CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2004

Lecture 8: Semaphores, Monitors, & Condition Variables

8.0 Main Points:

- Definition of semaphores
- Example of use of semaphores to solve the bounded buffer problem
- Definition of monitors and condition variables
- Demonstration of their use in Producer/Consumer problem

8.1 Motivation

Writing concurrent programs is hard because you need to worry about multiple concurrent activities reading and writing the same memory. It is hard because ordering matters.

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

This lecture and the next, present a couple ways of structuring the sharing.

8.2 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 the original UNIX.

Semaphores have a non-negative integer value, and support the following two operations:

- **semaphore->P():** an atomic operation that waits for semaphore to become positive, then decrements it by 1. (Think of this as the "wait" operation)
- **semaphore->V():** an atomic operation that increments semaphore by 1, waking up a waiting P, if any. (Think of this as the "signal" operation)

Semaphores are like integers, except:

1. No negative values.
2. Only operations allowed 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: like a lock (has a boolean value). Initialized to 1. P waits until value is 1, and then sets it to 0. V sets value to 1, waking up a waiting P, if any.

8.3 Two uses of semaphores

8.3.1 Mutual exclusion (initial value = 1)

Binary semaphores can be used for mutual exclusion: initial value of 1; P() is called before the critical section; and V() is called after the critical section.

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

8.3.2 Scheduling constraints (initial value = 0)

Locks are fine for mutual exclusion, but what if you want a thread to wait for something? For example, suppose you had to implement Thread::Join, which must wait for a thread to terminate.
By setting the initial value to 0 instead of 1, we can implement waiting on a semaphore:

```
Initial value of semaphore = 0
```

```
Fork
Thread::Join calls P       // will wait until something makes
                           // the semaphore positive.

Thread finish calls V      // makes the semaphore positive
                           // and wakes up the thread
                           // waiting in Join.
```

8.4 Producer-consumer with a bounded buffer

8.4.1 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 | ld` (`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.

Solutions use semaphores for both mutex and scheduling.

8.4.2 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)

General rule of thumb: Use a separate semaphore for each constraint.

Note how semaphores are 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
```

8.4.3 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
}
```

8.4.4 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?

8.5 Motivation for monitors and condition variables

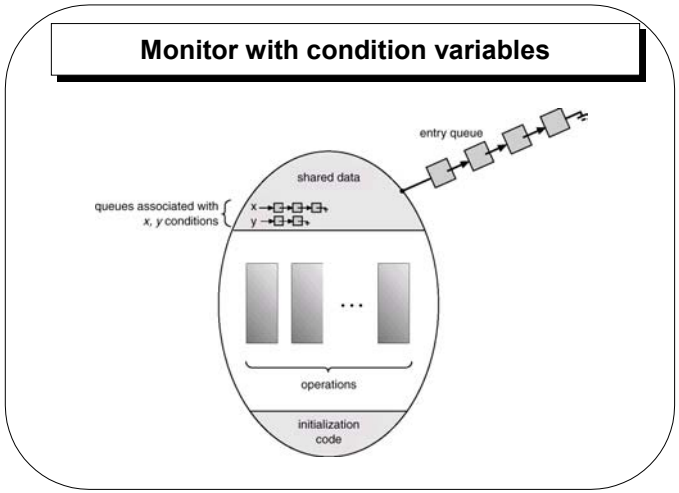
Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores. But the problem with semaphores is that they are dual purpose. They're used for both mutex and scheduling constraints. This makes the code hard to read, and hard to get right.

Idea in monitors is to separate these concerns: use *locks* for mutual exclusion and *condition variables* for scheduling constraints.

8.6 Monitor Definition

Monitor: *a lock and zero or more condition variables for managing concurrent access to shared data*

Note: Textbook describes monitors as a programming language construct, where the monitor lock is acquired automatically on calling any procedure in a C++ class. No widely-used language actually does this however! (although Java comes close, with its "synchronized" objects). In Nachos, and in many real-life operating systems, such as Windows NT, OS/2, or Solaris, monitors are used with explicit calls to locks and condition variables.



8.6.1 Lock

The **lock** provides mutual exclusion to the shared data. Remember:

- `Lock::Acquire` – wait until lock is free, then grab it
- `Lock::Release` – unlock, wake up anyone waiting in Acquire

Rules for using a lock:

- Always acquire before accessing shared data structure
- Always release after finishing with shared data.
- Lock is initially free.

Simple example: a synchronized list

```
AddToQueue() {
lock.Acquire();       // lock before using
shared data
put item on queue;    // ok to access shared
data
lock.Release();       // unlock after done
with shared
                      // data
}

RemoveFromQueue() {
lock.Acquire();       // lock before using
shared data
if something on queue // ok to access shared
data
     remove it;
lock.Release();       // unlock after done
with shared
                      // data
return item;
}
```

8.6.2 Condition variables

How do we change RemoveFromQueue to wait until something is on the queue?

Logically, we want to go to sleep inside of the critical section, but if we hold the lock when we go to sleep, other threads won't be able to get in to add things to the queue, to wake up the sleeping thread.

Key idea with condition variables: make it possible to go to sleep inside critical section, by **atomically** releasing lock at same time we go to sleep

Condition variable: *a queue of threads waiting for something* ***inside*** *a critical section*

Condition variables support three operations:

- **Wait()** – Release lock, go to sleep, re-acquire lock
  Releasing lock and going to sleep is atomic
- **Signal()** – Wake up a waiter, if any
- **Broadcast()** – Wake up all waiters

Rule: must hold lock when doing condition variable operations.

Note: In Birrell paper, he says can do signal outside of lock – IGNORE HIM (this is only a performance optimization, and likely to lead you to write incorrect code).

A synchronized queue, using condition variables:

```
AddToQueue() {
      lock.Acquire();
      put item on queue;
      condition.signal();
      lock.Release();
}

RemoveFromQueue() {
      lock.Acquire();
      while nothing on queue
            condition.wait(&lock);// release lock; go to
                                  // sleep; re-acquire lock
      remove item from queue;
      lock.Release();
      return item;
}
```


8.6.3 Mesa vs. Hoare monitors

Need to be careful about the precise definition of signal and wait.

Mesa-style: (Nachos, most real operating systems)

- Signaler keeps lock, processor
- Waiter simply put on ready queue, with no special priority. (in other words, waiter may have to wait for lock)

Hoare-style: (most textbooks)

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives lock, processor back to signaler when it exits critical section or if it waits again.

Above code for synchronized queuing happens to work with either style, but for many programs it matters which one you are using. With Hoare-style, you can change "while" in RemoveFromQueue to an "if", because the waiter only gets woken up if there's an item is on the list. With Mesa-style monitors, waiter may need to wait again after being woken up, because some other thread may have acquired the lock, and removed the item, before the original waiting thread gets to the front of the ready queue.

This means as a general principle, you **almost always** need to check the condition after the wait, with Mesa-style monitors (in other words, use a "while" instead of an "if").