Concurrency Review 2

15-440/640 Spring 2015

February 10

Last Lecture

- Why we want concurrency
- Approaches to implementing concurrency
- Need to manage concurrency (Synchronization)
- Mutex (lock) operations
- Lock granularity
- Deadlocks and avoidance

Beyond “simple” locks

- Semaphores
- Condition variables

Semaphores

- Non-negative counting variables
- Two atomic operations:
  - P(s) – block until s>0, then s--
  - V(s) – s++
  - P also called wait, down, test
  - V also called post, up, increment
- Can be thought of as a “counting” lock

Semaphores vs. Mutexes

- Mutexes
  - Binary
  - Really intended for locks
  - Concept of “holding” lock – only holder can unlock
- Semaphores
  - Store value
  - Intended as a signal
  - Any thread can do P, V

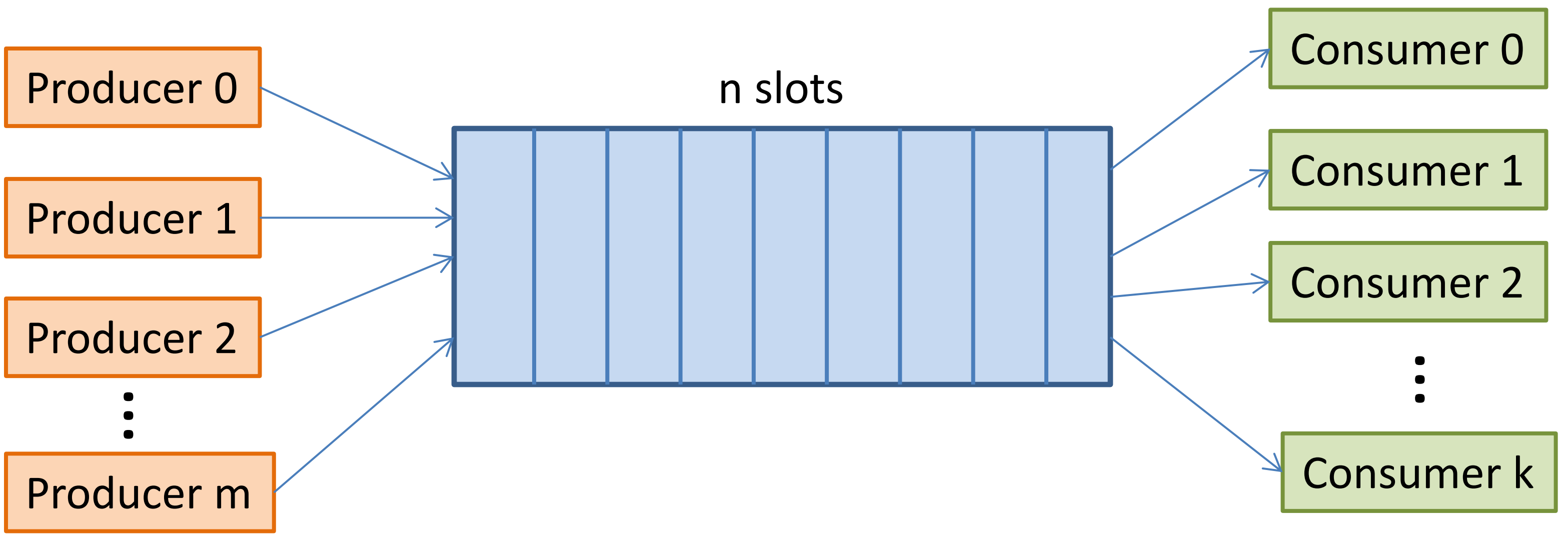
Semaphore Uses

- Age old problem:

- Waiting for any one of N resources
- Use semaphore, initialized to N
- First N visitors calling P() will not have to wait
- N+1st visitor calling P() will block
- Will be released when any one visitor is done, calling V()

Semaphores Uses

- Often used with mutex
- E.g. Producer-consumer on n-element buffer



Semaphores for producer-consumer

- Use a mutex to coordinate access to buffer,
  2 semaphores to manage number of resources

Semaphore free_slots, initialized to n
Semaphore filled_slots, initialized to 0
Mutex L
Buffer[n]

Producers:

```
P(free_slots)
Lock(L)
Add item to Buffer
Unlock(L)
V(filled_slots)
```

Consumers:

```
P(filled_slots)
Lock(L)
Remove item from Buffer
Unlock(L)
V(free_slots)
```

- Ok, so far, we have dealt with synchronizing access to shared data or resources

- Sometimes, we also want to coordinate when threads run

Event ordering between threads

- How can we make sure thread A reaches point X before B reaches Y ?
- One solution: use a semaphore, initialized to 0

Thread A	Thread B
…	…
X: V(s)	P(s)
…	Y: …

DIY scheduler

- Non-preemptive time sliced execution
- Each thread marked with “yield” points:

```
void yield() {
        V(s_sched);
        P(s[thread_id]);
}
```

- Scheduling thread main loop:

```
while(1) {
        P(s_sched);
        j = get_next_thread_id_to_run();
        V(s[j]);
}
```

- Initial value of s_sched determines number of concurrent threads.

On your mark, get set, go!

