# Midterm Exam Review

Computer System Structure

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers

Four Components of a Computer System
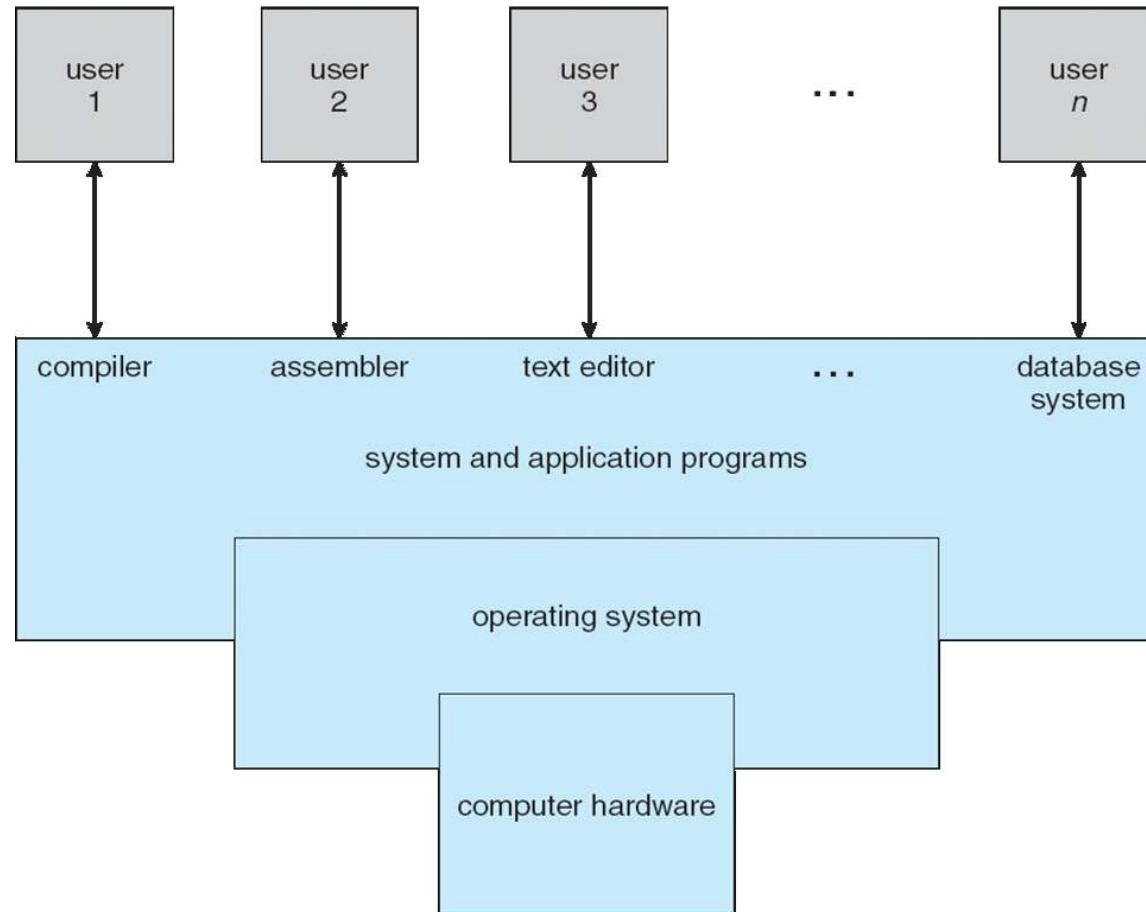
Operating System Definition

- OS is a **resource allocator**
    - Manages all resources
    - Decides between conflicting requests for efficient and fair resource use

- OS is a **control program**
    - Controls execution of programs to prevent errors and improper use of the computer

Computer System Organization

- Computer-system operation

  - One or more CPUs, device controllers connect through common bus providing access to shared memory

  - Concurrent execution of CPUs and devices competing for memory cycles. A memory controller synchronizes access to the memory.
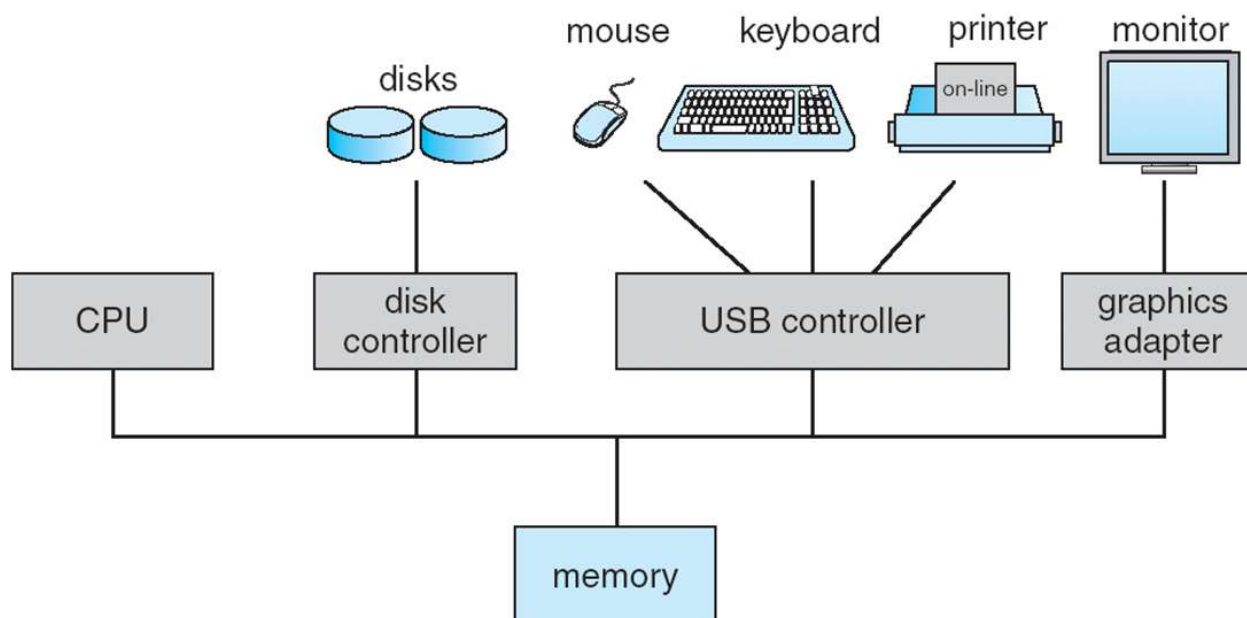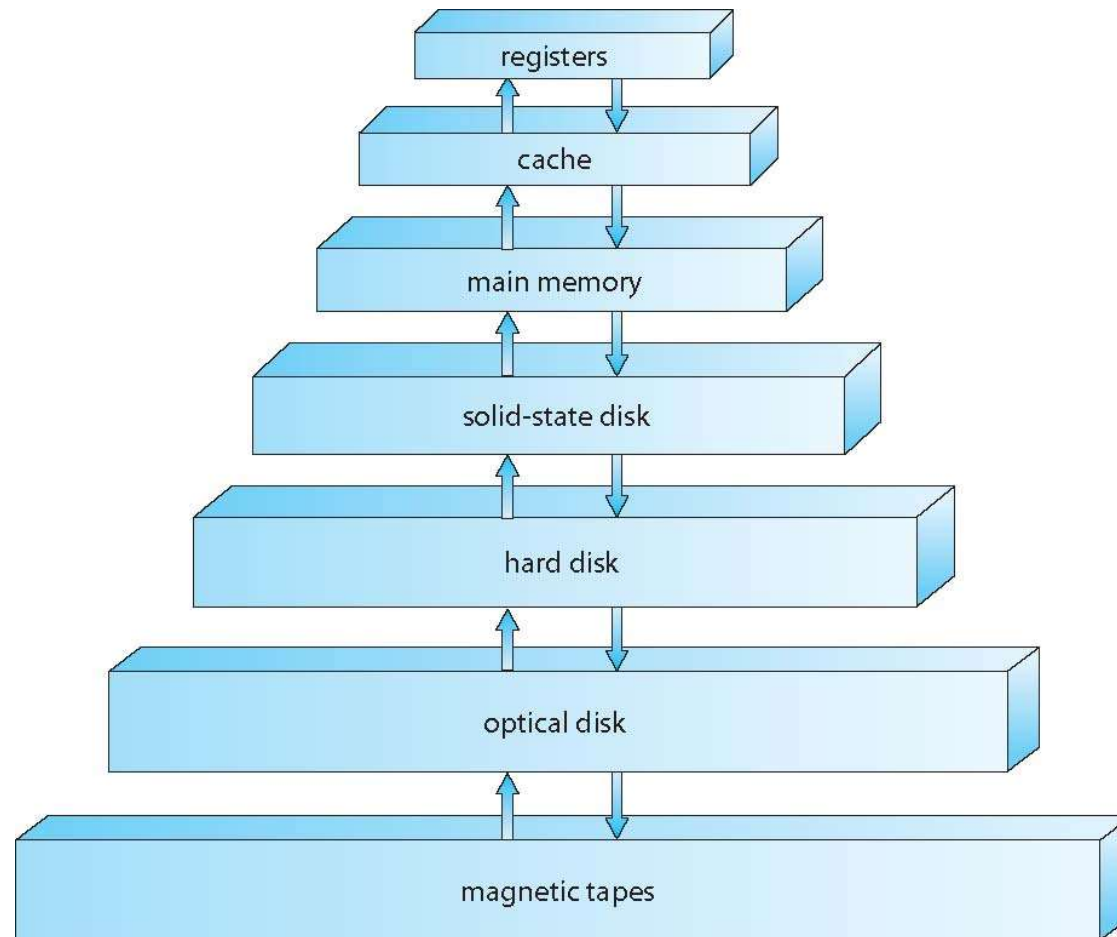
Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Storage-Device Hierarchy

