OPERATING SYSTEMS

UNIT – 1

INTRODUCTION AND SYSTEM ARCHITECTURE

1. What Operating systems Do?

Operating System is a program that acts as an intermediary between a user of a computer and the computer hardware

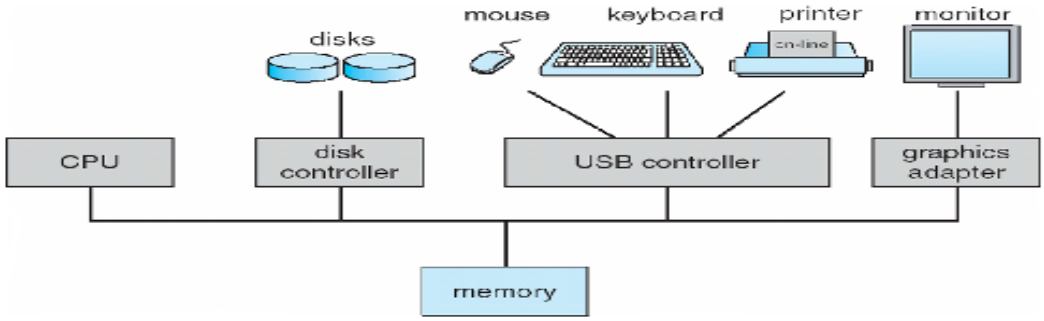
Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

2. Computer System Organization

Computer-System Operation:

When it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program.**it is stored in read-only memory (ROM) or electrically erasable programmable read-onlymemory (EEPROM), known by the general term **firmware,** within the computerhardware. It initializesall aspects of the system.



- ✓ The bootstrap program must know how to load the operating system and to start executing that system. The bootstrap program must locate and load into memory the operating system Kernel.
- ✓ The operating system then starts executing the first process,such as "init," and waits for some event to occur. The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.

- ✓ Hardware may trigger an interrupt at any timeby sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call).**

Storage Structure:

- ✓ Computer programs must be in main memory (also called **random-accessmemory** or **RAM)** to be executed.
- ✓ Main memory is the only large storage area that the processor can access directly. It commonlyis implemented in a semiconductor technology called **dynamic random-accessmemory (DRAM),** which forms an array of memory words. Each word has itsown address.
- ✓ A typical instruction-execution cycle, as executed on a system with a **von-Neumann** architecture.first fetches an instruction from memory and storesthat instruction in the **instruction register.** The instruction is then decodedand may cause operands to be fetched from memory and stored in someinternal register. After the instruction on the operands has been executed, the result may be stored back in memory.

1. Main memory is usually too small to store all needed programs and datapermanently.
2. Main memory is a *volatile* storage device that loses its contents whenpower is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be ableto hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk,** which provides storage for both programs and data.

3. Computer System Architecture

A computer system may be organized in a number of different ways, according to the number of general-purpose processors used.

Single-Processor Systems

- ✓ A single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.
- ✓ Almost all systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers
- ✓ All of these special-purpose processors run a limited instruction set and do not run user processes. Sometimes they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status.

For example, a disk-controller microprocessor receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm.

Multiprocessor systems

- ✓ Multiprocessor systems (also known as **parallel systems** or **tightly coupled systems)** are growing in importance. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main advantages:

1. Increased throughput: By increasing the number of processors, we expect to get more work done in less time.

2. Economy of scale: Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.

3. Increased reliability: If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.

If we have ten processors and one fail, then each of the remaining nine processors can pick up a share of the work of the failed processor.

The most common systems use **symmetric multiprocessing (SMP),** inwhich each processor performs all tasks within the operating system. SMP means that all processors are peers; no master-slave relationship exists between processors.

