LECTURE NOTES

ON

OPERATING SYSTEMS

III B. Tech I semester (JNTUH-R13)

UNIT – I

Introduction to Operating System

Operating System :

A program that acts as an intermediary between a user of a computer and the computer hardware Operating system goals:

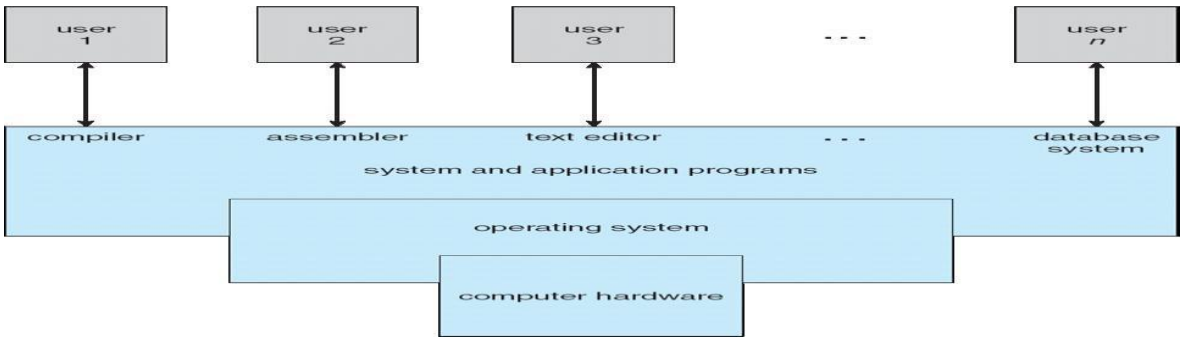
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure :

Computer system can be divided into four components

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

Four Components of a Computer System



Operating System Definition

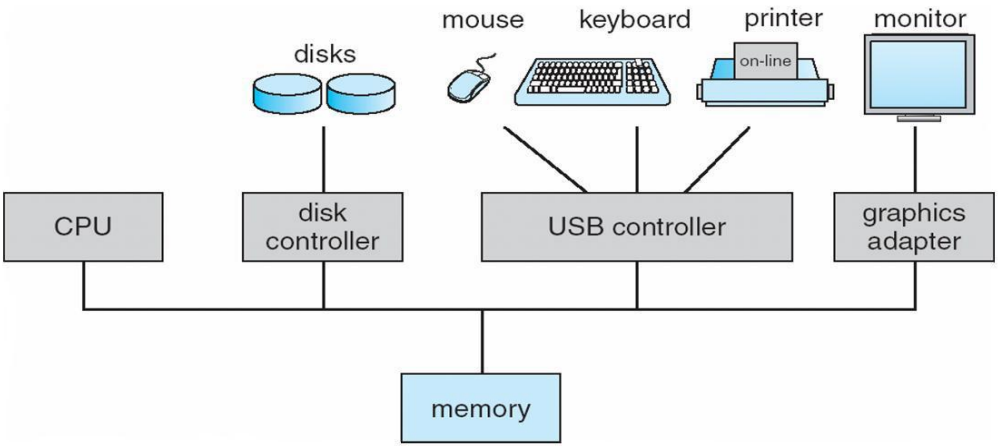
- OS is a **resource allocator**
- Manages all resources
- Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
- Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition
- Everything a vendor ships when you order an operating system" is good approximation
  But varies wildly
- "The one program running at all times on the computer" is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program

Computer Startup

- **bootstrap program** is loaded at power-up or reboot
- Typically stored in ROM or EPROM, generally known as **firmware**
- Initializes all aspects of system
- Loads operating system kernel and starts execution

Computer System Organization

- Computer-system operation
- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory cycles



Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing An *interrupt*

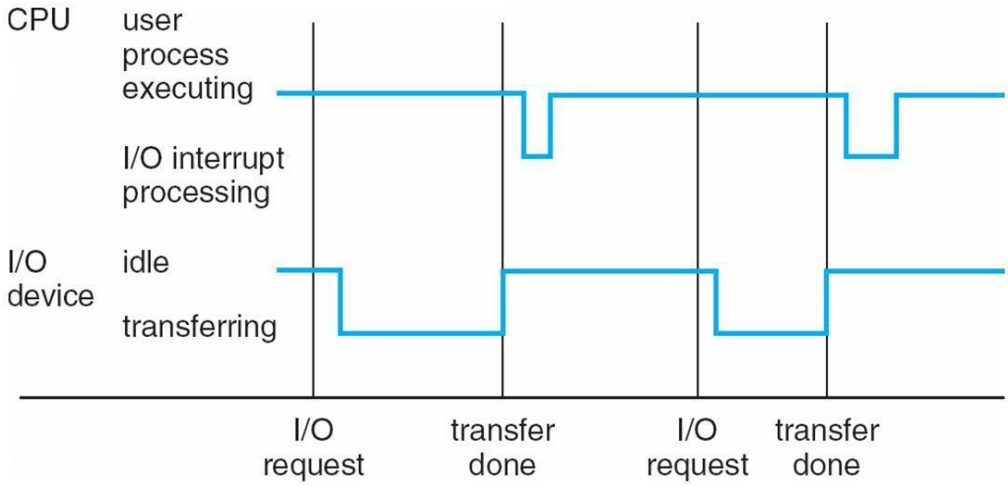
Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*nA *trap* is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
- **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Timeline



I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
- Wait instruction idles the CPU until the next interrupt
- Wait loop (contention for memory access)
- At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
- **System call** – request to the operating system to allow user to wait for I/O completion
- **Device-status table** contains entry for each I/O device indicating its type, address, and **state**
- Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte.

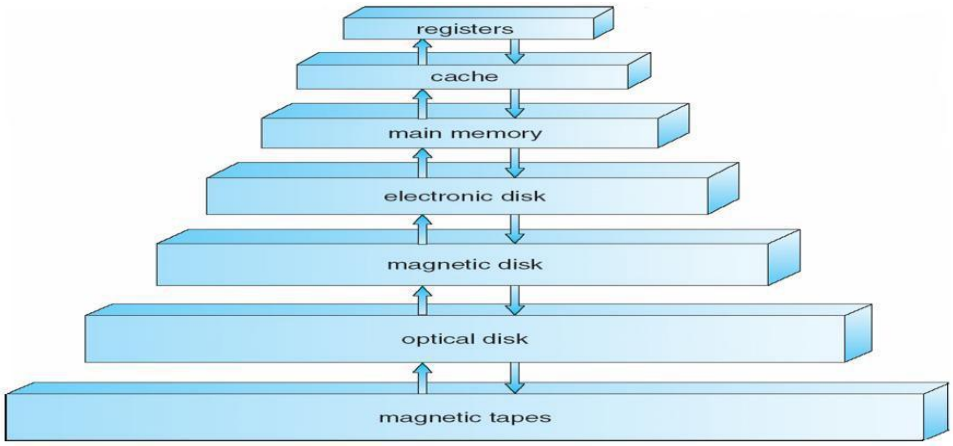
Storage Structure

- Main memory – only large storage media that the CPU can access directly
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
- Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
- The **disk controller** determines the logical interaction between the device and the computer

Storage Hierarchy

- Storage systems organized in hierarchy
- Speed
- Cost
- Volatility

Caching – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage



Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
- If it is, information used directly from the cache (fast)
- If not, data copied to cache and used there
- Cache smaller than storage being cached
- Cache management important design problem
- Cache size and replacement policy

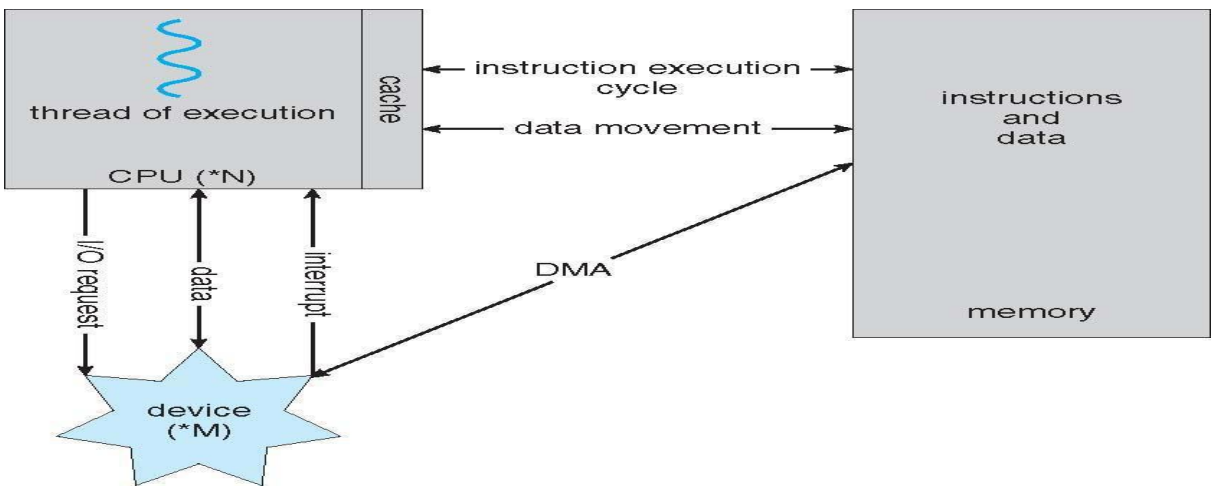
Computer-System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
- Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
- Also known as parallel systems, tightly-coupled systems

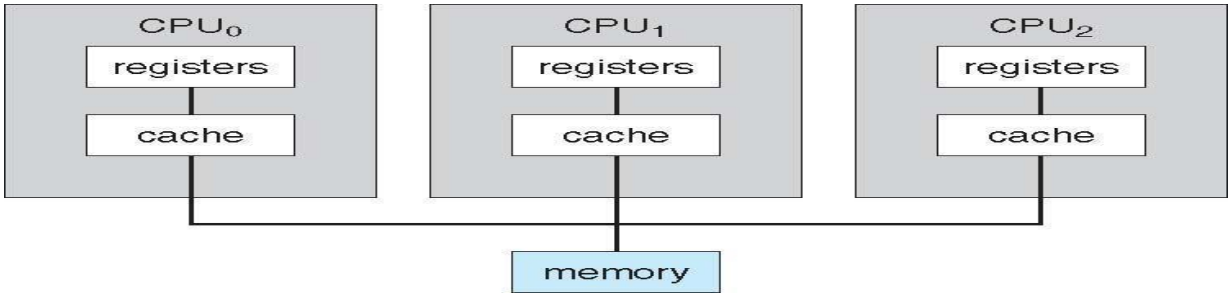
Advantages include

1.Increased throughput
2.Economy of scale
3.Increased reliability – graceful degradation or fault tolerance Two types

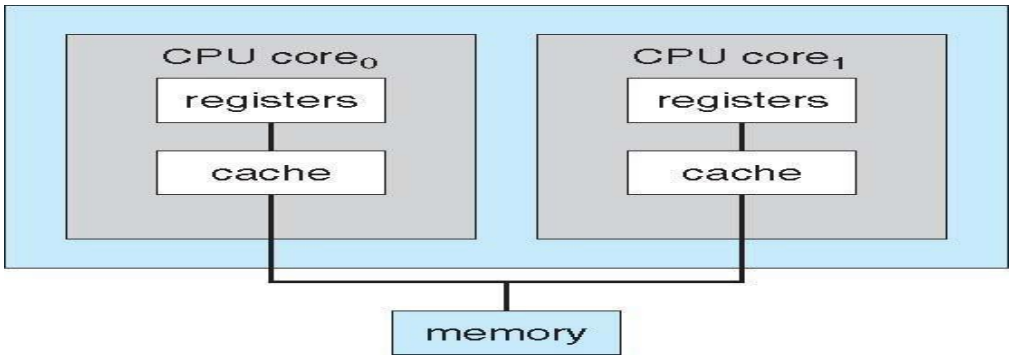
1.Asymmetric Multiprocessing
2.Symmetric Multiprocessing



How a Modern Computer Works Symmetric Multiprocessing Architecture



A Dual-Core Design



Clustered Systems

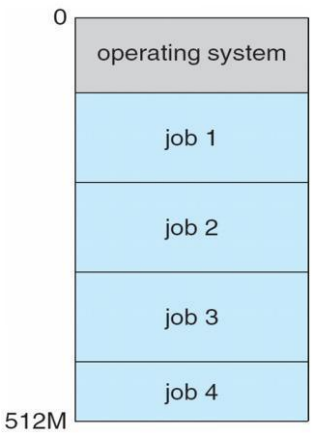
- Like multiprocessor systems, but multiple systems working together
- Usually sharing storage via a storage-area network (SAN)
- Provides a high-availability service which survives failures
  Asymmetric clustering has one machine in hot-standby mode
  Symmetric clustering has multiple nodes running applications, monitoring each other
- Some clusters are for high-performance computing (HPC) Applications must be written to use parallelization

Operating System Structure

- **Multiprogramming** needed for efficiency
- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to Execute
- A subset of total jobs in system is kept in memory
- One job selected and run via **job scheduling**
- When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
- **Response time** should be < 1 second
- Each user has at least one program executing in memory [**process**
- If several jobs ready to run at the same time [**CPU scheduling**
- If processes don't fit in memory, **swapping** moves them in and out to run

Virtual memory allows execution of processes not completely in memory.

Memory Layout for Multiprogrammed System



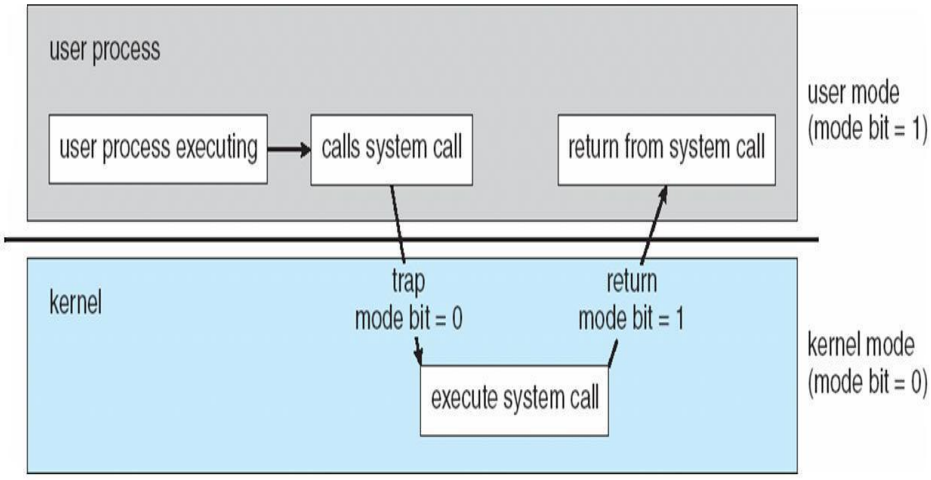
Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
- Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each Other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
- **User mode** and **kernel mode**
- **Mode bit** provided by hardware

- Provides ability to distinguish when system is running user code or kernel code
- Some instructions designated as **privileged**, only executable in kernel mode
- System call changes mode to kernel, return from call resets it to user
-

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
- Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



OPERATING SYSTEM FUNCTIONS

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
- CPU, memory, I/O, files
- Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
- Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
- Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

- The operating system is responsible for the following activities in connection with process management:
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
- Optimizing CPU utilization and computer response to users
- **Memory management activities**
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
- Abstracts physical properties to logical storage unit - **file**
- Each medium is controlled by device (i.e., disk drive, tape drive)
- Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
- Files usually organized into directories
- Access control on most systems to determine who can access what
- **OS activities include**
- Creating and deleting files and directories
- Primitives to manipulate files and dirs
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a "long" period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms

Mass-Storage Activities

- Free-space management
- Storage allocation
- Disk scheduling
- Some storage need not be fast
- Tertiary storage includes optical storage, magnetic tape
- Still must be managed
- Varies between WORM (write-once, read-many-times) and RW (read-write)

DISTRIBUTED SYSTEMS

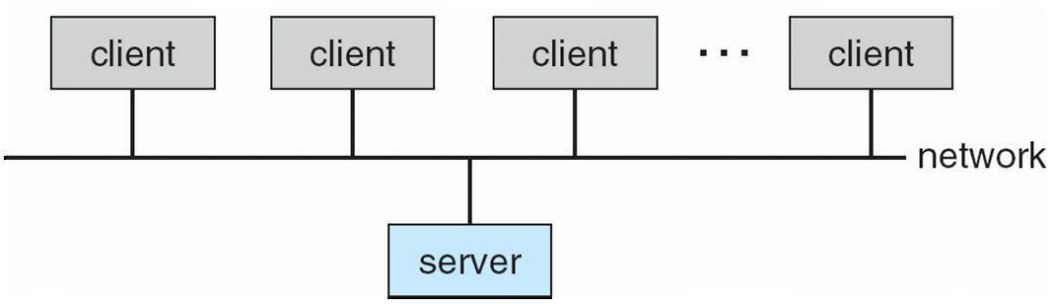
Computing Environments

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 same resources
- 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

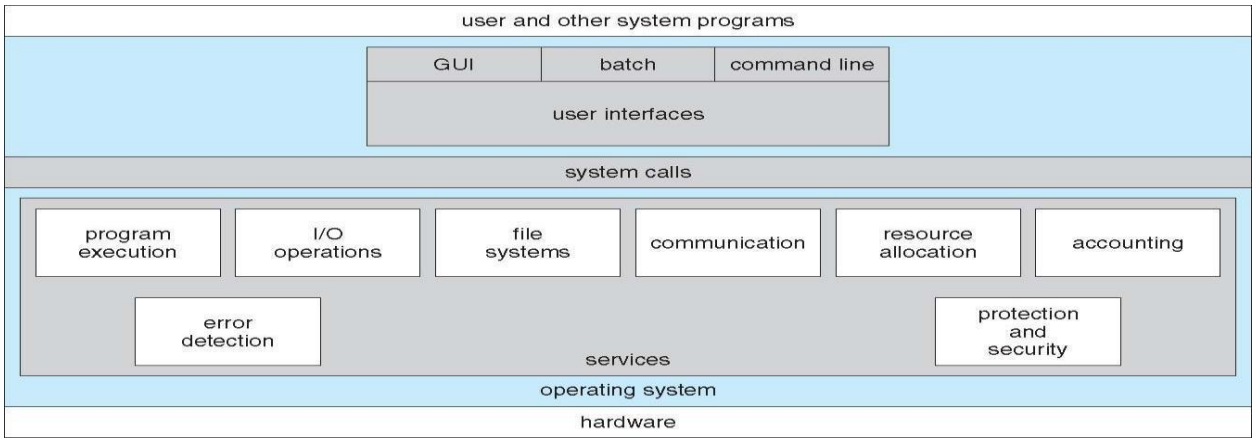
Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary closed-source
- 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 Sun Solaris

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.

A View of Operating System Services



Operating System Services

- One set of operating-system services provides functions that are helpful to the user (Cont):l
  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 defending external I/O devices from invalid access attempts
- If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

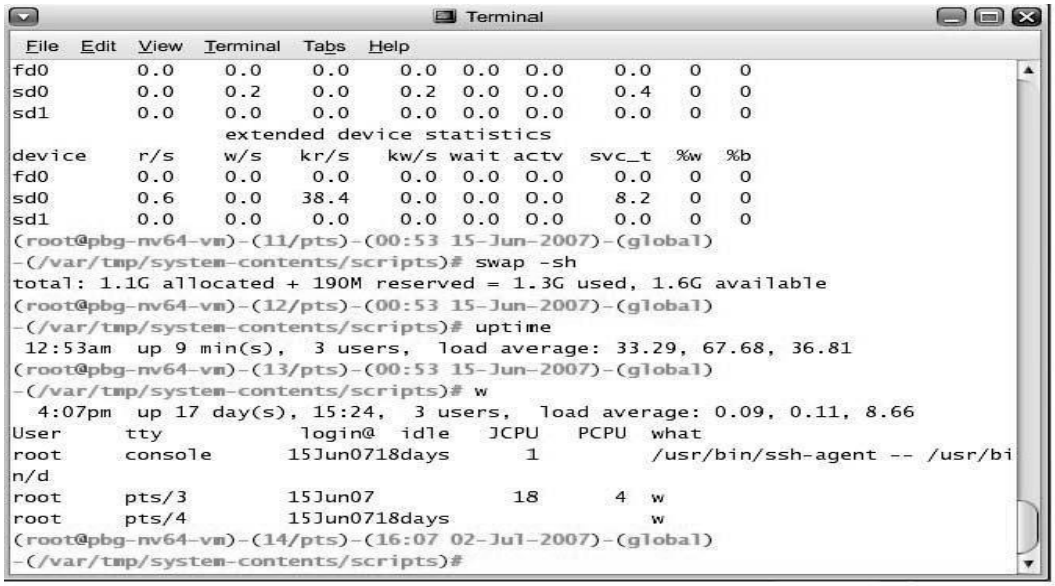
User Operating System Interface - CLI

- 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

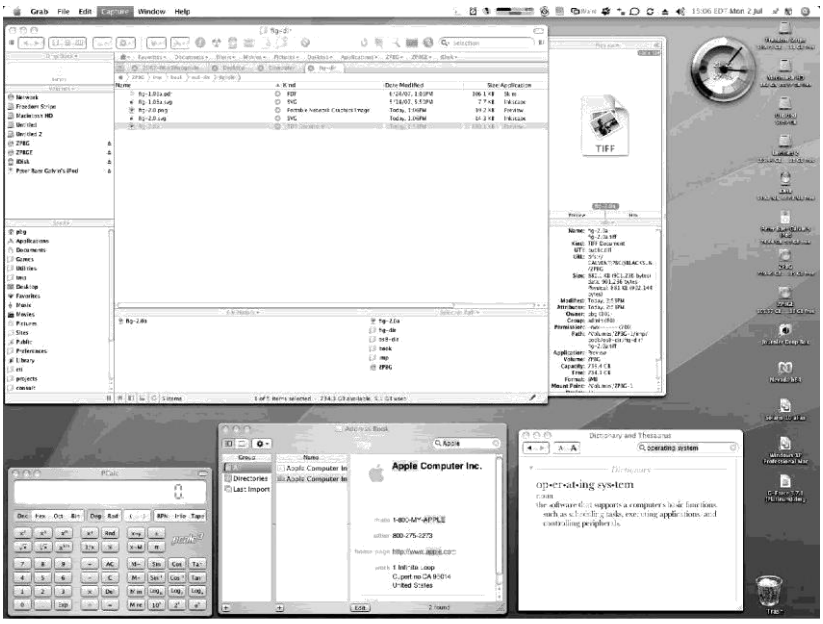
User Operating System Interface - GUI

- 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 shells available
- Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Bourne Shell Command Interpreter



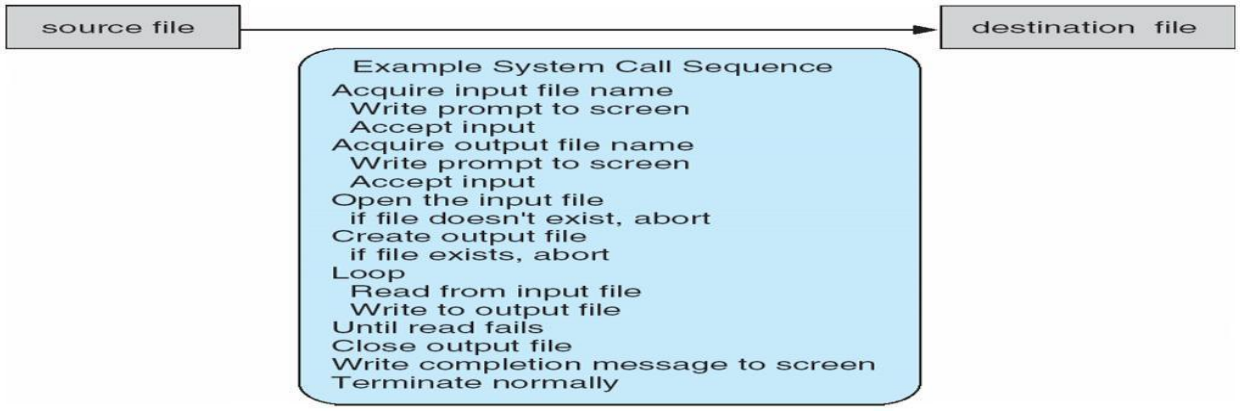
The Mac OS X GUI



System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call usen Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

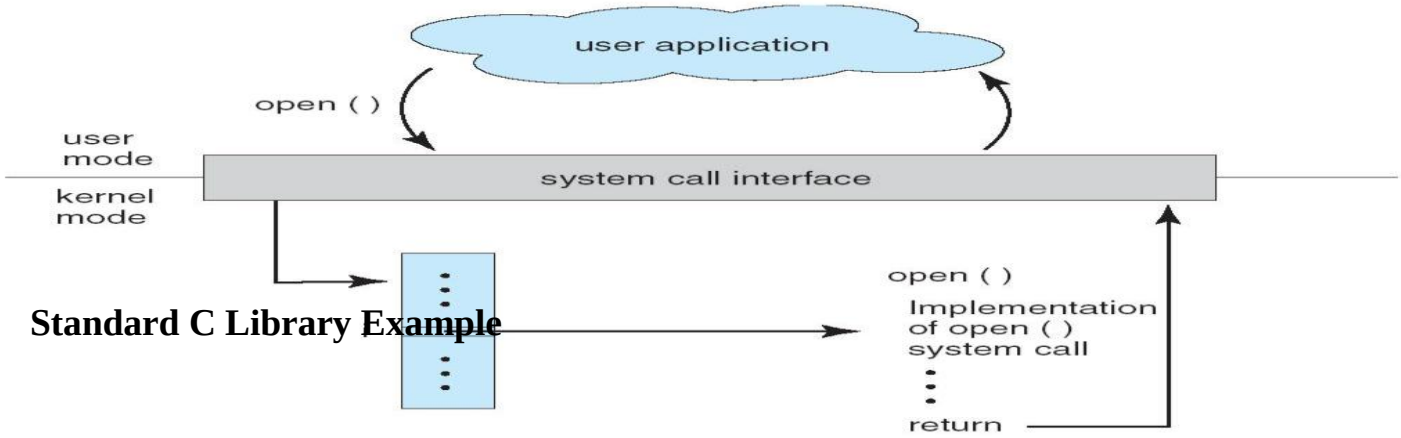
Example of System Calls



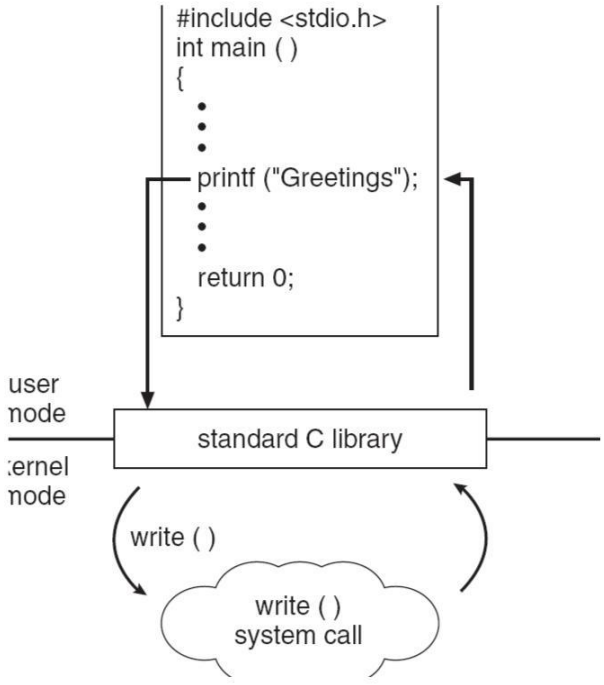
System Call Implementation

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

API – System Call – OS Relationship



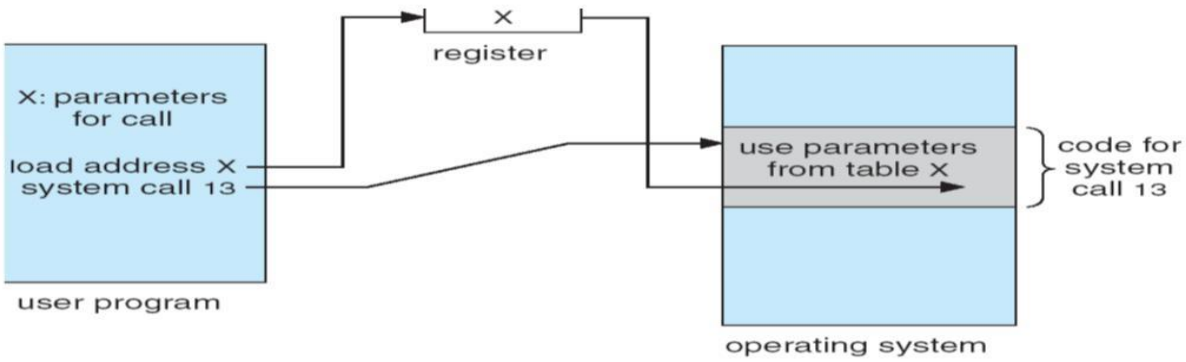
Standard C Library Example



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
- Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
- Simplest: pass the parameters in *registers*
  In some cases, may be more parameters than registers
- Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register . This approach taken by Linux and Solaris
- Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