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)

- Information in use copied from slower to faster storage temporarily

- Faster storage (cache) checked first to determine if information is there

    - If it is, information used directly from the cache (fast)

    - If not, data copied to cache and used there

- Cache smaller than storage being cached

    - Cache management important design problem

    - Cache size and replacement policy
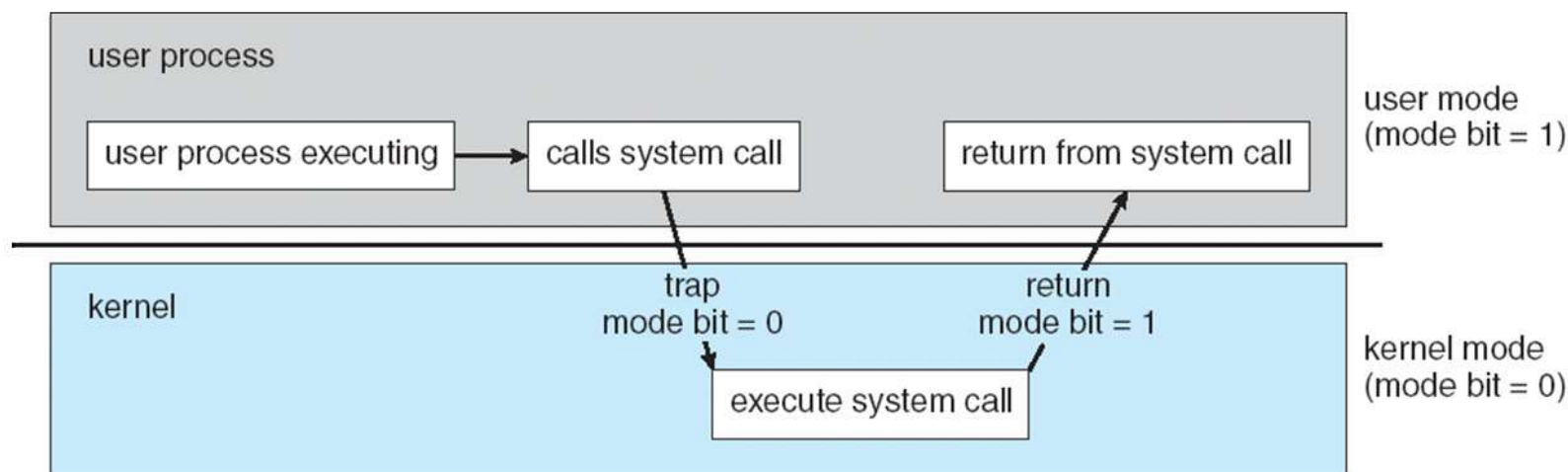
Operating System Structure

- **Multiprogramming** (**Batch system**) needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job

- **Timesharing** (**multitasking**) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

Operating-System Operations (cont.)

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware (e.g., CS register in CPU)
    - ▸ Provides ability to distinguish when system is running user code or kernel code
    - ▸ Some instructions designated as **privileged**, only executable in kernel mode
    - ▸ System call changes mode to kernel, return from call resets it to user

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users

- One set of operating-system services provides functions that are helpful to the user:

  - **User interface** - Almost all operating systems have a user interface (**UI**).

    - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **Batch**

  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)

  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):

  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

  - **Communications** – Processes may exchange information, on the same computer or between computers over a network

    - Communications may be via shared memory or through message passing (packets moved by the OS)

  - **Error detection** – OS needs to be constantly aware of possible errors

    - May occur in the CPU and memory hardware, in I/O devices, in user program

    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing

    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
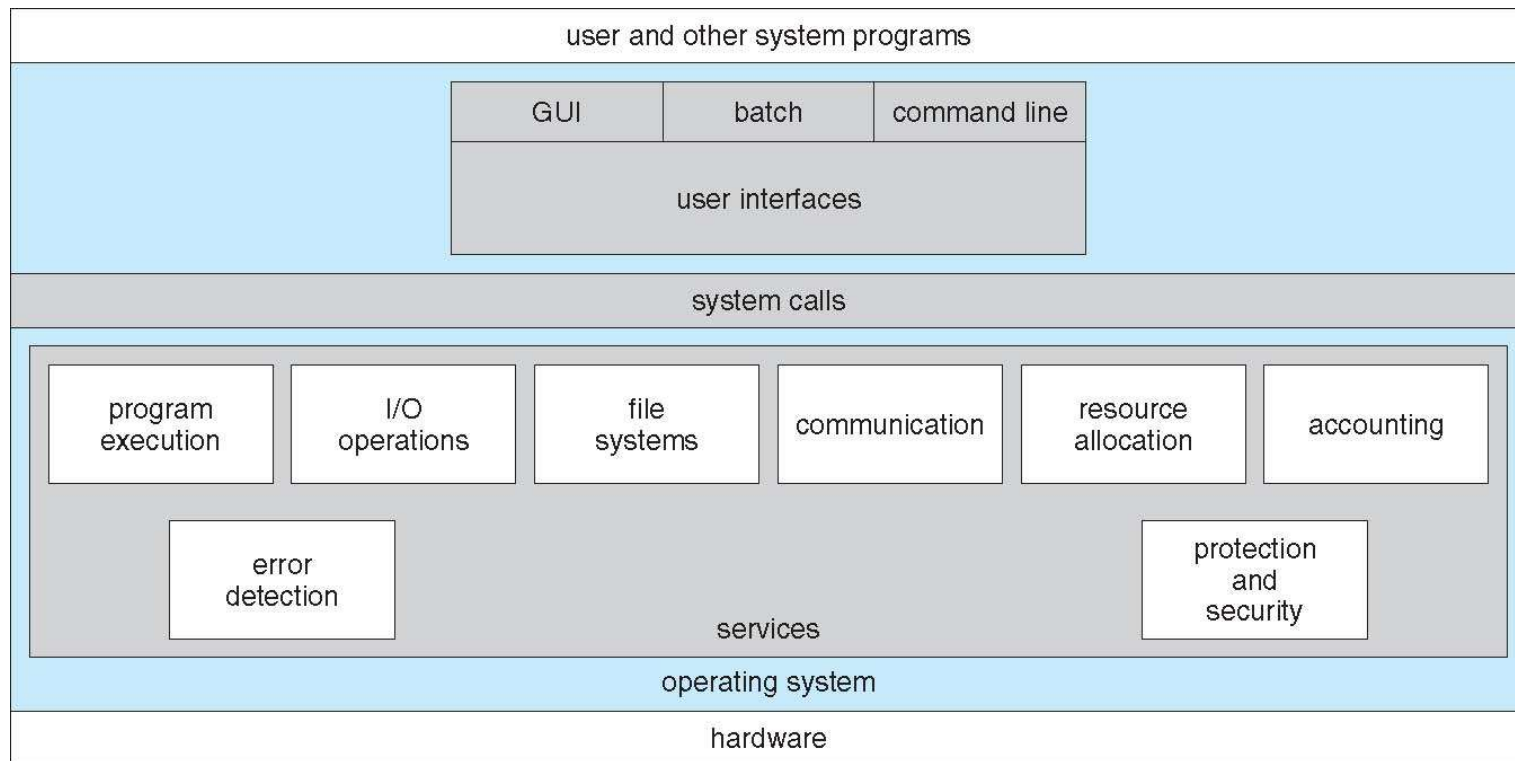
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - ▸ Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - ▸ **Protection** involves ensuring that all access to system resources is controlled
    - ▸ **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services

user and other system programs		

GUI	batch	command line
user interfaces		

system calls

program execution	I/O operations	file systems	communication	resource allocation	accounting

error detection				protection and security

services

operating system

hardware

System Calls

- Programming interface to the services provided by the OS

- Typically written in a high-level language (C or C++)

- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use

- Three most common APIs are Windows API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
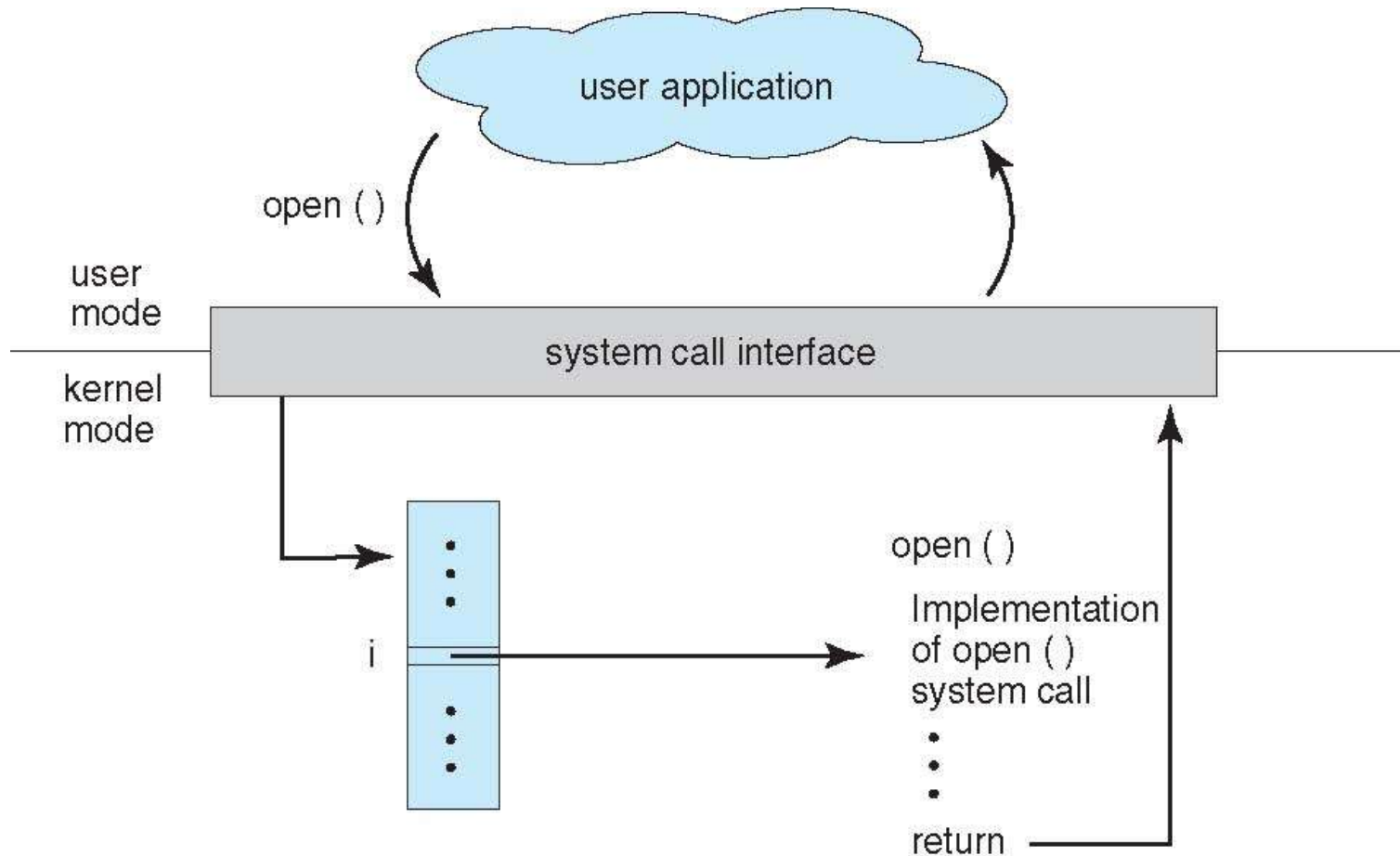
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship

Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful

- Internal structure of different Operating Systems can vary widely

- Start the design by defining goals and specifications

- Highest level: affected by choice of hardware, type of system

- The requirements can be divided into **User** and **System** goals

  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast

  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
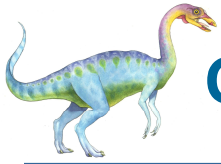
Operating System Design and Implementation (Cont.)

- Important principle to separate

  Policy: *What* will be done?
  Mechanism: *How* to do it?

- Mechanisms determine how to do something, policies decide what will be done

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

- Specifying and designing an OS is highly creative task of **software engineering**

Operating System Structure

- General-purpose OS is very large program

- Various ways to structure ones

  - Simple structure – MS-DOS

  - More complex -- UNIX

  - Layered – an abstrcation

  - Microkernel -Mach

Process Concept

- An operating system executes a variety of programs:
    - Batch system – **jobs**
    - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms *job* and *process* almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
    - The program code, also called **text section**
    - Current activity including **program counter**, processor registers
    - **Stack** containing temporary data
        - Function parameters, return addresses, local variables
    - **Data section** containing global variables
   - **Heap** containing memory dynamically allocated during run time

Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State
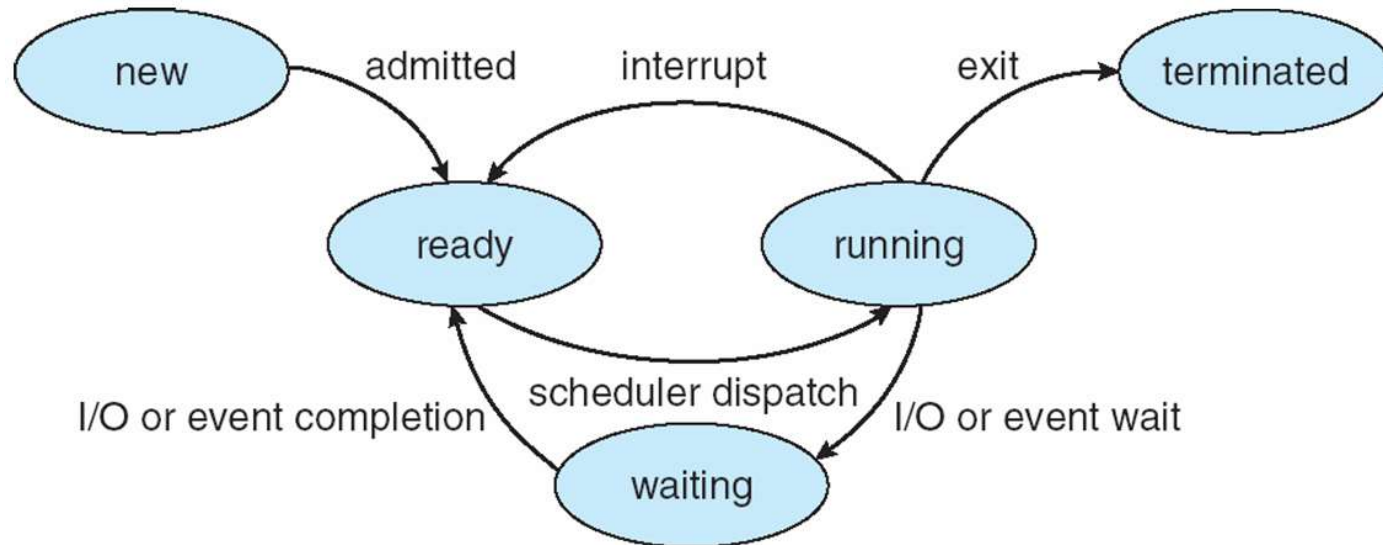
Process Control Block (PCB)

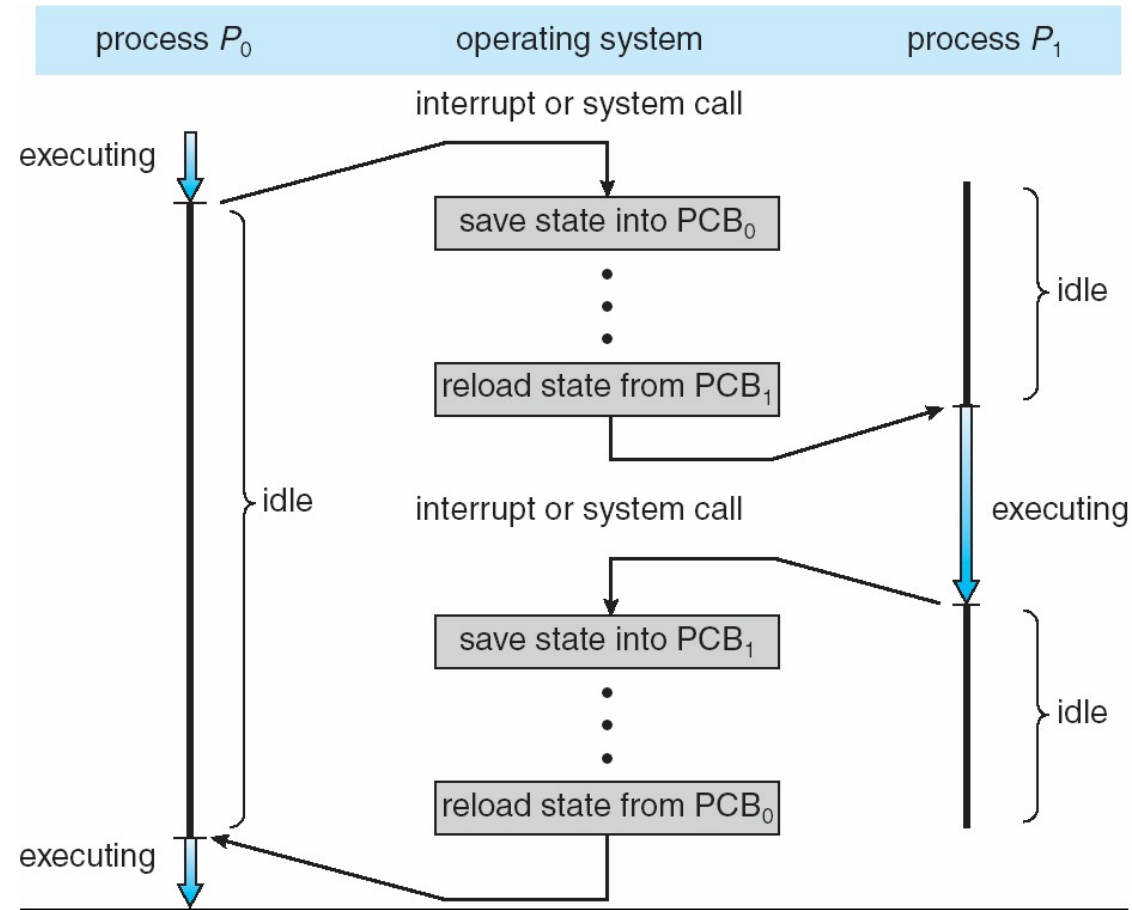Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc

- Program counter – location of instruction to next execute

- CPU registers – contents of all process-centric registers

- CPU scheduling information- priorities, scheduling queue pointers

- Memory-management information – memory allocated to the process

- Accounting information – CPU used, clock time elapsed since start, time limits

- I/O status information – I/O devices allocated to process, list of open files

| process state |
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| • • • |

CPU Switch From Process to Process

process P_0	operating system	process P_1

interrupt or system call

executing

save state into PCB_0

· · ·

reload state from PCB_1

idle

idle

interrupt or system call

executing

save state into PCB_1

· · ·

reload state from PCB_0

idle

executing

Process Scheduling

☐ Maximize CPU use, quickly switch processes onto CPU for time sharing

☐ **Process scheduler** selects among available processes for next execution on CPU

☐ Maintains **scheduling queues** of processes

  ☐ **Job queue** – set of all processes in the system

  ☐ **Ready queue** – set of all processes residing in main memory, ready and waiting to execute

  ☐ **Device queues** – set of processes waiting for an I/O device
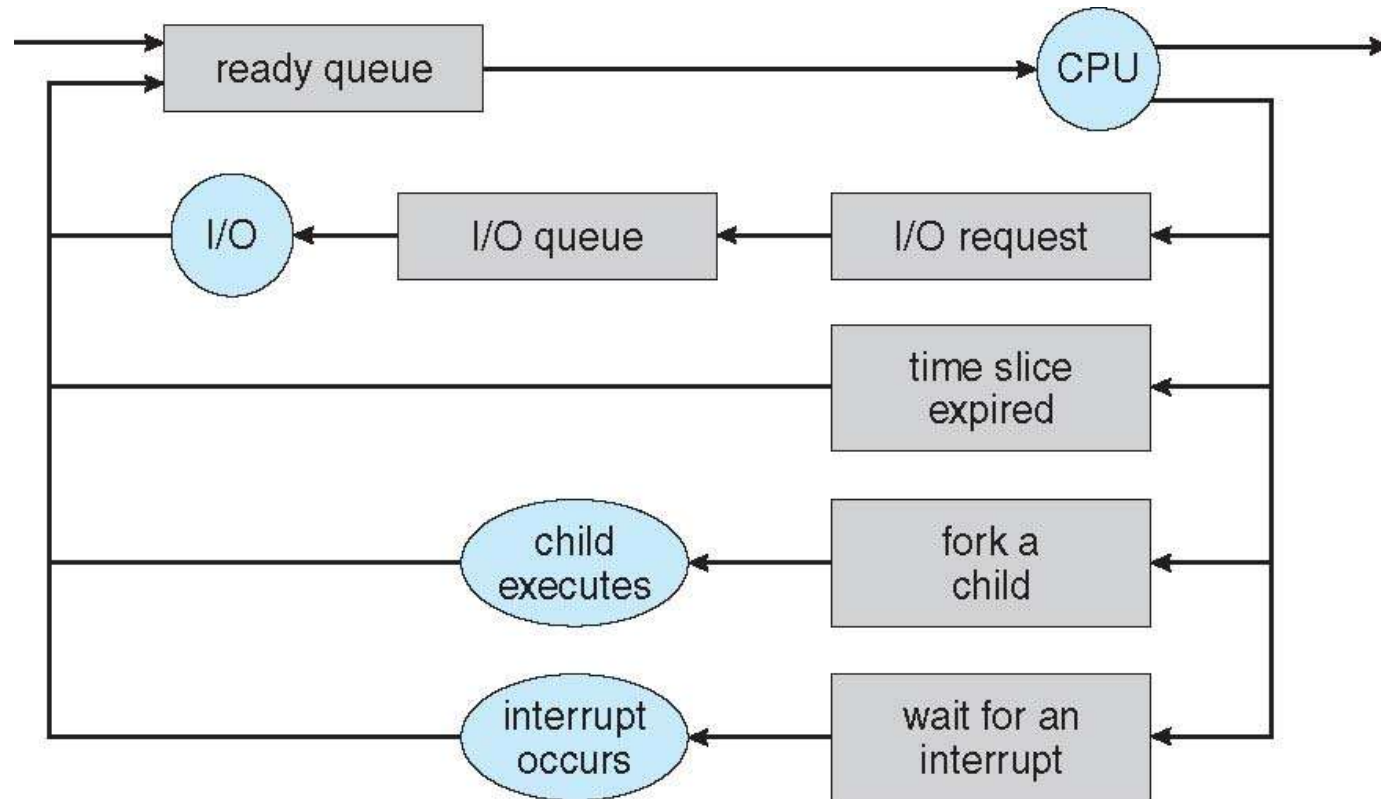
  ☐ Processes migrate among the various queues
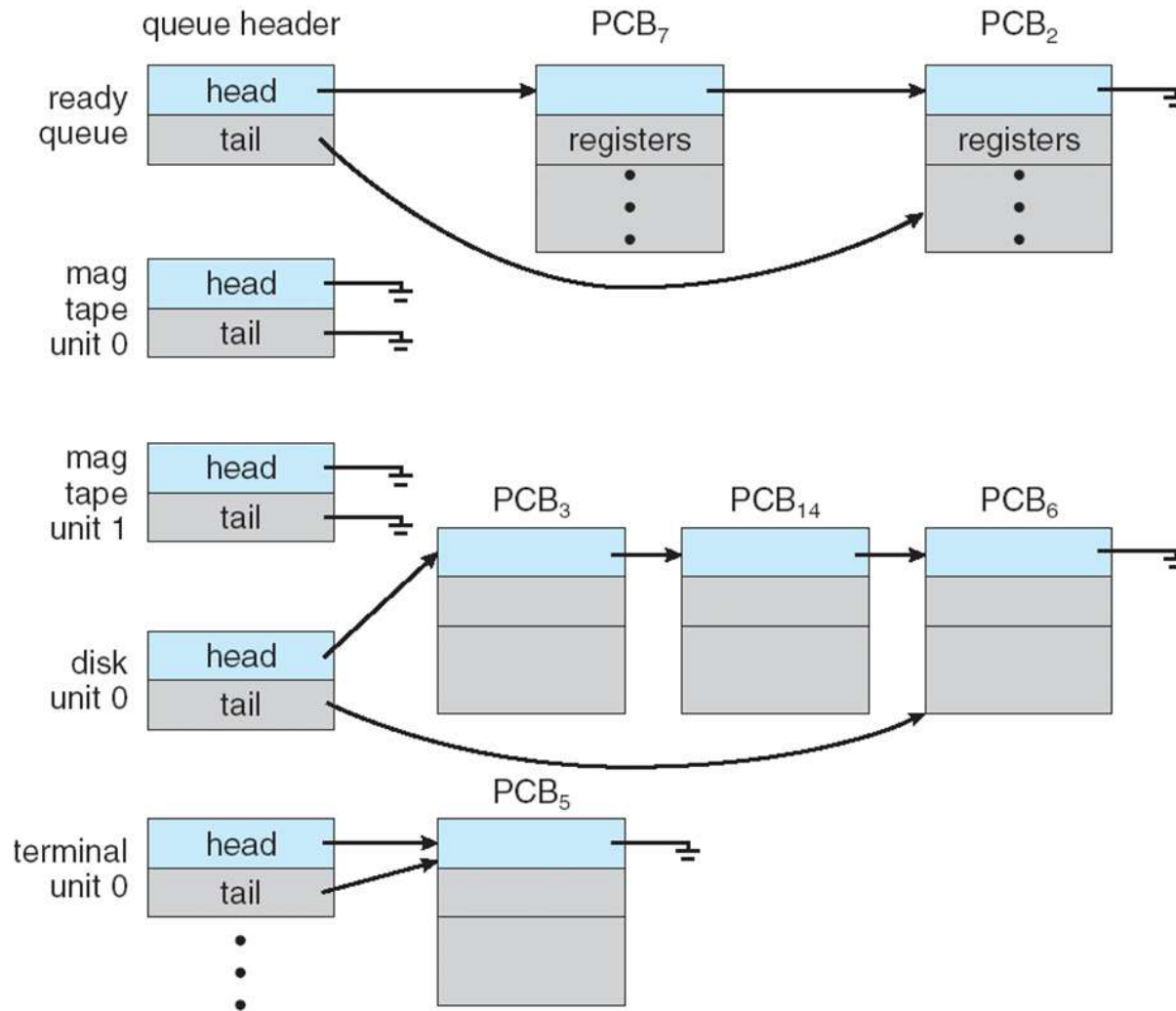
Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
    - Sometimes the only scheduler in a system
    - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
    - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
    - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
    - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix*

Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

- Generally, process identified and managed via a **process identifier** (**pid**)

- Resource sharing options

    - Parent and children share all resources

    - Children share subset of parent's resources

    - Parent and child share no resources

- Execution options

    - Parent and children execute concurrently

    - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent (has the same program as the parent)
  - Child has a program loaded into it
- UNIX examples
  - **fork()** system call creates new process. The new process consists of a copy of the address space of the original process.
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program

move itself off the ready queue until the termination of the child
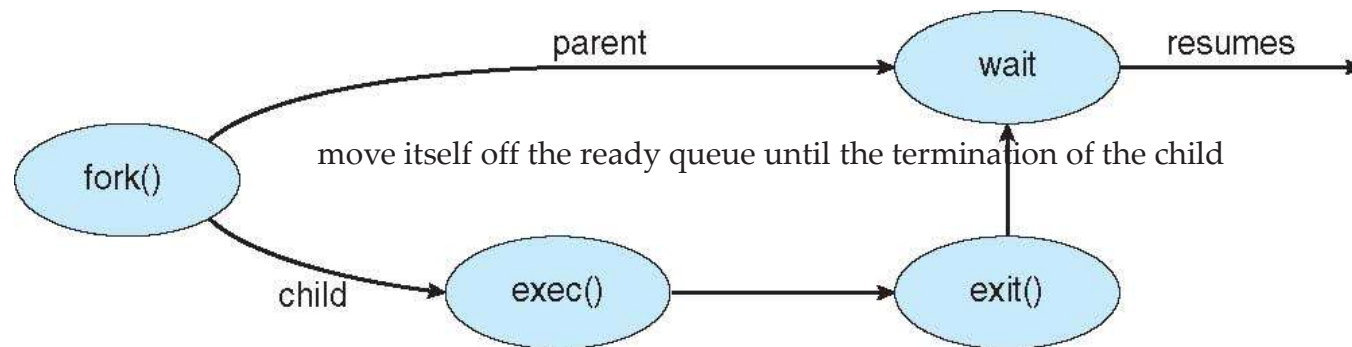
C Program Forking Separate Process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```
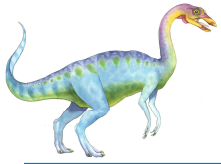
The only difference is that the value of pid for the child process is zero, while that for the parent is the actual pid of the child process.

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Interprocess Communication

- Processes within a system may be *independent* or *cooperating*

- Cooperating process can affect or be affected by other processes, including sharing data

- Reasons for cooperating processes:

  - Information sharing (shared files)

  - Computation speedup (parallel subtasks)

  - Modularity (system function divided into separate processes)

  - Convenience

- Cooperating processes need **interprocess communication** (**IPC**)

- Two models of IPC

  - **Shared memory**

  - **Message passing**

Communications Models

(a) Message passing.   (b) shared memory.

process A
process B

message queue					
m_0	m_1	m_2	m_3	...	m_n

kernel

(a)

process A
shared memory
process B

kernel

(b)
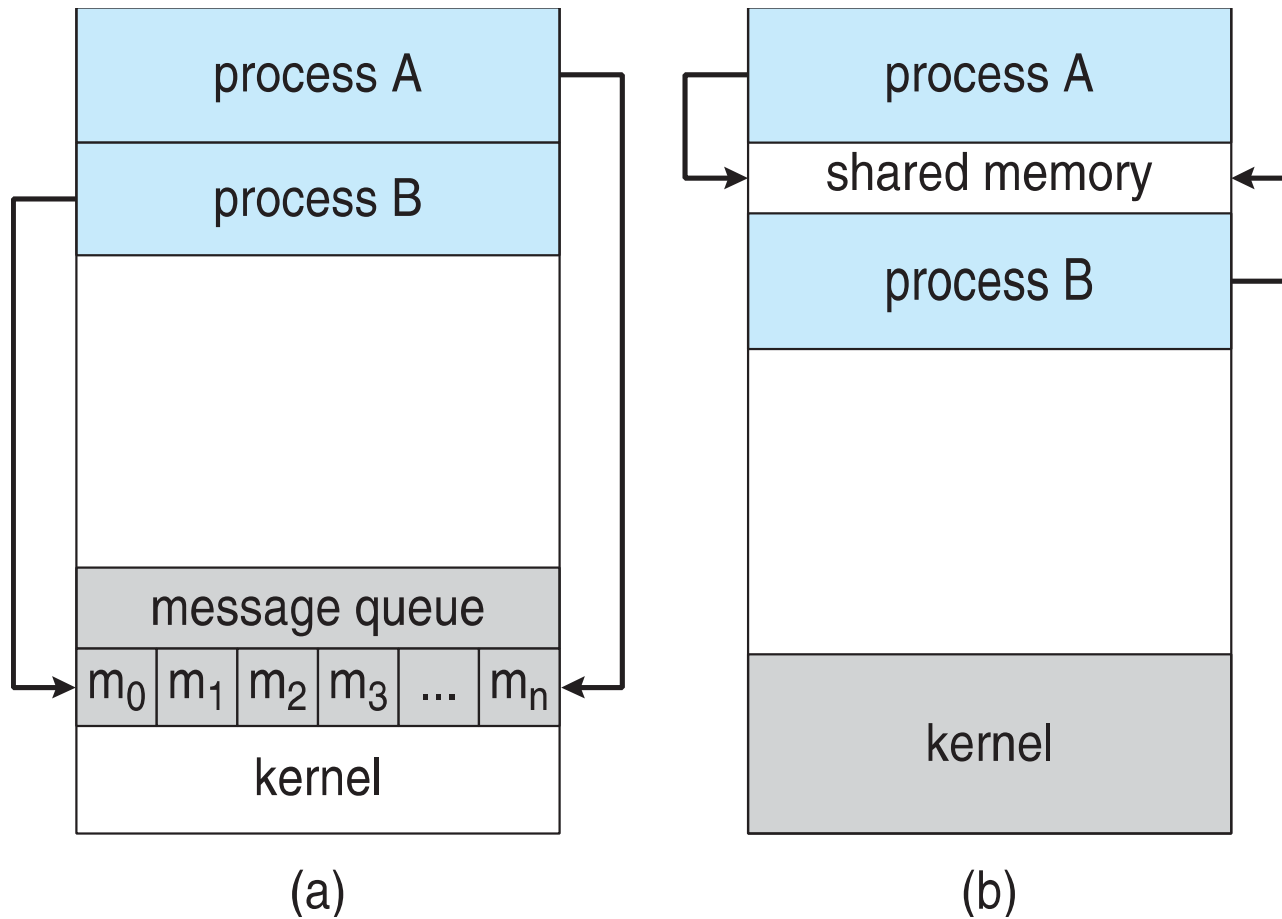
Race Condition

- **`counter++`** could be implemented as

    ```
    register1 = counter
    register1 = register1 + 1
    counter = register1
    ```

- **`counter--`** could be implemented as

    ```
    register2 = counter
    register2 = register2 - 1
    counter = register2
    ```

- Consider this execution interleaving with "count = 5" initially:

    S0: producer execute `register1 = counter`   {register1 = 5}
    S1: producer execute `register1 = register1 + 1`   {register1 = 6}
    S2: consumer execute `register2 = counter`   {register2 = 5}
    S3: consumer execute `register2 = register2 – 1`   {register2 = 4}
    S4: producer execute `counter = register1`   {counter = 6 }
    S5: consumer execute `counter = register2`   {counter = 4}

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$

- Each process has **critical section** segment of code
    - Process may be changing common variables, updating table, writing file, etc
    - When one process in critical section, no other may be in its critical section

- ***Critical section problem*** is to design protocol to solve this problem

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the *n* processes

Peterson's Solution

- Good algorithmic description of solving the problem

- Two process solution

- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted

- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`

- The variable `turn` indicates whose turn it is to enter the critical section

- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Algorithm for Process P_i
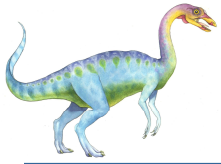
```
do {
```

```
        flag[i] = true;

        turn = j;

        while (flag[j] && turn = = j);
```

```
                critical section
```

```
        flag[i] = false;
```

```
                remainder section

} while (true);
```
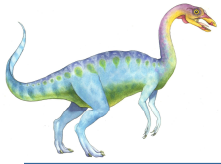
Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.

- All solutions below based on idea of **locking**

  - Protecting critical regions via locks

- Uniprocessors – could disable interrupts

  - Currently running code would execute without preemption

  - Generally too inefficient on multiprocessor systems

    ▸ Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions

    ▸ **Atomic** = non-interruptible

  - Either test memory word and set value

  - Or swap contents of two memory words

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;


    if (*value == expected)
        *value = new_value;

    return temp;

}
```

1. Executed atomically

2. Returns the original value of passed parameter "value"

3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers

- OS designers build software tools to solve critical section problem

- Simplest is mutex lock

- Protect a critical section by first `acquire()` a lock then `release()` the lock

    - Boolean variable indicating if lock is available or not

- Calls to `acquire()` and `release()` must be atomic

    - Usually implemented via hardware atomic instructions

- But this solution requires **busy waiting**

    - This lock therefore called a **spinlock**

acquire() and release()

- ```
  acquire() {
      while (!available)
          ; /* busy wait */
      available = false;
  }
  ```

- ```
  release() {
      available = true;
  }
  ```

- ```
  do {
      acquire lock
          critical section
      release lock
          remainder section
  } while (true);
  ```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait() operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **signal() operation**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain

- **Binary semaphore** – integer value can range only between 0 and 1

  - Same as a **mutex lock**

- Can solve various synchronization problems

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

  ```
  P1:
      S1;
      signal(synch);
  P2:
      wait(synch);
      S2;
  ```

- Can implement a counting semaphore **S** as a binary semaphore

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
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes

  - Bounded-Buffer Problem

  - Readers and Writers Problem

  - Dining-Philosophers Problem

Bounded-Buffer Problem

□ **n** buffers, each can hold one item

□ Semaphore `mutex` initialized to the value 1

□ Semaphore `full` initialized to the value 0

□ Semaphore `empty` initialized to the value n

Bounded Buffer Problem (Cont.)

☐ The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);

        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);

        ...
    /* remove an item from buffer to next_consumed */

        ...
    signal(mutex);
    signal(empty);

        ...
    /* consume the item in next consumed */

        ...
} while (true);
```

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
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Readers-Writers Problem (Cont.)

