OPERATING SYSTEMS
BCS303
SYLLABUS HIGHLIGHTS

Module 2

- Process Management
- Multi-threaded Programming
- Process Synchronization

Module-2: Process Management

Processes:

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To describe communication in client-server systems

Process Concept

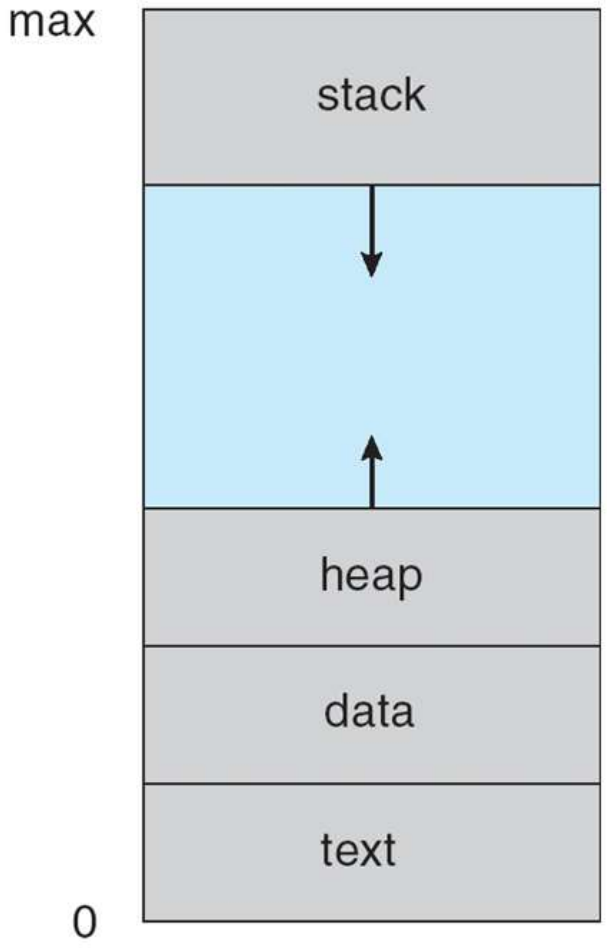
- A process is a set of sequential steps that are required to do a particular task.
- A process is an instance of a program in execution.
- For e.g.: in Windows, if we edit two text files, simultaneously, in notepad, then it means we are implementing two different instances of the same program.
- For an operating system, these two instances are separate processes of the same application

Process Concept

- A process needs certain resources such as:
  - CPU Time
  - Memory Files
  - I/O Devices

  to accomplish its task.
- These resources are allocated to the process either when it is created or while it is executing.

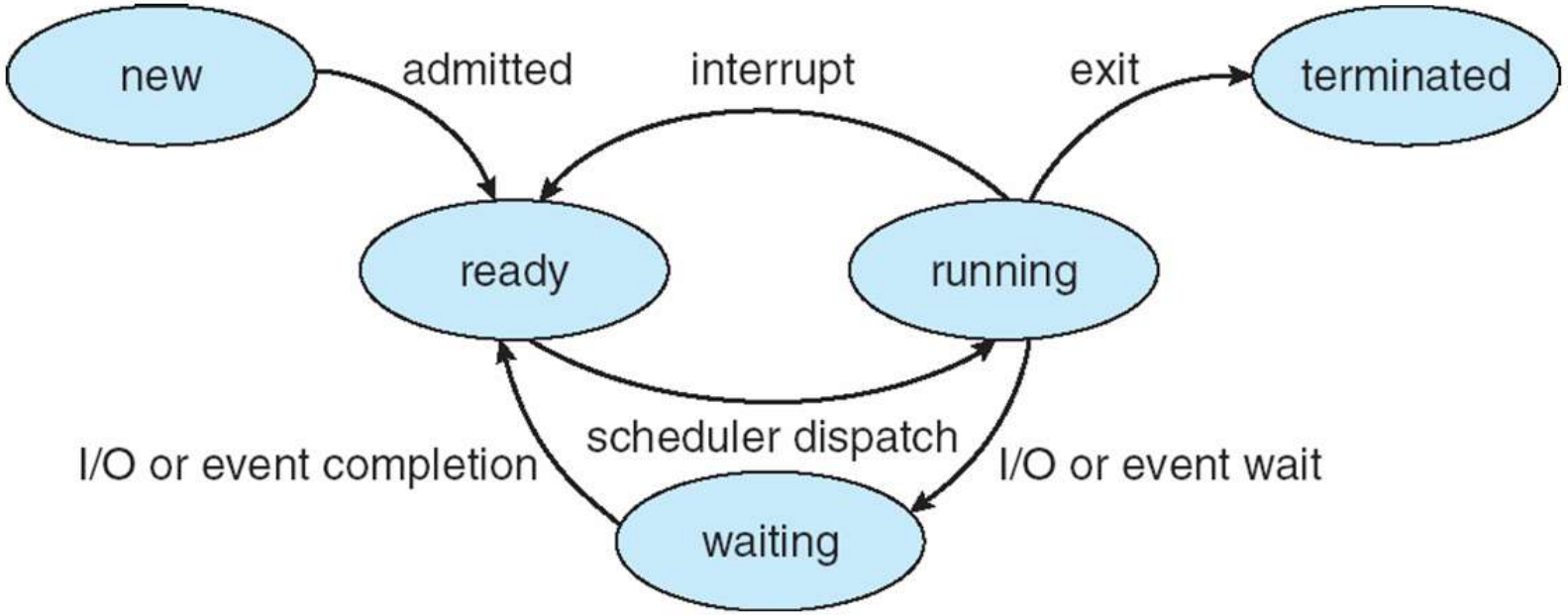
Process in Memory



Process State

- A process goes through a series of process states for performing its task.
- As a process executes, it changes state.
- Various events can cause a process to change state.

Diagram of Process State



Process States

- **New:**
  - A process that has just been created.
- **Ready:**
  - The process is ready to be executed.
- **Running:**
  - The process whose instructions are being executed is called running process.

Process States

- **Waiting:**
  - The process is waiting for some event to occur such as completion of I/O operation.
- **Terminated:**
  - The process has finished its execution.
- **Note:** Only one process can be *running* on any processor at any instant. However, there can be many processes in *ready* and *waiting* states.

Process States

- **Waiting:**
  - The process is waiting for some event to occur such as completion of I/O operation.
- **Terminated:**
  - The process has finished its execution.
- **Note:** Only one process can be *running* on any processor at any instant. However, there can be many processes in *ready* and *waiting* states.

Process Control Block (PCB)

- Process Control Block (PCB) is a data structure used by operating system to store all the information about a process.
- It is also known as Process Descriptor.
- When a process is created, the operating system creates a corresponding PCB.

Process Control Block (PCB)

- Information in a PCB is updated during the transition of process states.
- When a process terminates, its PCB is released.
- Each process has a single PCB.

Process Control Block (PCB)

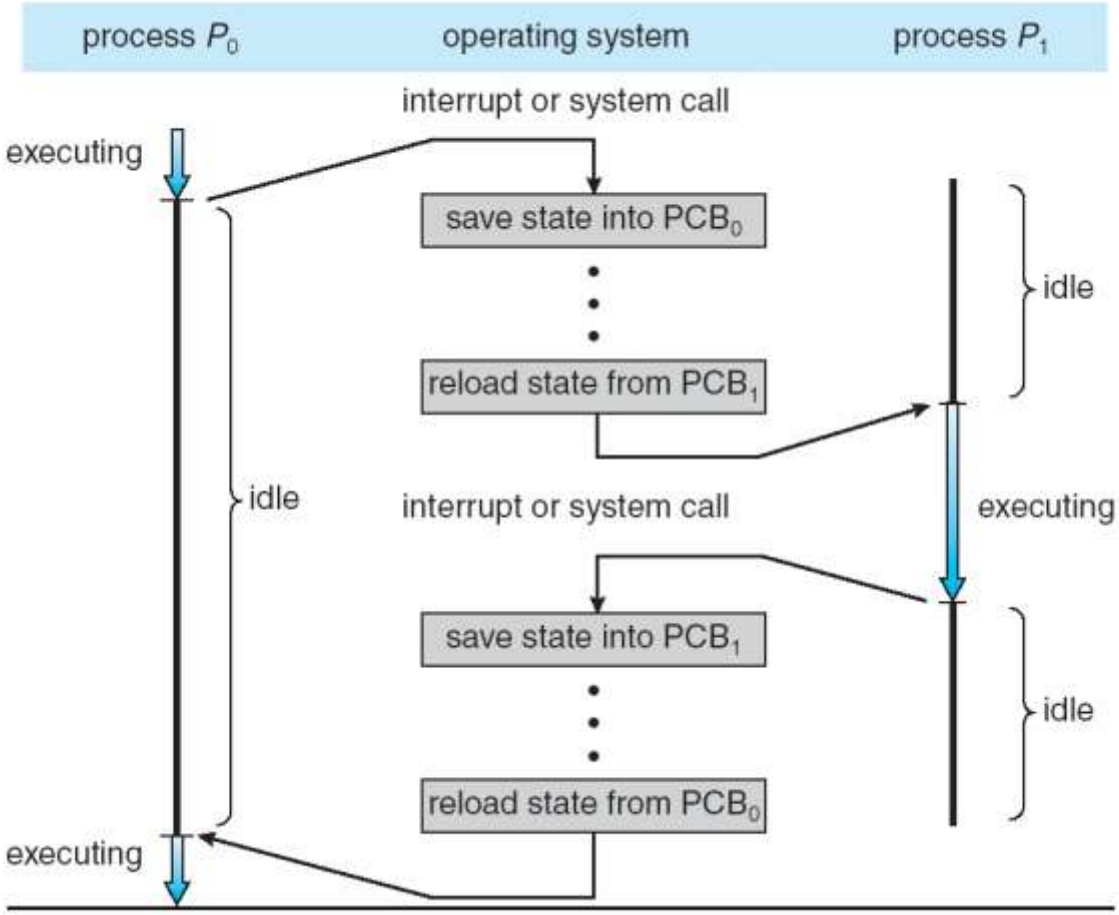
- **Process Number:** Each process is allocated a unique number for the purpose of identification.
- **Process State:** It specifies the current state of a process.
- **Program Counter:** It indicates the address of next instruction to be executed.

Process Control Block (PCB)

- **Registers:** These hold the data or result of calculations. The content of these registers is saved so that a process can be resumed correctly later on.
- **Memory Limits:** It stores the amount of memory units allocated to a process.
- **List of Open Files:** It stores the list of open files and there access rights.

Process Control Block (PCB)

CPU Switch From Process to Process



Process Scheduling

- In multiprogramming, several processes are kept in main memory so that when one process is busy in I/O operation, other processes are available to CPU.
- In this way, CPU is busy in executing processes at all times.
- This method of selecting a process to be allocated to CPU is called Process Scheduling.

