**UNIT -2**

**PROCESS MANAGEMENT**

**Process Concept**



An operating system executes a variety of programs:



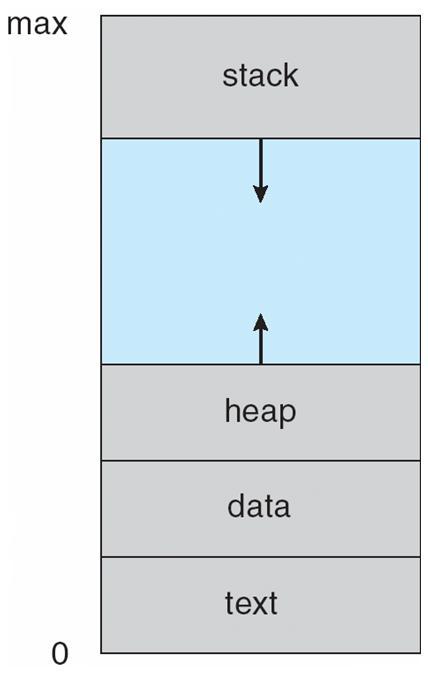
Batch system – jobs

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

A process includes:



program counter

stack

data section

**Process in Memory**

**Process State**

As a process executes, it changes *stat*e



**new**: The process is being created



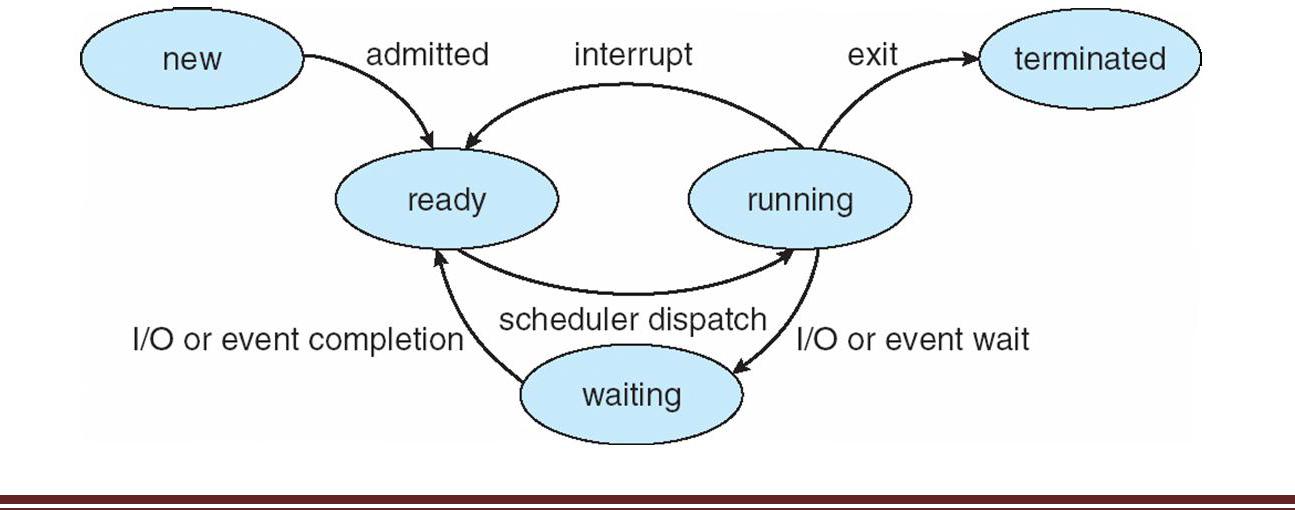
**running**: Instructions are being executed

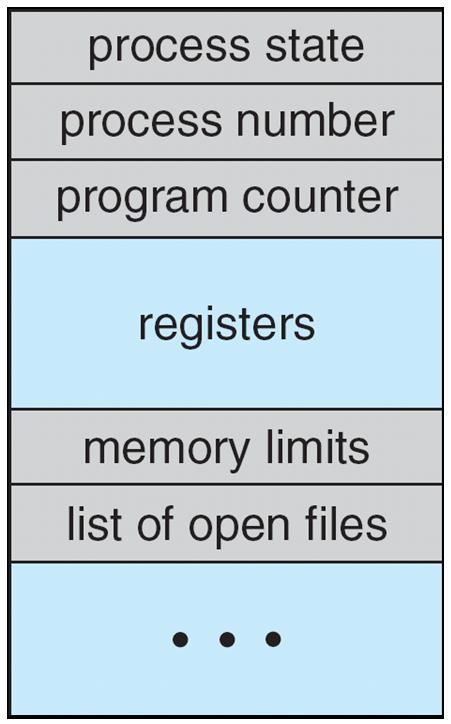
**waiting**: The process is waiting for some event to occur

**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



Process state

Program counter

CPU registers



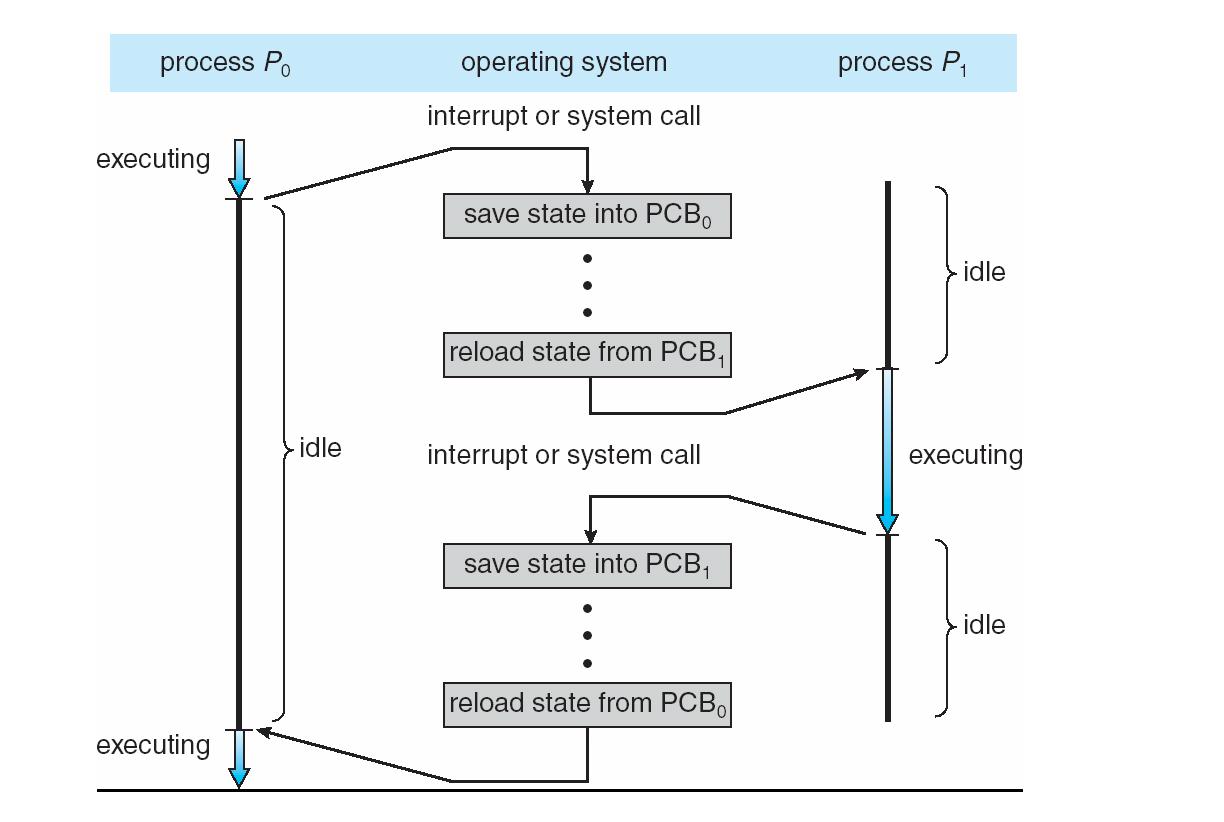
CPU scheduling information

Memory-management information

Accounting information

I/O status information

**CPU Switch From Process to Process**



**Process Scheduling Queues**



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

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

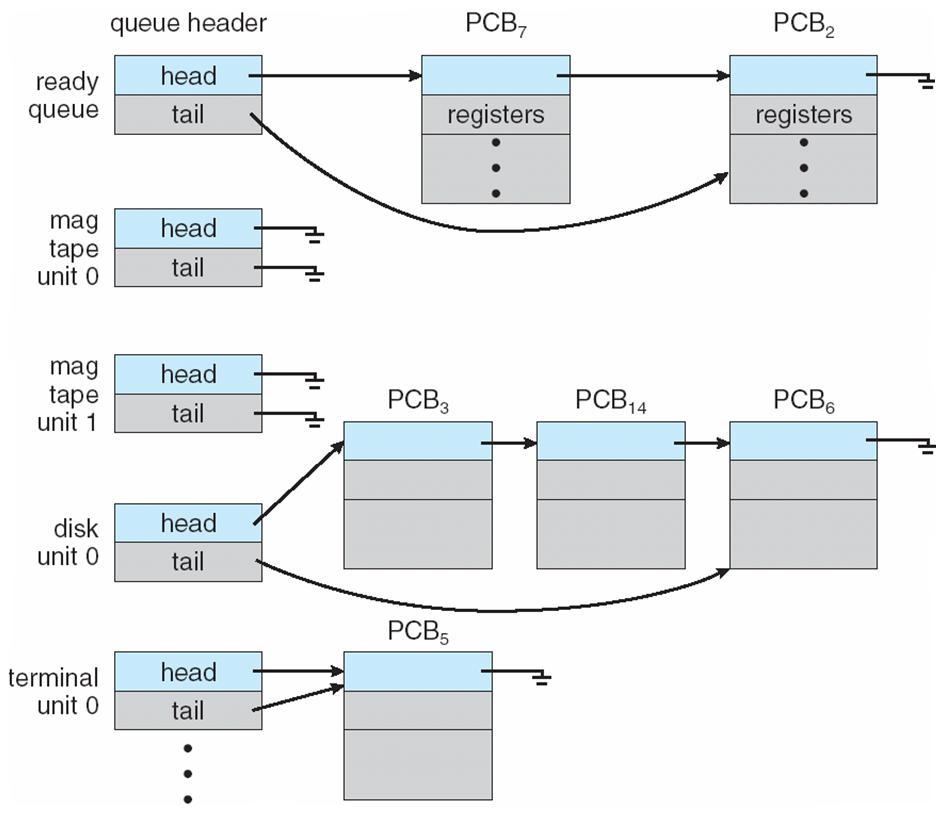
**Device queues** –set of processes waiting for an I/O device



