Semaphores

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Semaphores

- We have seen
  - Locks for mutual exclusion
  - Condition Variables for synchronization
- Semaphores are unified signaling mechanisms for **both mutual exclusion and synchronization**
  - Removes the need for counters, and additional boolean variables
- History
  - Proposed in 1968 by Dijkstra
  - Inspired by railroad semaphores:
    - Up/Down, Red/Green

Not more powerful!

- Everything you can do with locks+condvars you can do with semaphores, and vice-versa
- Sometimes the code looks much cleaner with one option than the other (we'll see examples)
- You will see both options used in practice
  - Depends on projects, people's preferences, languages, etc.
  - Some people are very opinionated about it
  - Some students after taking this course say they only like one of the two (Semaphores are strangely attractive to some)
- Once you truly understand concurrency, switching back and forth between the two options is really easy

Semaphore Operations

- A semaphore is an integer variable that is **never < 0**
- It can be initialized to any >=0 integer value
- The semaphore provides two atomic operations
- The P operation
  - P: from Dutch “proberen”, “to test”
  - Waits for the variable to be > 0 and then decrements the semaphore by 1
- The V operation
  - V: from Dutch “verhogen”, “to increment”
  - Increments the semaphore by 1
- Can be implemented from scratch using atomic hardware instructions
- Let’s live code a Semaphore class in Java right now…

Types of Semaphores

- Binary Semaphore:
  - Takes only values 0 and 1
    - Either enforced by the implementation with checks, or implicitly by initializing it to 0 and always calling P() after V()
  - Can be used for mutual exclusion
  - Can be used for signaling

- Counting Semaphores:
  - Takes any non-negative value
  - Typically used to count resources and block resource produces and consumers

Critical Section with Semaphores

- Doing a critical section with a (binary) semaphore (which I call “mutex” to remember it’s about mutual exclusion) is as simple as with a lock

```
semaphore_t mutex = 1;
int shared_variable;
void worker() {
  while(1) {
    P(mutex);
    shared_variable++;
    V(mutex);
  }
}
```

Critical Section with Semaphores

- Doing a critical section with a (binary) semaphore (which I call “mutex” to remember it’s about mutual exclusion) is as simple as with a lock

```
semaphore_t mutex = 1;
int shared_variable;
void worker() {
  while(1) {
    P(mutex);
    shared_variable++;
    V(mutex);
  }
}
```

Main difference with locks:

- A call to unlock() on an unlocked lock does nothing
  - but you shouldn’t really do it as it a bit incoherent
- A call to V() **always** increments the semaphore by one
  - so calling V() extra times is most likely a bug

Signaling Semaphores

- Another use of binary semaphore is to signal some event
  - A thread waits for an event by calling P
  - A thread signals the event by calling V
- Example: a "barrier" between two threads

Thread #1

```
. . .
. . .
V(ready1);
P(ready2);
. . .
. . .
```

Thread #2

```
. . .
. . .
V(ready2);
P(ready1);
. . .
. . .
```

Global Variables

```
semaphore ready1 = 0;
semaphore ready2 = 0;
```

Comparing with locks/condvars

semaphores

```
semaphore ready1 = 0;
semaphore ready2 = 0;
```

```
. . .
V(ready1);
P(ready2);
. . .
```

```
. . .
V(ready2);
P(ready1);
. . .
```

locks and cond vars

```
int  x =0;
cond_t cond;
lock_t mutex;
```

```
. . .
lock(mutex);
x++;
if (x < 2) {
    wait(cond, mutex);
} else {
    signal(cond);
}
unlock(mutex);
. . .
```

- Semaphores encapsulate the “counting variable”, thus shorter code
  - Generalizing to >2 threads requires an array of semaphores…
- Doing “two things at once” is great? or is it confusing?

Signaling with Semaphores

- Example: Thread #2 waits until Thread #1 sets flag to zero before doing something

Signaling with Semaphores

- Example: Thread #2 waits until Thread #1 sets flag to zero before doing something

```
int flag;
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do something>
V(mutex);
. . .
```

In a while loop to avoid spurious wakeups!