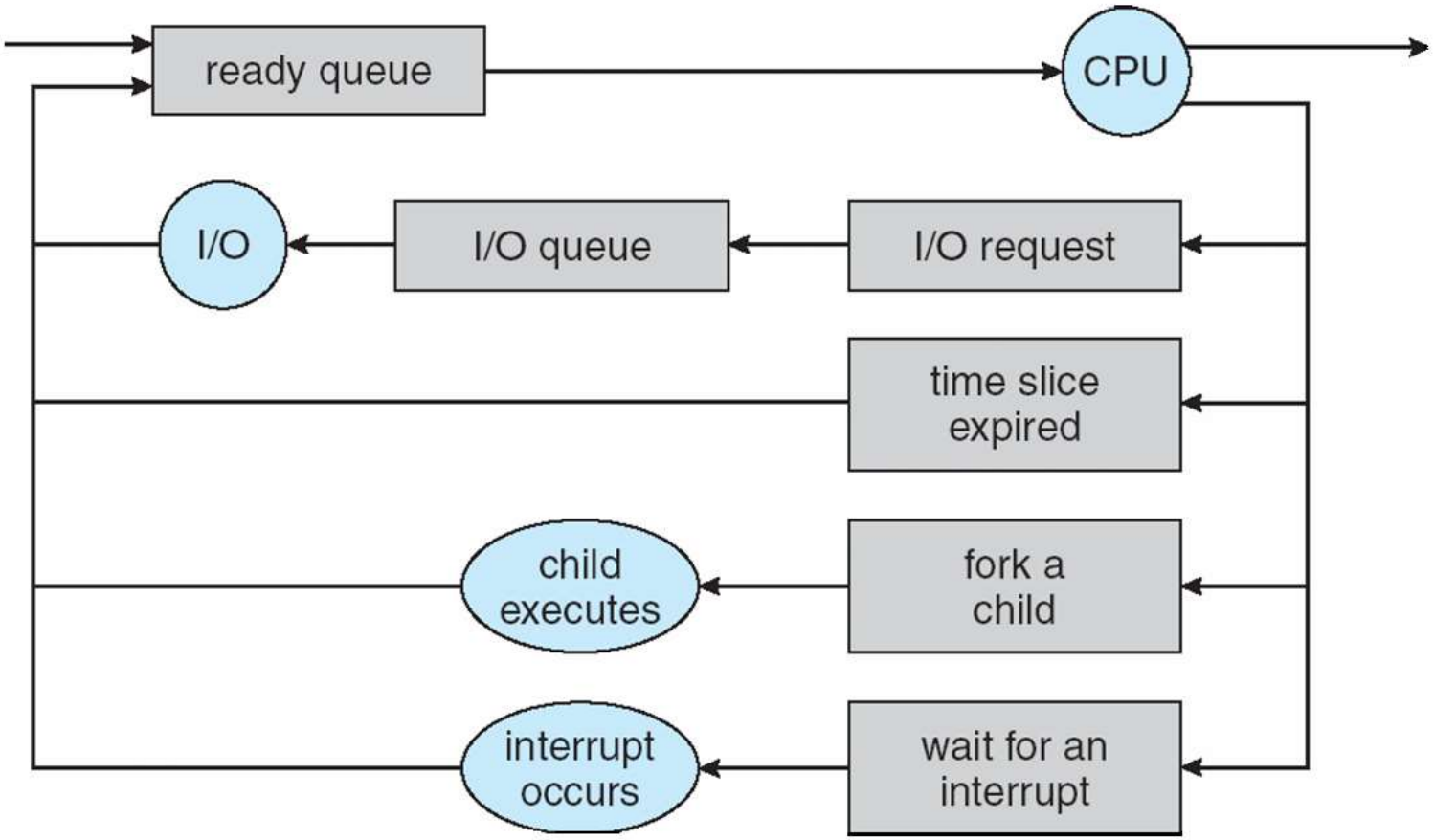
Process Scheduling

- Process scheduling consists of the following sub-functions:
  - **Scheduling:** Selecting the process to be executed next on CPU is called scheduling.
    - In this function a process is taken out from a pool of ready processes and is assigned to CPU.
    - This task is done by a component of operating system called **Scheduler**.

Process Scheduling

- **Dispatching:** Setting up the execution of the selected process on the CPU is called dispatching.
  - It is done by a component of operating system called **Dispatcher**.
  - Thus, a dispatcher is a program responsible for assigning the CPU to the process, that has been selected by the Scheduler.
- **Context Save:** Saving the status of a running process when its execution is to be suspended is known as context save.

Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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

Scheduling Queues

- In multiprogramming, several processes are there in ready or waiting state.
- These processes form a queue.
- The various queues maintained by operating system are:
  - Job Queue
  - Ready Queue
  - Device Queue

Scheduling Queues

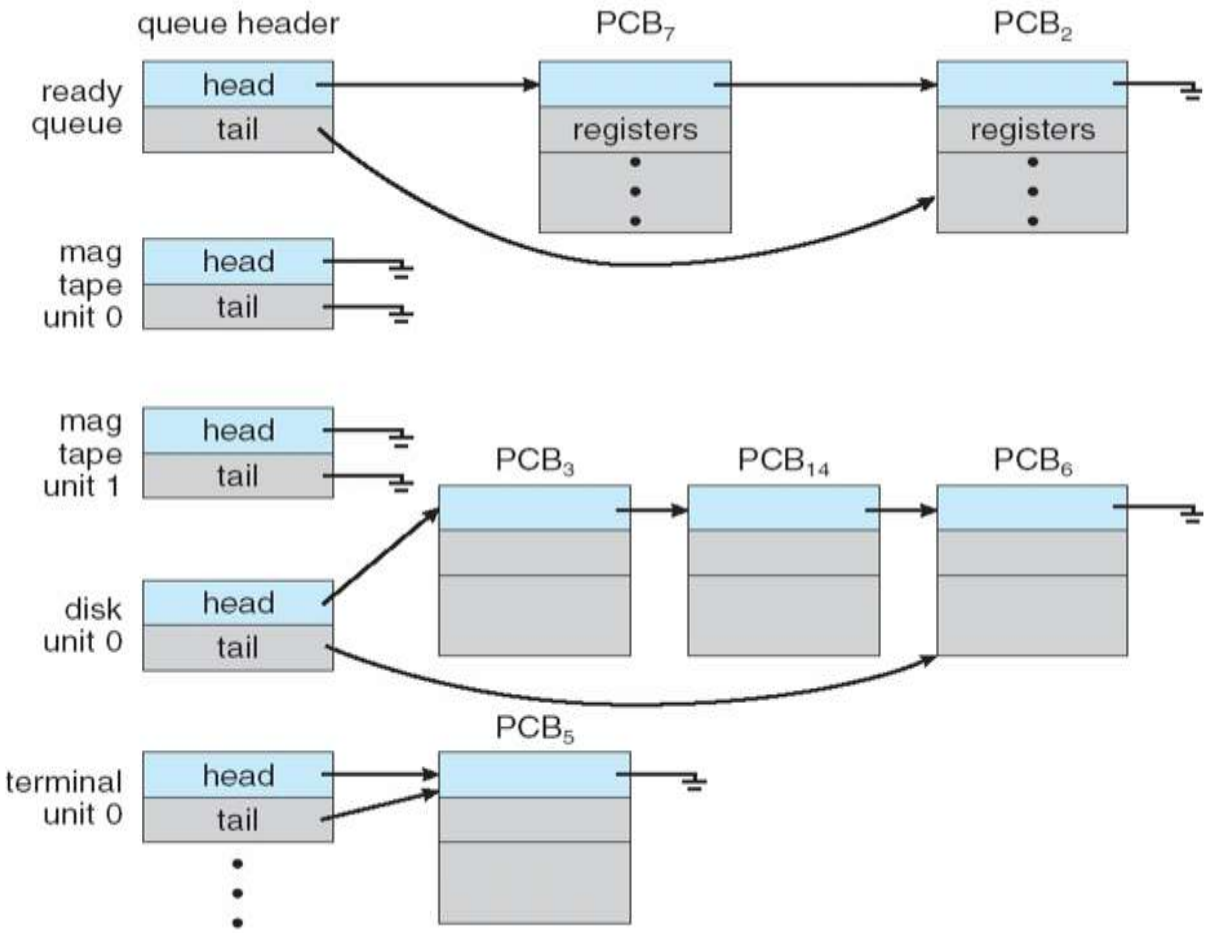
- **Job Queue:**
  - As the process enter the system, it is put into a job queue. This queue consists of all processes in the system.
- **Ready Queue:**
  - It is a doubly linked list of processes that are residing in the main memory and are ready to run.

Scheduling Queues

- **Device Queue:**
  - It contains all those processes that are waiting for a particular I/O device.
  - Each device has its own device queue.

- Diagram on the next slide shows the queues.

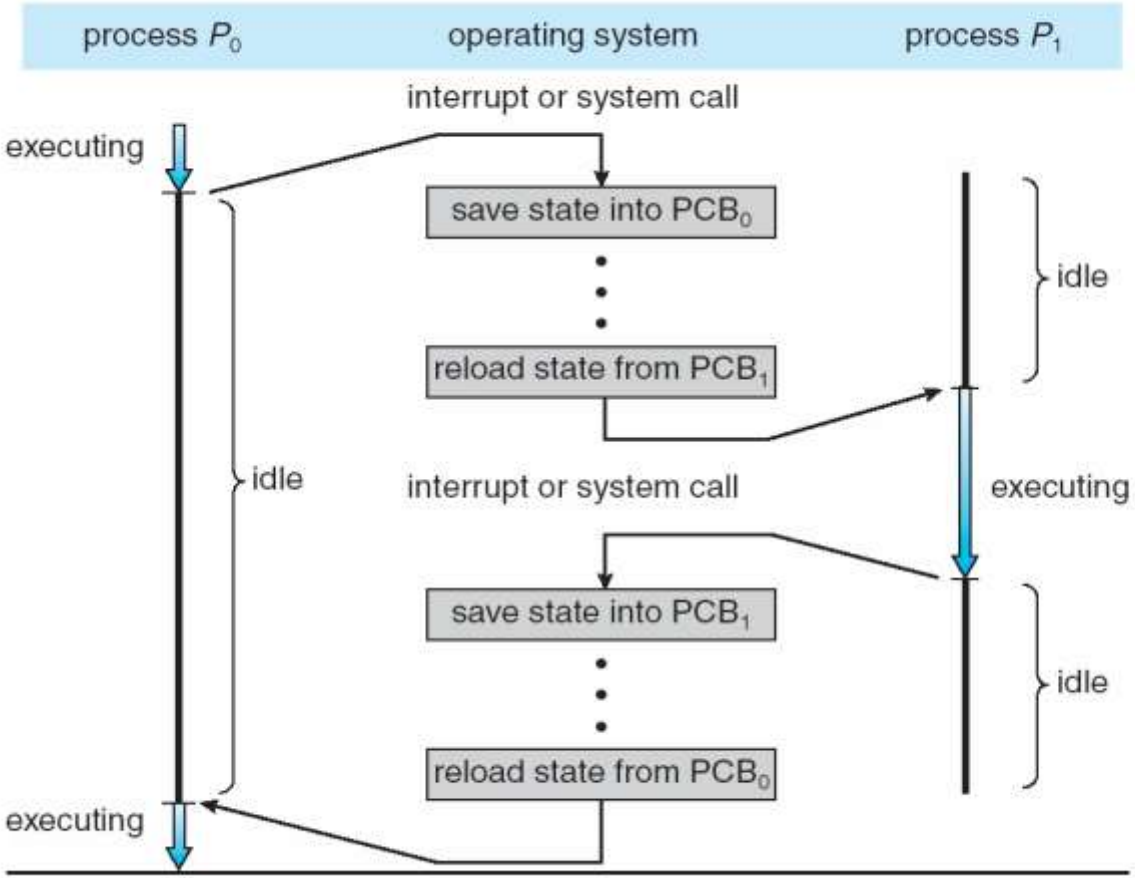
Scheduling Queues



Context Switch

- Switching the CPU from one process to another process requires saving the state of old process and loading the saved state of new process.
- This task is known as **Context Switch**.
- When context switch occurs, operating system saves the context of old process in its PCB and loads the saved context of the new process.

