Semaphore Usage: Basic Synchronization Problems

Lecture #15, [redacted]

Mutual Exclusion:

```
semaphore m = 1; // use 1 for mutex
wait(m); // decrements m, is an atomic instruction, waits for m to be
greater than zero
CRITICAL SECTION
signal(m); // increments m, is an atomic operation
```

In this lecture, we will look at a variety of standard synchronization constructs that can be built using semaphores. We will also look at some "classic" synchronization problems and solutions to them using these constructs.

Multiplex:

To allow a certain number of threads to be in their critical sections at the same time, set the semaphore to that number. For example, to allow up to five threads to be in their respective critical sections at the same time, initialize the shared semaphore as follows:

```
semaphore m = 5;
```

Signaling:

To make a statement, a, execute before a statement b in two separate threads, use:

```
semaphore m = 0;
Thread A                          Thread B
                                  wait(m);
a;                                b;
signal(m);
```

This forces B to wait if statement a hasn't been executed, but B proceeds if a had already been executed in the past. Convince yourself that wait() and signal must be atomic for this (and other constructs) to work as desired.

Rendezvous:

Semaphores can be used to make a point in the code where two threads will wait for one another to reach that point before proceeding.

```
semaphore aArrived = 0;
semaphore bArrived = 0;
Thread A                                signal(aArrived);
...code before rendezvous...        ...code after rendezvous...
wait(bArrived);
```

```
                                        wait(aArrived);
Thread B                                ...code after rendezvous...
...code before rendezvous...
signal(bArrived);
```

Barrier:

A barrier is like a rendezvous, but for more than two threads. If a thread reaches its critical point (not to be confused with a critical section, a critical point is simply the instruction right after the rendezvous point), it should wait for the other threads to reach their critical point before proceeding.

Initialization:

```
n = number of threads;
count = 0;
semaphore mutex = 1;
semaphore barrier = 0;
```

Code for the barrier:

```
// rendezvous point
wait(mutex);      // gets the mutex lock to update the count of the
                  // threads at the rendezvous
      count = count + 1;
signal(mutex);    // releases the mutex lock
if(count == n)    // if all the threads are at the rendezvous point,
                  // signal them to start proceeding
      signal(barrier);
wait(barrier);    // wait here until signaled
signal(barrier);  // signal another thread to start proceeding
// critical point
```

Important point to understand: the if clause which compares the value of count with n is not treated as a critical section. While this may be a source of race conditions in general (e.g., a thread gets context switched out right after finding that the clause is true, but before it is resumed someone else changes the count), it is OK here. This is because this condition will be found to hold by the nth thread to reach its rendezvous, following which no other thread will update the count.

The way this works is that the n - 1 threads to reach this block go through the code until the wait(barrier); statement. The nth thread comes along and signals one of the other threads to proceed. That thread then signals another thread and so on until all threads have moved into their critical point.

Do on your own: How would you modify the above to realize a reusable barrier? (See the Little Book of Semaphores for this, if needed).

FIFO Queue:

If threads need to be woken up in the order in which they called wait, a FIFO queue of semaphores can be used. You will need a queue, n + 1 semaphores, and two functions to replace wait() and signal();

Initialization:

```
      queue Q;    // a FIFO queue
      semaphore mutex = 1;
      semaphore Sem[1, ... , n] = [0, ..., 0];

// myWait and mySignal functions for thread i:

myWait(Sem[i]) {
      wait(mutex);
            Q.add(Sem[i]);    // Critical Section
      signal(mutex);
      wait(Sem[i]); }

mySignal() {
      wait(mutex);
            sem = Q.remove(); // Critical Section
      signal(mutex);
      signal(sem); }
```

Important/Classic Synchronization Problems:

Producer/Consumer:

The producer/consumer problem for one producer and one consumer.

Initialization:

```
      semaphore empty = N;    // keeps track of the number of empty slots in
                              // the buffer
      semaphore full = 0;     // keeps track of the number of items in the
                              // buffer
      semaphore mutex = 1;
      buffer[N];
```

Producer:

```
      while(1) {
            wait(empty);      // waits for the buffer to have an empty spot,
                              // then decrements the number of empty slots
            in = (in + 1) % N;
            wait(mutex);
            buffer[in] = item();
            signal(mutex);
            signal(full); }   // increments the full semaphore to show that
                              // an item has been produced
```

Consumer:

```
      while(1) {
            wait(full); // waits for at least one item to be in the buffer,
                        // then decrements the number of items in the buffer
            out = (out + 1) % N;
            wait(mutex);
```

```
            item = buffer[out];
            signal(mutex);
            signal(empty); }  // increments the empty semaphore to show that
                              // an item has been consumed
```

The critical sections in this code are where the threads deal with the buffer. They cannot modify the buffer simultaneously or else problems would arise. The liveness condition of the producer and consumer is a bounded wait of N. If the producer or consumer is waiting at either the wait(empty) or wait(full) statements respectively, then the other thread runs at most N times since that is the maximum number of times that either thread can go through its critical section before the waiting thread runs. For example, let's say that the buffer is empty and the consumer waiting at its entry point. The producer can run at most N times before it waits at its own entry point, letting the consumer run.

Generalized Producer/Consumer:

The producer/consumer problem generalized to allow multiple producers and consumers.

Initialization:

```
      semaphore empty = N;    // keeps track of the number of empty slots in
                              // the buffer
      semaphore full = 0;     // keeps track of the number of items in the
                              // buffer
      semaphore mutex = 1;
      buffer[N];
```

Producer:

```
      while(1) {
            wait(empty);      // waits for the buffer to have an empty spot,
                              // then decrements the number of empty slots
            wait(mutex);
            in = (in + 1) % N;
            buffer[in] = item();
            signal(mutex);
            signal(full); }   // increments the full semaphore to show that
                              // an item has been produced
```

Consumer:

```
      while(1) {
            wait(full); // waits for at least one item to be in the buffer,
                        // then decrements the number of items in the buffer
            wait(mutex);
            out = (out + 1) % N;
            item = buffer[out];
            signal(mutex);
            signal(empty); }  // increments the empty semaphore to show that
                              // an item has been consumed
```

The only change in the code between this and the previous is that the mutex lock has been moved so that only one producer or consumer can manipulate the buffer and in and out variables at a time to prevent the threads from interfering with one another since they all handle the same buffer and in and out variables. The in and out variables must now be shared between the producers and consumers, respectively (hence the need to include these into critical sections). The liveness conditions change compared to the single producer single consumer. E.g., the group of producers (or consumers) as a whole will still be offered bounded wait. However, an individual producer or consumer may starve. Make sure you understand clearly why this is so.

Exercise: implement this to offer bounded wait to each producer and consumer.