OPERATING SYSTEMS

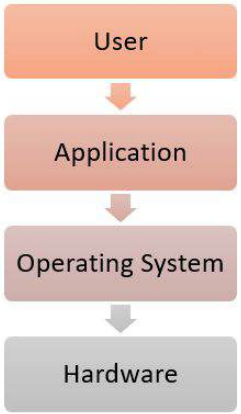
Introduction, Computer System Organization

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

General Definition

- An Operating System is a program that acts as an intermediary between a user of a computer and the computer hardware
- It provides a user-friendly environment in which a user may easily develop and execute programs. Otherwise, hardware knowledge would be mandatory for computer programming.

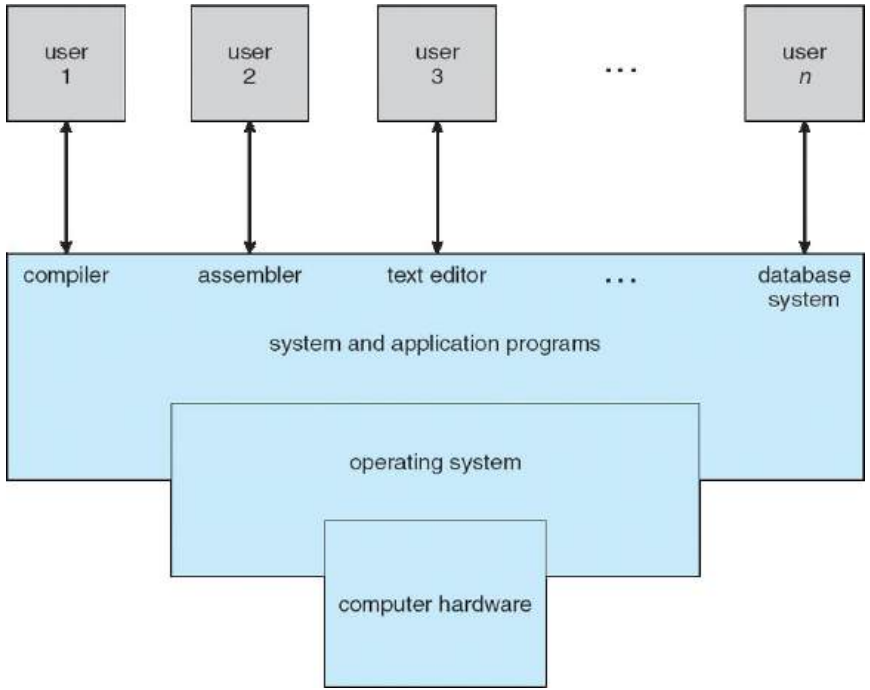


Operating System Goals

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner
- Manage resources such as
    - Memory
    - Processor(s)
    - I/O Devices

OPERATING SYSTEMS

Four Components of a Computer System (Abstract view)



OPERATING SYSTEMS

Computer System Structure

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

What Operating Systems Do

- Depends on the point of view user and system
- Users want convenience, **ease of use** and **good performance**
  - Don’t care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy.
  - Maximize resource utilization.
  - Available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share

What Operating Systems Do

- Users of dedicated systems such as workstations have dedicated resources but frequently use shared resources from servers.
    - resources like file, compute, and print servers are shared.
    - operating system is designed to compromise between individual usability and resource utilization
- Handheld computers are resource poor, optimized for usability and battery life.
- Some computers have little or no user interface, such as embedded computers in devices and automobiles

System View

- 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

OPERATING SYSTEMS

Defining Operating System

- The OS has many roles and functions
- The fundamental goal of computer systems is to execute user programs and to make solving user problems easier.
- The common functions of controlling and allocating resources are then brought together into one piece of software: the **operating system**
- "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 System Organization

- Computer-system consists of,
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - The CPU and the device controllers can execute concurrently, competing for memory cycles.
    - memory controller is provided to synchronize access to the memory.

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
- Each device controller has registers for action (like “read character from keyboard”) to take
- 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

OPERATING SYSTEMS

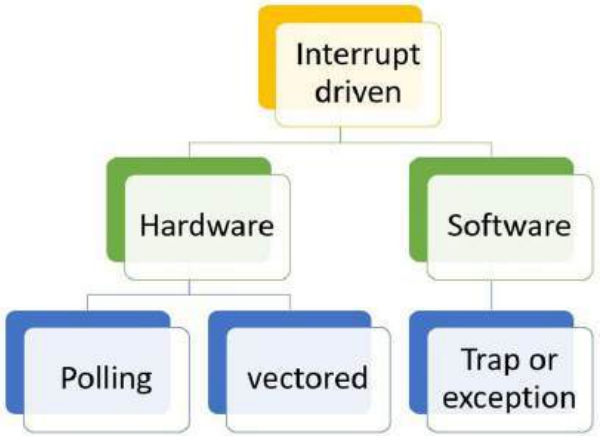
Computer System Organization

- When the system is booted, the first program that starts running is a Boostrap.
- It is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM).
- Bootstrap is known by the general term firmware, within the computer hardware.
- It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
- The bootstrap program must know how to load the operating system and how to start executing that system.
- The bootstrap program must locate and load into memory the operating system kernel.
- The first program that is created is **init**, after the OS is booted. It waits for the occurrence of event.

OPERATING SYSTEMS

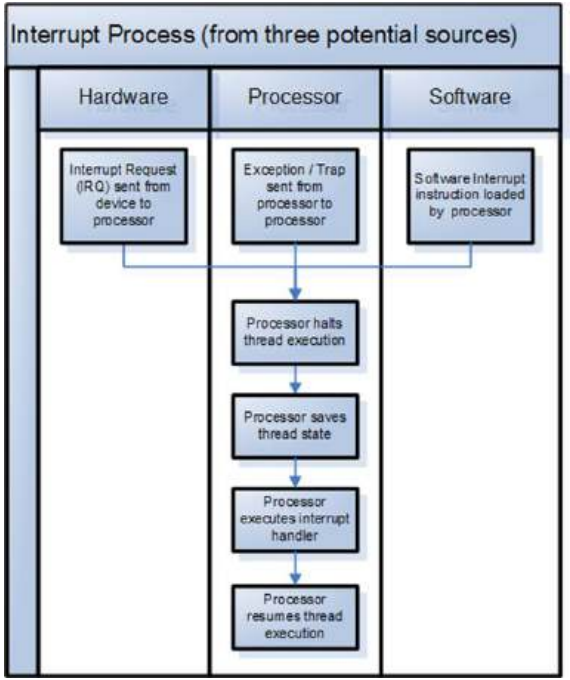
Common Functions of Interrupts

- An operating system is **interrupt driven**
- 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
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request

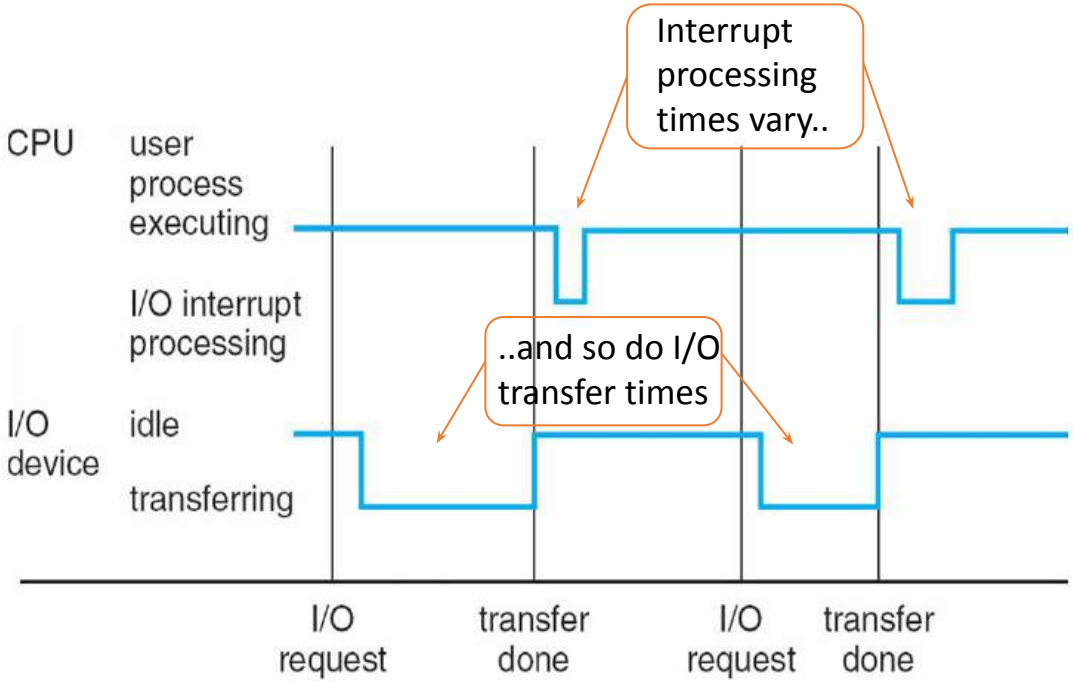


Interrupt Handling

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



Interrupt Timeline for a single process doing output

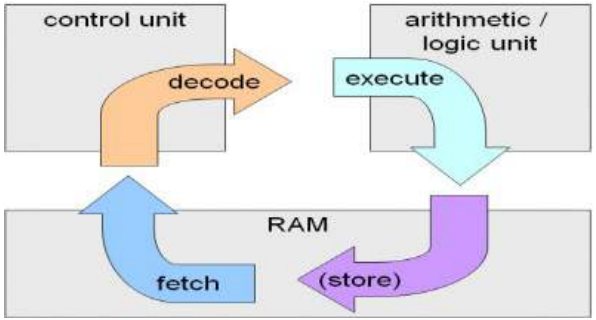


Storage Structure

- Main memory – only large storage media that the CPU can access directly (**Random access memory** and typically **volatile)**
  - Implemented with semiconductor technology called DRAM
- Computers use other forms of memory like ROM, EEPROM
- Smart phones have EEPROM to store factory installed programs.

Program Execution Model

- Typical instruction execution



- The processor fetches instructions from memory, decodes and executes them.
- The Fetch, Decode and Execute cycles are repeated until the program terminates.
- This is called the Von Neumann model of computing.

Image source: http://computersciencewiki.org/index.php/The_machine_instruction_cycle

OPERATING SYSTEMS

Storage Structure

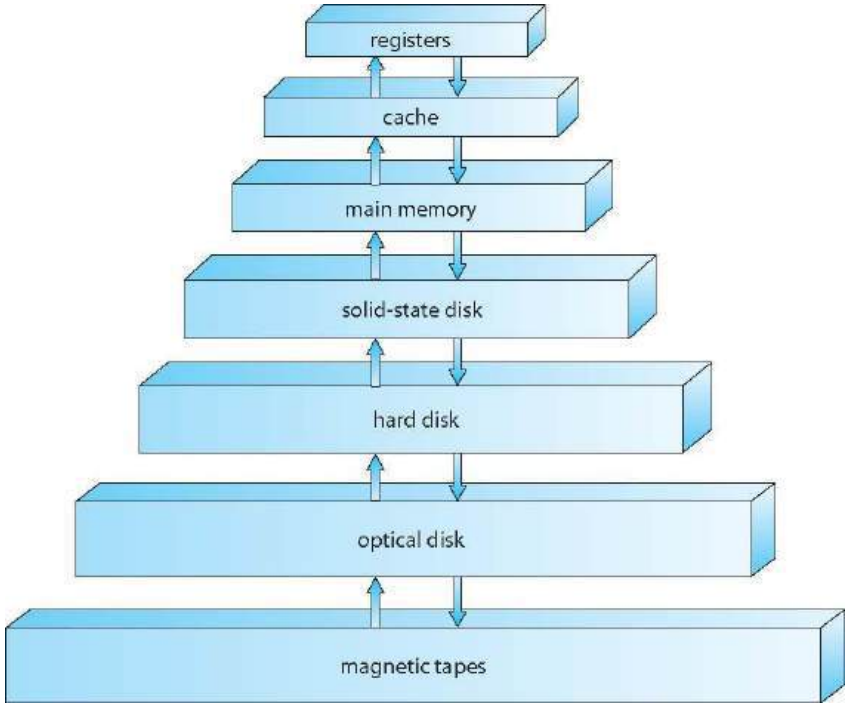
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- Hard disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Solid-state disks** – faster than hard disks, nonvolatile
  - Various technologies and becoming more popular
  - Flash memory used in camera's PDA's

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage

OPERATING SYSTEMS

Storage-Device Hierarchy



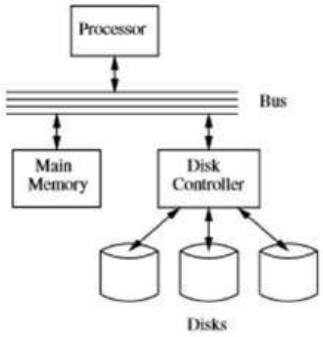
Caching

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

OPERATING SYSTEMS

I/O Structure

- Storage is a type of I/O device
- A large portion of operating-system code is dedicated to managing I/O
    - As reliability and performance of a system is the main concern.
- General-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each device controller is in charge of a specific type of device.
- **Device Driver** for each device controller to manage I/O
    - Provides uniform interface between controller and kernel
- **Small Computer-Systems Interface (SCSI)** controller enables to connect more devices.



I/O Structure

- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices.

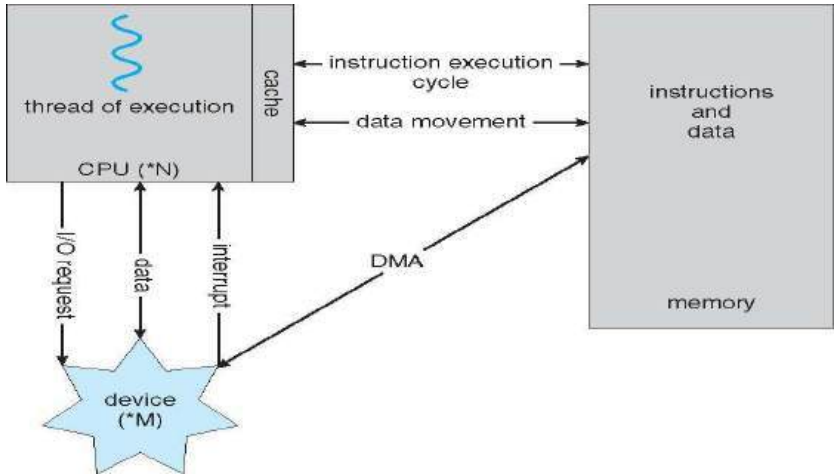
OPERATING SYSTEMS

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 OS 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
  - OS 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



Computer-System Architecture

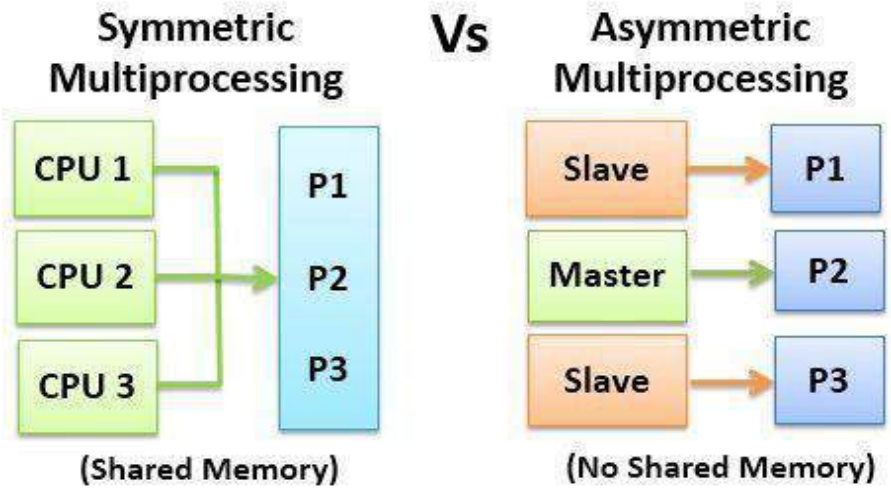
- Most systems use a single general-purpose processor
  - Most systems have other special-purpose processors as well.
  - Device specific processors like disk, keyboard, graphic controller
  - Special-purpose processors run a limited number of instructions
  - Special-purpose processors are low-level components built into the hardware
  - Managed by OS.
  - OS monitors the status.
- Example Disk controller microprocessor
  - Receives sequence of requests from CPU.
  - Implements its own disk queue and scheduling algorithm
  - Relieves the main CPU of the overhead of disk scheduling.

Computer-System Architecture

- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled** **systems**
  - Advantages include:
    - **Increased throughput**
    - **Economy of scale**
    - **Increased reliability** – graceful degradation or fault tolerance

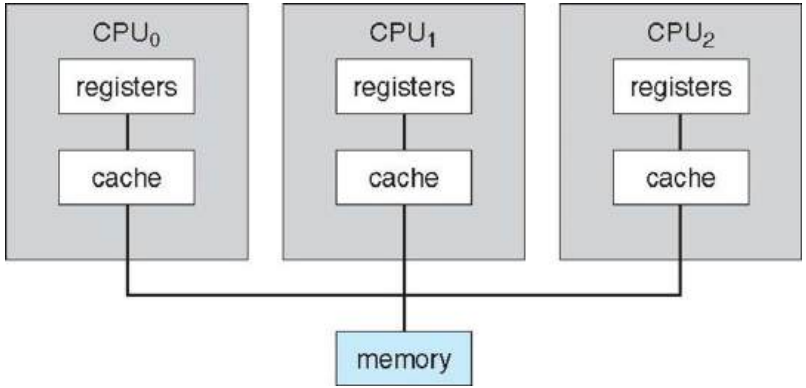
Two types of Multiprocessor Systems

1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
2. **Symmetric Multiprocessing** – each processor performs all tasks



Symmetric Multiprocessing Architecture

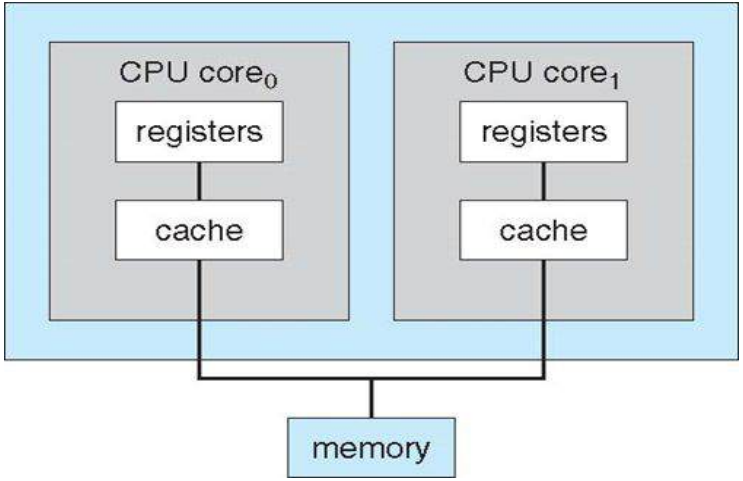
- In SMP all processors are peers; no boss–worker relationship exists between processors.
- Each processor has its own set of registers, as well as a private or local cache.
- All processors share physical memory.