Context Switch



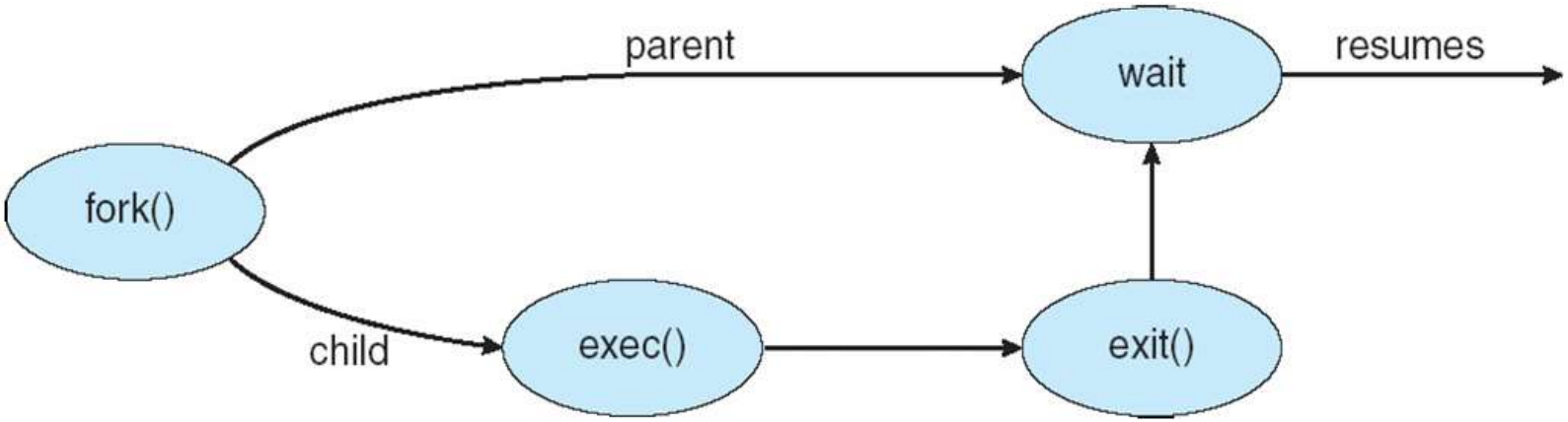
Process Creation

- **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
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

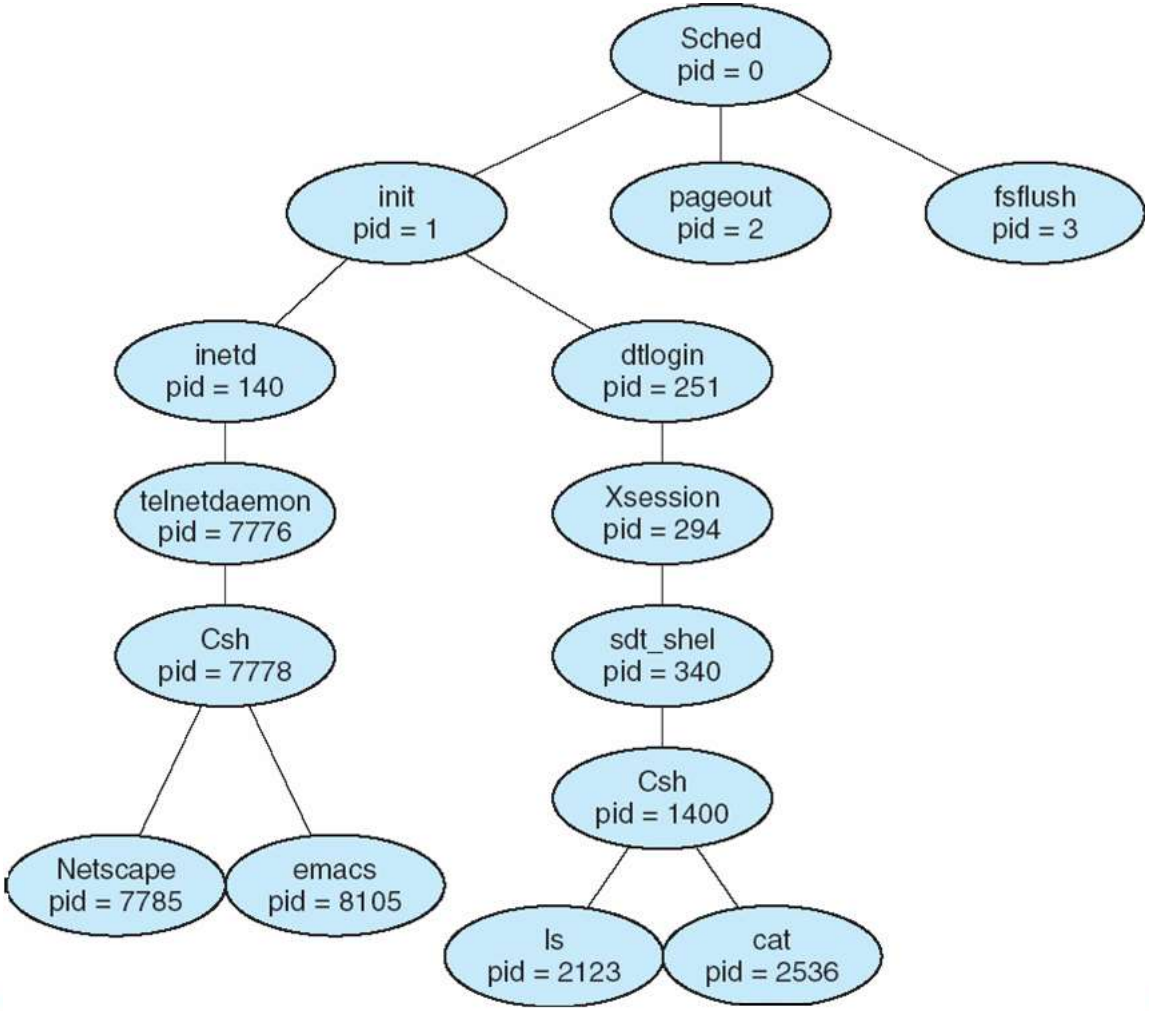
Process Creation



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-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");
                exit(0);
        }
}
```

A tree of processes on a typical Solaris



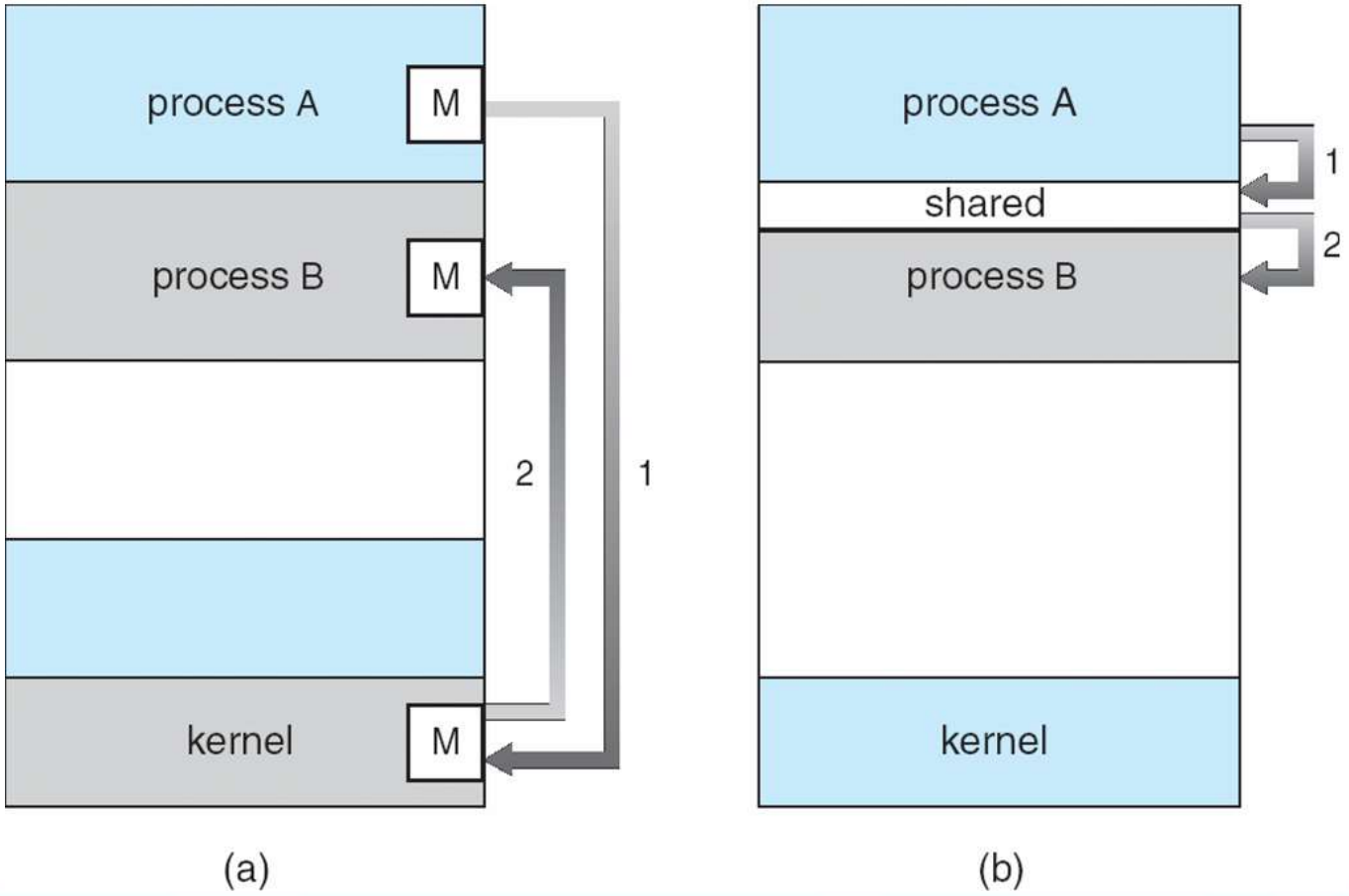
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

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
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
while (true) {
   /* Produce an item */
   while (((in = (in + 1) % BUFFER SIZE count)  == out)
      ;   /* do nothing -- no free buffers */
     buffer[in] = item;
     in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

