

Operating Systems

Overview of Syllabus

Unit 1- Introduction Introduction What Operating System do, Operating System structure, Operating system Operations. System Structures Operating system services, **System Calls**, Types of System calls Process Management Process concept, Process scheduling, Operations on processes

Unit-2 Multithreaded programming Overview, Multicore programming, Multithreading models, Thread libraries - pthreads CPU scheduling and Process Synchronization Basic concepts, scheduling criteria, scheduling algorithms-FCFS, SJF, RR, priority, Real-time CPU scheduling

Unit 3- Process Synchronization Background, The Critical section problem, Peterson's Solution Process Synchronization hardware, Mutex locks, Semaphores, Classic problems of synchronization

Unit 4- Main Memory Management Background, Swapping, Contiguous memory allocation, Segmentation, Paging, Structure of page table. Virtual memory Background, Demand Paging, Copy-on-write, Page replacement, Allocation of frames, Thrashing

Unit -5 File Systems File Naming, File Structure, File Types, File Access, File Attributes, File Operations, An example program using File-System calls, File-System Layout, Implementing Files

Why we need Operating System

- Communicate with the computer
- Learn concurrency, resource management, performance analysis, interfaces with complexity and computer system design of a computer system Operating System should be learnt
- Build or modify real operating system.
- Tune application performance. Understanding the services offered by an operating system will influence how you design applications.
- Administer and use system well. You will develop a better understanding of the structure of modern computing systems, from the hardware level through the operating system level and onto the applications level.
- Can apply techniques used in an OS to other areas;
 - interesting, complex data structures
 - conflict resolution
 - concurrency
 - resource management
- Challenge of designing large and complex systems

A program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

Execute user programs and make solving user problems easier

Make the computer system convenient to use

Use the computer hardware in an efficient manner

- **Control Freak** – Must never ever loose control of the hardware

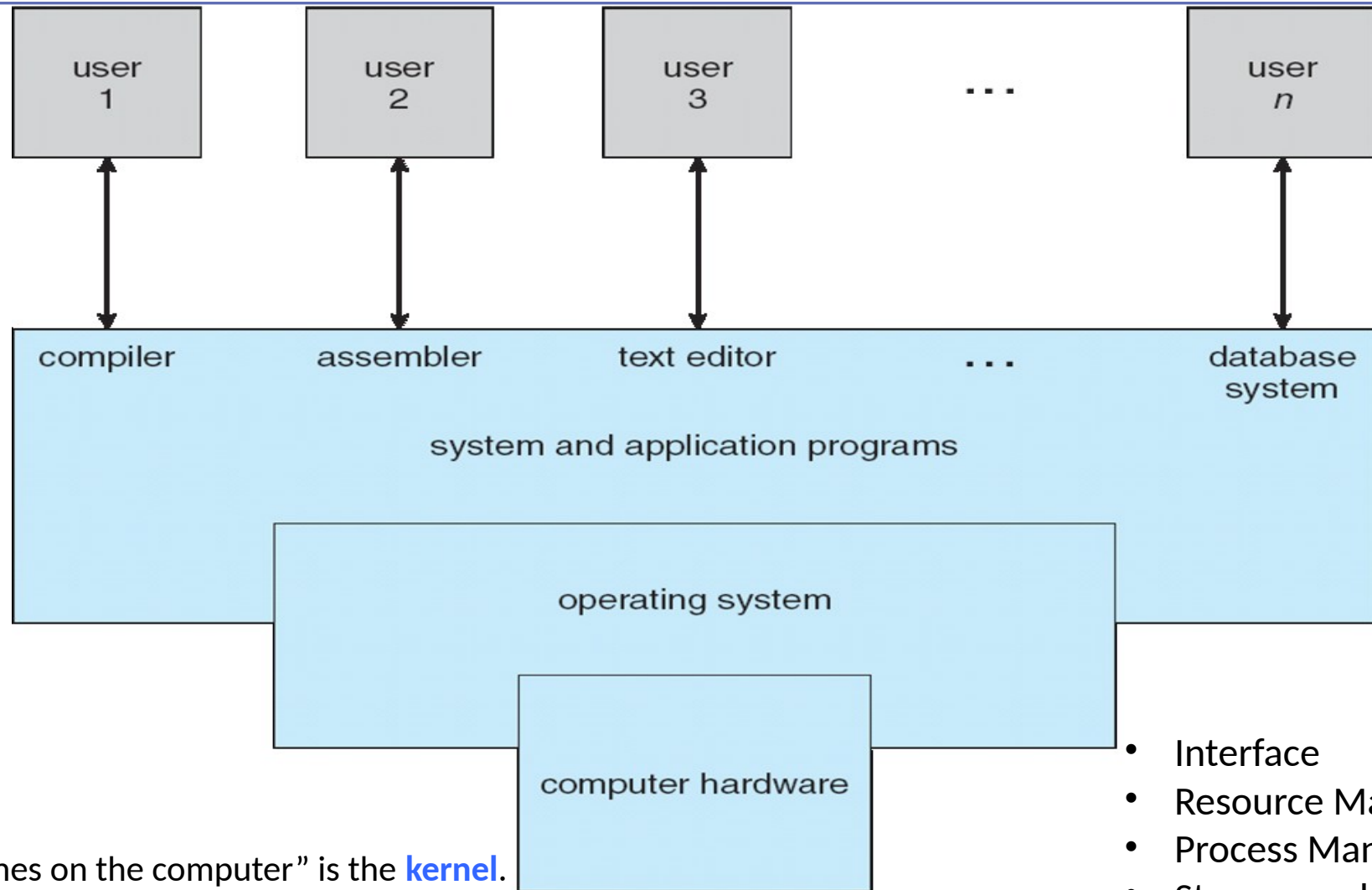
- **Resource Manager**

- Give resources to applications
- Take resources from applications
- Protection and Security

- **Great Pretender**

- Make resources look different
- Make finite resources appear as infinite resources