OPERATING SYSTEMS

Multi-Core Design

- A recent trend in CPU design is to include multiple computing cores on a single chip. Such multiprocessor systems are termed **multicore**.
- More efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.
- One chip with multiple cores uses significantly less power than multiple single-core chips.



A dual-core design with two cores placed on the same chip

Command to know the number of cores, cache details
$cat /proc/cpuinfo|more

Blade servers

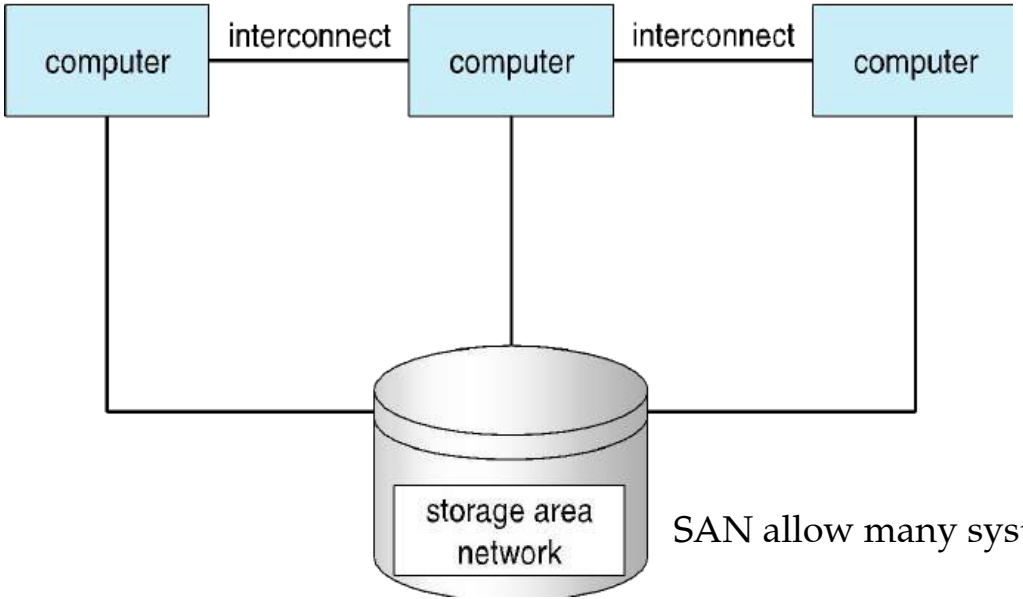
- **Blade servers** are a recent development in which multiple processor boards, I/Oboards, and networking boards are placed in the same chassis.
  - blade-processor board boots independently and runs its own operating system.
  - Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers.
  - In essence, these servers consist of multiple independent multiprocessor systems.

OPERATING SYSTEMS

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**
  - Some have **distributed lock manager** (**DLM**) to avoid conflicting operations (Ex: when multiple hosts access the same data on shared storage)

Clustered Systems

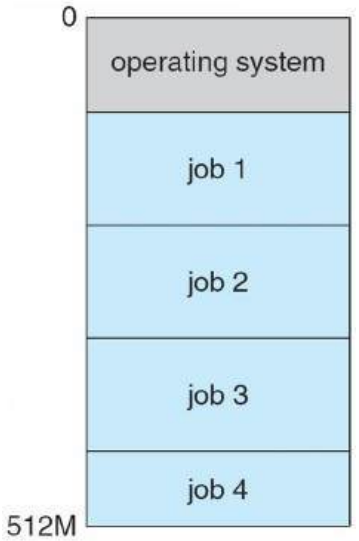


SAN allow many systems to attach to a pool of storage

General structure of a clustered system.

Operating-System Structure - Multiprogramming

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



Operating-System Structure - Multitasking

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

Operating-System Operations

- **Interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap):**
    - Software error (e.g., division by zero)
    - Request for operating system service
    - Other process problems include infinite loop, processes modifying each other or the operating system

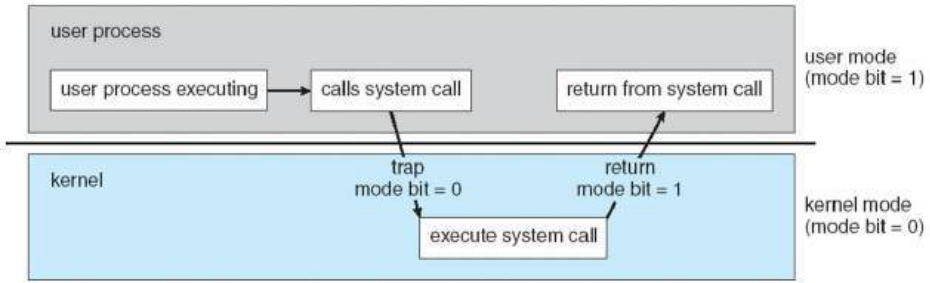
OPERATING SYSTEMS

Dual-Mode and Multimode Operation

- **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
- Increasingly CPUs support multi-mode operations
  - i.e. **virtual machine manager** (**VMM**) mode for guest **VMs**

Transition from user to kernel mode

- When a trap or interrupt occurs, hardware switches from user mode to kernel mode (changes the state of the mode bit to 0).
- When the request is fulfilled, the system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.



Timer

- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after a specified period (fixed 1/60 sec or variable 1 msec to 1 sec)
  - A **variable timer** is generally implemented by a fixed-rate clock and a counter.
  - Operating system sets the counter (privileged instruction)
  - Every time the clock ticks, the counter is decremented.
  - When counter reaches zero, an interrupt occurs
  - Timer can be used to prevent a user program from running too long (terminate the program)

Array:

- An array is a simple data structure in which each element can be accessed directly.
- Main Memory constructed with array.
- How the data is accessed?
- Items with multiple bytes are accessed as item number × item size
- But what about storing an item whose size may vary?
- what about removing an item if the relative positions of the remaining items must be preserved?

Lists

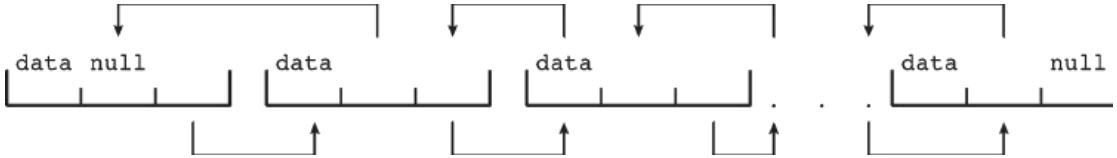
- Standard programming data structures are used extensively in OS

Singly linked list

- The items in a list must be accessed in a particular order.
- common method for implementing this structure is a linked list

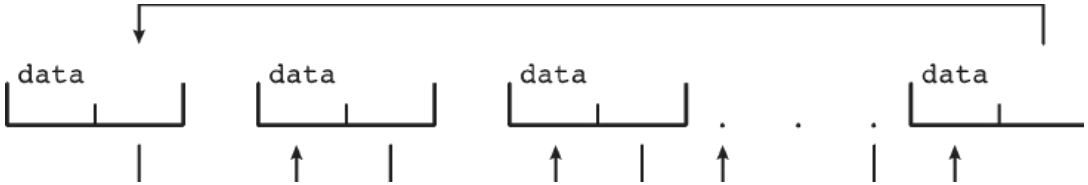
- In a **singly linked list,** each item points to its successor.

Doubly linked list



In a **doubly linked list**, a given item can refer either to its predecessor or to its successor.

Circular linked list



In a **circularly linked list**, the last element in the list refers to the first element, rather than to null.

Lists

Advantages:

- Linked lists accommodate items of varying sizes.
- Allow easy insertion and deletion of items

Disadvantages:

- Performance for retrieving a specified item in a list of size n is linear — O(n) , as it requires potentially traversing all n elements in the worst case.

Usage:

- Lists are used by the some of the kernel algorithms
- Constructing more powerful data structures such as stacks and queues

Stack - a sequentially ordered data structure that uses **LIFO** principle for adding and removing items

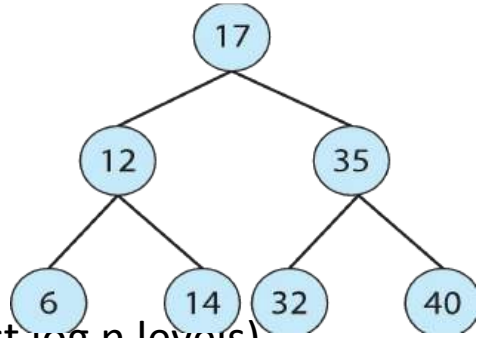
- OS often uses a stack when involving function calls.
- Parameters, local variables and the return address are **push**ed onto the stack when a function is called
- Return from the function call **pop**s those items off the stack

Queue - a sequentially ordered data structure that uses **FIFO** principle for adding and removing items

- Tasks waiting to be run on an available CPU are organized in queues
- Print jobs sent to a printer are printed in the order of submission

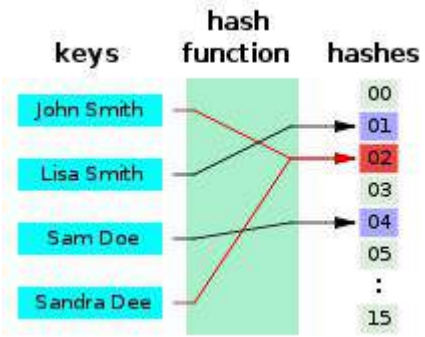
Trees

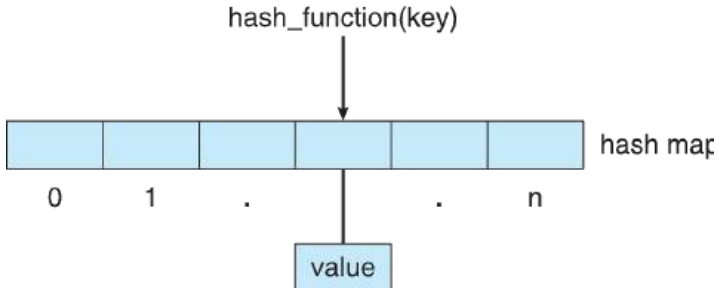
- Data structure used to represent data hierarchically.
- Data values in a tree structure are linked through parent–child relationships
- **Binary search tree**
  - ordering between 2 children: left <= right
  - Search performance is *O(n)*
- **Balanced binary search tree –** (a tree containing n items has at most log n levels)
  - Search performance is *O(log n)*
  - Used by Linux for selecting which task to run next (CPU-Scheduling algorithm)



Hash Functions and Maps

- Hash functions can result in the same output value for 2 inputs
- **Hash function** can be used to implement a **hash map**
  - Maps or associates key:value pairs using a hash function
  - Search performance is *O(1)*





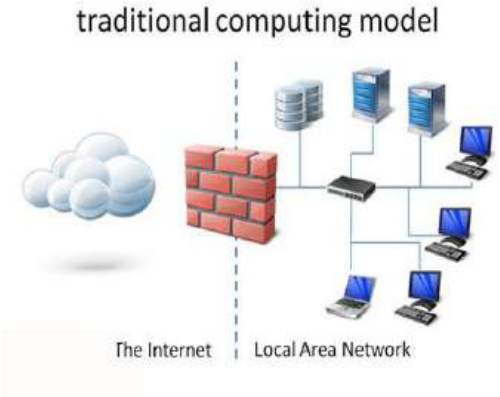
Bitmaps

Bitmap - string of *n* binary digits representing the status of *n* items

- Availability of each resource is indicated by the value of a binary digit
  - 0 – resource is available
  - 1 – resource is unavailable
- Value of the **i**th position in the bitmap is associated with the **i**th resource
  - Example: bitmap 001011101 shows resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available
- Commonly used to represent the availability of a large number of resources (disk blocks)

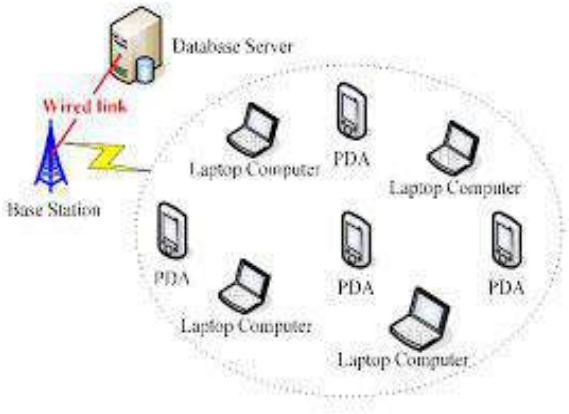
Computing Environments – Traditional

- Stand-alone general purpose machines
- But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers** (**thin clients**) are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks



Computing Environments – Mobile

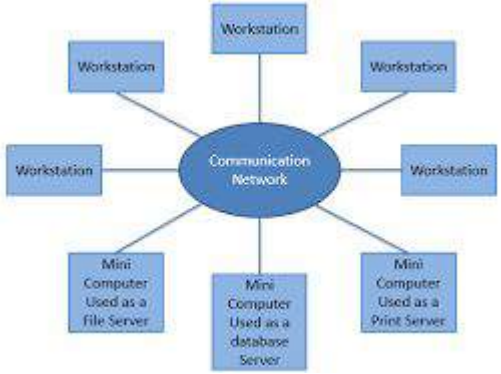
- Handheld smartphones, tablets, etc
- What is the functional difference between them and a "traditional" laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**



Computing Environments – Distributed

Distributed computing

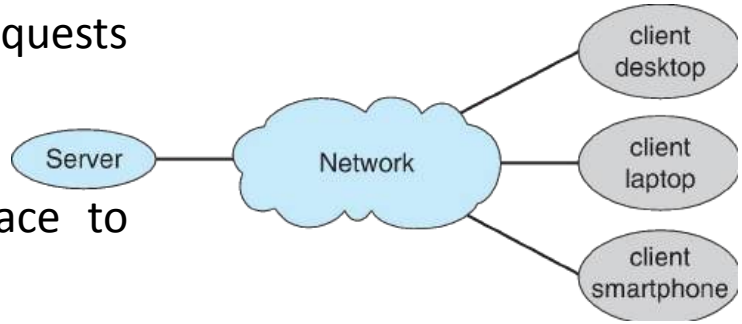
- Collection of separate, possibly heterogeneous systems networked together
  - **Network** is a communications path, **TCP/IP** most common
    - **Local Area Network** (**LAN**)
    - **Wide Area Network** (**WAN**)
    - **Metropolitan Area Network** (**MAN**)
    - **Personal Area Network** (**PAN**)



- **Network Operating System** provides features between systems across network
  - Communication scheme allows systems to exchange messages
  - Illusion of a single system

Client-Server Computing

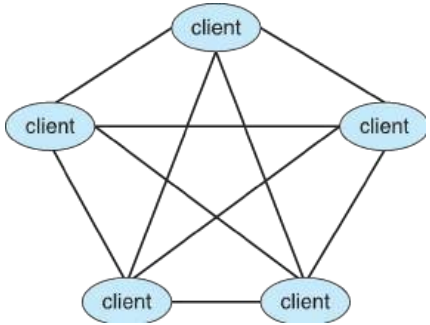
- Dumb terminals replaced by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
  - **Compute-server system** provides an interface to client to request services (i.e., database)
  - **File-server system** provides interface for clients to store and retrieve files



OPERATING SYSTEMS

Computing Environments – Peer-to-Peer

- 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, **Voice over IP** (**VoIP**) such as Skype

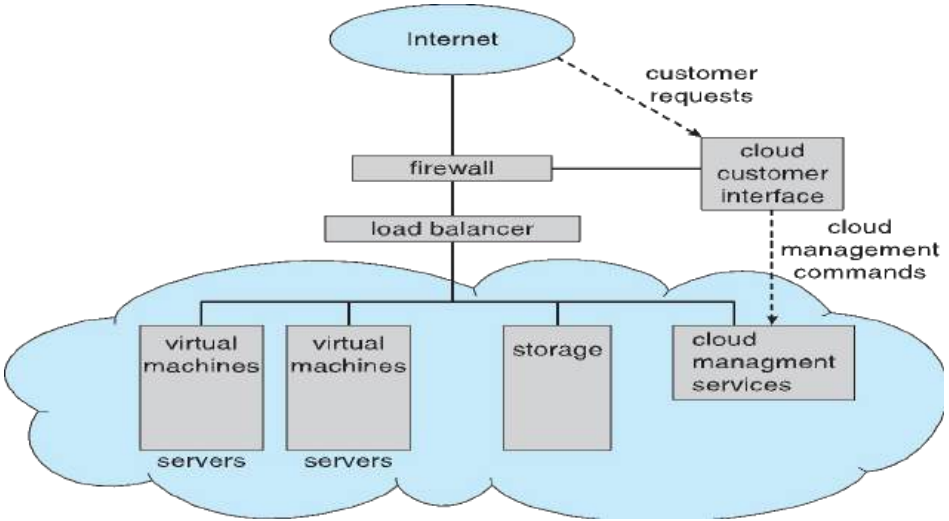


Computing Environments – Virtualization

- Allows operating systems to run applications within other OSes
  - Vast and growing industry
- **Emulation** used when source CPU type different from target type (i.e. PowerPC to Intel x86)
  - Generally slowest method
  - When computer language not compiled to native code – **Interpretation**
- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
  - Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
  - **VMM** (virtual machine Manager) provides virtualization services

Computing Environments – Cloud Computing

- Cloud computing environments composed of traditional OSes, plus VMMs, plus cloud management tools
  - Internet connectivity requires security like firewalls
  - Load balancers spread traffic across multiple applications



Computing Environments – Cloud Computing

- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage
- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components

Computing Environments – Cloud Computing

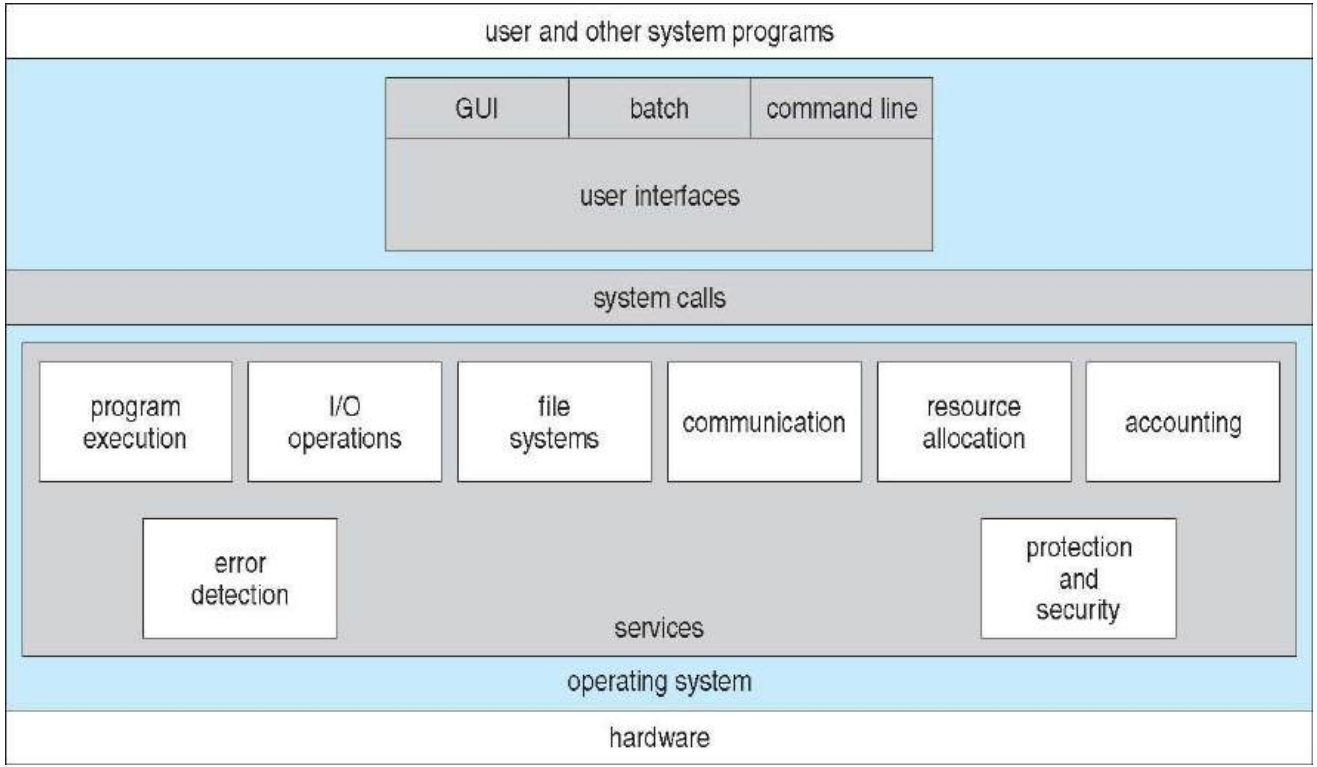
- Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
- Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
- Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