```
while (true) {
     while (in == out)
          ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
return item;
}
```

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

send(*A, message*) – send a message to mailbox A

receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

`msg_send(), msg_receive(), msg_rpc()`

  - Mailboxes needed for commuication, created via

`port_allocate()`

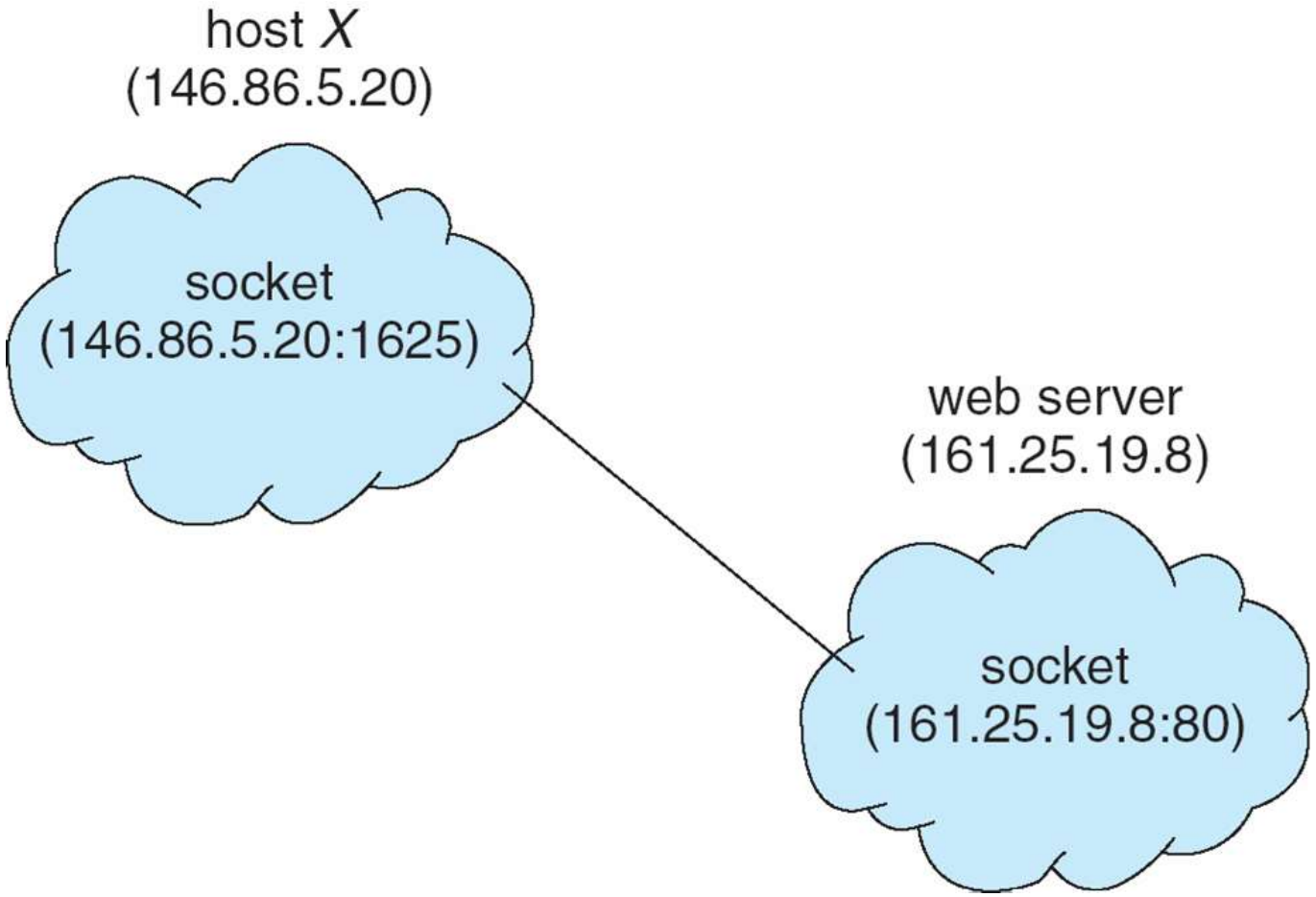
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

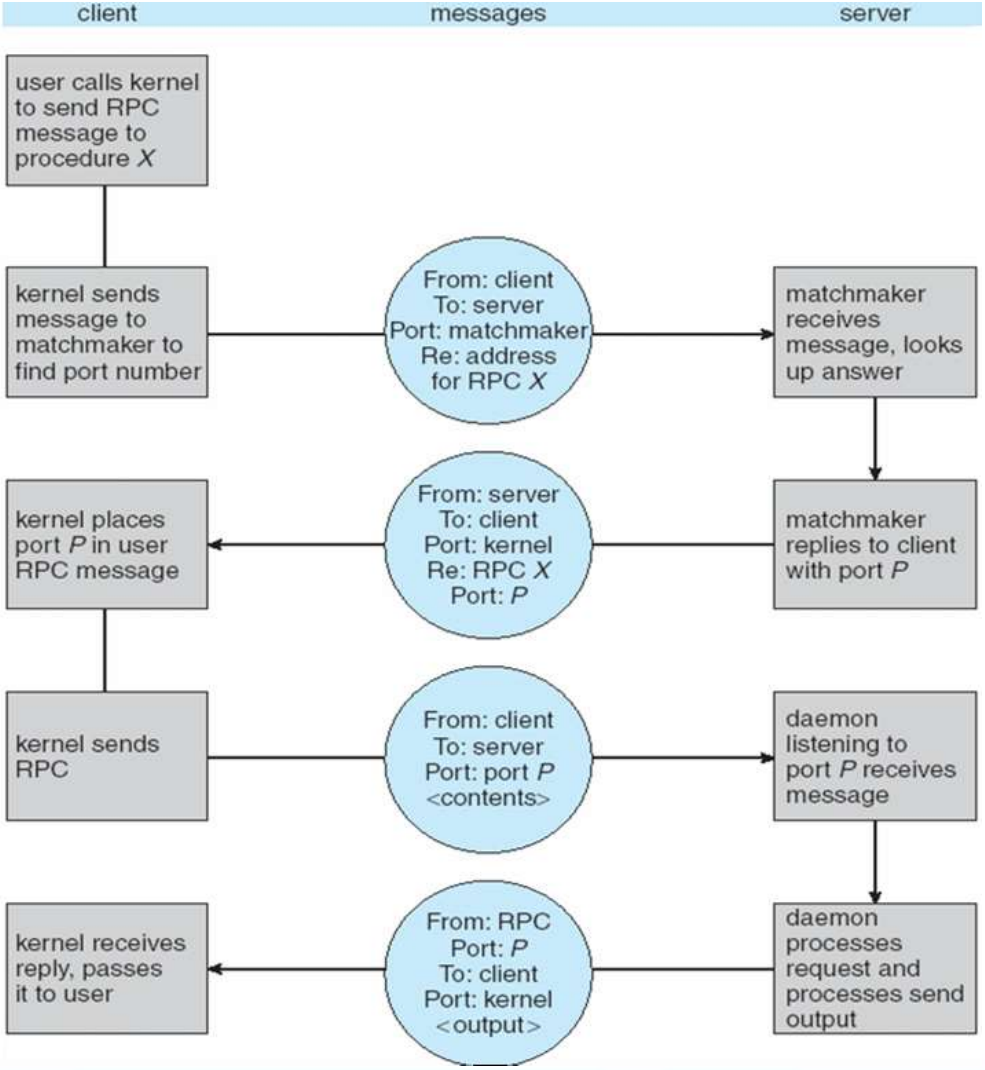
Socket Communication



Remote Procedure Calls

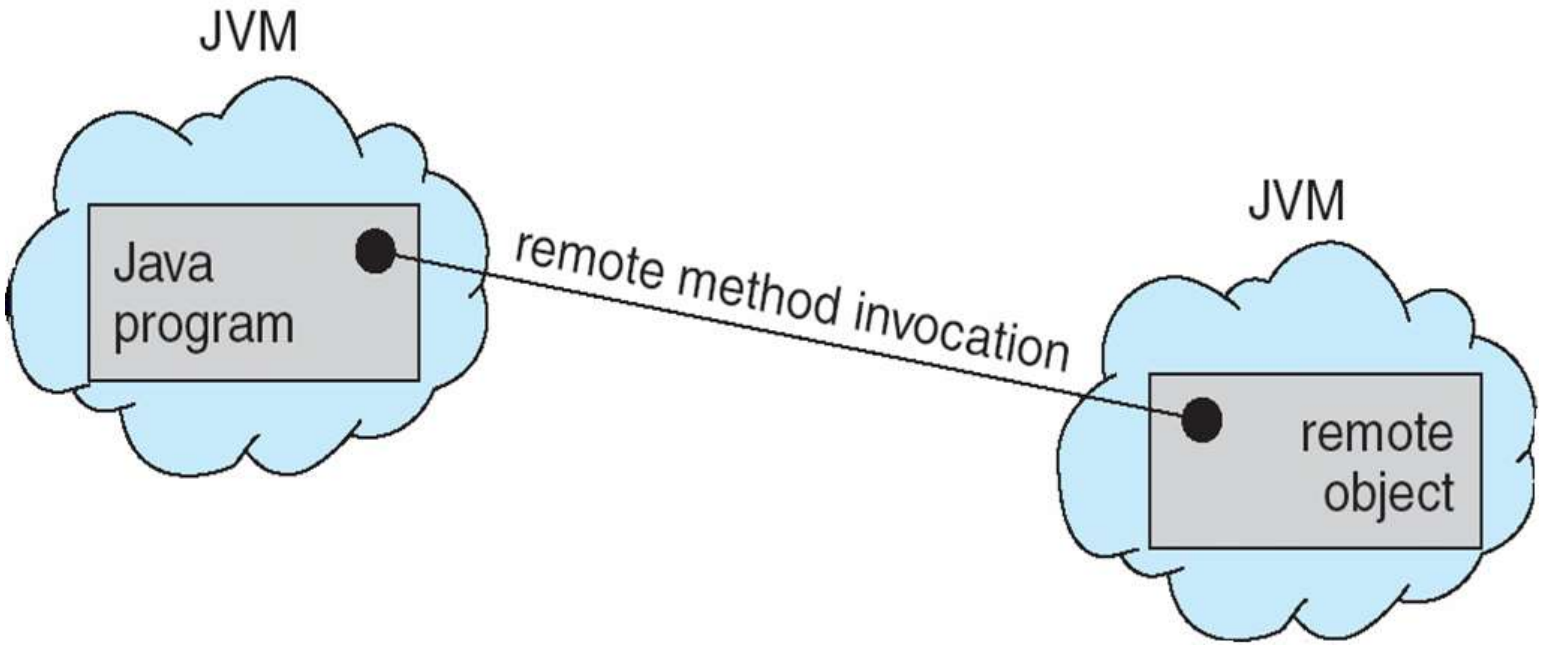
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server

Execution of RPC



Remote Method Invocation

Remote Method Invocation (RMI) is a Java mechanism similar to RPCs



Module 2

- Multi-Threaded Programming
- Process Synchronization

Chapter 3 Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Thread

A thread is a single sequential flow of execution of the tasks of a process.

A thread is a lightweight process and the smallest unit of CPU utilization. Thus, a thread is like a miniprocess.

Each thread has a thread id, program counter, register set and a stack.

A thread undergoes different states such as new, ready, running, waiting and terminated similar to that of a process.

However, a thread is not a program as it cannot run on its own. It runs within a program.

Multi-Threading

A process can have single thread of control or multiple threads of control.

If a process has single thread of control, it can perform only one task at a time.

Many modern operating systems have extended the process concept to allow a process to have multiple threads.

Thus, allowing the process to perform multiple tasks at the same time.

This concept is known as **Multi-Threading**.

Multi-Threading

For e.g.:

The tasks in a web browser are divided into multiple threads.

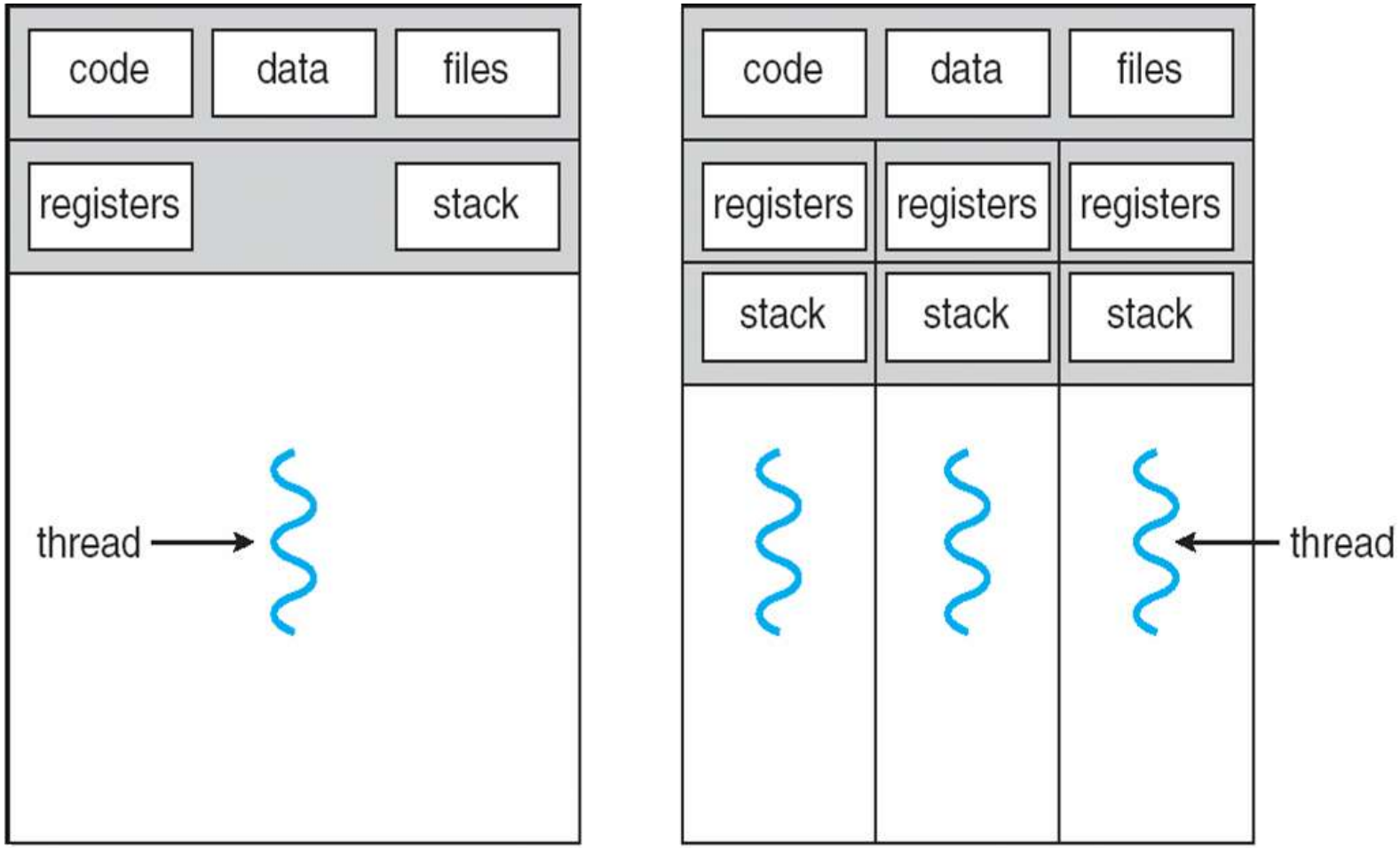
Downloading the images, downloading the text and displaying images and text.

While one thread is busy in downloading the images, another thread displays it.

The various operating systems the implement multithreading are Windows XP, Vista, 7, Server 2000 onwards, Linux etc.