Computer System Architecture



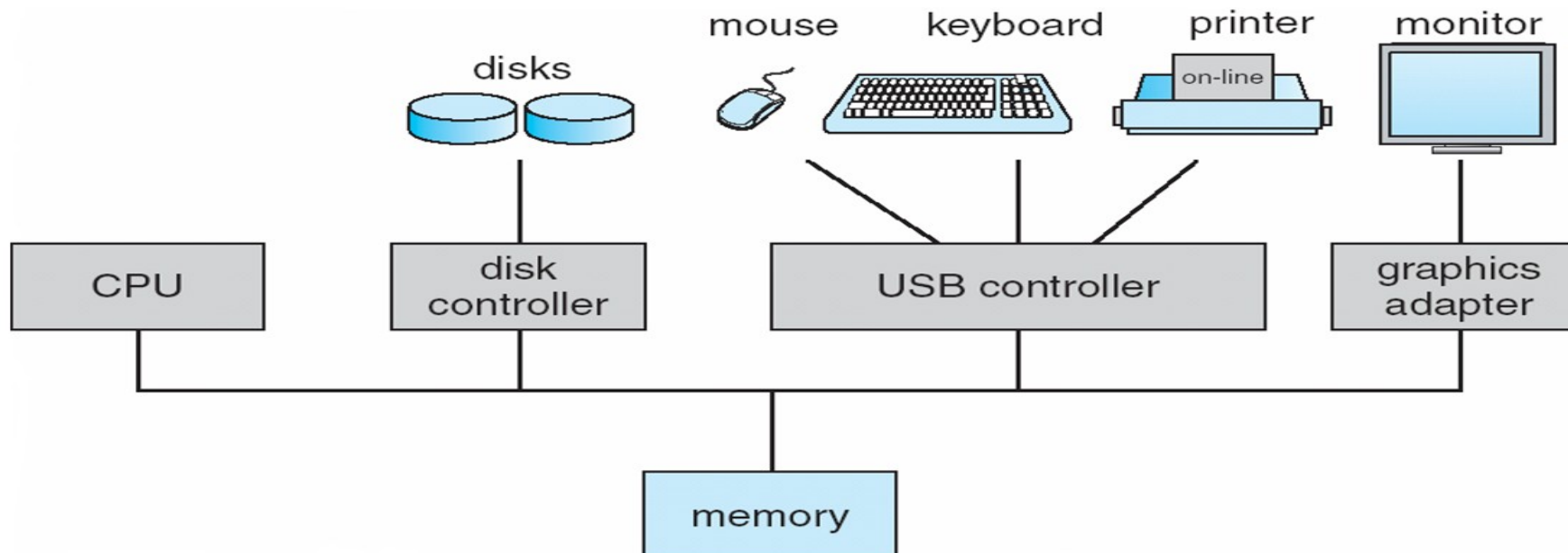
the one program running at all times on the computer” is the **kernel**.

Application programs – define the ways in which the system resources are used to solve the computing problems of the users

- Interface
- Resource Management
- Process Management
- Storage and Memory Management
- Security

Computer-system operation

- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory cycles
- To ensure the memory access to all the devices we have **Memory Controller**



Bootstrap Program

- Initial Program that runs up when the system is powered
- Typically stored in ROM or EPROM, generally known as **firmware**
- Initializes all aspects of system
- Loads operating system kernel and starts execution

Interrupt

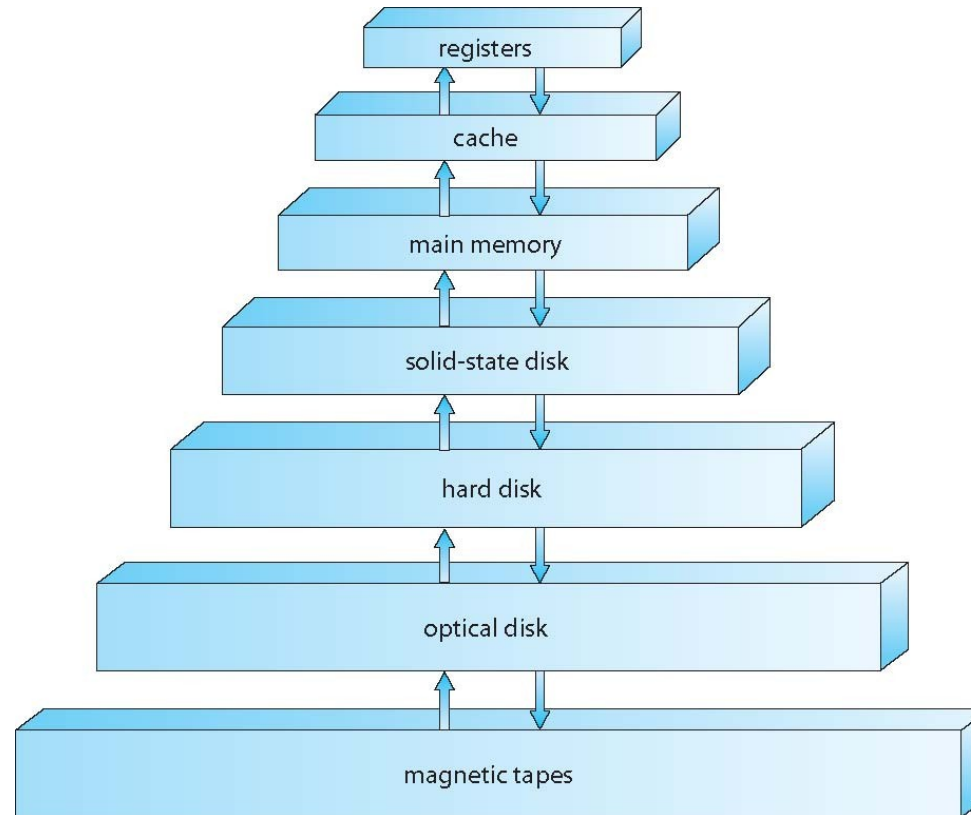
- The occurrence of the event is signaled by interrupt from Hardware and Software
- Hardware may trigger an interrupt by sending the signal to CPU

System Call (Monitor Call)

- Software may trigger the interrupt by sending the special signal that is called as system call or monitor call

Storage Structure

- Expensive, Larger in Size,
- Access time increases



- Expensive,
Small in Size,
Fast

Storage systems organized in hierarchy

Speed

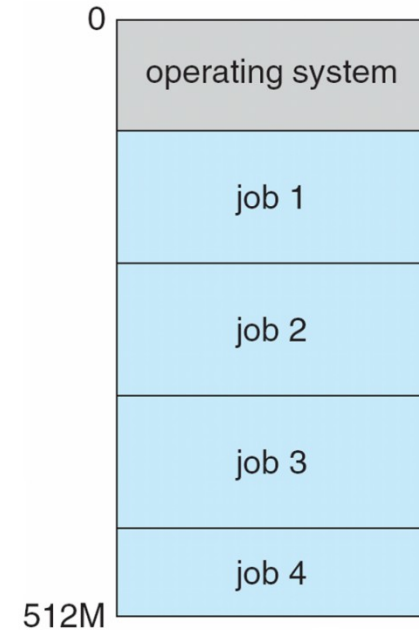
Cost

Volatility

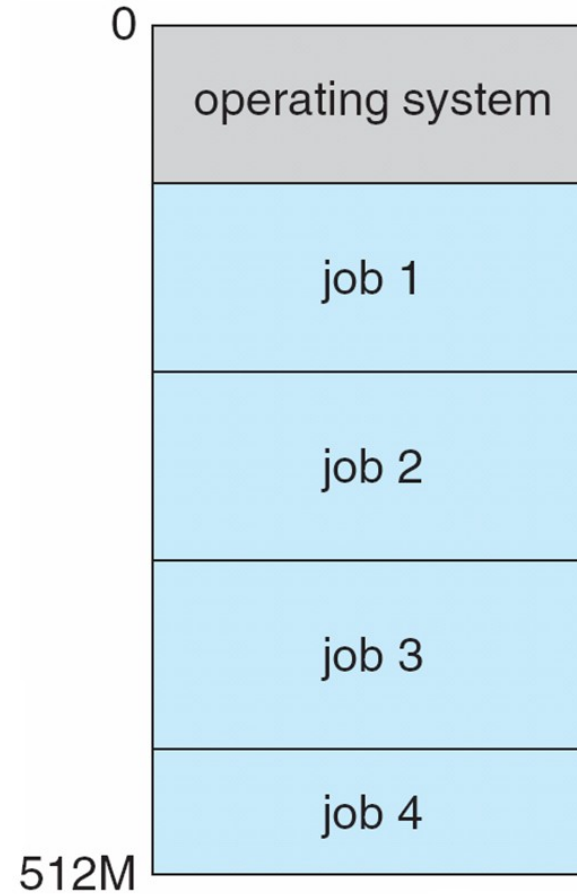
Caching – copying information into faster storage system; main memory can be viewed as a cache for secondary storage