Computing Environments – Real-Time Embedded Systems

- Real-time embedded systems most prevalent form of computers
  - Vary considerable, special purpose, limited purpose OS, **real-time OS**
  - Use expanding
- Many other special computing environments as well
  - Some have OSes, some perform tasks without an OS
- Real-time OS has well-defined fixed time constraints
  - Processing ***must*** be done within constraint
  - Correct operation only if constraints met

OPERATING SYSTEMS

A View of Operating System Services



OPERATING SYSTEMS

Services

- Operating systems provide an environment for execution of programs and services to programs and users
- Set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - **Command-Line (CLI) -** Command interpreters (shells)
    - **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. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)

Services

- **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

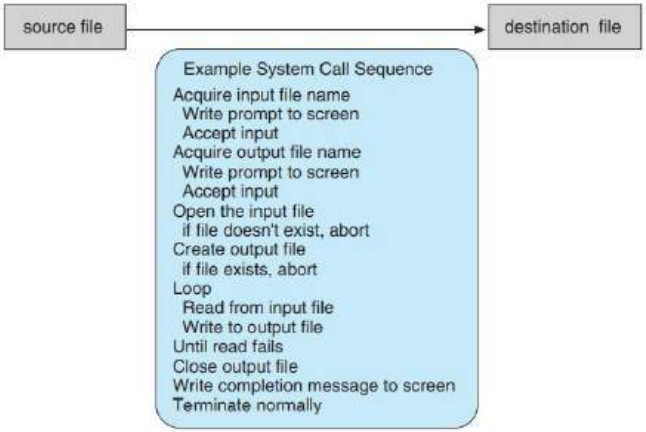
Services

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources

- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

System calls

- **System calls** provide an interface to the services made available by an operating system.
- These calls are generally available as routines written in a higher level programming language or assembly-language
- Example: Sequence of system calls used for copying contents from one file to another



Types of System calls

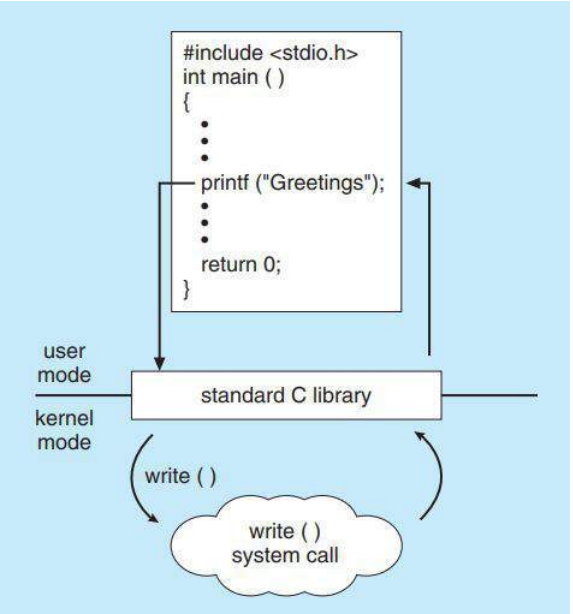
System calls can be grouped roughly into six major categories:

1. Process control
2. File manipulation
3. Device manipulation
4. Information maintenance
5. Communications
6. Protection

	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() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Example of a system call

- A C program invokes the printf() statement
- The C library intercepts this call and invokes the necessary system call **write()** in the operating system
- The C library takes the value returned by write() and passes it back to the user program



Design goals

- Start the design by defining goals and specifications
- Affected by choice of hardware and the type of system
- Requirements can be categorized as **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

Operating-System Design and Implementation

- Important principle to separate

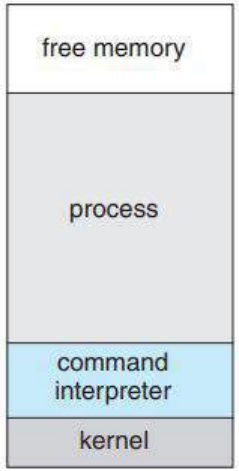
  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?

- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer to prevent a user program from running too long)
- Specifying and designing an OS is highly creative task of **software engineering**

OPERATING SYSTEMS

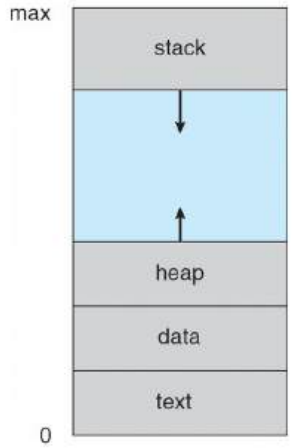
Process Concept

- 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
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

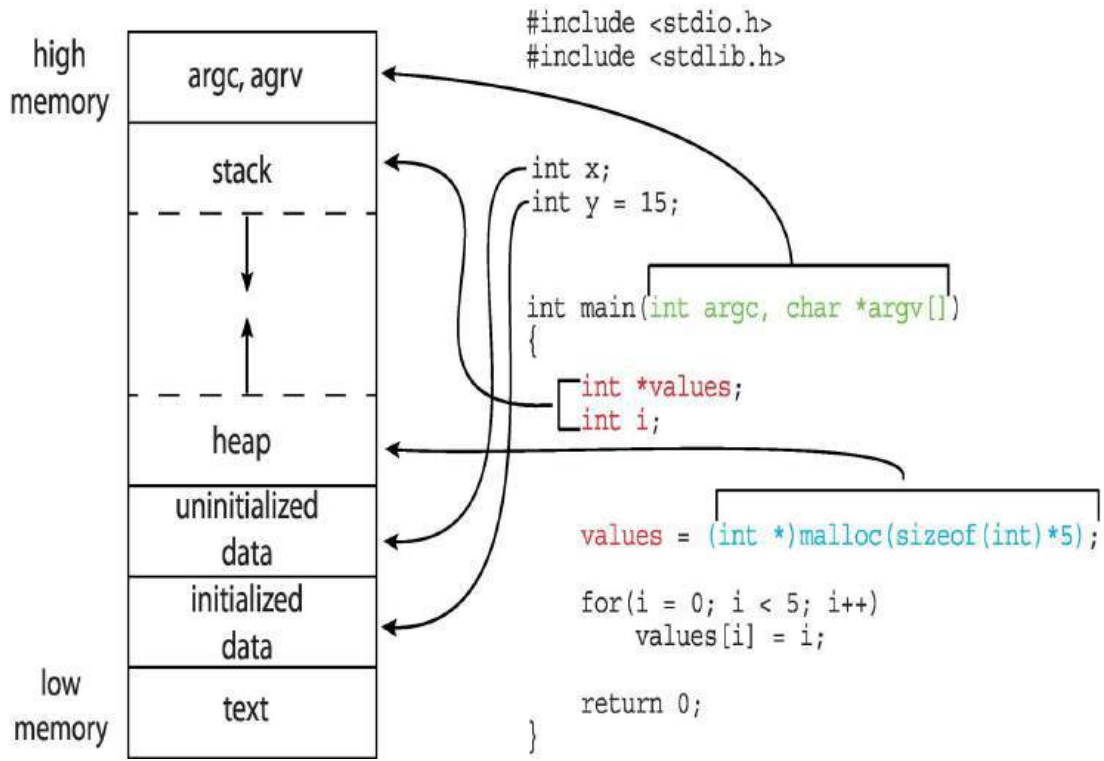


Structure of a process in memory

- The program code, also called **text section.**
  - Includes current activity including **program counter**, processor registers
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



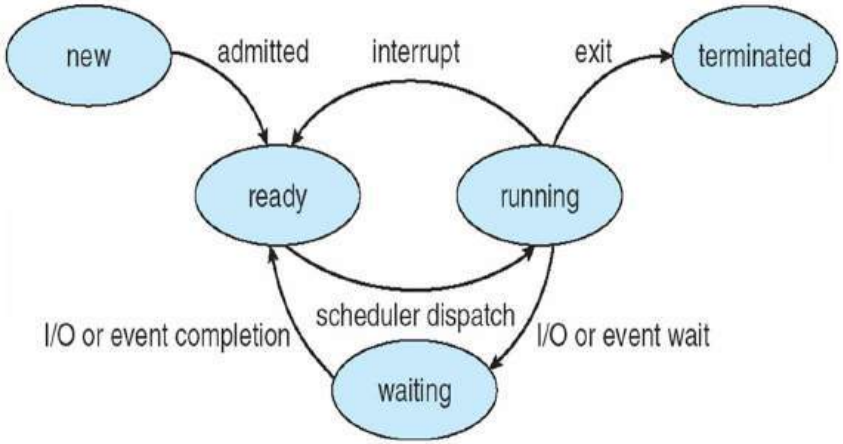
Memory Layout of a C Program



Process State

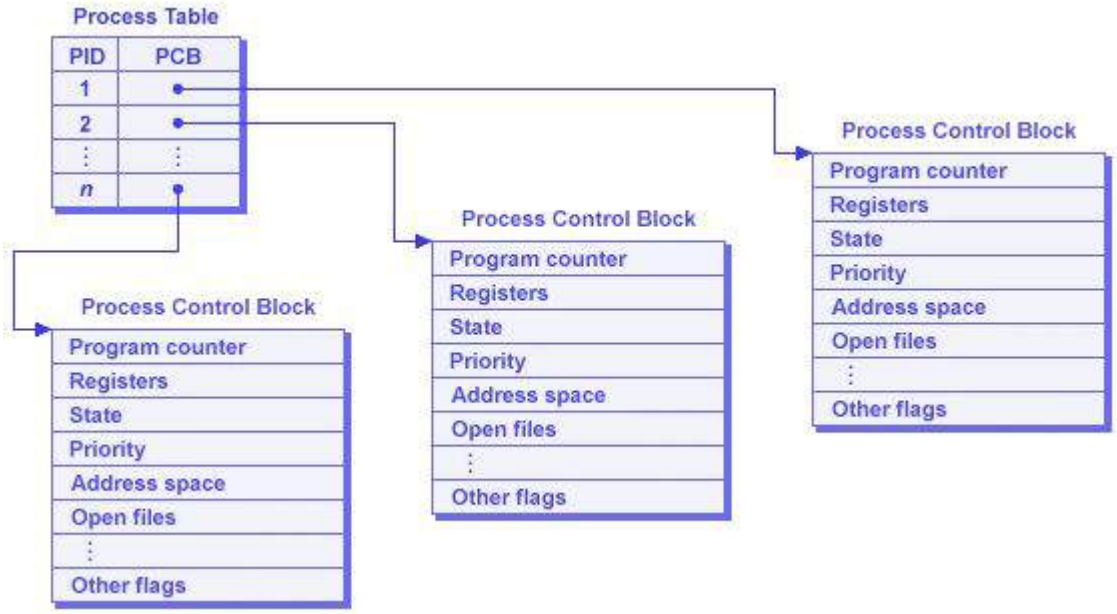
As a process executes, it changes **state**

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution



OPERATING SYSTEMS

Process Control Block (PCB)



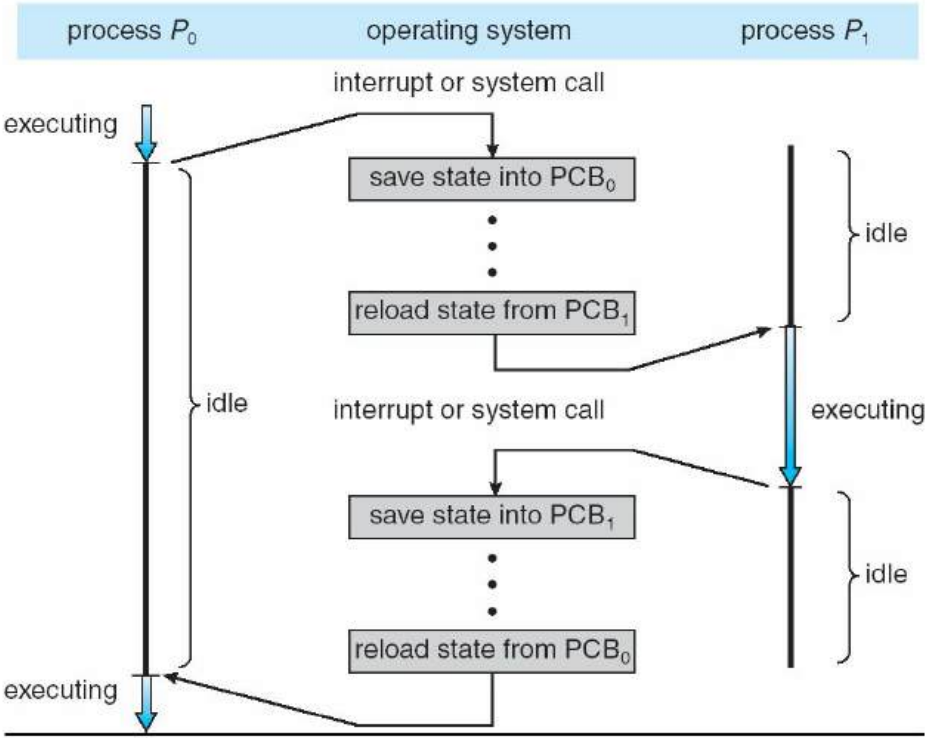
Process Control Block (PCB)

Each process is represented in the operating system by a Process Control Block (also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

OPERATING SYSTEMS

CPU switch from process to process

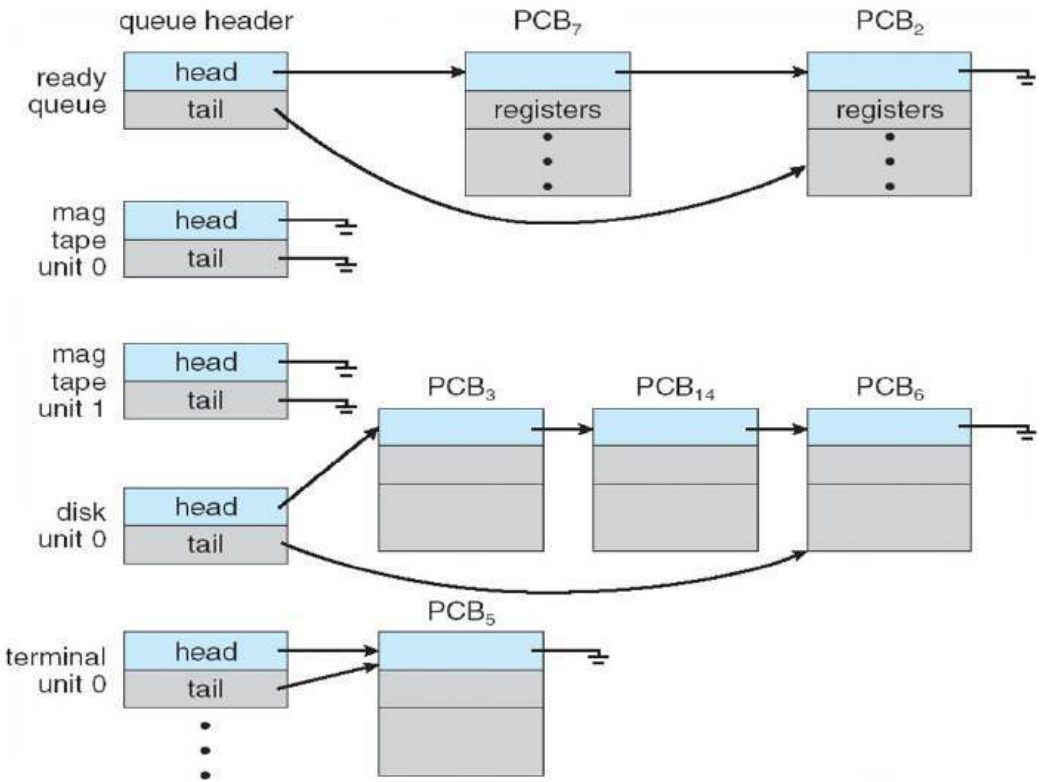


Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

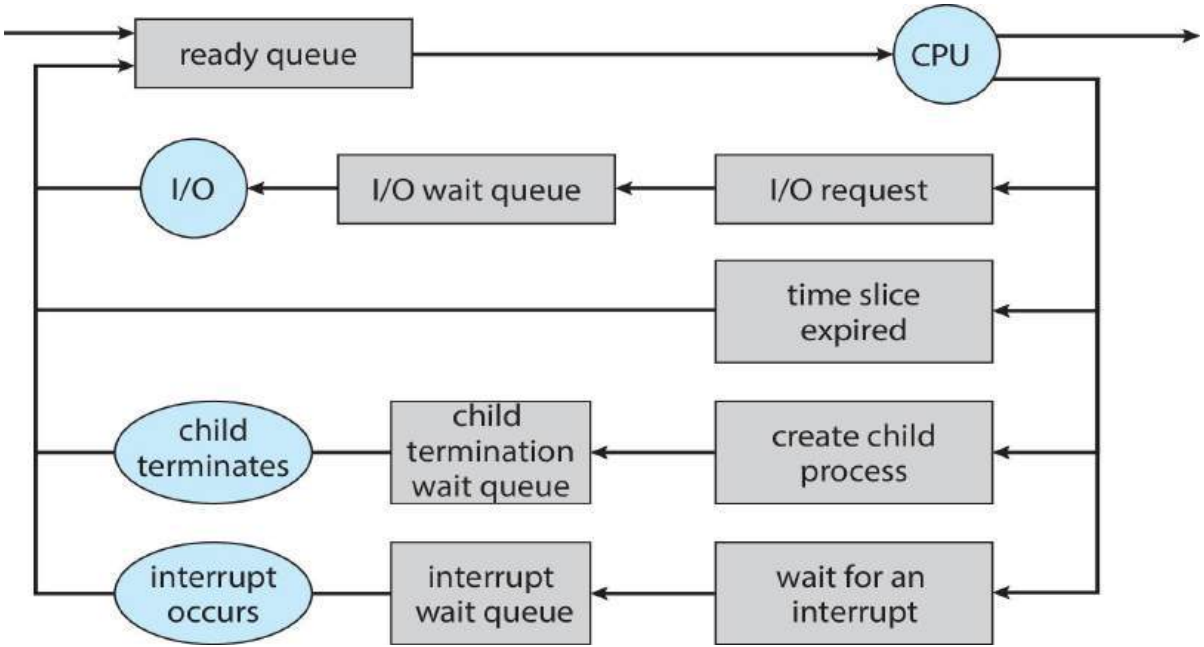
OPERATING SYSTEMS

Ready Queue And Various I/O Device Queues



OPERATING SYSTEMS

Representation of Process Scheduling



Schedulers

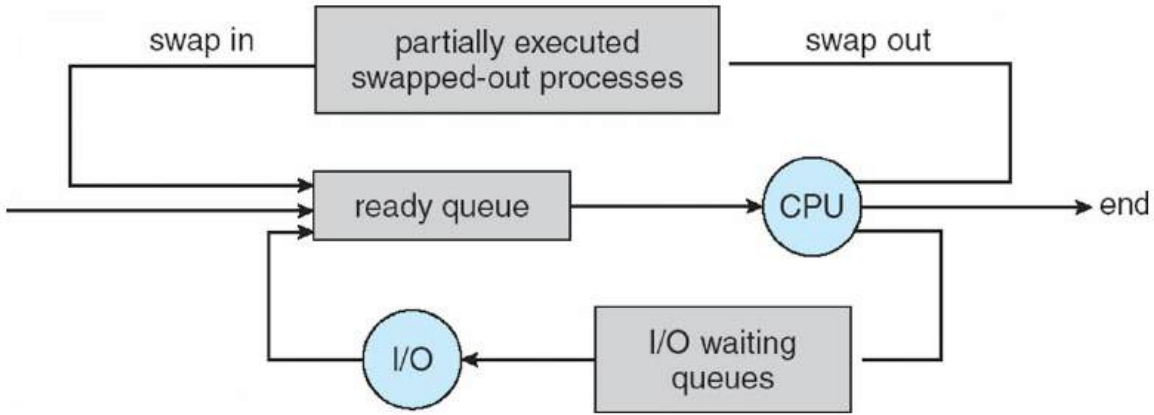
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

