Lecture 7: Semaphores and Bounded Buffer

7.0 Main Points:

Definition of semaphores
Example of use of semaphores

7.1 Motivation

Synchronization is a way of coordinating multiple concurrent activities that are using shared sate. What are the right synchronization abstractions, to make it easy to build correct concurrent programs?

7.1 Definition of Semaphores

Semaphores are a kind of generalized lock, first defined by Dijkstra in the late 60's. Semaphores are the main synchronization primitive used in UNIX.

Semaphores have a positive integer value, and support only the following two operations:

semaphore->P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1

semaphore->V(): an atomic operation that increments semaphore by 1, waking up a waiting P, if any

Semaphores are like integers, except:

1. No negative values
2. Only operations are P and V -- can't read or write value, except to set it initially
3. Operations must be atomic -- two P's that occur together can't decrement the value below zero. Similarly, thread going to sleep in P won't miss wakeup from V, even if they both happen at about the same time.

Binary semaphore: instead of an integer value, has a boolean value. P waits until value is 1, then sets it to 0. V sets value to 1, waking up a waiting P if any.

7.2 Two uses of semaphores

7.2.1 Mutual exclusion

When semaphores are used for mutual exclusion, the semaphore has an initial value of 1, and P() is called before the critical section, and V() is called after the critical section.

```
semaphore->P();
// critical section goes here
semaphore->V();
```

7.2.1 Scheduling constraints

Semaphores can also be used to express generalized scheduling constraints.

For example, you can implement Thread::Join using semaphores:

```
Initial value of semaphore = 0

Thread::Join calls P
Thread finish calls V
```

7.3 Producer-consumer with a bounded buffer

7.3.1 Problem definition

Producer puts things into a shared buffer, consumer takes them out. Need synchronization for coordinating producer and consumer.

Example: cpp | cc1 | cc2 | as (cpp produces bytes for cc1, which consumes them, and in turn produces bytes for cc2 ...)

Don't want producer and consumer to have to operate in lockstep, so put a fixed-size **buffer** between them; need to synchronize access to this buffer. Producer needs to wait if buffer is full; consumer needs to wait if buffer is empty.

Another example: Coke machine. Producer is delivery person; consumers are students and faculty.

Solution uses semaphores for both mutex and scheduling.

7.3.2 Correctness constraints for solution

1) Consumer must wait for producer to fill buffers, if none full (scheduling constraint)

2) Producer must wait for consumer to empty buffers, if all full (scheduling constraint)

3) Only one thread can manipulate buffer queue at a time (mutual exclusion)

Use a separate semaphore for each constraint; note semaphores being used in multiple ways.

```
Semaphore fullBuffers; // consumer's constraint
                    // if 0, no coke in machine
Semaphore emptyBuffers;// producer's constraint
               // if 0, nowhere to put more coke
Semaphore mutex;    // mutual exclusion
```

7.3.3 Semaphore solution

```
Semaphore fullBuffers = 0 // initially, no coke!
Semaphore emptyBuffers = numBuffers;
               // initially, number of empty slots
               // semaphore used to count how many
               // resources there are!
```

```
Semaphore mutex = 1; // no one using the machine

Producer() {
  emptyBuffers.P(); // check if there’s space
                    // for more coke
  mutex.P();        // make sure no one else
                    // is using machine
  put 1 coke in machine
  mutex.V();        // ok for others to use
machine
  fullBuffers.V();  // tell consumers there’s now
a
  }                 // coke in the machine

Consumer() {
  fullBuffers.P();  // check if there’s a coke in
                    // the machine
  mutex.P();        // make sure no one else
                    // is using machine
  take 1 coke out;
  mutex.V();        // next person’s turn
  emptyBuffers.V();    // tell producer we need
more
```

7.3.4 Questions

Why does producer P + V different semaphores than the consumer?

Is order of P’s important?

Is order of V’s important?

What if we have 2 producers or 2 consumers? Do we need to change anything?