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**
-
- **Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system**



Memory Layout for Multiprogrammed System



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

CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running

- **Response time** should be < 1 second
- A program loaded into memory and executing is called. ⇒ **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 m



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
- The operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program that is running.

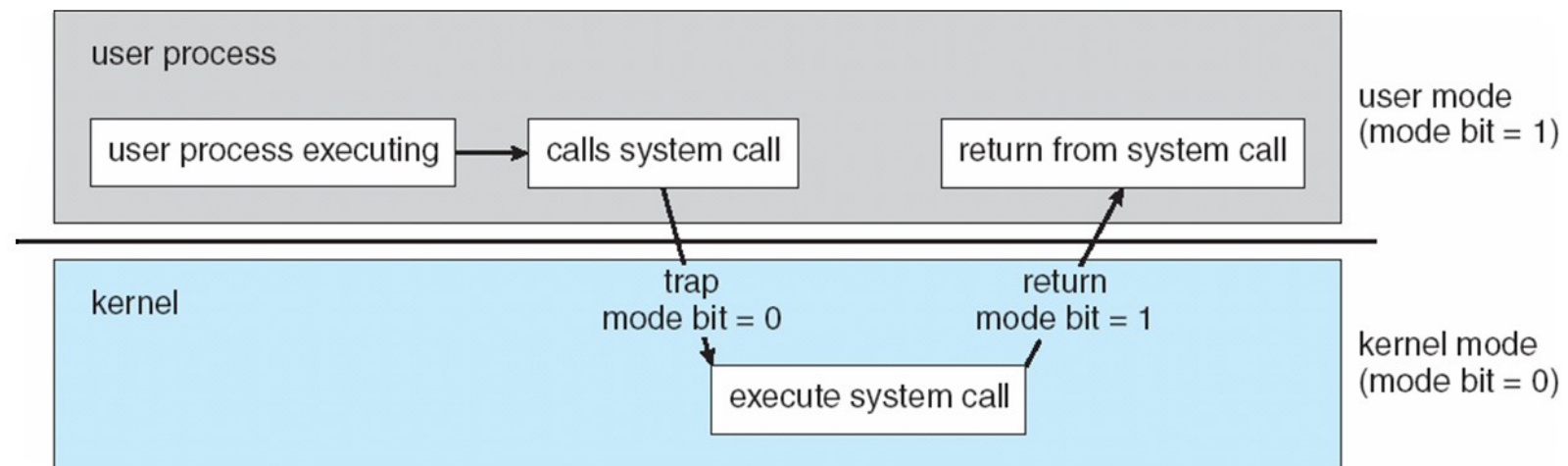
Dual-mode operation allows OS to protect itself and other system components

User mode and **kernel mode**

Mode bit provided by hardware

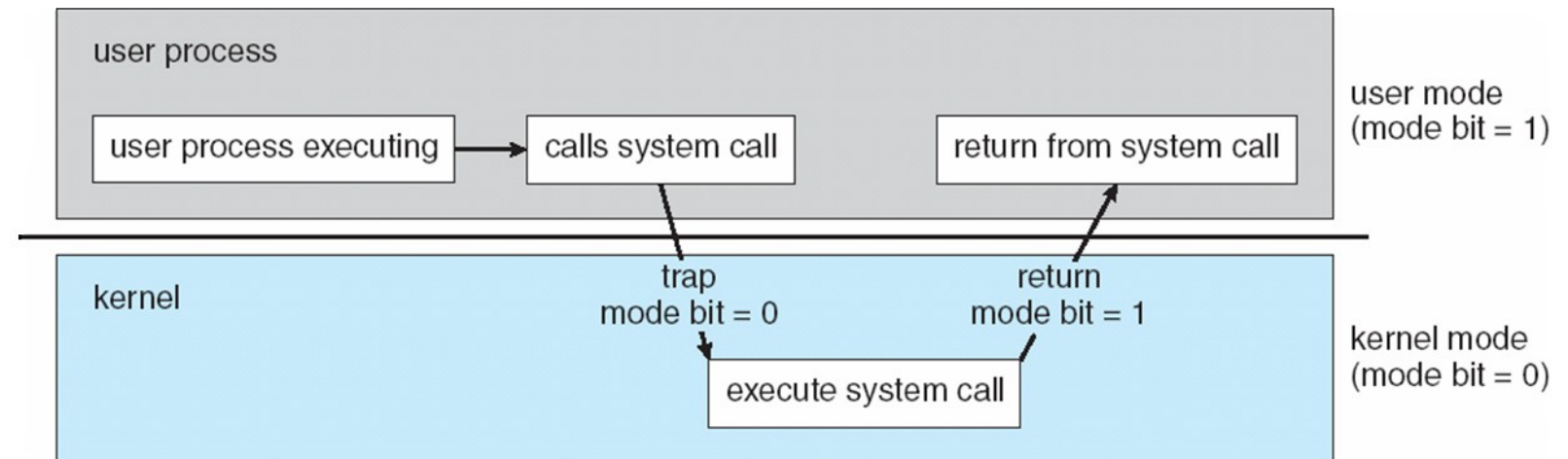
Kernel (0) and User(1)

When the computer system is executing on behalf of a user application, the system is in user mode. a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfil the request.



Operating-System Operations (cont.)

- At system boot time, the hardware starts in kernel mode.
- The operating system is then loaded and starts user applications in user mode.
- Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).
- Thus, whenever the operating system gains control of the computer, it is in kernel mode.
- The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.



- 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

- Timer to prevent infinite loop / process hogging resources
- operating system maintains control over the CPU. Prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system.
- To accomplish this goal, we can use a timer.
 - Timer is set to interrupt the computer after some time period
 - Keep a counter that is decremented by the physical clock.
 - Operating system set the counter (privileged instruction)
 - When counter zero generate an interrupt
 - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Operating systems provide an environment for execution of programs and services to programs and users

One set of operating-system services provides functions that are helpful to the user:

User interface - Almost all operating systems have a user interface (**UI**).

Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**,

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)

Error detection - OS needs to be constantly aware of possible errors

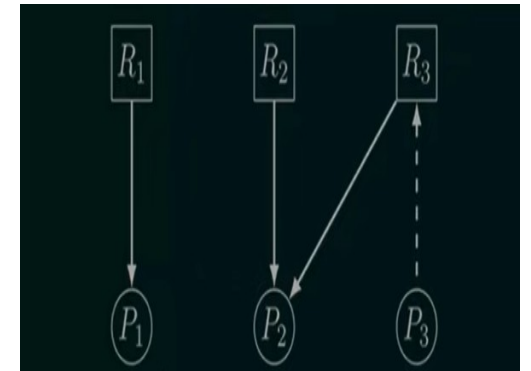
May occur in the CPU and memory hardware, in I/O devices, in user program

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

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

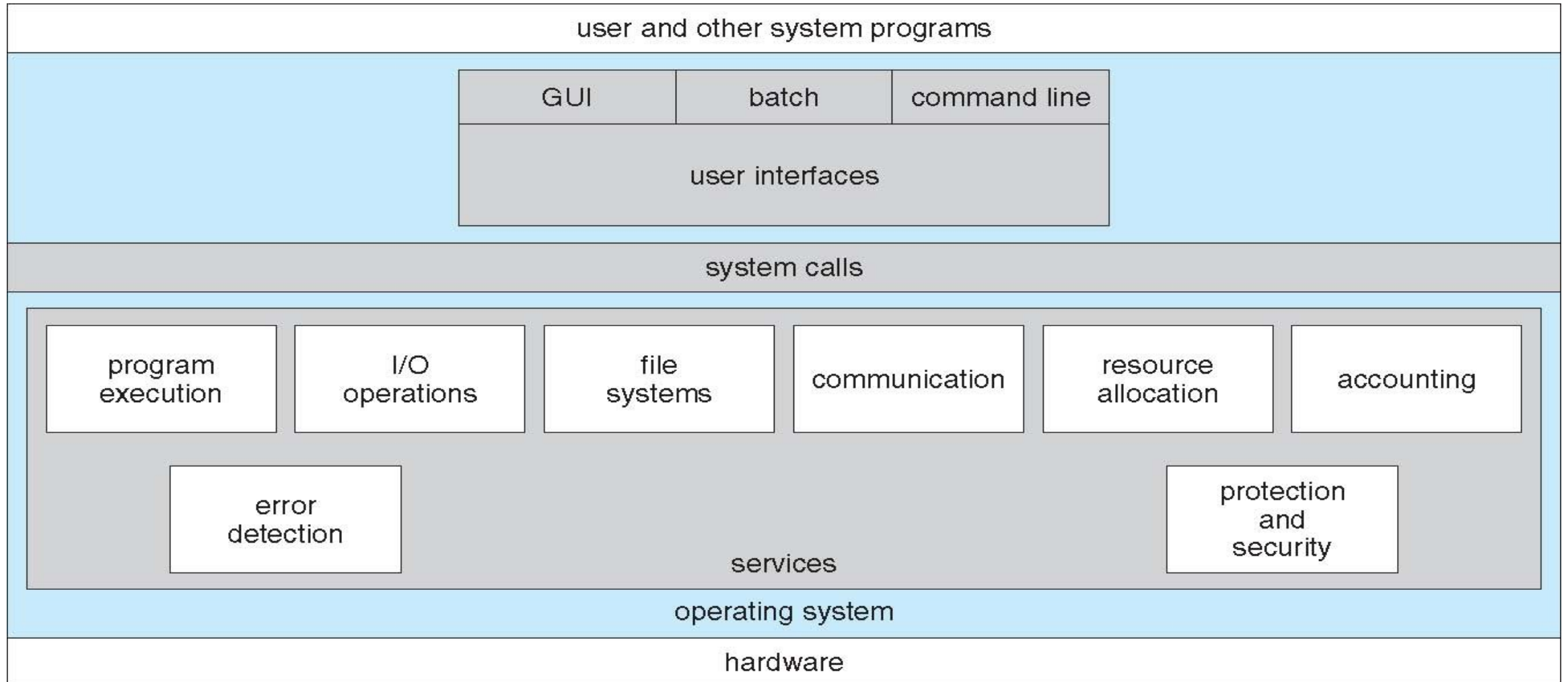
Protection and security - The owners of information stored in a multiuser or networked computer system may

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



CLI or **command interpreter** allows direct command entry

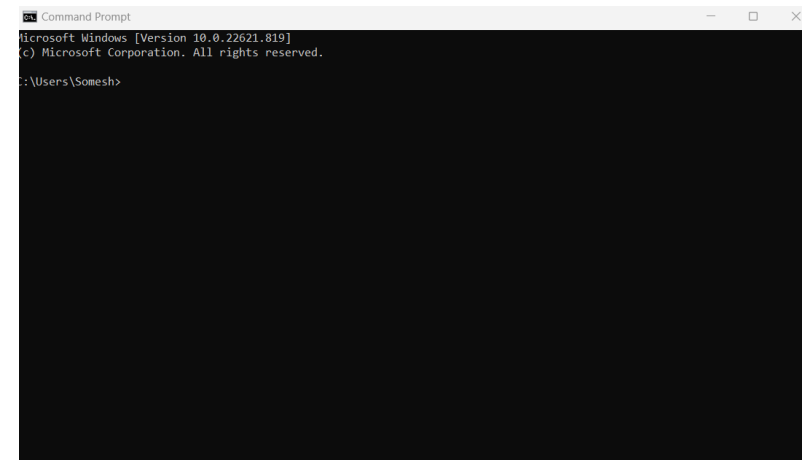
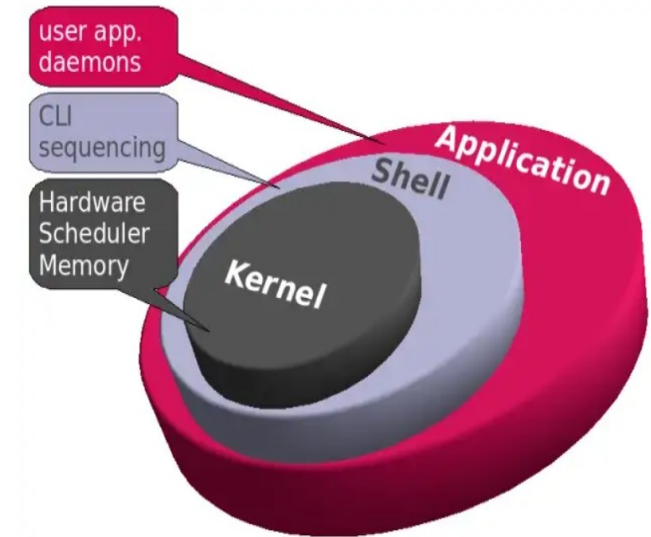
Sometimes implemented in kernel, sometimes by systems program

shells - interface to the operating system

Primarily fetches a command from user and executes it

Sometimes commands built-in, sometimes just names of programs

If the latter, adding new features doesn't require shell modification



User-friendly **desktop** metaphor interface

Usually mouse, keyboard, and monitor

Icons represent files, programs, actions, etc

Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**))

Invented at Xerox PARC

Many systems now include both CLI and GUI interfaces

Microsoft Windows is GUI with CLI “command” shell

Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available

Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)


What is System Call in Operating System?