Schedulers

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Representation of Process Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



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
  - The more complex the OS and the PCB the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU multiple contexts loaded at once

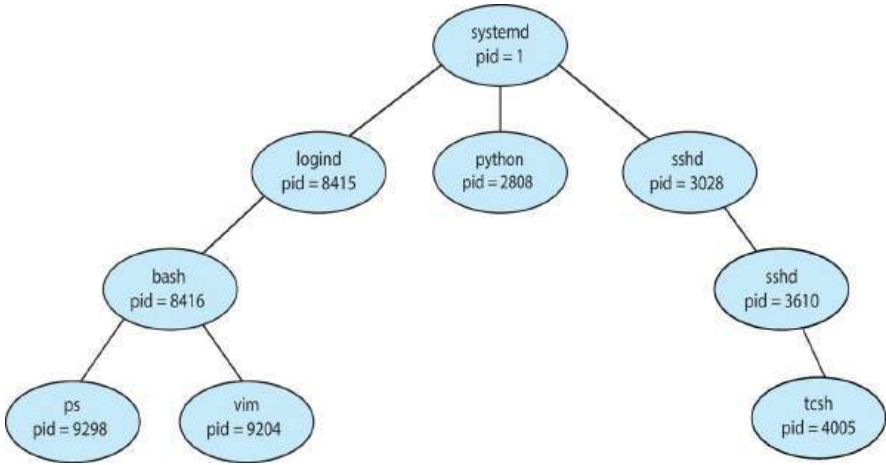
Operations on Processes

System must provide mechanisms for:

- process creation
- process termination

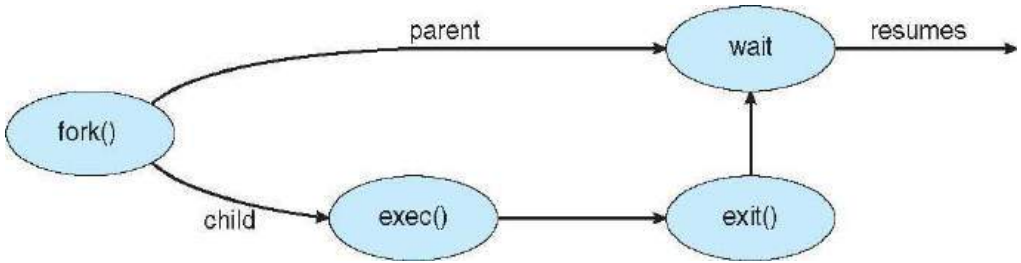
Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



Process creation using fork()

- 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



C Program forking Separate Process

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

int main()
{
pid_t pid;

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

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

    return 0;
}
```

Process Termination

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

OPERATING SYSTEMS

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call**.** The call returns status information and the pid of the terminated process
  - **pid = wait(&status);**
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait** , process is an **orphan**

Process Identifiers

- Every process has a unique **process ID**, a non-negative integer
- The process ID is the only well-known identifier of a process that is always unique
- It is often used as a piece of other identifiers, to guarantee uniqueness.
- Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse.
- Most UNIX systems implement algorithms to delay reuse, so that newly created processes are assigned IDs different from those used by processes that terminated recently

fork()

- An existing process can create a new one by calling the **fork** function

```
#include <unistd.h>
pid_t fork(void);
```

- The new process created by fork is called the child process
- Return value in the child is 0
- Return value in the parent is the process ID of the new child
- Return value is -1 on error
- The child is a copy of the parent. The child gets a copy of the parent's data space, heap, and stack

exit()

- A process can terminate normally in five ways
  1. Executing a return from the main function
  2. Calling the **exit** function.
  3. Calling the _exit or _Exit function.
  4. Executing a return from the start routine of the last thread in the process.
  5. Calling the pthread_exit function from the last thread in the process

A process can terminate abnormally in three ways

  6. Calling abort
  7. When the process receives certain signals
  8. The last thread responds to a cancellation request

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- This kernel code closes all the open descriptors for the process, releases the memory that it was using

OPERATING SYSTEMS

wait() and waitpid()

- A process that calls wait or waitpid can
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error, if it doesn't have any child processes
- If the process is calling wait because it received the SIGCHLD signal, wait will return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID on success, 0 if state hasn't changed or −1 on failure

OPERATING SYSTEMS

waitid()

- waitid() allows a process to specify which children to wait for.
- Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Returns: 0 if OK, −1 on error

OPERATING SYSTEMS

exec()

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created;
- exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

- Return: −1 on error, no return on success

OPERATING SYSTEMS

getpid() and getppid()

```
#include <unistd.h>
pid_t getpid(void);

pid_t getppid(void);
```

- **getpid**() returns the process ID (PID) of the calling process.

- **getppid**() returns the process ID of the parent of the calling process.
- This will be either the ID of the process that created this process using **fork**() or if that process has already terminated, the ID of the process to which this process has been re-parented

OPERATING SYSTEMS

Race condition

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- The fork function is a lively breeding ground for race conditions, if the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork.
- In general, which process runs first cannot be predicted
- A process that wants to wait for a child to terminate must call one of the wait functions.
- If a process wants to wait for its parent to terminate, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

- The problem with this type of loop, called polling, is that it wastes CPU time, as the caller is awakened every second to test the condition.
- To avoid race conditions and to avoid polling, some form of signaling is used between multiple processes

CPU Scheduling - Basic Concepts

- In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

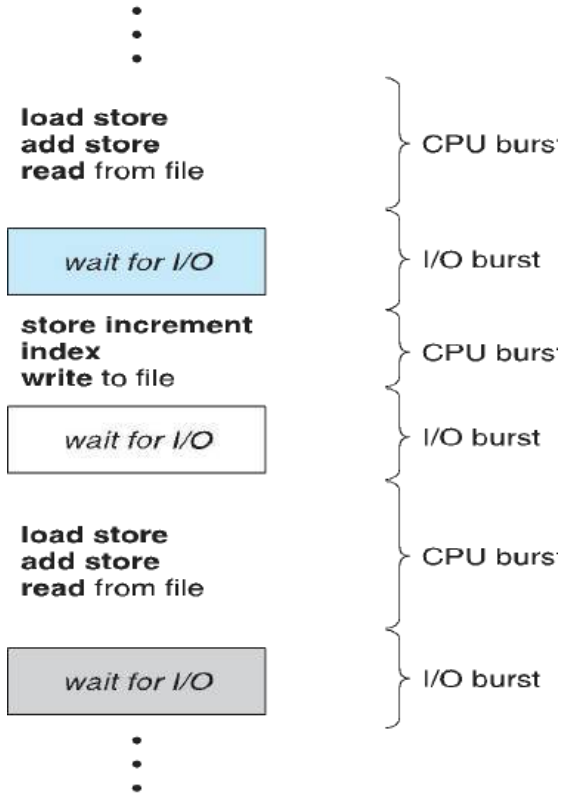
CPU Scheduling - Basic Concepts (Contd)

- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.
- Scheduling of this kind is a fundamental operating-system function.

OPERATING SYSTEMS

Alternating Sequence of CPU and I/O bursts

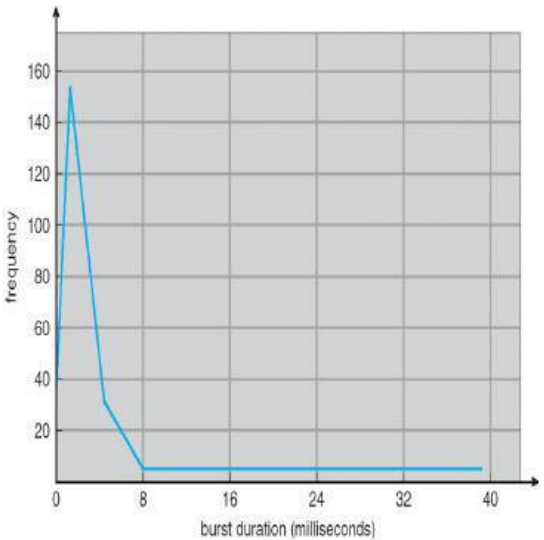
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



OPERATING SYSTEMS

Histogram of CPU-burst Times

- The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in the Figure.
- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.



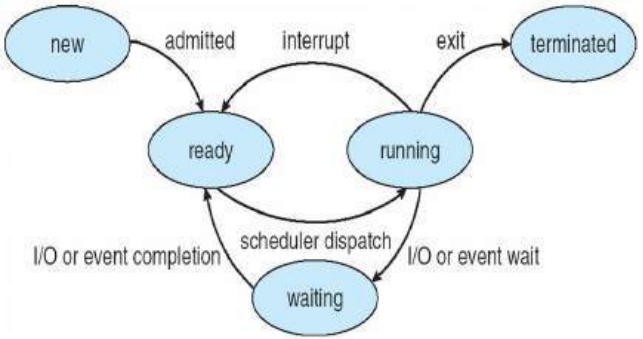
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
  - It can be FIFO, Priority, queue, tree, unordered linked list
  - The records in the queue are PCB's of the processes

OPERATING SYSTEMS

Preemptive Scheduling

- 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**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
- Scheduling algorithms used in windows3.x, non-preemptive
- Win 95 onwards used preemptive
- Preemptive Scheduling Algorithm is used in Macintosh OS



Preemptive vs Non-Preemptive Scheduling

- Unfortunately, pre-emptive scheduling can result in race conditions when data are shared among several processes. Ex: While one process is updating the shared data, it is pre-empted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

- A pre-emptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully pre-emptive when running in kernel mode.

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 (performance metric)
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

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

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

  The Gantt Chart for the schedule is:

  Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

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

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1

  The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3

- ***The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly***

FCFS Scheduling (Cont.)

- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes
- FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU
- FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each process to get a share of the CPU at regular intervals.
- It is not desirable to allow one process to keep the CPU for an extended period

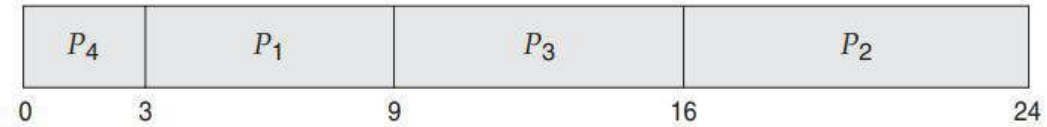
Shortest-Job-First (SJF) Scheduling

- Associate with each process the **length of its next CPU burst**
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Compute an approximation of the length of the next CPU burst

Example of SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Note: If FCFS scheduling is used, average waiting time = (0 + 6 + 14 + 21) / 4 = 10.25 ms.

Determining Length of Next CPU Burst

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

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

where τ_{n+1} is the predicted value for the next CPU burst,

The parameter α controls the relative weight of recent and past history in the prediction .

Commonly, α = 1/2, so recent history and past history are equally weighted

The value of t_n contains most recent information

The value τ_n stores the past history

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

Determining Length of Next CPU Burst

Calculate the exponential averaging with $T_1 = 10$, $\alpha = 0.5$ and the algorithm is SJF with previous runs as 8, 7, 4, 16.

Initially $T_1 = 10$ and $\alpha = 0.5$ and the run times given are 8, 7, 4, 16 as it is shortest job first,

So the possible order in which these processes would serve will be 4, 7, 8, 16 since SJF is a non-preemptive technique.

So, using formula: $T_2 = \alpha * t_1 + (1-\alpha)T_1$

we have,

$T_2 = 0.5*4 + 0.5*10 = 7$, here $t_1 = 4$ and $T_1 = 10$

$T_3 = 0.5*7 + 0.5*7 = 7$, here $t_2 = 7$ and $T_2 = 7$

$T_4 = 0.5*8 + 0.5*7 = 7.5$, here $t_3 = 8$ and $T_3 = 7$

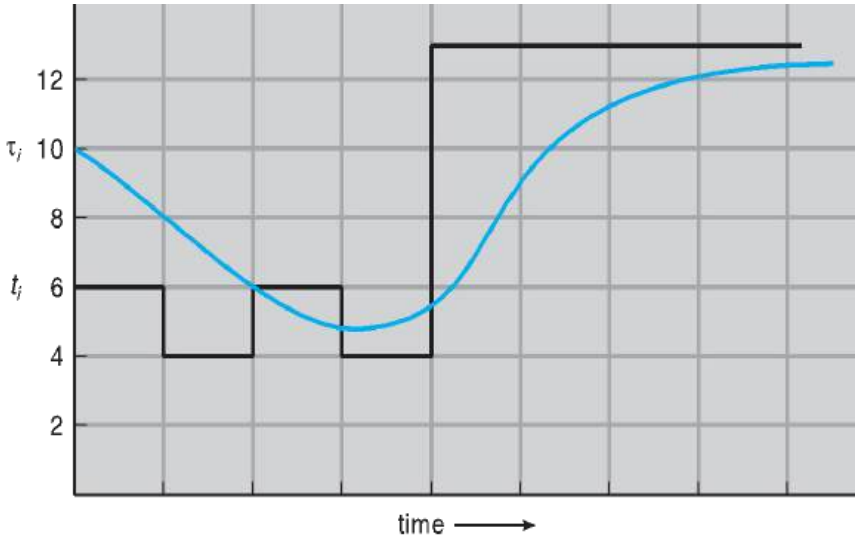
$T_5 = 0.5*16 + 0.5*7.5 = 11.8$, here $t_4 = 16$ and $T_4 = 7.5$

So the prediction for 5th burst time will be $T_5 = 11.8$

Src: https://www.geeksforgeeks.org/shortest-job-first-cpu-scheduling-with-predicted-burst-time/

OPERATING SYSTEMS

Prediction of the Length of the Next CPU Burst



$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

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

Examples of Exponential Averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

- α =0
  - $\tau_{n+1} = \tau_n$
  - Most recent CPU burst does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  - $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots$
  - $+(1-\alpha)^j \alpha\, t_{n-j} + \ldots$
  - $+(1-\alpha)^{n+1} \tau_0$
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

OPERATING SYSTEMS

More examples

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use non-preemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Burst Time
P_1	8
P_2	4
P_3	1

- What is the average wait time for these processes with the FCFS scheduling algorithm?
  Average wait time = (0 + 8 + 12)/3 = 6.67

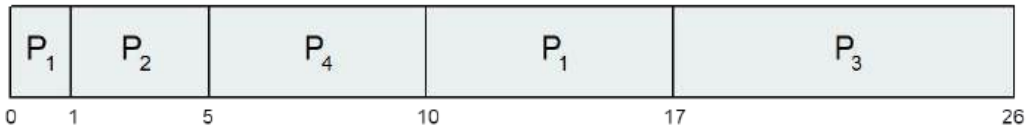
- What is the average wait time for these processes with the SJF scheduling algorithm?
  Average wait time = (5 + 1 + 0)/3 = 2

Example of Shortest-remaining-time-first

- Preemptive SJF Scheduling is sometimes called SRTF
- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority Scheduling Gantt chart

- Average waiting time = (6 + 0 + 16 + 18 + 1) / 5 = 41/5 = 8.2

Round-Robin Scheduling

- Round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length
- The ready queue is treated as a circular queue
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum

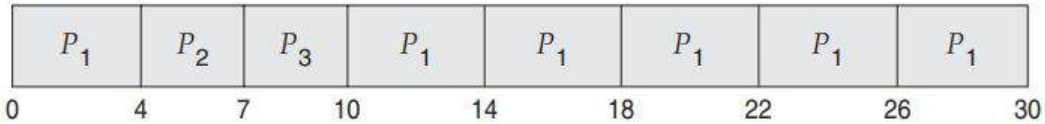
OPERATING SYSTEMS

Round-Robin Scheduling Example

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

- RR scheduling Gantt chart using a time quantum of 4 milliseconds



  - P1 waits for 6 milliseconds (10 - 4)
  - P2 waits for 4 milliseconds
  - P3 waits for 7 milliseconds.
- The average waiting time = (6 + 4 + 7)/3 = 5.66 milliseconds

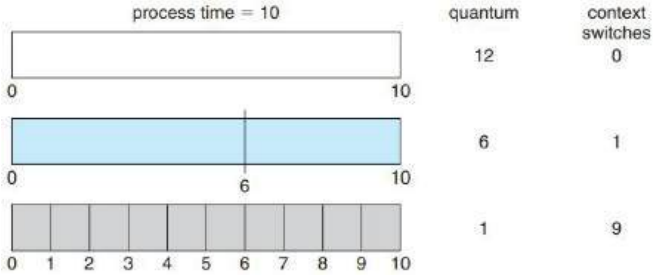
Round-Robin Scheduling Performance

- 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.
- Each process must wait no longer than (n − 1) × q time units until its next time quantum.
  - For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.
- Performance of the RR algorithm depends heavily on the size of the time quantum
- If the time quantum is extremely large, the RR policy is the same as the FCFS policy
- If the time quantum is extremely small, the RR approach can result in a large number of context switches

OPERATING SYSTEMS

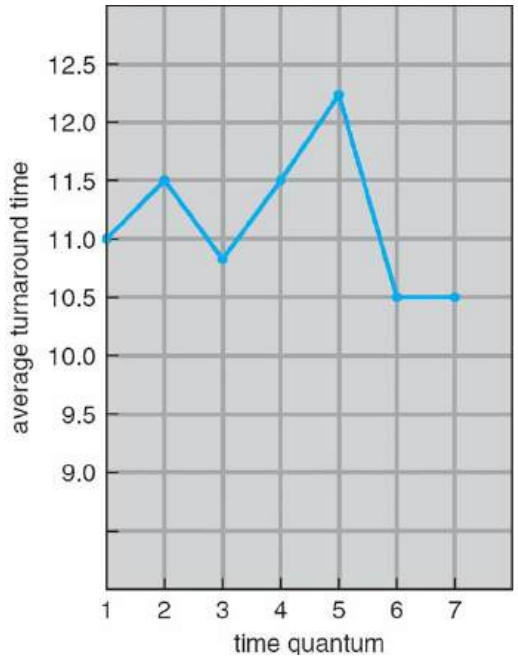
Example of Round-Robin Scheduling Performance

- Consider only one process of 10 time units.
- If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- If the quantum is 6 time units, the process requires 2 quanta, resulting in one context switch.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly



- In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
- The time required for a context switch is typically less than 10 microseconds. Thus, the context-switch time is a small fraction of the time quantum.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than the time quantum

OPERATING SYSTEMS

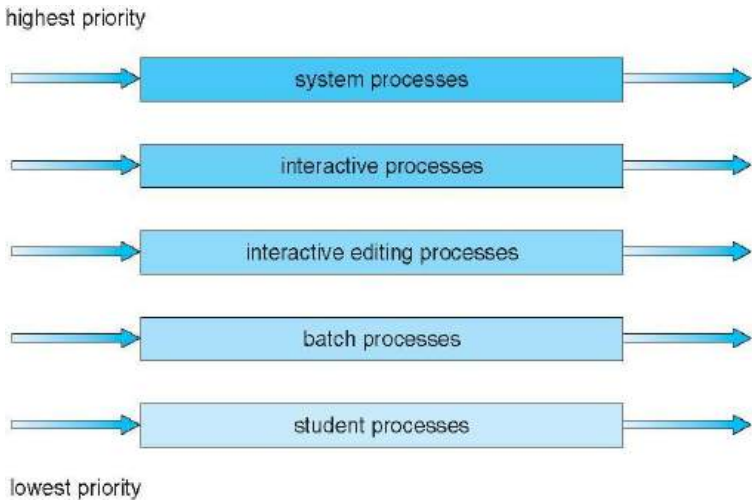
Multilevel Queue

- Ready queue is partitioned into separate queues:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently assigned to a queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- In addition, scheduling must be done between the queues
  - Fixed priority scheduling; (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

OPERATING SYSTEMS

Multilevel Queue Scheduling

- Each queue has absolute priority over lower-priority queues
- No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.



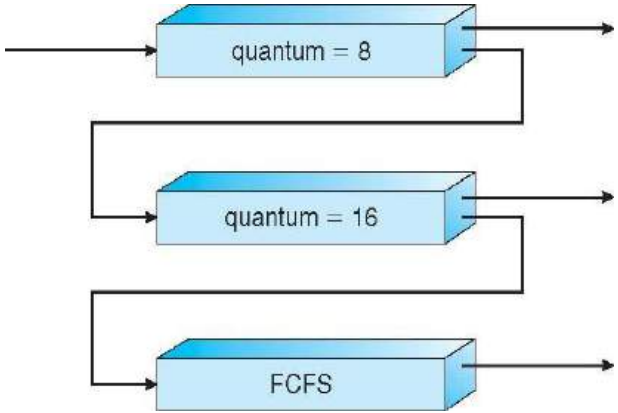
Multilevel Feedback Queue

- 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

OPERATING SYSTEMS

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

OPERATING SYSTEMS

Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

  - The scheduler first executes all processes in queue 0.
  - Only when queue 0 is empty will it execute processes in queue 1.
  - Processes in queue 2 will be executed only if queues 0 and 1 are empty.
  - A process that arrives for queue 1 will preempt a process in queue 2.
  - A process in queue 1 will in turn be preempted by a process arriving in queue 0.

OPERATING SYSTEMS

Multiple-processor Scheduling

If multiple CPUs are available, **load sharing** becomes possible, but scheduling problems become correspondingly more complex.

1. **Asymmetric multiprocessing**
   - CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server.
   - The other processors execute only user code.
   - Asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.
2. **Symmetric multiprocessing (SMP)**
   - Each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
   - All modern operating systems support SMP

OPERATING SYSTEMS

Processor Affinity

- When a process has been running on a specific processor, the data most recently accessed by the process populate the cache of the processor.
- Successive memory accesses by the process are often satisfied in cache memory.
- If the process migrates to another processor, the contents of cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**
- **Soft affinity** - OS will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors
- **Hard affinity** – OS provides system calls for process to specify a subset of processors on which it may run

OPERATING SYSTEMS

Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
- **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system.
- Load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute

1. **Push migration** - a specific task periodically checks the load on each processor and if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors
2. **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor.

- Push and pull migration need not be mutually exclusive

Linux Scheduling Through Version 2.5

- Process Scheduling in Linux
- Linux kernel ran a variation of standard UNIX scheduling algorithm
- It did not support for SMP systems
- It had poor performance for larger processes

Linux Scheduling Version 2.5

- Kernel moved to constant order *O*(1) scheduling time
- Supported SMP systems - processor affinity and load balancing between processors with good performance
- Poor response times for the interactive processes that are common on many desktop computer systems

OPERATING SYSTEMS

Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS) is the default scheduling algorithm.
- Based on **Scheduling classes**
  - Each class is assigned a specific priority.
  - Using different scheduling classes, the kernel can accommodate different scheduling algorithms
  - Scheduler picks the highest priority task in the highest scheduling class
  - Two scheduling classes are included, others can be added
    1. A scheduling class with CFS algorithm
    2. A real-time scheduling class

Linux Scheduling in Version 2.6.23 + (Cont.)

Completely Fair Scheduler (CFS)

- CFS scheduler assigns a **proportion of CPU processing time** to each task.
- Proportion calculated based on **nice value** which ranges from -20 to +19
  - A numerically lower nice value indicates a higher relative priority.
  - Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values
- Calculates **target latency** – interval of time during which task should run at least once
  - Proportions of CPU time are allocated from the value of targeted latency computed.
  - Target latency can increase if number of active tasks increase

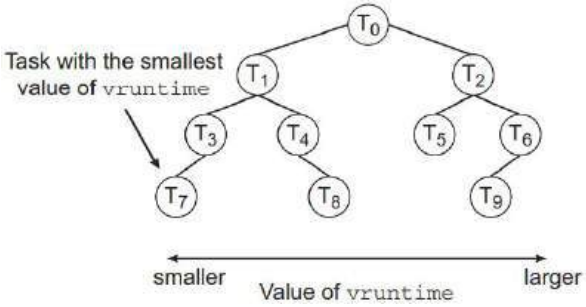
PES UNIVERSITY ONLINE

Completely Fair Scheduler (CFS)

- CFS scheduler doesn't directly assign priorities
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

Completely Fair Scheduler (CFS)

- Each runnable task is placed in a balanced binary search tree whose key is based on the value of **vruntime**
- When a task becomes runnable, it is added to the tree
- When a task is not runnable, it is deleted from the tree
- Navigating the tree to discover the task to run (leftmost node) will require O(lg N) operations (where N is the number of nodes in the tree).



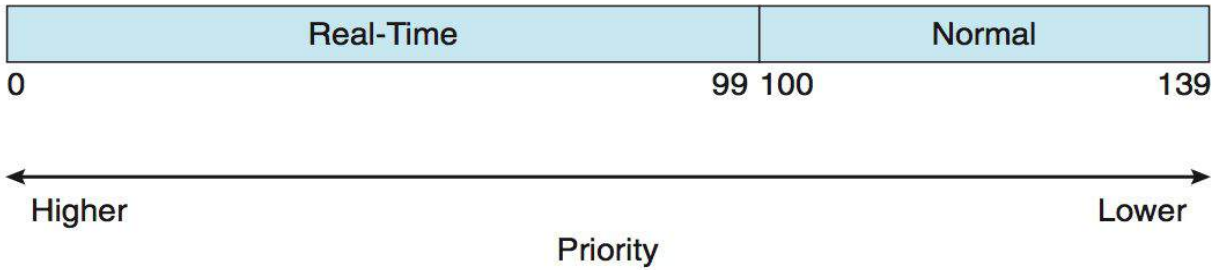
PES UNIVERSITY ONLINE

Completely Fair Scheduler (CFS)

- Assume that two tasks have the same nice values.
- One task is I/O-bound and the other is CPU-bound
- The value of **vruntime** will be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task.
- If the CPU-bound task is executing when the I/O-bound task becomes eligible to run, the I/O-bound task will preempt the CPU-bound task.

Linux Real Time Scheduling

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities (0-99)
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Windows Scheduling

- Uses priority based pre-emptive scheduling algorithm
- Scheduler ensures that the **highest-priority thread** will always run
- Windows kernel that handles scheduling is called the **dispatcher**
- Thread selected by the dispatcher runs until it is preempted by a higher-priority thread or until it terminates or until its time quantum ends or until it calls a blocking system call
- Dispatcher uses a 32-level priority scheme
  - **Variable class** is 1-15, **real-time class** is 16-31
  - Priority 0 is memory-management thread
- Queue for each priority class
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Windows API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Thread Priorities

- A thread within a given priority class also has a relative priority
- Priority of each thread is based on both the priority class it belongs to (top row in the diagram) and its relative priority within that class (left column in the diagram)

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

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Windows distinguishes between foreground process and background processes
- Foreground processes given 3x priority boost
- This priority boost gives the foreground process three times longer to run before a time-sharing preemption occurs.

What is a shell?

- Instructions entered in response to the shell prompt have the following syntax: **command [arg1] [arg2] .. [argn]**
- The brackets [] indicate that the arguments are optional. Many commands can be executed with or without arguments
- The shell parses the words or tokens (commandname , options, filesnames[s]) and gets the kernel to execute the commands assuming the syntax is correct.
- Typically, the shell processes the complete line after a carriage return (cr) (carriage return) is entered and finds the program that the command line wants executing.
- The shell looks for the command to execute either in the specified directory if given (./mycommand) or it searches through a list of directories depending on your $PATH variable.

OPERATING SYSTEMS

Common Environment variables

- **SHELL**: This describes the shell that will be interpreting any commands you type in. In most cases, this will be bash by default, but other values can be set if you prefer other options.
- **TERM**: This specifies the type of terminal to emulate when running the shell. Different hardware terminals can be emulated for different operating requirements. You usually won't need to worry about this though.
- **USER**: The current logged in user.
- **PWD**: The current working directory.
- **OLDPWD**: The previous working directory. This is kept by the shell in order to switch back to your previous directory by running cd -.
- **PATH**: A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- **HOME**: The current user's home directory.

Common SHELL variables

- **BASHOPTS**: The list of options that were used when bash was executed. This can be useful for finding out if the shell environment will operate in the way you want it to.
- **BASH_VERSION**: The version of bash being executed, in human-readable form.
- **BASH_VERSINFO**: The version of bash, in machine-readable output.