Signaling with Semaphores

- Example: Thread #2 waits until Thread #1 sets flag to zero before doing something

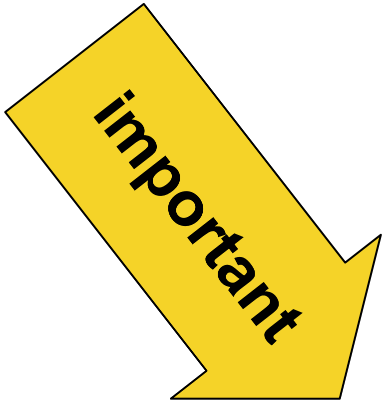
```
int flag;
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do something>
V(mutex);
. . .
```



Equivalent to a wait() on condition variable
- release the mutex
- wait
- reacquire the mutex

Comparing with locks/condvars

```
int flag;
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do something>
V(mutex);
. . .
```

```
int flag;
lock_t mutex;
cond_t cond;
```

Thread #1

```
. . .
lock(mutex);
flag--;
if (flag == 0)
  signal(cond);
unlock(mutex);
. . .
```

Thread #2

```
. . .
lock(mutex);
while (flag != 0) {
  wait(cond, mutex);
}
<do something>
unlock(mutex);
. . .
```

```
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

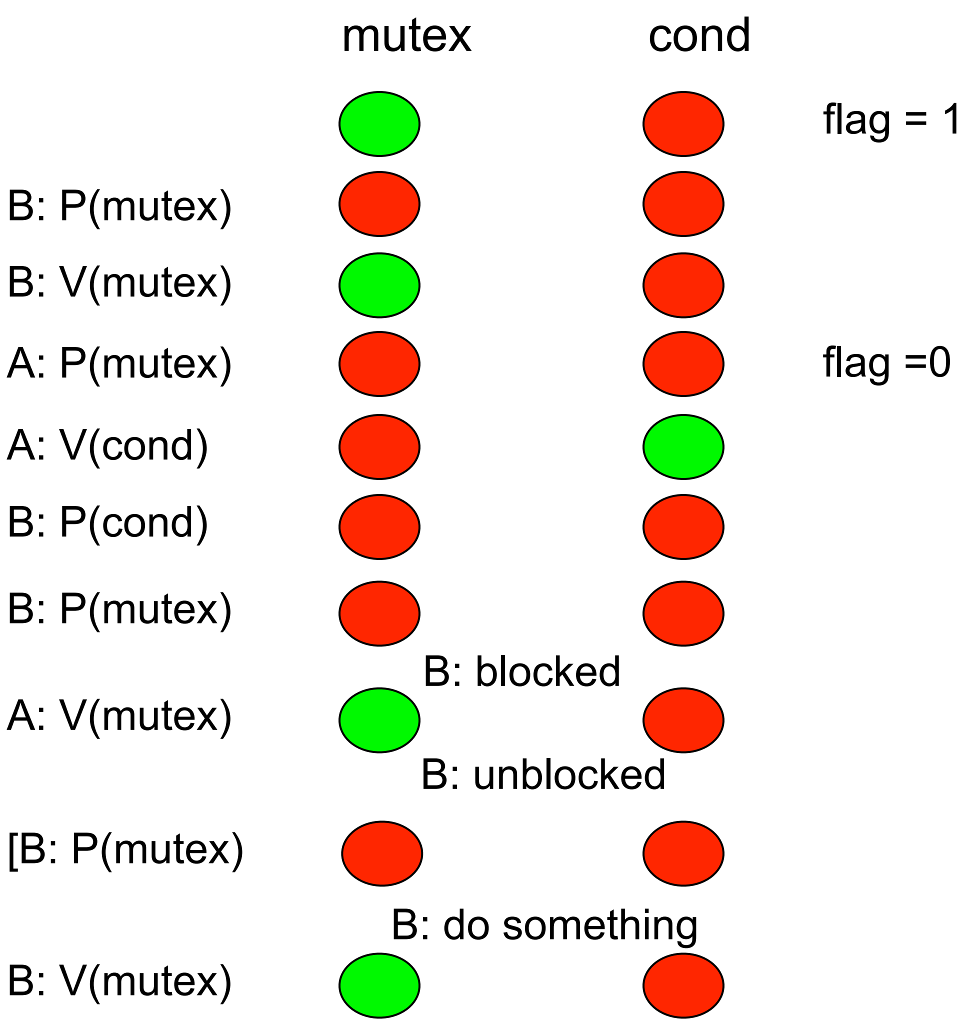
A

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

B

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do something>
V(mutex);
. . .
```

One possible interleaved execution:



Can we optimize this?

```
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do something>
V(mutex);
```

- Can we remove some calls to P() and V()?
- Consider the following line of reasoning:
  - The flag is, say, = 1
  - Thread #2 shows up first, does P(mutex)/V(mutex), then P(cond), and blocks, as it should
  - Thread #1 shows up, P(mutex), sets the flag to 0. It then does V(cond), as it should
  - Thread #1 then does V(mutex). This is because Thread #2 will need to enter the critical section after waking up from P(cond)
  - So we have the following:
    - Thread #1 is in the critical section
    - It wakes up Thread #2, which should then enter the critical section right away
  - Optimization: Don’t call V(mutex) on Thread #1 and don’t call P(mutex) on Thread #2
- Intuitive explanation: Thread #1 allows Thread #2 to “continue” in the critical section
- This is called “passing the baton”

Passing the Baton

```
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do something>
V(mutex);
```

```
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
  // transfer
     "privileges"
else
  V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  // receive "privileges"
  P(mutex);
}
<do something>
V(mutex);
. . .
```

If A is in a critical section, and A needs to wake up B that should enter the critical section after waking up, and A is done with the critical section, then A can just “skip” the V(mutex) and B can “skip” the P(mutex), and it works!