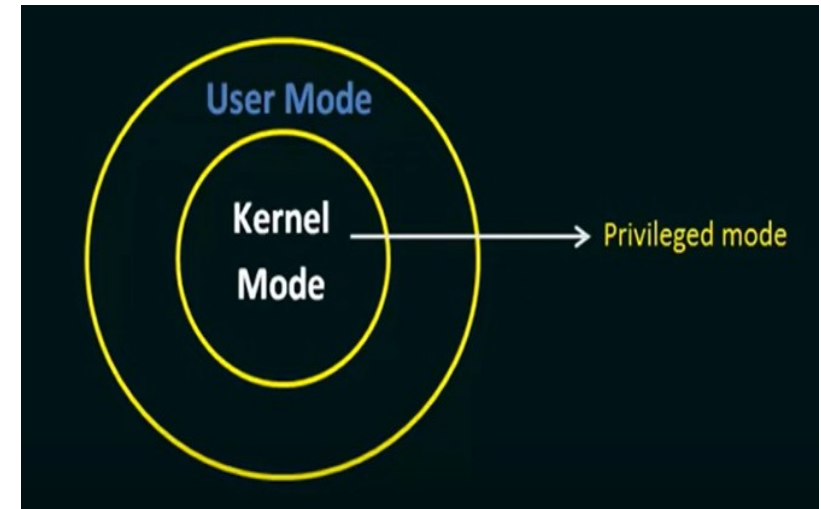
A **system call** is a mechanism that provides the interface between a process and the operating system. It is a **programmatic method in which a computer program requests a service from the kernel of the OS.**

System call offers the services of the operating system to the user programs via API (Application Programming Interface).

Typically written in a high-level language (C or C++)

System calls are the only entry points for the kernel system.


 User \longleftrightarrow Kernel ===== **Mode Shifting**



Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use

Example of the system call to write the simple program to read the data from source file and copy to the destination file



Step-1 ---- Get the Input File Name (From which the data has to be copied)

Step-2 ---- Prompt on the Screen (User to Enter the Input File Name)

Step-3 ----- Accept the Input User has entered (Input File Name)

Step-4 -----Acquire the Output file name

Step-5----- Prompt on the Screen (User to Enter the Output File Name)

Step-6 ----- Accept the Input User has entered (Output File Name)

Step - 7 --- Open the Input File

Step-8 -- If the Input file doesn't exist the pop up the error

Step - 9 -- Read from the Input file

Step-10 - Write to the Output file name (Step 9 and 10 will be in loop until all the data is copied)