```
ubuntu@ip-172-31-41-208:~$ echo $BASH_VERSION
5.0.17(1)-release
```

OPERATING SYSTEMS

SHELL basics

- Creating shell variables:

```
ubuntu@ip-172-31-41-208:~$ MY_VAR="Hello world"
```

- Accessing the value of any shell or environmental variable:

```
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
Hello world
```

- Spawning a child shell process

```
ubuntu@ip-172-31-41-208:~$ bash
```

- In the child shell process, the shell variable defined in the parent is not available

```
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
```

- To terminate the child shell process,

```
ubuntu@ip-172-31-41-208:~$ exit
```

- To pass the shell variables to child shell processes, you need to **export** the variable

```
ubuntu@ip-172-31-41-208:~$ export MY_VAR="Hello world"
ubuntu@ip-172-31-41-208:~$ bash
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
Hello world
```

SHELL basics

- Removing a shell variable

```
ubuntu@ip-172-31-41-208:~$ unset MY_VAR
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR

ubuntu@ip-172-31-41-208:~$
```

- For setting environment variables at login, edit the **.profile** file in the **$HOME** directory and add the export command

```
ubuntu@ip-172-31-41-208:~$ cd $HOME
ubuntu@ip-172-31-41-208:~$ vi .profile
```

OPERATING SYSTEMS

SHELL control flow - if

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

Example: To check if the file exists

```
#!/bin/bash

if [ -f welcome.txt ]; then
   echo "File exists"
else
   echo "File does not exist"
fi
```

OPERATING SYSTEMS

SHELL control flow - loops

```
#!/bin/bash

for i in {1..5}
do
   echo "i = $i"
done
```

Nested loop:

```
#!/bin/bash

for i in {1..5}
do

   for j in {1..3}
   do
      echo "i = $i and j = $j"
   done
done
```

OPERATING SYSTEMS

SHELL control flow – continue and break

```
#!/bin/bash

for i in {1..5}
do

   if [ $i -eq 3 ]
   then
      continue
   fi

   for j in {1..3}
   do
      echo "i = $i and j = $j"
   done
done
```

```
#!/bin/bash

for i in {1..5}
do

   if [ $i -eq 3 ]
   then
      break
   fi

   for j in {1..3}
   do
      echo "i = $i and j = $j"
   done
done
```

OPERATING SYSTEMS

cron examples

SCHEDULE	SCHEDULED VALUE
5 0 * 8 *	At 00:05 in August.
5 4 * * 6	At 04:05 on Saturday.
0 22 * * 1-5	At 22:00 on every day-of-week from Monday through Friday.

PES UNIVERSITY
CELEBRATING 50 YEARS

OPERATING SYSTEMS
Interprocess Communication

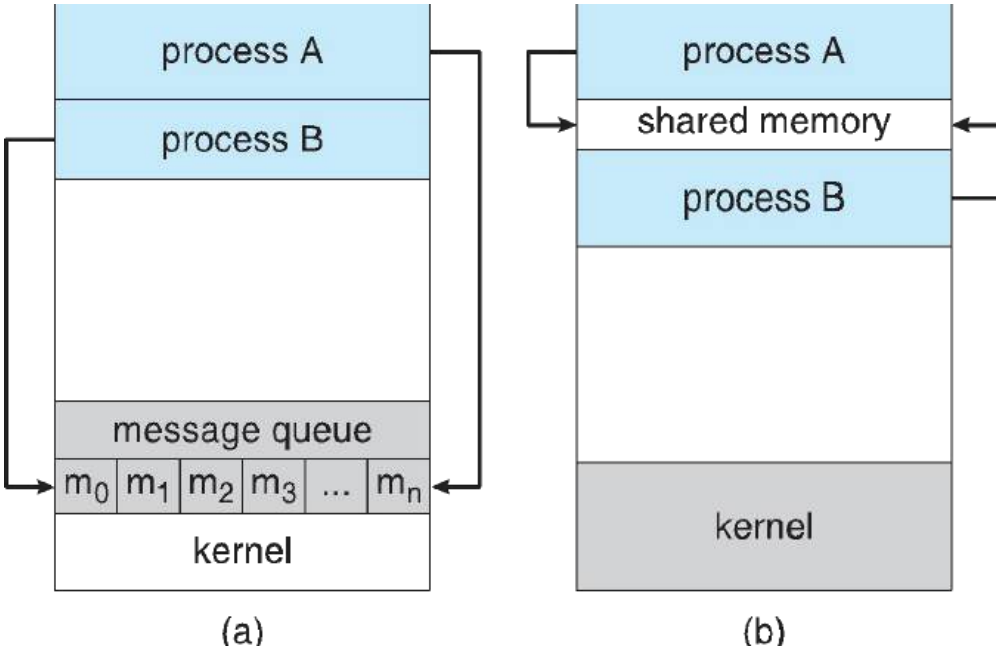
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

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

Communication Models

- Two models of IPC
  a) **Message passing**
  b) **Shared memory**



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
    - Consumer may have to wait for new items, but the producer can always produce new items
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Consumer must wait if the buffer is empty; producer must wait if the buffer is full

Bounded-Buffer – Producer Consumer

- Shared data

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

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

- Shared buffer is implemented as a circular array with 2 logical pointers: **in** and **out**
- Buffer is empty when **in == out;** buffer is full when **((in** + 1) % BUFFER_SIZE) == **out**
- Variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer
- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next_produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next_consumed */
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

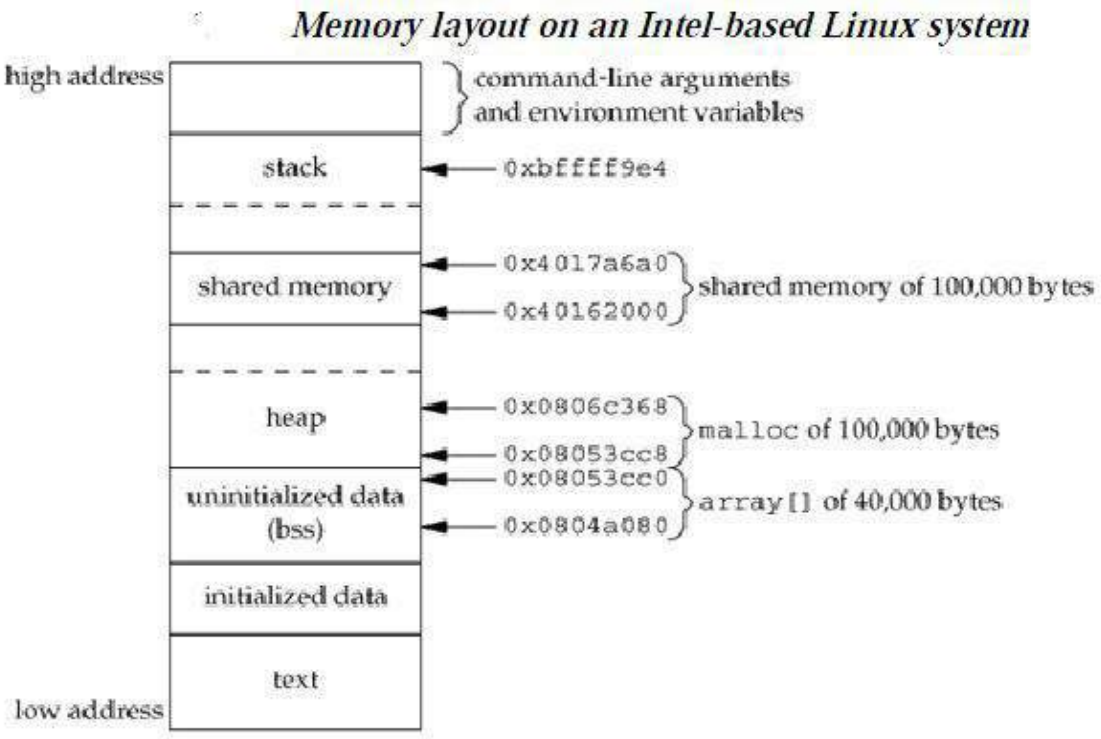
OPERATING SYSTEMS

Shared Memory (Cont.)

- Shared memory allows two or more processes to share a given region of memory.
- Shared memory is the fastest form of IPC, because the data does not need to be copied between the client and the server.
- The only trick in using shared memory is synchronizing access to a given region among multiple processes.
- If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done.
- Often, semaphores are used to synchronize shared memory access. (record locking can also be used.)

OPERATING SYSTEMS

Shared Memory (Cont.)



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*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

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

Indirect Communication

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

Indirect Communication (Cont.)

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **send**(*A, message*) – send a message to mailbox A
- **receive**(*A, message*) – receive a message from mailbox A

Indirect Communication (Cont.)

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

OPERATING SYSTEMS

Message Passing - Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous** between the sender and the receiver

Message Passing - Synchronization (Cont.)

- Producer-consumer becomes trivial

```
 message next_produced;
while (true) {
   /* produce an item in next_produced */
send(next_produced);
}
```

```
message next_consumed;
while (true) {
  receive(next_consumed);

  /* consume the item in next_consumed */
}
```

OPERATING SYSTEMS

Buffering

- Queue of messages attached to the link (direct or indirect); messages reside in a temporary queue
- Queues can be implemented in one of three ways
    1. Zero capacity – no messages are queued on a link.

       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
- Zero-capacity case is sometimes referred to as a message system with no buffering; other cases are referred to as systems with automatic buffering

Pipes

- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems.
- Pipes have two limitations.
  1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
  2. Pipes can be used only between processes that have a **common ancestor**. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

pipe()

- A pipe is created by calling the **pipe()** function.