Passing the Baton

```
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag == 0)
  V(cond);
V(mutex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  P(mutex);
}
<do someth
V(m
```

```
semaphore_t mutex = 1;
semaphore_t cond = 0;
```

Thread #1

```
. . .
P(mutex);
flag--;
if (flag ==
  )
ex);
. . .
```

Thread #2

```
. . .
P(mutex);
while (flag != 0) {
  V(mutex);
  P(cond);
  / receive "privileges"
  (mutex);
}
<do something>
V(mutex);
. . .
```

You can see why some find semaphores powerful but confusing

If A is in a critica ection, and A needs to wake up B that should enter the critical section after waking up, and A is done with the critical section, then A can just "skip" the V(mutex) and B can "skip" the P(mutex), and it works!

Split Binary Semaphores

- A typical usage of binary semaphores is to do mutual exclusion and signaling at the same time
- Consider a specific Producer/Consumer problem
  - We have an arbitrary number of producers
  - We have an arbitrary number of consumers
  - We have a buffer that can contains **a single element**, consumed by consumers and produced by producers
  - Consumers must be delayed while the buffer is empty
  - Producers must be delayed while the buffer is full
- This can be easily implemented with 2 binary semaphores

Single Buffer Prod/Cons

```
semaphore_t empty = 1;
semaphore_t full = 0;
```

```
void producer() {
  while(true) {
    P(empty);
    buffer = <some value>;
    V(full);
  }
}
```

```
void consumer() {
  while(true) {
    P(full);
    consume(buffer);
    V(empty);
  }
}
```

Single Buffer Prod/Cons

```
semaphore_t empty = 1;
semaphore_t full = 0;
```

```
void producer() {
  while(true) {
    P(empty);
    buffer = <some value>;
    V(full);
  }
}
```

```
void consumer() {
  while(true) {
    P(full);
    consume(buffer);
    V(empty);
  }
}
```

- There is a simple “ping-pong” between the full and the empty semaphores
  - $0 \leq full + empty \leq 1$ (called a “split binary semaphore” effect)
- We get mutual exclusion “for free”
- The above is called split binary semaphores

Split Binary Semaphores

```
Thread #1: while (true) { P(X);  <S>;  V(Y); }
Thread #2: while (true) { P(Y);  <T>;  V(X); }
```

- Semaphores are initialized to (X=0,Y=1)
- They alternate between (X=0,Y=1) and (X=1,Y=0)
- Example: Starting with (X=0,Y=1)
  - Thread #1 cannot get to statement <S>
  - Thread #2 sets Y to 0
  - Thread #2 executes statement <T>
  - Thread #2 sets X to 1
  - We now have (X=1,Y=0)
  - Thread #2 cannot get to statement <T>
  - Thread #1 execute statement <S>
  - ...

Split Binary Semaphores

- This is the kind of “hand off” we had discussed when trying to implement producer/consumer only with locks
- With locks, I mentioned it was error prone
- Therefore, this is error-prone too: a code with tons of P() and V() hand-offs on many different semaphores will be very hard to understand/ debug/maintain
  - Giving semaphores good names is paramount
- But for simple cases it’s very readable and elegant
  - And we try to keep cases simple with concurrency, since going “fancy” is difficult regardless

General (non-binary) Semaphores

- Semaphores that take values higher than 1 are typically used to control access to a limited number of resources
  - In the previous example we controlled access to a single resource, i.e., one buffer slot
- The value of the semaphore indicates the number of free resources, from 0 to N
- Let's look at the "bounded buffer" producer/consumer problem
  - We already did this with condition variables, but we'll see now that with semaphores it's a bit easier

Bounded Buffer Prod/Cons

- Problem statement:
  - Arbitrary numbers of producers and consumers
  - The buffer can only store N elements
  - As we did before, our buffer will be a queue
- In our split binary semaphore example, mutual exclusion was enforced implicitly with the full/empty semaphores
- With general semaphores, we need an extra semaphore for mutual exclusion
- Let’s look at the code

