Lecture 7: Synchronization (cont.)

Instructor: Mitch Neilsen

Office: N219D

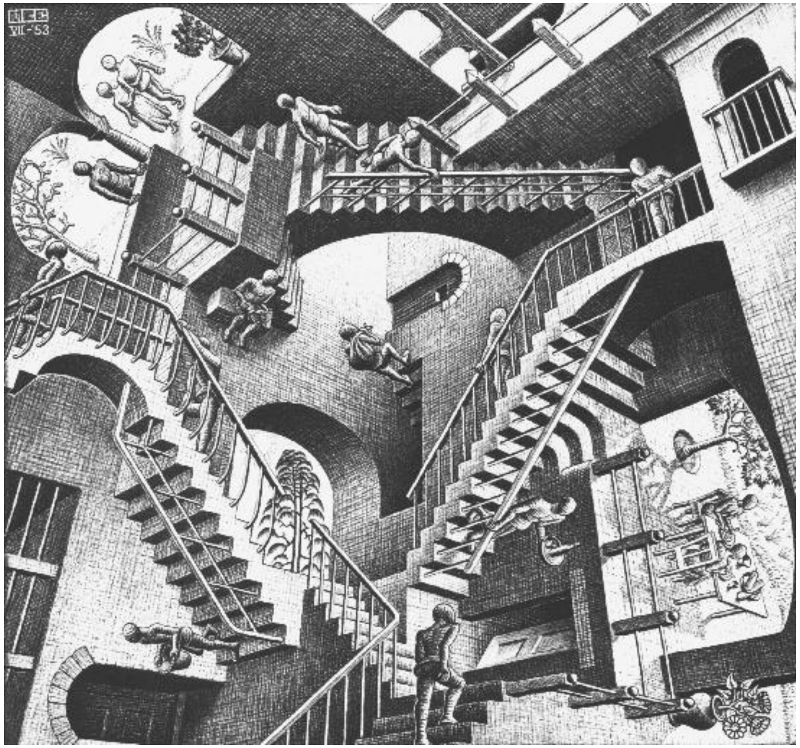
Outline

- Reading:
  - Ch. 4 - Threads
  - Ch. 5 - CPU Scheduling
  - Ch. 6 - Synchronization
- Project 1: Scheduling and Synchronization
  - Alarm Clock
  - Priority-based Scheduler
  - Synchronization and Priority Inheritance
  - [Extra Credit] MLFQ Scheduler

Quote of the Day

"Only those who attempt the absurd will achieve the impossible."

-- M. C. Escher



Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
            mutex_unlock (&mutex);  // <--- Why?
            thread_yield ();
            mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
            mutex_unlock (&mutex);
            thread_yield ();
            mutex_lock (&mutex);
        }

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        /* consume the item in nextConsumed */
    }
}
```

Condition variables

- **Busy-waiting in application is a bad idea**
  - Thread consumes CPU even when can't make progress
  - Unnecessarily slows other threads and processes
- **Better to inform scheduler of which threads can run**
- **Typically done with condition variables**
- void cond_init (cond_t *, ...);
  - Initialize
- void cond_wait (cond_t *c, mutex_t *m);
  - Atomically unlock m and sleep until c signaled
  - Then re-acquire m and resume executing
- void cond_signal (cond_t *c);
  void cond_broadcast (cond_t *c);
  - Wake one/all threads waiting on c

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
          cond_wait (&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
            cond_wait (&nonempty, &mutex);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        /* consume the item in nextConsumed */
    }
}
```

Condition variables (continued)

- **Why must** cond_wait **both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
}
```

Condition variables (continued)

- **Why must** cond_wait **both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
}
```

- **Can end up stuck waiting when bad interleaving**

```
PRODUCER                                    CONSUMER
while (count == BUFFER_SIZE);
mutex_unlock (&mutex);
                                              mutex_lock (&mutex);
                                              ...
                                              count--;
                                              cond_signal (&nonfull);
cond_wait (&nonfull);
```

Implementing synchronization

- **User-visible mutex is straight-forward data structure**

```
typedef struct mutex {
   bool is_locked;          /* true if locked */
   thread_id_t owner;       /* thread holding lock, if locked */
   thread_list_t waiters; /* threads waiting for lock */

   lower_level_lock_t lk; /* Protect above fields */
};
```

- **Need lower-level lock** lk **for mutual exclusion**
  - Internally, mutex_* functions bracket code with lock(mutex->lk) . . . unlock(mutex->lk)
  - Otherwise, data races! (E.g., two threads manipulating waiters)
- **How to implement** lower_level_lock_t**?**
  - Could use Peterson's algorithm, but typically a bad idea (too slow and don't know maximum number of threads)

Approach #1: Disable interrupts

- **Only for apps with** $n:1$ **threads (1 kthread)**
  - Cannot take advantage of multiprocessors
  - But sometimes most efficient solution for uniprocessors
- **Have per-thread "do not interrupt" (DNI) bit**
- lock (lk)**: sets thread's DNI bit**
- **If timer interrupt arrives**
  - Check interrupted thread's DNI bit
  - If DNI clear, preempt current thread
  - If DNI set, set "interrupted" (I) bit & resume current thread
- unlock (lk)**: clears DNI bit *and* checks I bit**
  - If I bit is set, immediately yields the CPU

Approach #2: Spinlocks

- **Most CPUs support atomic read-[modify-]write**
- **Example:** int test_and_set (int *lockp);
  - Atomically sets *lockp = 1 and returns old value
  - Special instruction – can't be implemented in portable C
- **Use this instruction to implement *spinlocks*:**

```
#define lock(lockp)     while (test_and_set (lockp))
#define trylock(lockp)  (test_and_set (lockp) == 0)
#define unlock(lockp)   *lockp = 0
```

- **Spinlocks implement mutex's** lower_level_lock_t
- **Can you use spinlocks instead of mutexes?**
  - Wastes CPU, especially if thread holding lock not running
  - Mutex functions have short C.S., less likely to be preempted
  - On multiprocessor, sometimes good to spin for a bit, then yield

Synchronization on x86

- **Test-and-set only one possible atomic instruction**
- **x86** xchg **instruction, exchanges reg with mem**
  - Can use to implement test-and-set

```
_test_and_set:
        movl    8(%esp), %edx      # %edx = lockp
        movl    $1, %eax           # %eax = 1
        xchgl   %eax, (%edx)       # swap (%eax, *lockp)
        ret
```

- **CPU locks memory system around read and write**
  - Recall xchgl always acts like it has lock prefix
  - Prevents other uses of the bus (e.g., DMA)
- **Usually runs at memory bus speed, not CPU speed**
  - Much slower than cached read/buffered write

Kernel Synchronization

- **Should kernel use locks or disable interrupts?**
- **Old UNIX had non-preemptive threads, no mutexes**
  - Interface designed for single CPU, so count++ etc. not data race
  - . . . *Unless* memory shared with an interrupt handler

```
int x = splhigh (); // Disable interrupts
// Touch data shared with interrupt handler
splx (x);           // Restore previous state
```

  - C.f., Pintos intr_disable / intr_set_level
- **Used arbitrary pointers like condition variables**
  - int [t]sleep (void *ident, int priority, ...);
    put thread to sleep; will wake up at priority (~cond_wait)
  - int wakeup (void *ident);
    wake up all threads sleeping on ident (~cond_broadcast)

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses locks
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses locks
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**
  - Yes! Can't sleep in interrupt handler, so can't wait for lock
  - So even modern OSes have support for disabling interrupts
  - Often uses DNI trick, which is cheaper than masking interrupts in hardware

Semaphores [Dijkstra]

- **A *Semaphore* is initialized with an integer** N
- **Provides two functions:**
  - sem_wait (S) (originally called P, called ***sema_down*** in **Pintos**)
  - sem_signal (S) (originally called V, called ***sema_up*** in **Pintos**)
- **Guarantees** sem_wait **will return only** N **more times than** sem_signal **called**
  - Example: If $N == 1$, then semaphore is a mutex with sem_wait as lock and sem_signal as unlock
- **Semaphores allow elegant solutions to some problems**

Semaphore

A semaphore is a structure consisting of 2 parts:

```
struct semaphore {
    int count;  // number of resources available
    queue Q;  // queue of process/thread ids of blocked
}
```

Shorthand notation:

semaphore S = 1 → S.count = 1, S.Q = { }

Operations on Semaphores

There are two basic semaphore operations:

sem_wait(S):

    if (S.count > 0) then S.count = S.count -1;

    else block calling process in S.Q;

sem_signal(S):

    if (S.Q is non-empty) then wakeup a process in S.Q;

    else S.count = S.count + 1;

Semaphore Example: Mutual Exclusion

Semaphore S = 1;

Thread A:	Thread B:
sem_wait(S);	sem_wait(S);
(do work in critical section CS);	(do work in CS);
sem_signal(S);	sem_signal(S);

Semaphore Example: Order Execution

Semaphore S = 0;

Thread A → Thread B:

Thread A:

(do work);

sem_signal(S);

Thread B:

sem_wait(S);

(do work);

Semaphore producer/consumer

- **Can re-write producer/consumer to use three semaphores**
- **Semaphore** mutex **initialized to 1**
  - Used as mutex, protects buffer, in, out. . .
- **Semaphore** full **initialized to 0**
  - To block consumer when buffer empty
- **Semaphore** empty **initialized to N**
  - To block producer when queue full

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        sem_wait (&empty);
        sem_wait (&mutex);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&mutex);
        sem_signal (&full);
    }
}

void consumer (void *ignored) {
    for (;;) {
        sem_wait (&full);
        sem_wait (&mutex);
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&mutex);
        sem_signal (&empty);
        /*  consume the item in nextConsumed */
    }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Synchronization Problems

- Bounded-Buffer (Producer-Consumer) Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (cont.)

- The structure of a writer process

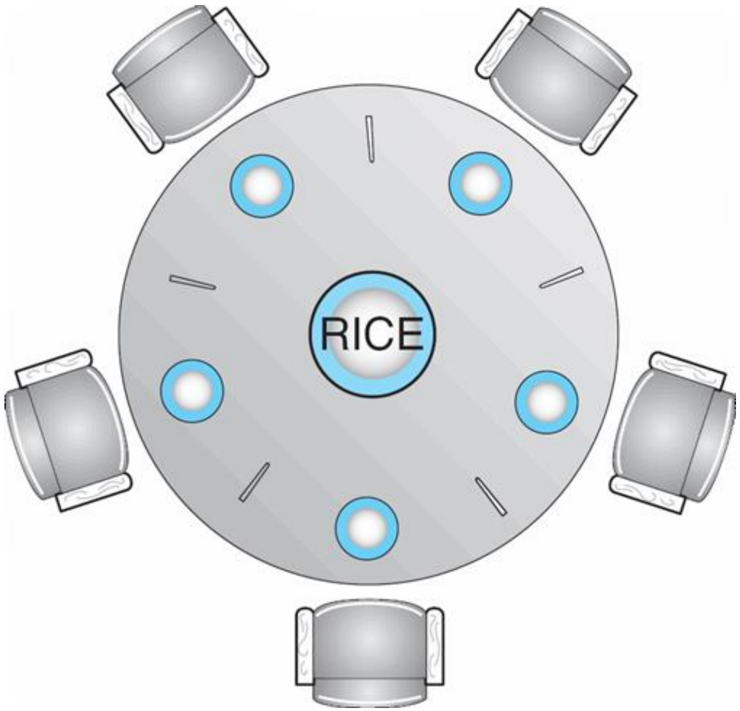
```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
            // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore **chopstick [5]** each initialized to 1

Dining-Philosophers Problem (cont.)

- The structure of Philosopher *i*:

```
do  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

              //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

              //  think

} while (TRUE);
```

Problems with Semaphores

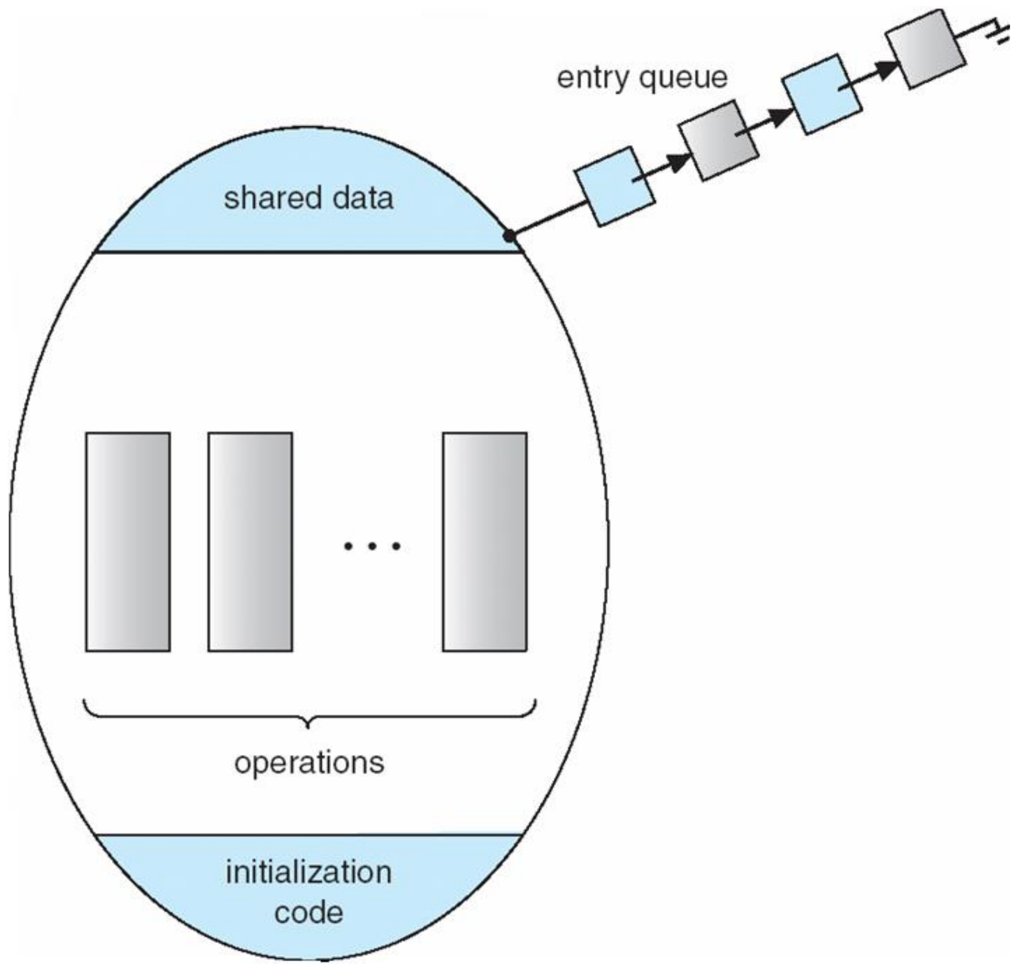
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
            …
   procedure Pn (…) {……}

    initialization code ( ….) { … }
            …
}
```

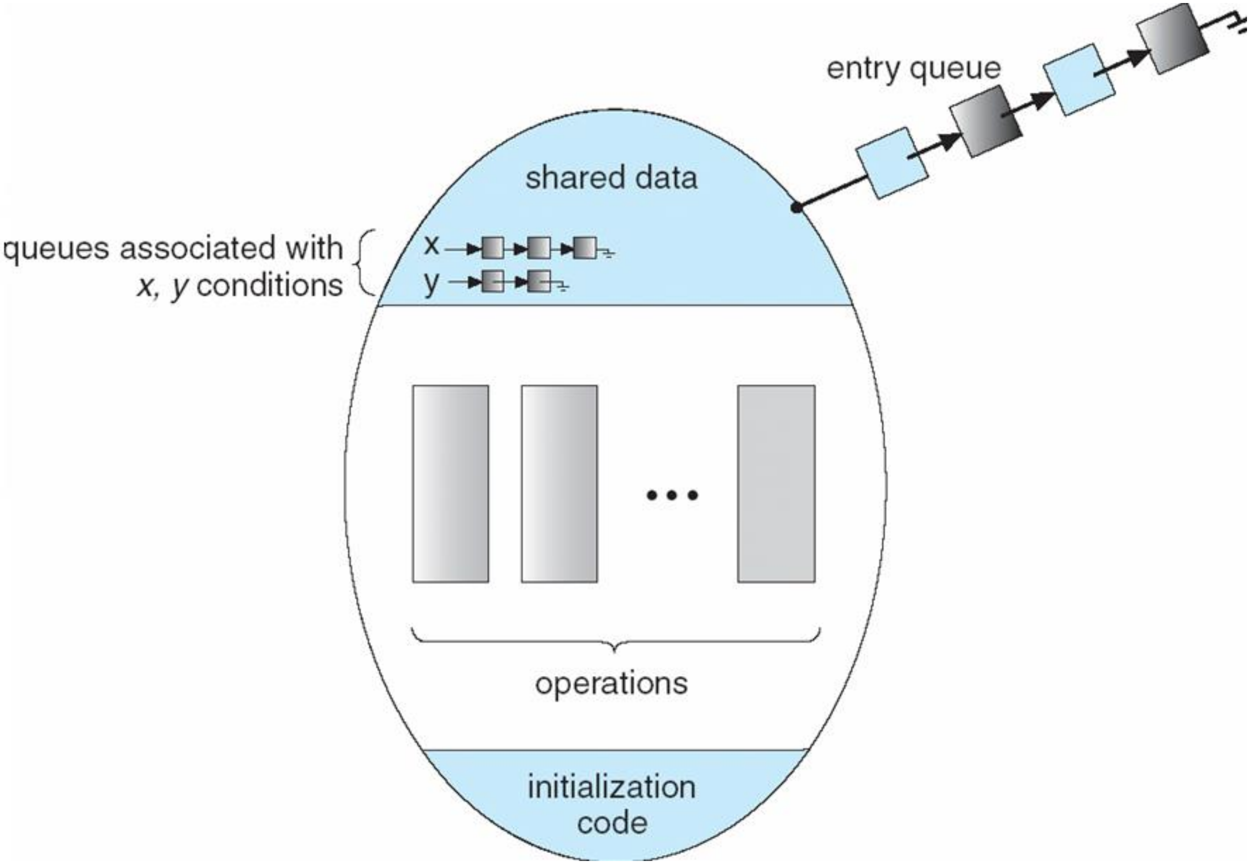
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
 {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
         state[i] = THINKING;
                // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
     }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
             self[i].signal () ;
         }
 }

  initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
 }
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
                  body of F;

    …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

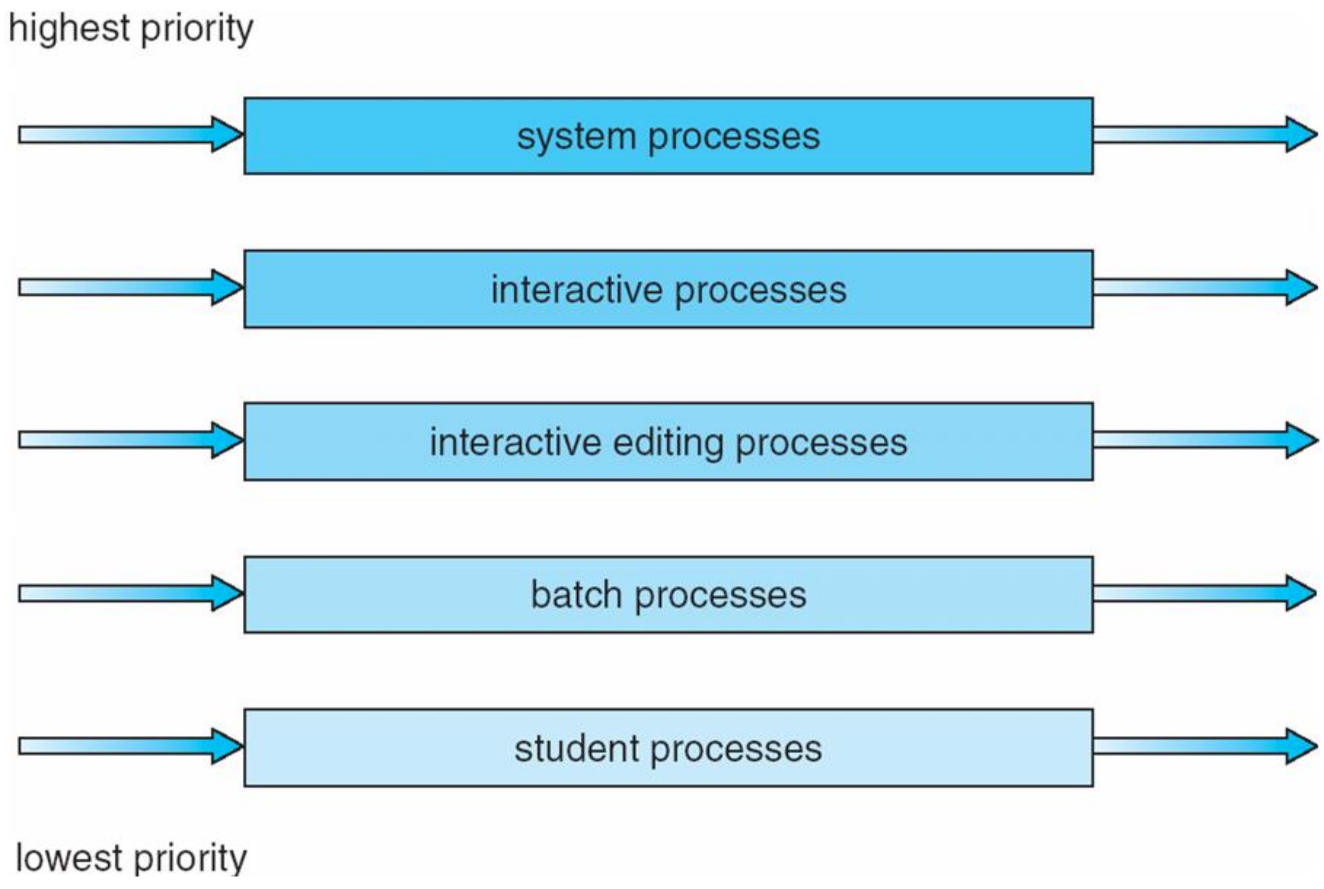
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spin locks

Multilevel Queue Scheduling



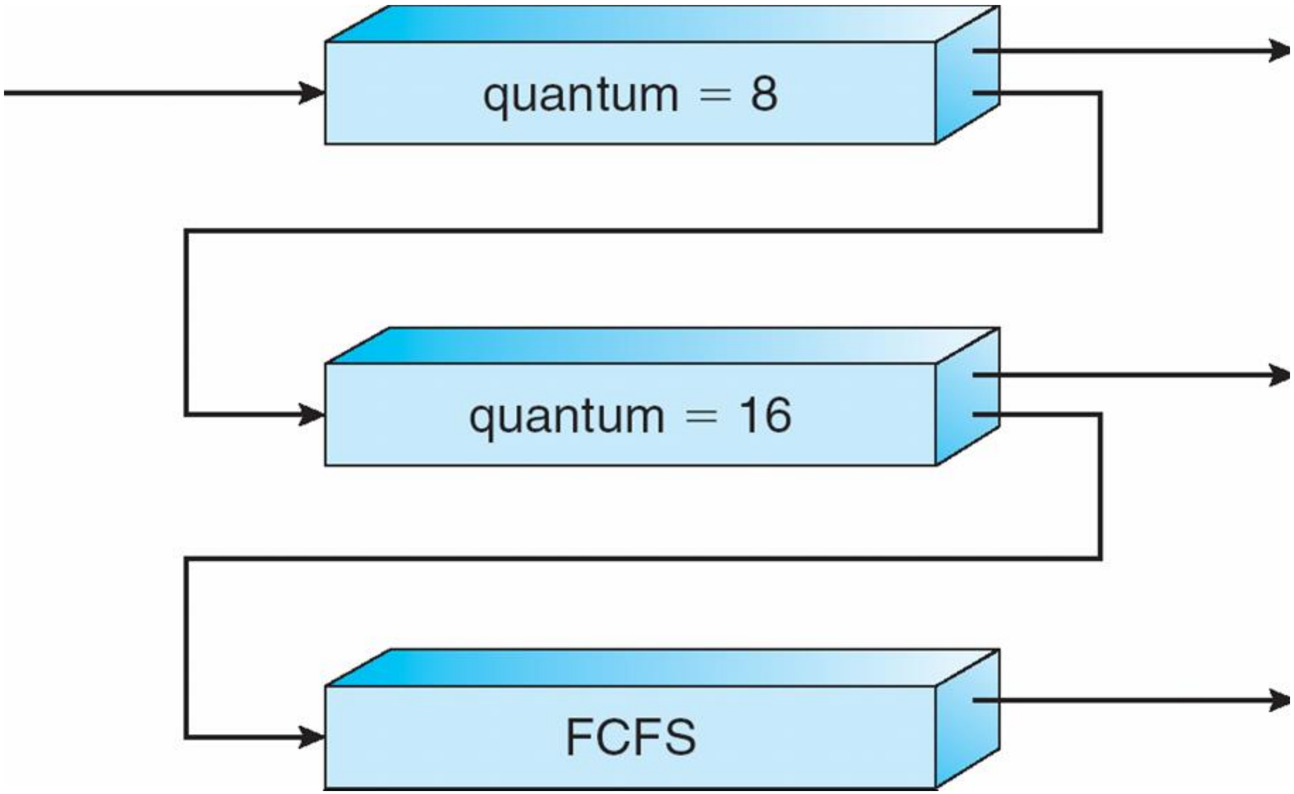
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

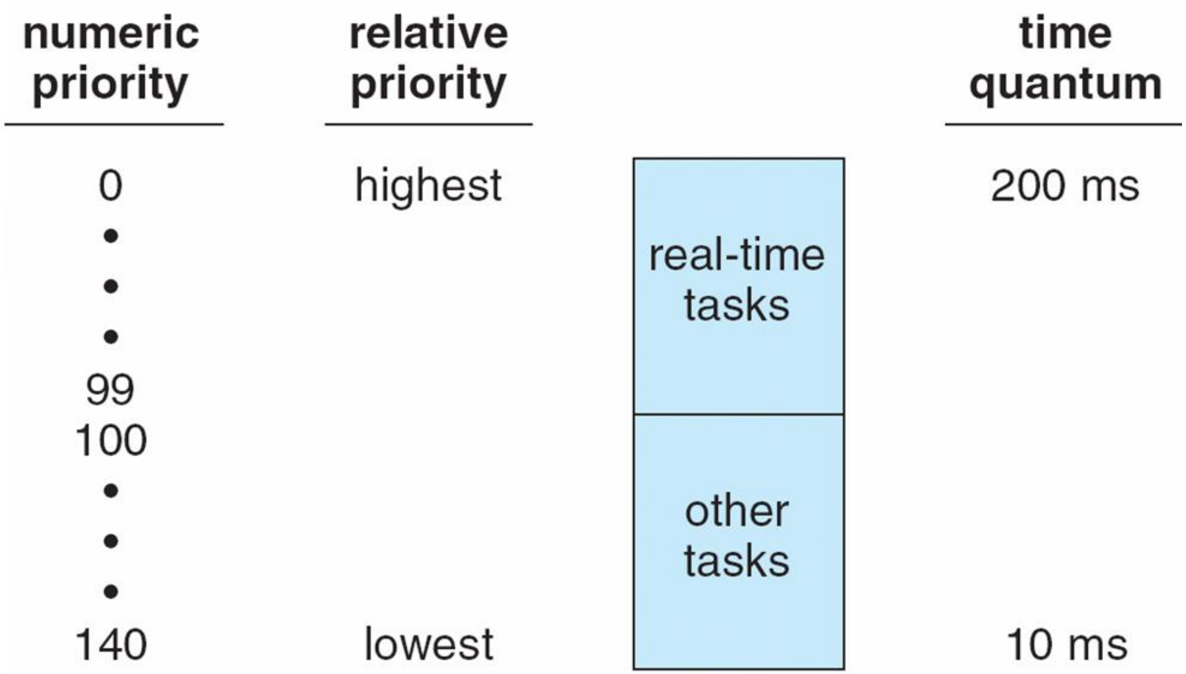
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Linux Scheduling

- Constant order *O*(1) scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140



Summary

- Read Ch. 1-6
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Project 1 – Scheduling and Synchronization