```
#include <unistd.h>
int pipe(int fd[2]);
```

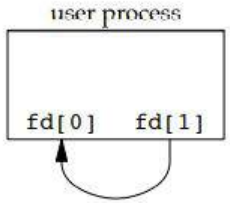
- Return value: 0 if OK, −1 on error
- Two file descriptors are returned through the fd argument:
  1. fd[0] is open for reading
  2. fd[1] is open for writing.
- The output of fd[1] is the input for fd[0]

OPERATING SYSTEMS

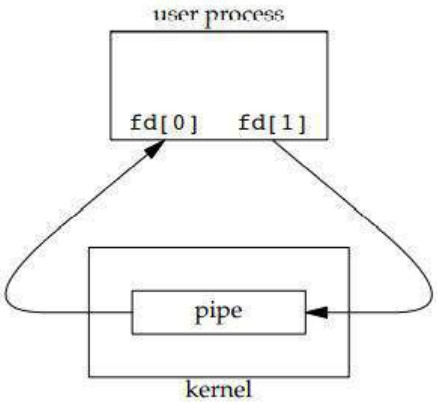
pipe()

- Two ways to picture a half-duplex pipe

1. The two ends of the pipe connected in a single process



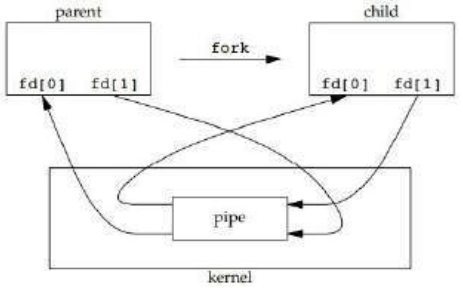
2. The data in the pipe flows through the kernel



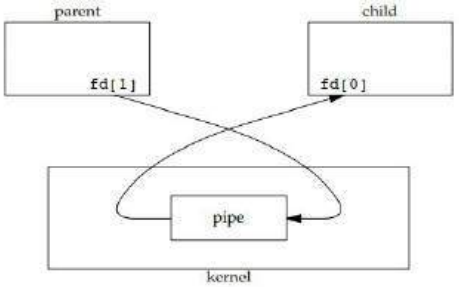
OPERATING SYSTEMS

pipe()

- Normally, a process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa



- For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]) and the child closes the write end (fd[1]).
- For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]



Pipe from parent to child

OPERATING SYSTEMS

popen() and pclose()

- A common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the **popen()** and **pclose()** functions
- These two functions handle all the work: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);

int pclose(FILE *fp);
```

- The function **popen()** does a fork and exec to execute the cmdstring and returns a standard I/O file pointer.
- If type is "r", the file pointer is connected to the standard output of cmdstring.
- If type is "w", the file pointer is connected to the standard input of cmdstring
- The function popen() returns file pointer if OK, NULL on error
- The function **pclose()** closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell or -1 on error

OPERATING SYSTEMS

FIFOs

- FIFOs are sometimes called **named pipes**.
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.
- With FIFOs, unrelated processes can exchange data
- Creating a FIFO is similar to creating a file

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

- Functions mkfifo() and mkfifoat() return 0 if OK, −1 on error
- Once a FIFO has been created using mkfifo() or mkfifoat() , it can be opened using open(). Normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.
- There are two uses for FIFOs.
    1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
    2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers

OPERATING SYSTEMS

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- A new queue is created or an existing queue opened by **msgget**.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
```

- Returns message queue ID if OK, −1 on error
- New messages are added to the end of a queue by **msgsnd**.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

- Every message has a positive long integer type field, a non-negative length, and the actual data bytes all of which are specified to **msgsnd** when the message is added to a queue.
- Messages are fetched from a queue by **msgrcv**.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

- Messages need not be fetched in a first-in, first-out order. Instead, can be fetched based on their type field

Semaphores

- A semaphore is a counter used to provide access to a shared data object for multiple processes.
- To obtain a shared resource, a process needs to do the following:
  1. Test the semaphore that controls the resource.
  2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
  3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.
- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1.
- If any other processes are asleep, waiting for the semaphore, they are awakened.

OPERATING SYSTEMS

Semaphores

- First need to obtain a semaphore ID by calling the semget function

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

- Returns: semaphore ID if OK, −1 on error
- **semctl** function is the catchall for various semaphore operations.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

- The function **semop** atomically performs an array of operations on a semaphore set

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

- The semoparray argument is a pointer to an array of semaphore operations
- Returns 0 if OK, −1 on error

Shared memory

- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server
- The challenge in using shared memory is synchronizing access to a given region among multiple processes.
- Semaphores and mutexes are used to synchronize shared memory access
- Function **shmget** is used to obtain a shared memory identifier

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

- Returns shared memory ID if OK, −1 on error

Shared memory

- Function **shmctl** is the catchall for various shared memory operations

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Returns 0 if OK, −1 on error
- The cmd argument specifies one of the commands to be performed on the segment specified by shmid
- Once a shared memory segment has been created, a process is attached to this memory segment by calling **shmat**

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

- Returns: pointer to shared memory segment if OK, −1 on error

Shared memory

- Function call **shmdt** will detach a shared memory segment from a process

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

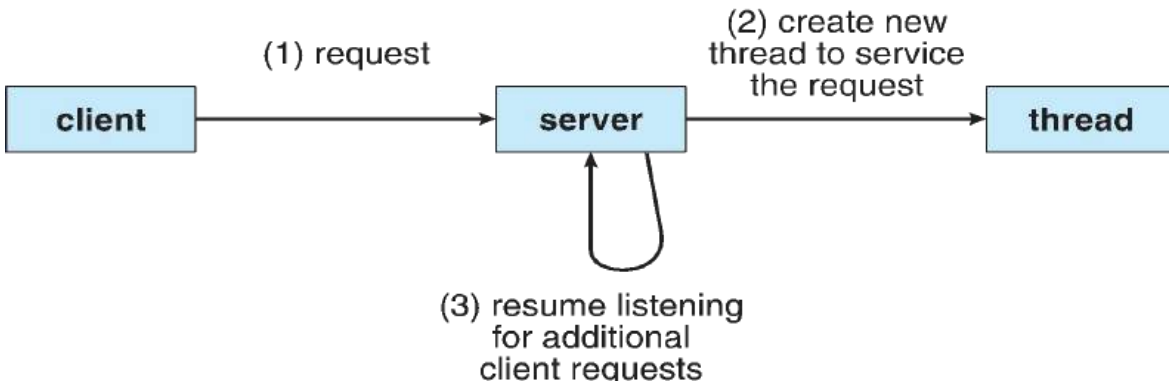
- Returns 0 if OK, −1 on error
- A shared memory segment can be removed by calling **shmctl** with a command of IPC_RMID.

Overview and Motivation

- A Thread is a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- It consists of thread ID, Program counter, a register set and stack
- Shares with other threads of same process its code, data,file descriptors, signals
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads.
- Application 1: internet browser.
  - numerous tabs open at a given time
  - Multiple threads of execution are used to load content, display animations, play a video, fetch data from a network and so on.

Multithreaded Server Architecture

- Process creation is heavy-weight while thread creation is light-weight
- Process creation if time consuming, resource intensive
- Threads also play a vital role in remote procedure call (RPC) systems
- Can simplify code, increase efficiency
- Kernels are generally multithreaded



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked or performing lengthy operations.
  - It is useful in designing user interfaces.
  - A multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.
  - Example: click on the button
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing.
  - Programmer needs to specify the techniques for sharing
  - But threads share the memory and other resources
  - Sharing of resources helps in having many threads within the same address space

Benefits

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching.
  - In Solaris, for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.
- **Scalability –** process can take advantage of multiprocessor architectures.
  - Threads can run on multiple cores parallelly

Processes vs Threads

Process

- Will by default not share memory
- Most file descriptors not shared
- Don't share filesystem context
- Don't share signal handling

Thread

- Will by default share memory
- Will share file descriptors
- Will share filesystem context
- Will share signal handling

Attributes shared by Threads

- process ID and parent process ID;process group ID and session ID;
- controlling terminal;
- process credentials (user and group Ids);open file descriptors;
- record locks created using fcntl();signal dispositions;
- file system–related information: umask, cwd and root directory;
- resource limits;CPU time consumed (as returned by times());
- resources consumed (as returned by getrusage()); nice value (set by setpriority() and nice()).

Attributes specific to Threads

- thread ID ;signal mask;
- Thread-specific data ;
- the errno variable;
- floating-point environment (see fenv(3));
- stack (local variables and function call linkage information i.e CPU registers saved in the called function's stack frame when one function calls another function and restored for the calling function when the called function returns)
- and a few more …….

Thread Concepts

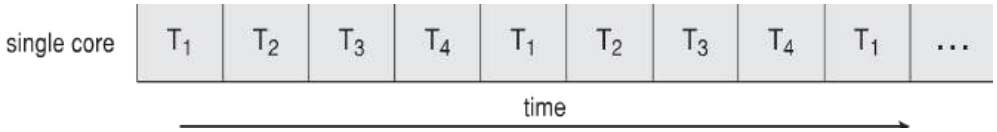
- A thread consists of the following information necessary to represent an execution context within a process.
- thread ID that identifies the thread within a process
- Every thread has a thread ID
- set of register values
- stack
- scheduling priority and policy
- signal mask
- Errno variable
- Thread-specific/thread-private data (each thread to access its own separate copy of the data, without worrying about synchronizing access with other threads)

Thread Concepts (Cont.)

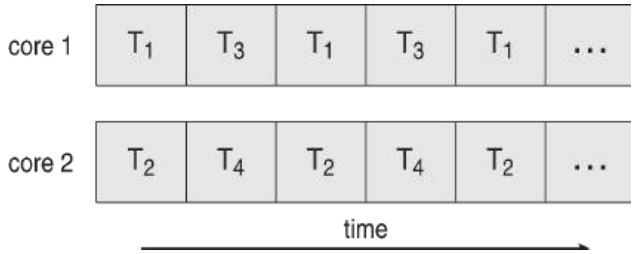
- When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread.
- The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask
- The set of pending signals for the thread is cleared.
- The pthread functions usually return an error code when they fail. They don't set errno like the other POSIX functions.

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



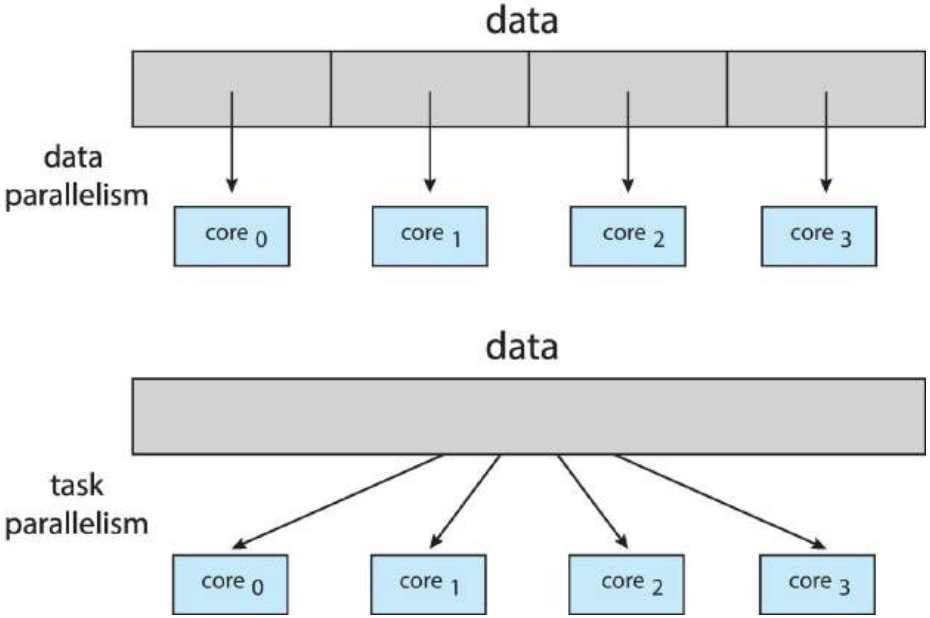
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is portion of an application that needs to be done in serial
- *N* processing cores

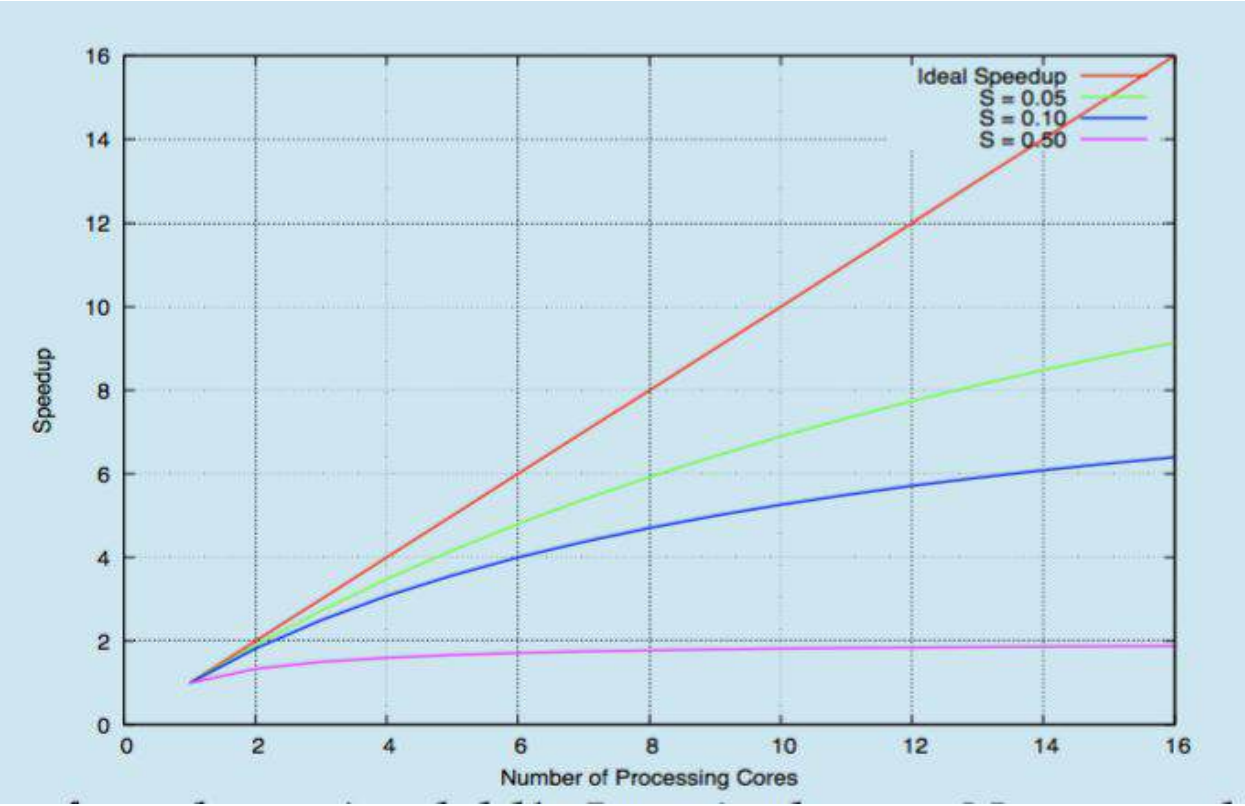
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

  Serial portion of an application has disproportionate effect on performance gained by adding additional cores
- But does the law take into account contemporary multicore systems?

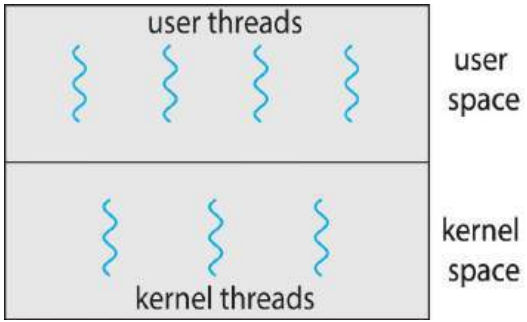
OPERATING SYSTEMS

Amdahl's Law



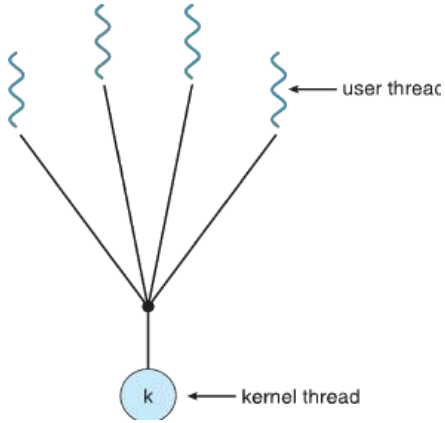
User Threads and Kernel Threads

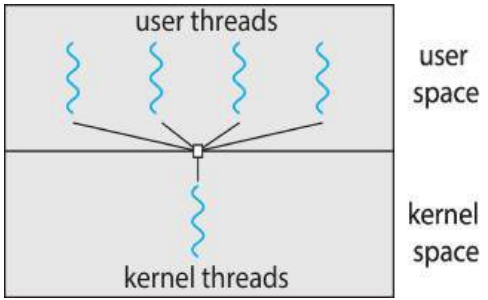
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X



Many-to-One

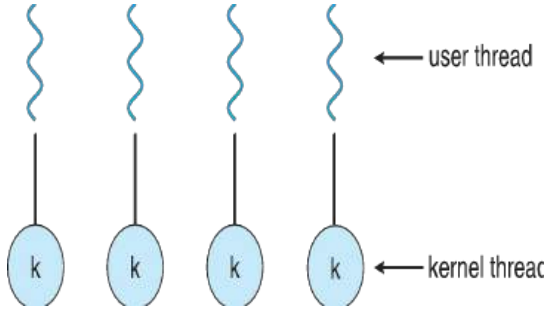
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

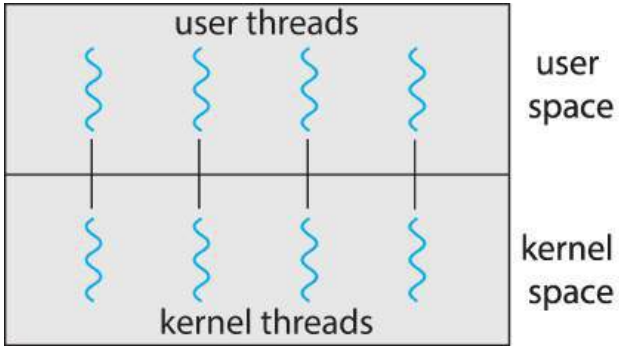




One-to-One

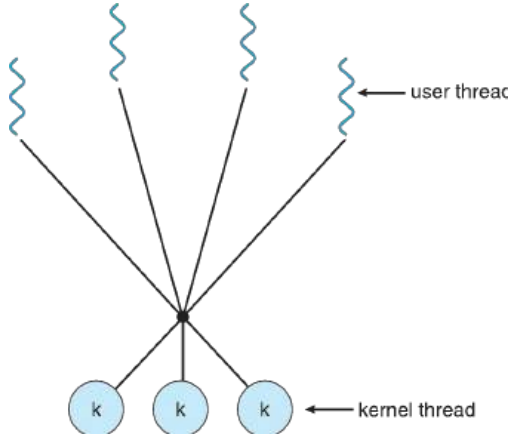
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

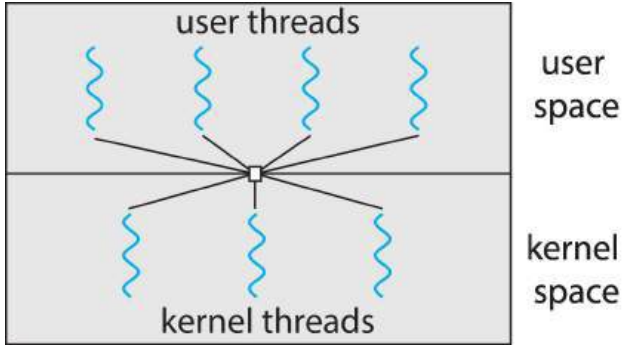




Many-to-Many Model

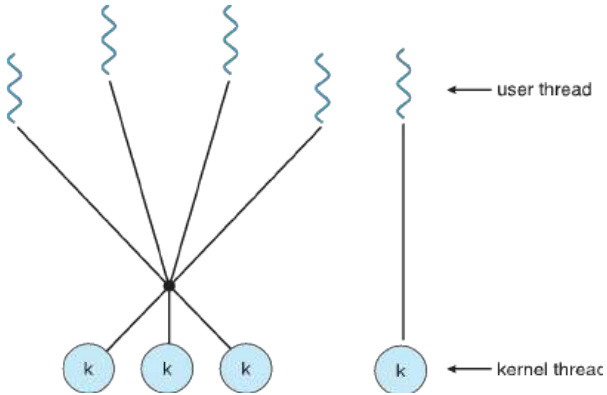
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

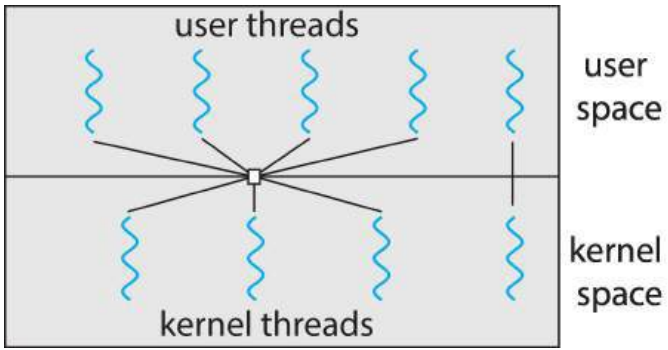




Two-level Model

- 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





Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP.
- Scheduling of user level threads (ULT) to kernel level threads (KLT) via leight-weight process (LWP) by the application developer.
  - **Leightweight Process (LWP) :**
    Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources
- Scheduling of kernel level threads by the system scheduler to perform different unique OS functions.

Thread Scheduling

- 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
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

OPERATING SYSTEMS

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
 /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Thread Libraries: Pthreads

- Provides the programmer with an API for creating and managing threads.
- Two primary ways of implementing a thread library.
  - Provide a library entirely in user space with no kernel support.
    - All code and data structures for the library exist in user space.
  - Implement a kernel-level library
    - Code and data structures for the library exist in kernel space.
    - Invoking a function directly by the operating system.

Thread Libraries: Pthreads

- Three main thread libraries are in use today:
  - (1) POSIX Pthreads,
  - (2)Win32,
  - (3) Java.
- Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
- The Win32 thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.

Thread Libraries: Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows threads

- Windows implements the Win32 API as its primary API.
- A Windows application runs as a separate process, and each process may contain one or more threads.
- Windows uses the one-to-one mapping.
- Windows also provides support for a **fiber** library, which provides the functionality of the many-to-many model

Windows threads

- The general components of a thread include:
  - A thread ID uniquely identifying the thread
  - A register set representing the status of the processor.
  - A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
  - A private storage area various run-time libraries and dynamic link libraries (DLLs)

Windows Threads

- Threads are created in theWin32 API using the CreateThread() function the Win32 API.
- Using the WaitForSingleObject() function,which causes the creating thread to block until the child thread has exited

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Windows Multithreaded C Program (Cont.)

