Monitors — Last Time

Monitors consist of a lock and one or more condition variables. They ensure that only one thread can enter the monitor and access shared data at a time.

A monitor operation:

- Encapsulates the shared data you want to protect.
- Acquires the mutex at the start.
- Operates on the shared data.
- Temporarily releases the mutex if it can't complete by waiting on a condition variable.
- Signals other threads if a condition they are waiting on becomes true.
- Reacquires the mutex when it can continue.
- Releases the mutex at the end.

Today

- Comparing Semaphores and Monitors
- In Class Monitor Problem

Comparing Semaphores and Monitors

Can we build monitors out of semaphores? After all, semaphores provide atomic operations and queuing. Does the following work?

```
condition->Wait() { semaphore->wait(); }
condition->Signal() { semaphore->signal(); }
```

But condition variables only work inside a lock. If we use semaphores inside a lock, we have may get *deadlock*. Why?

How about this?

```
condition->Wait(Lock *lock) {
   lock->Release();
   semaphore->wait();
   lock->Acquire();
}

condition->Signal() {
  semaphore->signal();
}
```

Comparing Semaphores and Monitors

Condition variables do not have any history, but semaphores do.

- On a condition variable signal, if no one is waiting, the signal is a no-op.
  $\Longrightarrow$ If a thread then does a condition→Wait, it *waits*.
- On a semaphore signal, if no one is waiting, the value of the semaphore is incremented.
  $\Longrightarrow$ If a thread then does a semaphore→Wait, the value is decremented and the thread *continues*.

In other words, semaphore→Wait and Signal are commutative, the result is the same regardless of the order of execution. Condition variables are not, and as a result they must be in a critical section to access state variables and do their job.

...It is possible to implement monitors with semaphores...

Implementing Monitors with Semaphores

```
class Monitor {
  public:
    void ConditionWait();   // Condition Wait
    void ConditionSignal(); // Condition Signal
  private:
    <shared data>;          // data protected by monitor
    semaphore cvar;         // suspends thread on a wait
    int waiters;            // number threads waiting on
                            // a cvar--one per condition
    semaphore lock;         // controls entry to monitor
    semaphore next;         // suspends this thread
                            // when it signals another
    int nextCount;          // number threads suspended
}                           // on next
Monitor::Monitor {
   cvar = 0;    // Nobody waiting on condition variable
   waiters = 0;
   lock = FREE; // Nobody in the monitor
   next = 0;    // Nobody suspended inside the monitor
   nextCount = 0;
}

// Wrapper code for all methods on the shared data
Monitor::someMethod () {
   lock->Wait();        // lock the monitor
   <ops on data and calls to ConditionWait()>
   <and ConditionSignal()>
   if (nextCount > 0)
      next->Signal();   // resume a suspended thread
   else
      lock->Signal();   // allow new thread into monitor
}
```

Implementing Monitors with Semaphores

```
Monitor::ConditionWait() {              // Condition Wait
   waiters += 1;
   if (nextCount > 0)
      next->Signal();   // resume a suspended thread
   else
       lock->Signal();  // allow new thread in monitor
   cvar->wait();        // wait on the condition
   waiters -= 1;
}
Monitor::ConditionSignal(){          // Condition Signal
   if (waiters > 0) {    // don't signal if there is no
                         // thread waiting
      nextCount += 1;
      cvar->Signal();          // Semaphore Signal
      next->Wait();            // Semaphore Wait
      nextCount -= 1;
     }
}
```

Is this Hoare semantics or Mesa semantics? What would you change to provide the other semantics?

Monitor Problem

- Find a partner

Monitor Problem Solution

Next Time

Deadlock conditions, prevention, and detection.