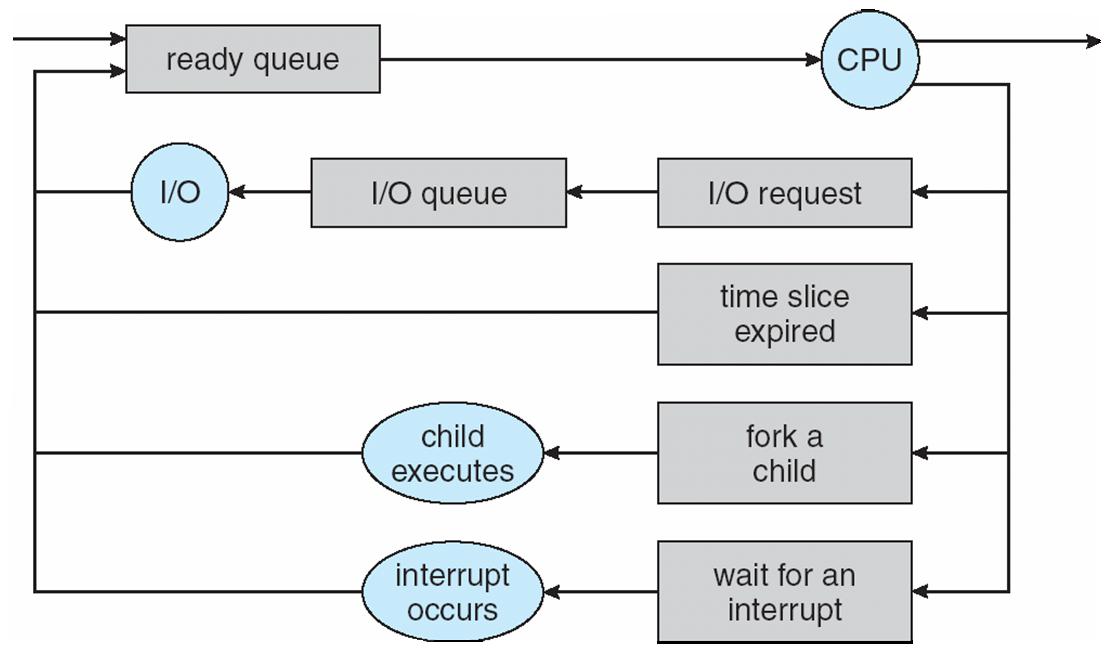
Processes migrate among the various queues



**Ready Queue and Various I/O Device Queues**



**Representation of Process Scheduling**



**Schedulers**



**Long-term scheduler** (or job scheduler)–selects which processes should be brought into the readyqueue

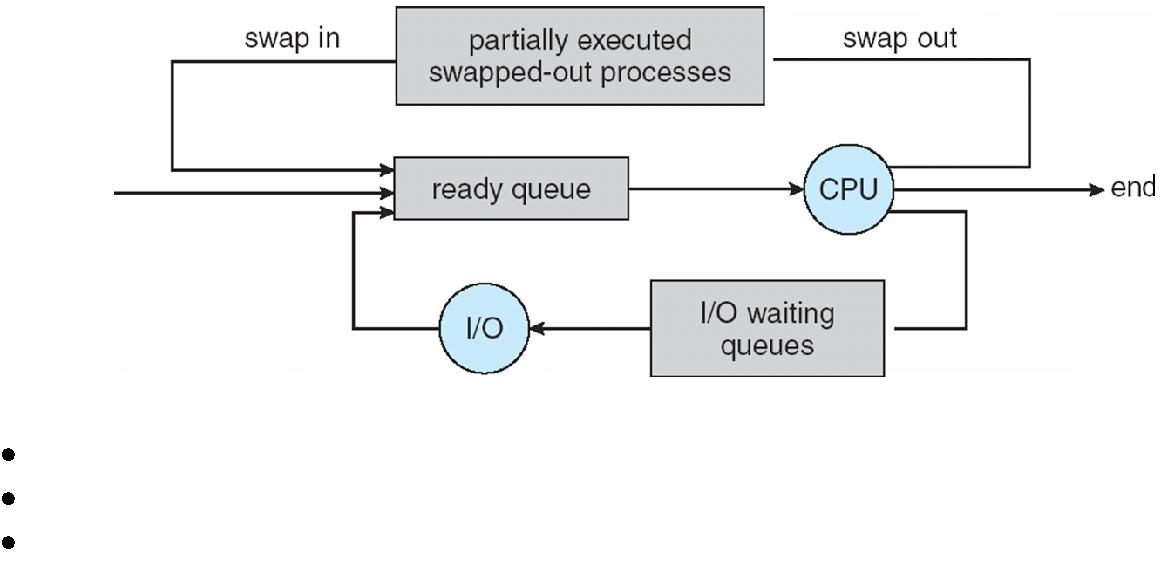


**Short-term scheduler** (or CPU scheduler)–selects which process should be executed next andallocates CPU



Dept. of Computer Science and Engineering

**Addition of Medium Term Scheduling**



Short-term scheduler is invoked very frequently (milliseconds) Þ (must be fast)

Long-term scheduler is invoked very infrequently (seconds, minutes) Þ (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

**Context Switch**



When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch



Context of a process represented in the PCB Context-switch time is overhead; the system does no useful work while switching Time dependent on hardware support



**Process Creation**



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



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

Address space

Child duplicate of parent



Child has a program loaded into it

UNIX examples

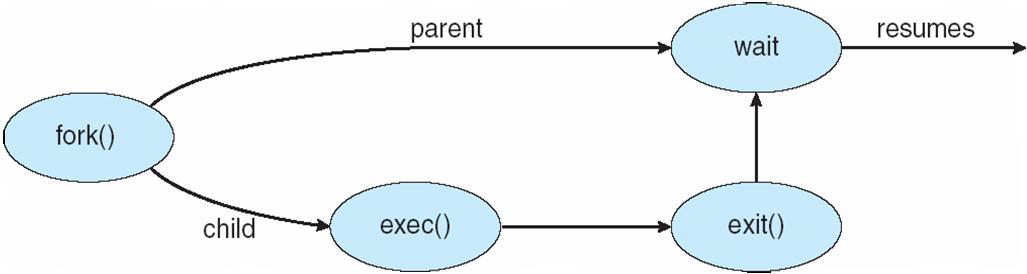
**fork** system call creates new process

**exec** system call used after a **fork** to replace the process’ memory space with a new program



Dept. of Computer Science and Engineering

**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);

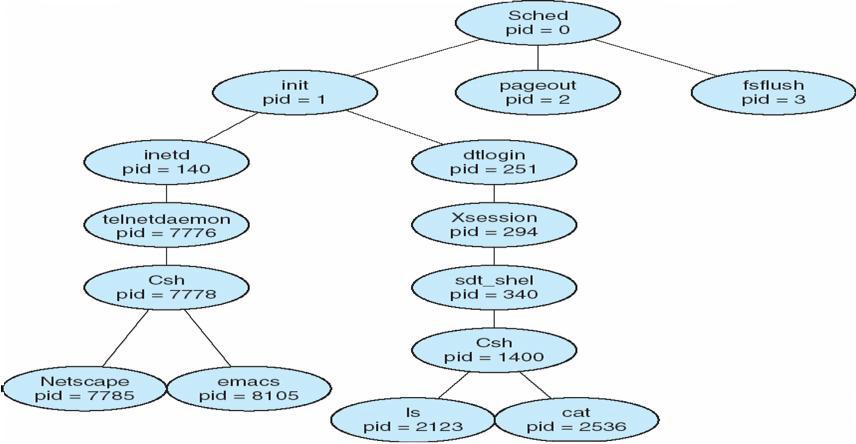
printf ("Child Complete");

exit(0);

}

}

**A tree of processes on a typical Solaris**



Dept. of Computer Science and Engineering

**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**)

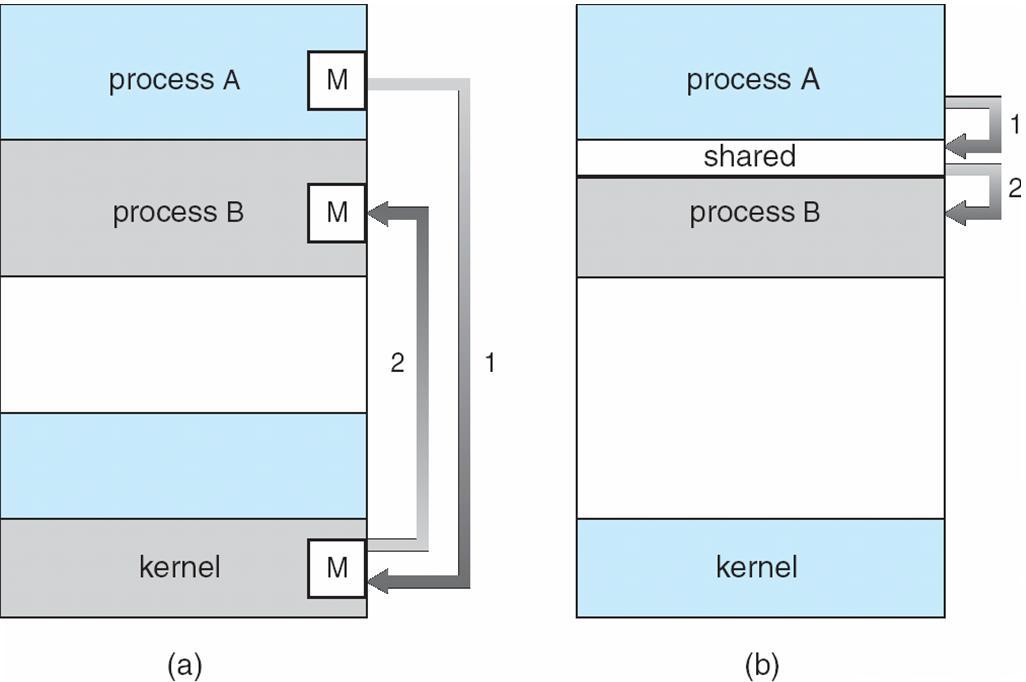


Two models of IPC

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 processAdvantages of process cooperation



Information sharing

Computation speed-up

Modularity

Convenience



Dept. of Computer Science and Engineering

**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;

}

**Interprocess Communication – Message Passing**



Mechanism for processes to communicate and to synchronize their actions



Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

**send**(*message*)–message size fixed or variable

**receive**(*message*)

If *P* and *Q* wish to communicate, they need to:

establish a *communication link* between them



exchange messages via send/receive

Implementation of communication link

physical (e.g., shared memory, hardware bus)

logical (e.g., logical properties)



Dept. of Computer Science and Engineering

**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 QProperties 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 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 AMailbox sharing

*P1, P2,* and *P3* share mailbox A



*P1*, sends; *P2* and *P3* 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



Dept. of Computer Science and Engineering

**Buffering**

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

1. Zero capacity – 0 messages
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 - POSIX**



POSIX Shared Memory

Process first creates shared memory segment

segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR); Process wanting access to that shared memory must attach to it shared memory = (char \*) shmat(id, NULL, 0); Now the process could write to the shared memory



printf(shared memory, "Writing to shared memory");