```
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

  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 **counter** that keeps track of the number of full buffers. Initially, **counter** 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 in next_produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE; /* Variable in points to the next free position in the buffer */
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE; /* Variable out points to the first full position in the buffer */
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

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

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

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

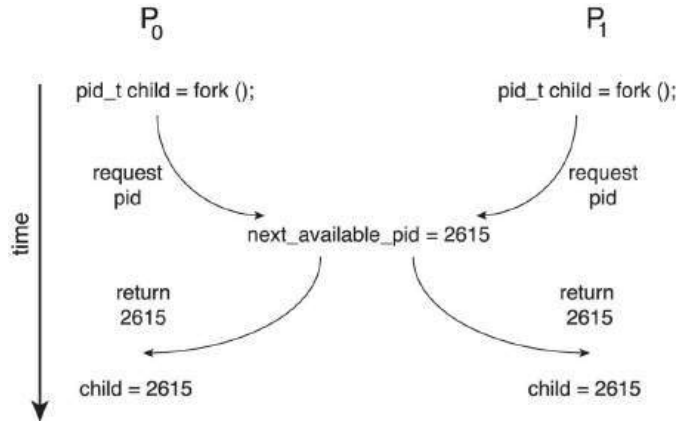
```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

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

Race Condition (Another Example)

- Processes P_0 and P_1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Critical Section

- General structure of process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode.
  - Preemptive kernels are difficult to design on SMP architectures
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
  - It is free from race conditions on kernel data structures
- Preemptive kernel may be more responsive. Suitable for real-time systems.

OPERATING SYSTEMS

Peterson's Solution

- Software-based solution to the Critical Section problem.
- Good algorithmic description of solving the problem
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- **Peterson's Solution** restricted to two process.
- P_i and P_j are two process, j=1-i
- Peterson solution requires two processes share two data items:
  - 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

Algorithm for Process P_i

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
        /* critical section */
    flag[i] = false;
        /* remainder section */
}
```

The structure of process P_i in Peterson's solution

Software Solution

- To prove solution is correct, we need show
- Mutual exclusion is preserved

  $\mathbf{P_i}$ enters critical section only if:

  turn = i

  and **turn** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Review of Peterson's solution

Code for process i

```
do{
flag[i]=TRUE
turn=j
while(flag[j]&&turn==j);//Do-nop
   critical section
flag[i]=FALSE;
Reminder section
}while(TRUE)
```

Code for process j

```
do{
flag[j]=TRUE
turn=i
while(flag[i]&&turn==i);//Do-nop
   critical section
flag[j]=FALSE;
Reminder section
}while(TRUE)
```

- **Principles of Concurrency**
- relative speed of execution of processes is not predictable.
- system interrupts are not predictable
- scheduling policies may vary

Synchronization Hardware

- Software based solutions are not guaranteed to work on modern computer architectures
- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks.
- synchronization can be done through Lock & Unlock technique
- Locking part is done in the Entry Section. After locking the process enter critical section.
- The process is moved to the Exit Section after it is done with execution in CS.
- Unlock is done in exit section.
- This process is designed in such a way that all the three conditions of the Critical Sections are satisfied

Synchronization Hardware

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

test_and_set Instruction

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.
- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

test_and_set Instruction

Definition:

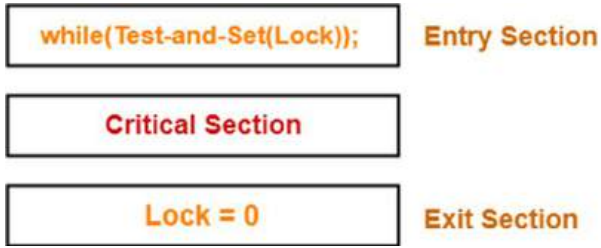
```
boolean test_and_set (boolean *target)
  {
     boolean rv = *target;
     *target = TRUE;
     return rv:
  }
```

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

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */
      /* critical section */
  lock = false;
      /* remainder section */
} while (true);
```



compare_and_swap Instruction

Definition:

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

  if (*value == expected)
    *value = new_value;
return temp;
}
```

```
do{
while(compare_and_swap(&lock,0,1)!=0);
Critical section
lock=0
Remainder section
}while(true)
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if *value == expected. That is, the swap takes place only under this condition.
4. In the x86 (since 80486) and Itanium architectures this is implemented as compare and exchange (CMPXCHG) instruction

Solution using compare_and_swap()

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
     ; /* do nothing */
   /* critical section */
 lock = 0;
   /* remainder section */
} while (true);
```

Mutual exclusion is satisfied

Do not satisfy bounded waiting requirement

Bounded-waiting Mutual Exclusion with test_and_set

This test_and_set algorithm satisfies all the critical section requirements

The common data structures are

boolean waiting[n];

boolean lock;

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&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);
```

Mutex Locks

- Previous solutions i.e hardware and software solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
- This lock therefore called a **spinlock**
- It is problem in real time systems
- Busy waiting wastes CPU cycles

Advantages of spinlocks

- no context switch is required when a process must wait on a lock.
- context switch may take considerable time.
- When locks are expected to be held for short times, spinlocks are useful
- These are employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

Solution to Critical-section Problem Using Locks

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

Implementation of acquire and release

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

acquire() and release()

- Solution to a critical section problem using mutex
- **do {**

acquire lock

critical section

release lock

remainder section

} while (true);

OPERATING SYSTEMS

Scenario 2

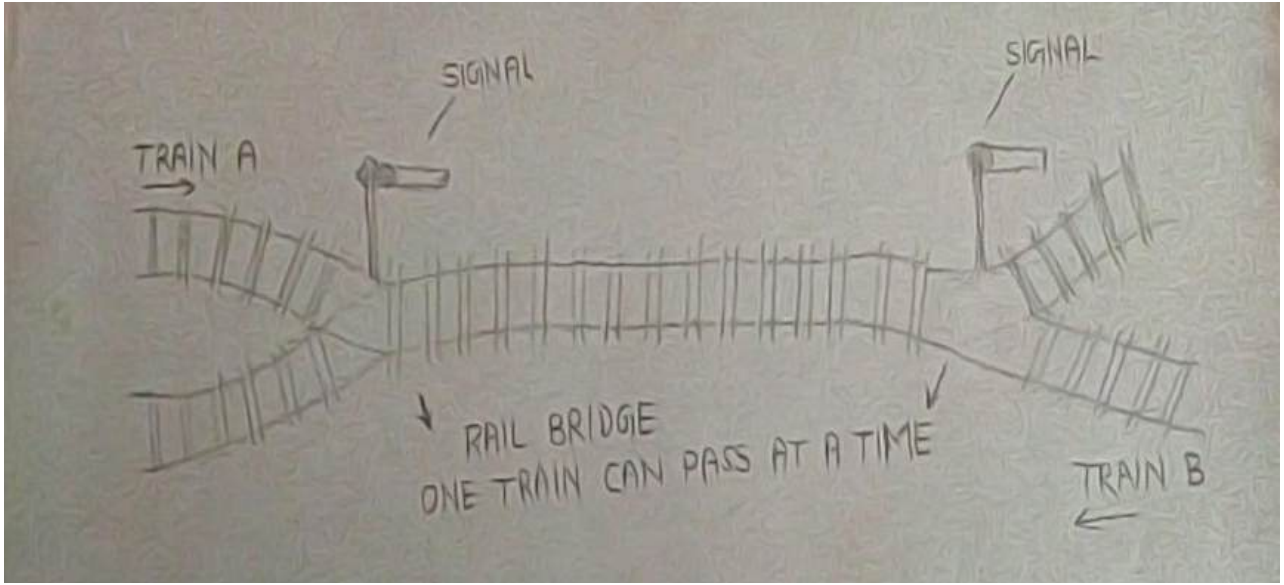


Image src: https://www.linkedin.com/pulse/concurrency-semaphore-real-life-kunal-saxena/

Scenario 1

- Consider a library of an university with 10 rooms
- At a time one room can be used by only one student by informing the front desk for reading.
- Once he completes reading, he has to inform the front desk.
- Person at front desk knows how many rooms are available for use and how many are occupied, how may of them are waiting.
- Once the room is vacant, who will get the chance to occupy the room?

Idea Src:https://en.wikipedia.org/wiki/Semaphore_(programming)

Semaphore

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

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

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

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

Semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

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

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

Semaphore Implementation

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

int value;

struct process *list;

} semaphore;

Semaphore Implementation with no Busy waiting (Cont.)

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

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Problems with Semaphores (Cont.)

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Priority Inheritance Protocol

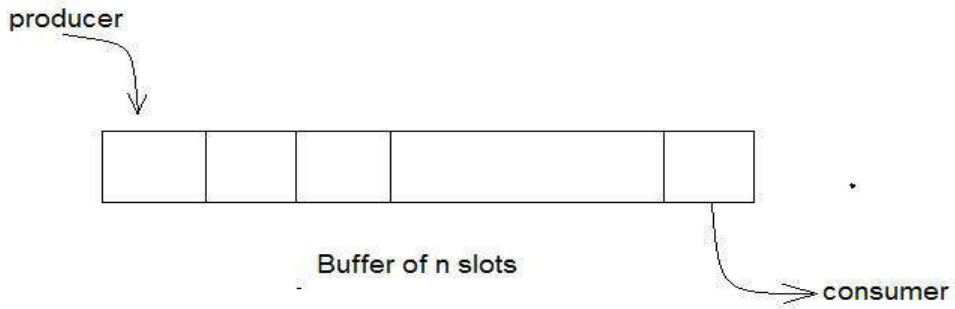
- When several tasks are waiting for the same critical resource, the task which is currently holding this critical resource is given the highest priority among all the tasks which are waiting for the same critical resource.
- Now after the lower priority task having the critical resource is given the highest priority then the intermediate priority tasks can not preempt this task. This helps in avoiding priority inversion.
- When the task which is given the highest priority among all tasks, finishes the job and releases the critical resource then it gets back to its original priority value (which may be less or equal).
- It allows the different priority tasks to share the critical resources.

Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**.
  - **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume that **P3** is holding semaphore **S** and that **P1** is waiting for **S** to be released
- Assume that **P2** is assigned the CPU and preempts **P3**
  - **P3** is still holding semaphore **S**
  - **P1** is waiting for **S** to be released
- What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used.

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n



Bounded-Buffer Problem (Cont.)

- The structure of the producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty); //wait until empty > 0 and then decrement 'empty'
    wait(mutex); //acquire lock
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex); //release a lock
    signal(full); //increment full
} while (true);
```

