Using Synchronization

Administrative

Lab 2 due TONIGHT at MIDNIGHT.
   Sign up for demos on the Web.
   Please e-mail a second copy of your writeup to the *TA you sign
   up to demo with (unless you demo with me).

Lab 3-4:
   Lab #3 is due two weeks from today (2/25); Lab #4 slips to after
   spring break (4/6).
   Get started early on these labs.

Reading: Nutt, chapters 10 and 9.
Midterm exam Thursday 3/11, just before spring break.
Mesa Semantics for Monitors

Suppose again that purple signals blue in the original example.

*Mesa semantics*: the signalled thread transitions back to the **ready** state.

- There is no **suspended** state; the signaller continues until it exits the monitor or waits.
- The signalled thread **eventually** enters the monitor again and returns from its **wait**.
- Mesa semantics are easy to understand and implement...


\[
\begin{array}{c|c}
\text{state} & P1() \quad 1 \\ 
\hline
\text{ready} & \text{to enter} \\
\text{block} & \text{ed} \\
\text{signal}() & \text{(Mesa)} \\
\text{wait}() & \text{(Mesa)} \\
\end{array}
\]

BUT: the signalled thread must examine the monitor state again after the **wait**, as the state may have changed since the **signal**.

*Loop before you leap!*

Mutual Exclusion in Java

Mutexes and condition variables are built in to every Java object.

- no explicit classes for mutuxes and condition variables

Every object is/has a **monitor** with methods and state.

- At most one thread may “own” any given object’s monitor.
- A thread becomes the owner of an object’s monitor by executing a method declared as **synchronized**
  
  some methods may choose not to enforce mutual exclusion (unsynchronized)

  by executing the body of a **synchronized** statement

  supports finer-grained locking than “pure monitors” allow

  exactly identical to the Modula-2 “LOCK(m) DO” construct in Birrell
Wait/Notify in Java

Every Java object may be treated as a condition variable for threads using its monitor.

```java
public class Object {
    void notify();  /* signal */
    void notifyAll(); /* broadcast */
    void wait();
    void wait(long timeout);
}
```

```java
public class PingPong (extends Object) {
    public synchronized void PingPong() {
        while(true) {
            notify();
            wait();
        }
    }
}
```

A thread must own an object’s monitor to call wait/notify, else the method raises an `IllegalMonitorStateException`.

Wait(*) waits until the timeout elapses or another thread notifies, then it waits some more until it can re-obtain ownership of the monitor: Mesa semantics.

Guidelines for Choosing Lock Granularity

1. *Keep critical sections short.* Push “noncritical” statements outside of critical sections to reduce contention.

2. *Limit lock overhead.* Keep to a minimum the number of times mutexes are acquired and released.
   
   Note tradeoff between contention and lock overhead.

3. *Use as few mutexes as possible, but no fewer.*
   
   Choose lock scope carefully: if the operations on two different data structures can be separated, it *may* be more efficient to synchronize those structures with separate locks.
   
   Add new locks only as needed to reduce contention. “Correctness first, performance second!”
More Locking Guidelines

1. Write code whose correctness is obvious.
2. Strive for symmetry.
   - Show the Acquire/Release pairs.
   - Factor locking out of interfaces.
   - Acquire and Release at the same layer in your “layer cake” of abstractions and functions.
3. Hide locks behind interfaces.
4. Avoid nested locks.
   - If you must have them, try to impose a strict order.
5. Sleep high; lock low.
   - Design choice: where in the layer cake should you put your locks?

Tricks of the Trade #1

```c
int initialized = 0;
Lock initMx;

void Init() {
    InitThis(); InitThat();
    initialized = 1;
}

void DoSomething() {
    if (!initialized) { /* fast unsynchronized read of a WORM datum */
        initMx.Lock(); /* gives us a "hint" that we’re in a race to write */
        if (!initialized) /* have to check again while holding the lock */
            Init();
        initMx.Unlock(); /* slow, safe path */
    }
    DoThis(); DoThat();
}
```
Things Your Mother Warned You About #1

