Lecture #9: Monitors, Condition Variables, and Readers-Writers

****

Review -- 1 min

****

The big picture: threads, shared objects, synchronization variables protecting the shared objects

2 synchronization actions
- lock
- scheduling constraint

Semaphore used for both

Example problem: bounded buffer

Implementing semaphores: turn off interrupts, test&set

****

Outline - 1 min

****

monitors
condition variables
Approach
example: readers-writers problem

****

Preview - 1 min

****

Finishing up synchronization
Other aspects of parallelism: deadlock, scheduling

****

Lecture - 20 min

****

1. Motivation for monitors

Semaphores a huge step up – just think of trying to do bounded buffer problem with just loads and stores

(busy waiting?)

3 problems with semaphores
Problem 1 – semaphores are dual purpose – mutex, scheduling constraints
→ hard to read code
→ hard to get code right

Problem 2 -- Semaphores have "hidden" internal state
Problem 3 – careful interleaving of "synchronization" and "mutex" semaphores

→ waiting for a condition is independent of mutex locks (to examine shared variables)
→ either cleverly define condition to map exactly to semaphore semantics (e.g., "12 buffers so initialize semaphore to 12" what if you don't know ahead of time how many buffers?) OR clever code (interleaving mutex V() with check condition P()) OR both

idea of monitor – separate these concerns: use locks for mutex and condition variables for scheduling constraints

philosophy – think about Join() example with producer/consumer. Just one line of code to make it work with semaphores, but need to think a bit to convince self it really works – relying on semaphore to do both mutex (via atomicity) and condition. What happens when you change the code later to, say, give different priorities to different consumers?

2. Monitor definition
monitor – a lock and zero or more condition variables for managing concurrent access to shared data

NOTE: Historically monitors were first a programming language construct, where the monitor lock is automatically acquired on calling any procedure in a C++ class. (Java does something like this – you can specify that certain routines are *synchronized*) Book tends to describe it this way.

But you don’t need this – monitors are also a set of programming *conventions* that you should follow when doing thread programming in C or C++ (or Modula c.f. Birrell): explicit calls to locks and condition variables

I will teach the “manual” version of monitors (and require that you do things manually on the projects) because I want to make sure it is clear what is going on and why.

2.1 Lock

The **lock** provides mutual exclusion to the shared data

Lock::Acquire() -- wait until lock is free, then grab it
Lock::Release() – unlock; wake up anyone waiting in Acquire

Rules for using a lock

- Always acquire before accessing shared data structure
- Always release after finishing with shared data
- Lock is initially free

Simple example: a synchronized list

```
AddToQueue(){
  lock.Acquire();          // lock before using shared data
  put item on queue;       // ok to access shared data
  lock.Release()           // unlock after done w. shared data
}

RemoveFromQueue(){
  lock.Acquire();          // lock before using shared data
  if something on queue remove it
  lock.Release();                   // unlock after done
  return item;
}
```

Aside:
If you have exceptions (as in Java), another variation is:

```
Foo(){
  try{
    lock.lock();
     …
     return item;
  }
  finally{
     lock.unlock();
  }
```

2.2 Condition variables
How do we change removeFromQueue to wait until something is on the queue?

Logically, want to go to sleep inside of critical section, but if hold lock when go to sleep, other threads won't be able to get in to add things to queue, to wake up sleeping thread

(Recall that for semaphores, we had essentially this problem and we solved it by cleverly doing our "accounting" for synchronization before we grabbed the lock for mutex. This type of subtle reasoning in programs worries me.)

Key idea with condition variables: make it possible to go to sleep inside critical section, by **atomically** releasing lock at same time we go to sleep

Condition variable: a queue of threads waiting for something **inside** a critical section

3 operations
Wait() – release lock; go to sleep; reaquire lock

- ♦ releasing lock and going to sleep are atomic

Signal() – wake up a waiter, if any
Broadcast() – wake up all waiters

RULE: must hold lock when doing condition variable operations

In lecture, I'll follow convention: require lock as parameter to condition variable operations. Get in the habit; other systems don't always require this

Some will tell you you can do signal outside of lock. IGNORE THEM. This is only a (small) performance optimization, and it is likely to lead you to write incorrect code.

A synchronized queue with condition variables

```
AddToQueue(){
  lock.Acquire();
  put item on queue;
  condition.signal(&lock);
  lock.Release();
}

RemoveFromQueue(){
  lock.Acquire();
  while nothing on queue{
        condition.wait(&lock); // release lock; go to sleep; require
   }
   remove item from queue;
   lock.Release();
   return item;
}
```

2.3 Mesa/Hansen v. Hoare monitors

Need to be careful about precise defn of signal and wait

Mesa/Hansen-style: (most real operating systems)
    Signaller keeps lock, processor

Waiter simply put on ready queue, with no special priority.
(In other words, waiter may have to **wait** to re-acquire lock)

Hoare-style: (most textbooks)
Signaller gives up lock and CPU to waiter; waiter runs immediately
Waiter gives up lock, processor back to signaller, when it exits critical section or if it waits again

Code above for synchronized queuing happens to work with either style, but for many programs it matters which you are using.