Bounded-Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
  wait(full); // wait until full > 0 and then decrement 'full'
  wait(mutex); // acquire the lock
  ...
  /* remove an item from buffer to next_consumed */
  ...
  signal(mutex); // release the lock
  signal(empty); // increment 'empty'
  ...
  /* consume the item in next consumed */
  ...
} while (true);
```

Readers-Writers Problem

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Readers-Writers Problem

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

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

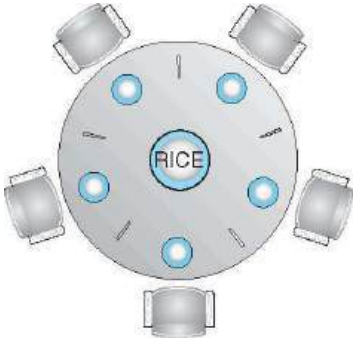
- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
           wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read count--;
        if (read_count == 0)
           signal(rw_mutex);
        signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



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

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
        //  eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
            //  think
} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

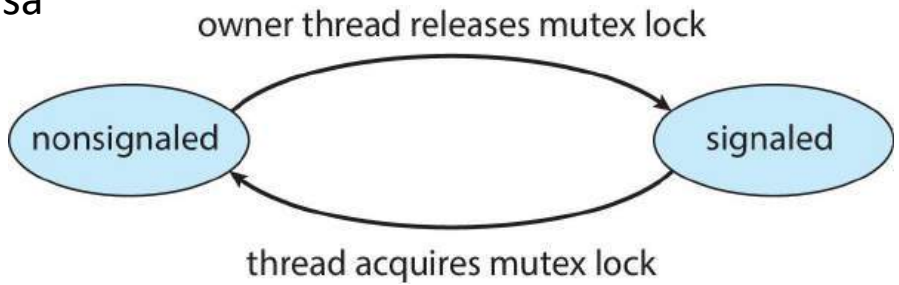
- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - For reasons of efficiency, kernel ensures that a thread will never be preempted while holding a spinlock.
- Also provides **dispatcher objects** outside the kernel, to synchronize mutex locks, semaphores, events, and timers
  - **Events**
    4. An event acts much like a condition variable (i.e notify a waiting thread when a desired condition occurs)
  - Timers notify one or more thread when time expired

Windows Synchronization - Mutex dispatcher object

- Dispatcher objects may be in either a **signaled-state (**object available and a thread will not block) or a **non-signaled state** (object not available, thread will block)
- A Relationship exists between the state of a dispatcher object and the state of a thread.
  - State of a thread changes from ready to waiting and vice-versa



Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization (Cont.)

- Atomic variables
  atomic_t is the data type for atomic integer
- Consider the variables
  atomic_t counter;
  int value;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Pthreads (POSIX) Synchronization

- Pthreads API is OS-independent, widely used on UNIX, Linux, and macOS
- It provides:
  - mutex locks
  - semaphores
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores (have actual names in the file system) can be shared by multiple unrelated processes
- Unnamed semaphores can be used only by threads belonging to the same process.

POSIX Named Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor (A high-level abstraction that provides a convenient and effective mechanism for process synchronization) , POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX Condition Variables

- Thread waiting for the condition **a == b** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

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 (< a few 100 instructions)
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread (i.e. the thread holding the lock is not currently in run state), block and sleep waiting for signal of lock being released

Solaris Synchronization (Cont.)

- Uses **condition variables**
- Uses **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
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Introduction

- Multiprogramming environment: several processes compete to limited number of resources
- A Process is holding a resource(R1) and is waiting for the resources(R2).
- The resource R2 is held by another process..
- Waiting state of processes will not change, as the requested resource is held by the waiting process.
- This situation is called deadlock

System Model

- System consists of finite number of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - *Physical resources: CPU cycles, memory space, I/O devices, printer, tape drives.*
  - *Logical resources: semaphores, mutex locks, files.*
- Each resource type R_i has W_i instances.
- Ex: if the system has 2 CPU's then CPU has 2 instances
- Each process utilizes a resource as follows:
  - **Request :** Process makes request to the resource. Eg. system call like request(), open(), wait(),allocate() etc
  - **Use:** operates on these resources
  - **Release:** process releases the resources. Eg. Using a system call like release(), close(), signal(), free etc.
- Request and release of semaphore, acquire and release of lock on mutex

Deadlock with physical resources

Example 1

- Consider a system with 3 CD RW drives.
- Suppose 3 processes(p0,p1,p2) are holding one drive each.
- What happens,
  - If a process p0 makes a request for one more drive

Example 2

- Consider a system with one printer one DVD drive.
- Process Pi is holding printer and process Pj is holding DVD drive.
- What happens if
  - Process Pi request DVD and Pj requests printer

Does dead lock occur in example 1 and 2?

Deadlock with Semaphores

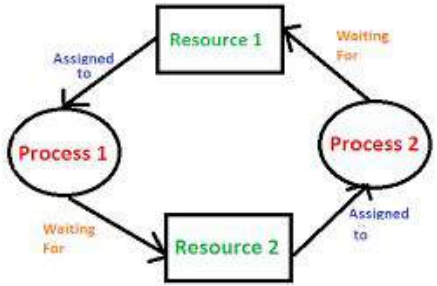
- Data:
  - A semaphore **S1** initialized to 1
  - A semaphore **S2** initialized to 1
- Two processes P1 and P2
- **P1:**

  wait(s1)

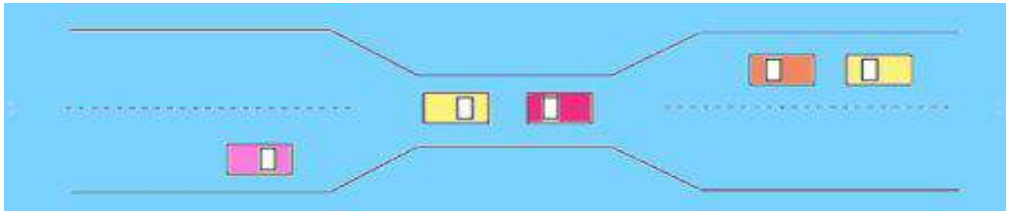
  wait(s2)
- **P2:**

  wait(s2)

  wait(s1)



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

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource (sharable resources like Read-only files do not require mutually exclusive access and thus cannot be involved in a deadlock.
- **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 $\{P_0, P_1, \ldots, P_n\}$ 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, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

- Deadlocks are described precisely with directed graphs called system resource-allocation graph

A graph consists of set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

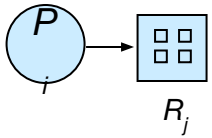
Resource-Allocation Graph (Cont.)

A set of vertices *V* and a set of edges *E*.

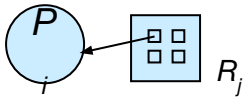
- Process

- Resource Type with 4 instances

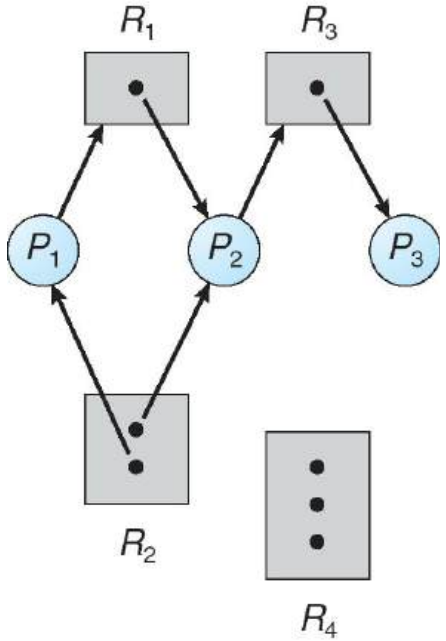
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource-Allocation Graph

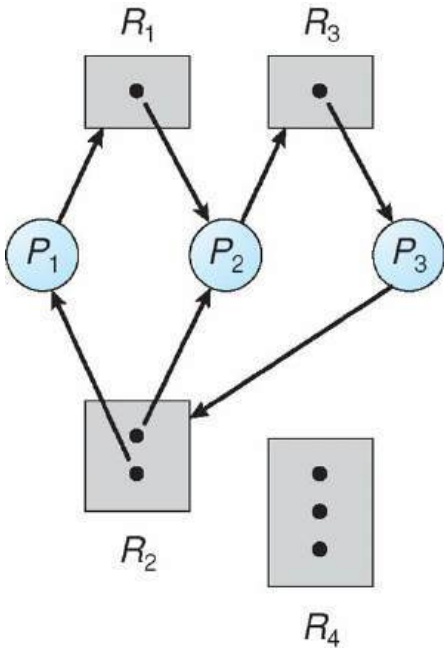


Set of process: P1,P2,P3

Set of Resources: R1,R2,R3

OPERATING SYSTEMS

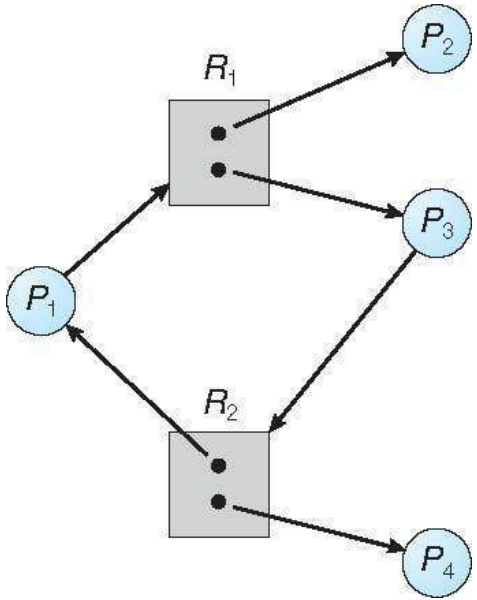
Resource-Allocation Graph With A Deadlock



P1□R1□ P2□R3□ P3□R2□P1

P2□R3□P3□R2□P2

Resource-Allocation Graph With A Cycle but No Deadlock



P1□R1□P3□R2□P1

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, the system may or may not be in a deadlocked state

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
  - Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  - Allow the system to enter into deadlocked state, detect it, and recover.
  - Ignore the problem all together and pretend that deadlocks never occur in the system.

Deadlock Prevention

4 Necessary conditions for deadlock to occur

- **Mutual Exclusion**
  - At least one resource must be non sharable
    - Ex. printers and tape drives, mutex locks
  - Sharable resources do not require mutual exclusion
    - Ex . Read-only files
- **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 allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **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 –**
  - Each resource will be assigned with a numerical number.
  - A process can request the resources increasing/decreasing order of numbering.

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Resources={R1,R2,…..Rm}
- Resources are assigned unique number
- Each process request resource in increasing order enumeration
- we define a one-to-one function F: R⟶N, where N is the set of natural numbers
- Protocol 1: Process makes request for Ri and then for Rj. Resources Rj request is allowed if and only if F(Rj)>F(Ri).
- Protocol 2: Process requesting an instance of resource type Rj, must have released any resource Ri , such that F(Ri)>=F(Rj).
- If these protocols are used then circular wait will not exist.

Circular Wait

If these two protocols are used, then the circular-wait condition cannot hold.

Proof by contradiction:

- We can demonstrate this fact that by assuming that circular wait condition cannot hold
- Consider set of processes P={P0,P1,……Pn}
- Let us consider process P0 is waiting for resource held by P1, P1 waiting for P2,……..Pn-1 is waiting resource held by Pn, Pn is waiting for resources held by P0.
- Generalizing this, Process Pi is waiting for resources Ri, Ri is held by Pi+1 and it is making request for Ri+1
- We must have F(Ri)<F(Ri+1)
- But this condition means that $F(R0) < F(R1) < \ldots < F(Rn) < F(R0)$.
- By transitivity, $F(R0) < F(R0)$, which is impossible
- Therefore no circular wait.

Circular Wait

Example

- F(tape drive)=1
- F(disk drive)=5
- F(printer)=12
- F(tape drive)<F(printer)

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Invalidating the circular wait condition is most common.
- Assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:
  first_mutex = 1
  second_mutex = 5

  code for **thread_two** could not be written as follows:

OPERATING SYSTEMS

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount) {
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

OPERATING SYSTEMS

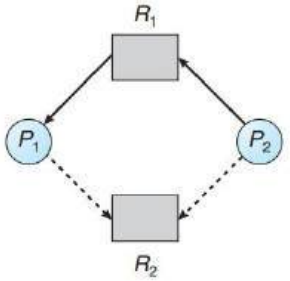
Deadlock Detection

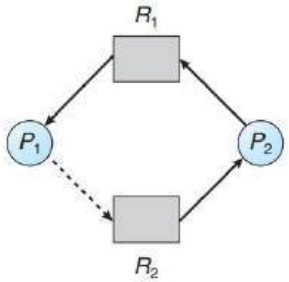
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

OPERATING SYSTEMS

Single Instance of Each Resource Type

- In a resource allocation graph, a claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge is represented in the graph by a dashed line.
- 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

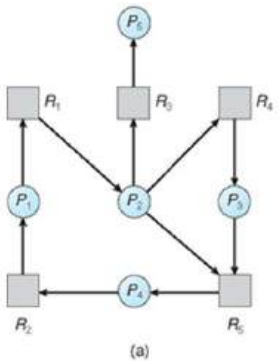




OPERATING SYSTEMS

Resource-Allocation Graph and Wait-for Graph

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q
- A **deadlock exists** in the system if and only if the **wait-for graph contains a cycle**.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations.



Several Instances of a Resource Type

- **Available***:* A vector of length ***m*** indicates the number of available resources of each type. If Available[j] = k, then k instances of resource type R_j are available
- **Max**: An n × m matrix defines the maximum demand of each process. If Max[i][j] = k, then process P_i may request at most k instances of resource type R_j .
- **Allocation***:* An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process P_i is currently allocated k instances of resource type R_j
- **Request***:* An ***n* x *m*** matrix indicates the current request of each process. If ***Request* [*i*][*j*] = *k***, then process $\boldsymbol{P_i}$ is requesting ***k*** more instances of resource type $\boldsymbol{R_j}$.

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 ***Allocation***$_i$ ***≠ 0***, then
   Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a)***Finish*[*i*] == *false***
   (b)***Request***$_i$ ***≤ Work***

   If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work = Work + Allocation***$_i$
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked. If **Finish[i] == true** for all i, then the system is in a safe state

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

OPERATING SYSTEMS

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation			*Request*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i]* = *true*** for all ***i***

Example of Detection Algorithm (Cont.)

- P_2 requests an additional instance of type **C**

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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.

OPERATING SYSTEMS

Deadlock Avoidance

- Deadlock-prevention algorithms, prevent deadlocks by limiting how requests can be made.
- The limits ensure that at least one of the necessary conditions for deadlock cannot occur.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.
- With this additional knowledge of complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

Deadlock Avoidance

- The simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need.
- Given this apriori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the processes.

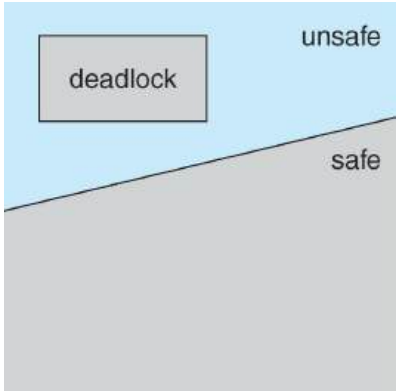
Safe State

- A state is ***safe*** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
- A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if, for each *Pi* , the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with $j < i$.
- In this situation, if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished.
- When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
- When *Pi* terminates, *Pi*+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be ***unsafe***

OPERATING SYSTEMS

Safe, Unsafe and Deadlock States

- A safe state is not a deadlocked state.
- Conversely, a deadlock state is an unsafe state.
- Not all unsafe states are deadlocks.
- An unsafe state may lead to a deadlock.
- As long as the state is safe, the OS can avoid unsafe states.
- In an unsafe state, the OS cannot prevent processes from requesting resources in such a way that a deadlock occurs.
- The behavior of processes controls unsafe states.
- If a system is in safe state no deadlocks
- If a system is in unsafe state possibility of deadlock
- Goal for Avoidance ensure that a system will never enter an unsafe state.



OPERATING SYSTEMS

Example - Safe, Unsafe and Deadlock States

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- Consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2.
- Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.
- Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives. (Thus, there are three free tape drives.)
- At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition. Process *P*1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *P*0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P*2 can get all its tape drives and return them
- At time *t*1, process *P*2 requests and is allocated one more tape drive. The system is no longer in a safe state.
- Since process *P*0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process *P*2 may request six additional tape drives and have to wait, resulting in a deadlock

OPERATING SYSTEMS

Deadlock avoidance algorithms

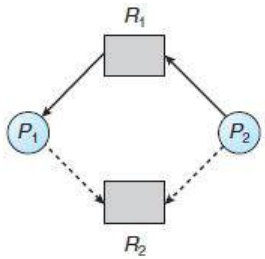
- Avoidance algorithms ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.
- Thus, resource utilization may not be optimal

Resource Allocation Graph algorithm

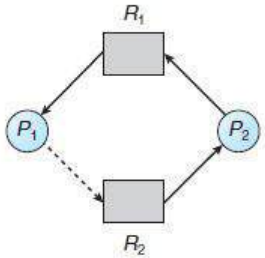
- In addition to the request and assignment edges in a resource allocation graph, we introduce a new type of edge, called a **claim edge**
- A claim edge *Pi* ⟶ *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process *Pi* requests resource *Rj* , the claim edge *Pi* ⟶ *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi* , the assignment edge *Rj* ⟶ *Pi* is reconverted to a claim edge *Pi* ⟶ *Rj*
- The resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
- Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* ⟶ *Rj* to an assignment edge *Rj* ⟶ *Pi* does not result in the formation of a cycle in the resource-allocation graph.

OPERATING SYSTEMS

Resource Allocation Graph algorithm example



Resource-allocation graph for deadlock avoidance.



An unsafe state in a resource-allocation graph.

- Consider the above resource-allocation graph.
- Suppose that *P*2 requests *R*2. Although *R*2 is currently free, we cannot allocate it to *P*2, since this action will create a cycle in the graph below.
- A cycle indicates that the system is in an unsafe state.
- If *P*1 requests *R*2, then a deadlock will occur.

Banker's algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.
- We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types

OPERATING SYSTEMS

Banker's algorithm data structures

- **Available –** A vector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] equals *k,* then *k* instances of resource type *Rj* are available.
- **Max**. - An *n* × *m* matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k,* then process *Pi* may request at most *k* instances of resource type *Rj* .
- **Allocation** - An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k,* then process *Pi* is currently allocated *k* instances of resource type *Rj* .
- **Need** - An *n* × *m* matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] equals *k,* then process *Pi* may need *k* more instances of resource type *Rj* to complete its task.
- ***Need***[*i*][*j*] = ***Max***[*i*][*j*] **−** ***Allocation***[*i*][*j*].
- Treat each row in the matrices ***Allocation*** and ***Need*** as vectorsThe vector ***Allocation****i* specifies the resources currently allocated to process *Pi* ; the vector ***Need****i* specifies the additional resources that process *Pi* may still request to complete its task
- These data structures vary over time in both size and value.

OPERATING SYSTEMS

Banker's algorithm for determining whether or not a system is in a safe state

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n,* respectively.

   Initialize ***Work*** = ***Available*** and ***Finish***[*i*] = ***false*** for *i* = 0, 1, ..., *n* − 1.

2. Find an index *i* such that both

   a. ***Finish***[*i*] == ***false***

   b. ***Need****i* ≤***Work***

   If no such *i* exists, go to step 4

3. ***Work*** =***Work*** + ***Allocation****i*

Finish[*i*] = ***true***

Go to step 2.

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

- This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Banker's algorithm for determining whether requests can be safely granted

Let ***Request****i* be the request vector for process *Pi* . If ***Request****i* [*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj* .

When a request for resources is made by process *Pi* , the following actions are taken:

1. If ***Request****i* ≤ ***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

   Available = ***Available–Request****i* ;

   Allocation*i* = ***Allocation****i* + ***Request****i* ;

   Need*i* = ***Need****i* –***Request****i* ;

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources.

If the new state is unsafe, then *Pi* must wait for ***Request****i* **,** and the old resource-allocation state is restored.

OPERATING SYSTEMS

Banker's algorithm example

- Consider a system with five processes *P*0 through *P*4 and three resource types *A, B,* and *C.*
- Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances.
- Suppose that, at time *T*0, the following snapshot of the system has been taken

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.
- A request for (3,3,0) by *P*4 cannot be granted, since the resources are not available.
- A request for (0,2,0) by *P*0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

Signals

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously depending on the source of and the reason for the event being signaled
- All signals, whether synchronous or asynchronous, follow the same pattern:
  1. A signal is generated by the occurrence of a particular event.
  2. The signal is delivered to a process.
  3. Once delivered, the signal must be handled.

OPERATING SYSTEMS

Signals – synchronous and asynchronous

- Examples of synchronous signal include illegal memory access and division by 0.
- If a running program performs either of these actions, a signal is generated.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire.
- Typically, an asynchronous signal is sent to another process.

Signal handlers

- A signal may be ***handled*** by one of two possible handlers:
  1. A default signal handler
  2. A user-defined signal handler
- Every signal has a **default signal handler** that the kernel runs when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.
- Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

Signal handling

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- Delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- In general, the following options exist:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.

Signal handling

- The method for delivering a signal depends on the type of signal generated.
- For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.
- However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>) should be sent to all threads.
- The standard UNIX function for delivering a signal is

  kill(pid t pid, int signal)
- This function specifies the process (pid) to which a particular signal (signal) is to be delivered

Signal handling

- Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.
- An asynchronous signal may be delivered only to those threads that are not blocking it.
- Because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.
- POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

  pthread kill(pthread t tid, int signal)

Signals in windows

- Windows does not explicitly provide support for signals, it allows to emulate them using **asynchronous procedure calls (APCs)**.
- The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.
- An APC is roughly equivalent to an asynchronous signal in UNIX
- The APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process

Signals in Linux

- A signal is a kind of software interrupt, used to announce asynchronous events to a process
- **SIGINT** is a signal generated when a user presses Control-C. This will terminate the program from the terminal.
- **SIGALRM** is generated when the timer set by the alarm function goes off.
- **SIGABRT** is generated when a process executes the abort function.
- **SIGSTOP** tells LINUX to pause a process to be resumed later.
- **SIGCONT** tells LINUX to resume the processed paused earlier.
- **SIGSEGV** is sent to a process when it has a segmentation fault.
- **SIGKILL** is sent to a process to cause it to terminate at once.