When done a process can detach the shared memory from its address space shmdt(shared memory);

**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()

**Examples of IPC Systems – Windows XP**



Message-passing centric via local procedure call (LPC) facility Only works between processes on the same system

Uses ports (like mailboxes) to establish and maintain communication channels

Communication works as follows:

The client opens a handle to the subsystem’s connection port object The client sends a connection request

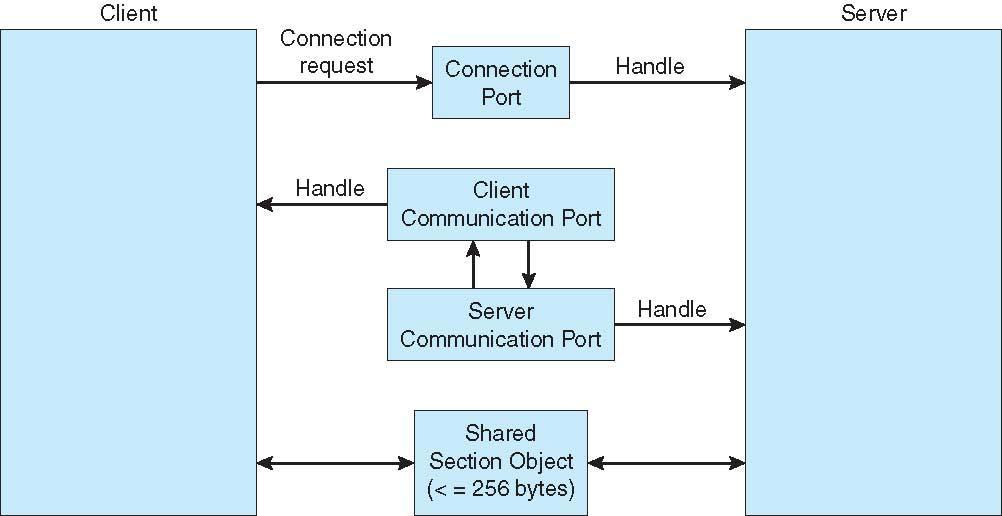
The server creates two private communication ports and returns the handle to one of them to the client The client and server use the corresponding port handle to send messages or callbacks and to listen for

replies



Dept. of Computer Science and Engineering

**Local Procedure Calls in Windows XP**



**Threads**

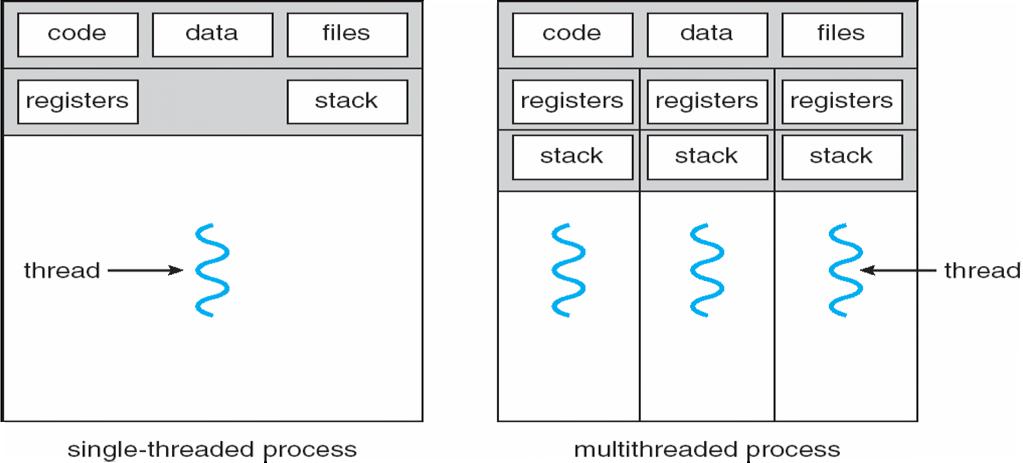


To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems



To discuss the APIs for the Pthreads, Win32, and Java thread libraries To examine issues related to multithreaded programming

**Single and Multithreaded Processes**



**Benefits**



Responsiveness



Resource Sharing

Economy

Scalability

**Multicore Programming**

Multicore systems putting pressure on programmers, challenges include



**Dividing activities**

**Balance**

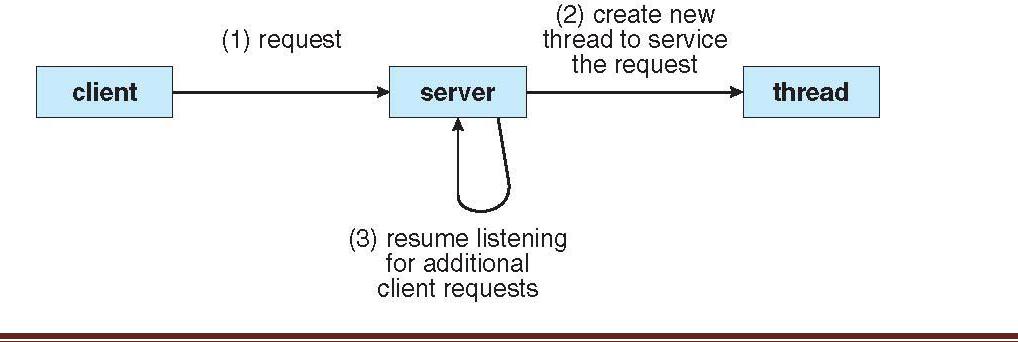
**Data splitting**



**Data dependency**

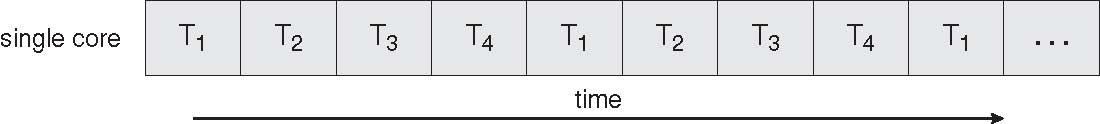
**Testing and debugging**

**Multithreaded Server Architecture**

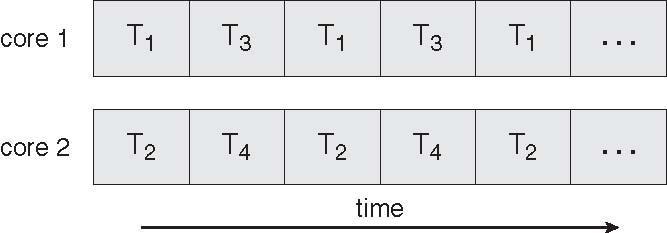


Dept. of Computer Science and Engineering

**Concurrent Execution on a Single-core System**



**Parallel Execution on a Multicore System**



**User Threads**



Thread management done by user-level threads librarynThree primary thread libraries:

POSIX Pthreadsl Win32 threads

Java threads

**Kernel Threads**

Supported by the Kernel

Examples



Windows XP/2000



Solaris

Linux

Tru64 UNIX

Mac OS X

**Multithreading Models**

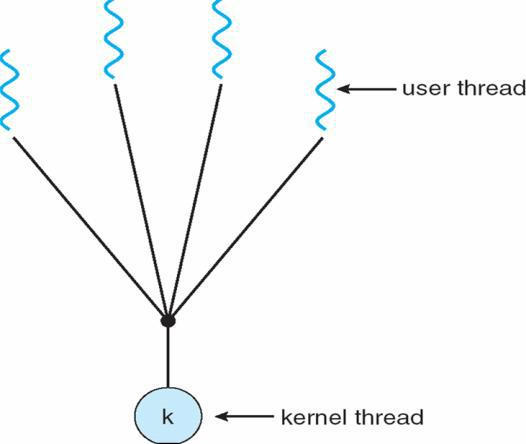


Many-to-One

One-to-One



Many-to-Many



**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

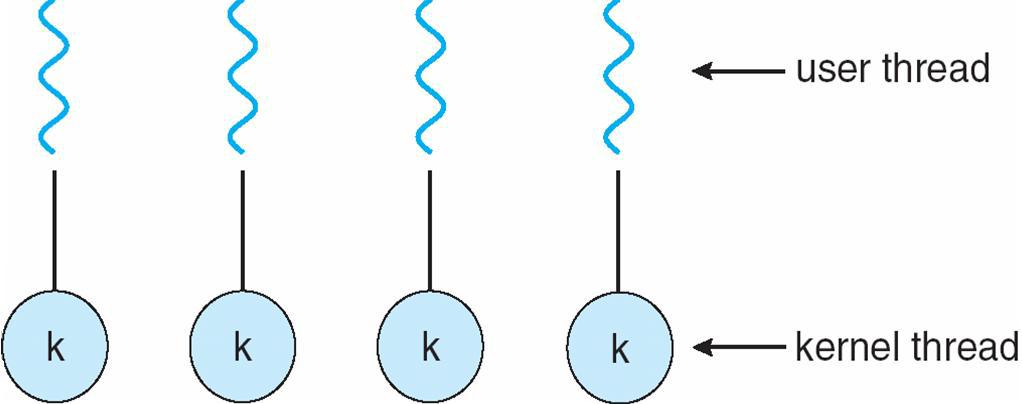
Windows NT/XP/2000

Linux



Dept. of Computer Science and Engineering

Solaris 9 and later



**Many-to-Many Model**

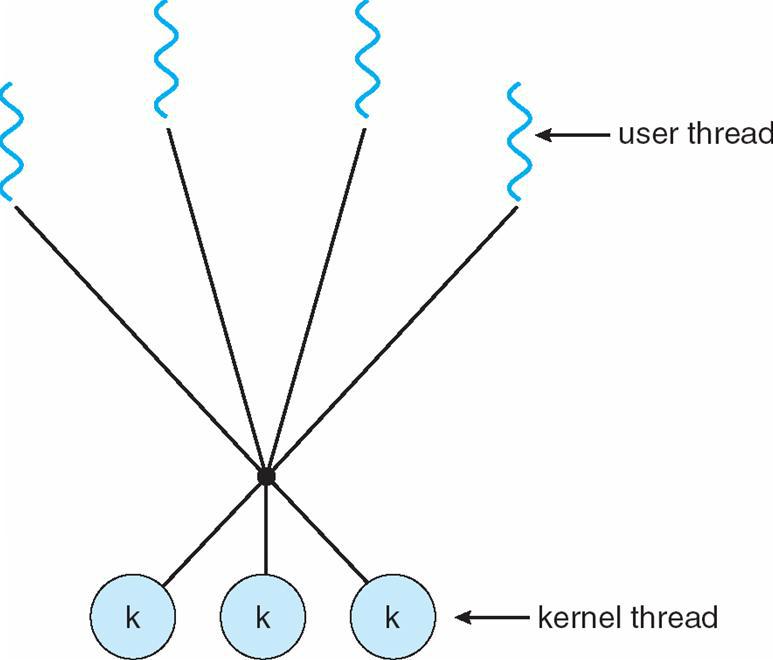


Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create a sufficient number of kernel threads