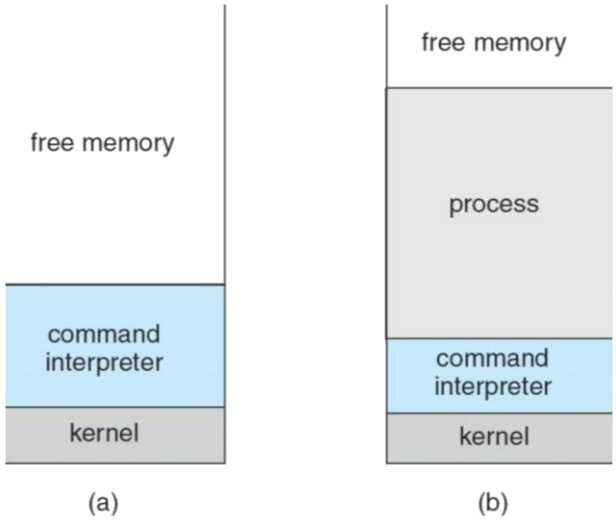
Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

Examples of Windows and Unix System Calls

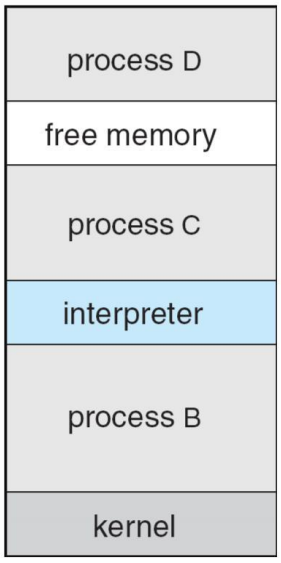
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

MS-DOS execution



(a) At system startup (b) running a program

FreeBSD Running Multiple Programs



System Programs

System programs provide a convenient environment for program development and execution. The can be divided into:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Application programs

Most users' view of the operation system is defined by system programs, not the actual system calls

- Provide a convenient environment for program development and execution
- Some of them are simply user interfaces to system calls; others are considerably more complex
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
- Some ask the system for info - date, time, amount of available memory, disk space, number of users
- Others provide detailed performance, logging, and debugging information
- Typically, these programs format and print the output to the terminal or other output devices
- Some systems implement a registry - used to store and retrieve configuration information

File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text
- Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
- Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems
- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

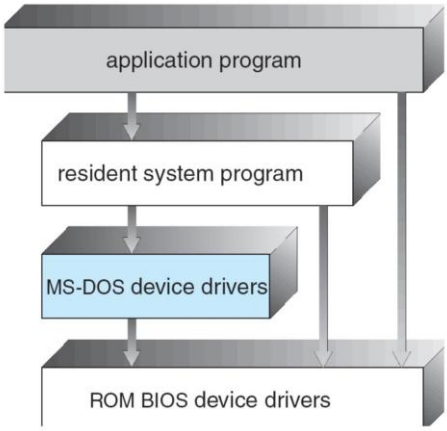
Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Important principle to separate
- Policy**:** What will be done?
  Mechanism: How to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

Simple Structure

- MS-DOS – written to provide the most functionality in the least space
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of Functionality are not well separated

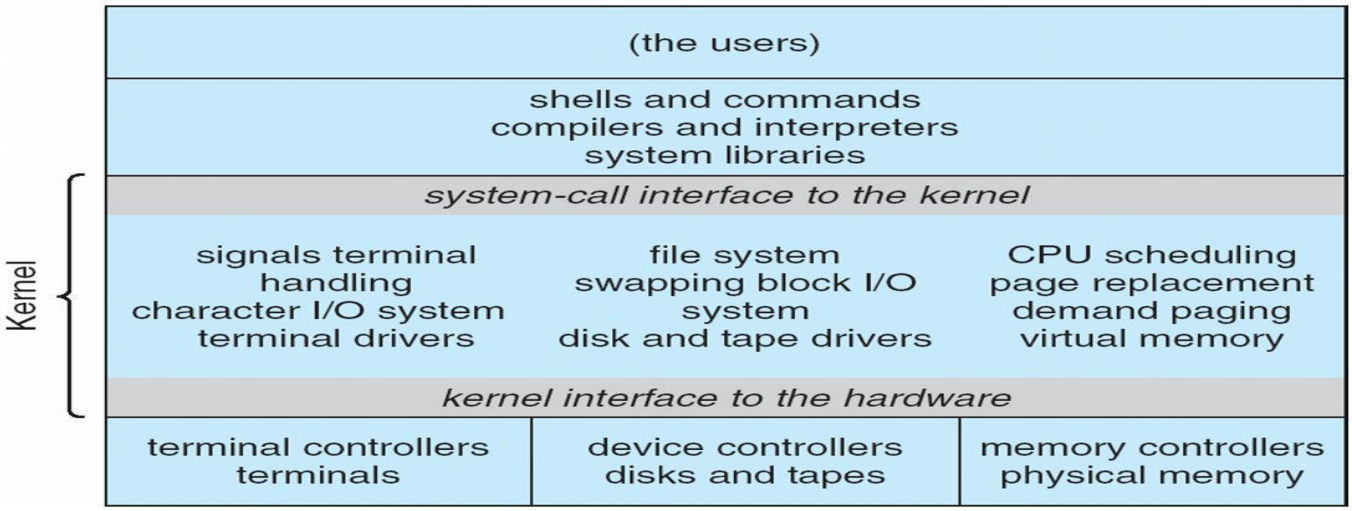
MS-DOS Layer Structure



Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layer

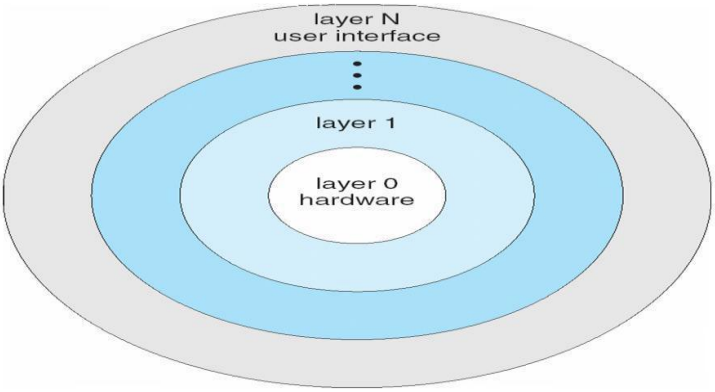
Traditional UNIX System Structure



UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
- Systems programs
- The kernel
  Consists of everything below the system-call interface and above the physical hardware
  Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

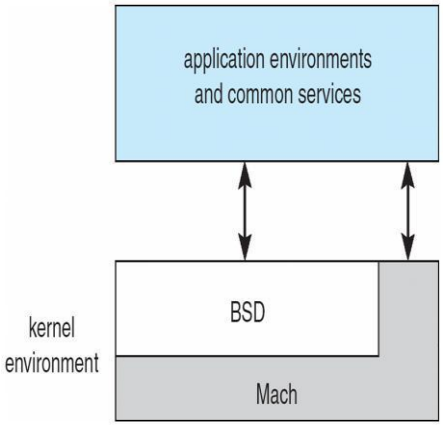
Layered Operating System



Micro kernel System Structure

- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure
- Detriments:
- Performance overhead of user space to kernel space communication

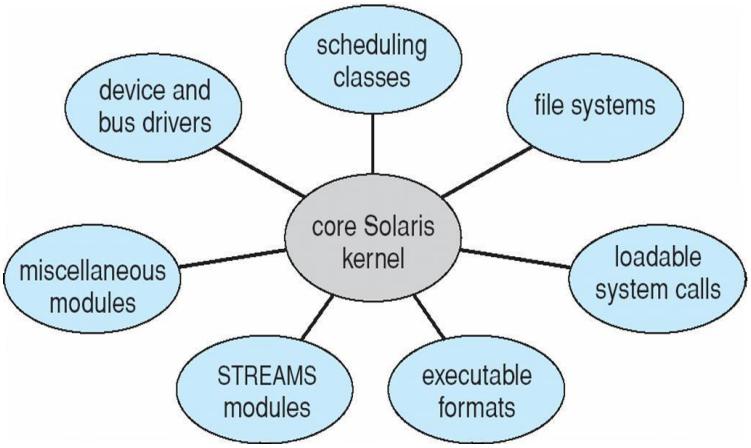
Mac OS X Structure



Modules

- Most modern operating systems implement kernel modules
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible.

Solaris Modular Approach

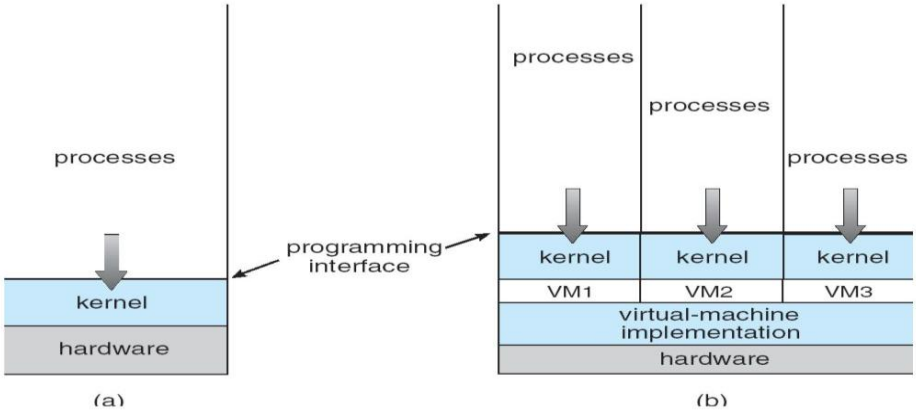


Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

Virtual Machines History and Benefits

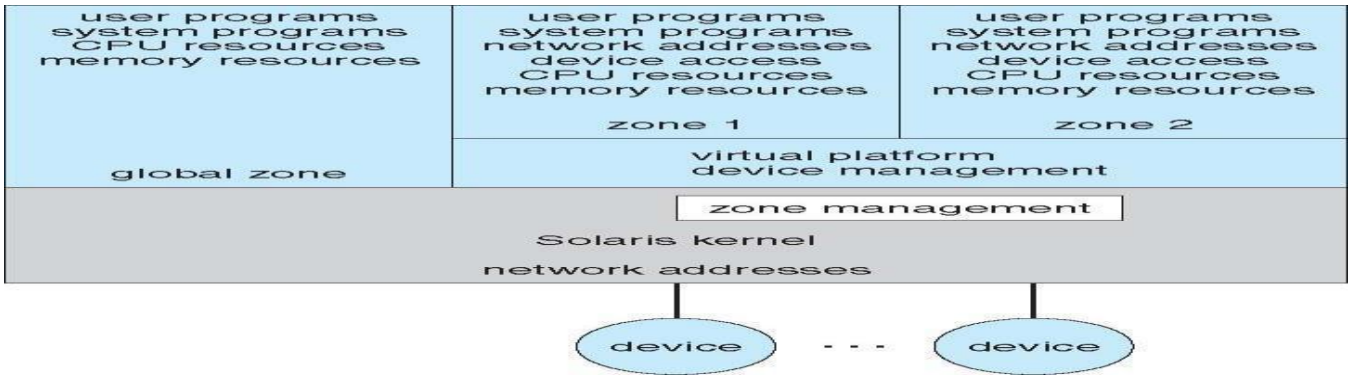
- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commutate with each other, other physical systems via networking
- Useful for development, testing
- Consolidation of many low-resource use systems onto fewer busier systems
- "Open Virtual Machine Format", standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms



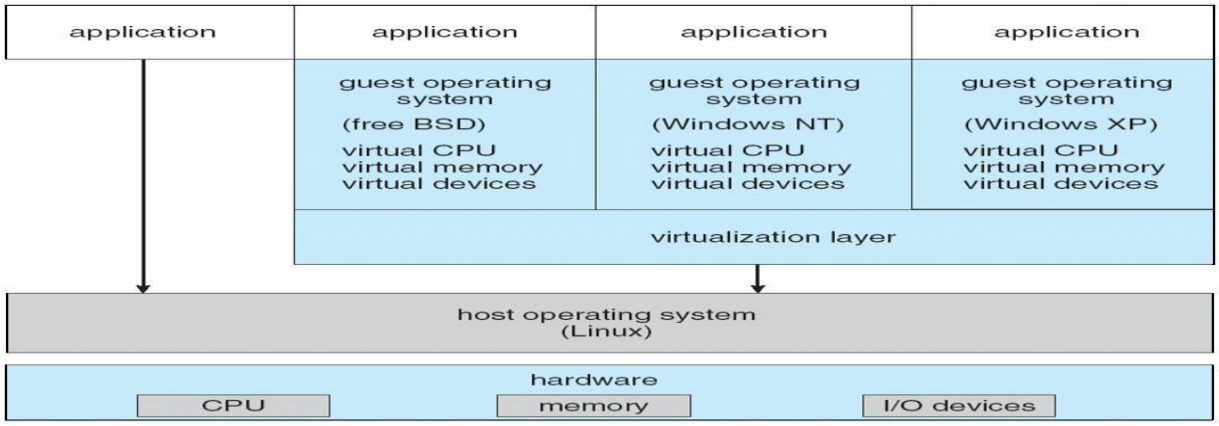
Para-virtualization

- Presents guest with system similar but not identical to hardware
- Guest must be modified to run on paravirtualized hardwareF
- Guest can be an OS, or in the case of Solaris 10 applications running in containers

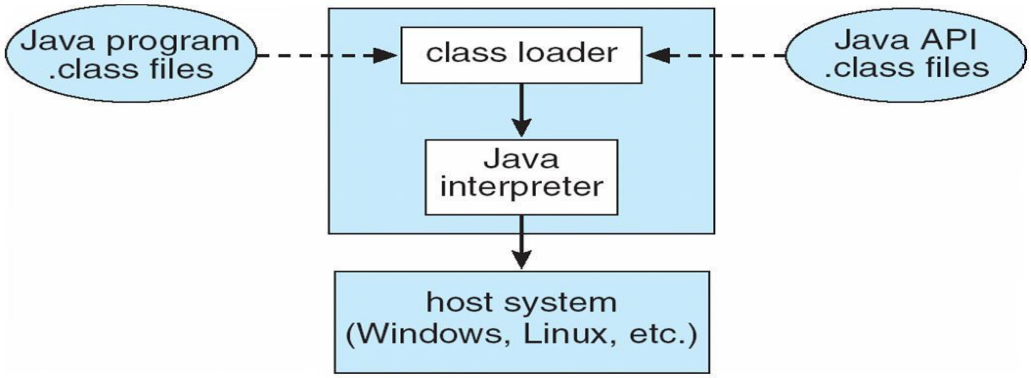
Solaris 10 with Two Container



VMware Architecture



The Java Virtual Machine



Operating-System Debugging

- Debugging is finding and fixing errors, or bugs
- OSes generate log files containing error information
- Failure of an application can generate core dump file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
- Kernighan's Law: "Debugging is twice as hard as writing the code in the rst place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- Probes fire when code is executed, capturing state data and sending it to consumers of those probes