With Hoare-style, can change "while" in RemoveFromQueue to "if" because the waiter only gets woken up if item on the list.
With Mesa-style, waiter may need to wait again after being woken up b/c some other thread may have acquired the lock and removed the item before the original waiting thread gets to the front of the ready queue.

This means that as a general principle, you **almost always** need to check the condition after the wait, with mesa-style monitors (e.g., use a "while" instead of an "if")

Admin – 3 min

Project 2 available

3. Programming strategy:

(See "Programming with threads" handout for more details)

Goal: Systematic ("cookbook") way to write *easy to read and understand* and *correct* multi-threaded programs

Fall 2001 midterm:

- Every program with incorrect semantic behavior violated at least one rule
- >90% of programs that violated at least one rule were "obviously" semantically incorrect (that is, I could see the bug within seconds of looking at the program; there may have been additional bugs…)
  - All that violate one rule are *wrong* – they are harder to read, understand, maintain, …
  - Since I've declared "violating rule is *wrong*", huge reduction in bugs in exams and projects

A. General approach

1. Decompose problem into objects

object oriented style of programming – encapsulate shared state and synchronization variables inside of objects

Note:
(1) Shared objects are separate from threads
(2) Shared object encapsulates code, synchronization variables, and state variables

Warning: most examples in the book are lazy and talk about "thread 1's code" and "thread 2's code", etc. This is b/c most of the "classic" problems were studied before OO programming was widespread, and the textbooks have not caught up

Hint: don't manipulate synchronization variables or shared state variables in the code associated with a thread, do it with the code associated with a shared object.

Each thread tends to have a "main" loop that accesses shared objects *but the thread object does not include locks or condition variables in its state, and the thread's main loop code does not directly access locks or cv's.*

Locks and CVs are encapsulated in the shared objects.

Why?

(1) Locks are for synchronizing across multiple threads. Doesn’t make sense for one thread to “own” a lock!
(2) Encapsulation – details of synchronization are internal details of a shared object. Caller should not know about these details.

“Let the shared objects do the work.”

1A. Identify units of concurrency. Make each a thread with a go() method. Write down the actions a thread takes at a high level.

1b. Identify shared chunks of state. Make each shared *thing* an object. Identify the methods on those objects – the high-level actions made by threads on these objects.

1C. Write down the high-level main loop of each thread.

Advice: stay high level here. Don't worry about synchronization yet. Let the objects do the work for you.

Separate threads from objects. The code associated with a thread should not access shared state directly (and so there should be no access to locks/condition variables in the “main” procedure for the thread.) Shared state *and synchronization* should be encapsulated in shared objects.

Now, for each object:

2. Write down the synchronization constraints on the solution. Identify the type of each constraint: *mutual exclusion* or *scheduling*

3. Create a lock or condition variable corresponding to each constraint

4. Write the methods, using locks and condition variables for coordination

B. Coding standards/style

These are **required standards** in class. See the handout for details!

1. Always do things the same way

2. Always use monitors (condition variables + locks)

Almost always more clear than semaphores + "always do things the same way"

3. Always hold lock when operating on a condition variable

You signal on a condition variable because you just got done manipulating shared state. You proceed when some condition about a shared state becomes true. Condition variables are useless without shared state and shared state is useless without holding a lock.

4. Always grab lock at beginning of procedure and release it right before return

- Simplifies reading your code ("always do things the same way")
- If you find yourself wanting to release lock in middle of a procedure, 99% of time code would be more clear if you split it into two procedures

5. Always use

```
while(predicateOnStateVariables(...) == true/false){
                condition->wait(&lock);
}
```

not

```
if(...){…
```

(Where `PredicateOnStateVariables(...)` looks at the state variables of the current object to decide if it is OK to proceed.)

`While` works any time `if` does, and it works in situations when `if` doesn't. By rule 1, you should do things the same way every time.

`If` breaks modularity

When you always use ***while***, you are given incredible freedom about where you put the signal()'s. In fact, signal() becomes a *hint* -- you can add more signals to a correct program in arbitrary places and it remains a correct program!
→ Can determine correctness of signal calls and wait calls locally

6. (Almost) never `sleep()`

Never use sleep() to wait for another thread to do something. The correct way to wait for a condition to become true is to wait() on a condition variable.

sleep() is only appropriate when there is a particular real-world moment in time when you want to perform some action. If you catch yourself writing {\tt while(some condition)\{sleep();\}}, treat this is a big red flag that you are probably making a mistake.