Step-11 - Show the message the coping is done

Step- 12 - Close both the files

To perform all these steps we will be requiring the system calls

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t  read(int fd, void *buf, size_t count)
```

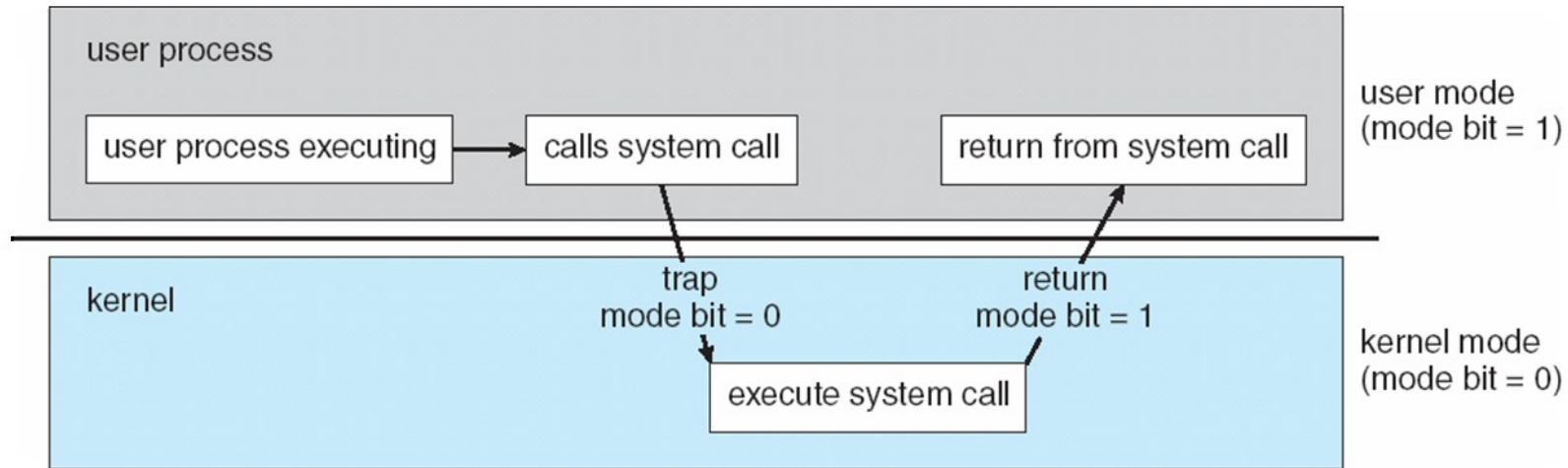
return value	function name	parameters
-----------------	------------------	------------

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

How System Call Works



Working of System call As you can see in the above-given System Call example diagram.

Step 1) The processes executed in the user mode till the time a system call interrupts it.

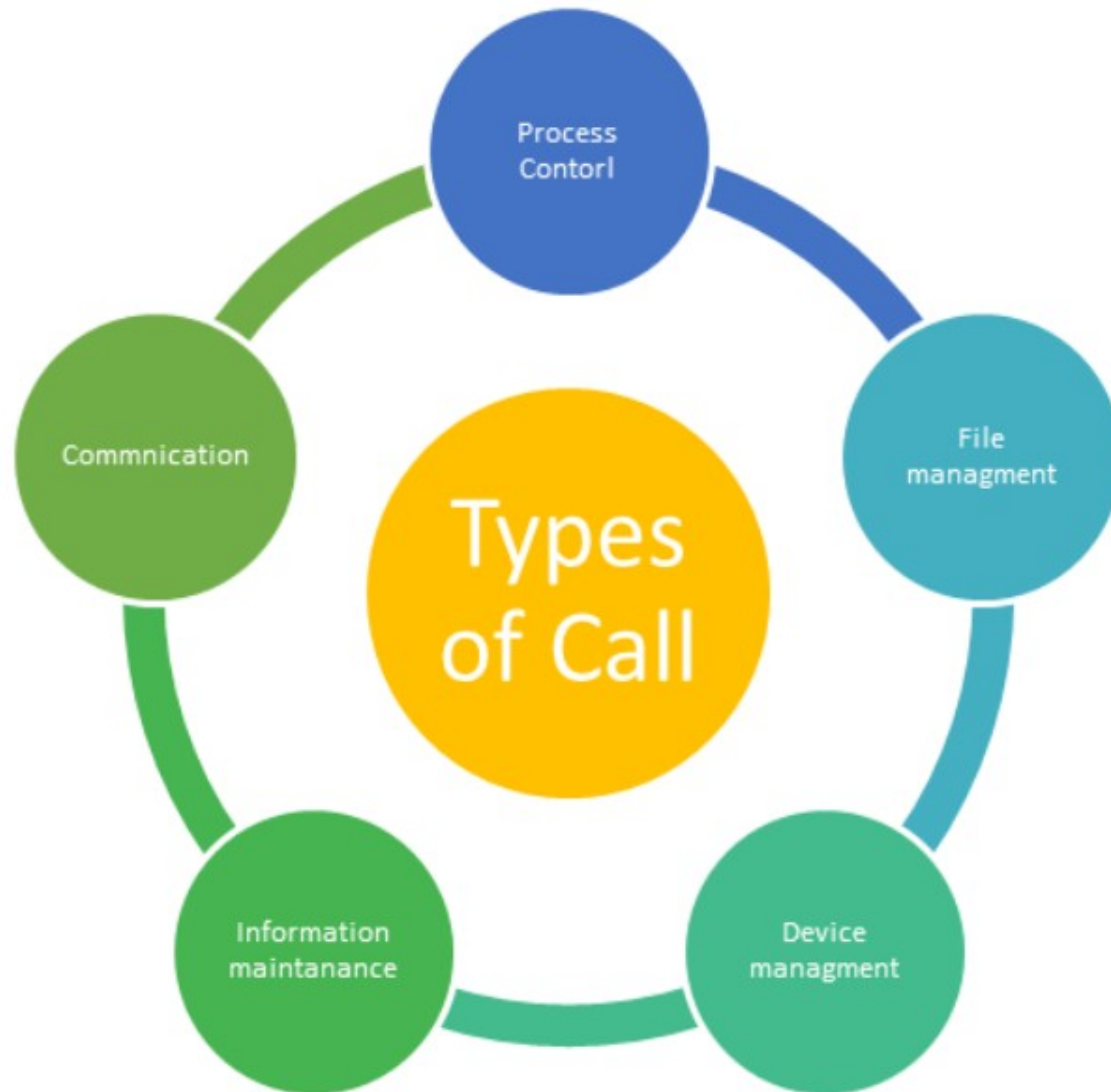
Step 2) After that, the system call is executed in the kernel-mode on a priority basis.

Step 3) Once system call execution is over, control returns to the user mode.,

Why we require System calls

- Reading and writing from files demand system calls.
- If a file system wants to create or delete files, system calls are required.
- System calls are used for the creation and management of new processes.
- Network connections need system calls for sending and receiving packets.
- Access to hardware devices like scanner, printer, need a system call.

Types of System calls



Process Control

This system calls perform the task of process creation, process termination, etc.

Functions:

- End(Completes its execution)and Abort(When the errors occurs)
- Load and Execute the process
- Create Process and Terminate Process (Getting the Process Attributes and Set attributes)
- Wait and Signal Event
- Allocate and free memory

Process Control

This system calls perform the task of process creation, process termination, etc.

Functions:

- End(Completes its execution)and Abort(When the errors occurs)
- Load and Execute the process
- Create Process and Terminate Process (Getting the Process Attributes and Set attributes)
- Wait and Signal Event
- Allocate and free memory (Allocate the

File Management

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

Device Management

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions:

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

Information Maintenance