Lock dirtyLock;
List dirtyList;
Lock wiredLock;
List wiredList;

struct buffer {
    unsigned int flags;
    struct OtherStuff etc;
};

void MarkWired(buffer *b) {
    wiredLock.Acquire();
    b->flags |= WIRED;
    wiredList.Append(b);
    wiredLock.Release();
}

Lock dirtyLock;
List dirtyList;
Lock wiredLock;
List wiredList;

#define WIRED    0x1
#define DIRTY    0x2
#define FREE      0x4

void MarkDirty(buffer *b) {
    dirtyLock.Acquire();
    b->flags |= DIRTY;
    dirtyList.Append(b);
    dirtyLock.Release();
}

Example: Reader/Writer Lock

A reader/write lock or SharedLock is a new kind of “lock” that is similar to our old definition:

• supports Acquire and Release primitives
• guarantees mutual exclusion when a writer is present

But: a SharedLock provides better concurrency for readers when no writer is present.

class SharedLock {
    AcquireRead(); /* shared mode */
    AcquireWrite(); /* exclusive mode */
    ReleaseRead();
    ReleaseWrite();
}
Reader/Writer Lock Illustrated

Multiple readers may hold the lock concurrently in shared mode.

If each thread acquires the lock in exclusive (*write) mode, SharedLock functions exactly as an ordinary mutex.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Read</th>
<th>Write</th>
<th>Max Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Yes</td>
<td>No</td>
<td>Many</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Yes</td>
<td>Yes</td>
<td>One</td>
</tr>
<tr>
<td>Not Holder</td>
<td>No</td>
<td>No</td>
<td>Many</td>
</tr>
</tbody>
</table>

Writers always hold the lock in exclusive mode, and must wait for all readers or writer to exit.

Reader/Writer Lock: First Cut

```c
int i; /* # active readers, or -1 if writer */
Lock rwMx;
Condition rwCv;

SharedLock::AcquireWrite() {
    rwMx.Acquire();
    while (i != 0)
        rwCv.Wait(&rwMx);
    i = -1;
    rwMx.Release();
}

SharedLock::AcquireRead() {
    rwMx.Acquire();
    while (i < 0)
        rwCv.Wait(&rwMx);
    i += 1;
    rwMx.Release();
}

SharedLock::ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    rwCv.Broadcast();
    rwMx.Release();
}

SharedLock::ReleaseRead() {
    rwMx.Acquire();
    i = 1;
    if (i == 0)
        rwCv.Signal();
    rwMx.Release();
}
```
The Little Mutex Inside SharedLock

Reader/Writer Example

This implementation has weaknesses discussed in [Birrell89].

- **spurious lock conflicts** (on a multiprocessor): multiple waiters contend for the mutex after a signal or broadcast.
  
  *Solution*: drop the mutex before signalling.
  
  (If the signal primitive permits it.)

- **spurious wakeups**
  
  *ReleaseWrite* awakens writers as well as readers.
  
  *Solution*: add a separate condition variable for writers.

- **starvation**
  
  How can we be sure that a waiting writer will *ever* pass its acquire if faced with a continuous stream of arriving readers?
Reader/Writer Lock: Second Try

SharedLock::AcquireWrite() {
    rwMx.Acquire();
    while (i != 0)
        wCv.Wait(&rwMx);
    i = -1;
    rwMx.Release();
}

SharedLock::AcquireRead() {
    rwMx.Acquire();
    while (i < 0)
        ...rCv.Wait(&rwMx)...
    i += 1;
    rwMx.Release();
}

SharedLock::ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    if (readersWaiting)
        rCv.Broadcast();
    else
        wcv.Signal();
    rwMx.Release();
}

SharedLock::ReleaseRead() {
    rwMx.Acquire();
    i -= 1;
    if (i == 0)
        wCv.Signal();
    rwMx.Release();
}

Guidelines for Condition Variables

1. Understand/document the condition(s) associated with each CV.
   What are the waiters waiting for?
   When can a waiter expect a signal?
2. Always check the condition to detect spurious wakeups after returning
   from a wait: “loop before you leap”!
   Another thread may beat you to the mutex.
   The signaller may be careless.
   A single condition variable may have multiple conditions.
3. Don’t forget: signals on condition variables do not stack!
   A signal will be lost if nobody is waiting: always check the wait
   condition before calling wait.