Solaris 10 dtrace Following System Call

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                      U
  0   -> _XEventsQueued                   U
  0     -> _X11TransBytesReadable         U
  0     <- _X11TransBytesReadable         U
  0     -> _X11TransSocketBytesReadable   U
  0     <- _X11TransSocketBytesreadable   U
  0     -> ioctl                          U
  0       -> ioctl                        K
  0         -> getf                       K
  0           -> set_active_fd            K
  0           <- set_active_fd            K
  0         <- getf                       K
  0         -> get_udatamodel             K
  0         <- get_udatamodel             K
...
  0         -> releasef                   K
  0           -> clear_active_fd          K
  0           <- clear_active_fd          K
  0           -> cv_broadcast             K
  0           <- cv_broadcast             K
  0         <- releasef                   K
  0       <- ioctl                        K
  0     <- ioctl                          U
  0   <- _XEventsQueued                   U
  0 <- XEventsQueued                      U
```

Operating System 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 – II
Process Management

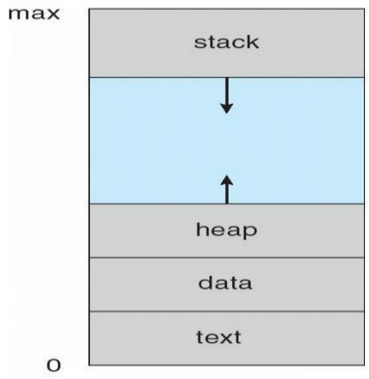
Process Concept

- 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

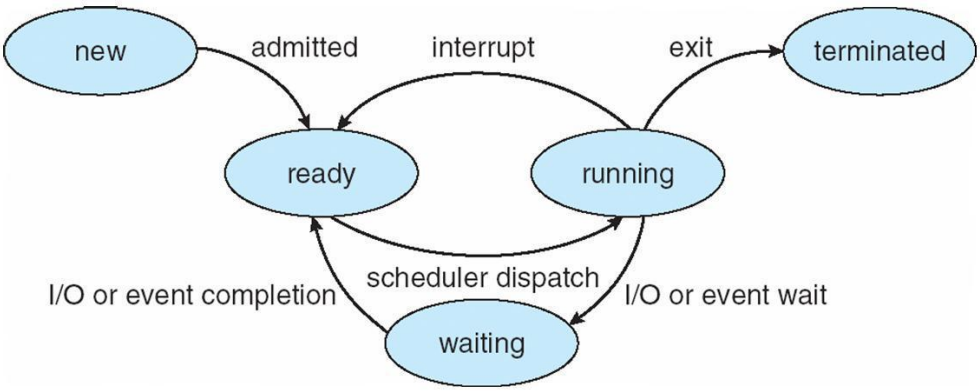


Process State

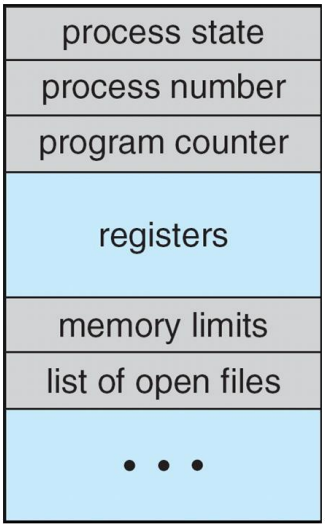
As a process executes, it changes *stat*e

- **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



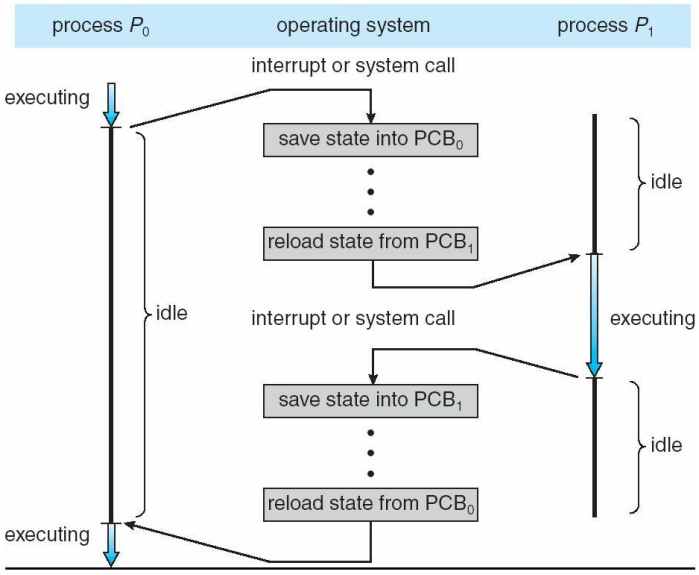
Process Control Block (PCB)



Information associated with each process

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

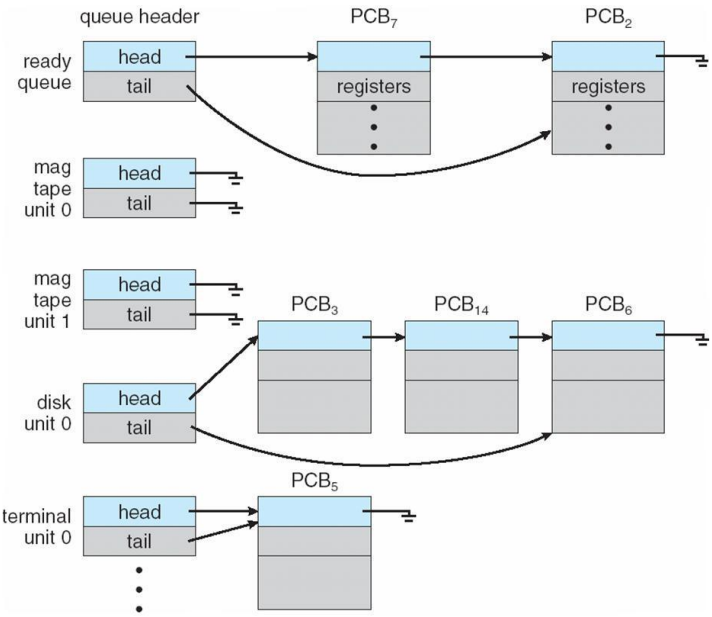
CPU Switch From Process to Process



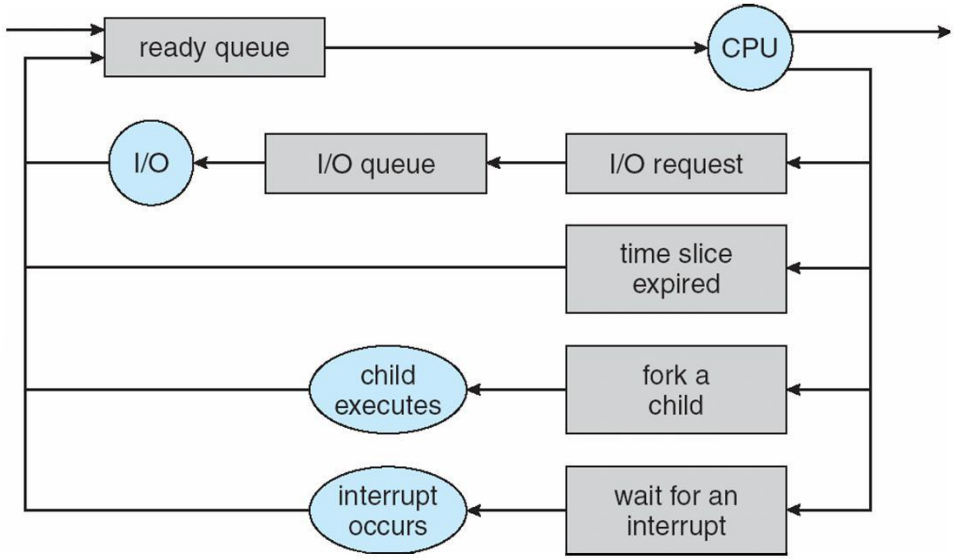
Process Scheduling Queues

- **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



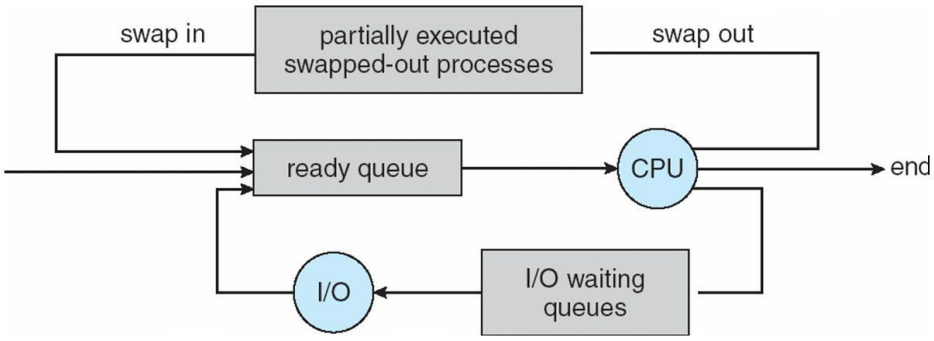
Representation of Process Scheduling



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

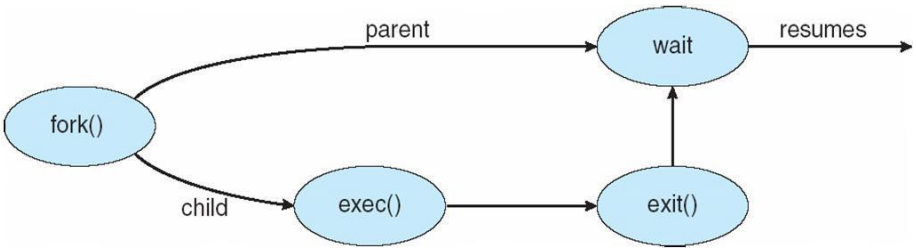
Context Switch

- 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



Process Termination

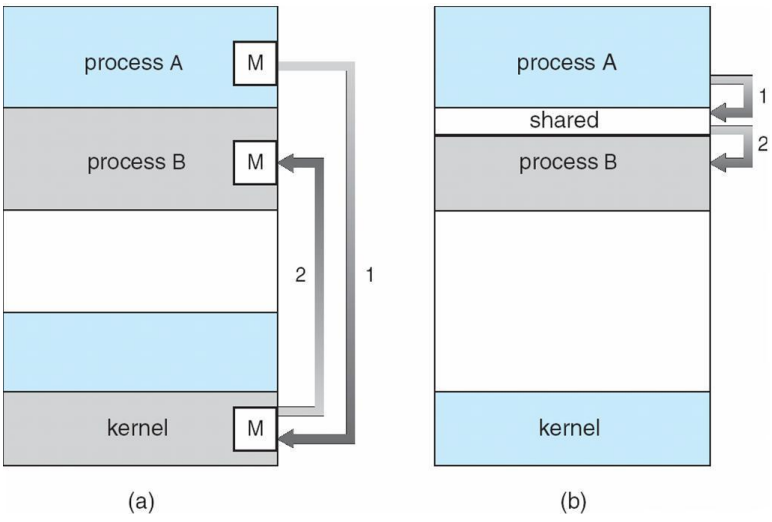
- Process executes last statement and asks the operating system to delete it (**exit**)
- Output data from child to parent (via **wait**)
- Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
- Child has exceeded allocated resources
- 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
- *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

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

Communications in Client-Server Systems

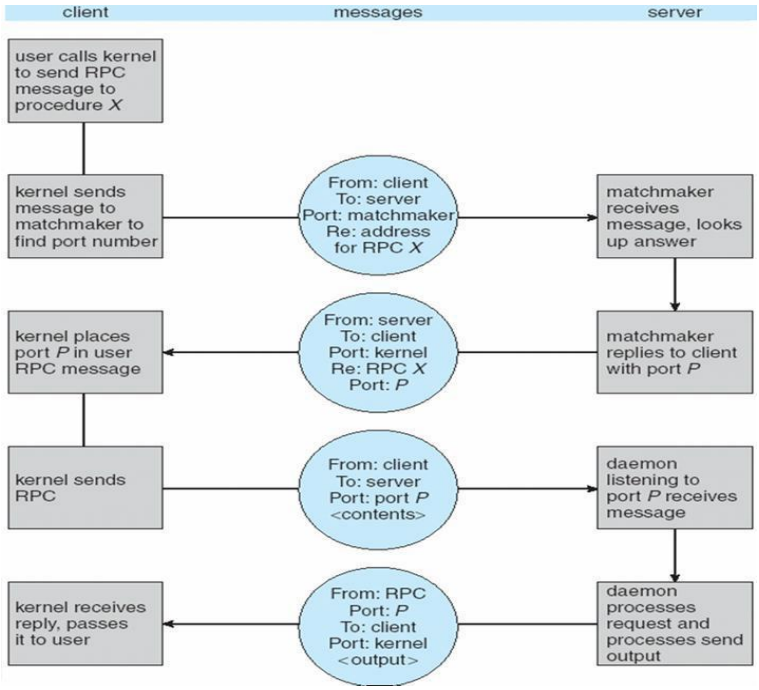
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

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

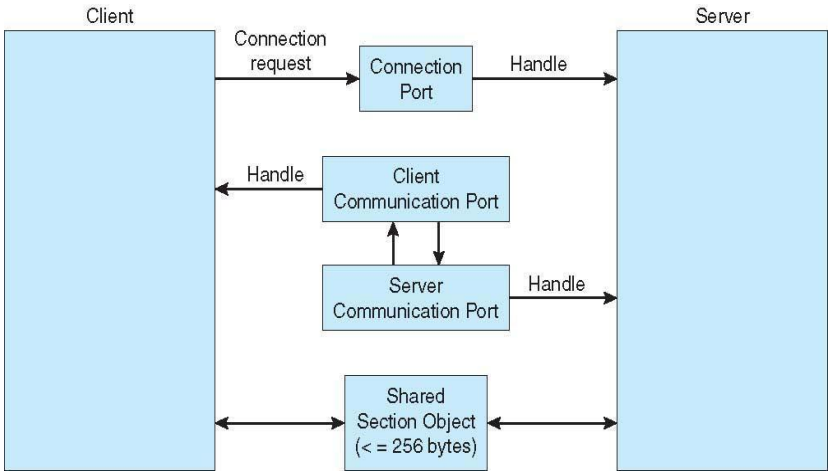
Execution of RPC

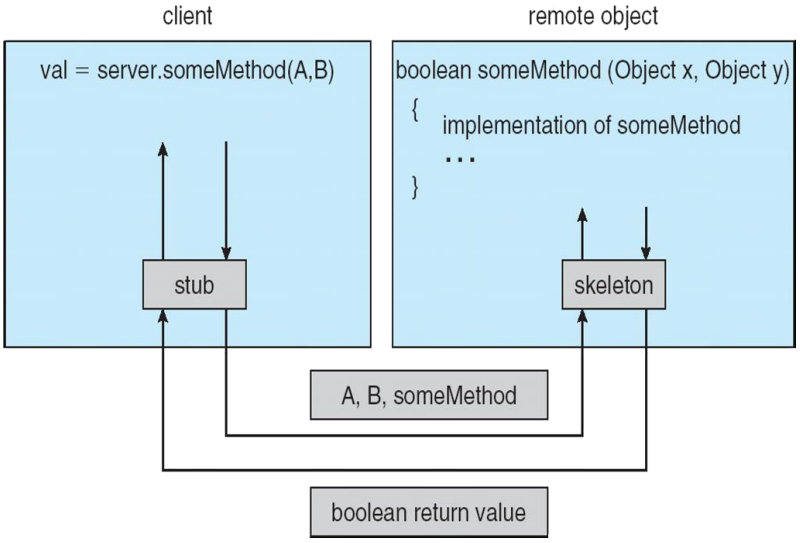


Remote Method Invocation

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

Marshalling Parameters





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

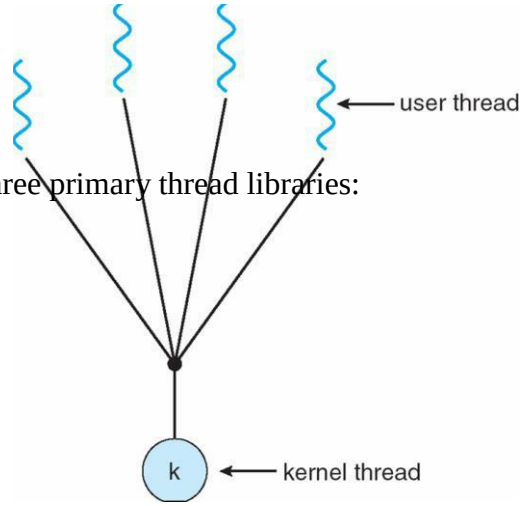
Benefits

- 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
-



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

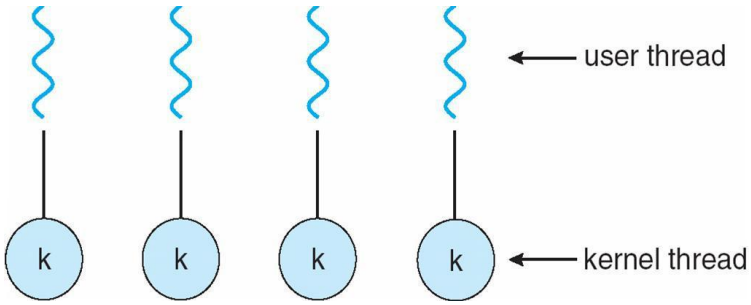
Multithreading Models

- 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



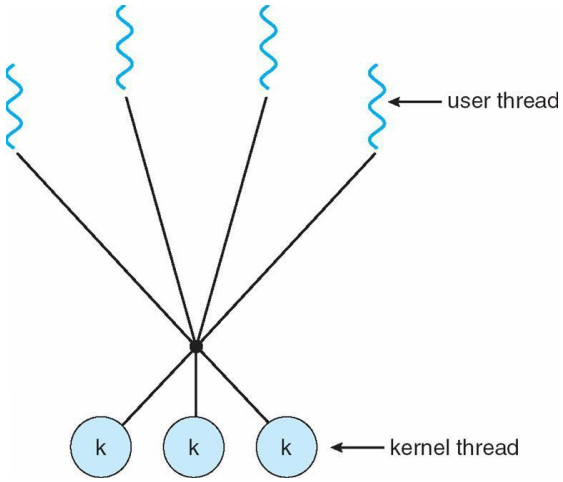
One-to-One

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

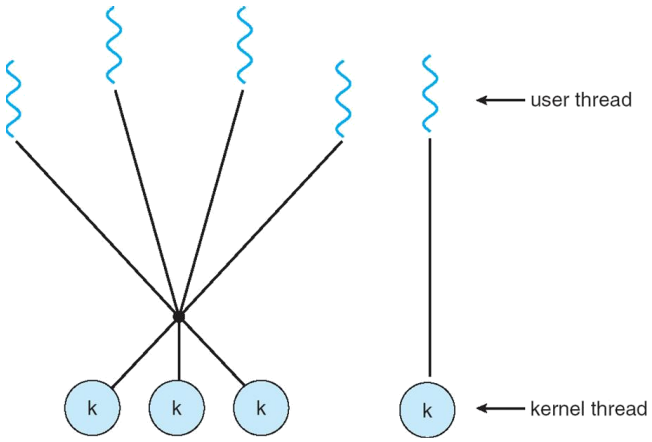
Windows NT/2000 with the *ThreadFiber* package



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

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

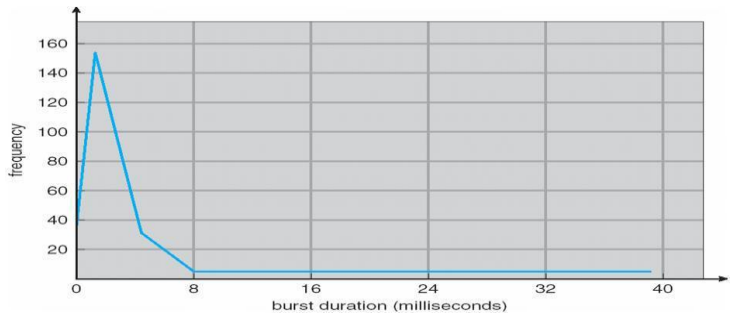
Thread Cancellation

- 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

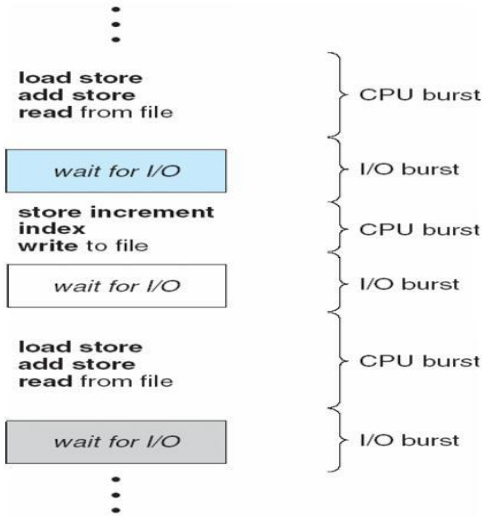
CPU Scheduling

- 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 **non preemptive**

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

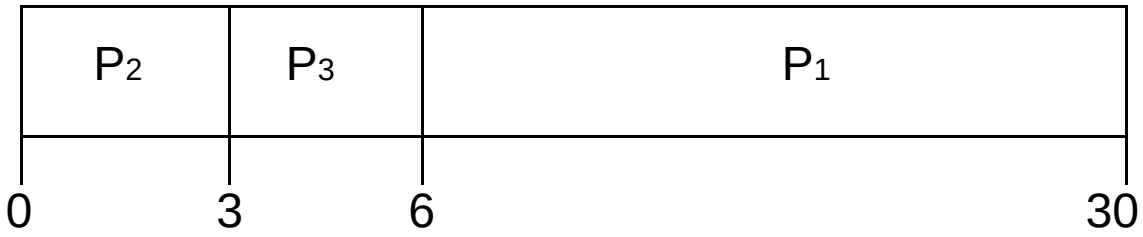
Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1* , *P2* , *P3*
The Gantt Chart for the schedule is:

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:nnnn Waiting 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



Shortest-Job-First (SJF) Scheduling

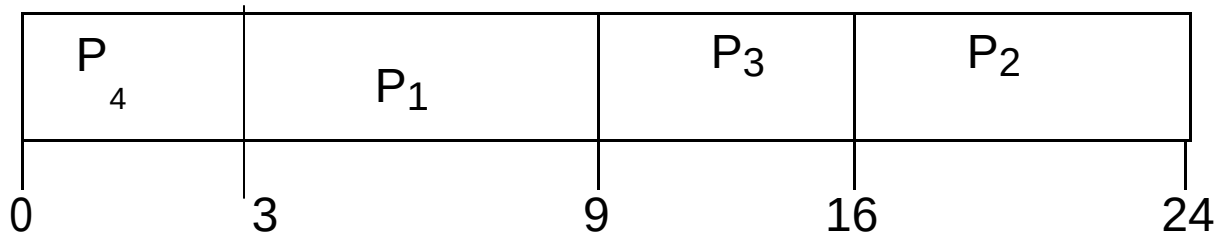
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

The difficulty is knowing

Process	Arrival Time	Burst Time
P1	0.0	0.0
P2	2.0	2.0
P3	4.0	4.0
P4	5.0	5.0

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

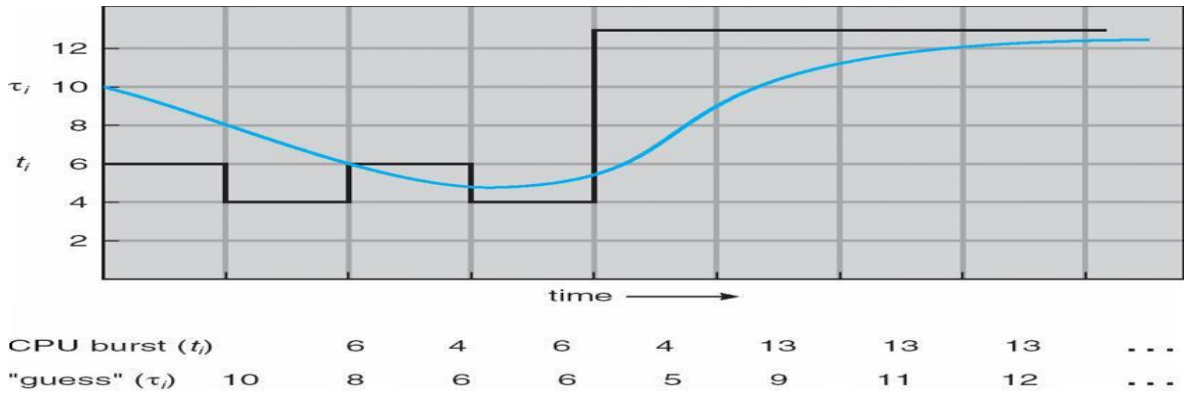
SJF scheduling chart
average waiting time = (3 + 16 + 9 + 0) / 4 = 7the length of the next CPU request



1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \le \alpha \le 1$
4. Define :

- **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
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* + …

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

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

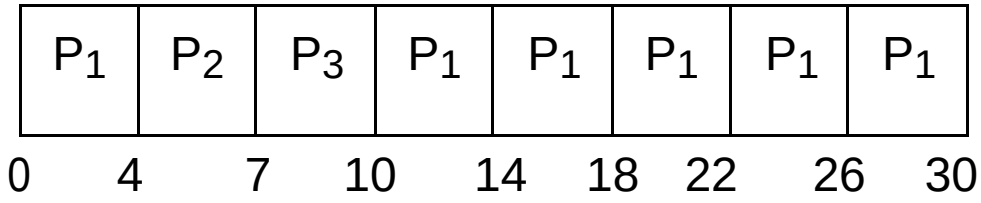
Round Robin (RR)

- 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

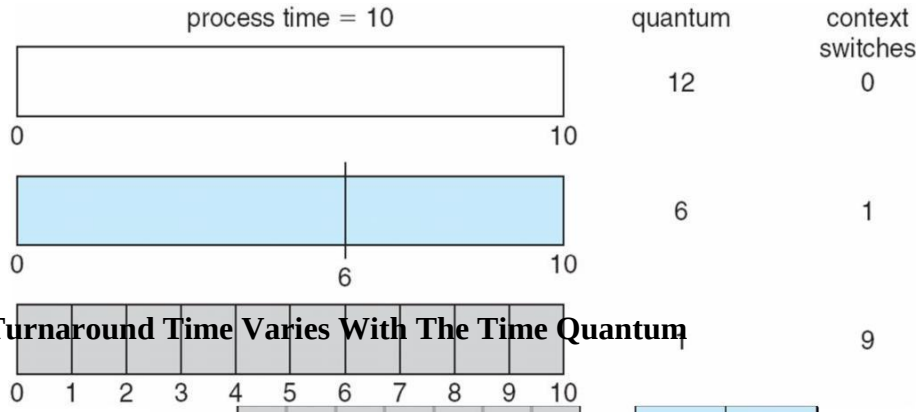
Process	**Burst Time**
P1	24
P2	3
P3	3

The Gantt chart is:



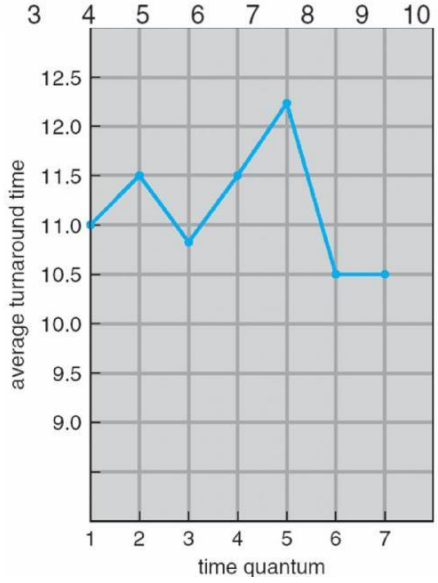
Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

process	time
P_1	6
P_2	3
P_3	1
P_4	7

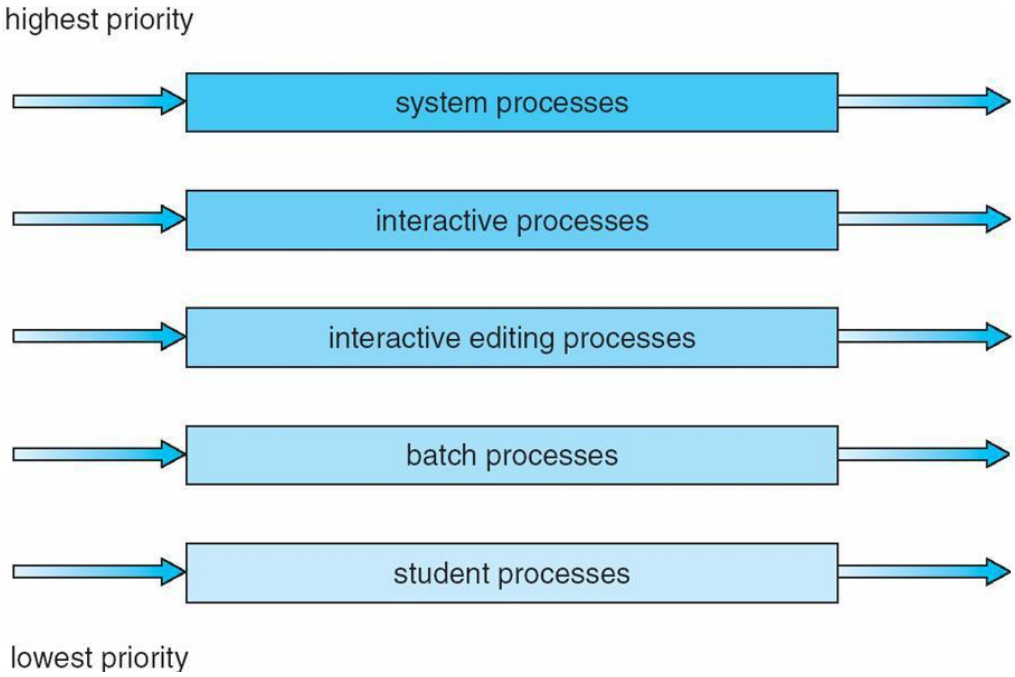


Multilevel Queue

- 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

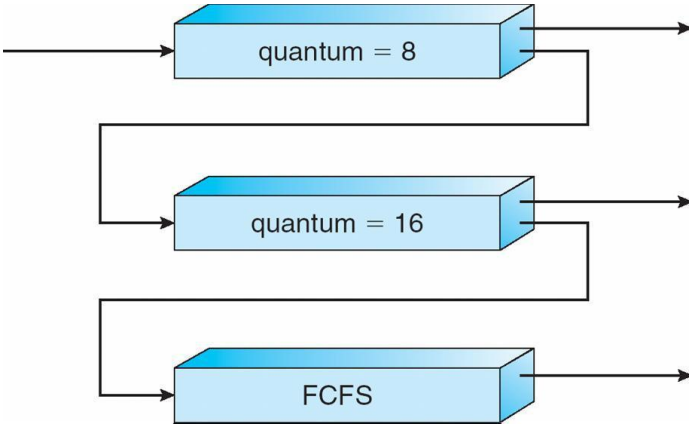
method used to determine which queue a process will enter when that process needs service

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 *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



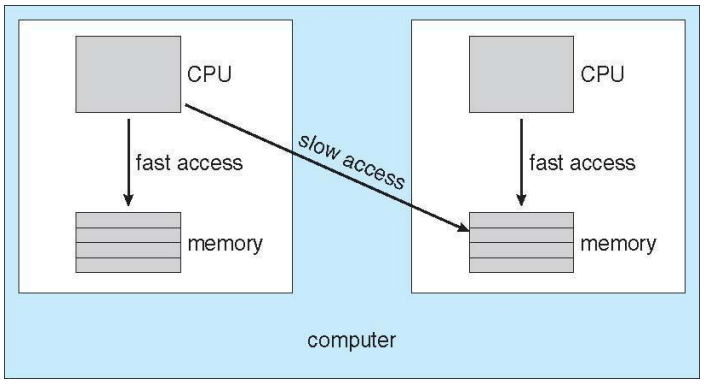
Thread Scheduling

- 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

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**

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

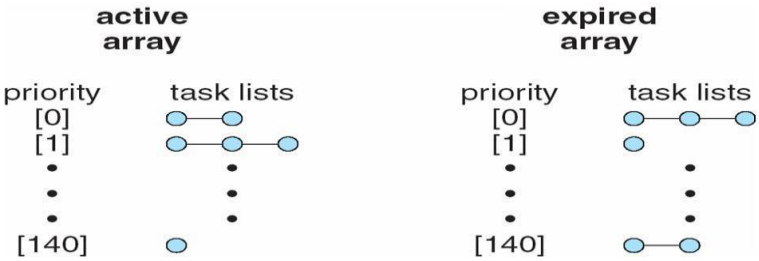
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

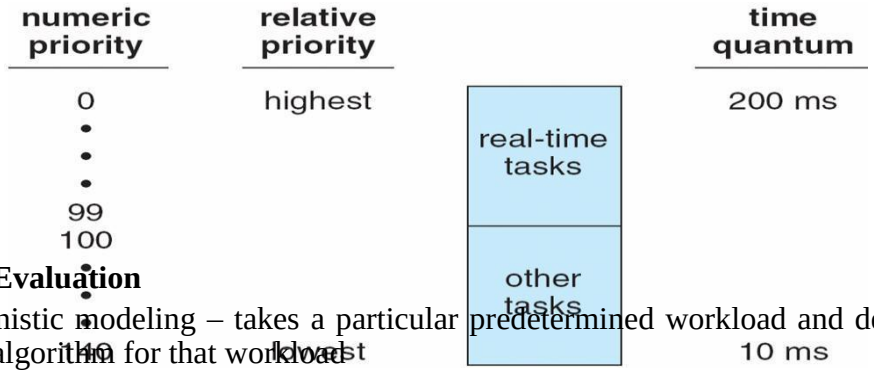
Linux Scheduling

- 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



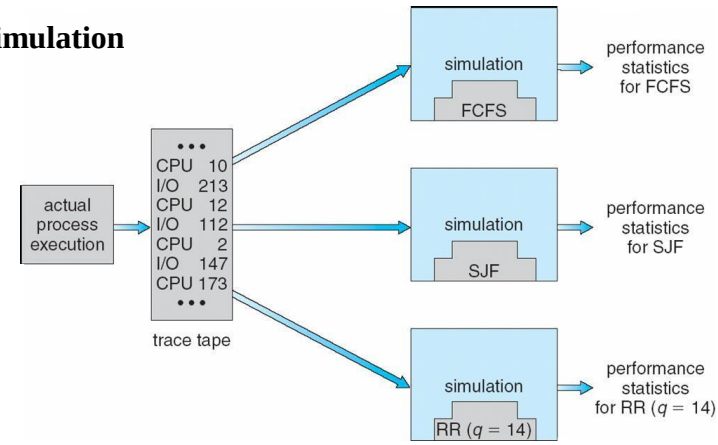
List of Tasks Indexed According to Priorities



Algorithm Evaluation

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

Evaluation of CPU schedulers by Simulation



Process Synchronization

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

Producer

```
while (true) {

    /* produce an item and put in nextProduced */
     while (count == BUFFER_SIZE)
        ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) %
          BUFFER_SIZE; count++;
}
```

Consumer

```
while (true) {
       while (count == 0)
           ; // do nothing
           nextConsumed =  buffer[out];
            out = (out + 1) %
                BUFFER_SIZE; count--;
        /*  consume the item in nextConsumed
}
```

Race Condition

count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

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

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

Solution to Critical-Section Problem

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

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

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

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
- int turn;
- Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Algorithm for Process Pi

```
do {
    flag[i] =
    TRUE; turn = j;
    while (flag[j] && turn ==
        j); critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
  Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions Atomic = non-interruptable
- Either test memory word and set value Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

TestAndSet Instruction

Definition:

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

Solution using TestAndSet

Shared boolean variable lock., initialized to false. Solution:

```
            do {
        while ( TestAndSet (&lock ))
                ;   // do nothing
              //    critical
        section lock = FALSE;
              //      remainder section
 } while (TRUE);
```

Swap Instruction

Definition:

```
void Swap (boolean *a, boolean *b)
 {
       boolean temp =
       *a; *a = *b;
       *b = temp:
 }
```

Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```
    do {
          key = TRUE;
          while ( key == TRUE)
                Swap (&lock, &key );

                  // critical section
           lock = FALSE;
                  //      remainder section
     } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet() do {

```
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
    key = TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder
section } while (TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting nSemaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
- Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
 wait (S) {
 while S <= 0
      ; // no-op
  S--;
  }
  signal (S)
 { S++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
- Also known as mutex locksnCan implement a counting semaphore S as a binary semaphore
- Provides mutual exclusionSemaphore mutex; // initialized to do {

```
                wait (mutex);
  // Critical Section
signal (mutex);
                                        // remainder section
} while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.

- Could now have busy waiting in critical section implementation But implementation code is short
  Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
- value (of type integer)
- pointer to next record in the list
- Two operations:
- block – place the process invoking the operation on the appropriate waiting queue.
- wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Implementation of wait:

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
      add this process to S-
      >list; block();
  }
}
```

Implementation of signal:

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
      remove a process P from S-
      >list; wakeup(P);
  }
}
```

Deadlock and Starvation

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

P1	P0
wait (Q);	wait (S);
wait (S);	wait (Q);
	.
	.
	.
signal (Q);	signal (S);
signal (S);	signal (Q);

Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- The structure of the producer process

```
do { // produce an item in nextp wait (empty);

        wait (mutex);
            // add the item to the buffer
         signal (mutex);
         signal (full);
} while (TRUE);
```

The structure of the consumer process

```
do {            wait (full);
        wait (mutex);
              //  remove an item from buffer to nextc
        signal (mutex);
        signal (empty);

              // consume  the  item  in
nextc } while (TRUE);
```

Readers-Writers Problem

A data set is shared among a number of concurrent processes

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

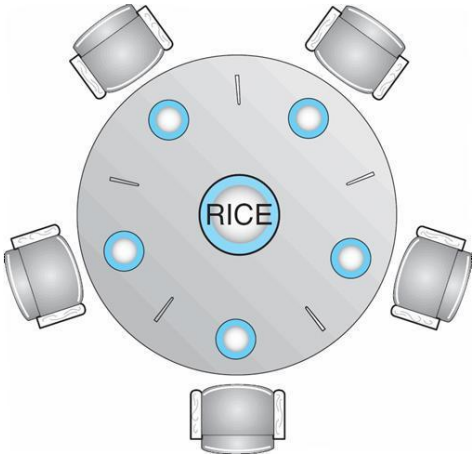
The structure of a writer process

```
do {                wait (wrt) ;

              // writing is performed
          signal (wrt) ;
} while (TRUE);
```

The structure of a reader process do {

```
wait (mutex) ;
readcount ++ ;
if (readcount ==
1) wait (wrt) ;
signal (mutex)

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

Dining-Philosophers Problem



- Shared data
- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1
- The structure of Philosopher *i*:

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

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

          //  think
} while (TRUE);
```

Problems with Semaphores

Incorrect use of semaphore operations: l signal (mutex)

….

wait (mutex)

wait (mutex) …

wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

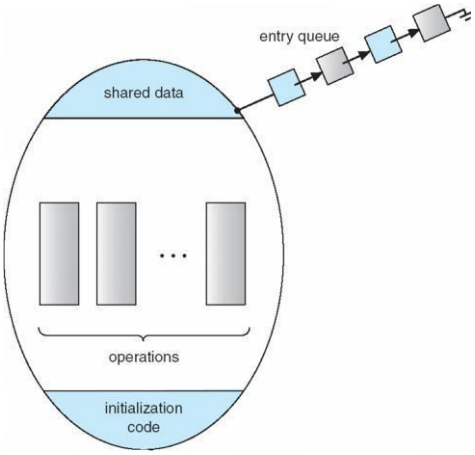
Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Only one process may be active within the monitor at a time

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

Schematic view of a Monitor



Condition Variables

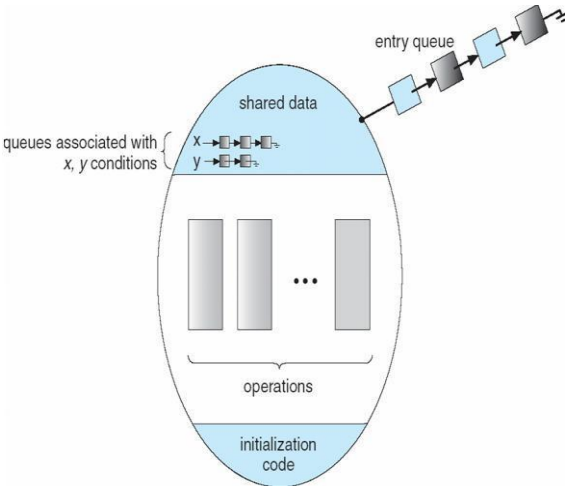
condition x, y;

Two operations on a condition variable:

x.wait () – a process that invokes the operation is suspended.