☐ The structure of a writer process

```
do {
      wait(rw_mutex);

        ...
      /* writing is performed */

        ...

    signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

☐ The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)

        wait(rw_mutex);

    signal(mutex);

        ...
        /* reading is performed */

        ...

    wait(mutex);
        read count--;
        if (read_count == 0)

    signal(rw_mutex);

    signal(mutex);

} while (true);
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- 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);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling

    - Allow at most 4 philosophers to be sitting simultaneously at the table.

    - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.

    - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

Condition Variables

- **condition x, y;**

- Two operations are allowed on a condition variable:

  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**

  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**

    - If no **x.wait()** on the variable, then it has no effect on the variable

Monitor with Condition Variables

Monitor Solution to Dining Philosophers

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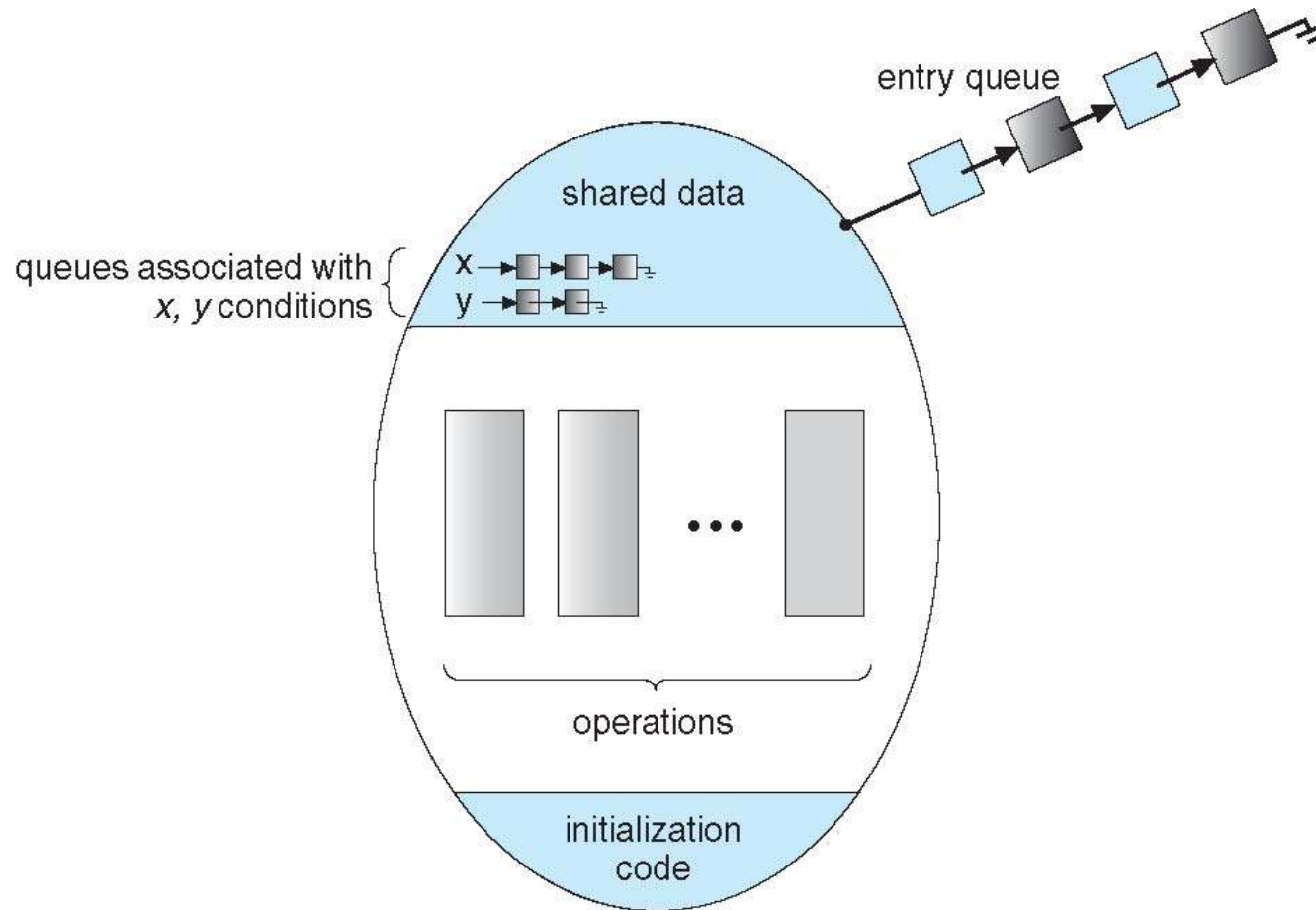monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    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);
    }
```

```
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);
```

☐ No deadlock, but starvation is possible

CPU Scheduling - Basic Concepts

- Maximum CPU utilization obtained with multiprogramming

- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait

- **CPU burst** followed by **I/O burst**

- CPU burst distribution is of main concern

⋮

load store **add store** **read** from file	CPU burst
wait for I/O	I/O burst
store increment **index** **write** to file	CPU burst
wait for I/O	I/O burst
load store **add store** **read** from file	CPU burst
wait for I/O	I/O burst

⋮
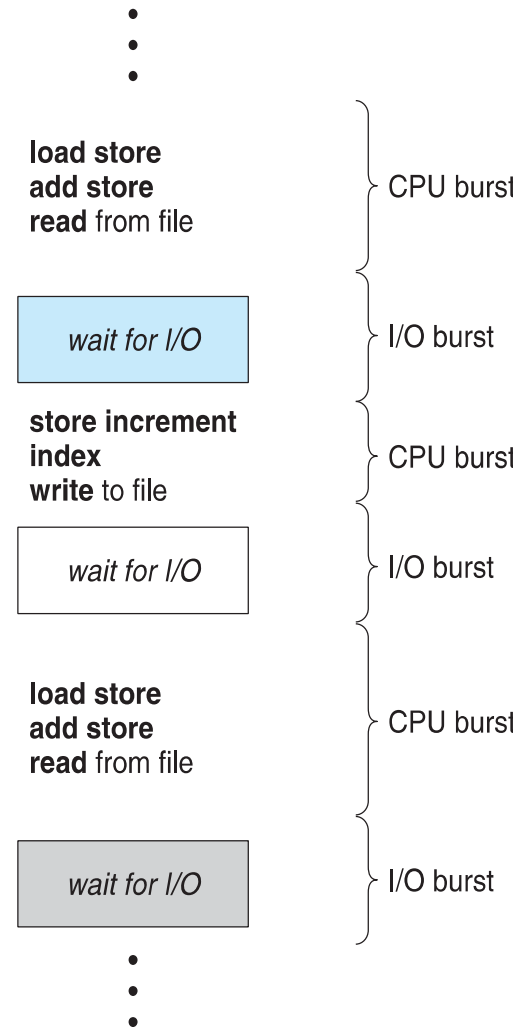
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

  - switching context

  - switching to user mode

  - jumping to the proper location in the user program to restart that program

- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible

- **Throughput** – # of processes that complete their execution per time unit

- **Turnaround time** – amount of time to execute a particular process

- **Waiting time** – amount of time a process has been waiting in the ready queue

- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

P₁	P₂	P₃

0   24   27   30

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst

  - Use these lengths to schedule the process with the shortest time

- SJF is optimal – gives minimum average waiting time for a given set of processes

  - The difficulty is knowing the length of the next CPU request

  - Could ask the user

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one

  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
  2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Commonly, α set to ½

- Preemptive version called **shortest-remaining-time-first**

Priority Scheduling

- A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive

- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

- Problem ≡ **Starvation** – low priority processes may never execute

- Solution ≡ **Aging** – as time progresses increase the priority of the process

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.

- Timer interrupts every quantum to schedule next process

- Performance

  - q large $\Rightarrow$ FIFO

  - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

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 FCFS
        - 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 FCFS and receives 16 additional milliseconds
        - If it still does not complete, it is preempted and moved to queue Q_2

quantum = 8

quantum = 16

FCFS