In multithreading, a thread can share its code, data and resources with other threads of same process.

Single Thread & Multi-Thread

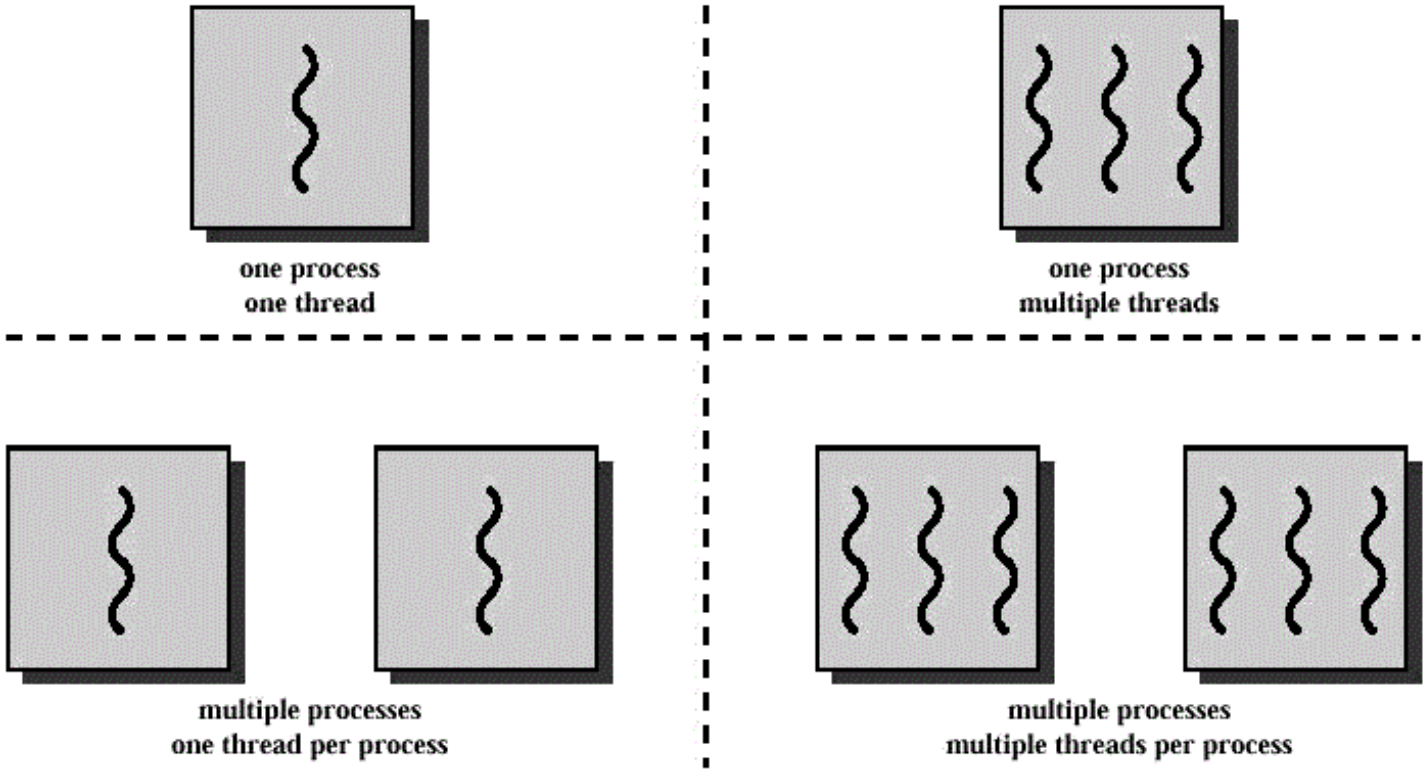


single-threaded process

multithreaded process

Threads & Processes

An idea of how threads & processes can be related to each other is depicted in the fig.:



Threads & Processes

There are several similarities and differences between a thread and a process:

Similarities:

Like process, each thread has its own program counter and stack.

Threads share CPU just as a process.

Threads also run sequentially, like a process.

Threads can create child threads.

Threads have the same states as process: new, ready, running, waiting and terminated.

Threads & Processes

Differences:

Each process has its own distinct address space in the main memory. On the other hand, all threads of a same process share same address space.

Threads require less system resources than a process.

Threads are not independent of each other, unlike processes.

Threads take less time for creation and termination than a process.

It takes less time to switch between two threads than to switch between two processes.

Types of Threads

Threads are of three types:

- Kernel Level Threads
- User Level Threads
- Hybrid Threads

Kernel Level Threads

Threads of processes defined by operating system itself are called **Kernel Level Threads**.

In these types of threads, kernel performs thread creation, scheduling and management.

Kernel threads are used for internal workings of operating system.

Kernel threads are slower to create and manage.

The various operating systems that support kernel level threads are: Windows 2000, XP, Solaris 2.

User Level Threads

The threads of user application process are called **User Level Threads**.

They are implemented in the user space of main memory.

User level library (functions to manipulate user threads) is used for thread creation, scheduling and management without any support from the kernel.

User level threads are fast to create and manage.

Hybrid Threads

In hybrid approach, both kernel level threads and user level threads are implemented.