Solaris prior to version 9

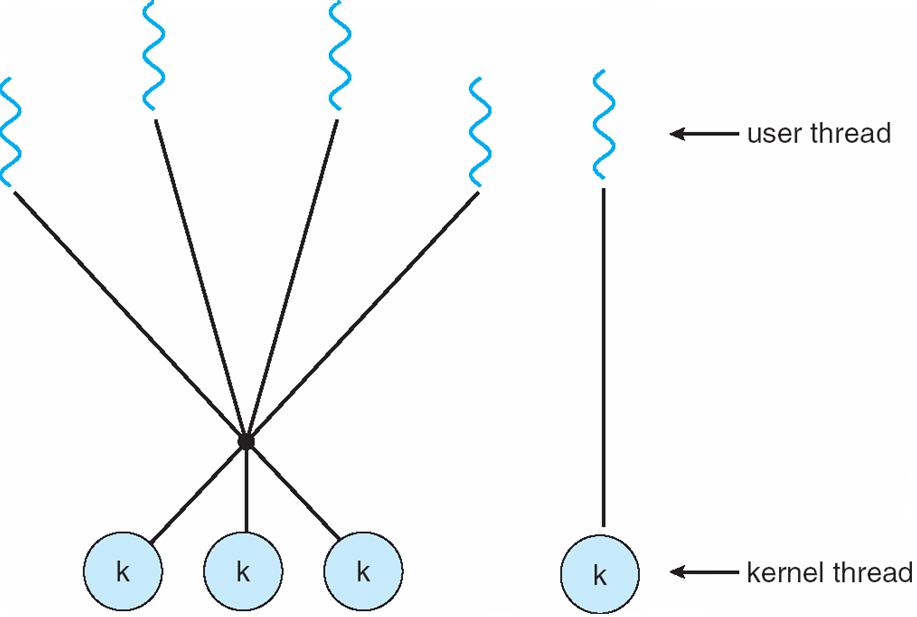
Windows NT/2000 with the *ThreadFiber* package



**Two-level Model**

Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

Examples



IRIX



HP-UX

Tru64 UNIX

Solaris 8 and earlier



Dept. of Computer Science and Engineering

**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:lExtending 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

**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

**Signal Handling**



Signals are used in UNIX systems to notify a process that a particular event has occurred

A signal handler is used to process signals

1.Signal is generated by particular event

2.Signal is delivered to a process



3.Signal is handled

Options:

Deliver the signal to the thread to which the signal applies

Deliver the signal to every thread in the process

Deliver the signal to certain threads in the process

Assign a specific threa to receive all signals for the process

**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



Dept. of Computer Science and Engineering

**Thread Specific Data**



Allows each thread to have its own copy of data

Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

**Scheduler Activations**



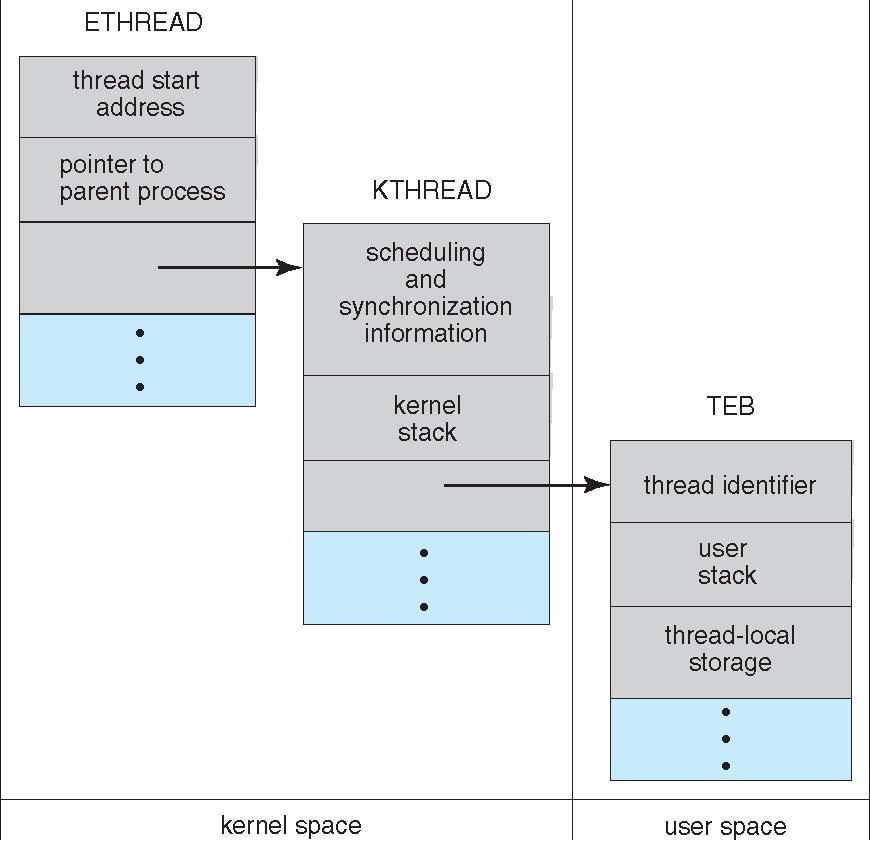
Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application



Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library



This communication allows an application to maintain the correct number kernel threads **Windows XP Threads**



Implements the one-to-one mapping, kernel-level



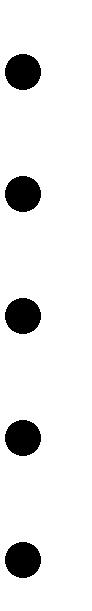
Each thread contains

A thread id

Register set

Separate user and kernel stacks

Private data storage area



The register set, stacks, and private storage area are known as the context of the threads

The primary data structures of a thread include:

ETHREAD (executive thread block)

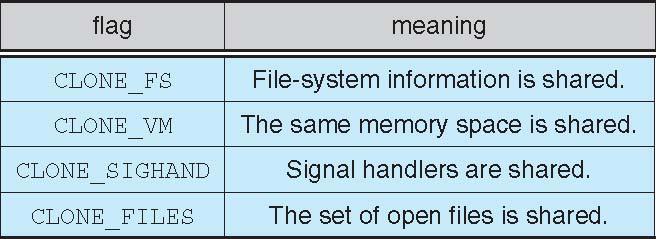
KTHREAD (kernel thread block)

TEB (thread environment block)



Dept. of Computer Science and Engineering

**Linux Threads**



Linux refers to them as *tasks* rather than *threads*

Thread creation is done through **clone()** system call

**clone()** allows a child task to share the address space of the parent task (process)

**CPU Scheduling**

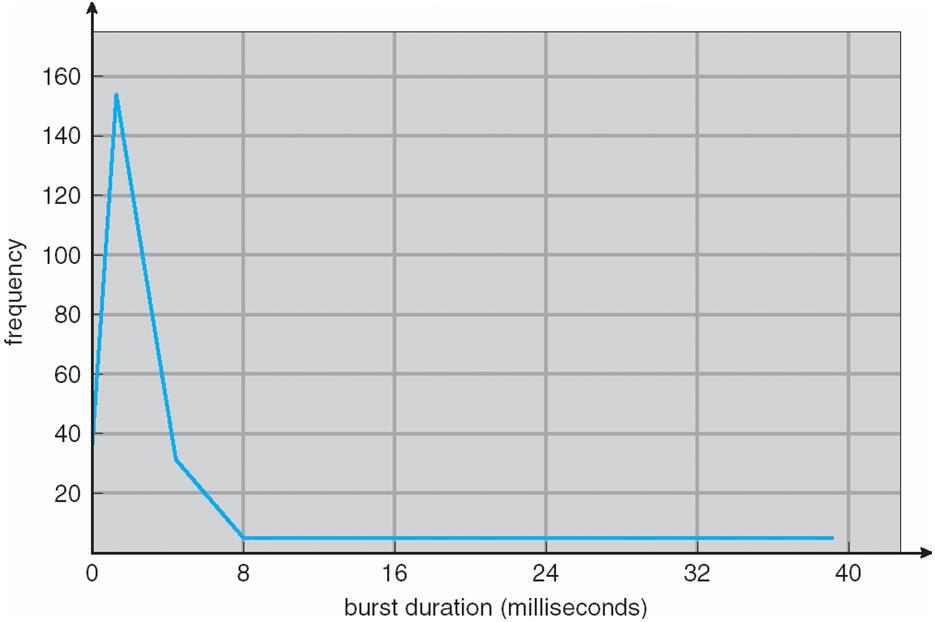


To introduce CPU scheduling, which is the basis for multiprogrammed operating systems To describe various CPU-scheduling algorithms

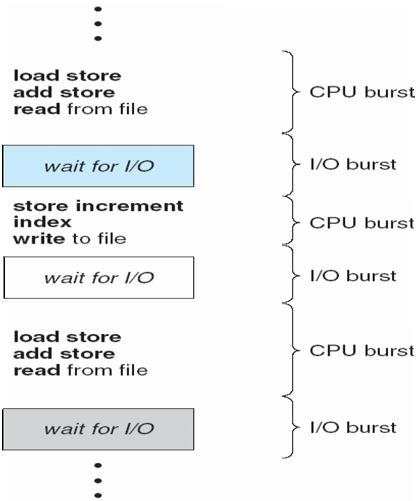
To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system Maximum CPU utilization obtained with multiprogramming

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 isproduced, not output (for time-sharing environment)



Max CPU utilization

Max throughput

Min turnaround time

Min waiting time

Min response time



Dept. of Computer Science and Engineering

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **First-Come, First-Served (FCFS) Scheduling** | | | | | |  |  |  |  |  |
| Process |  | | Burst Time |  | |  |  |  |  |  |
| *P1* | | | 24 |  |  |  |  |  |  |  |
| *P2* | | | 3 |  |  |  |  |  |  |  |
| *P3* | | | 3 |  |  |  |  |  |  |  |
| Suppose that the processes arrive in the order: *P1* , *P2* , *P3* | | | | | |  |  |  |  |  |
| The Gantt Chart for the schedule is: | | |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | |  |
|  |  | P1 |  |  |  | P2 |  | P3 |  |  |
|  | |  |  | |  |  |  |  |  |  |
|  | |  |  | |  |  |  | 30 |  |  |
| 0 | | | 24 | | | 27 | |  |  |

Waiting time for *P1* = 0; *P2* = 24; *P3* = 27

Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order

*P2* , *P3* , *P1*

The Gantt chart for the schedule is:nnnnWaiting time for *P1 =* 6*; P2* = 0*; P3 =* 3nAverage waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case

*Convoy effect* short process behind long process

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | P2 |  |  | P3 |  | P1 |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| 0 | | 3 | | 6 | | 30 | |