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attribute

Communication:

These types of system calls are specially used for interprocess communications.

Functions:

- **Create, delete communications connections**
- **Send, receive message**
- **Help OS to transfer status information**
- **Attach or detach remote devices**

There are two common models of interprocess communication

Message passing model and the Shared-memory model.

- Messages can be exchanged between the processes either directly or indirectly through a common mailbox.
- **a connection must be opened.**
- The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network.
- **Process has to be identified** (get hostid() and get processid())
- The recipient process usually must give its permission for communication to take place with an **accept connection()**
- The source of the communication, known as the **client**, and the receiving daemon, known as a **server**
- Then exchange messages by using **read message()** and **write message()** system calls. The **close connection()** call

Communication:

Shared Memory Model

In the shared-memory model, processes use **shared memory create** and **shared memory attach** system calls to create and gain access to regions of memory owned by other processes.

Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.

Shared memory requires that two or more processes agree to remove this restriction.

They can then exchange information by reading and writing data in the shared areas.

The form of the data and the location are determined by the processes and are not under the operating system's control.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously

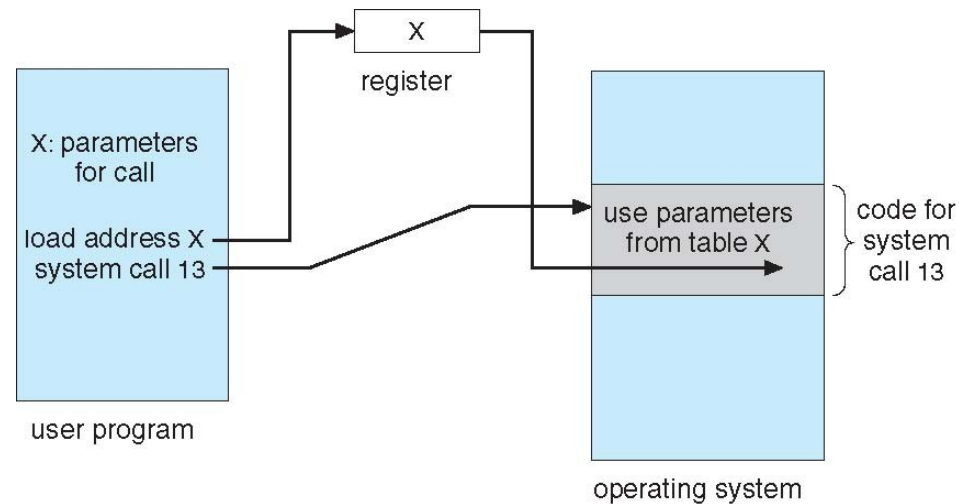
Three general methods used to pass parameters to the OS

1. Simplest: pass the parameters in registers

In some cases, may be more parameters than registers

2. Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register

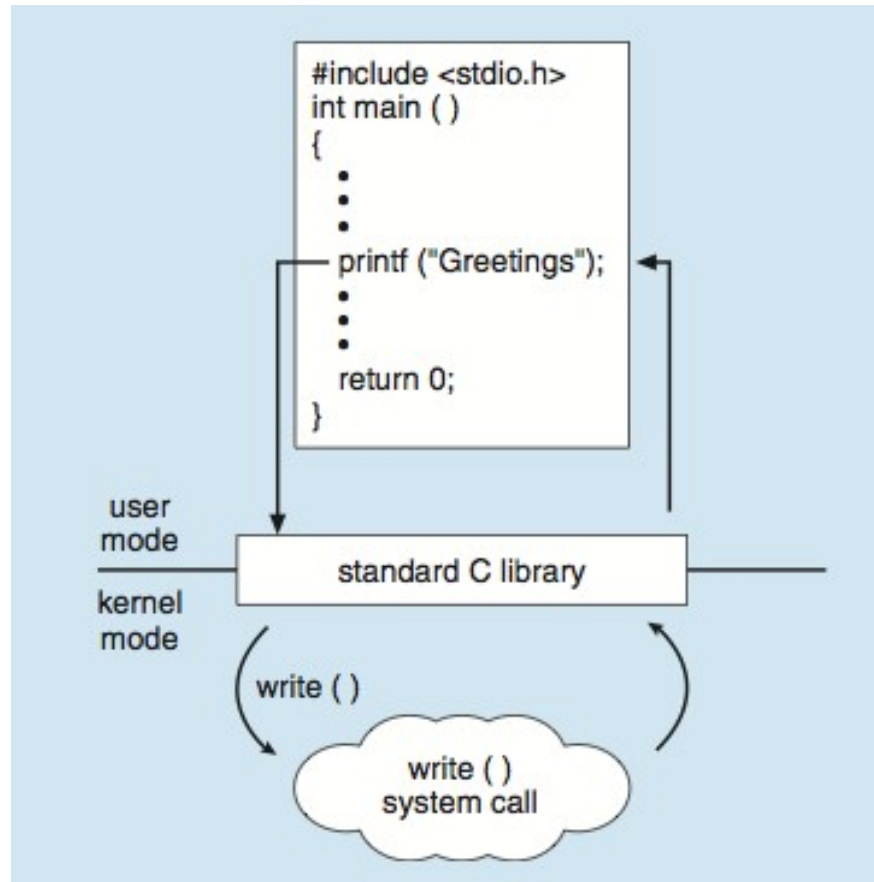
This approach taken by Linux and Solaris



3. Parameters should be pushed on or popped off the stack by the operating system

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