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

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
{
enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];
void pickup (int i) {
```

```
        state[i] =
        HUNGRY; test(i);
        if (state[i] != EATING) self [i].wait;
    }

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


void test (int i) {
          if ( (state[(i + 4) % 5] != EATING)
          && (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) )
              { state[i] = EATING ;
          self[i].signal () ;
           }
    }
     initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
    }
}
```

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

```
DiningPhilosophters.pickup (i);
    EAT
DiningPhilosophers.putdown (i);
```

Monitor Implementation Using Semaphores

Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next-count = 0;nEach procedure F will be replaced by
   wait(mutex);
      …
body of F;
      …
   if (next_count >
    0) signal(next)
   else
    signal(mutex);nMutual exclusion within a monitor is ensured.
```

Monitor Implementation

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

```
semaphore x_sem; // (initially = 0)
int x-count = 0;nThe operation x.wait can be implemented as:

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

The operation x.signal can be implemented as:

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

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time)
        { if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- **Implements a variety of locks to support** multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data

- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
- An event acts much like a condition variable

Linux Synchronization

- Linux:lPrior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
- semaphores
- spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
- mutex locks
- condition variablesnNon-portable extensions include:
- read-write locks
- spin locks

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
- Here we are concerned with changes to stable storage – disk
- Transaction is series of read and write operations
- Terminated by commit (transaction successful) or abort (transaction failed) operation Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
- Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
- Example: disk and tape
- Stable storage – Information never lost
- Not actually possible, so approximated via replication or RAID to devices with independent failure modes

- Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
- Log on stable storage, each log record describes single transaction write operation,
  - including Transaction name
  - Data item name
  - Old value
  - New value
- <Ti starts> written to log when transaction Ti starts
- <Ti commits> written when Ti commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

Using the log, system can handle any volatile memory errors

- Undo(Ti) restores value of all data updated by Ti
- Redo(Ti) sets values of all data in transaction Ti to new values
- Undo(Ti) and redo(Ti) must be idempotent
- Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
- If log contains <Ti starts> without <Ti commits>, undo(Ti)
- If log contains <Ti starts> and <Ti commits>, redo(Ti)

Checkpoints

Log could become long, and recovery could take long
Checkpoints shorten log and recovery time.
Checkpoint scheme:
1.Output all log records currently in volatile storage to stable
storage 2.Output all modified data from volatile to stable storage
3.Output a log record <checkpoint> to the log on stable storage
Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
- Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T0 and T1
- Execute T0, T1 atomically

- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T0 then T1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- Nonserial schedule allows

- overlapped execute
- Resulting execution not necessarily incorrect
- Consider schedule S, operations Oi, Oj
- Conflict if access same data item, with at least one write
- If Oi, Oj consecutive and operations of different transactions & Oi and Oj don't conflict
- Then S' with swapped order Oj Oi equivalent to S
- If S can become S' via swapping nonconflicting operations
- S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
- Follow locking protocol for access control

- Locks
- Shared – Ti has shared-mode lock (S) on item Q, Ti can read Q but not write Q
- Exclusive – Ti has exclusive-mode lock (X) on Q, Ti can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
- Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
- Growing – obtaining locks
- Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction Ti associated with timestamp TS(Ti) before Ti starts
- TS(Ti) < TS(Tj) if Ti entered system before Tj
- TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
- If TS(Ti) < TS(Tj), system must ensure produced schedule equivalent to serial schedule where Ti appears before Tj

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
- W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
- R-timestamp(Q) – largest timestamp of successful read(Q)
- Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order Suppose Ti executes read(Q)

- If TS(Ti) < W-timestamp(Q), Ti needs to read value of Q that was already overwritten
  - read operation rejected and Ti rolled back
- If TS(Ti) ≥ W-timestamp(Q)
  - read executed, R-timestamp(Q) set to max(R-timestamp(Q), TS(Ti))

Timestamp-ordering Protocol

Supose Ti executes write(Q)

If TS(Ti) < R-timestamp(Q), value Q produced by Ti was needed previously and Ti assumed it would never be produced

Write operation rejected, Ti rolled back

If TS(Ti) < W-tiimestamp(Q), Ti attempting to write obsolete value of Q

Write operation rejected and Ti rolled back

Otherwise, write executed

Any rolled back transaction Ti is assigned new timestamp and restarted

Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

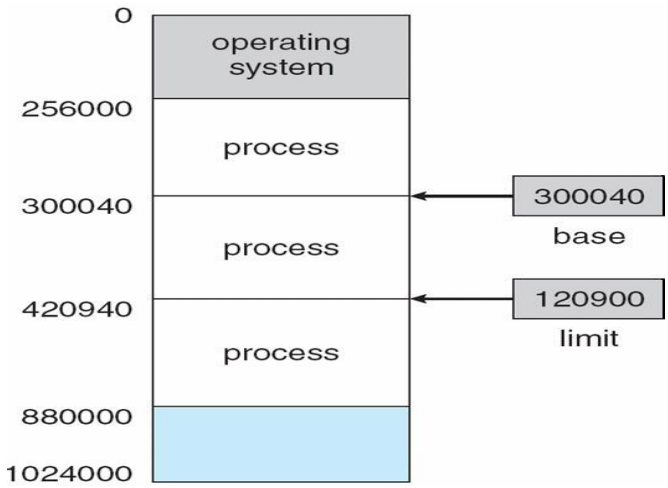
UNIT – III

Memory Management

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

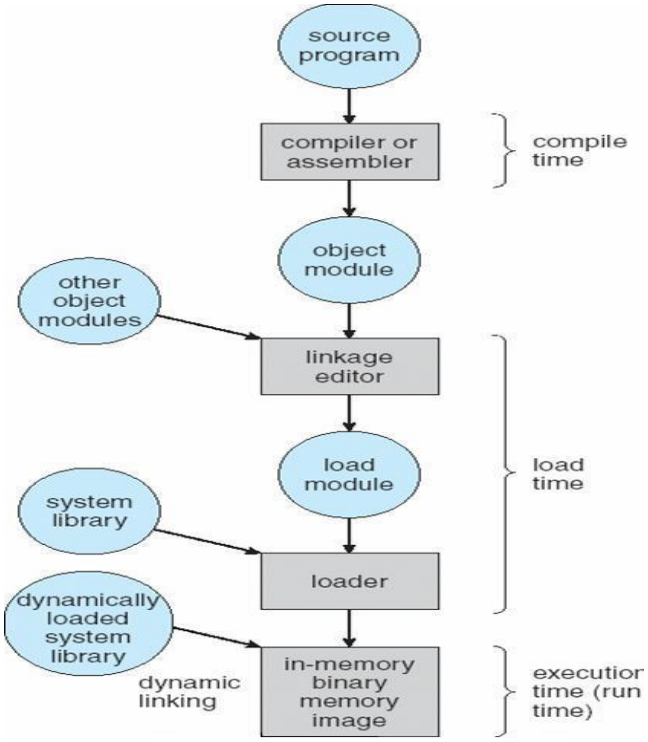
A pair of **base** and **limit** registers define the logical address space



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



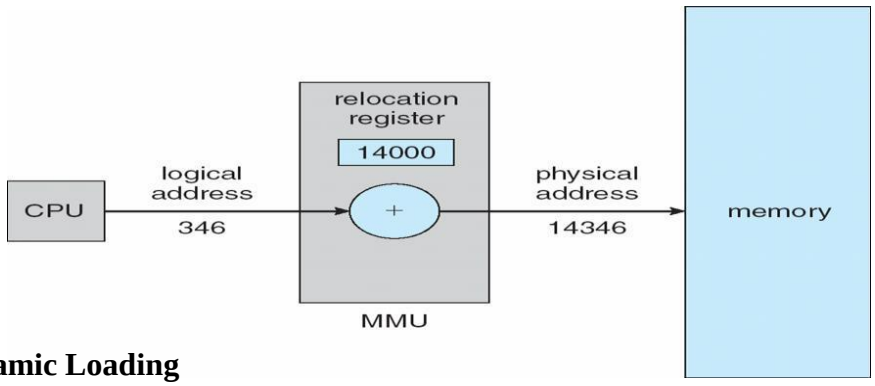
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded

- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

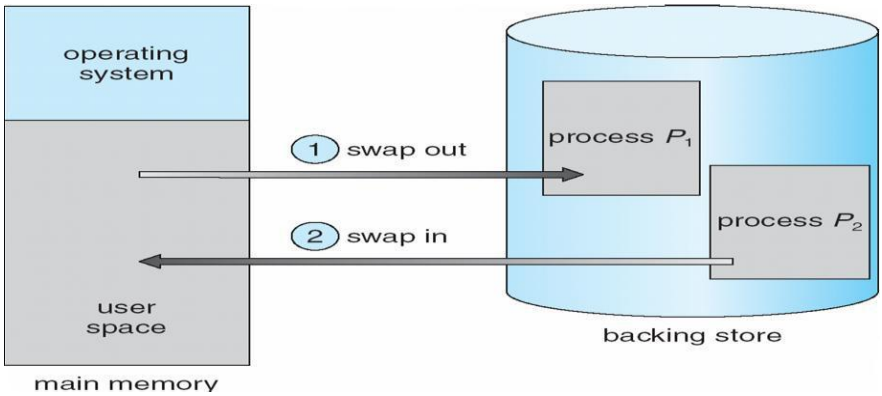
Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
System maintains a **ready queue** of ready-to-run processes which have memory images on disk

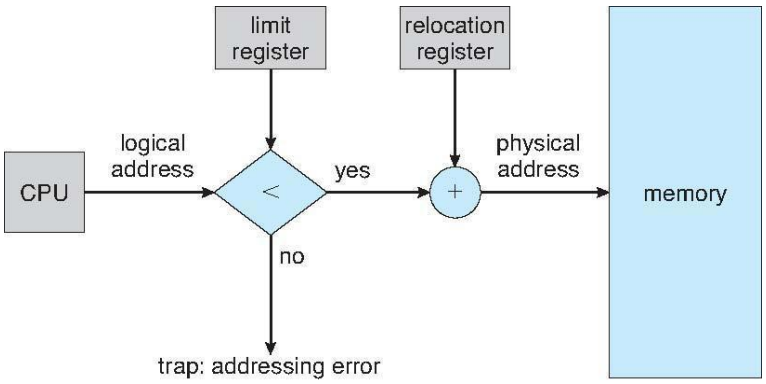
Schematic View of Swapping



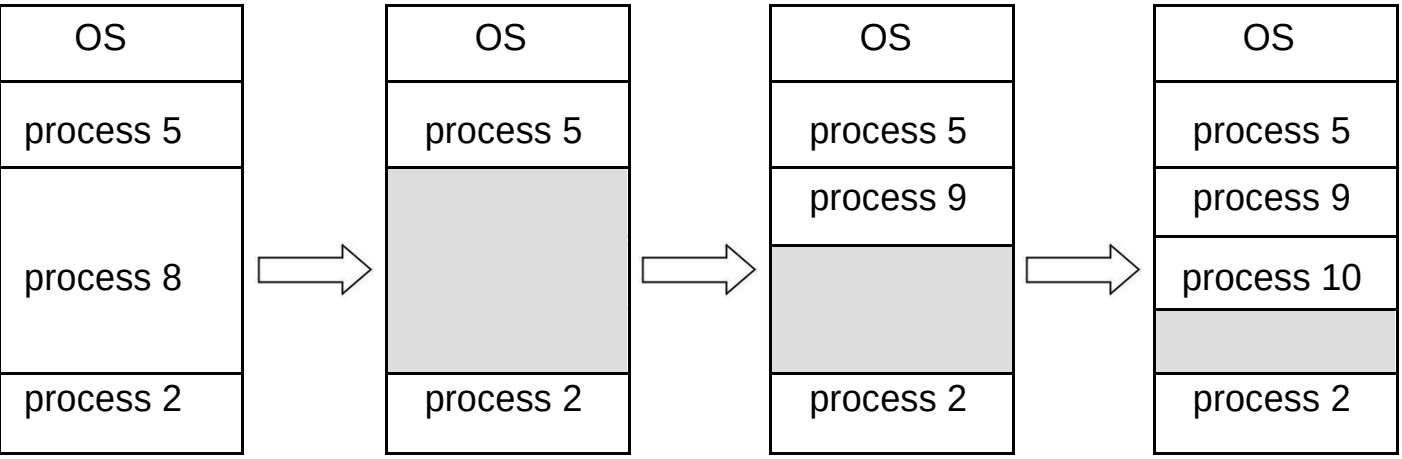
Contiguous Allocation

- Main memory usually into two partitions:
- Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory Relocation registers used to protect user processes from each other, and from changing operating-system code and data
- Base register contains value of smallest physical address
- Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers



- Multiple-partition allocation
- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
- a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
- Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time.

- I/O problem
  Latch job in memory while it is involved in I/O
  Do I/O only into OS buffers

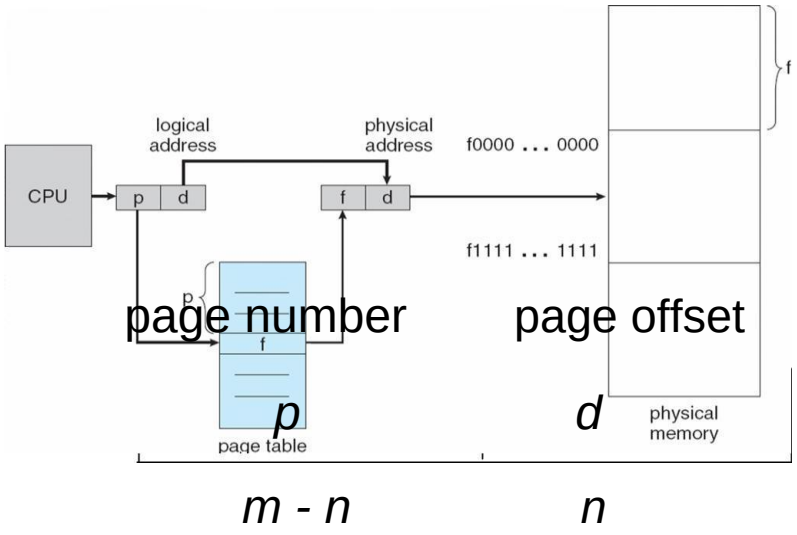
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages** Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

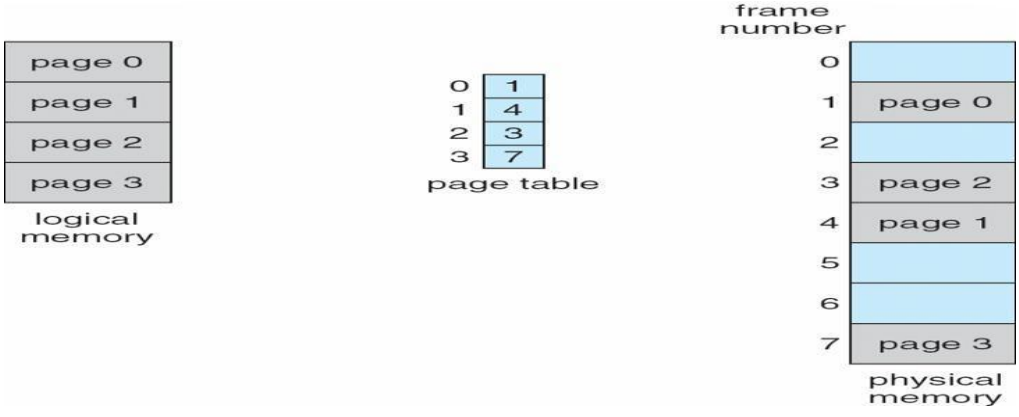
Address Translation Scheme

- Address generated by CPU is divided into
- **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical address space 2*m and page size 2*n

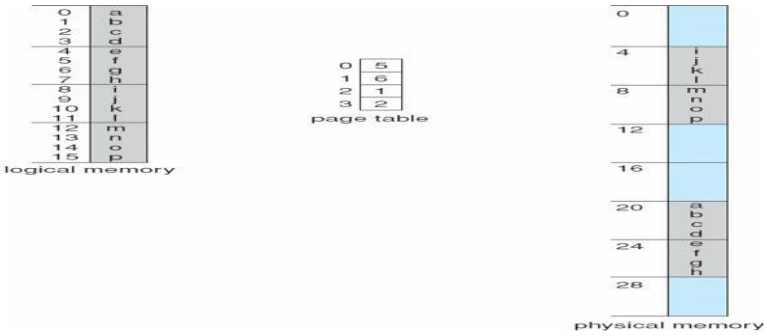
Paging Hardware



Paging Model of Logical and Physical Memory

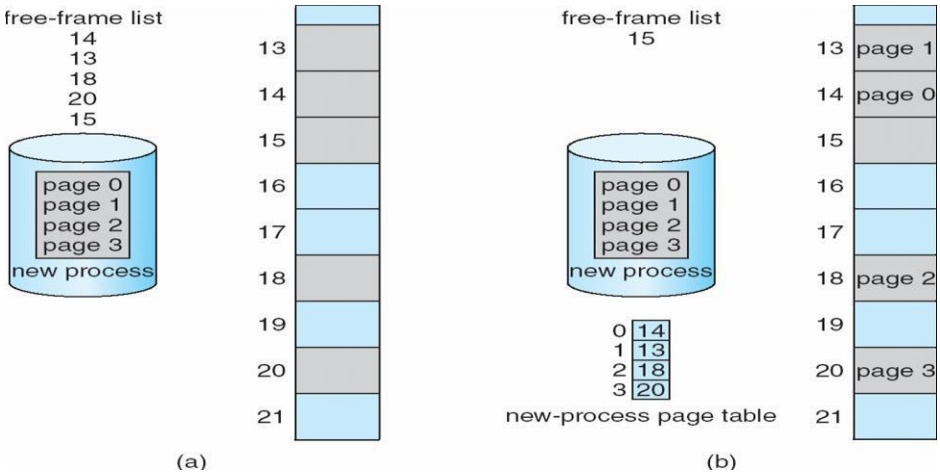


Paging Example



32-byte memory and 4-byte pages

Free Frames



Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

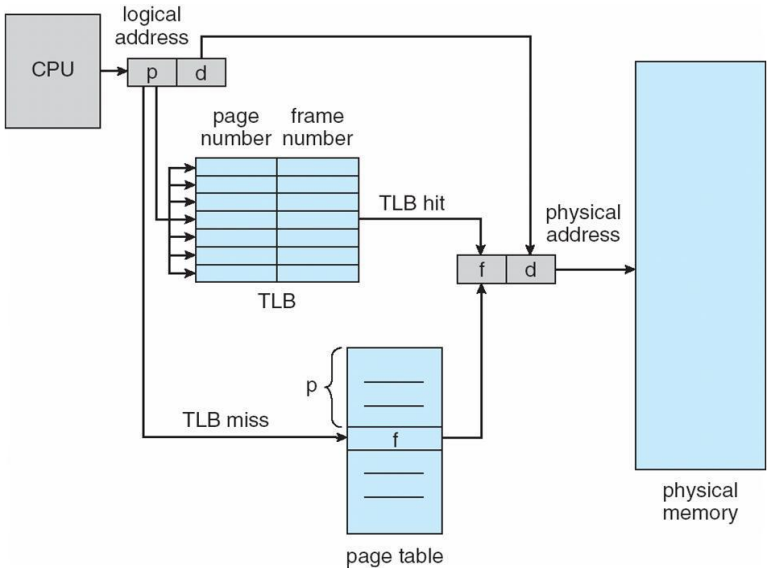
Associative Memory

- Associative memory – parallel search
- Address translation (p, d)
- If p is in associative register, get frame # out

- Otherwise get frame # from page table in memory

Page #	Frame #

Paging Hardware With TLB



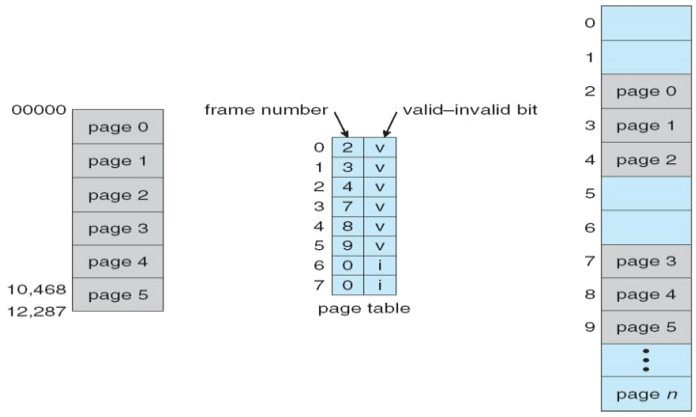
Effective Access Time

- Associative Lookup = e time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = an **Effective Access Time** (EAT)

$$EAT = (1 + e)\,a + (2 + e)(1 - a) = 2 + e - a$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space
- **Valid (v) or Invalid (i) Bit In A Page Table**



Shared Pages

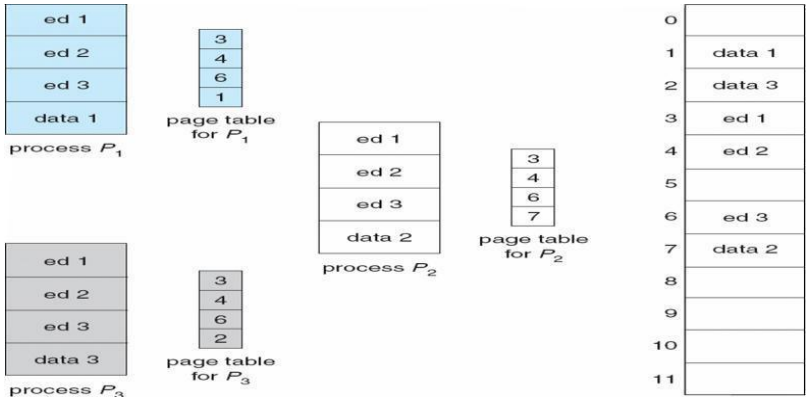
Shared code

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



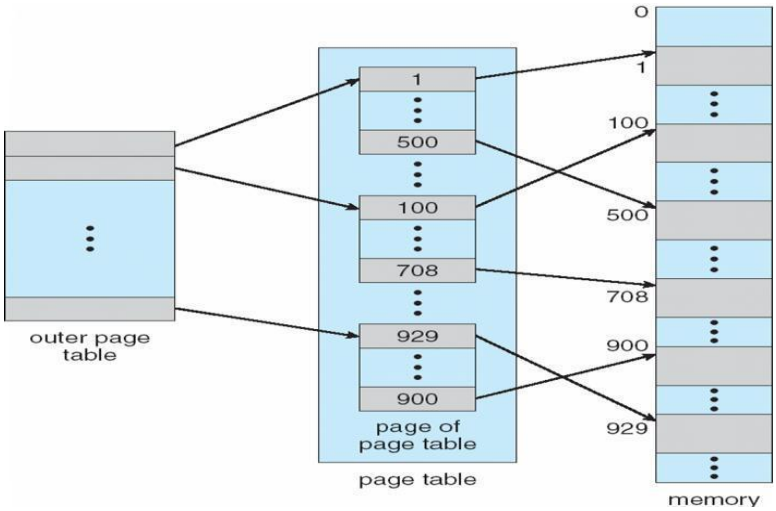
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

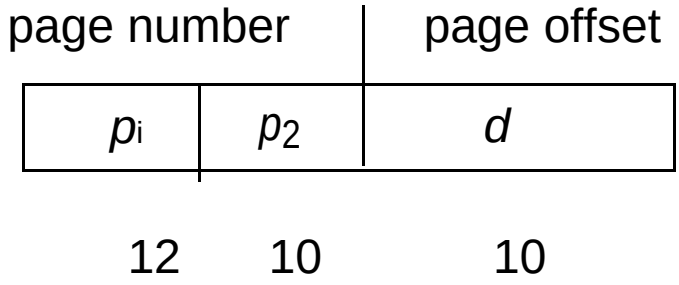
Two-Level Page-Table Scheme



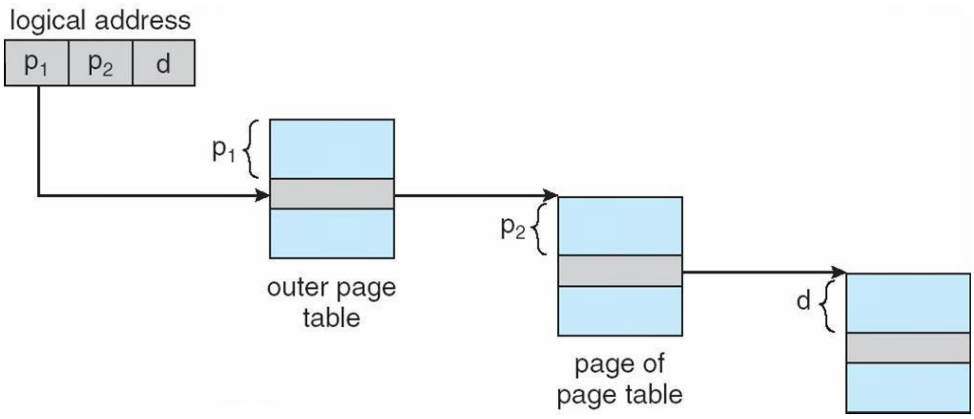
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
- a page number consisting of 22 bits
- a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
- a 12-bit page number
- a 10-bit page offset
- Thus, a logical address is as follows:

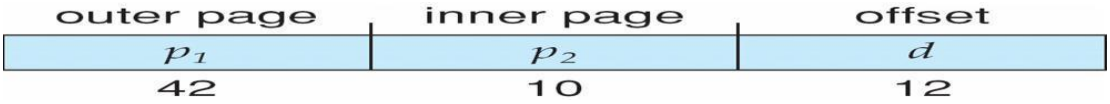
where *pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table

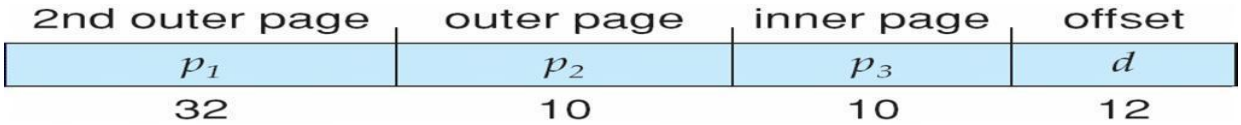


Address-Translation Scheme



Three-level Paging Scheme

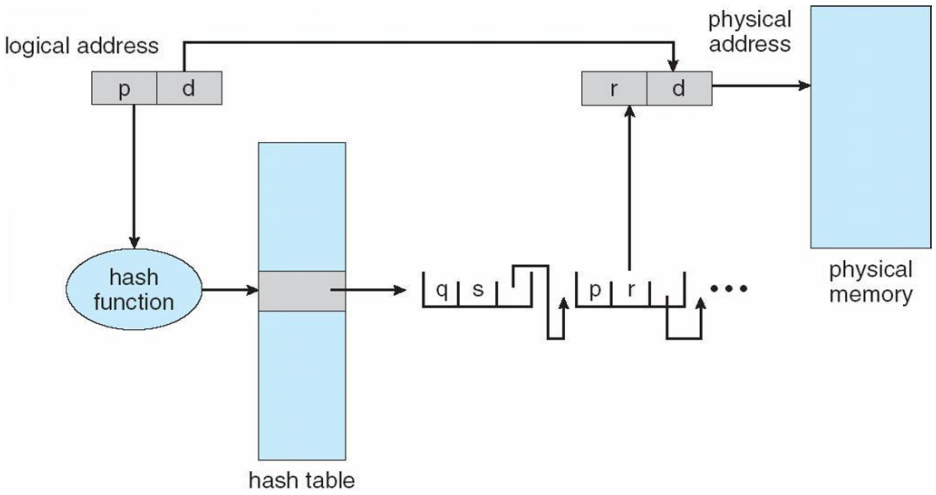




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
- If a match is found, the corresponding physical frame is extracted

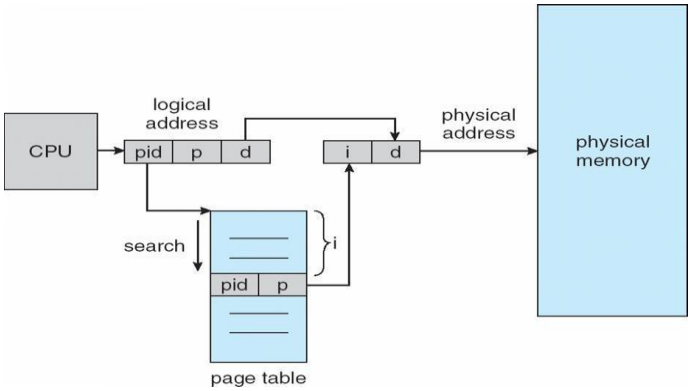
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture

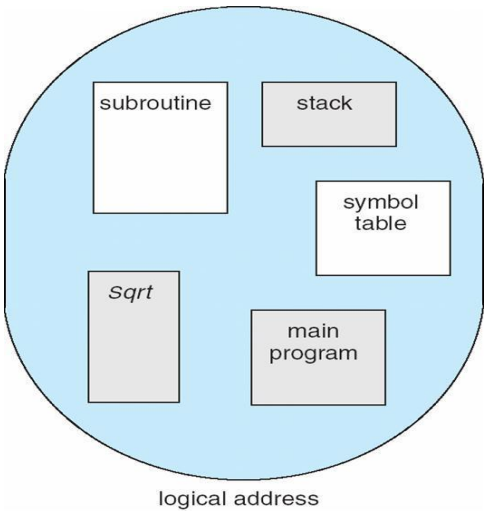


Segmentation

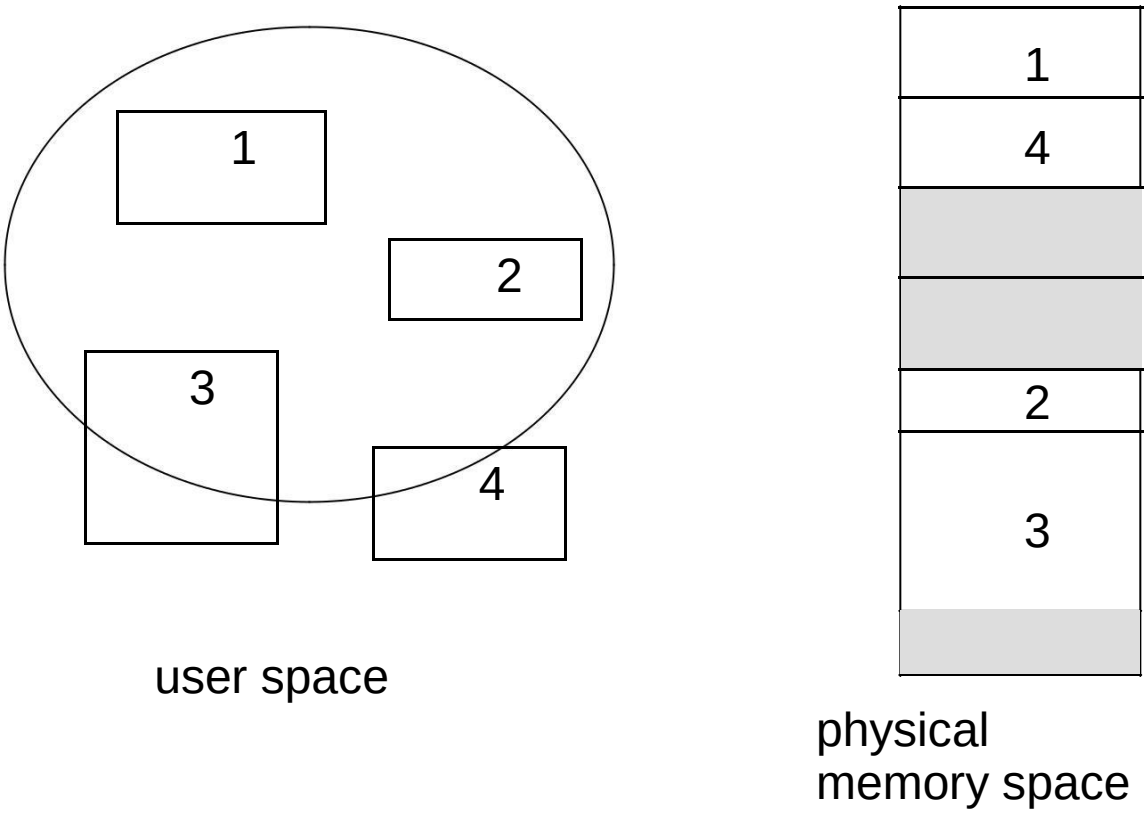
Memory-management scheme that supports user view of memory

- A program is a collection of segments
- A segment is a logical unit such as:
- main program
- procedure function
- method
- object
- local variables, global variables
- common block
- stack
- symbol table
- arrays

User's View of a Program



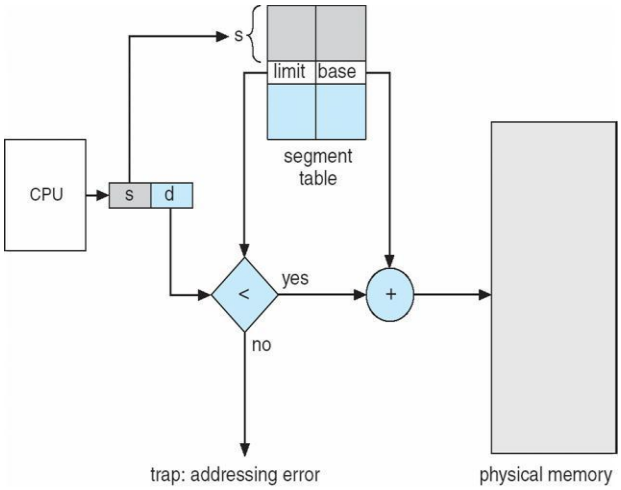
Logical View of Segmentation



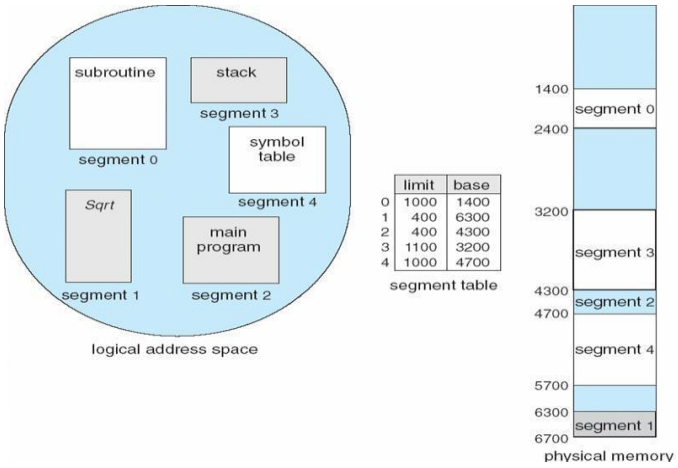
Segmentation Architecture

- Logical address consists of a two tuple:
  - <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program; segment number ***s*** is legal if ***s*** < **STLR**
- Protection
- With each entry in segment table associate:
- validation bit = 0 Þ illegal segment
- read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



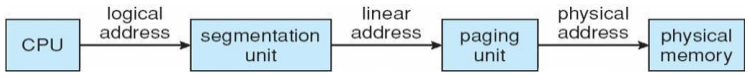
Example of Segmentation

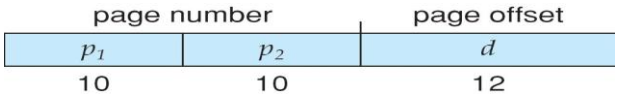


Example: The Intel Pentium

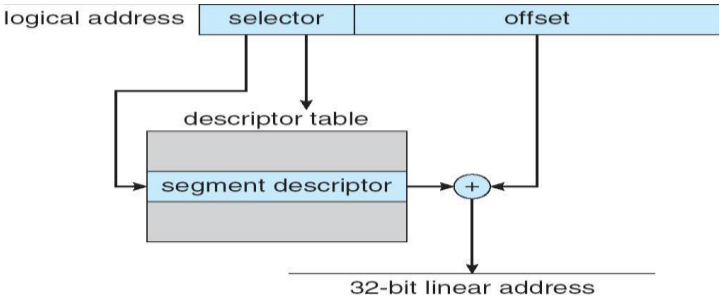
- Supports both segmentation and segmentation with paging
- CPU generates logical address
- Given to segmentation unit
  Which produces linear addresses
- Linear address given to paging unit
  Which generates physical address in main memory Paging units form equivalent of MMU

Logical to Physical Address Translation in Pentium

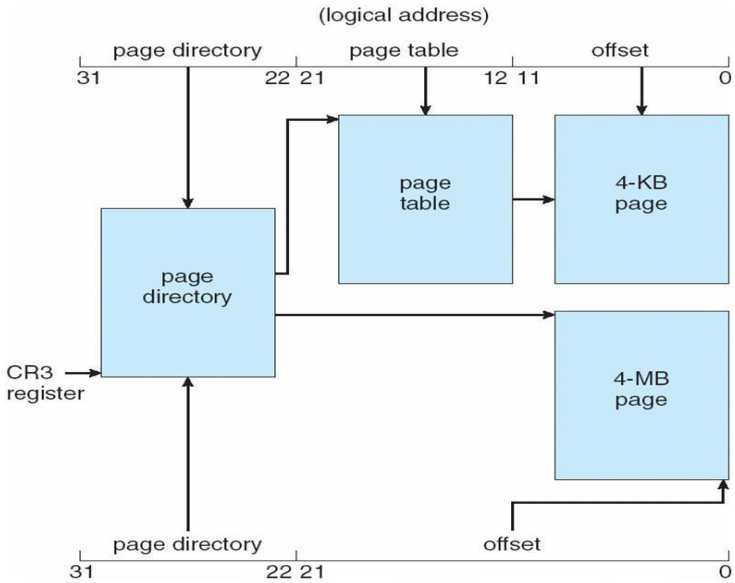




Intel Pentium Segmentation

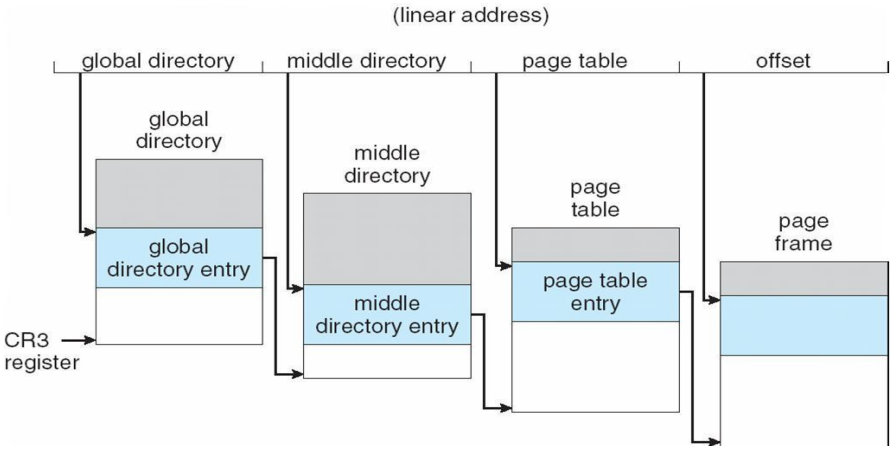


Pentium Paging Architecture



Linear Address in Linux

Three-level Paging in Linux



UNIT – IV
File System Implementation

File Concept
- Contiguous logical address spaceand Types:
- Data
- numeric
- character
- binary
- Program

File Structure
- None - sequence of words, bytes
- Simple record structure
- Lines
- Fixed length
- Variable length
- Complex Structures
- Formatted document
- Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
- Operating system
- Program

File Attributes
- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **Delete**
- **Truncate**
- *Open(Fi)* – search the directory structure on disk for entry *Fi*, and move the content of entry to memory
- *Close (Fi)* – move the content of entry *Fi* in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage open files:
- File pointer: pointer to last read/write location, per process that has the file open
- File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes close it
- Disk location of the file: cache of data access information
- Access rights: per-process access mode information

Open File Locking

- Provided by some operating systems and file systems
- Mediates access to a file
- Mandatory or advisory:
- **Mandatory** – access is denied depending on locks held and requested
- **Advisory** – processes can find status of locks and decide what to do

File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Access Methods

Sequential Access

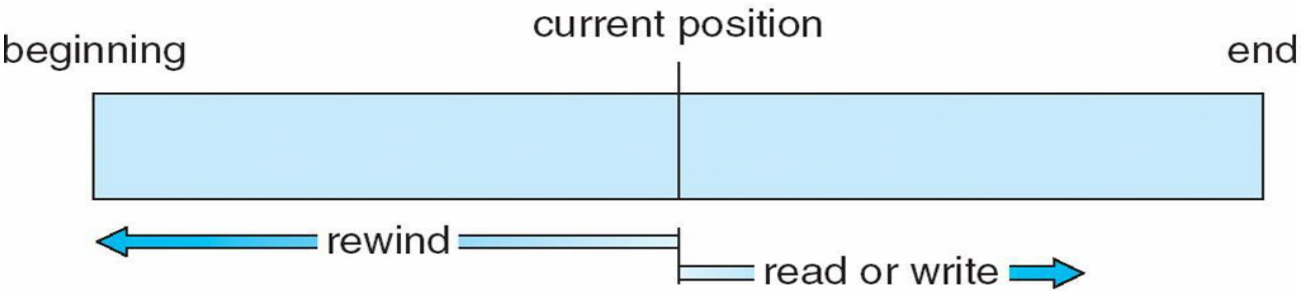
- read next
- write
- reset
- no read after last write
- (rewrite)

Direct Access

- read n
- write n
- position to n
- read next
- write next
- rewrite n

n = relative block number

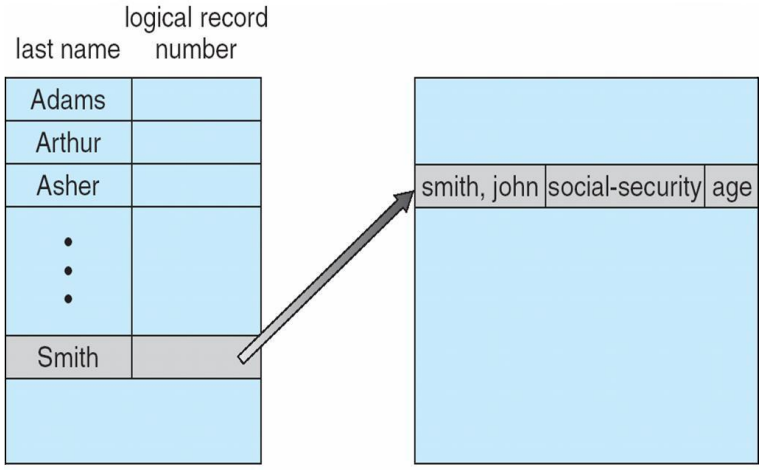
Sequential-access File



Simulation of Sequential Access on Direct-access File

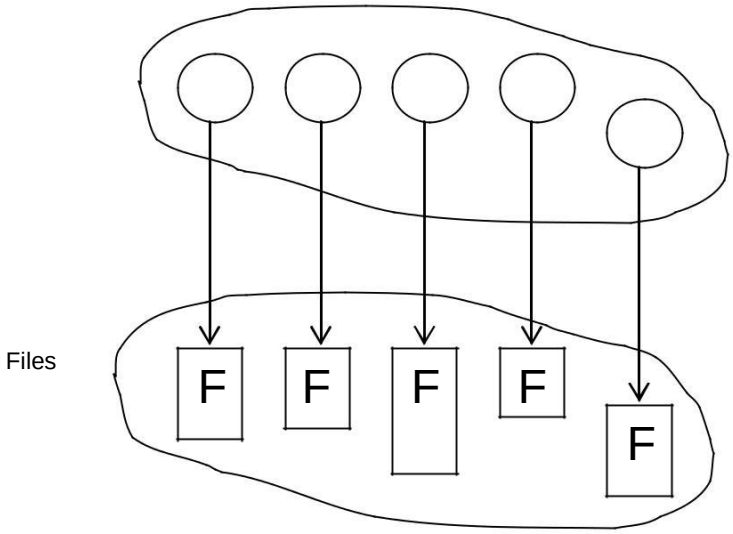
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Example of Index and Relative Files



Directory Structure

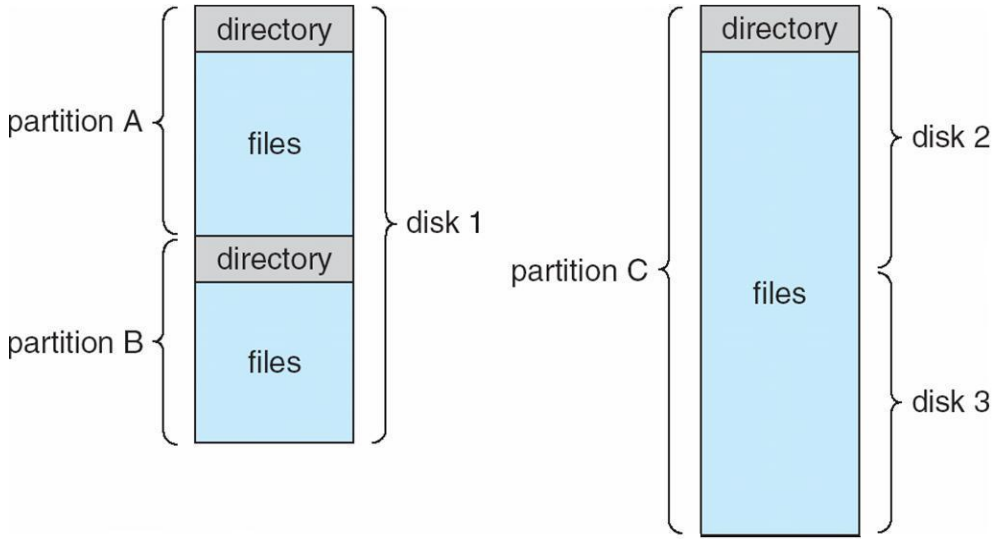
- A collection of nodes containing information about all files
- Both the directory structure and the files reside on disk
- Backups of these two structures are kept on tapes



Disk Structure

- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system's info in device directory or volume table of contents
- As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer

A Typical File-system Organization



Operations Performed on Directory

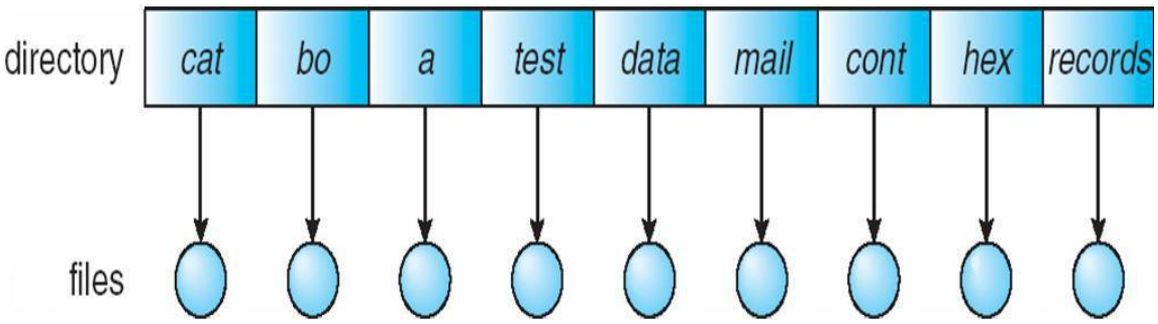
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Organize the Directory (Logically) to Obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
- Two users can have same name for different files
- The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

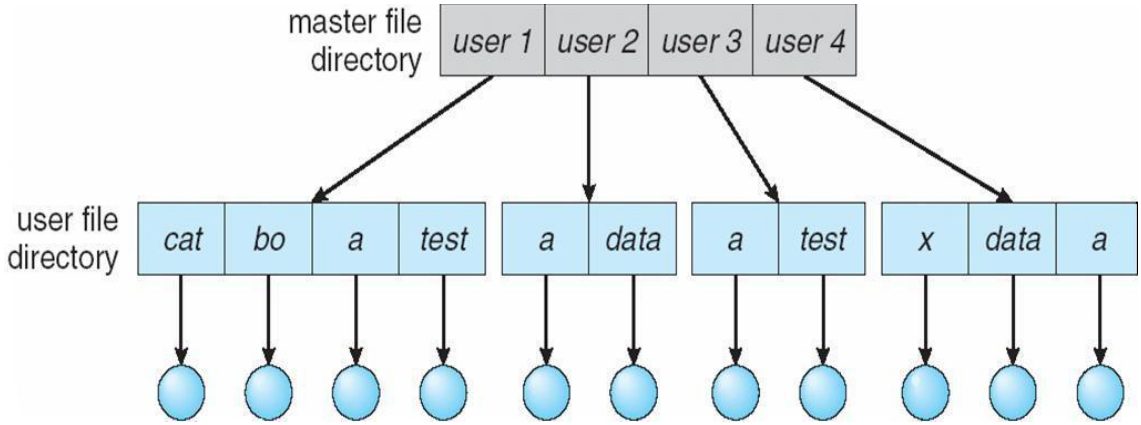
- A single directory for all users



- Naming problem
- Grouping problem

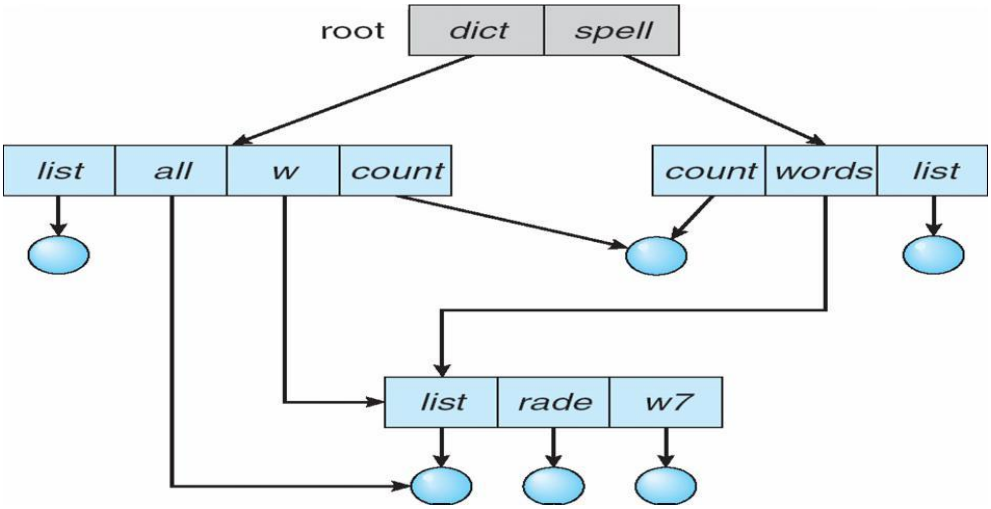
Two-Level Directory

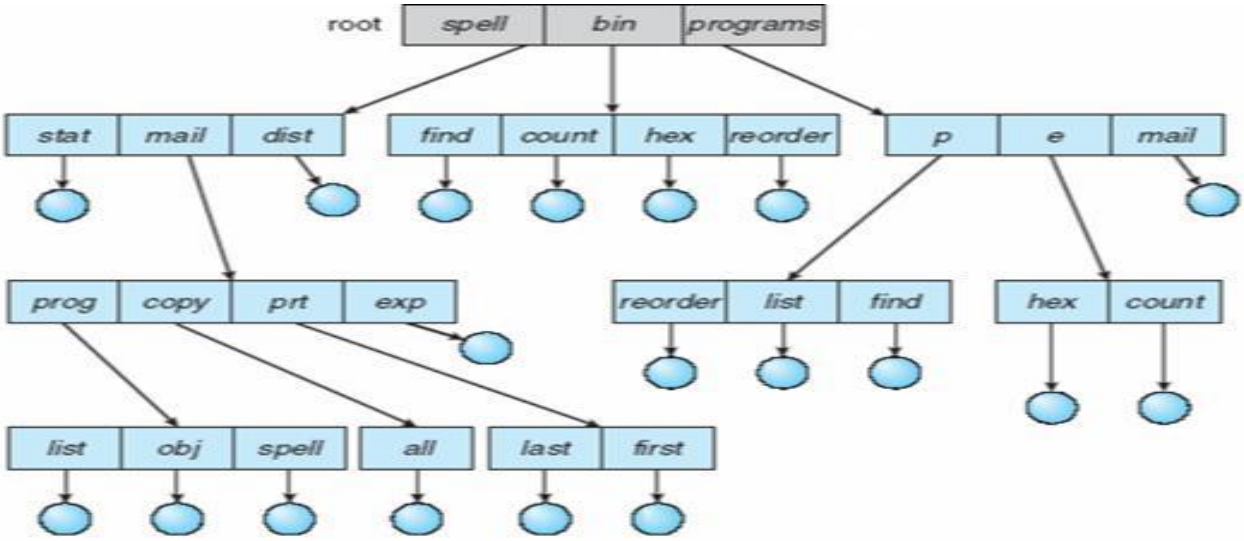
- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories

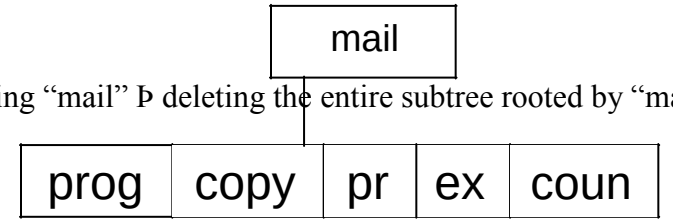




Creating a new file is done in current directory

- Delete a file rm <file-name>
- Creating a new subdirectory is done in current directory
  mkdir <dir-name>
  Example: if in current directory /mail

mkdir count



- Deleting "mail" Þ deleting the entire subtree rooted by "mail"

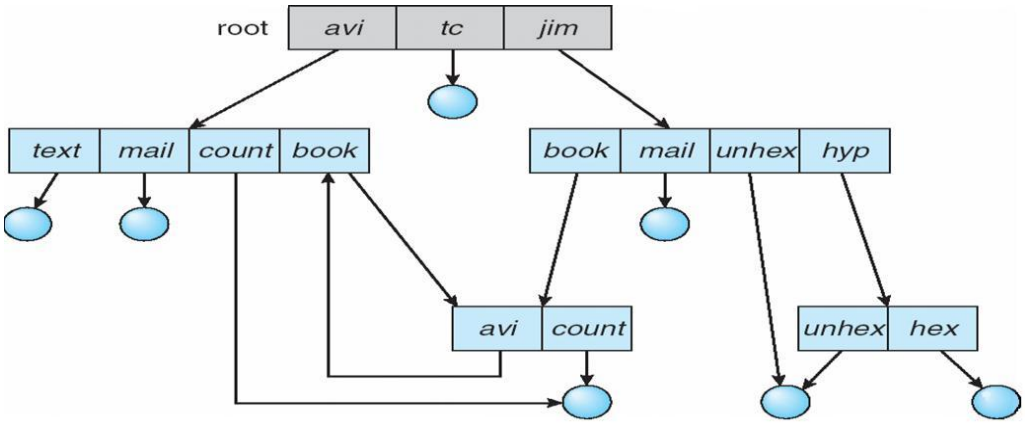
Acyclic-Graph Directories

- Have shared subdirectories and files

Two different names (aliasing)If *dict* deletes *list* Þ dangling pointer

- Solutions:
- Backpointers, so we can delete all pointers Variable size records a problem
- Backpointers using a daisy chain organization
- Entry-hold-count solution
- New directory entry type
- **Link** – another name (pointer) to an existing file
- **Resolve the link** – follow pointer to locate the file

General Graph Directory

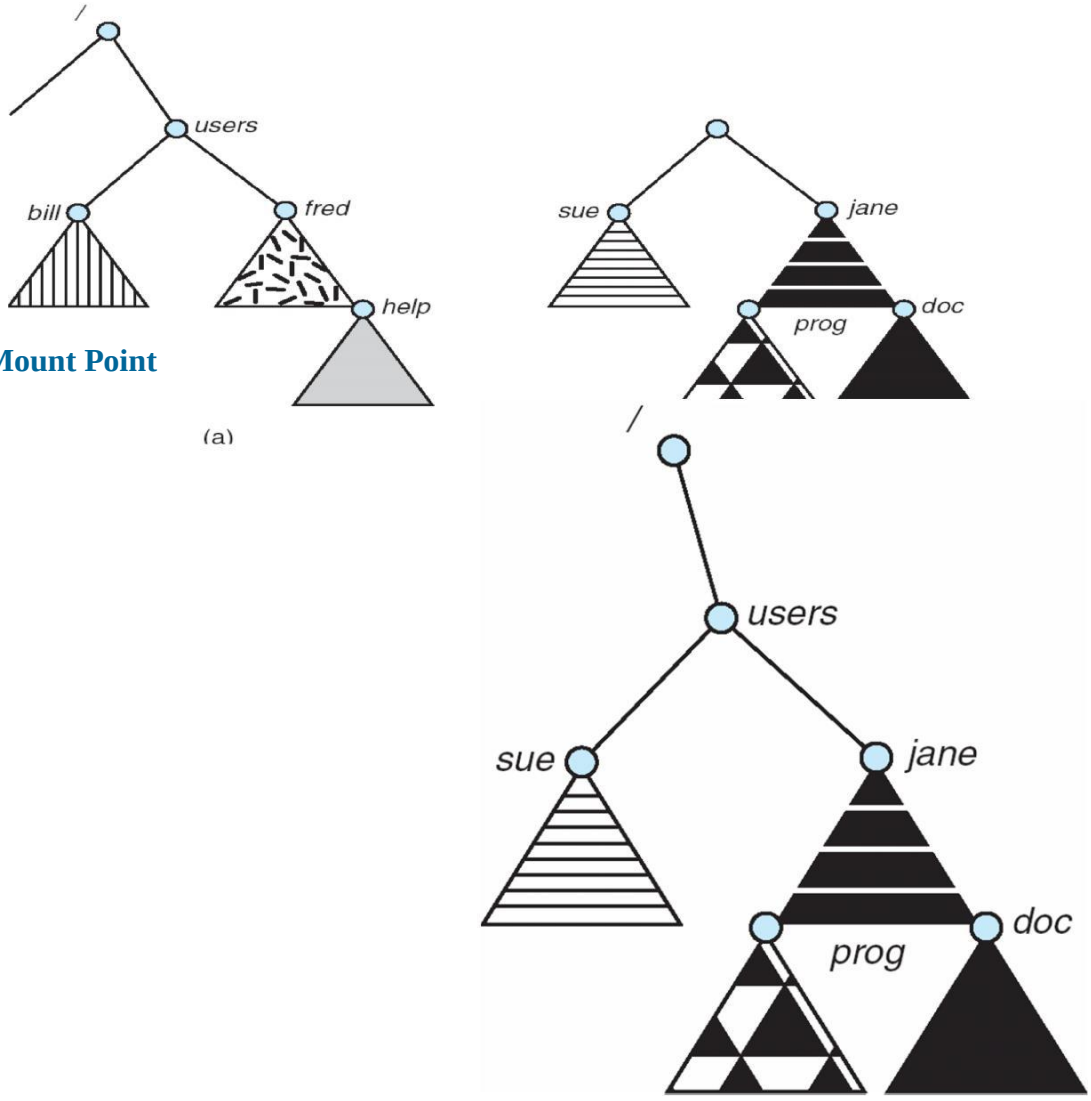


- How do we guarantee no cycles?
- Allow only links to file not subdirectories
- Garbage collection
- Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point**

(a) Existing. (b) Unmounted Partition



File Sharing

- Sharing of files on multi-user systems is desirable Sharing may be done through a **protection** scheme On distributed systems, files may be shared across a network Network File System (NFS) is a common distributed file-sharing method

File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user
  Group IDs allow users to be in groups, permitting group access rights

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
- Manually via programs like FTP

- Automatically, seamlessly using **distributed file systems**
- Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
- Server can serve multiple clients
- Client and user-on-client identification is insecure or complicated
- **NFS** is standard UNIX client-server file sharing protocol
- **CIFS** is standard Windows protocol
- Standard operating system file calls are translated into remote calls
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

File Sharing – Failure Modes

- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

File Sharing – Consistency Semantics

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
- Similar to Ch 7 process synchronization algorithms
- Tend to be less complex due to disk I/O and network latency (for remote file systems
- Andrew File System (AFS) implemented complex remote file sharing semantics
- Unix file system (UFS) implements:
- Writes to an open file visible immediately to other users of the same open file
- Sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics
- Writes only visible to sessions starting after the file is closed

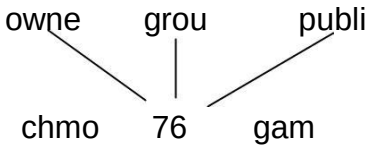
Protection

- File owner/creator should be able to control:
- what can be done
- by whomTypes of access
- **Read**
- **Write**
- **Execute**
- **Append**
- **Delete**
- **List**

- **Access Lists and Groups**
- Mode of access: read, write, execute
- Three classes of users
- RWX

a) **owner access** RWX	7	Þ	1 1 1
b) **group access** RWX	6	Þ	1 1 0
c) **public access**	1	Þ	0 0 1

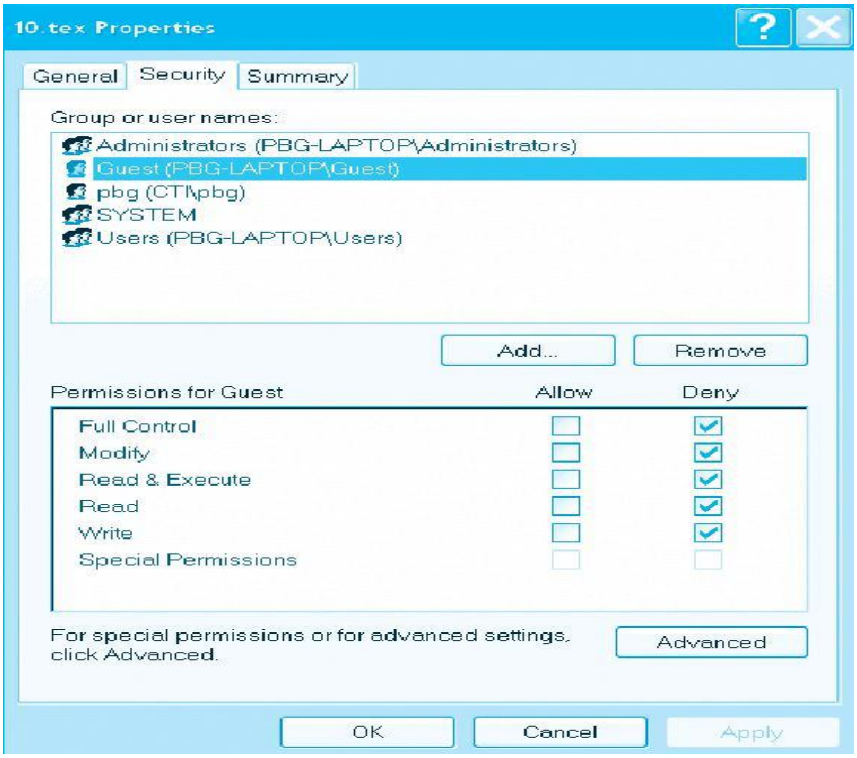
- Ask manager to create a group (unique name), say G, and add some users to the group. For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game

Windows XP Access-control List Management



A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

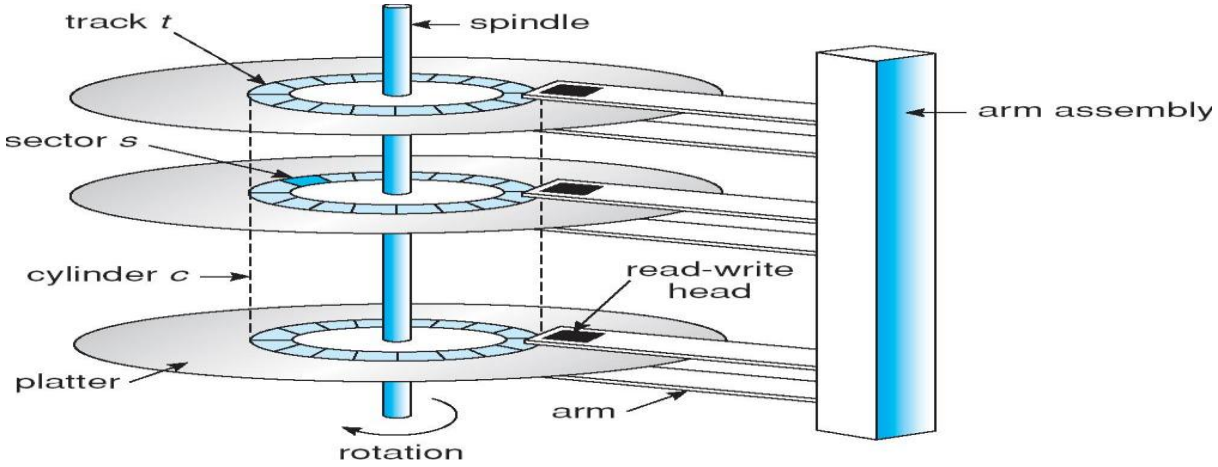
Mass-Storage Systems

Describe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices Explain the performance characteristics of mass-storage devices Discuss operating-system services provided for mass storage, including RAID and HSM

Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern
computers Drives rotate at 60 to 200 times per second
Transfer rate is rate at which data flow between drive and computer
Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency)
Head crash results from disk head making contact with the disk surface
That's bad
Disks can be removable
Drive attached to computer via I/O bus
Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI
Host controller in computer uses bus to talk to disk controller built into drive or storage array

Moving-head Disk Mechanism



Magnetic tape
Was early secondary-storage medium
Relatively permanent and holds large quantities of data
Access time slow
Random access ~1000 times slower than disk
Mainly used for backup, storage of infrequently-used data, transfer medium between systems
Kept in spool and wound or rewound past read-write head
Once data under head, transfer rates comparable to
disk 20-200GB typical storage
Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT

Disk Structure

Disk drives are addressed as large 1-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially Sector 0 is the first sector of the first track on the outermost cylinder

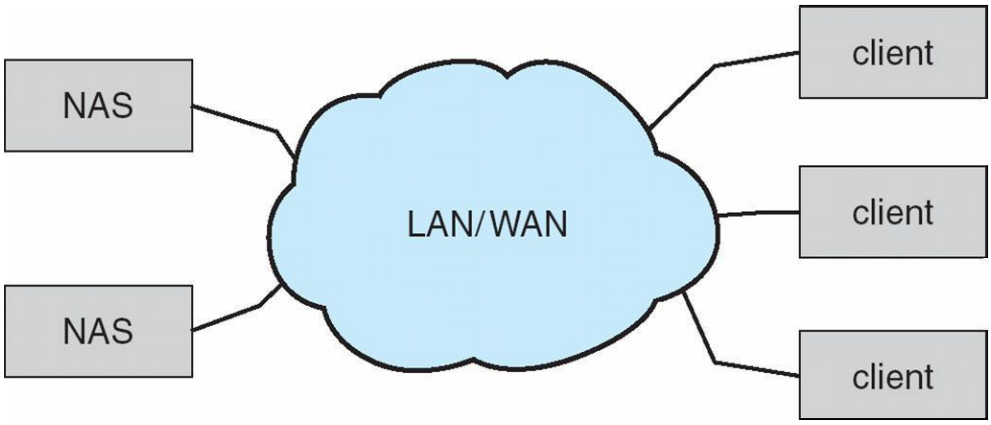
Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost

Disk Attachment

Host-attached storage accessed through I/O ports talking to I/O busses
SCSI itself is a bus, up to 16 devices on one cable, SCSI initiator requests operation and SCSI targets perform tasks
Each target can have up to 8 logical units (disks attached to device
controller FC is high-speed serial architecture
Can be switched fabric with 24-bit address space – the basis of storage area networks (SANs) in which many hosts attach to many storage units
Can be arbitrated loop (FC-AL) of 126 devices

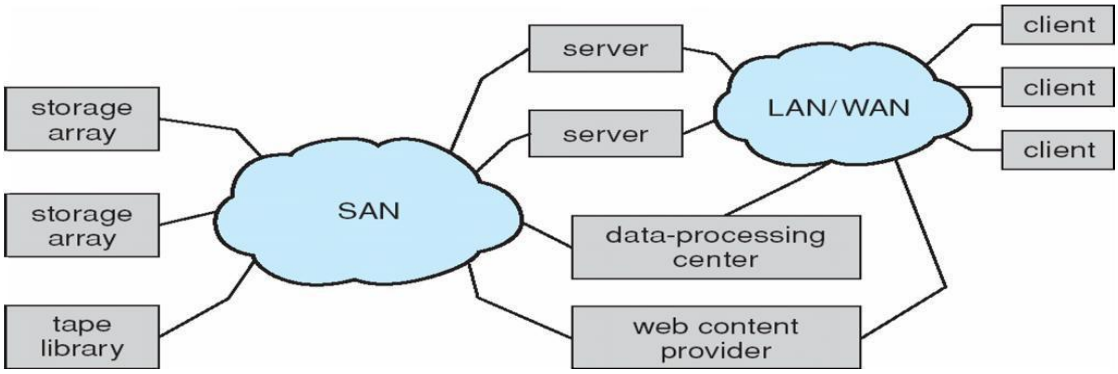
Network-Attached Storage

Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus)
NFS and CIFS are common protocols
Implemented via remote procedure calls (RPCs) between host and
storage New SCSI protocol uses IP network to carry the SCSI protocol



Storage Area Network

Common in large storage environments (and becoming more common)
Multiple hosts attached to multiple storage arrays - flexible



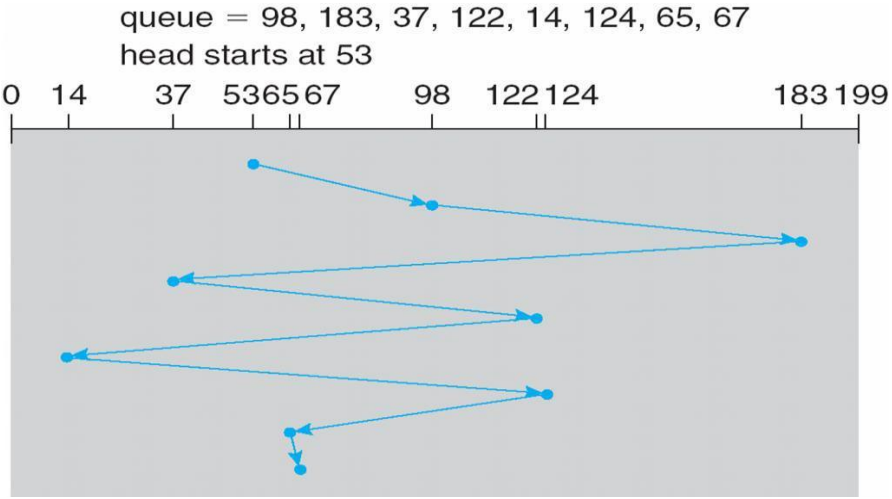
Disk Scheduling

The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
Access time has two major components
Seek time is the time for the disk are to move the heads to the cylinder containing the desired sector
Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head Minimize seek time
Seek time » seek distance
Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
Several algorithms exist to schedule the servicing of disk I/O requests .We illustrate them with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67
Head pointer 53

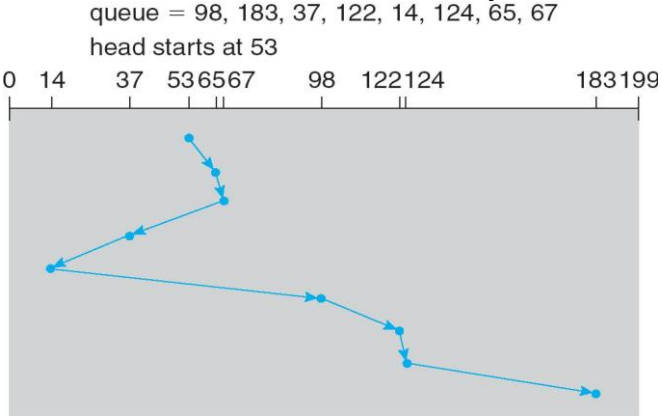
FCFS

Illustration shows total head movement of 640 cylinders



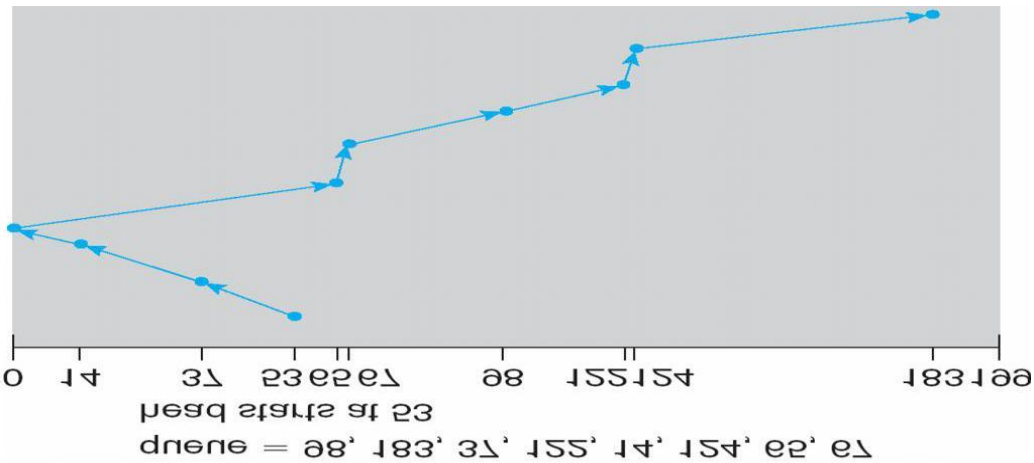
SSTF

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

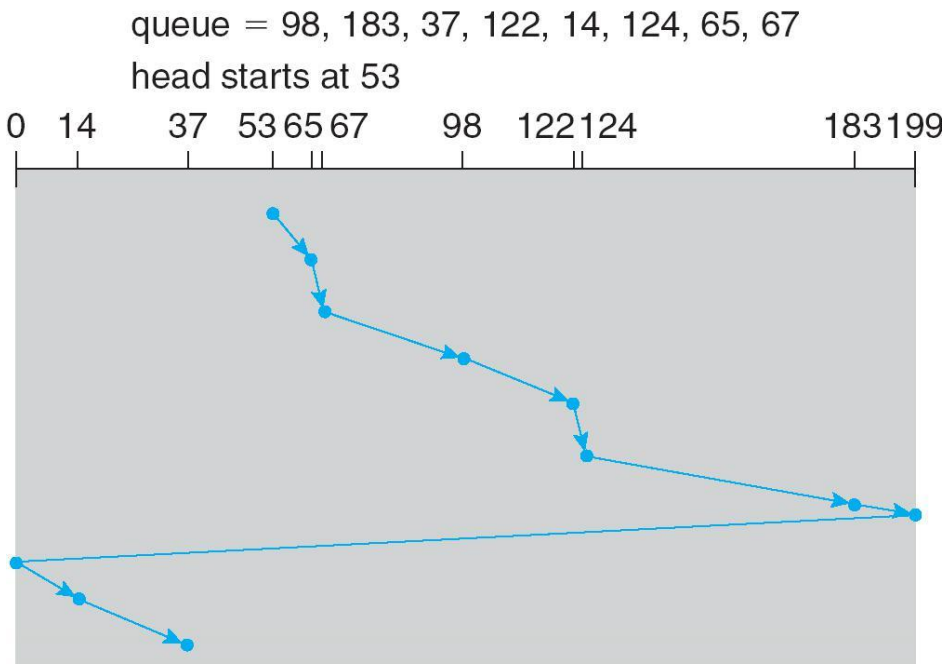


SCAN

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. SCAN algorithm Sometimes called the elevator algorithm
Illustration shows total head movement of 208 cylinders

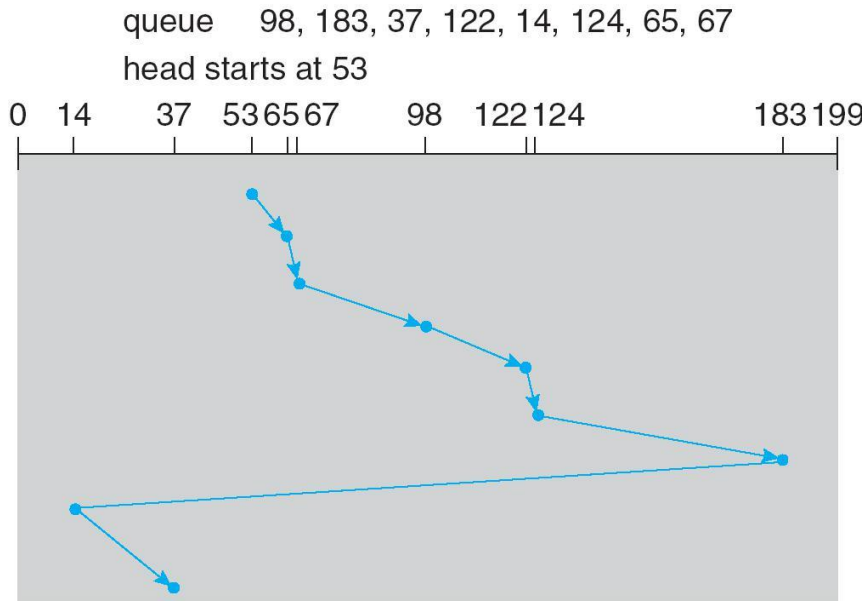


C-SCAN
Provides a more uniform wait time than SCAN
The head moves from one end of the disk to the other, servicing requests as it goes
When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
Treats the cylinders as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



C-LOOK

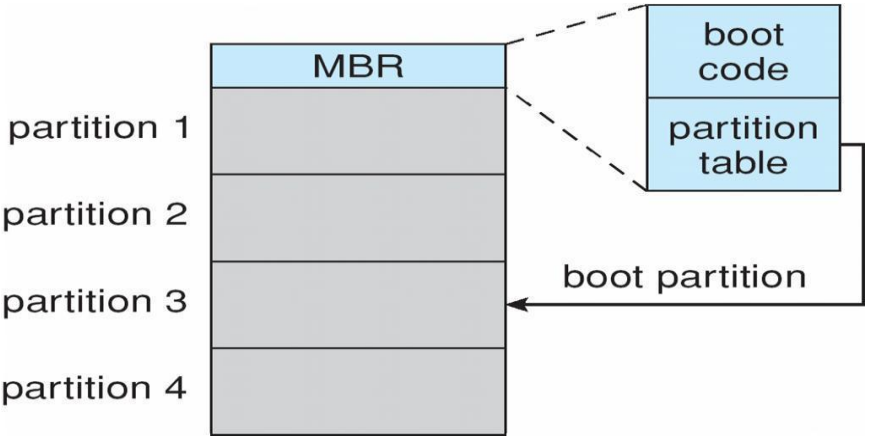
Version of C-SCAN
Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk



Selecting a Disk-Scheduling Algorithm
SSTF is common and has a natural appeal
SCAN and C-SCAN perform better for systems that place a heavy load on the disk Performance depends on the number and types of requests
Requests for disk service can be influenced by the file-allocation method
The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary
Either SSTF or LOOK is a reasonable choice for the default algorithm

Disk Management
Low-level formatting, or physical formatting — Dividing a disk into sectors that the disk controller can read and write
To use a disk to hold files, the operating system still needs to record its own data structures on the disk Partition the disk into one or more groups of cylinders
Logical formatting or "making a file system"
To increase efficiency most file systems group blocks into clusters Disk I/O done in blocks
File I/O done in clusters Boot block initializes system
The bootstrap is stored in ROM
Bootstrap loader program
Methods such as sector sparing used to handle bad blocks

Booting from a Disk in Windows 2000



Swap-Space Management

Swap-space — Virtual memory uses disk space as an extension of main memory

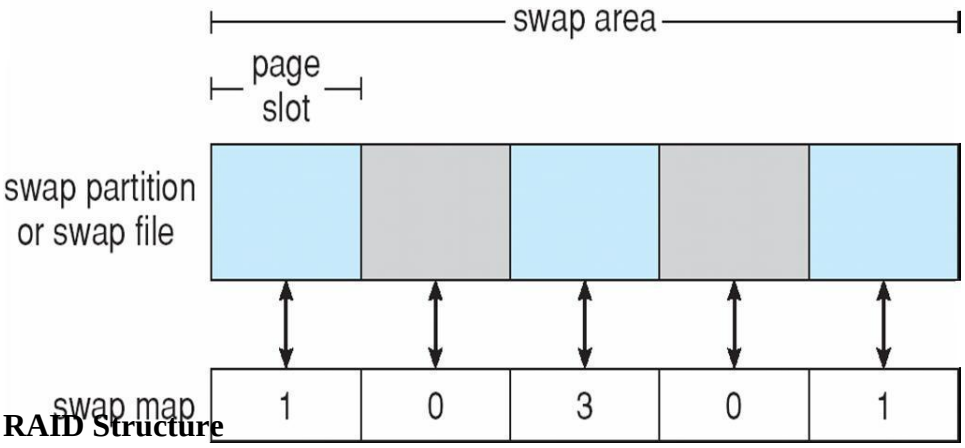
Swap-space can be carved out of the normal file system, or, more commonly, it can be in separate disk partition

Swap-space management

4.3BSD allocates swap space when process starts; holds text segment (the program) and data segment Kernel uses swap maps to track swap-space use

Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created

Data Structures for Swapping on Linux Systems



RAID Structure

RAID – multiple disk drives provides reliability via redundancy Increases the mean time to failure Frequently combined with NVRAM to improve write performance

RAID is arranged into six different levels

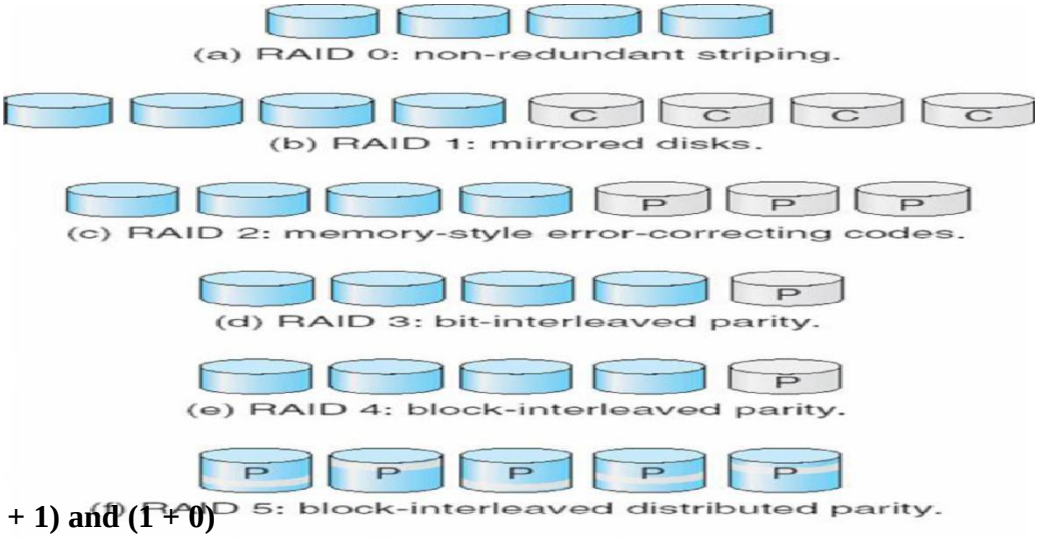
Several improvements in disk-use techniques involve the use of multiple disks working cooperatively Disk striping uses a group of disks as one storage unit RAID schemes improve performance and improve the reliability of the storage system by storing redundant data

Mirroring or shadowing (RAID 1) keeps duplicate of each disk

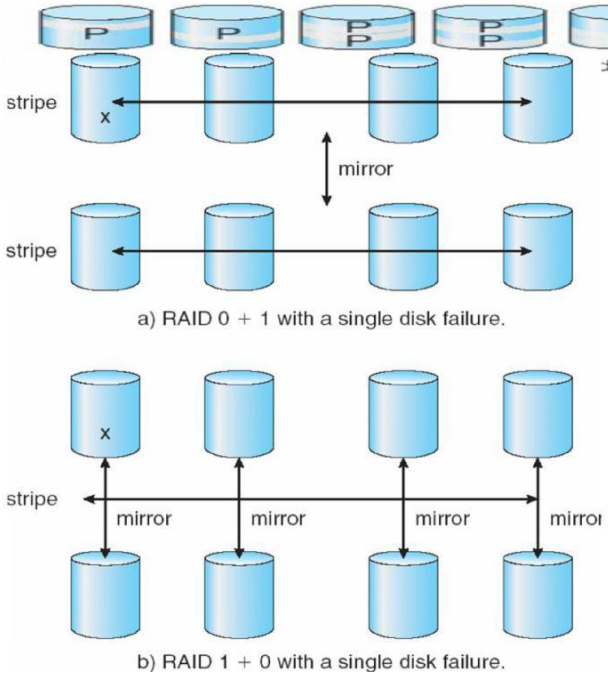
Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability Block interleaved parity (RAID 4, 5, 6) uses much less redundancy

RAID within a storage array can still fail if the array fails, so automatic replication of the data between arrays is common

Frequently, a small number of hot-spare disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them



RAID (0 + 1) and (1 + 0)



Extensions

RAID alone does not prevent or detect data corruption or other errors, just disk failures Solaris ZFS adds checksums of all data and metadata

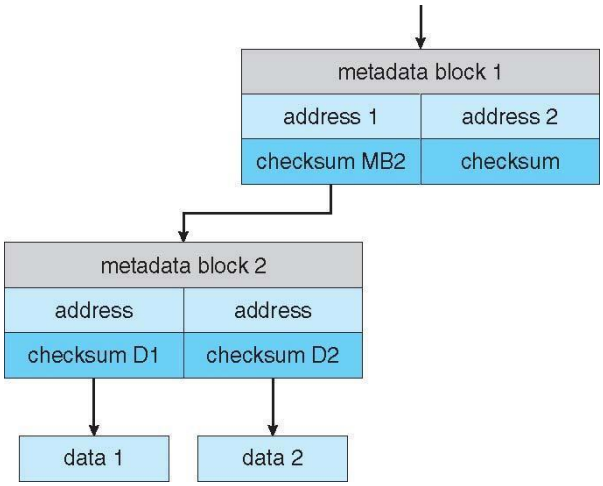
Checksums kept with pointer to object, to detect if object is the right one and whether it changed Can detect and correct data and metadata corruption

ZFS also removes volumes, partitions.

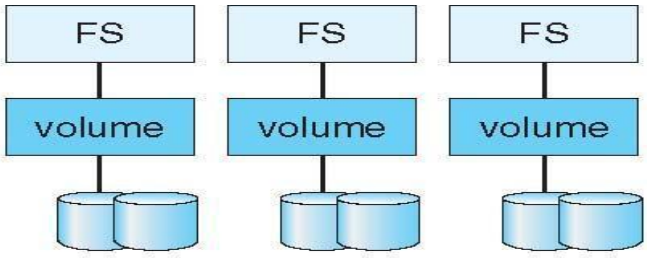
Disks allocated in pools

File systems with a pool share that pool, use and release space like “malloc” and “free” memory allocate / release calls

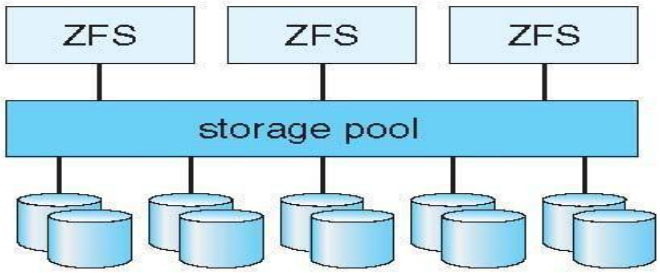
ZFS Checksums All Metadata and Data



Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

Stable-Storage Implementation

Write-ahead log scheme requires stable storage To implement stable storage:
Replicate information on more than one nonvolatile storage media with independent failure modes
Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery

Tertiary Storage Devices

Low cost is the defining characteristic of tertiary storage Generally, tertiary storage is built using removable media Common examples of removable media are floppy disks and CD-ROMs; other types are available

Removable Disks

Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB

Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure

A magneto-optic disk records data on a rigid platter coated with magnetic
material Laser heat is used to amplify a large, weak magnetic field to record a bit

Laser light is also used to read data (Kerr effect)

The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes Optical disks do not use magnetism; they employ special materials that are altered by laser light

WORM Disks

The data on read-write disks can be modified over and over

WORM ("Write Once, Read Many Times") disks can be written only once

Thin aluminum film sandwiched between two glass or plastic platters

To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered

Very durable and reliable

Read-only disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded

Tapes

Compared to a disk, a tape is less expensive and holds more data, but random access is much slower

Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data

Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library

stacker – library that holds a few tapes silo

– library that holds thousands of tapes

A disk-resident file can be archived to tape for low cost storage; the computer can stage it back into disk storage for active use

Operating System Support

Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications For hard disks, the OS provides two abstraction:

Raw device – an array of data blocks

File system – the OS queues and schedules the interleaved requests from several applications

Application Interface

Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk

Tapes are presented as a raw storage medium, i.e., and application does not not open a file on the tape, it opens the whole tape drive as a raw device

Usually the tape drive is reserved for the exclusive use of that application

Since the OS does not provide file system services, the application must decide how to use the array of blocks

Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it

Tape Drives

The basic operations for a tape drive differ from those of a disk drive

locate() positions the tape to a specific logical block, not an entire track (corresponds to seek())

The read position() operation returns the logical block number where the tape head
is The space() operation enables relative motion
Tape drives are "append-only" devices; updating a block in the middle of the tape also effectively erases everything beyond that block
An EOT mark is placed after a block that is written

File Naming

The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer
Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data. Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way

Hierarchical Storage Management (HSM)

A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks
Usually incorporate tertiary storage by extending the file
system Small and frequently used files remain on disk
Large, old, inactive files are archived to the jukebox
HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data

Speed

Two aspects of speed in tertiary storage are bandwidth and latency Bandwidth is measured in bytes per second
Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time Data rate when the data stream is actually flowing
Effective bandwidth – average over the entire I/O time, including seek() or locate(), and cartridge switching
Drive's overall data rate
Access latency – amount of time needed to locate data
Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds
Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds
Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk
The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives
A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour

Reliability

A fixed disk drive is likely to be more reliable than a removable disk or tape drive An optical cartridge is likely to be more reliable than a magnetic disk or tape A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed

Cost

Main memory is much more expensive than disk storage The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives

UNIT – V

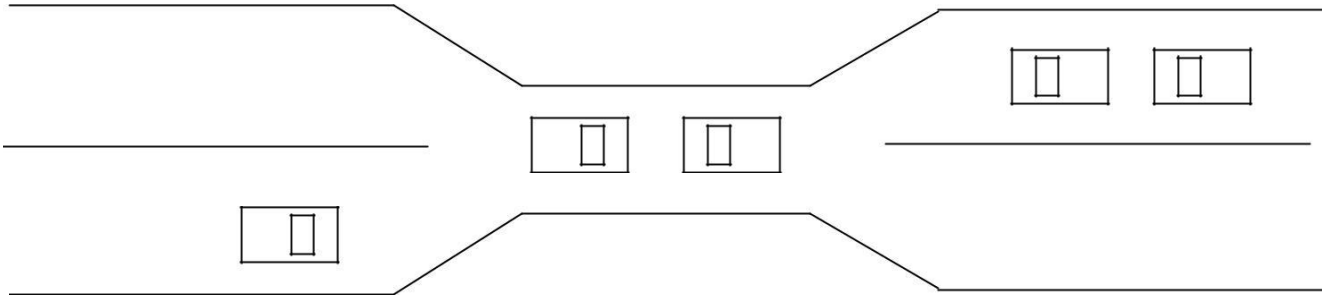
DEADLOCKS

To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
To present a number of different methods for preventing or avoiding deadlocks in a computer system.

The Deadlock Problem
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
Example
System has 2 disk drives
*P*1 and *P*2 each hold one disk drive and each needs another one Example
semaphores *A* and *B*, initialized to 1

*P*0 *P1*
wait (A); wait(B)
wait (B); wait(A)

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model
- Resource types *R*1, *R*2, . . ., *R*m
- *CPU cycles, memory space, I/O devices*

- Each resource type *R*i has *W*i instances. Each process utilizes a resource as follows:

request
use
release

Deadlock Characterization
Deadlock can arise if four conditions hold simultaneously

Mutual exclusion: only one process at a time can use a resource

Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task

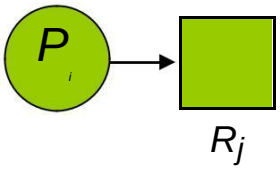
Circular wait: there exists a set {*P*0, *P*1, …, *P*0} of waiting processes such that *P*0 is waiting for a resource that is held by *P*1, *P*1 is waiting for a resource that is held by
*P*2, …, *Pn*–1 is waiting for a resource that is held by
*P*n, and *P*0 is waiting for a resource that is held by *P*0.

Resource-Allocation Graph
A set of vertices *V* and a set of edges
E V is partitioned into two types:
P = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system
R = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system request edge – directed edge *P*1 ® *Rj*
assignment edge – directed edge *Rj* ® *Pi*

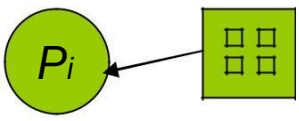
Process

Resource Type with 4 instances

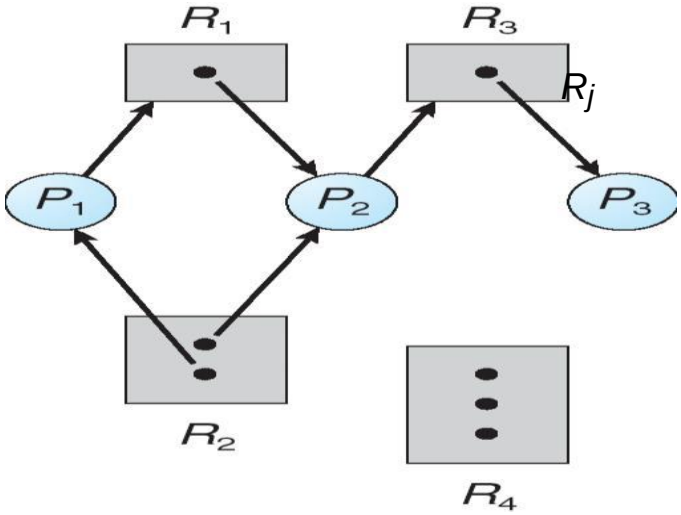
Pi requests instance of *Rj*n



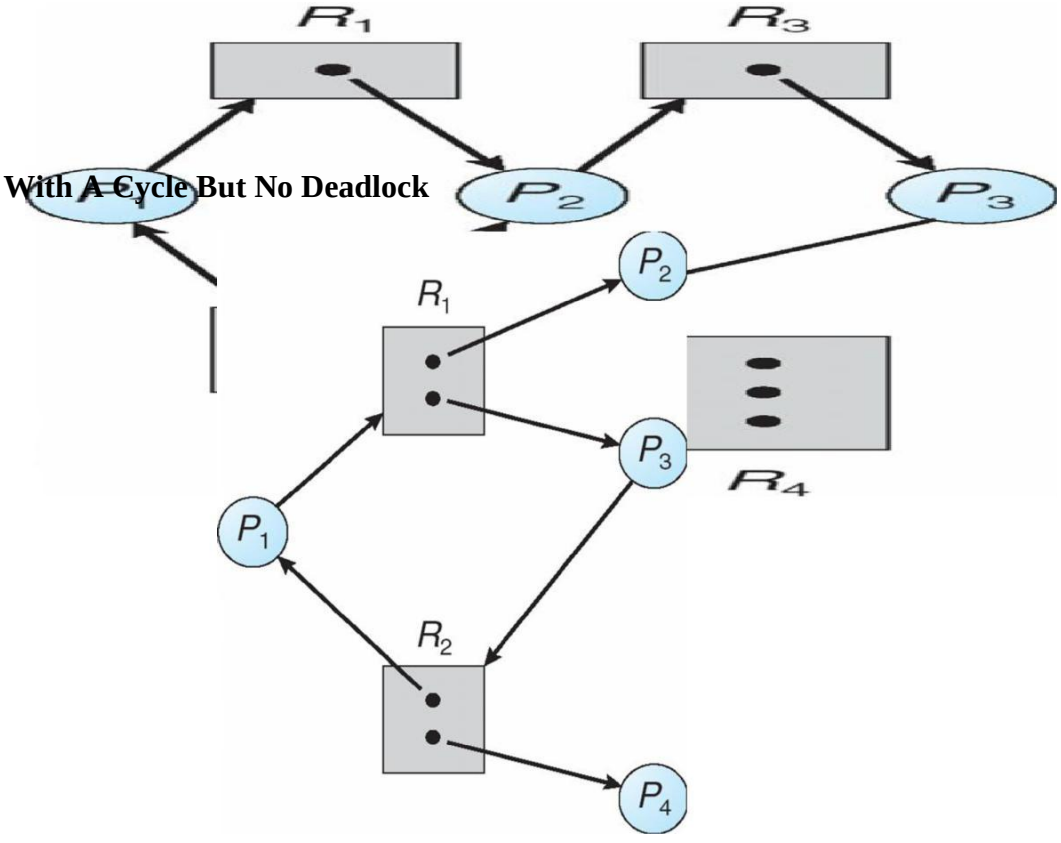
Pi is holding an instance of *Rj*



Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock

Basic Facts
If graph contains no cycles no deadlock If graph contains a cycle if only one instance per resource type, then deadlock
if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks
Ensure that the system will *never* enter a deadlock state Allow the system to enter a deadlock state and then recover Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention
Restrain the ways request can be made

Mutual Exclusion – not required for sharable resources; must hold for non sharable resources

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
Low resource utilization; starvation possible

No Preemption –
If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released Preempted resources are added to the list of resources for which the process is waiting Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

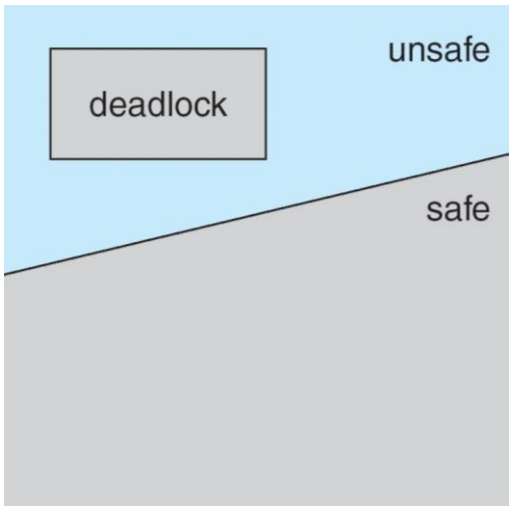
Deadlock Avoidance
Requires that the system has some additional *a priori* information
available
Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State
When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
System is in safe state if there exists a sequence <*P1, P2, …, Pn*> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j* < *i*n That is:
If Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* have finished
When *Pj* is finished, *Pi* can obtain needed resources, execute, return allocated resources, and
terminate When *Pi* terminates, *Pi* +1 can obtain its needed resources, and so on

Basic Facts

If a system is in safe state Þ no deadlocks. If a system is in unsafe state Þ possibility of deadlock Avoidance Þ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



Avoidance algorithms

- Single instance of a resource type
- Use a resource-allocation graph
- Multiple instances of a resource type
- Use the banker's algorithm

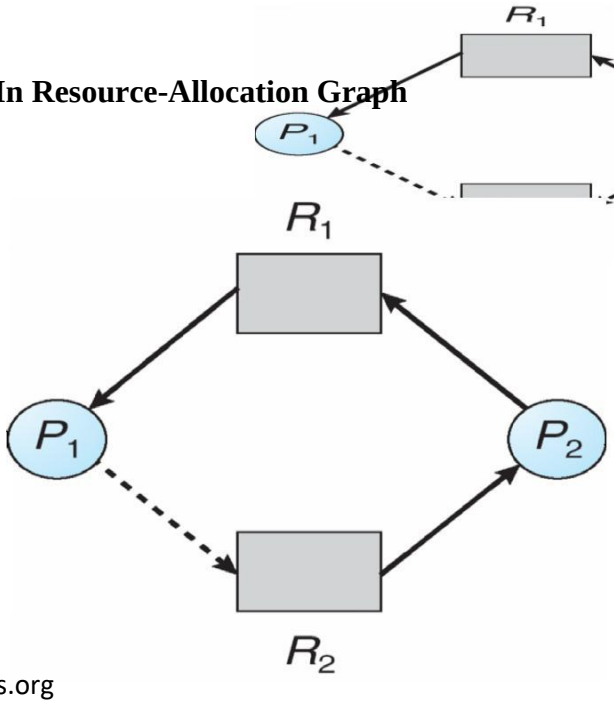
Resource-Allocation Graph Scheme

Claim edge *Pi* ® *Rj* indicated that process *Pj* may request resource *Rj*; represented by a dashed linen Claim edge converts to request edge when a process requests a resource Request edge converted to an assignment edge when the resource is allocated to the process

When a resource is released by a process, assignment edge reconverts to a claim edge Resources must be claimed *a priori* in the system

Resource-Allocation Graph

Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm
Suppose that process *Pi* requests a resource *Rj*
The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm
Multiple instances Each process must a priori claim maximum use When a process requests a resource it may have to wait When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm
Let *n* = number of processes, and *m* = number of resources types.
Available*:* Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type *Rj* available
Max*: n x m* matrix. If *Max* [*i,j*] = *k,* then process *Pi* may request at most *k* instances of resource type *Rj* **Allocation***: n* x *m* matrix. If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj***Need***: n* x *m* matrix. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj* to complete its task

Need [*i,j] = Max*[*i,j*] – *Allocation* [*i,j*]

Safety Algorithm
1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work = Available
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1
2. Find and *i* such that both:
   (a) *Finish* [*i*] = *false*(b) *Needi* £ *Work*
   If no such *i* exists, go to step 4

3. *Work = Work + Allocationi*
   Finish[*i*] = *true*
   go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i
1. Request = request vector for process *Pi*. If *Requesti* [*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*1. If *Requesti* £ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If *Requesti* £ *Available,* go to step 3. Otherwise *Pi* must wait, since resources are not available
3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:
   Available = Available – Request;
   Allocationi = Allocationi + Requesti;
   Needi = Needi – Requesti;
If safe Þ the resources are allocated to Pi
If unsafe Þ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm
n5 processes *P*0 through *P*4;

3 resource types:

A (10 instances), *B* (5instances), and *C* (7 instances)
Snapshot at time *T*0:

	Allocation	*Max*	*Available*	*A B C*	*A B C*	*A B C*
*P*0	0 1 0	7 5 3	*P*1	2 0 0	3 2 2	
*P*2	3 0 2	9 0 2				
*P*3	2 1 1	2 2 2				
*P*4	0 0 2	4 3 3				

The content of the matrix *Need* is defined to be *Max – Allocation*

Need	*A B C*
*P*0	7 4 3
*P*1	1 2 2
*P*2	6 0 0
*P*3	0 1 1
*P*4	4 3 1

The system is in a safe state since the sequence < *P*1, *P*3, *P*4, *P*2, *P*0> satisfies safety criteria

Example: *P*1 Request (1,0,2)

Check that Request £ Available (that is, (1,0,2) £ (3,3,2) Þ true

	Allocation	*Need*	*Available*
	ABC	*ABC*	*ABC*
P0	*010*	*743*	*230*
P1	*302*	*020*	
P2	*301*	*600*	
P3	*211*	*011*	
P4	*002*	*431*	

Executing safety algorithm shows that sequence < *P*1, *P*3, *P*4, *P*0, *P*2> satisfies safety requirement Can request for (3,3,0) by *P*4 be granted?
Can request for (0,2,0) by *P*0 be granted?

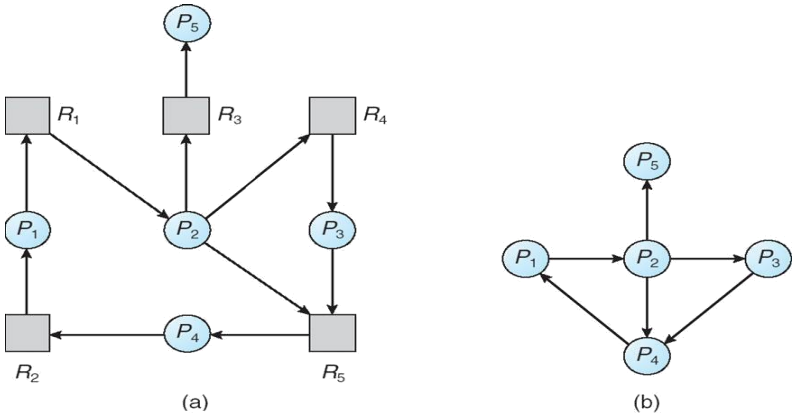
Deadlock Detection

Allow system to enter deadlock state Detection algorithm Recovery scheme

Single Instance of Each Resource Type

Maintain *wait-for* graph
Nodes are processes
Pi ® *Pj* if *Pi* is waiting for *Pj* Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
An algorithm to detect a cycle in a graph requires an order of *n*2 operations, where *n* is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

Available*:* A vector of length *m* indicates the number of available resources of each type. **Allocation***:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. **Request***:* An *n* x *m* matrix indicates the current request of each process. If *Request* [*ij*] = *k*, then process *Pi* is requesting *k* more instances of resource type. *Rj*.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
(a) *Work* = *Available*(b) For *i* = 1,2, …, *n*, if *Allocationi* ¹ 0, then
Finish[i] = false; otherwise, *Finish*[i] = *true*2. Find an index *i* such that both:
(a) *Finish*[*i*] == *false*(b) *Requesti* £ *Work* If no such *i* exists, go to step 4
3. *Work* = *Work* + *Allocationi*
Finish[*i*] = *true*
go to step 24. If *Finish*[*i*] == false, for some *i*, 1 £ *i* £ *n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then *Pi* is deadlocked

Algorithm requires an order of O(*m* x *n*2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

nFive processes *P*0 through *P*4; three resource types
A (7 instances), *B* (2 instances), and *C* (6 instances)
nSnapshot at time *T*0:

	Allocation	*Need*	*Available*
	ABC	*ABC*	*ABC*
P0	*010*	*000*	*000*
P1	*200*	*202*	
P2	*303*	*000*	
P3	*211*	*100*	
P4	*002*	*002*	

Sequence <*P*0, *P*2, *P*3, *P*1, *P*4> will result in *Finish*[*i*] = true for all *i*

*P*2 requests an additional instance of type *C*

Request	*A*	*B*	*C*
*P*0	0	0	0
*P*1	2	0	1
*P*2	0	0	1
*P*3	1	0	0
*P*4	0	0	2

State of system?
Can reclaim resources held by process *P*0, but insufficient resources to fulfill other processes; requests
Deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4

Detection-Algorithm Usage
When, and how often, to invoke depends on:
How often a deadlock is likely to occur?
How many processes will need to be rolled back?
one for each disjoint cycle If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination
Abort all deadlocked processes Abort one process at a time until the deadlock cycle is eliminated. In which order should we choose to abort?
Priority of the process
How long process has computed, and how much longer to completion Resources the process has used
Resources process needs to complete
How many processes will need to be terminated Is process interactive or batch?

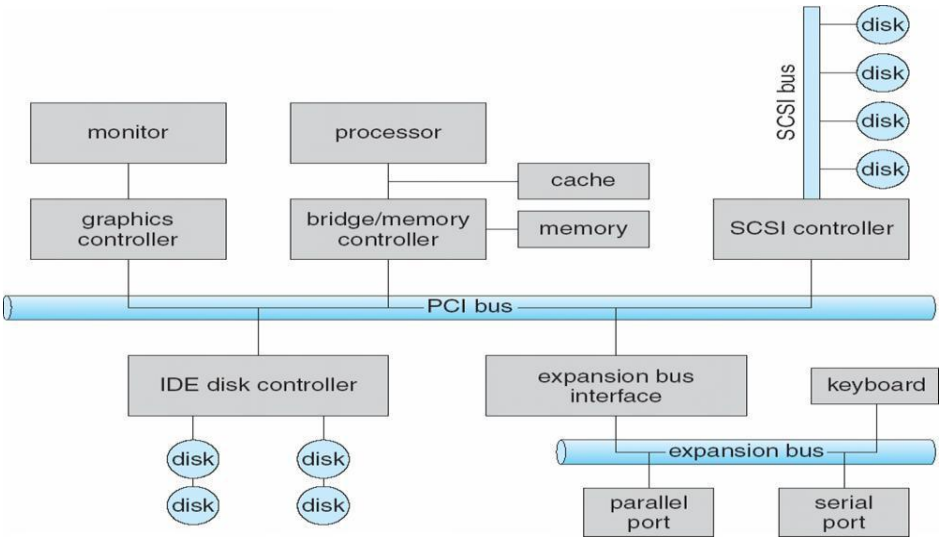
Recovery from Deadlock: Resource Preemption
Selecting a victim – minimize cost Rollback – return to some safe state, restart process for that state Starvation – same process may always be picked as victim, include number of rollback in cost factor

I/O Systems
Explore the structure of an operating system's I/O subsystem
Discuss the principles of I/O hardware and its complexity
Provide details of the performance aspects of I/O hardware and software

I/O Hardware
Incredible variety of I/O devices
Common concepts
Port
Bus (daisy chain or shared direct access)
Controller (host adapter)
I/O instructions control devices
Devices have addresses, used by

Direct I/O instructions
Memory-mapped I/O

A Typical PC Bus Structure



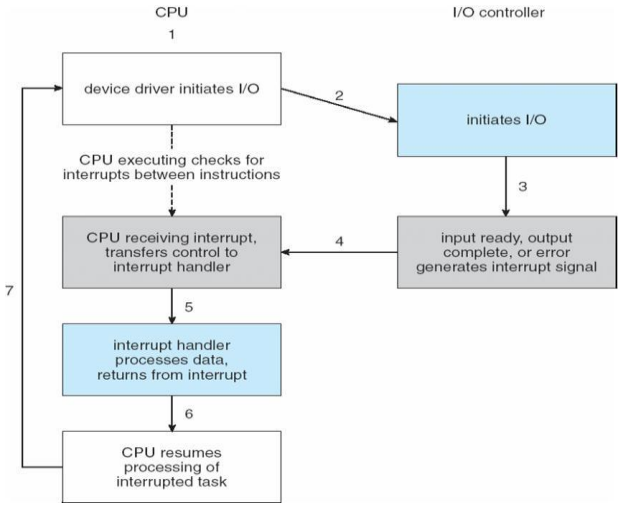
Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling
Determines state of
device command-ready
busy
Error Busy-wait cycle to wait for I/O from device

Interrupts
CPU Interrupt-request line triggered by I/O device Interrupt handler receives interrupts Markable to ignore or delay some interrupts Interrupt vector to dispatch interrupt to correct handler Based on priority Some nonmarkable Interrupt mechanism also used for exceptions

Interrupt-Driven I/O Cycle



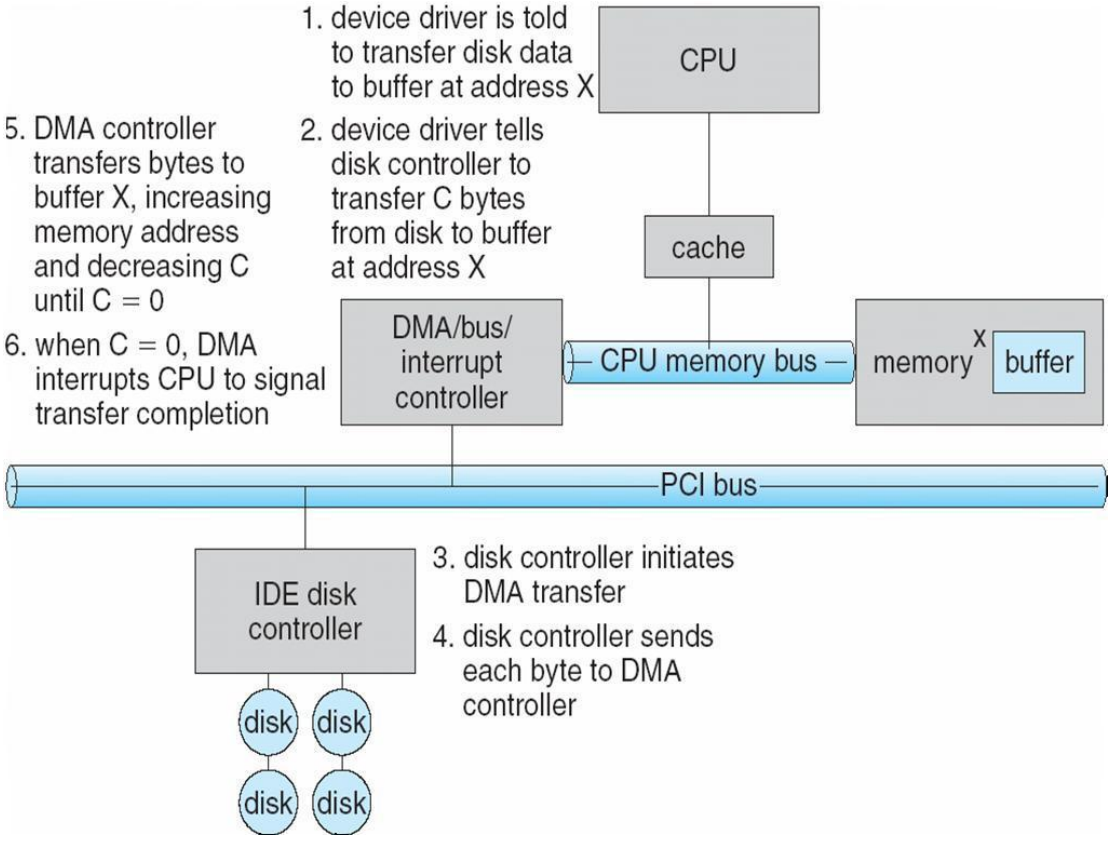
Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Direct Memory Access

Used to avoid programmed I/O for large data movement Requires DMA controller Bypasses CPU to transfer data directly between I/O device and memory.

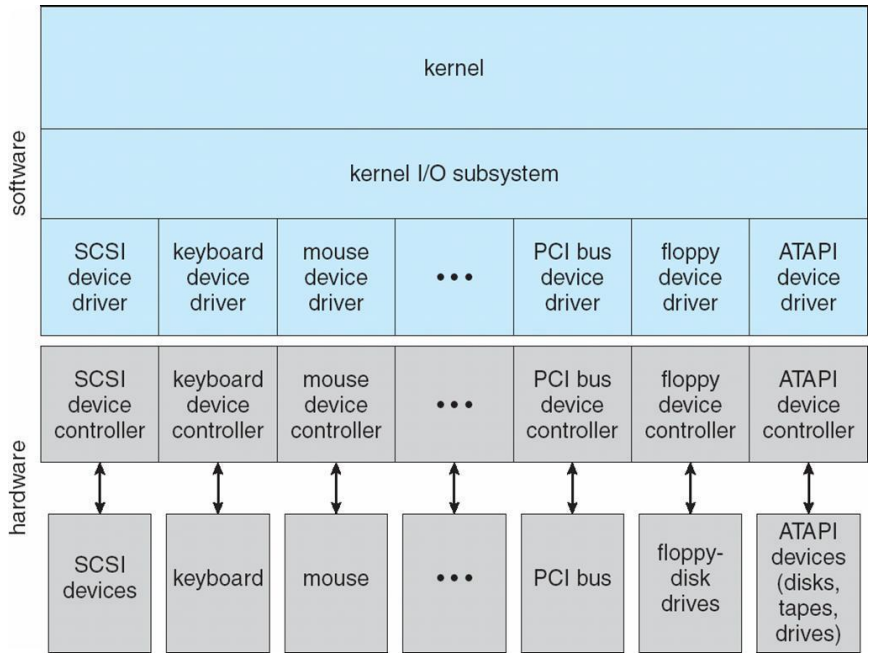
Six Step Process to Perform DMA Transfer



Application I/O Interface

I/O system calls encapsulate device behaviors in generic classes
Device-driver layer hides differences among I/O controllers from kernel Devices vary in many dimensions
Character-stream or block
Sequential or random-access
Sharable or dedicated Speed of operation
read-write**,** read only**,** or write only

A Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

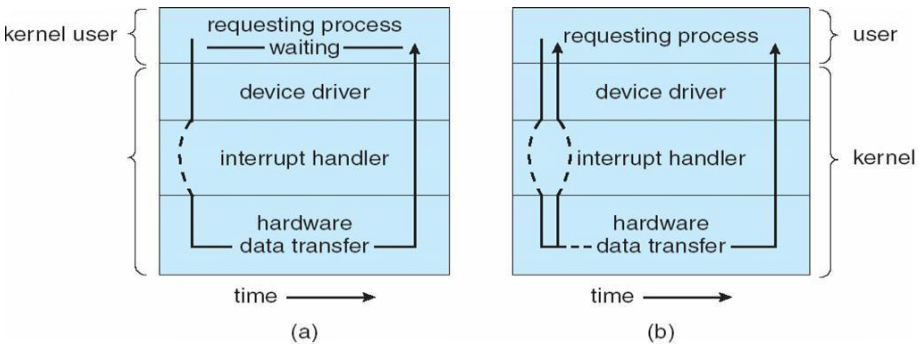
Block and Character Devices
Block devices include disk drives
Commands include read, write, seek
Raw I/O or file-system access
Memory-mapped file access possible Character devices include keyboards, mice, serial ports
Commands include get(), put()Libraries layered on top allow line editing

Network Devices
Varying enough from block and character to have own interface Unix and Windows NT/9*x*/2000 include socket interface
Separates network protocol from network operation
Includes select() functionality Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clocks and Timers
Provide current time, elapsed time, timer Programmable interval timer used for timings,
periodic interruptsnioctl() (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Non blocking I/O

Blocking - process suspended until I/O completed
Easy to use and understand
Insufficient for some needs Nonblocking - I/O call returns as much as
available User interface, data copy (buffered I/O)
Implemented via multi-threading
Returns quickly with count of bytes read or written Asynchronous - process runs while I/O executes Difficult to use
I/O subsystem signals process when I/O completed
Two I/O Methods

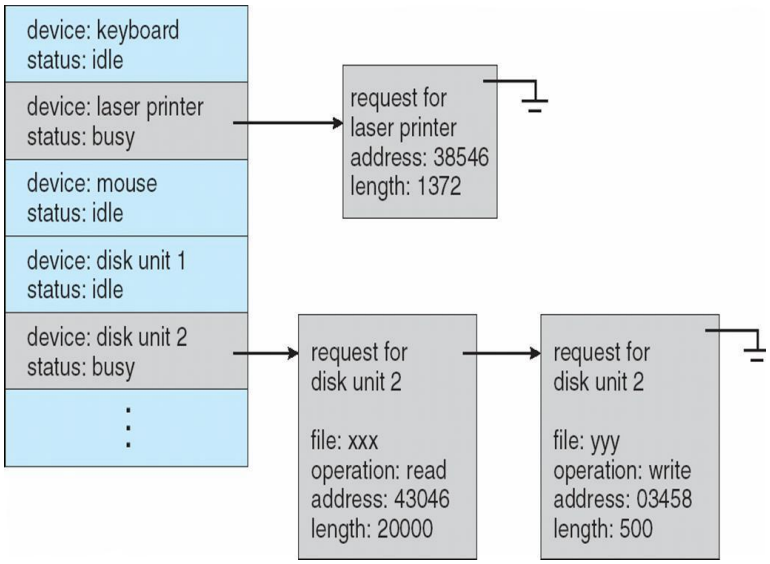


Synchronous Asynchronous

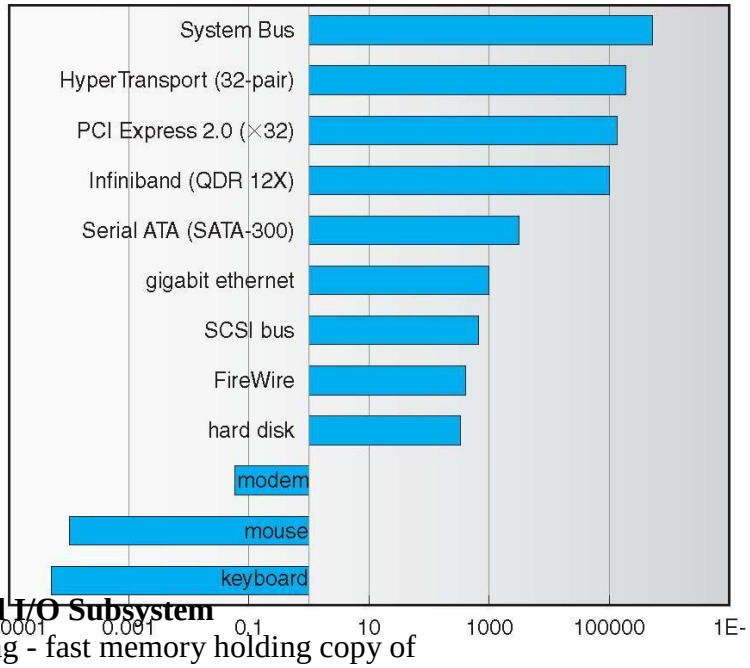
Kernel I/O Subsystem
Scheduling
Some I/O request ordering via per-device queue
Some OSs try fairness Buffering - store data in memory while transferring between devices To cope with device speed mismatch

To cope with device transfer size mismatch
To maintain “copy semantics”

Device-status Table



Sun Enterprise 6000 Device-Transfer Rates



Kernel I/O Subsystem

Caching - fast memory holding copy of
data Always just a copy
Key to performance Spooling - hold output for a device
If device can serve only one request at a time
i.e., Printing Device reservation - provides exclusive access to a device
System calls for allocation and deallocation
Watch out for deadlock

Error Handling

OS can recover from disk read, device unavailable, transient write failures Most return an error number or code when I/O request fails System error logs hold problem reports

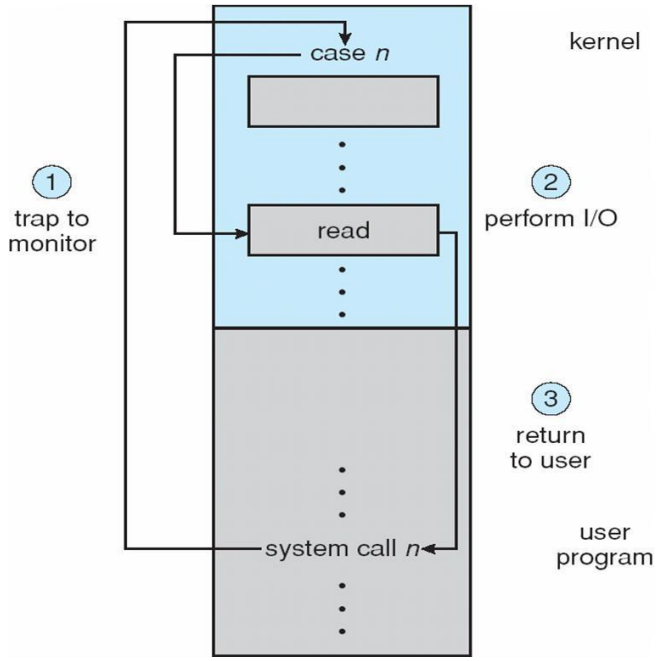
I/O Protection

User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions All I/O instructions defined to be privileged

I/O must be performed via system calls

Memory-mapped and I/O port memory locations must be protected too

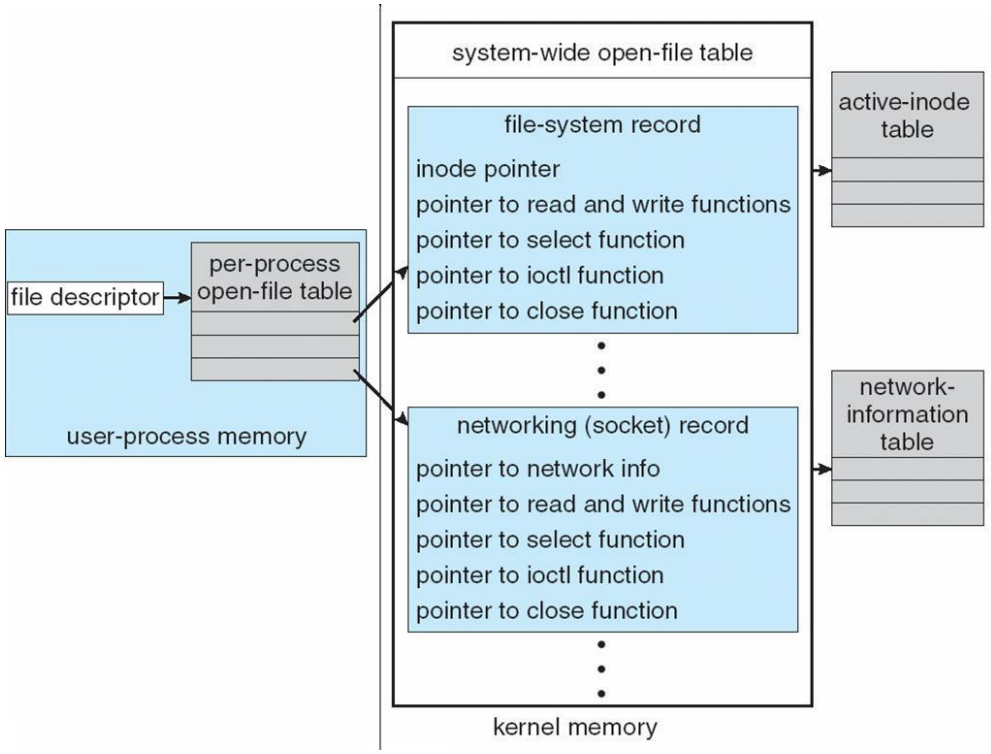
Use of a System Call to Perform I/O



Kernel Data Structures

Kernel keeps state info for I/O components, including open file tables, network connections, character device state Many, many complex data structures to track buffers, memory allocation, “dirty” blocks Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure

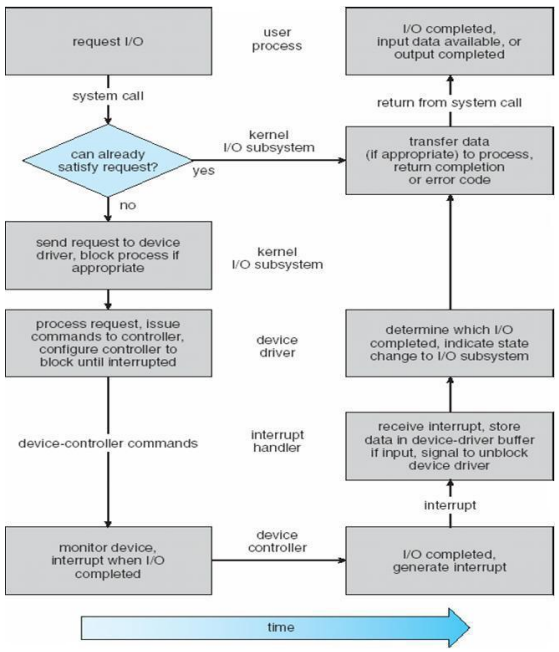


I/O Requests to Hardware Operations

Consider reading a file from disk for a process:

Determine device holding file Translate
name to device representation
Physically read data from disk into buffer
Make data available to requesting process
Return control to process

Life Cycle of An I/O Request

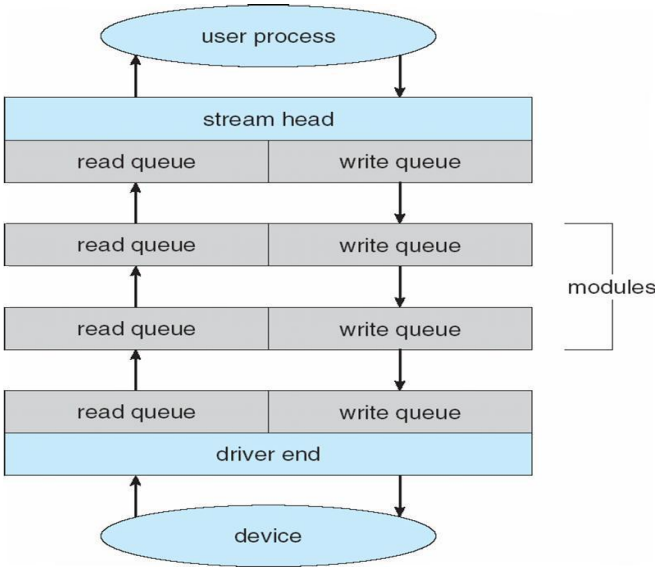


STREAMS

STREAM – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
A STREAM consists of:
- STREAM head interfaces with the user process
- driver end interfaces with the device
- zero or more STREAM modules between them.

Each module contains a read queue and a write queue
Message passing is used to communicate between queues

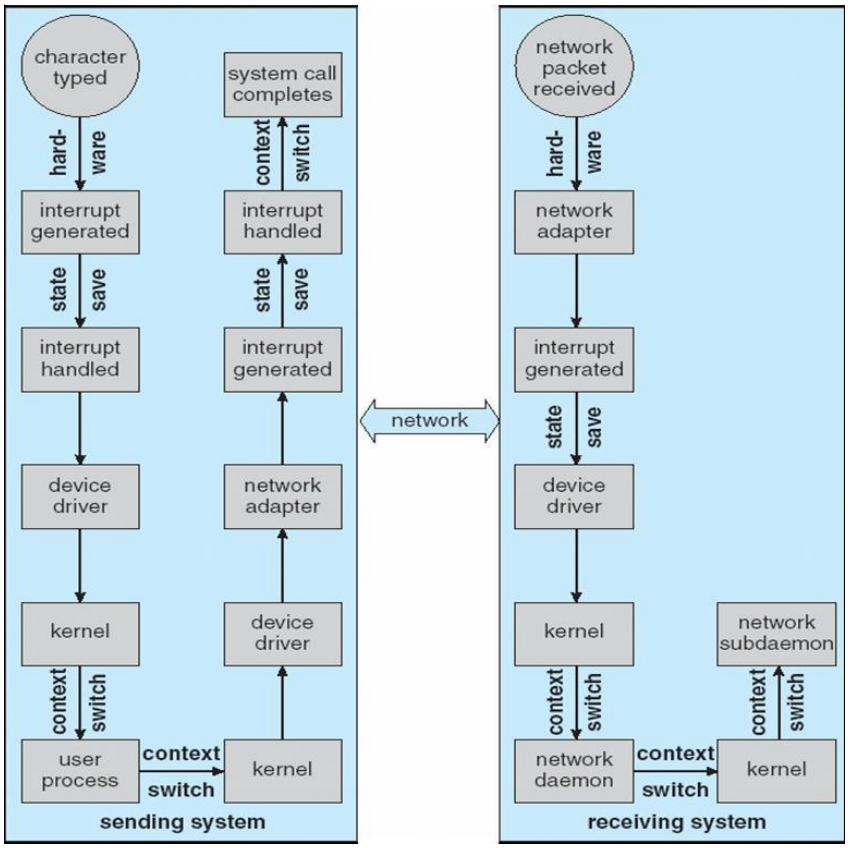
The STREAMS Structure



Performance

I/O a major factor in system performance: Demands CPU to execute device driver, kernel I/O code Context switches due to interrupts
Data copying
Network traffic especially stressful

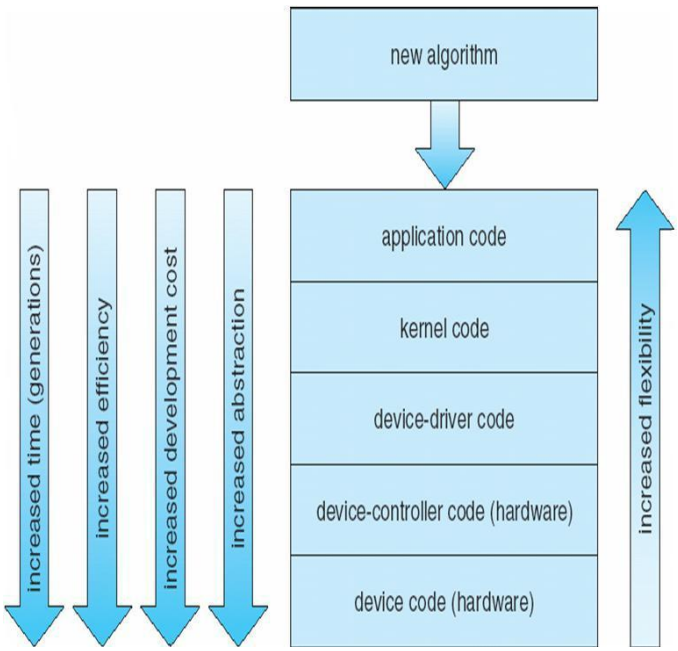
Intercomputer Communications



Improving Performance

Reduce number of context
switches Reduce data copying
Reduce interrupts by using large transfers, smart controllers,
polling Use DMA
Balance CPU, memory, bus, and I/O performance for highest throughput

Device-Functionality Progression



P R O T E C T I O N

Goals of Protection

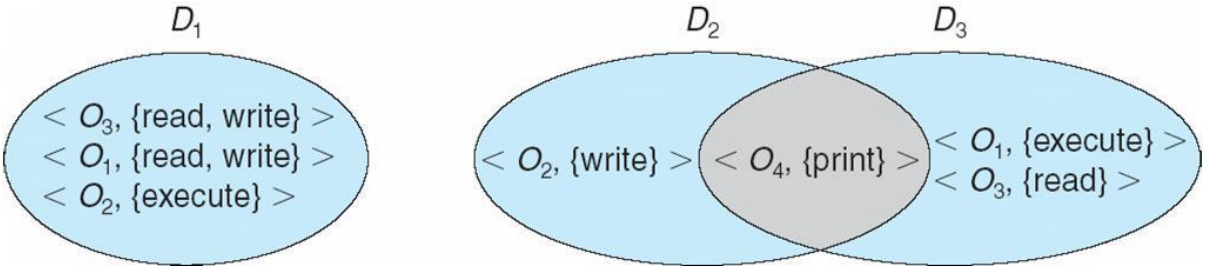
Operating system consists of a collection of objects, hardware or software. Each object has a unique name and can be accessed through a well-defined set of operations Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do son

Principles of Protection

Guiding principle – principle of least privilege
Programs, users and systems should be given just enough privileges to perform their tasks

Domain Structure

Access-right = <*object-name, rights-set*>
where *rights-set* is a subset of all valid operations that can be performed on the object. Domain = set of access-rights



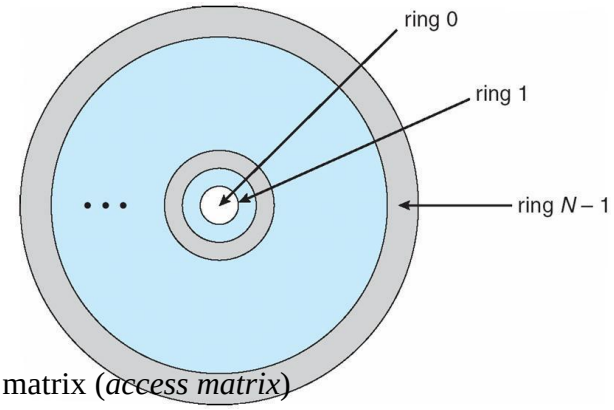
System consists of 2 domains:

- User
- Supervisor UNIX

Domain = user-id
Domain switch accomplished via file system
Each file has associated with it a domain bit (setuid bit)
When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset

Domain Implementation (MULTICS)

Let *Di* and *Dj* be any two domain rings
If *j* < *I* Þ *Di* Í *Dj*



Access Matrix

View protection as a matrix (*access matrix*)
Rows represent domains
Columns represent objects

Access(i, j) is the set of operations that a process executing in Domaini can invoke on Objectj

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

If a process in Domain *Di* tries to do "op" on object *Oj*, then "op" must be in the access matrix Can be expanded to dynamic protection
Operations to add, delete access rights
Special access rights:

- *Owner of Oi*
- *Copy op from Oi to Oj*
- *Control – Di can modify Dj access rights*
- *Transfer – switch from domain Di to Dj*

Access matrix design separates mechanism from policy
Mechanism
Operating system provides access-matrix + rules
If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced

Policy
User dictates policy
Who can access what object and in what mode

Implementation of Access Matrix
Each column = Access-control list for one object
Defines who can perform what operation.

Domain 1 = Read, Write
Domain 2 = Read
Domain 3 = Read

M Each Row = Capability List (like a key)
Fore each domain, what operations allowed on what objects.
Object 1 – Read
Object 4 – Read, Write, Execute

Object 5 – Read, Write, Delete, Copy

Access Matrix of Figure A With Domains as Objects

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access Matrix with *Copy* Rights

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

Access Matrix With *Owner* Rights

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	owner		write

Modified Access Matrix of Figure B

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Access Control

Protection can be applied to non-file resources

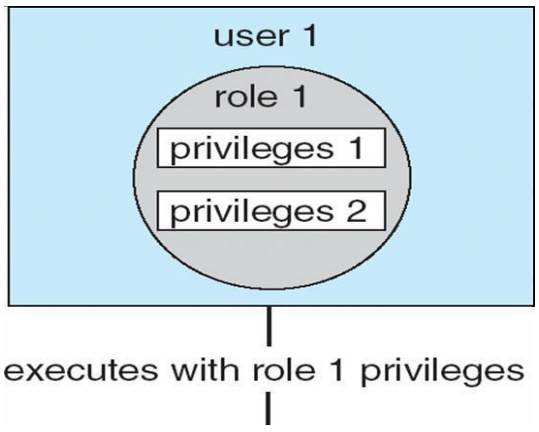
Solaris 10 provides role-based access control **(**RBAC**)** to implement least privilege

Privilege is right to execute system call or use an option within a system call

Can be assigned to processes

Users assigned roles granting access to privileges and programs

Role-based Access Control in Solaris 10



Revocation of Access Rights
Access List – Delete access rights
from access list Simple
Immediate Capability List – Scheme required to locate capability in the system before capability can be revoked Reacquisition

Capability-Based Systems
Hydra
Fixed set of access rights known to and interpreted by the system
Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights Cambridge CAP System
Data capability - provides standard read, write, execute of individual storage segments associated with object Software capability -interpretation left to the subsystem, through its protected procedures

Language-Based Protection
Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

Protection in Java 2
nProtection is handled by the Java Virtual Machine (JVM)nA class is assigned a protection domain when it is loaded by the JVMnThe protection domain indicates what operations the class can (and cannot) performnIf a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library

Stack Inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
			open(Addr a):
			...
			checkPermission
			(a, connect);
			connect (a);