4. Operating System Structure

DISTRIBUTED SYSTEMS

Computing Environments

Traditional computer

- ✓ Blurring over time
- ✓ Office environment
  - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
  - Now portals allowing networked and remote systems access to sameresources
- ✓ Home networks
  - Used to be single system, then modems
  - Now firewalled, networked
- ✓ Client-Server Computing
- ✓ Dumb terminals supplanted by smart PCs
- ✓ Many systems now **servers**, responding to requests generated by **clients**
  - **Compute-server** provides an interface to client to request services (i.e. database)
  - **File-server** provides interface for clients to store and retrieve files

Peer-to-Peer Computing

- ✓ Another model of distributed system
- ✓ P2P does not distinguish clients and servers
- ✓ Instead all nodes are considered peers

- ✓ May each act as client, server or both
- ✓ Node must join P2P network
  - Registers its service with central lookup service on network, or
  - Broadcast request for service and respond to requests for service via **discovery protocol**
- ✓ Examples include *Napster* and *Gnutella*

Web-Based Computing

- ✓ Web has become ubiquitous
- ✓ PCs most prevalent devices
- ✓ More devices becoming networked to allow web access
- ✓ New category of devices to manage web traffic among similar servers: **load balancers**
- ✓ Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers

Open-Source Operating Systems

- ✓ Operating systems made available in source-code format rather than just binary closedsource
- ✓ Counter to the copy protection and Digital Rights Management (DRM) movement
- ✓ Started by Free Software Foundation (FSF), which has "copyleft" GNU Public License (GPL)
- ✓ Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and SunSolaris

5. Operating System Services

- ✓ 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
- ✓ **File-system manipulation -** The file system is of particular interest. Obviously, 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

- ✓ 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 - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
- ✓ **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 defendingexternal I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. Achain is only as strong as its weakest link.

User Operating System Interface - CLI

- ✓ Command Line Interface (CLI) or command interpreter allows direct command entry
    - Sometimes implemented in kernel, sometimes by systems program
    - Sometimes multiple flavors implemented – shells
    - Primarily fetches a command from user and executes it
- ✓ Sometimes commands built-in, sometimes just names of programs
- ✓ If the latter, adding new features doesn't require shell modification

User Operating System Interface - GUI

- ✓ User-friendly desktop metaphor interface
- ✓ Usually mouse, keyboard, and monitor
- ✓ Icons represent files, programs, actions, etc
- ✓ Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- ✓ Invented at Xerox PARC
- ✓ Many systems now include both CLI and GUI interfaces
- ✓ Microsoft Windows is GUI with CLI "command" shell

- ✓ Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shellsavailable
- ✓ Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

6. System Calls and its types

- ✓ **System calls** provide an interface to the services made available by an operatingsystem. These calls are generally available as routines written in C and C++.
- ✓ System calls occur in different ways, depending on the computer in useOften. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read.
- ✓ Three general methods are used to pass parameters to the operating system.

a. The simplest approach is to pass the parameters in *registers.*
b. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a *block,* or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.4). This is the approach taken by Linux and Solaris. Parameters also can be placed, or *pushed*onto the *stack* by the program and *popped* off the stack by the operating system.
c. Some operating systems prefer the block or stack method, because thoseapproaches do not limit the number or length of parameters being passed.

Types of System Calls

- ✓ System calls can be grouped roughly into five major categories: **process control, file manipulation, device manipulation, information maintenance,** and **communications.**
- ✓ A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap.

The types of system calls normally provided by an operating system.

- **Process control**
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- **File management**
  - create file, delete file

    - open, close
    - read, write, reposition
    - get file attributes, set file attributes

- **Device management**
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
- **Information maintenance**
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes
- **Communication**
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices.

7. Operating Systems Generation

- ✓ Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
    - SYSGEN program obtains information concerning the specific configuration of the hardware system
    - *Booting* – starting a computer by loading the kernel
    - *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

System Boot

- Operating system must be made available to hardware so hardware can start it
- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location
    - Firmware used to hold initial boot code