**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

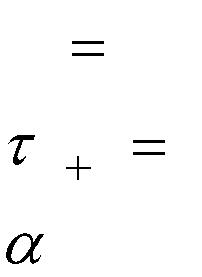
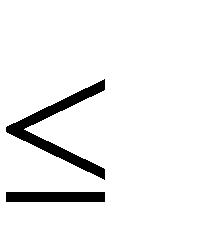
 SJF is optimal – gives minimum average waiting time for a given set of processes The difficulty is knowing

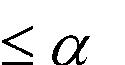
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Process |  | Arrival Time |  | Burst Time |  |
|  | *P1* | 0.0 | |  | 6 |  |
|  | *P2* | 2.0 | |  | 8 |  |
|  | *P3* | 4.0 | |  | 7 |  |
|  | *P4* | 5.0 | |  | 3 |  |
|  |  | |  |  |  |  |
|  |  | |  |  |  |  |

SJF scheduling chart

average waiting time = (3 + 16 + 9 + 0) / 4 = 7the length of the next CPU request

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | P |  | P1 |  |  | P3 |  | P2 | |  |  |
|  | 4 |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | |  |
| 0 | | 3 | | 9 | | 16 | | 24 | | |  |
| **Determining Length of Next CPU Burst** | | | |  |  |  |  |  |  |  |  |
| 1. *tn* | | actual length of *nth* CPU burst | | | | | |  |  |  |  |
| 2. *n*1 | | predicted value for the next CPU burst | | | | | | | | |  |



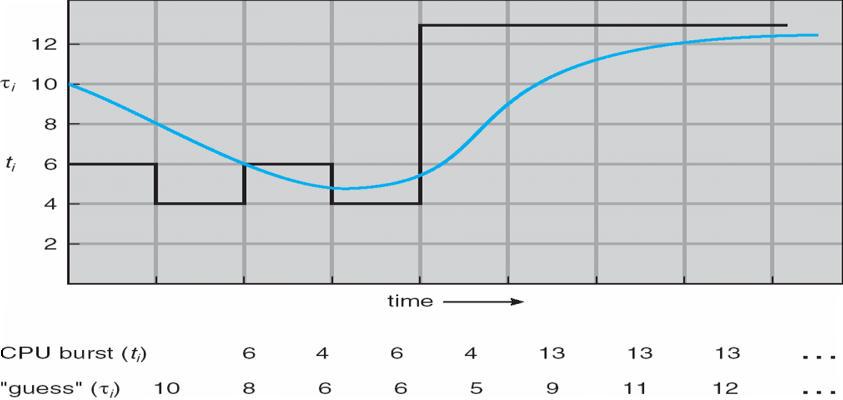
1. , 0  1
   1. Define :



Can only estimate the length

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

**Prediction of the Length of the Next CPU Burst**



**Examples of Exponential Averaging**

a =0

tn+1 = tn

Recent history does not count

a =1

tn+1 = a *t*n

Only the actual last CPU burst counts

If we expand the formula, we get:

t*n*+1 = a t*n*+(1 *-* a*)*a *tn* -1 + …

*+(*1 - a *)j*a *tn* -*j* + …



Dept. of Computer Science and Engineering

*+(*1 - a *)n*+1t0

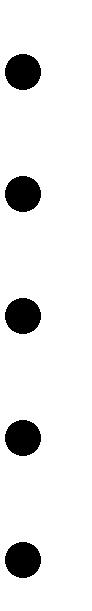
Since both a and (1 - a) are less than or equal to 1, each successive term has less weight than its predecessor

**Priority Scheduling**



A priority number (integer) is associated with each process

The CPU is allocated to the process with the highest priority (smallest integer º highest priority)



Preemptive

nonpreemptive

SJF is a priority scheduling where priority is the predicted next CPU burst time Problem º **Starvation** – low priority processes may never execute

Solution º **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.



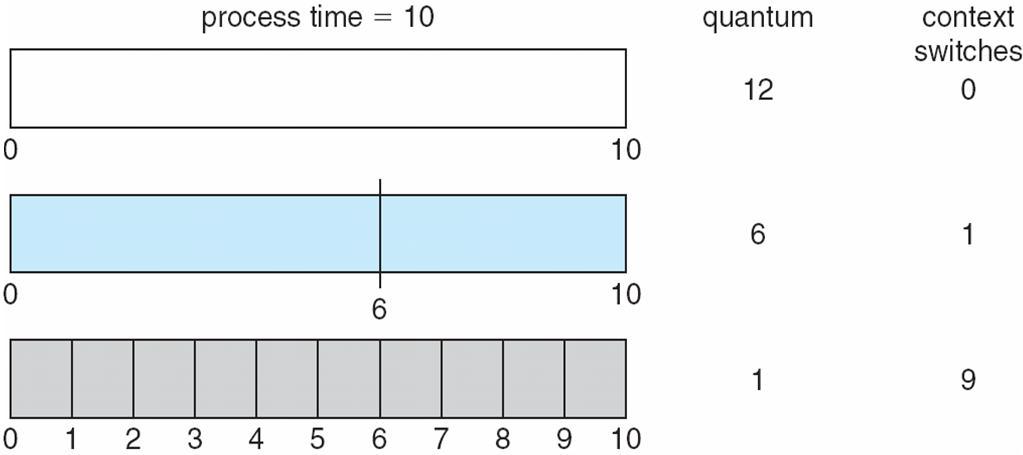
Performance

*q* large Þ FIFO

*q* small Þ *q* must be large with respect to context switch, otherwise overhead is too high

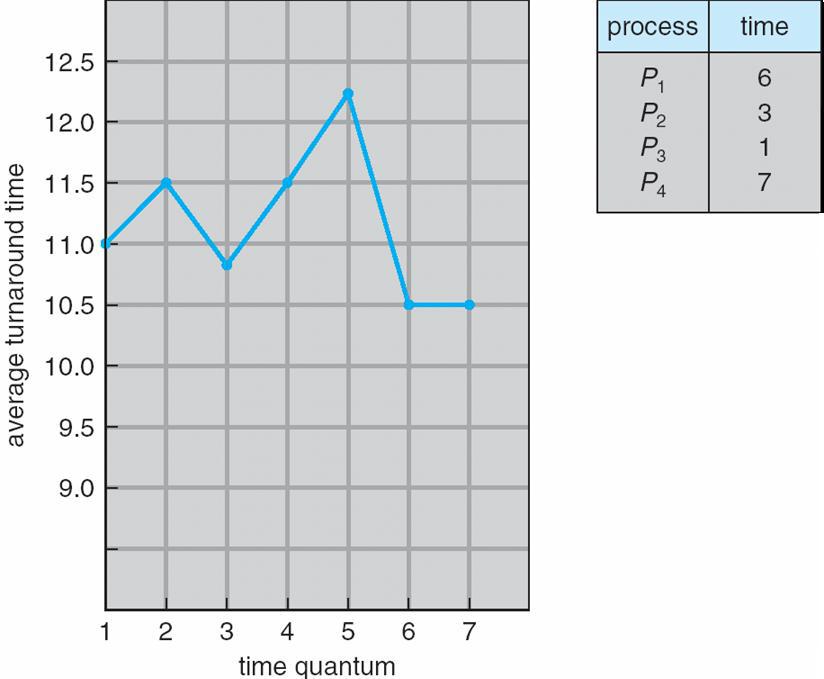
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Example of RR with Time Quantum = 4** | | | | | | | | | | |  |  |  |  |  |  |  |  |  |  |
| Process |  | |  |  |  | Burst Time | | | | |  |  |  |  |  |  |  |  |  |  |
| *P1* |  | |  |  |  |  |  | 24 |  |  |  |  |  |  |  |  |  |  |  |  |
| *P2* | | |  |  |  |  |  | 3 |  |  |  |  |  |  |  |  |  |  |  |  |
| *P3* | | |  |  |  |  |  | 3 |  |  |  |  |  |  |  |  |  |  |  |  |
| The Gantt chart is: | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | |  |  |  |  | |  |  |  | |  |  |  |  |  |  |  |  |  |  |
| Typically, | |  | P1 |  | P2 | |  | P3 | P1 | |  | P1 |  |  | P1 |  | P1 |  | P1 |  |
|  | |  |  |  |  | |  |  |  | |  |  |  |  |  |  |  |  |  |  |
| 0 | | | 4 | | 7 | | | 10 | | | 14 | | 18 | | 22 | | 26 | | 30 | |

**Time Quantum and Context Switch Time**



Dept. of Computer Science and Engineering

**Turnaround Time Varies With The Time Quantum**



**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

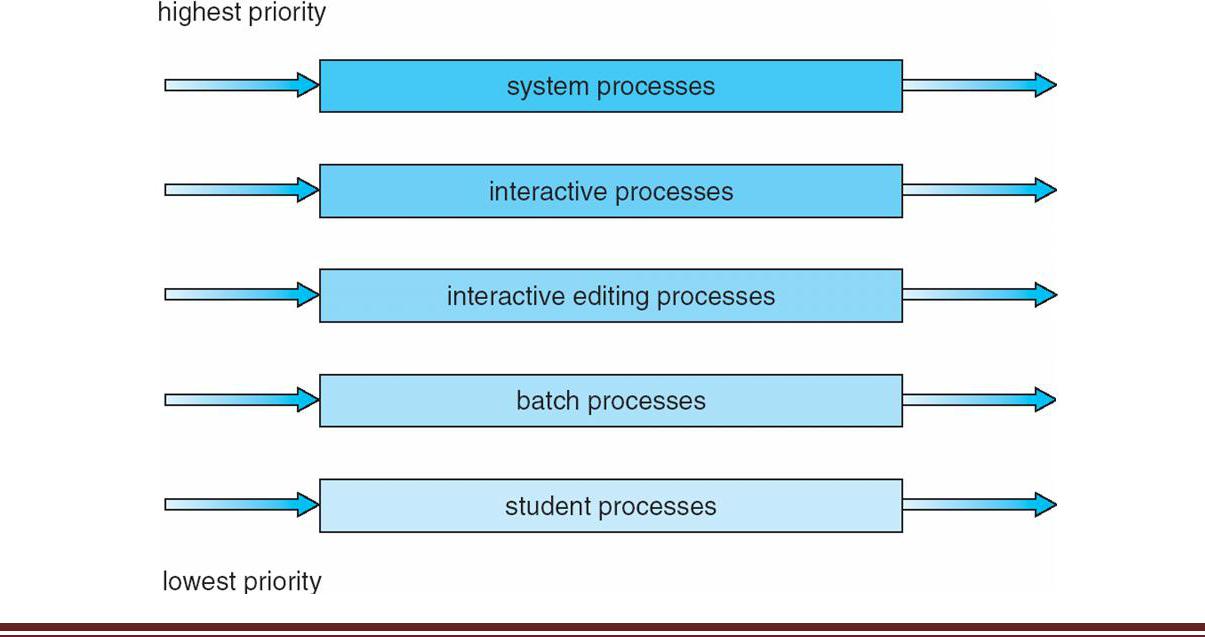
Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.



