CS162
Operating Systems and Systems Programming
Lecture 9

Synchronization Continued, Readers/Writers example, Scheduling

February 23rd, 2015

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes*. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

Review: Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptyBuffers.P();     // Wait until space
    mutex.P();            // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();      // Tell consumers there is
                          // more coke
}
Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```

Condition Variables

- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
- **Condition Variable**: a queue of threads waiting for something inside a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();              // Get Lock
   queue.enqueue(item);         // Add item
   dataready.signal();          // Signal any waiters
   lock.Release();              // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();              // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();      // Get next item
   lock.Release();              // Release Lock
   return(item);
}
```

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

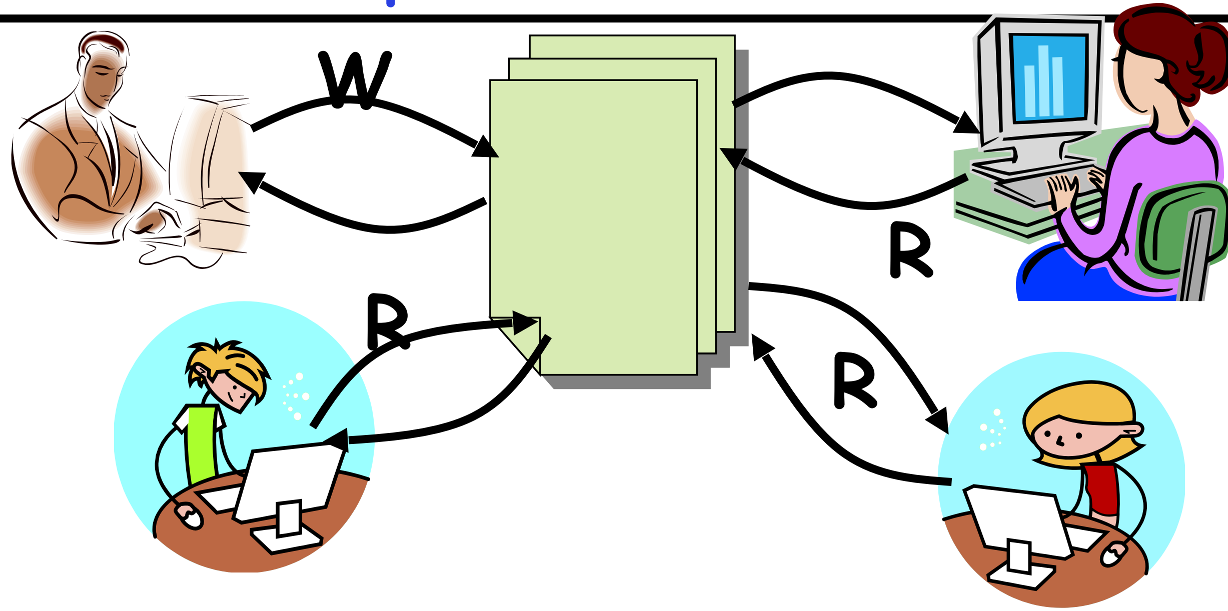
```
while (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();  // Get next item
```

  – Why didn't we do this?

```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();  // Get next item
```

- Answer: depends on the type of scheduling
  – Hoare-style (most textbooks):
    » Signaler gives lock, CPU to waiter; waiter runs immediately
    » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  – Mesa-style (most real operating systems):
    » Signaler keeps lock and processor
    » Waiter placed on ready queue with no special priority
    » Practically, need to check condition again after wait

Extended example: Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - `Reader()`

    ```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
    ```
  - `Writer()`

    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    okToWrite.signal();   // Wake up one writer
  lock.Release();
}
```

Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    okToWrite.signal();   // Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) { // Is it safe to read?
WR++;                   // No. Writers exist
okToRead.wait(&lock);   // Sleep on cond var
WR--;                   // No longer waiting
   }
AR++;                   // Now we are active!
```

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers make take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

Simulation(2)

- Next, W1 comes along:

```
while ((AW + AR) > 0) { // Is it safe to write?
   WW++;                // No. Active users exist
   okToWrite.wait(&lock);  // Sleep on cond var
   WW--;                // No longer waiting
}
AW++;
```

- Can't start because of readers, so go to sleep:

  AR = 2, WR = 0, AW = 0, WW = 1

- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

```
if (AR == 0 && WW > 0)  // No other active readers
   okToWrite.signal();  // Wake up one writer
```

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
if (WW > 0){                  // Give priority to writers
  okToWrite.signal();         // Wake up one writer
} else if (WR > 0) {          // Otherwise, wake reader
  okToRead.broadcast();       // Wake all readers
}
```

  – Writer wakes up reader, so get:

    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

- Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) {  // Is it safe to read?
   WR++;                 // No. Writers exist
   okToRead.wait(&lock); // Sleep on cond var
   WR--;                 // No longer waiting
   }
AR++;                    // Now we are active!
```

- What if we erase the condition check in Reader exit?

```
AR--;                    // No longer active
if (AR == 0 && WW > 0)   // No other active readers
   okToWrite.signal();   // Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

```
AR--;                    // No longer active
okToWrite.broadcast();   // Wake up one writer
```

- Finally, what if we use only one condition variable (call it "okToContinue") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
  lock.Release();
  semaphore.P();
  lock.Acquire();
}
Signal() { semaphore.V(); }
```

  - No: Condition vars have no history, semaphores have history:
    - » What if thread signals and no one is waiting?
    - » What if thread later waits?
    - » What if thread V's and no one is waiting?
    - » What if thread later does P?

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
  Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
    if semaphore queue is not empty
       semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

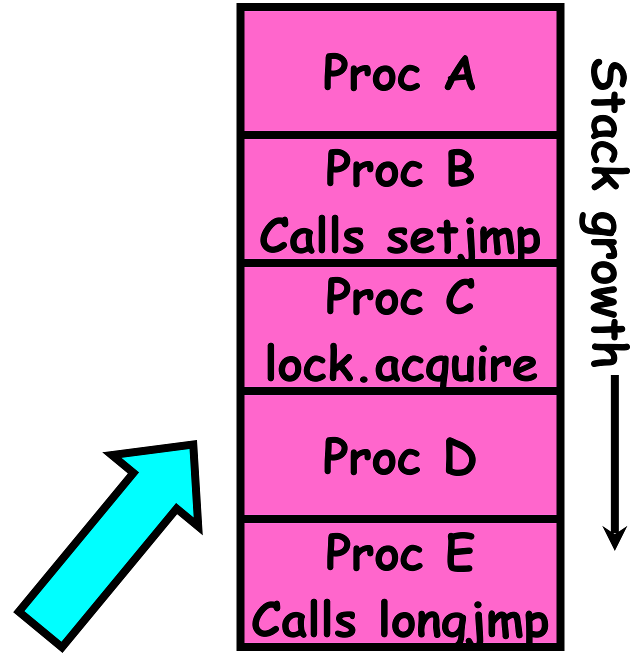
Check and/or update state variables
Wait if necessary

Check and/or update state variables

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know all the code paths out of a critical section

```
int Rtn() {
      lock.acquire();
      …
      if (exception) {
          lock.release();
          return errReturnCode;
      }
      …
      lock.release();
      return OK;
}
```



  - Watch out for `setjmp`/`longjmp`!
    - » Can cause a non-local jump out of procedure
    - » In example, procedure E calls longjmp, poping stack back to procedure B
    - » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

```
    void Rtn() {
    lock.acquire();
    …
    DoFoo();
    …
    lock.release();
  }
  void DoFoo() {
    …
    if (exception) throw errException;
    …
  }
```

  - Notice that an exception in DoFoo() will exit without releasing the lock!

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
   lock.acquire();
   try {
      …
      DoFoo();
      …
   } catch (…) {       // catch exception
      lock.release(); // release lock
      throw;          // re-throw the exception
   }
   lock.release();
}
void DoFoo() {
   …
   if (exception) throw errException;
   …
}
```

  - Even Better: auto_ptr<T> facility. See C++ Spec.
    » Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
   private int balance;
   // object constructor
   public Account (int initialBalance) {
      balance = initialBalance;
   }
   public synchronized int getBalance() {
      return balance;
   }
   public synchronized void deposit(int amount) {
      balance += amount;
   }
}
```

  – Every object has an associated lock which gets automatically acquired and released on entry and exit from a synchronized method.

Java Language Support for Synchronization (con't)

- **Java also has synchronized statements:**

```
synchronized (object) {
        …
}
```

  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:

```
    synchronized (object) {
    …
    DoFoo();
    …
}
void DoFoo() {
    throw errException;
}
```

Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method of block:
    - » `void wait(long timeout); // Wait for timeout`
    - » `void wait(long timeout, int nanoseconds); //variant`
  - How to signal in a synchronized method or block:
    - » `void notify();    // wakes up oldest waiter`
    - » `void notifyAll(); // like broadcast, wakes everyone`

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

`int mylock = FREE;`

`Acquire(&mylock)` – wait until lock is free, then grab

`Release(&mylock)` – Unlock, waking up anyone waiting

```
Acquire(int *lock) {
   disable interrupts;
   if (*lock == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      *lock = BUSY;
   }
   enable interrupts;
}
```

```
Release(int *lock) {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      *lock = FREE;
   }
   enable interrupts;
}
```

- Really only works in kernel – why?

In-Kernel Lock: Simulation

Value: 0 | waiters | owner

READY

Running

Thread A

Thread B

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

In-Kernel Lock: Simulation

Value: 1 | waiters | owner

READY

Running

Thread A

Thread B

INIT
```
   int value = 0;
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Running — Thread A

Running — Thread B

INIT
```
int value = 0;
```

Thread A:
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

Thread B:
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Running — Thread A

Running — Thread B

```
INIT
   int value = 0;
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Running

Thread A

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

Thread B

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

In-Kernel Lock: Simulation

Value: 1	waiters	owner	READY

Running

Thread A

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Running

Thread B

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

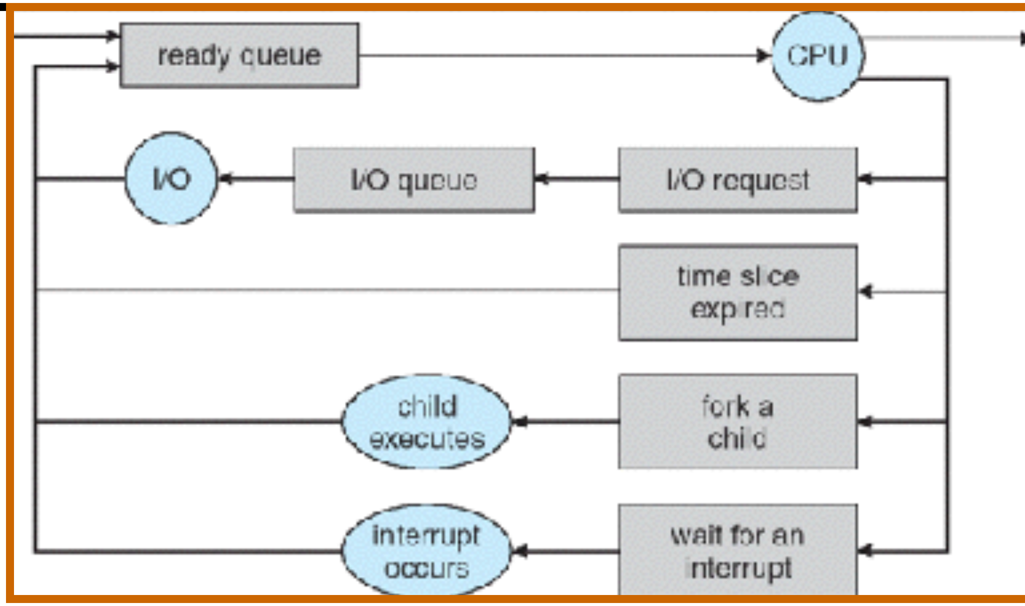
INIT

```
int value = 0;
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() // ??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Recall: CPU Scheduling

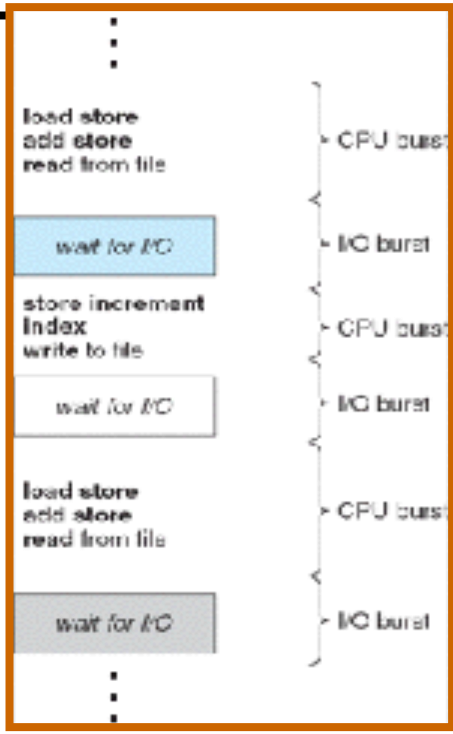


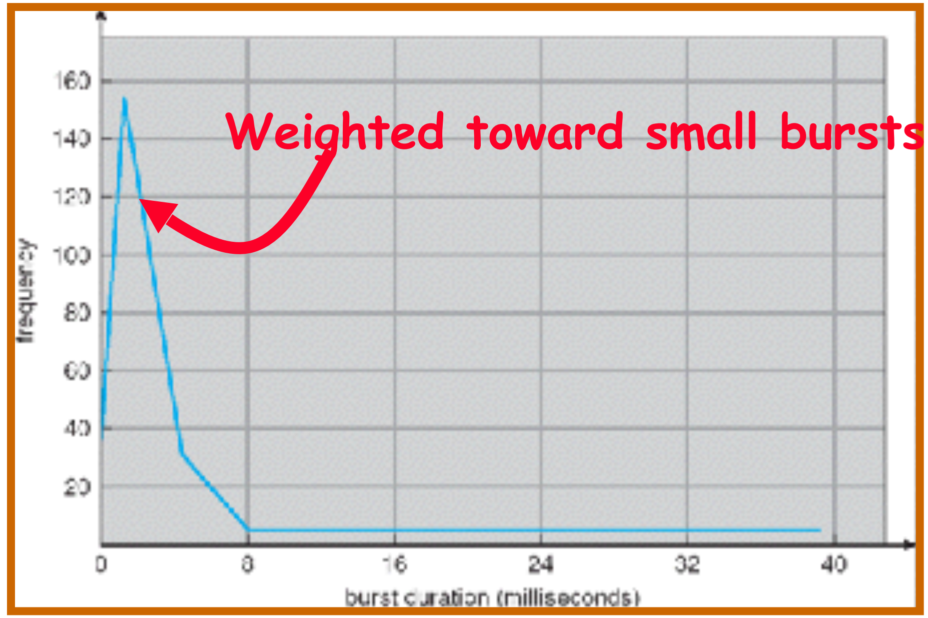
- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Assumption: CPU Bursts





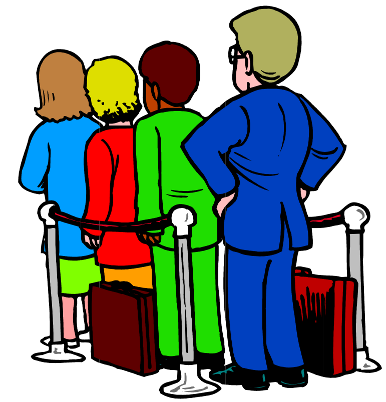
- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- **Minimize Response Time**
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- **Maximize Throughput**
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- **Fairness**
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better average response time by making system less fair

First-Come, First-Served (FCFS) Scheduling

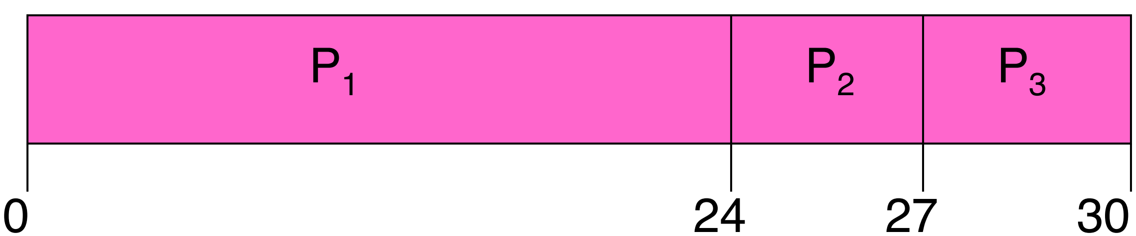
- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks



- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

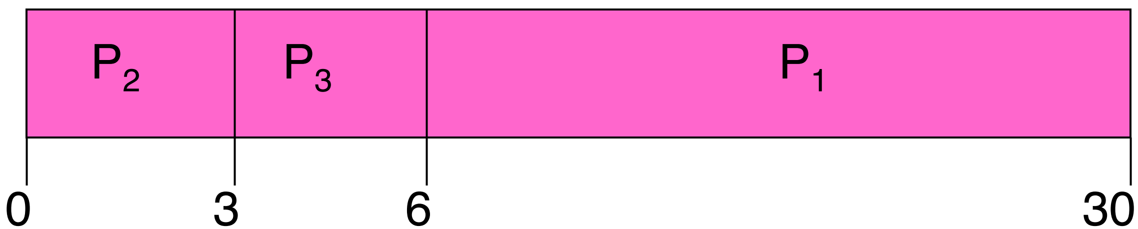
  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:



  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- Convoy effect: short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1
    Now, the Gantt chart for the schedule is:



  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: $(6 + 0 + 3)/3 = 3$
  - Average Completion time: $(3 + 6 + 30)/3 = 13$
- In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - » Safeway: Getting milk, always stuck behind cart full of small items. Upside: get to read about space aliens!

Summary

- **Semaphores**: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- **Monitors**: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- **Scheduling**: selecting a waiting process from the ready queue and allocating the CPU to it
- **FCFS Scheduling**:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- **Round-Robin Scheduling**:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length