- Suppose several threads want to wait for some event to occur, then all resume
- How can we do this?
  - Maybe use a semaphore. N threads blocked on it. When event occurs, do V() N times.
  - But last part is not atomic, can run into problems

Condition Variables

- Three operations:
  - Wait() – wait for condition
  - Signal() – wake one waiting thread
  - Broadcast() – wake all waiting threads

- Condition variables must be used with a mutex (we'll come back to this)

Barrier Synchronization

- Barrier: a point in the code where all threads stop, and wait for all others to get there too
- Common paradigm:
  - Threads perform computation phase A; wait for all to finish, then start phase B
  - Useful in many simulations:
    - Threads simulate 1 “time step” for their slice of the world
    - Wait for all threads, then exchange state
    - Repeat

Implementing a Barrier

- We can use a CV to wake everyone waiting
- When do we trigger the broadcast to the CV?
  - When the last thread reaches the barrier!
  - Use a counter to track number of waiting threads
- Pseudocode:

```
mutex L;
int num_waiting=0;
condvar cv;
```

```
barrier() {
        lock(L);
        num_waiting++;
        if (num_waiting==NUM_THREADS) {
                num_waiting=0;
                broadcast(cv);
                unlock(L);
        } else {
                unlock(L);
                wait(cv);
        }
}
```

What if some thread is still here when broadcast is called?

- Not quite right – there is a *race condition*

Race Conditions

- A race condition exists if a thread needs to reach point x before another thread reaches point y to make the program work
  - In example barrier code, if one thread does not reach wait by time broadcast is called, it won’t get past the barrier
  - Furthermore, at next barrier, num_waiting won’t reach NUM_THREADS, so no one calls broadcast and every thread is left blocked!

Condition Variables to the rescue

- CVs are designed to fix this particular race
- Wait() must be used with a mutex:

```
Lock(L)
Wait(CV,L) – atomically unlock L and wait
             when resumed, will have L locked
Unlock(L)
```

Fixed Barrier

- Barrier pseudocode with race removed:

```
mutex L;
int num_waiting=0;
condvar cv;
```

```
barrier() {
     lock(L);
     num_waiting++;
     if (num_waiting==NUM_THREADS) {
          num_waiting=0;
          broadcast(cv);
     } else wait(cv, L);
     unlock(L);
}
```

Summary of semaphores, CVs

- Semaphores
  - Signaling methods with an associated count
  - Great for coordinating access to a set of resources
  - Can also coordinate when threads run
- Condition Variables
  - Used to signal an event to multiple waiting threads
  - Great for barrier synchronization

What about other languages?

- We have illustrated concurrency concepts in C, since everything is explicit
- Most languages provide similar primitives
- Language syntax may hide synchronization operations, but they are really implemented in the same way

Java Threads

- Provide a class that implements Runnable:

```
public class MyClass implements Runnable {
    public void run() {
        /* do stuff */
    }
}
```

- To launch thread, instantiate a Thread object:

```
Thread t = new Thread( new MyClass() );
```

- Exits when run() returns
- Join operation:

```
t.join();
```

Java Threads - alternative

- Can also subclass Thread:

```
public class MyClass extends Thread {
    public void run() {
        /* do stuff */
    }
}
```

- Launch thread by creating instance and calling start:

```
( new MyClass() ).start();
```

Java Concurrency Management

- Java provides “synchronized” methods:

```
public class Account {
    private float balance;
    public synchronized void deposit(float v) {
        balance+=v;
    }
}
```

- Ensures mutual exclusion among all synchronized methods of an object instance

Synchronized, behind the scenes

- All Java objects have an “intrinsic lock”
- Synchronized methods simply acquire the lock at start, and release at return

```
public class Account {
    // implicit private mutex lock
    private float balance;
    public synchronized void deposit(float v) {
            // implicit lock(lock)
            balance+=v;
            // implicit unlock(lock)
    }
}
```

- Intrinsic locks are reentrant

Synchronized, not magical

- Fine grained locking at object instance level

```
public class Account {
    private float balance;
    public synchronized void deposit(float v) {
        balance+=v;
    }
    public synchronized void transfer(Accout dest, float v) {
        balance-=v;
        dest.deposit(v);
}
```

- Can still deadlock!
- Language construct is convenient, but may hide that fact that multiple locks are being acquired
- Non-synchronized methods can still mess things up

More control over locking

- Synchronized Statements:

```
synchronized( object ) {
        / * do critical section */
}
```

- Can use any object’s intrinsic lock
- Can instantiate generic Object to use as a lock:

```
private Object myLock = new Object();
```

- We can use this to control when locks are held, order of acquisition, etc.

Built-in Condition Variable Functionality

- wait, notify, notifyAll = wait, signal, broadcast
- Can use to implement barrier synchronization
- Can use to make a semaphore:

```
public class Semaphore {
        private int count = 0;
        public synchronized V() {
                count++;
                this.notify();
        }
        public synchronized P() {
                while (count==0) {
                        try { this.wait() }
                        catch (InterruptedException e) {}
                }
                count--;
        }
}
```

Java Concurrency Summary

- Use Thread objects to launch new threads
- Synchronized methods and synchronized statements provide mutual exclusion
- All objects have intrinsic locks
- wait, notify, and notifyAll provide CV-like functionality in all objects

Concurrency Review

- Why we want concurrency
- Approaches to using concurrency
- Synchronizing primitives: Mutex, semaphore, condition variables
- Perils of concurrency: race conditions, deadlocks, performance issues
- C and Java examples