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

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex P()</td>
<td>mutex V()</td>
<td>mutex P()</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
<td>part of critical section</td>
</tr>
</tbody>
</table>

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
    - `notify`: only one thread removed from wait queue
    - if no threads waiting, no effect

Example

```java
monitorFunc {
  int maxValues value;
  conditionMax value;
  monitor maxValues {
    thisMaxValue=maxValue;
    value=0;
  }
  synchronized {
    if(value==maxValue)
      value+=
      value=
  }
}
```
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

- 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

```java
public void doSomething()
{
    int value = 0;
    while (true)
    
        value = assignValue();
        atom.wait();
        atom.signal();
}
```

- Notice that the thread currently in monitor gets to wake up two threads and then exit.
- If the thread waiting on another `Condition` wins the monitor lock and then changes value in `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

- 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
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