For e.g.: Solaris 2.

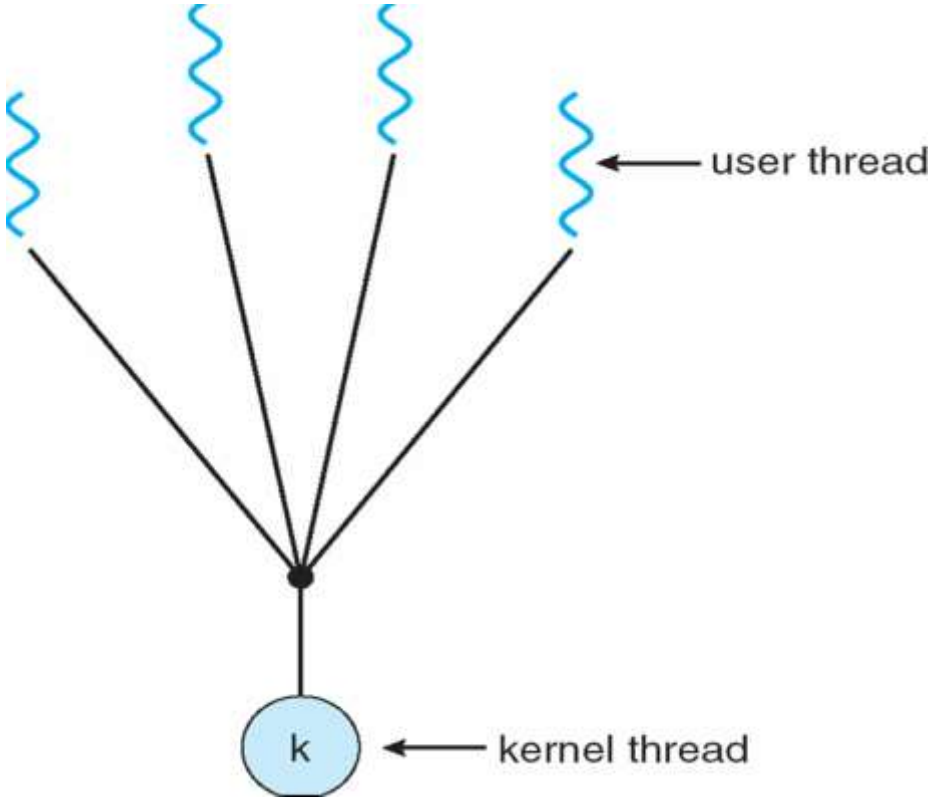
Multi-Threading Models

Depending on the support for user and kernel threads, there are three multithreading models:

- Many-to-One Model
- One-to-One Model
- Many-to-Many Model

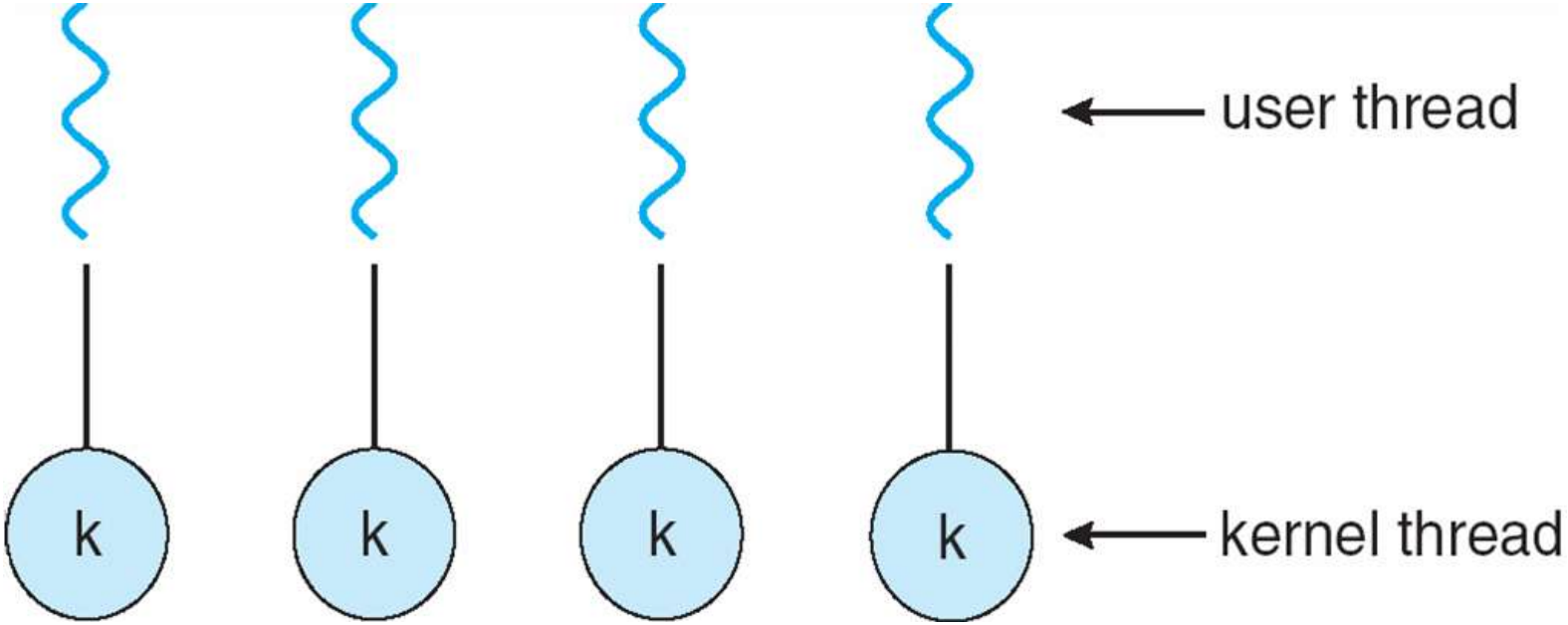
Many-to-One Model

- In this model, many user level threads are mapped to one kernel level thread.
- Threads are managed in user space.



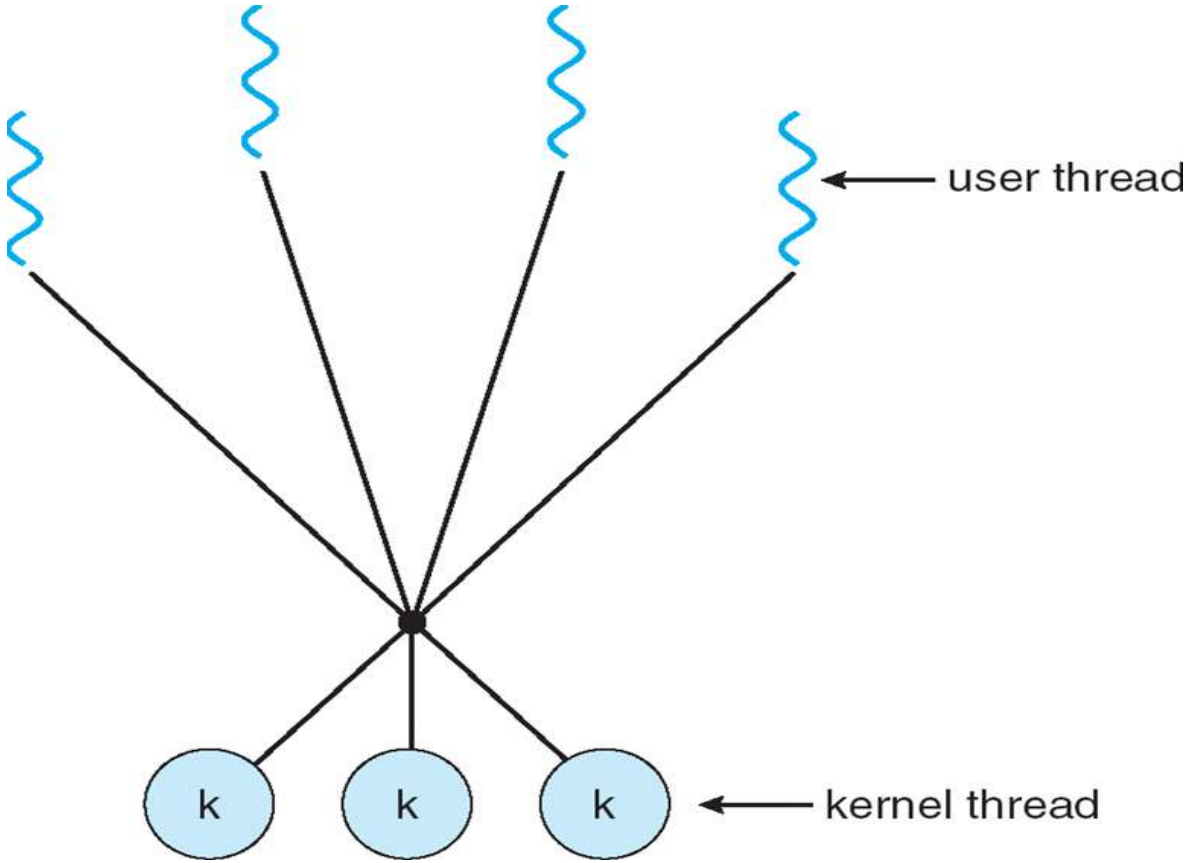
One-to-One Model

In this model, each user level thread is mapped to one kernel level thread.

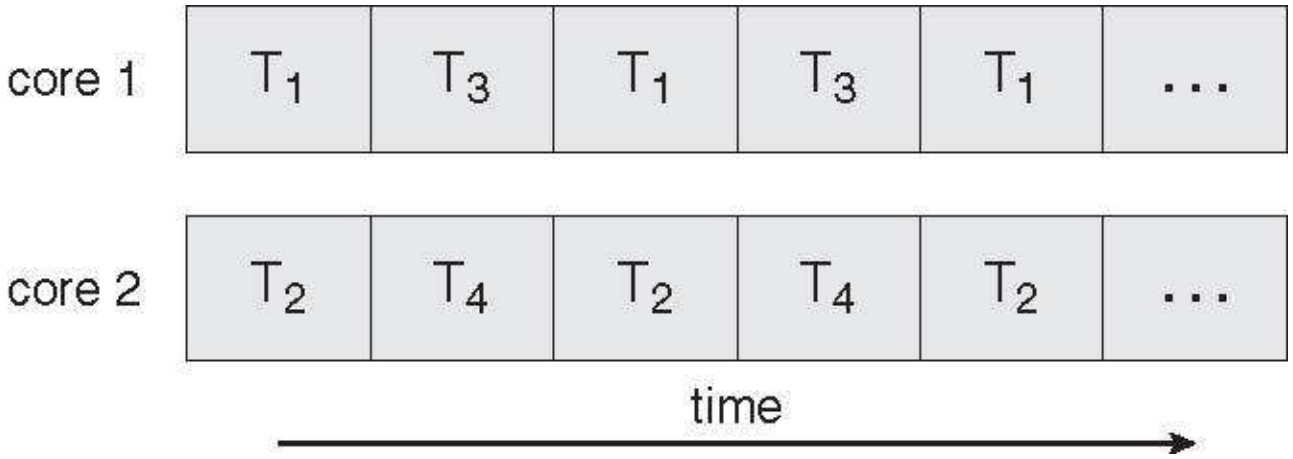


Many-to-Many Model

In this model, many user level threads are mapped to many kernel level threads.



Parallel Execution on a Multicore System



Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

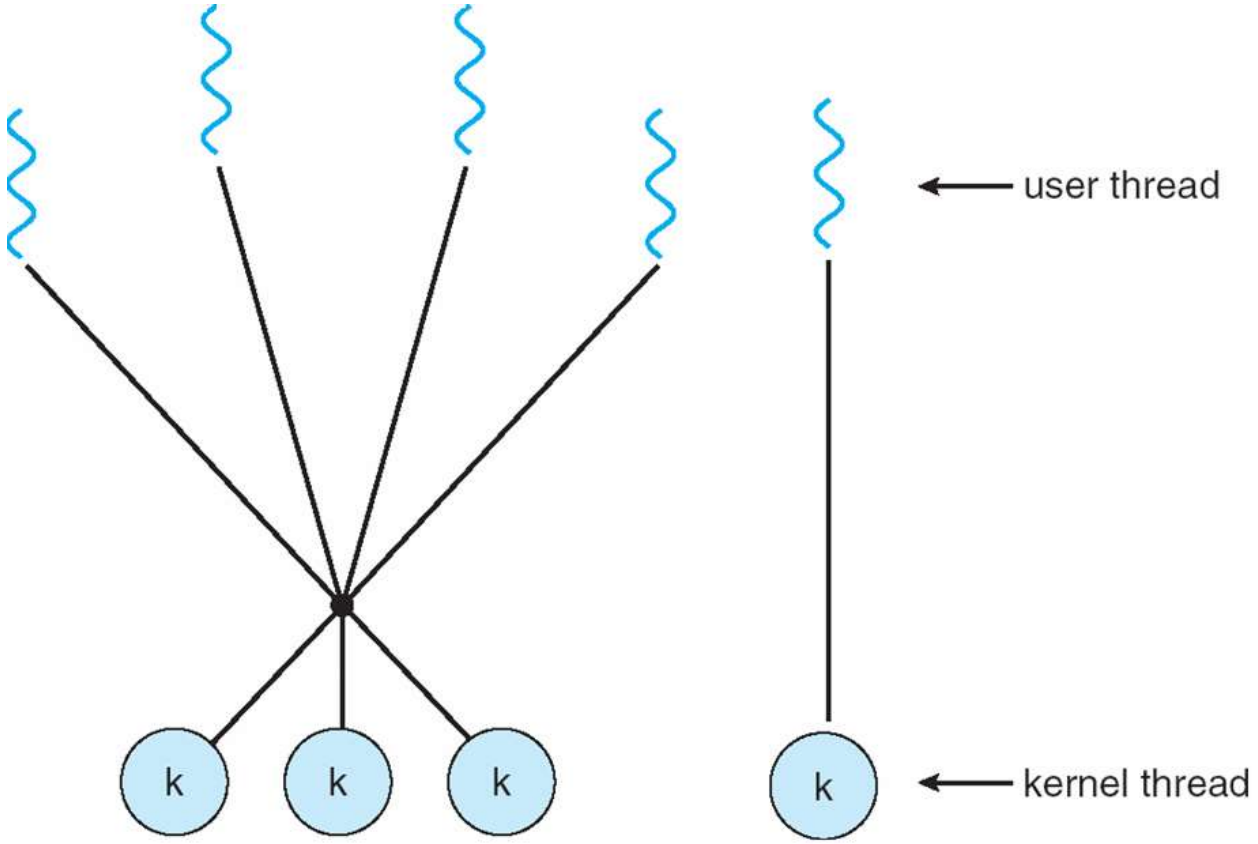
One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Two-level Model



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

End of Chapter 4

Chapter 5: Process Scheduling

Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling

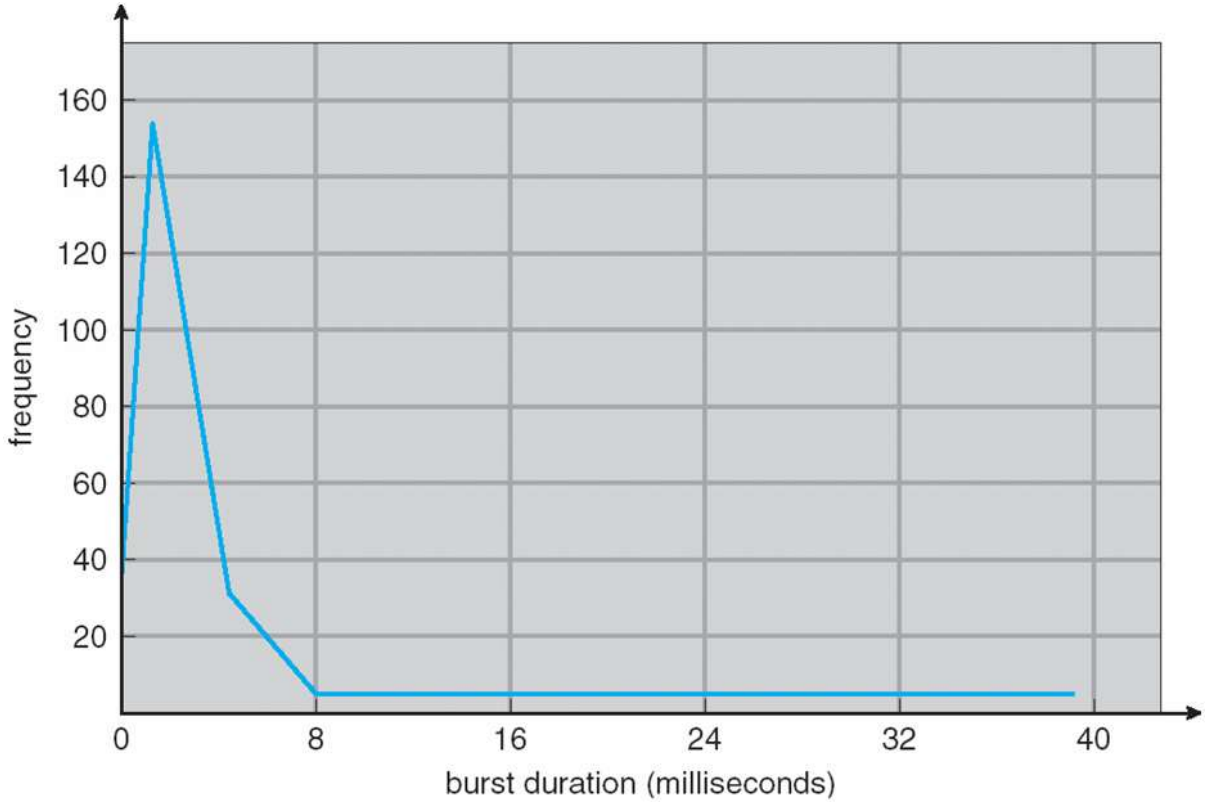
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

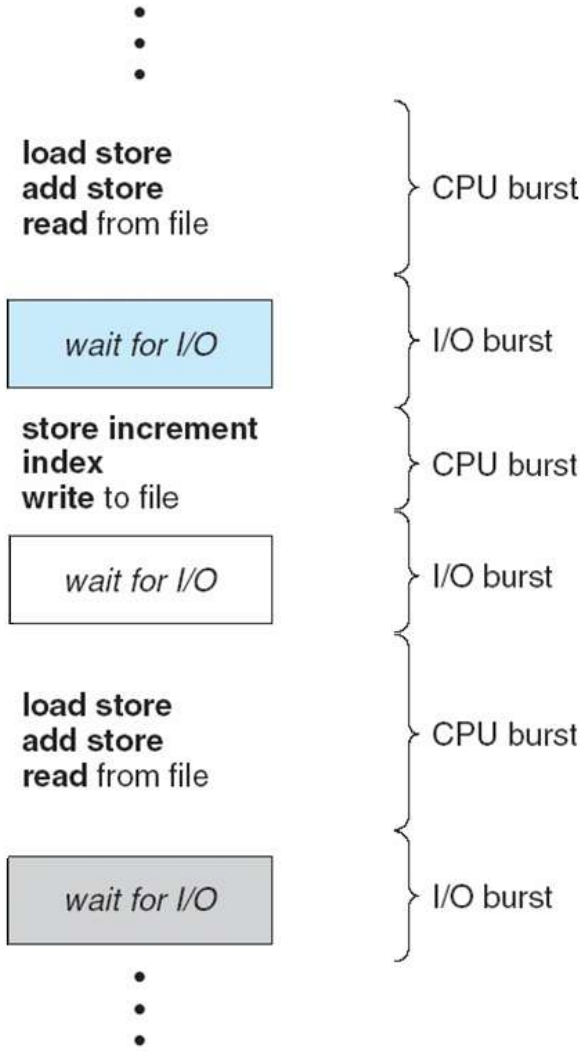
Basic Concepts

- 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** distribution

Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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

Scheduling Algorithm

- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

Scheduling Algorithm

- First-Come-First-Served (FCFS)
- Shortest Job First (SJF)
- Priority Scheduling
- Round-Robin Scheduling (RR)
- Multi-Level Queue Scheduling (MLQ)
- Multi-Level Feedback Queue Scheduling (MFQ)

Scheduling Algorithm

- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

Scheduling Algorithm

- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

Scheduling Algorithm

- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

Scheduling Algorithm

- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

Scheduling Algorithm

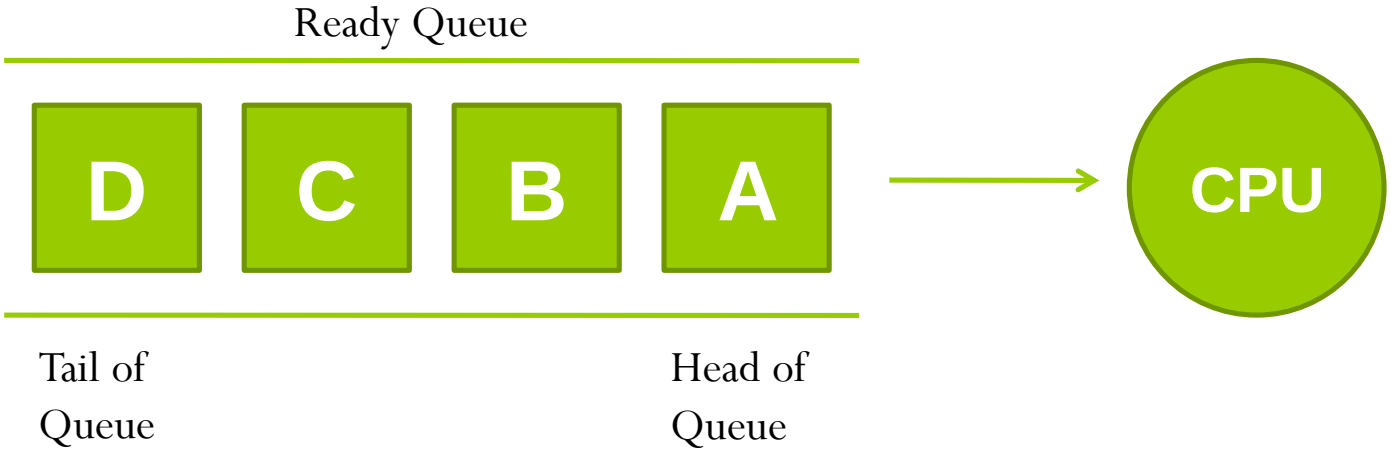
- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

Scheduling Algorithm