One attempt

```
semaphore_t freeSlots = N;
semaphore_t occupiedSlots = 0;
semaphore_t mutex = 1;
```

```
void producer() {
  while(true) {
    P(mutex);
    P(freeSlots);
    <add element to queue>
    V(mutex);
    V(occupiedSlots);
  }
}
```

```
void consumer() {
  while(true) {
    P(mutex);
    P(occupiedSlots)
    <remove element from queue>
    V(mutex);
    V(freeSlots);
  }
}
```

- Does this work? (poll)

Nope: Deadlock

```
void producer() {
  while(true) {
    P(mutex);
    P(freeSlots);
    <add element to queue>
    V(mutex);
    V(occupiedSlots);
  }
}
```

```
void consumer() {
  while(true) {
    P(mutex);
    P(occupiedSlots)
    <remove element from queue>
    V(mutex);
    V(freeSlots);
  }
}
```

- Does this work? **NO: DEADLOCK**
  - The buffer is full
  - Producer acquires binary semaphore mutex
  - Producer blocks trying to acquire semaphore freeSlots because the buffer is full
  - All consumers block trying to acquire binary semaphore mutex!

Swapping the calls to P()

```
semaphore_t freeSlots = n;
semaphore_t occupiedSlots = 0;
semaphore_t mutex = 1;
```

```
void producer() {
  while(true) {
    P(freeSlots);
    P(mutex);
    <add element to queue>
    V(mutex);
    V(occupiedSlots);
  }
}
```

```
void consumer() {
  while(true) {
    P(occupiedSlots)
    P(mutex);
    <remove element from queue>
    V(mutex);
    V(freeSlots);
  }
}
```

- Does this work? (poll)

Swapping the calls to P()

```
void producer() {
  while(true) {
    P(freeSlots);
    P(mutex);
    <add element to queue>
    V(mutex);
    V(occupiedSlots);
  }
}
```

```
void consumer() {
  while(true) {
    P(occupiedSlots)
    P(mutex);
    <remove element from queue>
    V(mutex);
    V(freeSlots);
  }
}
```

- Does this work? **YES**
  - Can be formally proven
  - But you can easily see that we removed the deadlock problem since now a thread first checks if it can do work before getting the mutex

Swapping the Calls to V()?

```
void producer() {
  while(true) {
    P(freeSlots);
    P(mutex);
    <add element to queue>
    V(occupiedSlots);
    V(mutex);
  }
}
```

```
void consumer() {
  while(true) {
    P(occupiedSlots);
    P(mutex)
    <remove element from queue>
    V(freeSlots);
    V(mutex);
  }
}
```

- We can also think of swapping the V() calls
- Does this work? (poll)

Swapping the Calls to V()?

```
void producer() {
  while(true) {
    P(freeSlots);
    P(mutex);
    <add element to queue>
    V(occupiedSlots);
    V(mutex);
  }
}
```

```
void consumer() {
  while(true) {
    P(occupiedSlots);
    P(mutex)
    <remove element from queue>
    V(freeSlots);
    V(mutex);
  }
}
```

- We can also think of swapping the V() calls
- Does this work? **YES**
- It doesn't matter in which order the two things a thread is waiting for are signaled given that both are needed (the V() calls can be in any order)
  - And besides, blocking threads just get back to the ready queue and there could be other threads ahead of them anyway

Reader/Writer

- Another classical concurrency model is the reader/writer problem
- We have two kinds of processes:
  - Readers: read records from a database
  - Writers: read and write records from a database
- Selective mutual exclusion
  - **Concurrent readers are allowed**
  - A writer should access the database in mutual exclusion with all other writers and readers
- Typical of database applications
  - e.g., a Web/database server with one thread per transaction

A Naive Solution

```
semaphore_t rw = 1;
```

```
void reader() {
  while(true) {
    P(rw);
    <read from the DB>
    V(rw);
  }
}
```

```
void writer() {
  while(true) {
    P(rw);
    <write to the DB>
    V(rw);
  }
}
```

- It this a good reader-writer solution? (poll)

A Naive Solution

```
semaphore_t rw = 1;
```

```
void reader() {
  while(true) {
    P(rw);
    <read from the DB>
    V(rw);
  }
}
```

```
void writer() {
  while(true) {
    P(rw);
    <write to the DB>
    V(rw);
  }
}
```

- **Not** a good solution: it works but implements too strict a constraint as there can be no concurrent database reads
- Loss of throughput/performance because concurrent reads should be allowed
  - In many applications, there are few writers and many readers

