Notes for 11/13/09

Coherence is all about writes to a single memory location.
Consistency is all about writes to multiple memory locations.

Thread 1	Thread 2

`a = 1`
`b = 1`
`flag = 1`
	`while (flag == 0)`
	`    print a`
	`    print b`

Without a consistency policy, we do not know what this prints.

Recall that strict consistency in a distributed system is impossible.

Sequential consistency – all processors see writes in the same order
This is the best that we can do on a distributed system.
All processors see writes in the same order, but this is not necessarily the correct order, i.e., in the order in which that they physically happened.
Problem: this is slow.

PRAM/Processor consistency – on any single processor, all processors see writes from that processor in the order that they happened
In other words, we do not care about the order of writes of one processor with respect to another
With this consistency model, you can still do things like locks or memory barriers

Semaphores:

```
atomic down(&s)
    wait until s > 0
    s--

atomic up(&s)
    s++
```

A binary semaphore can be used to implement mutual exclusion (although mutexes generally have much stronger semantics than a binary semaphore).

```
global s = 1
. . .
down(s) // lock
critical section
up(s) // unlock
```

Jed's barber example with an agent trying to get customers (this is not the sleeping barber problem even though it features a sleeping barber)

```
global mutex seatsTakenMutex
global semaphore seatsFull = 0
global semaphore seatsEmpty = SEATS

// barber thread
while (1)
    down(seatsFull)
    if (endOfBusinessDay && seatsTaken == 0)
        break
    lock(seatsTakenMutex)
    seatsTaken = seatsTaken - 1 // cut hair
    unlock(seatsTakenMutex)
    up(seatsEmpty)

// agent thread
while (1)
    down(seatsEmpty)
    lock(seatsTakenMutex)
    seatsTaken = seatsTaken + 1 // put customer in chair
    up(seatsFull)
```

How are up(seatsEmpty) and down(seatsEmpty) related? We want the agent to sleep if there are no seats empty.

How are down(seatsFull) and up(seatsFull) related? We want the barber to sleep if all seats are empty.

Priority inversion:

Thread	Priority (higher is better)
T1	10
T2	20
T3	50

If T1 has the lock, T2 is running, and T3 wants T1's lock, then we have priority inversion. T2 is going to be scheduled over T1, but, since T3 is waiting on the lock that T1 has, then T2 is effectively higher priority than T3, since T3 cannot be scheduled until T1 releases the lock.

Solution: priority inheritance
Until T1 releases the lock, T1 has the priority of T3, the priority of the highest thread waiting on the lock