Using Synchronization

Administrative

Lab 2 due Thursday at midnight.
   Sign up for demos on the Web; e-mail your writeup to me and your demoee, and cc: all group members.

Problem set will be available on the Web site this weekend.
   The best way to study for exams in this class is to drill synchronization problems. It will also help you with the Nachos labs.

Reading: Birrell, and Nutt: chapters 8-10 and start 3-4.
Midterm exam: Thursday 3/9, just before spring break.
The “Magic” of Semaphores and CVs

Any use of sleep/wakeup synchronization can be replaced with semaphores or condition variables.

- Most uses of blocking synchronization have some associated state to record the blocking condition.
  - e.g., list or count of waiting threads, or a table or count of free resources, or the completion status of some operation, or....
  - The trouble with sleep/wakeup is that the program must update the state atomically with the sleep/wakeup.

- Semaphores integrate the state into atomic P/V primitives.
  - ....but the only state that is supported is a simple counter.

- Condition variables (CVs) allow the program to define the condition/state, and protect it with an integrated mutex.

A Bounded Resource with a Counting Semaphore

```c
semaphore->Init(N);

int AllocateEntry() {
    int i;
    semaphore->Down();
    ASSERT(FindFreeItem(&i));
    slot[i] = 1;
    return(i);
}

void ReleaseEntry(int i) {
    slot[i] = 0;
    semaphore->Up();
}
```

A semaphore for an N-way resource is called a **counting semaphore**.

A caller that gets past a **Down** is guaranteed that a resource instance is reserved for it.

Problems?

- Note: the current value of the semaphore is the number of resource instances free to allocate.

- But semaphores do not allow a thread to read this value directly. Why not?
### Bounded Resource with a Condition Variable

```c
Mutex* mx;
Condition *cv;

int AllocateEntry() {
    int i;
    mx->Acquire();
    while(!FindFreeItem(&i))
        cv.Wait(mx);
    slot[i] = 1;
    mx->Release();
    return(i);
}

void ReleaseEntry(int i) {
    mx->Acquire();
    slot[i] = 0;
    cv->Signal();
    mx->Release();
}
```

“Loop before you leap.”

### Semaphores vs. Condition Variables

1. *Up* differs from *Signal* in that:
   - *Signal* has no effect if no thread is waiting on the condition.
     
     Condition variables are not variables! They have no value!
   - *Up* has the same effect whether or not a thread is waiting.
     Semaphores retain a “memory” of calls to *Up*.

2. *Down* differs from *Wait* in that:
   - *Down* checks the condition and blocks only if necessary.
     no need to recheck the condition after returning from *Down*
     wait condition is defined internally, but is limited to a counter
   - *Wait* is explicit: it does not check the condition, ever.
     condition is defined externally and protected by integrated mutex
Semaphores using Condition Variables

```c
void Down() {
    mutex->Acquire();
    ASSERT(count >= 0);
    while(count == 0) (Loop before you leap!)
        condition->Wait(mutex);
    count = count - 1;
    mutex->Release();
}

void Up() {
    mutex->Acquire();
    count = count + 1;
    condition->Signal(mutex);
    mutex->Release();
}
```

This constitutes a proof that mutexes and condition variables are at least as powerful as semaphores.

The Moat Problem

Travelers, knights, and troubadours arrive at the castle.
The castle guard decides when to lower the bridge to allow the arrivals into the castle.
If the bridge is down, new arrivals may enter immediately without waiting.
The guard doesn’t raise the bridge if there are people on it.

This can be solved easily using EventBarrier.
**EventBarrier**

*EventBarrier* combines semaphores and condition variables.

- *EventBarrier* has a binary “memory”, and no associated mutex.
- It has a “broadcast” to notify all waiting threads of an event.
- The broadcast primitive waits until the event is handled.

**EventBarrier::Wait()**

*If the EventBarrier is not in the signaled state, wait for it.*

**EventBarrier::Signal()**

*Signal the event, and wait for all waiters/arrivals to respond.*

**EventBarrier::Complete()**

*Notify EventBarrier that caller’s response to the event is complete. Block until all threads have responded to the event.*

---

**The Moat Problem with EventBarrier**

EventBarrier gate;

/* Called by knights etc. */
void EnterCastle() {
    gate.Wait();  /* wait for gate to open (if necessary) */
    CrossBridge();
    gate.Complete();  /* tell the guard it’s OK to close gate */
}

void GuardThread() {
    while (TRUE) {
        /* twiddle thumbs */
        /* watch for arriving travelers */
        /* decide when to open gate */
        WaitForOrderToOpenGate();
        gate.Signal();  /* open gate, wait for travelers to cross, close gate */
        /* gate is closed */
    }
}
**Another EventBarrier Example**

EventBarrier channel;

void OutputThread {
    while (TRUE) {
        ComputeDataToSend();
        channel.Wait();
        SendData();
        channel.Complete();
    }
}

void ChannelScheduler() {
    while (TRUE) {
        WaitUntilTimeToOpenChannel();
        channel.Signal(); /* open floodgate for burst of outgoing data */
        /* channel is closed */
    }
}

**Invariants:**

1. Output thread never blocks in Wait() if the channel is already open.
2. Channel never closes while a thread is sending data.
3. Each thread sends at most once each time the channel opens.

**SharedLock: 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.

often used in database systems

easy to implement using mutexes and condition variables

a classic synchronization problem
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.

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

<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>

Reader/Writer Lock: First Cut

```cpp
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

Limitations of the SharedLock Implementation

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 signaling.
  
  (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 signaler 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.
Guidelines for Choosing Lock Granularity


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

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

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

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

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

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

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