't enough when there are multiple producers and consumers
- In that situation, two producers could both be calling **`enqueue_unsafe()`**, potentially causing a race condition with the increment
- The solution is to one lock for producers and one lock for consumers
- We could use a single lock for both, but using two locks allows producers and consumers to modify the queue concurrently yet safely

```
void enqueue (queue_t *queue, data_t *data, sem_t *space, sem_t *items, pthread_mutex_t *producer_lock)
{
  sem_wait (space);
  pthread_mutex_lock (producer_lock);
  enqueue_unsafe (queue, data);
  pthread_mutex_unlock (producer_lock);
  sem_post (items);
}

data_t * dequeue (queue_t * queue, sem_t *space, sem_t *items, pthread_mutex_t *consumer_lock)
{
  sem_wait (items);
  pthread_mutex_lock (consumer_lock);
  data_t * data = dequeue_unsafe (queue);
  pthread_mutex_unlock (consumer_lock);
  sem_post (space);
  return data;
}
```

Upcoming

Next time…

- Readers-writers problem
- Dining philosophers

Reminders

- Finish Project 3
  - **Due Friday before midnight!**
- Read sections 8.4 and 8.5