UNIT – 2

PROCESS MANAGEMENT

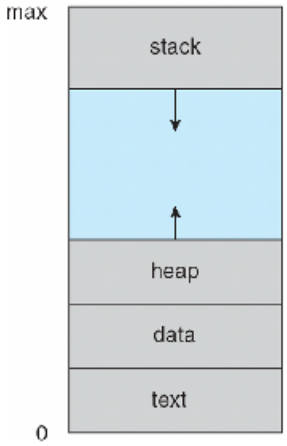
Process Concept

- 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

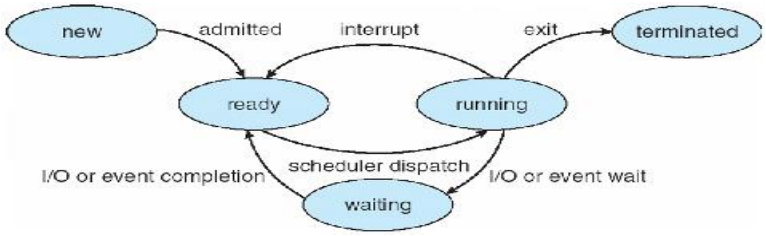


Process State

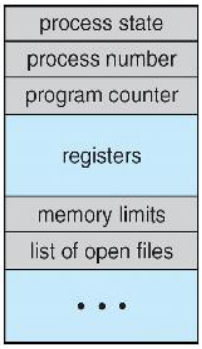
As a process executes, it changes *stat*e

- **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



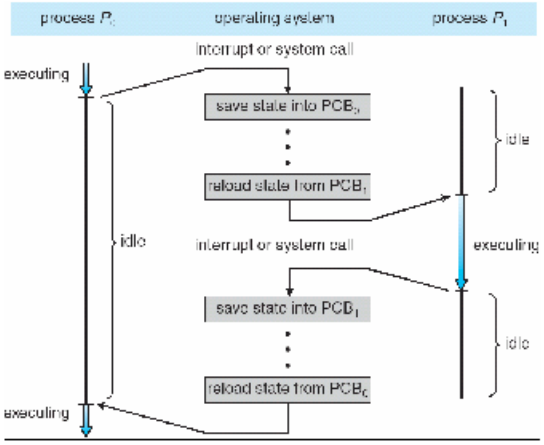
Process Control Block (PCB)



Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

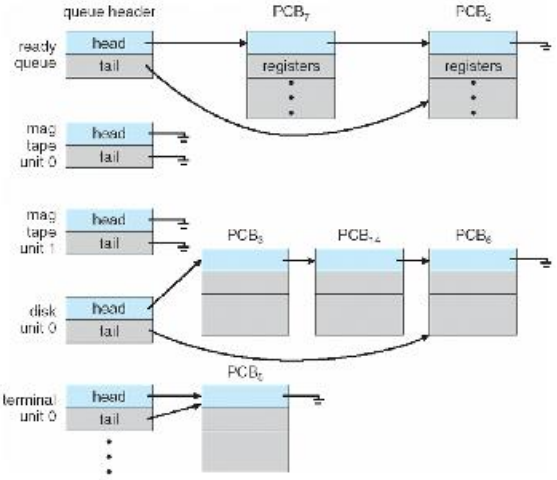
CPU Switch from Process to Process



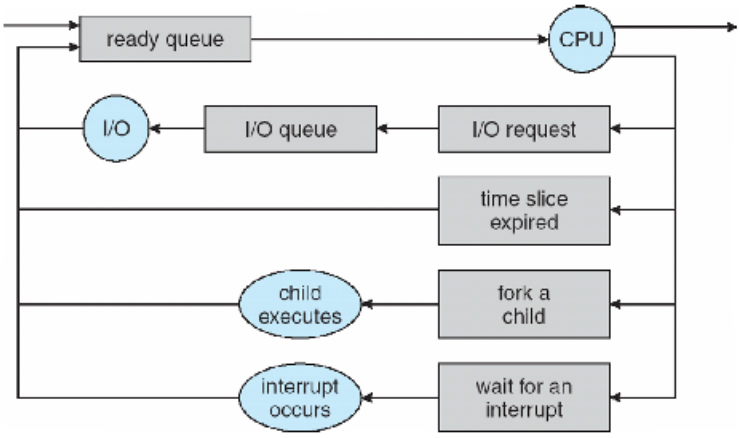
Process Scheduling Queues

- **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 ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be
- executed next and allocates CPU

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 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
- 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