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**



Dept. of Computer Science and Engineering

**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



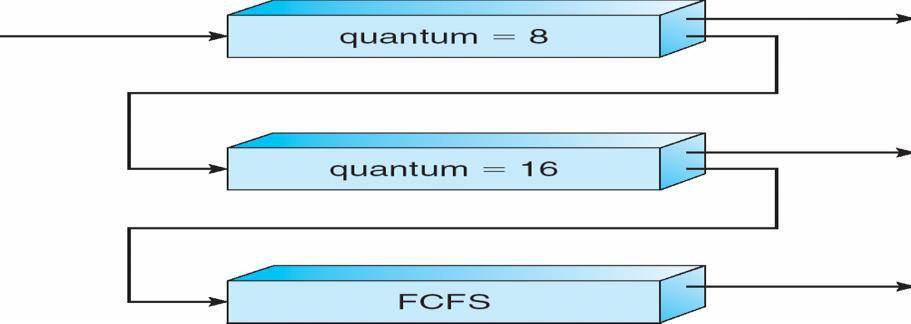
Scheduling

A new job enters queue *Q0* 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**



**Thread Scheduling**



Distinction between user-level and kernel-level threads Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP Known as **process-contention scope (PCS)** since scheduling competition is within the process Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system



**Pthread Scheduling**



API allows specifying either PCS or SCS during thread creation

PTHREAD SCOPE PROCESS schedules threads using PCS scheduling

PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

**Pthread Scheduling API**

#include <pthread.h>

#include <stdio.h>

#define NUM THREADS 5

int main(int argc, char \*argv[])



Dept. of Computer Science and Engineering

{

int i; pthread t tid[NUM THREADS];

pthread attr t attr;

/\* get the default attributes \*/

pthread attr init(&attr);

/\* set the scheduling algorithm to PROCESS or SYSTEM \*/ pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM); /\* set the scheduling policy - FIFO, RT, or OTHER \*/ pthread attr setschedpolicy(&attr, SCHED OTHER); /\* create the threads \*/

for (i = 0; i < NUM THREADS; i++)

pthread create(&tid[i],&attr,runner,NULL); /\* now join on each thread \*/

for (i = 0; i < NUM THREADS; i++)

pthread join(tid[i], NULL);

}

/\* Each thread will begin control in this function \*/

void \*runner(void \*param)

{

printf("I am a thread\n");

pthread exit(0);

}

**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 theneed for data sharing



**Symmetric multiprocessing (SMP)** –each processor is self-scheduling, all processes in common readyqueue, or each has its own private queue of ready processes

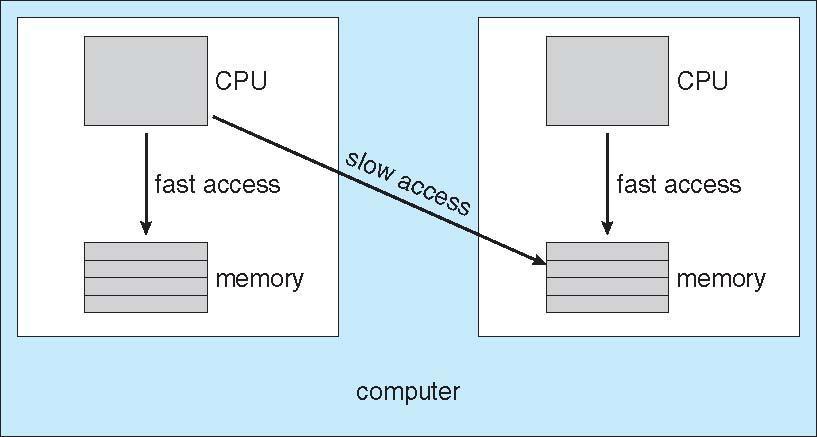


**Processor affinity** –process has affinity for processor on which it is currently running **soft affinity**



**hard affinity**

**NUMA and CPU Scheduling**



Dept. of Computer Science and Engineering

**Multicore Processors**



Recent trend to place multiple processor cores on same physical chip

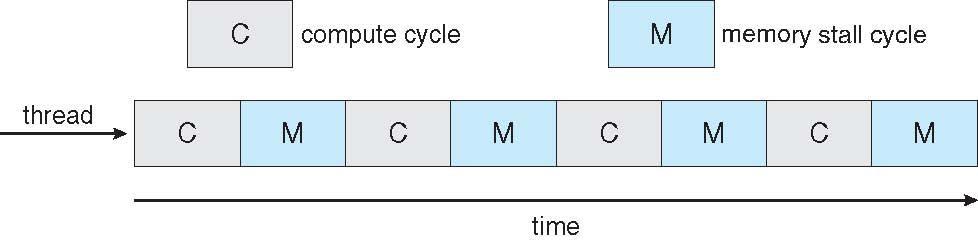
Faster and consume less power

Multiple threads per core also growing



Takes advantage of memory stall to make progress on another thread while memory retrieve happens

**Multithreaded Multicore System**



**Operating System Examples**

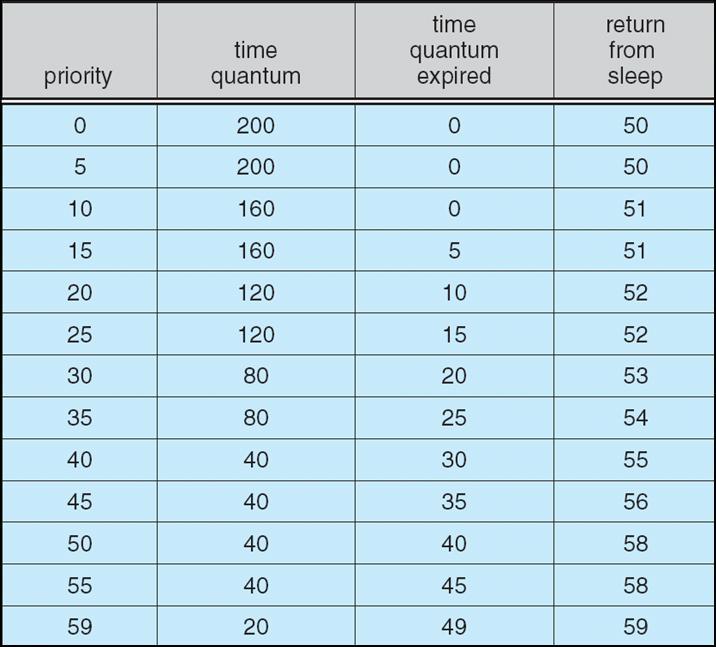


Solaris scheduling

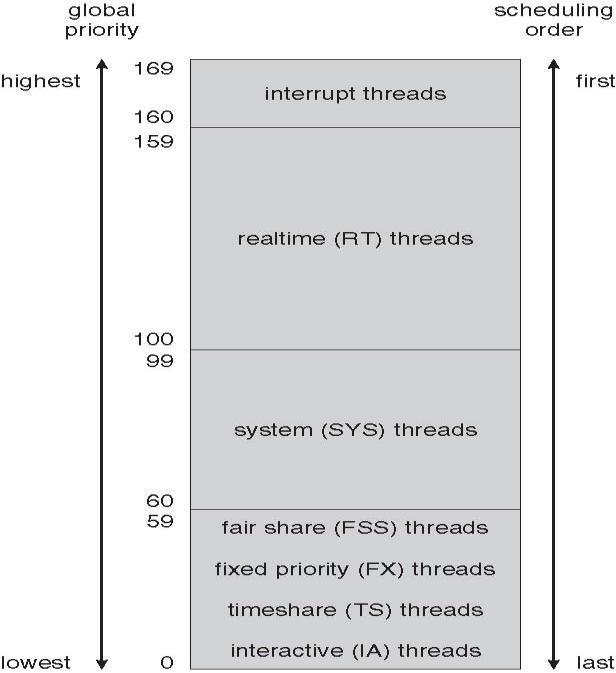
Windows XP scheduling

Linux scheduling

**Solaris Dispatch Table**



Dept. of Computer Science and Engineering



**Solaris Scheduling**

**Windows XP Priorities**



**Linux Scheduling**



Constant order *O*(1) scheduling time

Two priority ranges: time-sharing and real-time

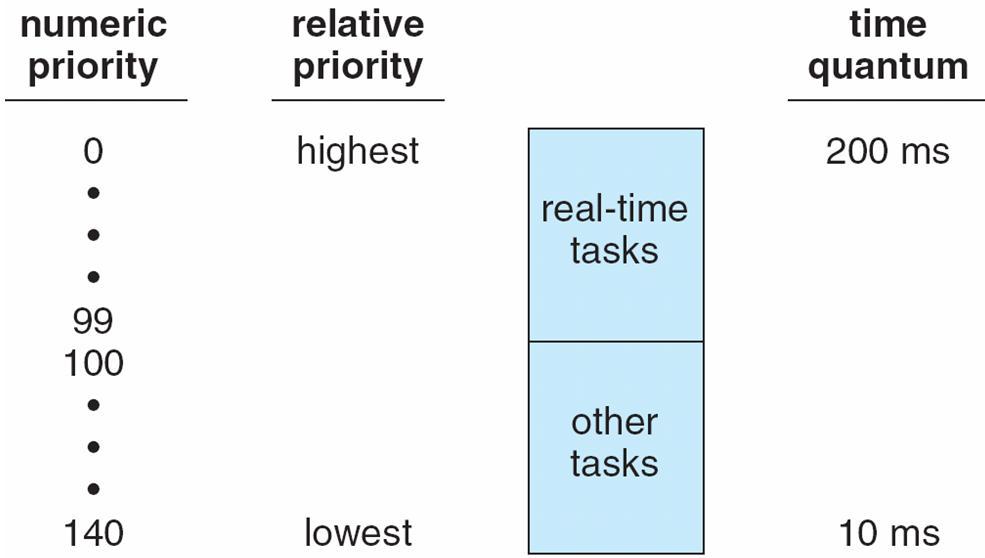
**Real-time** range from 0 to 99 and **nice** value from 100 to 140