- CPU Scheduling algorithms deal with the problem of deciding which process in ready queue should be allocated to CPU.
- Following are the commonly used scheduling algorithms:

First-Come, First-Served (FCFS) Scheduling

- In this scheduling, the process that requests the CPU first, is allocated the CPU first.
- Thus, the name *First-Come-First-Served.*
- The implementation of FCFS is easily managed with a FIFO queue.



FCFS Scheduling (Cont.)

- When a process enters the ready queue, its PCB is linked to the tail of the queue.
- When CPU is free, it is allocated to the process which is at the head of the queue.
- FCFS scheduling algorithm is ***non-preemptive***.
- Once the CPU is allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by I/O request.

Example of FCFS Scheduling

- Consider the following set of processes that arrive at time 0 with the length of the CPU burst time in milliseconds:

Process	Burst Time (in milliseconds)
P_1	24
P_2	3
P_3	3

Example of FCFS Scheduling

- Consider the following set of processes that arrive at time 0 with the length of the CPU burst time in milliseconds: Suppose that the processes arrive in the order: P_1, P_2, P_3.

P_1	24
P_2	3
P_3	3

- The Gantt Chart for the schedule is:

- Waiting Time for P_1 = 0 milliseconds
- Waiting Time for P_2 = 24 milliseconds
- Waiting Time for P_3 = 27 milliseconds

Example of FCFS Scheduling

- Average Waiting Time = (Total Waiting Time) / No. of Processes
  = (0 + 24 + 27) / 3
  = 51 / 3
  = 17 milliseconds

Shortest-Job-First (SJF) Scheduling

- In SJF, the process with the least estimated execution time is selected from the ready queue for execution.
- It associates with each process, the length of its next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length of next CPU burst, FCFS scheduling is used.
- SJF algorithm can be preemptive or non-preemptive.

Non-Preemptive SJF

- In non-preemptive scheduling, CPU is assigned to the process with least CPU burst time.
- The process keeps the CPU until it terminates.
- **Advantage:**
  - It gives minimum average waiting time for a given set of processes.
- **Disadvantage:**
  - It requires knowledge of how long a process will run and this information is usually not available.