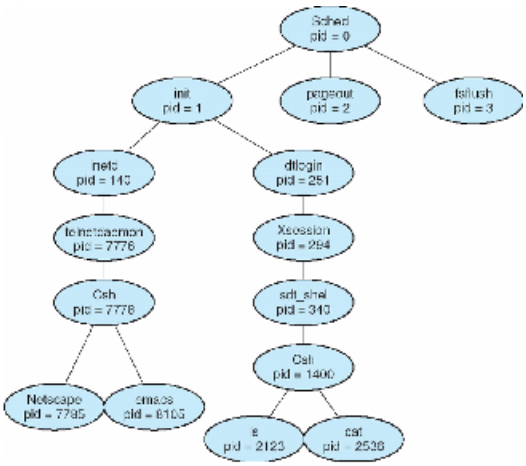
Process Creation

C Program Forking Separate Process

```
int main()
{
pid_tpid;
/* 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

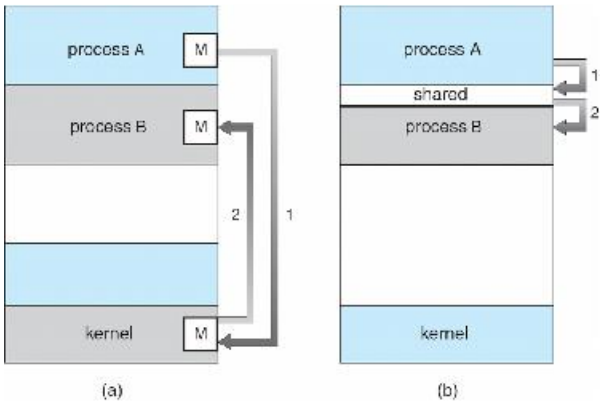


Process Termination

- 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;
}
```

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)
- **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
- 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
- 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 - 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

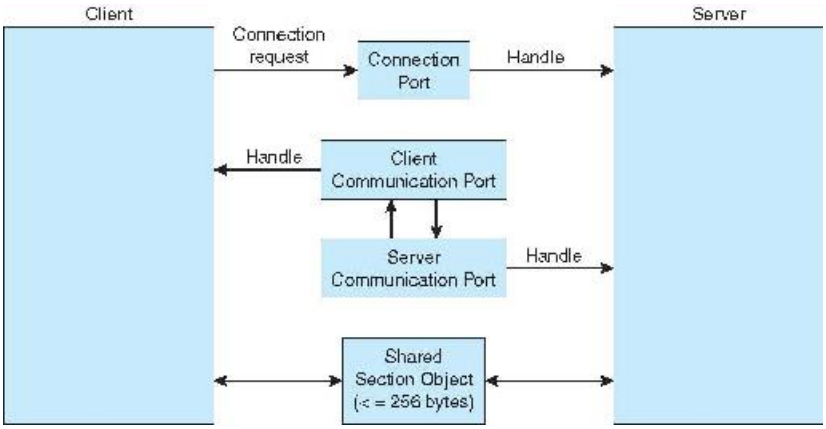
- Mach communication is message based
- Even system calls are messages
- Each task gets two mailboxes at creation- Kernel and Notify
- Only three system calls needed for message transfer
- msg_send(), msg_receive(), msg_rpc()
- Mailboxes needed for commuication, created via
- port_allocate()

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 ofthem to the client
- The client and server use the corresponding port handle to send messages or
- callbacks and to listen for replies

Local Procedure Calls in Windows XP



Communications in Client-Server Systems

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

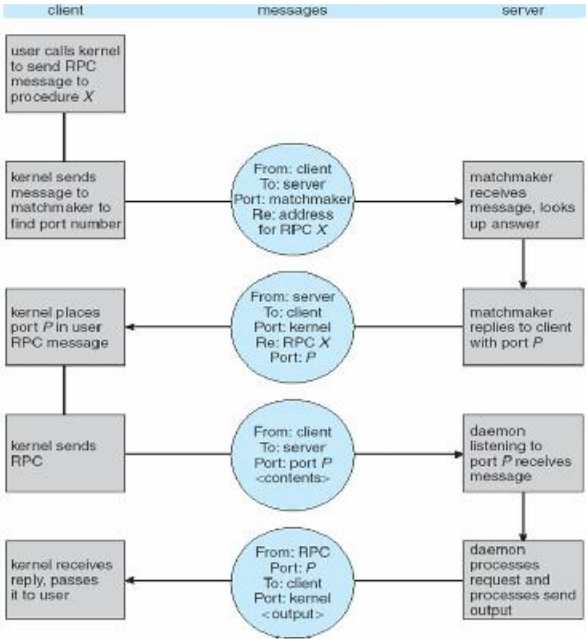
Sockets

- 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

Remote Procedure Calls

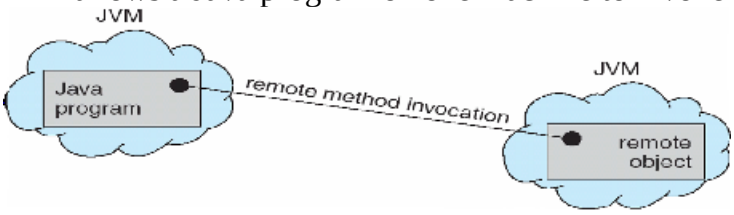
- Remote procedure call (RPC) abstracts procedure calls between processes on
- networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *Marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled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

Execution of RPC

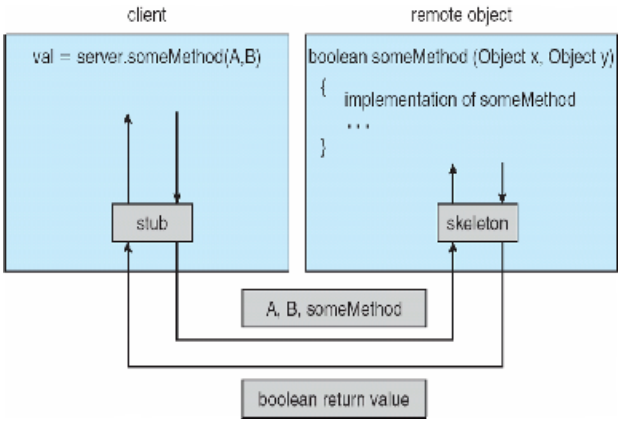


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



Marshalling Parameters

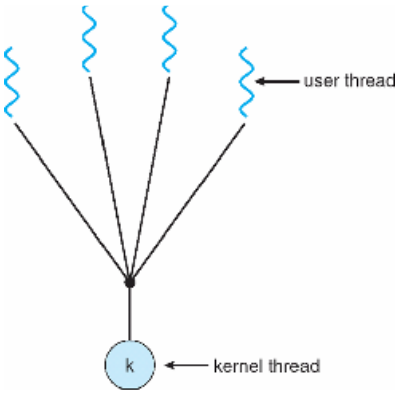


Multithreaded Programming

Multithreading Models

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

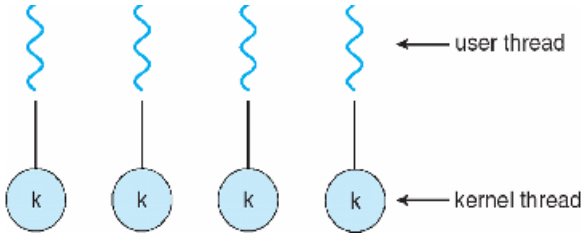
Many-to-One



Many user-level threads mapped to single kernel thread
Examples:

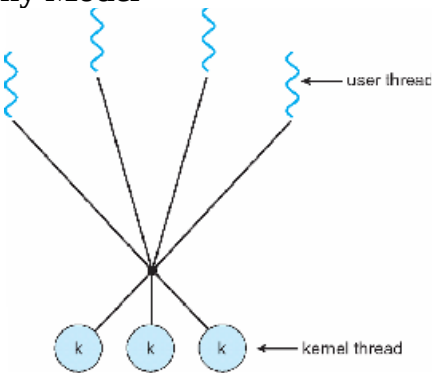
- Solaris Green Threads
- GNU Portable Threads

One-to-One



Each user-level thread maps to kernel thread
Examples
Windows NT/XP/2000
Linux
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

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 beCancelled

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

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

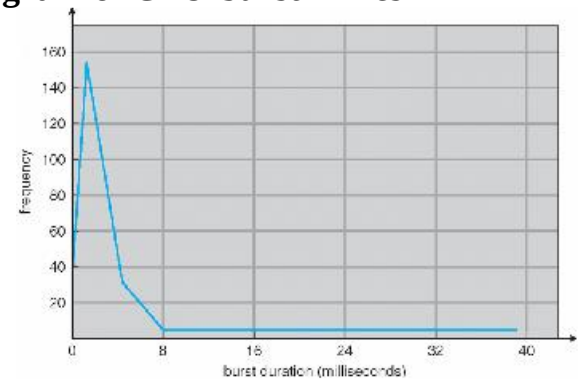
- Both M:M and Two-level models require communication to maintain the appropriate
- number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to
- the thread library
- This communication allows an application to maintain the correct number kernel
- Threads

PROCESS SCHEDULING:

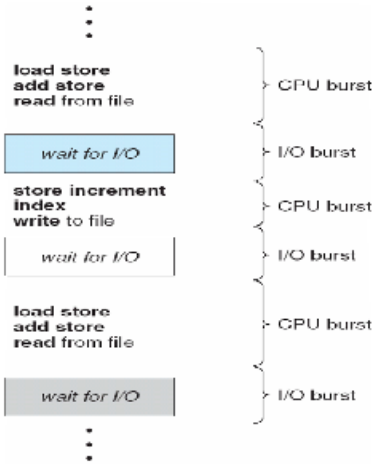
CPU Scheduling

- To introduce CPU scheduling, which is the basis for multiprogrammed operatingsystems
- 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/Owait
- **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)
- 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:

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

Suppose that the processes arrive in the orderP_2 ,P_3 , P_1
The Gantt chart for the schedule is:

Waiting time for P_1 = 6; P_2 = 0; P_3 = 3nAverage waitingtime: (6 + 0 + 3)/3 = 3
Much better than previous case

Convoy effect short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths toschedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

The difficulty is knowing

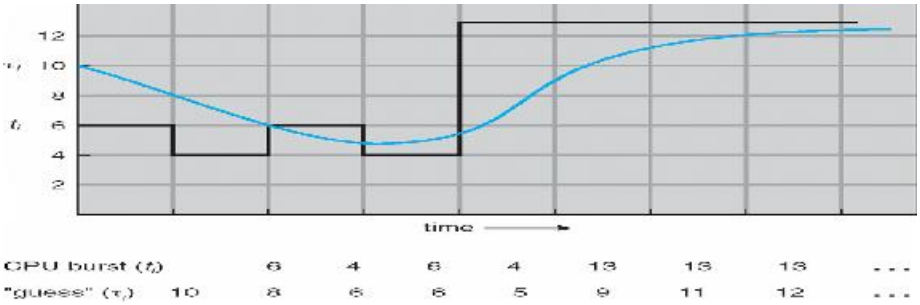
Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

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

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

Prediction of the Length of the Next CPU Burst



Examples of Exponential Averaging

a =0

$t_{n+1} = t_n$

Recent history does not count

a =1

$t_{n+1} = a\ t_n$

Only the actual last CPU burst counts

If we expand the formula, we get:

$t_{n+1} = a\ t_n + (1 - a) a t_{n-1} + \ldots$

$+ (1 - a)^j a\ t_{n-j} + \ldots$

$+ (1 - a)^{n+1} t_0$

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 highestpriority)
- 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 theend 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

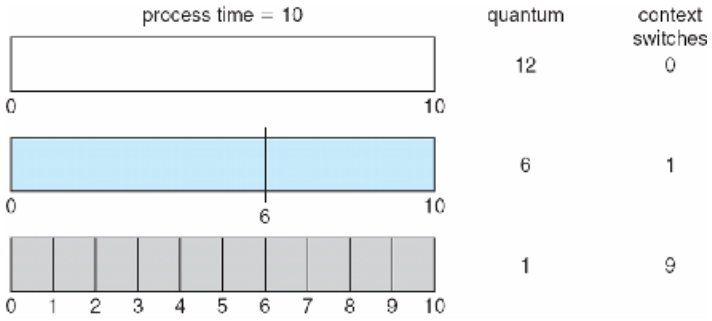
Example of RR with Time Quantum = 4

Process	Burst Time
P1	24
P2	3
P3	3

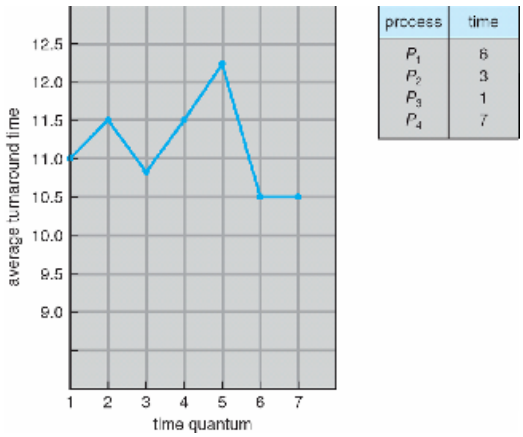
The Gantt chart is:

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

Time Quantum and Context Switch Time



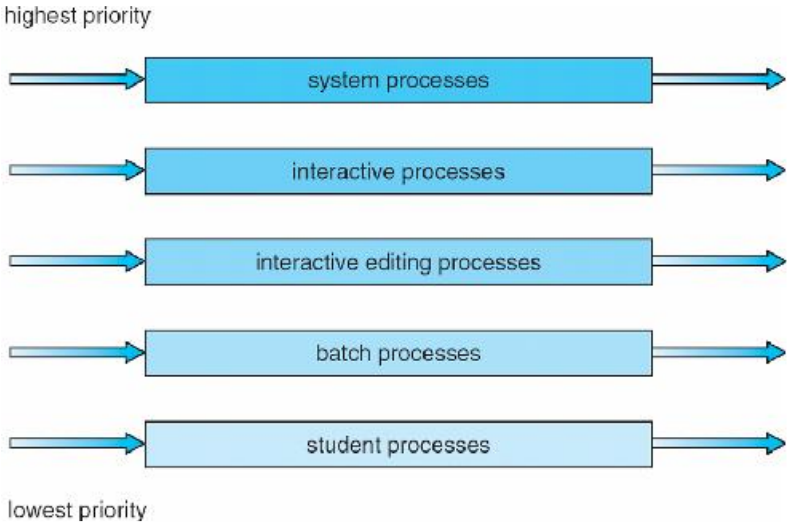
Turnaround Time Varies With The Time Quantum



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 and 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

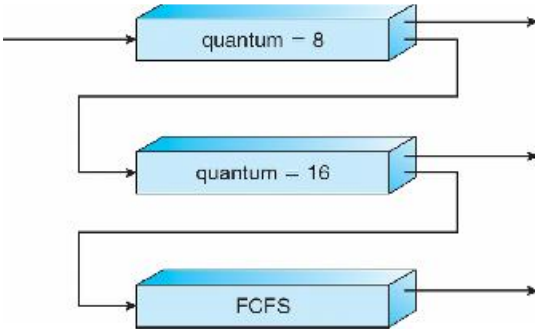
Three queues:

1. Q_0 – RR with time quantum 8 milliseconds
2. Q_1 – RR time quantum 16 milliseconds
3. 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



Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads torun on LWP
- Known as **process-contention scope (PCS)** since scheduling competition is within theprocess
- 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

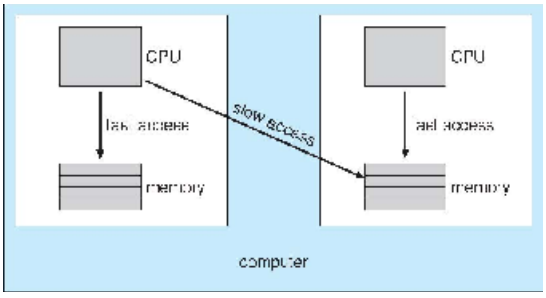
```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
int i; pthread t tid[NUM THREADS];
pthreadattr t attr;
/* get the default attributes */
pthreadattrinit(&attr);
/* set the scheduling algorithm to PROCESS or SYSTEM */
```

```
pthreadattrsetscope(&attr, PTHREAD SCOPE SYSTEM);
/* set the scheduling policy - FIFO, RT, or OTHER */
pthreadattrsetschedpolicy(&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");
pthreadexit(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 datastructures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, allprocesses in common ready queue, or each has its own private queue of readyprocesses
- **Processor affinity** – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

NUMA and CPU Scheduling

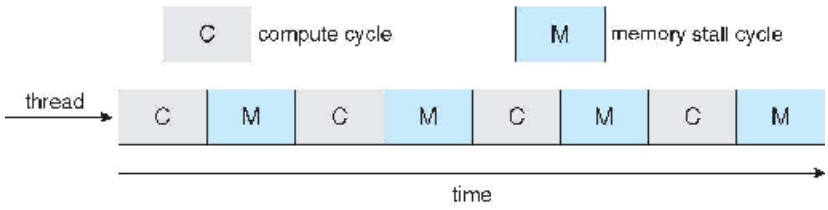


Multicore Processors

- 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



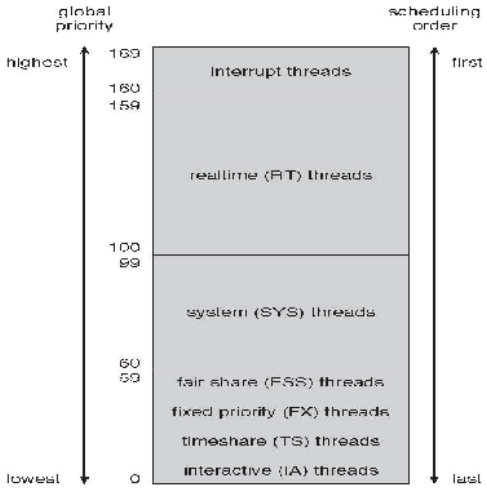
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling



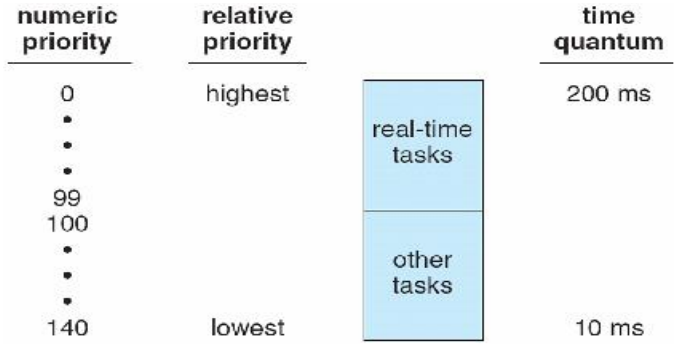
Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

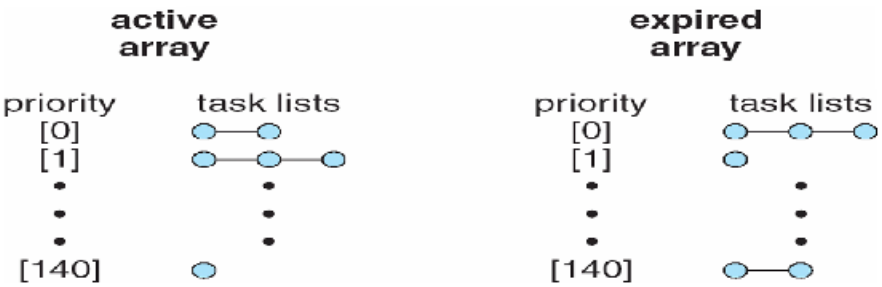
Linux Scheduling

- 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

Priorities and Time-slice length



List of tasks Indexed according to priorities



Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines theperformance of each algorithm for that workload
- Queueing modelsImplementation

Evaluation of CPU schedulers by Simulation