Dept. of Computer Science and Engineering

Shri Vishnu Engineering College for Women

**Priorities and Time-slice length**



**List of Tasks Indexed According to Priorities**



**Algorithm Evaluation**



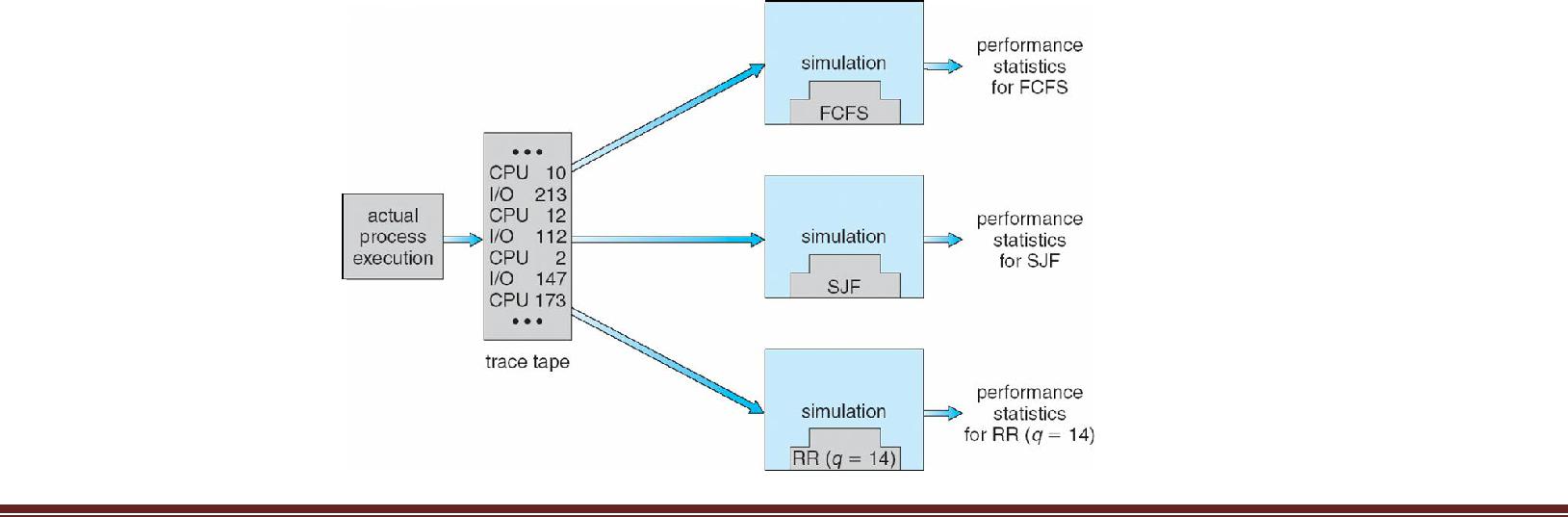
Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload



Queuing models

Implementation

**Evaluation of CPU schedulers by Simulation**



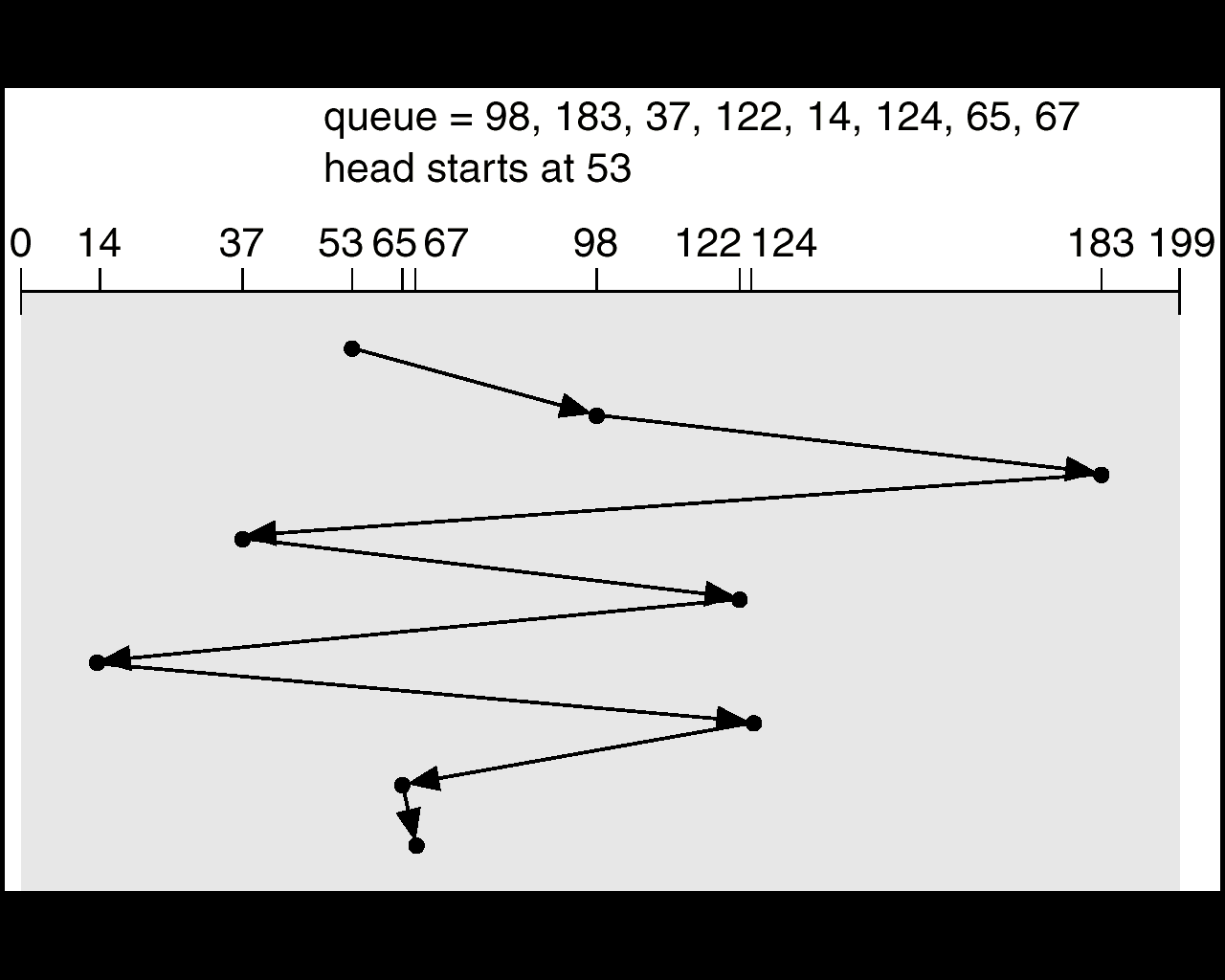
**Disk Scheduling**

* **We illustrate them with a request queue (0-199).**

**98, 183, 37, 122, 14, 124, 65, 67**

**Head pointer 53**

**FCFS**

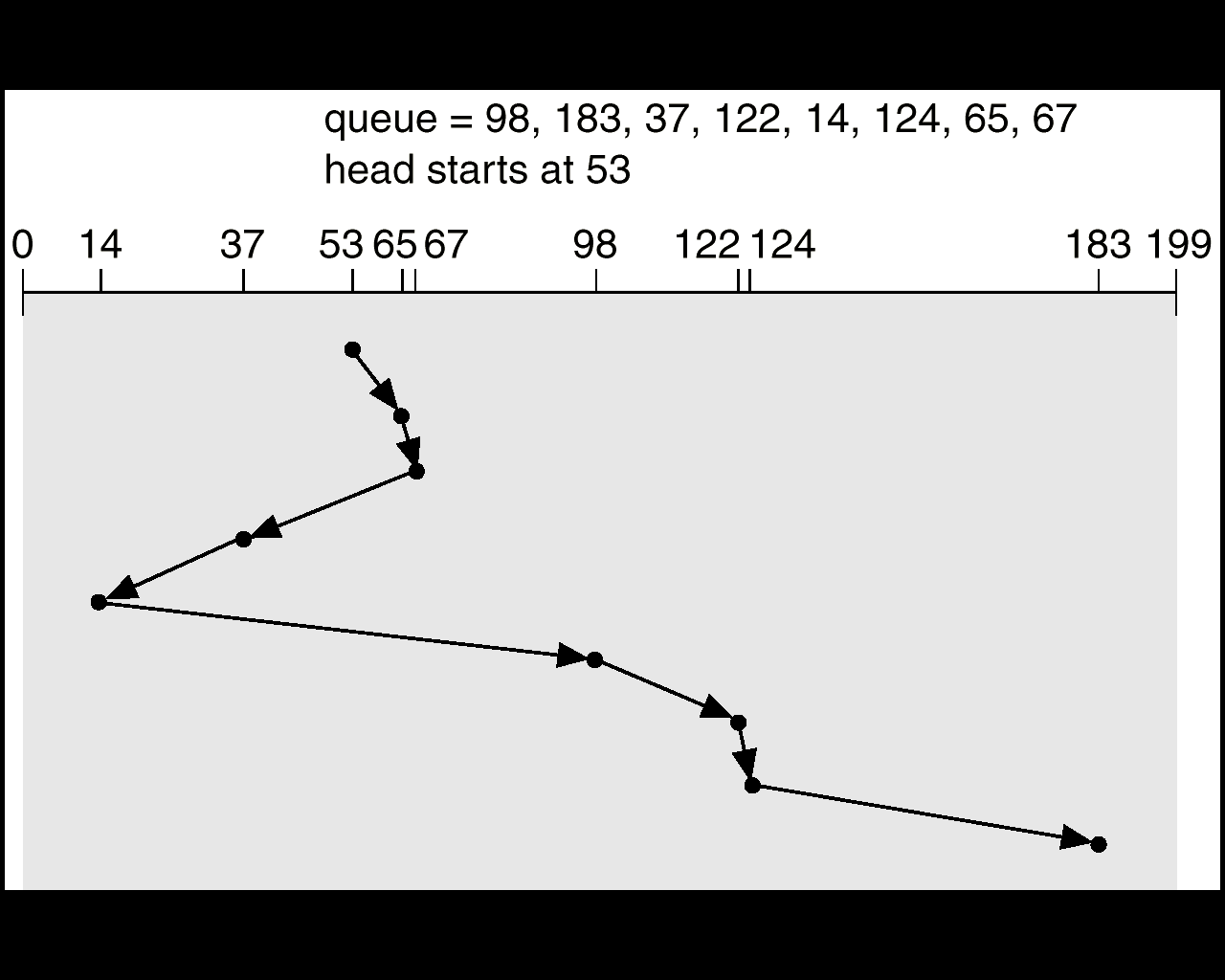
****

**Illustration shows total head movement of 640 cylinders**

**SSTF**

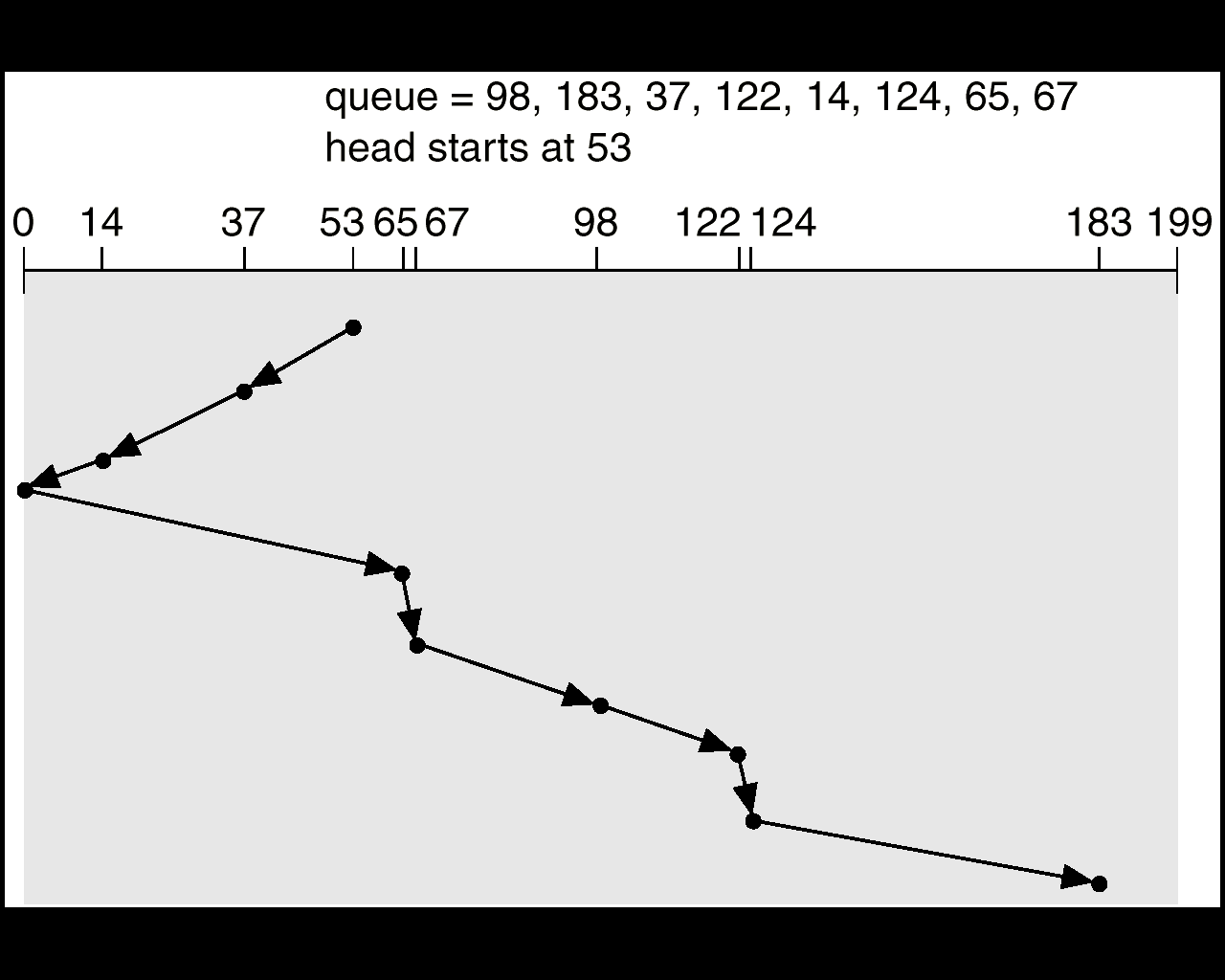
* **Selects the request with the minimum seek time from the current head position.**
* **SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.**
* **Illustration shows total head movement of 236 cylinders.**

**SSTF**

****

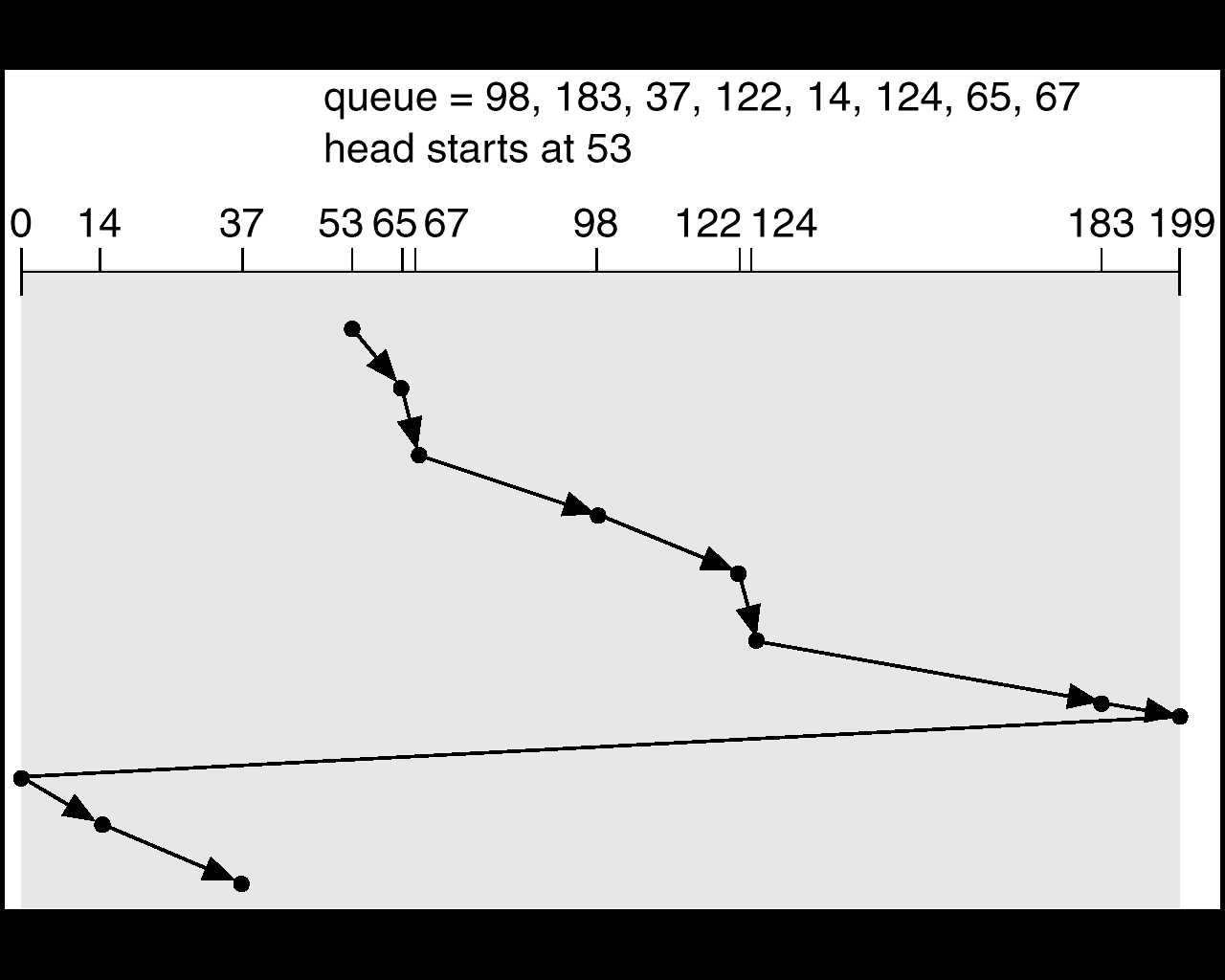
**SCAN**

* **The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.**
* **Sometimes called the *elevator algorithm*.**
* **Illustration shows total head movement of 208 cylinders.**

****

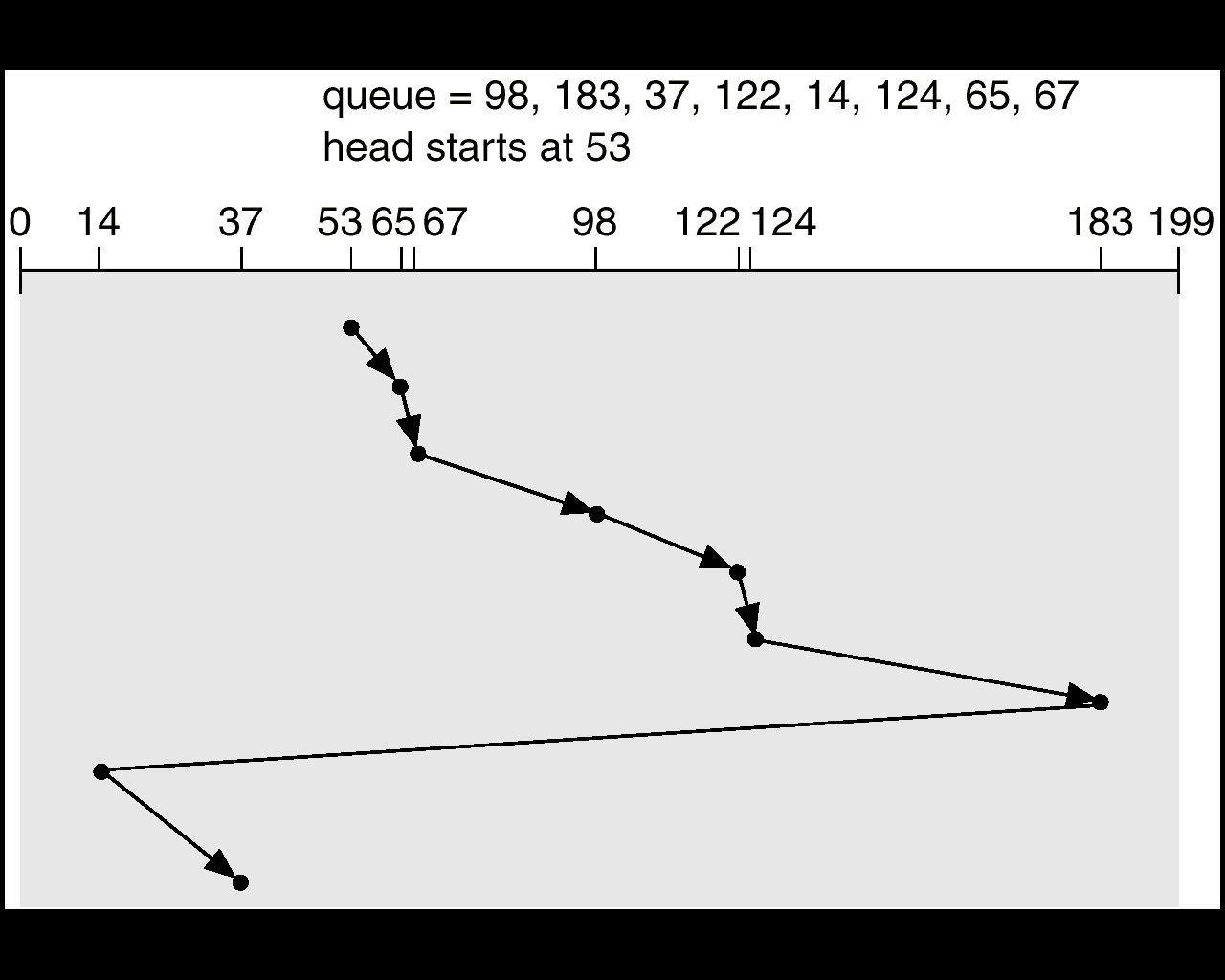
**C-SCAN**

* **Provides a more uniform wait time than SCAN.**
* **The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.**
* **Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.**

****

**C-LOOK**

* **Version of C-SCAN**
* **Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.**

****