Reader-Preferred Solution

- One simple fix is to allow multiple readers in a “greedy” fashion:
  - There is still a rw semaphore
  - While a reader is reading, other readers should be allowed in
  - Therefore we should have a variable, nr, keeping track of the current number of readers
  - That variable is used / updated by all readers, and should be protected by a mutual exclusion semaphore
- Let’s look at the code

Reader-Preferred Solution

```
void reader() {
  while(true) {

    P(mutex);
      if (nr == 0)  P(rw);  // I am first
      nr++;
    V(mutex);

    <read from the DB>

    P(mutex);
    nr--;
    if (nr == 0) V(rw);    // I am last
    V(mutex);
  }
}
```

```
semaphore_t mutex = 1;
semaphore_t rw = 1;
int nr = 0;
```

```
void writer() {
  while(true) {
    P(rw);
    <write to the DB>
    V(rw);
  }
}
```

Anybody sees the problem with this?

Reader-Preferred Solution

- The problem of the reader-preferred solution is that it is **too reader-preferred**
- There could be starvation of the writers
  - If there is always a reader able to read, the rw semaphore will be monopolized by readers forever
- Turns out it's very difficult to modify the code to make it fair between readers and writers
  - There is a classic solution that uses synchronization and the "passing the baton" technique
  - Based on a invariant condition and subtle signaling
  - You can look at it on your own if interested
- Let's instead look at a simple but pretty good solution

Maximum number of readers

- Let us define a maximum number of allowed concurrent readers, which simplifies the problem
  - And most likely makes sense for most applications
- Let’s say we allow at most N concurrent active readers
- We create a “resource” semaphore with initial value N
- Each reader needs to acquire one resource to be able to read
  - Therefore, N concurrent readers are allowed
- Each writer needs to acquire N resources to be able to write
  - Therefore, only one writer can be executing at a time and no readers can be executing concurrently
- Let’s look at the code

Reader/Writer

```
semaphore_t  sem = N;
```

```
void reader() {
  while(true) {
    P(sem);
    <read from the DB>
    V(sem);
  }
}
```

```
void writer() {
  while(true) {
    for (i=0; i<N; i++)
        P(sem);
    <write to the DB>
    for (i=0; i<N; i++)
        V(sem);
  }
}
```

Does this work? (consider multiple writers) (poll)

Reader/Writer

- Deadlock!
  - One could have two writers each start acquiring resources concurrently
  - For instance
    - Writer #1 holds 2 resource
    - Writer #2 holds N-2 resources
  - They're both blocked forever
- Solution: Don't allow two writers to execute the for loop of P() calls concurrently
- This can easily be done with mutual exclusion
- We need another semaphore!

```
void writer() {
  while(true) {
    for (i=0; i<N; i++)
      P(sem);
    <write to the DB>
    for (i=0; i<N; i++)
      V(sem);
  }
}
```

“OK” Reader/Writer Solution

```
semaphore_t  sem = N;
semaphore_t  wmutex = 1;
```

```
void reader() {
  while(true) {
    P(sem);
    <read from the DB>
    V(sem);
  }
}
```

```
void writer() {
  while(true) {
    P(wmutex);
    for (i=0; i<N; i++)
        P(sem);
    V(wmutex);
    <write to the DB>
    for (i=0; i<N; i++)
        V(sem);
  }
}
```

Reader-Writer Lock

- You may remember that I mentioned reader-writer locks
- This is a special kind of lock designed especially for the reader-writer problem
- java.util.concurrent.locks.ReentrantReadWriteLock

```
ReentrantReadWriteLock rwl =
                new ReentrantReadWriteLock();
. . .
rwl.readLock().lock();
. . .
rwl.readLock().unlock();
. . .
rwl.writeLock().lock();
. . .
rwl.writeLock().unlock();
. . .
```

java.util.concurrent Semaphore

- There is a `java.util.concurrent.Semaphore`
- It simply implements a semaphore
  - P() is called acquire()
  - V() is called release()
- It works exactly like you think it does

Pros/Cons for Semaphores

- Good
  - A single mechanism for many things
    - mutual exclusion, resource sharing, signaling/ blocking
  - General enough to solve any concurrency/ synchronization problem
  - Sometimes surprisingly elegant/short programs
- Bad
  - The fact that a single mechanism is used for multiple things can make a program very difficult to understand
  - Not very modular: e.g., the use of a semaphore in a thread depends on its use in another thread with dreaded “hand-off” behavior that may have been implemented

Conclusion

- As this point we've seen the two main low-level abstractions for thread synchronization
  - Locks + condition variables
  - Semaphores
- Next up, we look at famous concurrency problems
- But first, let's look at Assignment #7…