I'm sure there are valid exceptions to all of the above rules, but they are few and far between. And the benefit you get by occasionally
breaking the rules is unlikely to make up for the cost in your effort, extra debugging and maintenance cost, and loss of modularity.

C. Java rules

This year, we are using Java for the project. Java is a modern language with supports for threads from day 1. This is mostly good news. 2 issues:

(1) For production use: Support for some dangerous/undesirable constructs/styles of programming
(2) For teaching: “too much” support for multi-threading → someone can write code that invokes synchronization with our without knowing what’s going on

→ Coding standards for this class
(J1) Do not use synchronized blocks within method

This is a specific incarnation of rule (4) above “Always grab locks at beginning and release at the end”

The following is forbidden:

```
Foo(){
        …
        synchronized(this){
                …
        }
        …
}
```

Instead, move the synchronized block into its own method.

(J2) Cleanly separate *Threads* from *shared objects*

Classes that define Threads (e.g., that extend Thread or implement Runnable) should include per-thread state. They should not include shared state. They should not include locks or condition variables.

The model is threads operate on shared state (picture).

(J3) *For this class* the *synchronized* keyword is forbidden. Instead, explicitly allocate and invoke locks and condition variables.

The purpose of this rule is to make it easier to teach and learn how to think about synchronization.

Example (correct):

```
class Foo{
        SimpleLock lock;
        Condition c1;
        Condition c2;

        public Foo(){
                lock = new SimpleLock();
                c1 = lock.newCondition();
                c2 = lock.newCondition();
                …
        }

        public void doSomething(…){
                try{
                        lock.lock();
                        …
                        while(…){
                                c1.awaitUninterruptably();
                        }
                        …
                        c2.signal();
                }
                finally{
                        lock.unlock();
                }
        }
}
```

Example (acceptable):

```
class Foo{
        SimpleLock lock;
        Condition c1;
        Condition c2;

        public Foo(){
                lock = new SimpleLock();
                c1 = lock.newCondition();
                c2 = lock.newCondition();
                …
        }

        public void doSomething(…){
                lock.lock();
                …
                while(…){
                        c1.awaitUninterruptably();
                }
                …
                c2.signal();
                lock.unlock();
        }
}
```

Example (forbidden for this class; often correct in real world):

```
class Foo{

        public Foo(){
                …
        }

        public synchronized void doSomething(…){
                …
                while(…){
                        this.wait();
                }
                …
                this.signal();
        }

}
```

(Note that once you leave this class the above style can be used when an object needs one lock and one condition variable; if you need two condition variables, fall back on the manual version as in this class.)

D. Example/Basic template:

(1,2) Always use condition variables for code you write. Be able to *understand* code written in semaphores. But the coding standard your manager (me) is enforcing for this group is condition variables for synchronization

```
class Foo{

private:
// Synchronization variables
Lock mutex;
```

```
Cond condition1;
Cond condition2;
…
// State variables
…

public:
Foo::foo()
{
  /*
   * (#4) Always, grab mutex at start of procedure, release at
   * end (or at any return!!!). Reasoning: if there is a logical
   * set of actions to do when you hold a mutex, that logical
   * set of actions should be expressed as a procedure, right?
   */
    mutex->acquire(){
        Assert(invariants hold – shared variables in consistent state)
         …
          invariants may or may not hold; shared variables may be
          in inconsistent state



         …
          // (#5)always "while" never "if"
          while(shared variables in some state){
                assert(invariants hold)
                // (#3) Always hold lock when operating on C.V.
                condition1->wait(&mutex)
                assert(invariants hold)
          }
         …
          invariants may or may not hold; shared variables may be
          in inconsistent state
         …
         … // (#3) Always hold lock when operating on C.V.
         …condition2->signal(&mutex);
         …condition1->signal(&mutex);
         …
         Assert(invarients hold)
  }mutex->release()
}
}; // Class
```

Rule (#6) (Almost) never sleep()

Sleep(time) puts the current thread on a waiting queue at the timer – only use it to wait until a specific time, not to wait for an event of a different sort

Hint: sleep should never be in a while(…){sleep}

Problems with using sleep:

1) no atomic release/reacquire lock
2) really inefficient (example – cascading sleeps in Aname)
3) not logical

Warning: on the project and on exams, improper use of sleep will be regarded as strong evidence that you have no idea how to write multi-threaded programs and will affect your grade accordingly.

(I make this a point of emphasis b/c this error is so common in past years and easy to avoid.)

****

Summary - 1 min

****

Monitors represent the logic of the program. Wait if necessary, signal if change something so waiter might need to wake up.

```
mutex->lock
while (need to wait)
    cv->wait();
mutex->unlock

mutex->lock
do something so no need to wait
cv->signal();
mutex->unlock
```