Preemptive SJF

- In preemptive SJF, the process with the smallest estimated run-time is executed first.
- Any time a new process enters into ready queue, the scheduler compares the expected run-time of this process with the currently running process.
- If the new process's time is less, then the currently running process is preempted and the CPU is allocated to the new process.

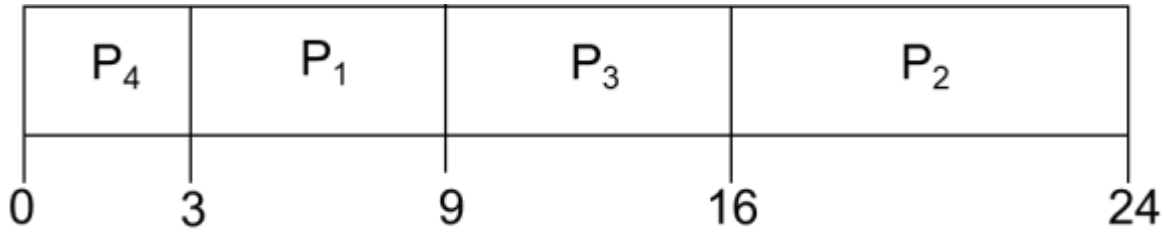
Example of Non-Preemptive SJF

Consider the following set of processes that arrive at time 0 with the length of the CPU burst time in milliseconds:

Process	Burst Time (in milliseconds)
P_1	6
P_2	8
P_3	7
P_4	3

Example of Non-Preemptive SJF

The Gantt Chart for the schedule is:



P_1	6
P_2	8
P_3	7
P_4	3

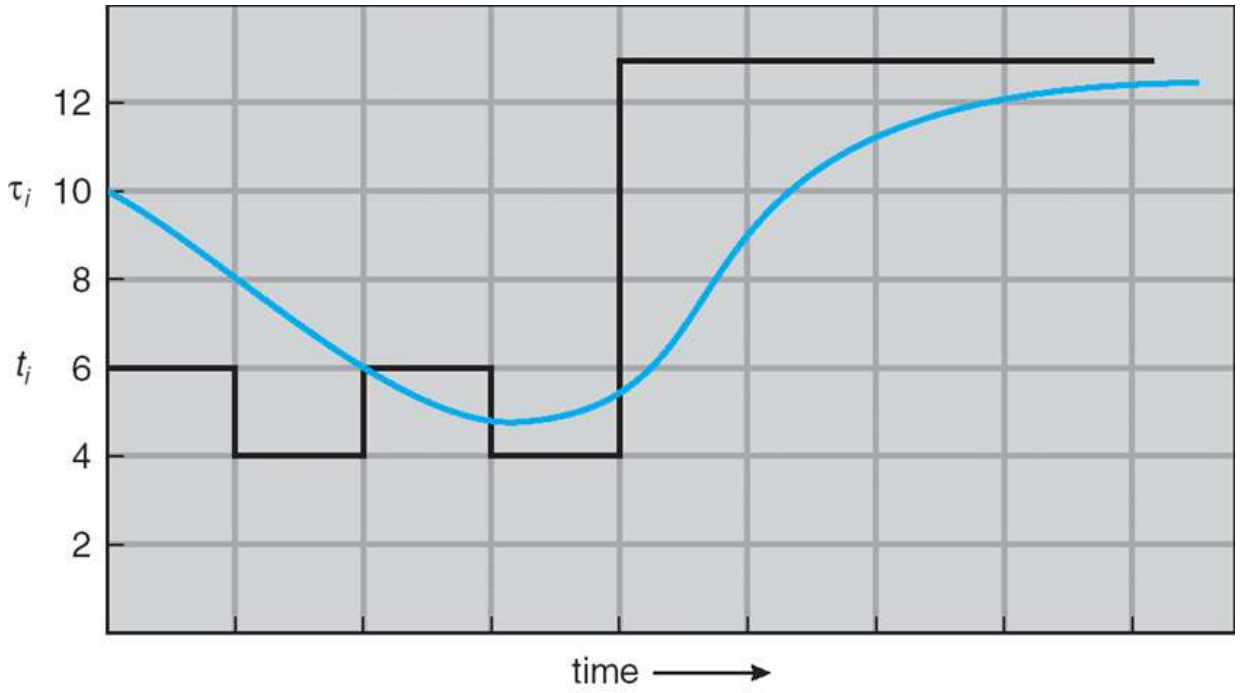
Waiting Time for P_4 = 0 milliseconds
Waiting Time for P_1 = 3 milliseconds
Waiting Time for P_3 = 9 milliseconds
Waiting Time for P_2 = 16 milliseconds

Determining Length of Next CPU Burst

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

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

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13 ...
"guess" (τ_i)	10	8	6	6	5	9	11	12 ...

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

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 a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

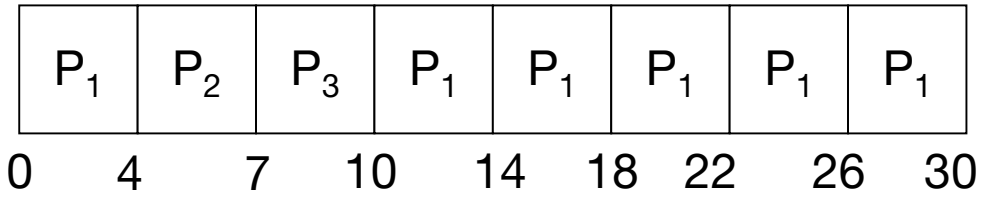
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), 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.
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

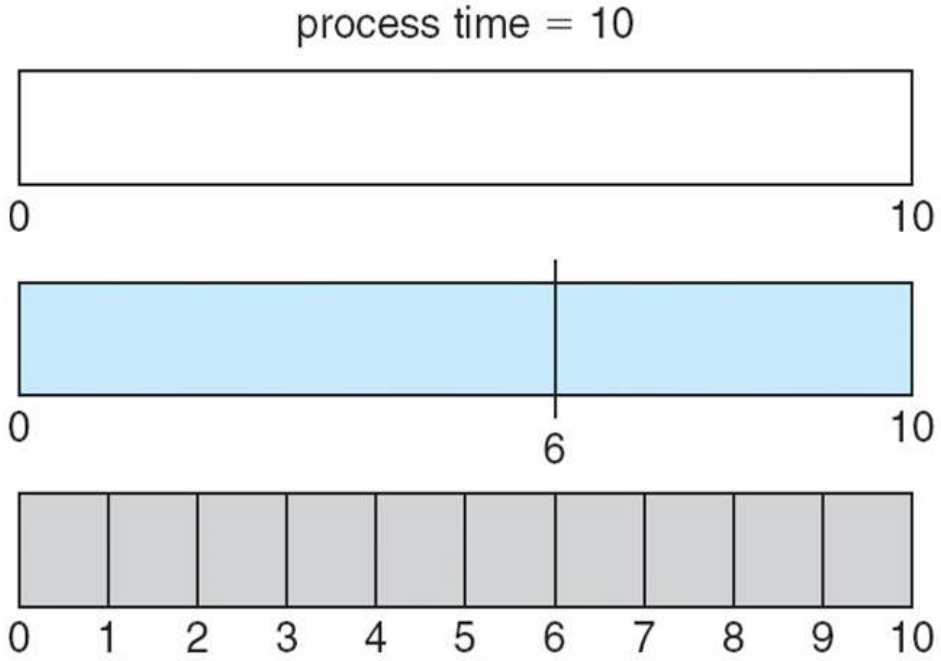
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



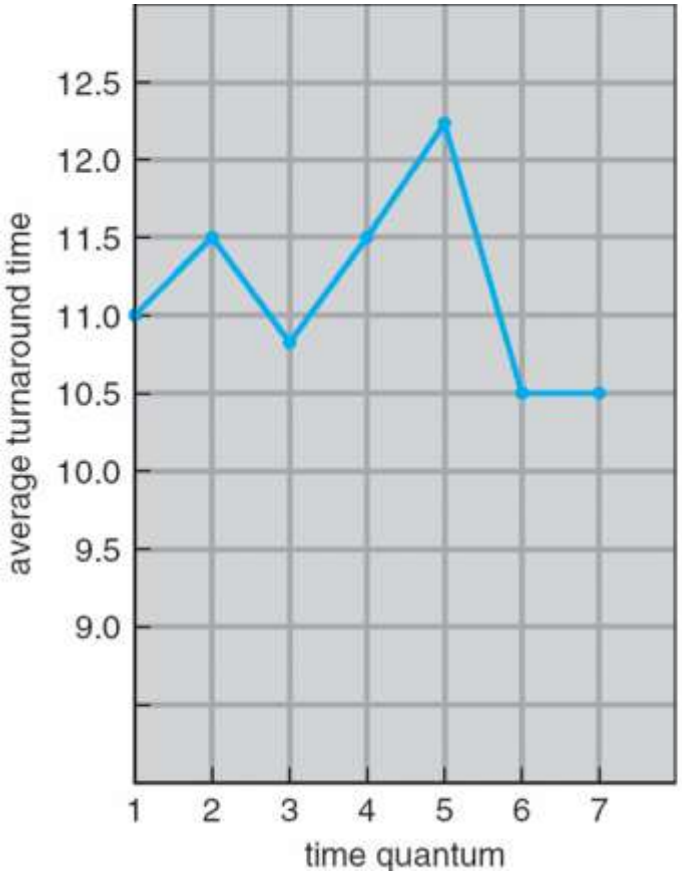
- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

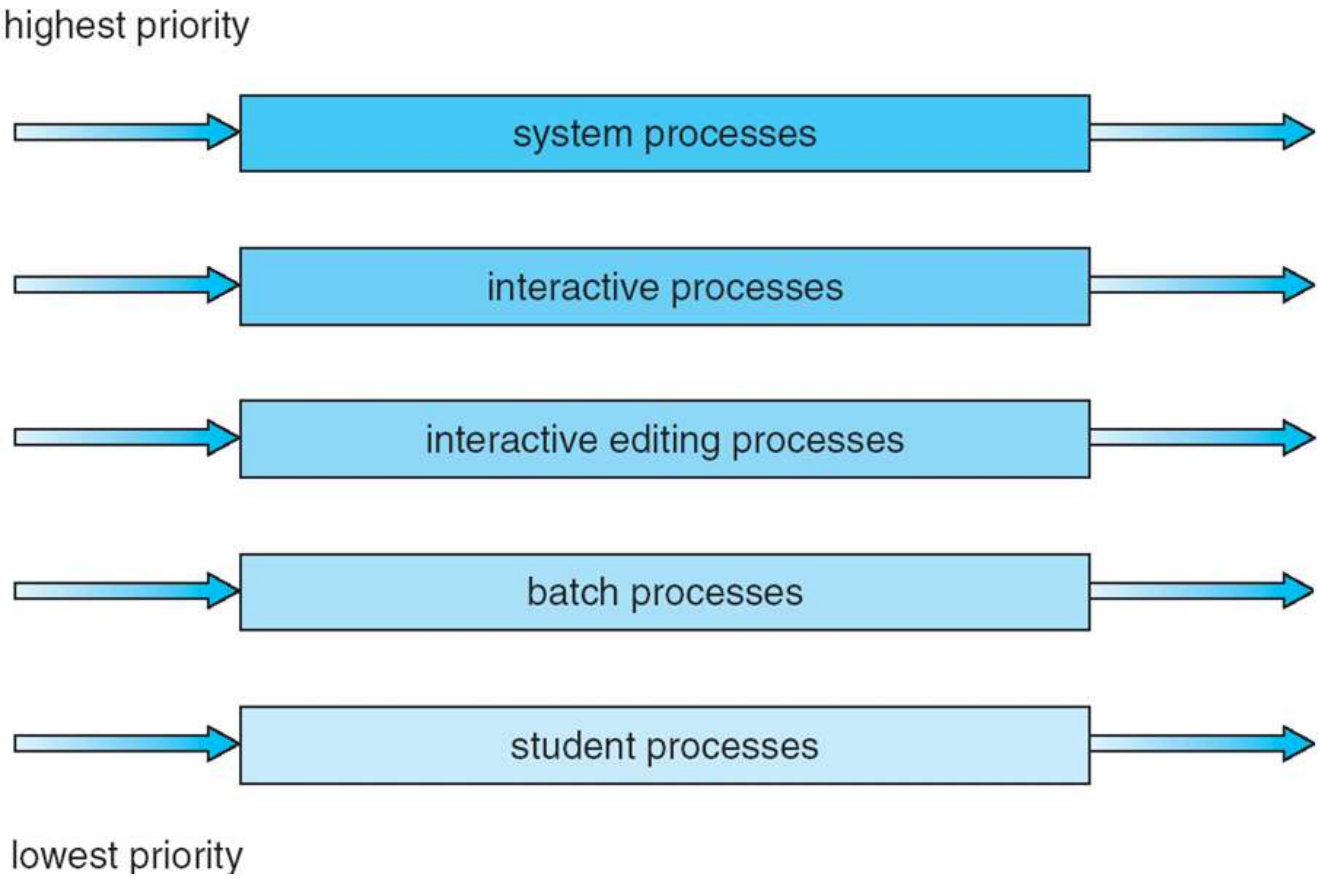


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- 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 Queue Scheduling



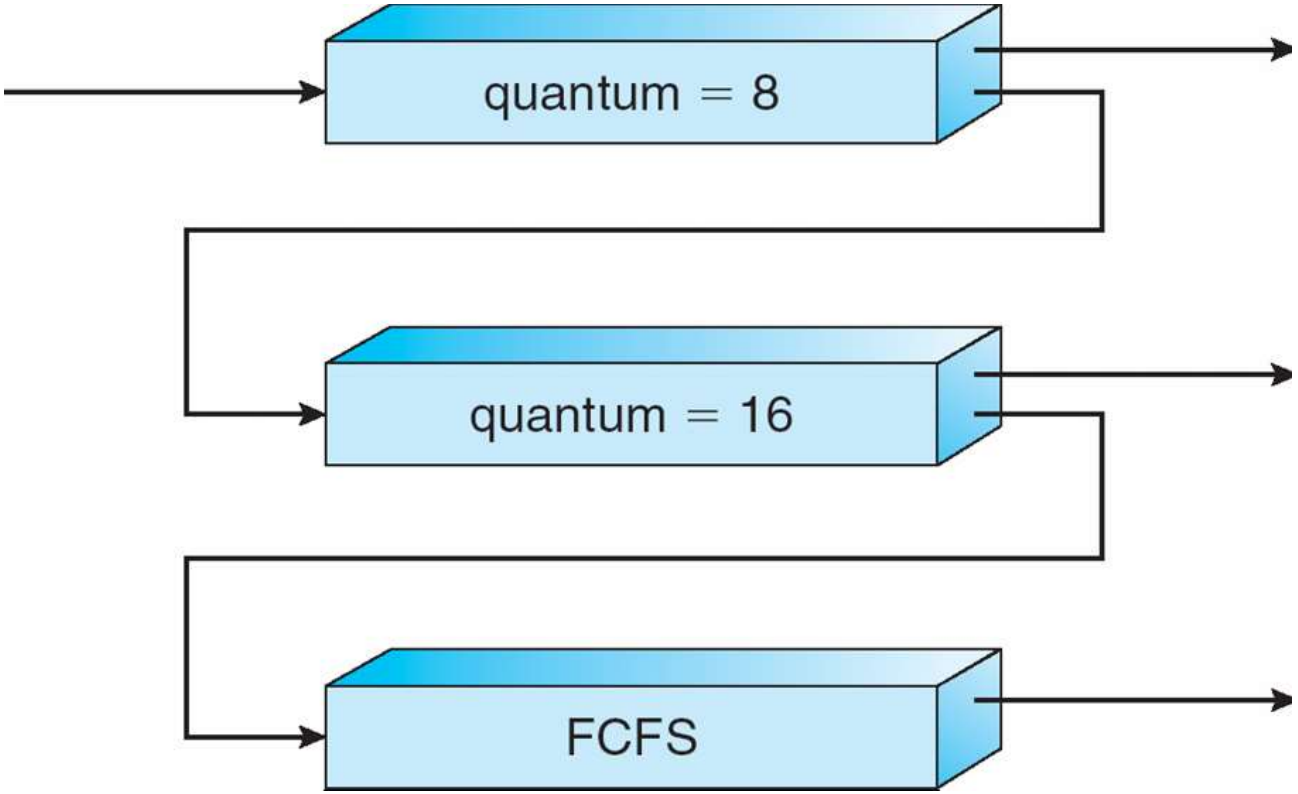
Multilevel Feedback Queue

- 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 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.

Multilevel Feedback Queues



Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens