Lecture 8: Monitors, Condition Variables and Readers-Writers

8.0 Main points:

Definition of monitors and condition variables
Illustrate their use by solving readers-writers problem

8.1 Motivation for monitors

Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores. But problem with semaphores is that they are dual purpose. 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.

8.2 Monitor Definition

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

Note: Silberschatz describes monitors as a programming language construct, where the monitor lock is acquired automatically on calling any procedure in a Java class. In (C++) 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.

8.2.1 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;
}
```

8.2.2 Condition variables

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

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;
}
```

8.2.3 Mesa vs. Hoare monitors

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

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

Signaller 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)

Signaller gives up lock, CPU to waiter; waiter runs immediately

Waiter gives lock, processor back to signaller when it exits critical section or if it waits again.

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").

8.3 Readers/Writers

8.3.1 Motivation

Shared database (for example, bank balances, or airline seats)

Two classes of users:
- Readers -- never modify database
- Writers -- read and modify database

Using a single lock on the database would be overly restrictive. Want:
- many readers at same time
- only one writer at same time

8.3.2 Constraints

1. Readers can access database when no writers (Condition okToRead)

2. Writers can access database when no readers or writers (Condition okToWrite)

3. Only one thread manipulates state variables at a time.

8.3.3 Solution

Basic structure of solution

```
Reader
```

```
    wait until no writers
    access database
    check out -- wake up waiting writer

  Writer
    wait until no readers or writers
    access database
    check out -- wake up waiting readers or writer
```

State variables:

```
    # of active readers -- AR  = 0
    # of active writers -- AW  = 0
    # of waiting readers -- WR  = 0
    # of waiting writers -- WW  = 0

    Condition okToRead = NIL
    Condition okToWrite = NIL
    Lock lock = FREE
```

Code:

```
  Reader() {
    // first check self into system
    lock.Acquire();
    while ((AW + WW) > 0) {   // check if safe to read
                           // if any writers, wait
      WR++;
      okToRead.Wait(&lock);
      WR--;
      }
    AR++;
    lock.Release();

    Access DB

    // check self  out of system
    lock.Acquire();
```

```
    AR--;
    if (AR == 0 && WW > 0)//if no other readers still
                           // active, wake up writer
      okToWrite.Signal(&lock);
    lock.Release();
  }

  Writer() {   //  symmetrical
    // check in
    lock.Acquire();
    while ((AW + AR) > 0) { // check if safe to write
                      // if any readers or writers, wait
      WW++;
      okToWrite->Wait(&lock);
      WW--;
      }
    AW++;
    lock.Release();

    Access DB

    // check out
    lock.Acquire();
    AW--;
    if (WW > 0)  // give priority to other writers
      okToWrite->Signal(&lock);
    else if (WR > 0)
      okToRead->Broadcast(&lock);
    lock.Release();
  }
```

Questions:

1. Can readers starve?

2. Why does checkRead need a while?

8.4 Comparison between semaphores and monitors

Illustrate the differences by considering: can we build monitors out of semaphores? After all, semaphores provide atomic operations and queueing.

Does this work?

```
Wait() { semaphore->P(); }

Signal() { semaphore->V(); }
```

Condition variables only work inside of a lock. If try to use semaphores inside of a lock, have to watch for deadlock.

Does this work?

```
Wait(Lock *lock) {
   lock->Release();
   semaphore->P();
   lock->Acquire();
}
Signal() {
   semaphore->V();
}
```

Condition variables have no history, but semaphores do have history.

What if thread signals and no one is waiting?
What if thread later calls wait?

What if thread V's and no one is waiting?
What if thread later does P?

In other words, P + V are commutative -- result is the same no matter what order they occur. Condition variables are not commutative: wait doesn't return until **next** signal. That's why they must be in a critical section -- need to access state variables to do their job.

Does this fix the problem?

```
Signal() {
   if semaphore queue is not empty
     semaphore->V();
}
```

Need to release lock and go to sleep atomically.

Is it possible to implement condition variables using semaphores? Yes, but exercise left to the reader!

8.5 Summary

Monitors represent the logic of the program -- wait if necessary, signal if change something so waiter might need to wake up.

```
lock
while (need to wait)
  wait();
unlock

lock
do something so no need to wait
signal();
unlock
```