Monitors

CS 537 - Introduction to Operating Systems

Issues with Semaphores

- Semaphores are useful and powerful
- BUT, they require programmer to think of every timing issue
  - easy to miss something
  - difficult to debug
- Examples

Example 1	Example 2	Example 3
mutex.P()	mutex.V()	mutex.P()
critical section	*critical section*	*part of critical section*
	mutex.P()	mutex.V()
		remaining critical section

Monitors

- Let the compiler handle the details!
- Monitors are a high level language construct for dealing with synchronization
  - similar to classes in Java
  - a monitor has fields and methods
- Programmer only has to say what to protect
- Compiler actually does the protection
  - compiler will use semaphores to do protection

Monitors

- Basic structure of monitor

```
monitor monitor-name {
        // fields
        // methods
}
```

- Only methods inside monitor can access fields
- At most ONE thread can be active inside monitor at any one time

Condition Variables

- Monitors utilize *condition variables*
- Two methods associated with each condition variable
  - *wait*: blocks a thread, places itself in a waiting queue for this condition variable, and allows another thread to enter the monitor
  - *signal*: pulls a single thread off the waiting queue of this condition variable
    - **note:** only one thread removed from wait queue
    - if no threads waiting, no effect

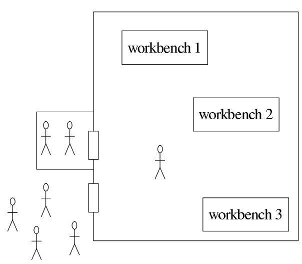
Example

```
monitor Foo {
    int maxValue, value;
    condition atMin, atMax;
    Foo(int maxValue) {
        this.maxValue = maxValue;
        value = 0;
    }
    void increment() {
        if(value == maxValue)
            atMax.wait;
        value++;
        atMin.signal;
    }
    void decrement() {
        if(value == 0)
            atMin.wait;
        value--;
        atMax.signal;
    }
}
```

Example

- Only the methods *increment* and *decrement* can access *maxValue, value, atMin, atMax*
- If one thread is executing in *increment*, then no other thread can be executing in either *increment* OR *decrement*
  - remember, only one thread allowed into a monitor at any one time
- Notice signal is always called at the end of increment and decrement
  - if no one waiting on the condition, no effect

Monitor Analogy



Monitor Analogy

- Imagine a single room
- Only one person allowed in room at a time
  - person locks the door to the room on entry
  - that person is doing some work in the room
- Anyone else wanting to do work in the room has to wait outside until door is unlocked
- If person in room decides to rest (but isn't finished with work), unlocks the door and goes into a side room (*wait*)

Monitor Analogy

- Multiple people can be in the side room
  - they are all taking a nap
- When person finishes working and leaves room, first check side room for anyone napping
  - if someone is napping, wake them up (*signal*)
  - otherwise, unlock the door and leave the room

More on *signal* and *wait*

- Assume P is running and Q is waiting
- Remember, only one thread inside the monitor at a time
- If P calls signal, there are two possible Q thread continuation strategies
  - *signal-and-hold*: P signals Q and then P blocks until Q leaves the monitor
  - *signal-and-continue*: P signals Q and then Q waits until P leaves the monitor

Signal-and-Continue

- Good points
  - P can exit the monitor and move on to other work
  - P can wakeup multiple threads before exiting the monitor
- Bad points
  - Q has to wait until P leaves the monitor so the condition it was waiting for may not be true anymore

Signal-and-Continue

- Example

```
public void decrement() {
  if(value == 0)
     atMin.wait;
  value--;
  atMax.signal;
  anotherCondition.signal;
}
```

- Notice that the thread currently in monitor gets to wakeup two threads and then exit
- If the thread waiting on *anotherCondition* wins the monitor lock and then changes *value* to *maxValue*, there will be a problem when the thread waiting on *atMax* wakes up
  - why?

Signal-and-Hold

- Good points
  - when Q wakes up it is guaranteed to have its condition still be true
  - P can wakeup multiple threads before exiting the monitor
- Bad points
  - P must wait immediately so it can't exit the monitor and do other work
- Revisit above example

Signal-and-Leave

- Allow P to continue after calling signal
- Force P to exit immediately
- Good points
  - when Q wakes up it is guaranteed to have its condition still be true
  - P can exit the monitor and move on to other work
- Bad points
  - P can only wakeup a single thread
- Good compromise

Implementing Signal and Wait

Programmer Code

```
monitor M {
    condition c;


    foo() {
        . . .
        c.wait;
        . . .
    }


    bar() {
        . . .
        c.signal;
    }

}
```

Compiler Code (psuedocode)

```
monitor M {
  sem_t mutex = 1;
  sen_t cs = 0;
  int cc = 0;
  foo() {
        mutex.down();
        . . .
        cc++;   mutex.up();   cs.down();   cc–;
        . . .
        mutex.up();
  }
  bar() {
        mutex.down();
        . . .
        if( cc > 0) { cs.up(); }
        else mutex.up();
  }
}
```

Implementing Signal and Wait

- Previous example is for a signal-and-leave system
- If a signal-and-hold technique is used, things are a bit different
  - keep a high priority count
    - thread that issues the signal is a high priority
  - on a wait call, check if any high priority threads waiting
  - on a signal, make the current thread wait in a high priority queue

Signal-and-Hold

```
monitor M {
  sem_t mutex = 1;
  sem_t cs = 0;   int cc = 0;
  sem_t hs = 0;   int hc = 0;
  foo() {
     mutex.down();
     . . .
     cc++;
     if(hc > 0) { hs.up(); }
     else { mutex.up(); }
     cs.down();   cc--;
     . . .
     if(hc > 0) { hs.up(); }
     else { mutex.up(); }
  }
  bar() {
     mutex.down();
     . . .
     if( cc > 0) { hc++;   cs.up();   hs.down();   hc--; }
     . . .
     if(hc > 0) { hs.up(); }
     else { mutex.up(); }
  }
}
```

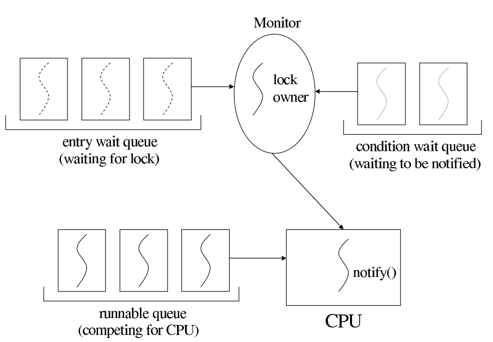
Alternative to Signal

- Major problem with signal is that it only wakes up one process
  - usually not much control over which process this is
- What if you want all the threads waiting on a condition to become runnable?
  - use *notify()*

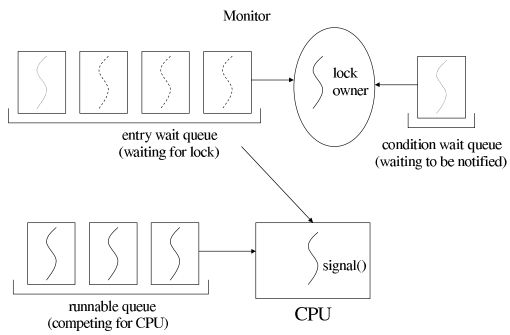
notify()

- *notify()* moves all threads currently blocked on some condition to the runnable state
  - only after a signal() is received
  - then all the threads compete to get the CPU
- Caution: good possibility the condition waiting for may no longer be true when a process gets the CPU
  - should do all wait calls inside of a while loop

Monitor Review



notify() Review



notify() Review