C program invoking printf() library call, which calls write() system call

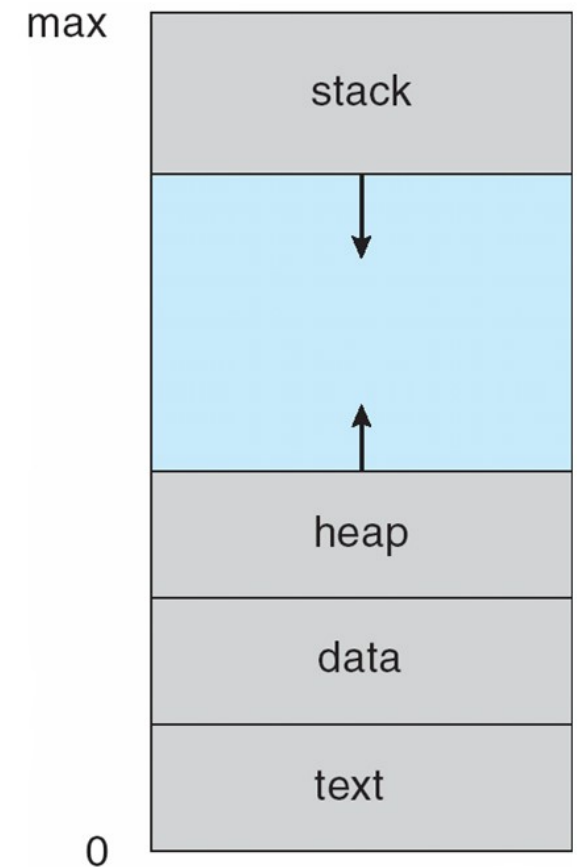


As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program.

Process – a program in execution;

Multiple parts

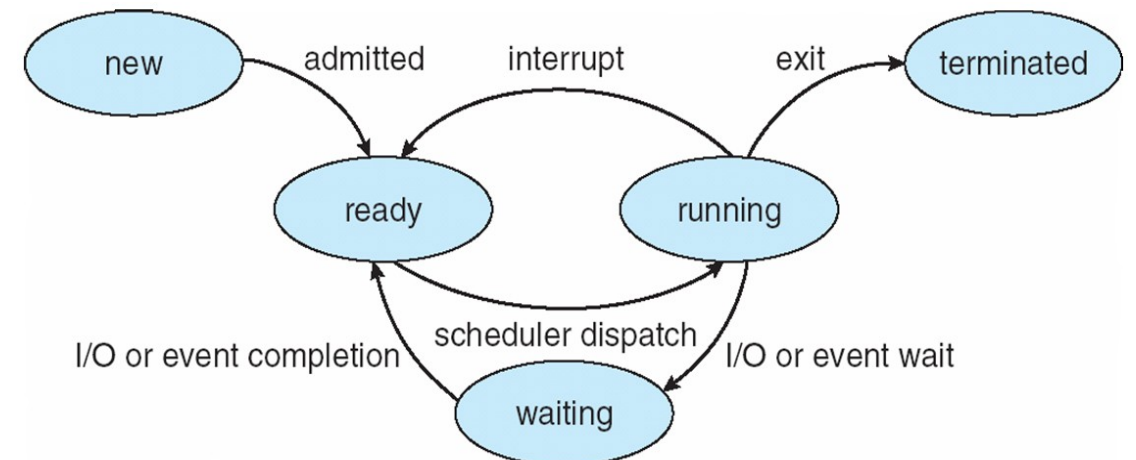
- The program code, also called **text section (Mainly Stores the Program)**
- **Data section** containing global and static variables
- **Heap** containing memory dynamically allocated during run time
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
 - Function parameters, return addresses, local variables



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

The state of a process is defined in part by the current activity of that process

- **new**: The process is being created (Program is ready for execution)
- **ready**: The process is waiting to be assigned to a processor (Waiting for CPU)
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **terminated**: The process has finished execution



Each Process is represented by the PCB (also called **task control block**)

Process state – running, waiting, etc

Program counter – location of instruction to next execute

CPU registers – Gives details of the registers used by the specific process

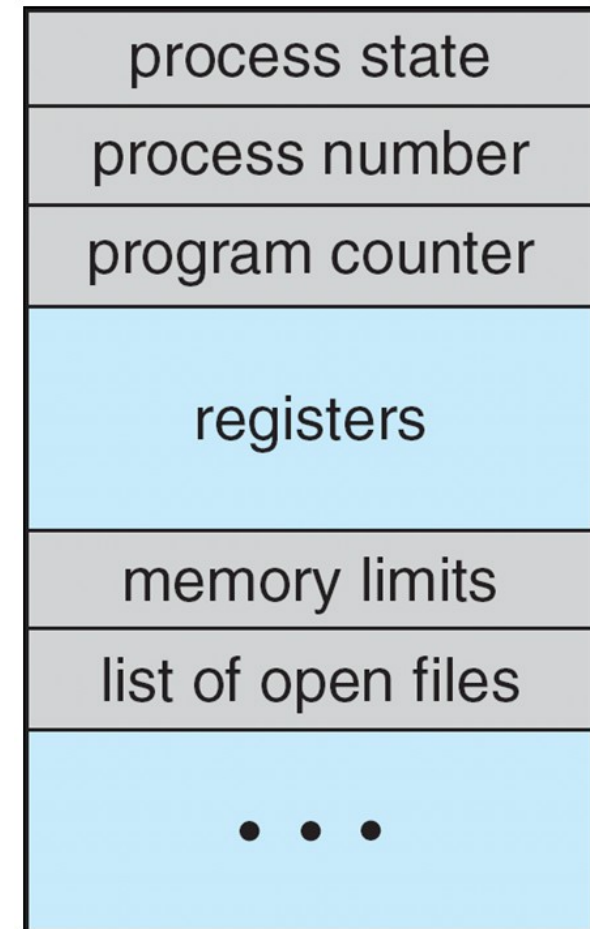
CPU scheduling information- priorities, scheduling queue pointers

(Basically determines which process should execute first)

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



Each Process is represented by the PCB (also called **task control block**)

Process state – running, waiting, etc

Program counter – location of instruction to next execute

CPU registers – Gives details of the registers used by the specific process

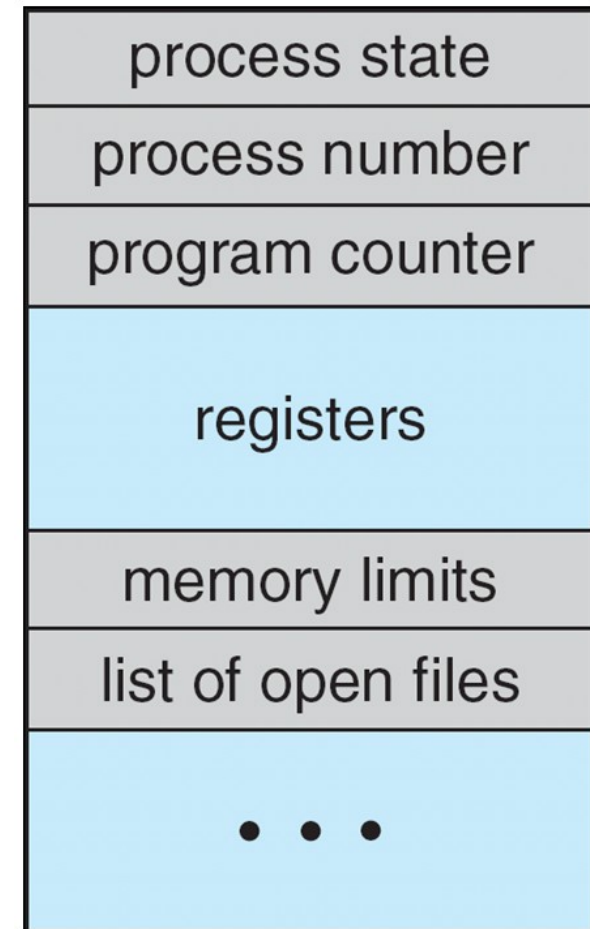
CPU scheduling information- priorities, scheduling queue pointers

(Basically determines which process should execute first)

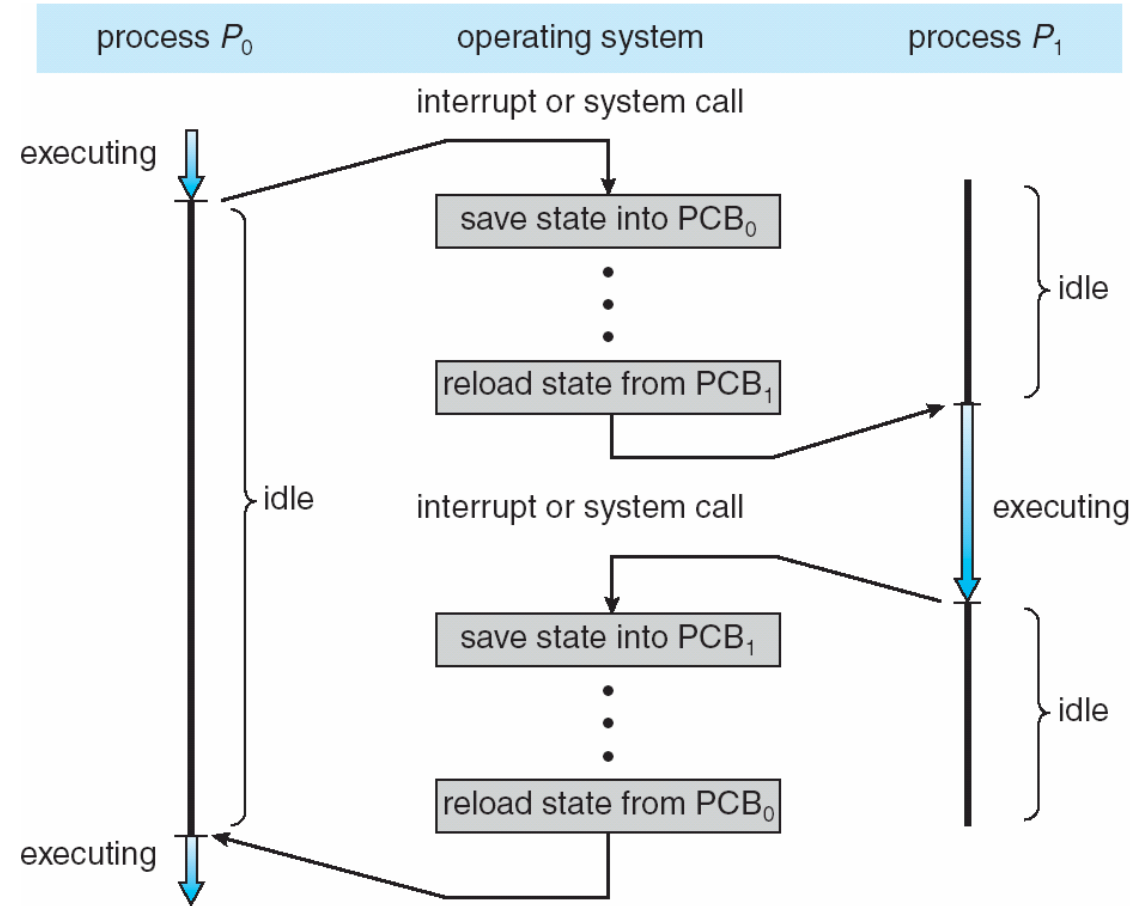
Memory-management information – memory allocated to the process

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



CPU Switch From Process to Process



What is a Thread?

A thread is a path of execution within a process. A process can contain multiple threads.

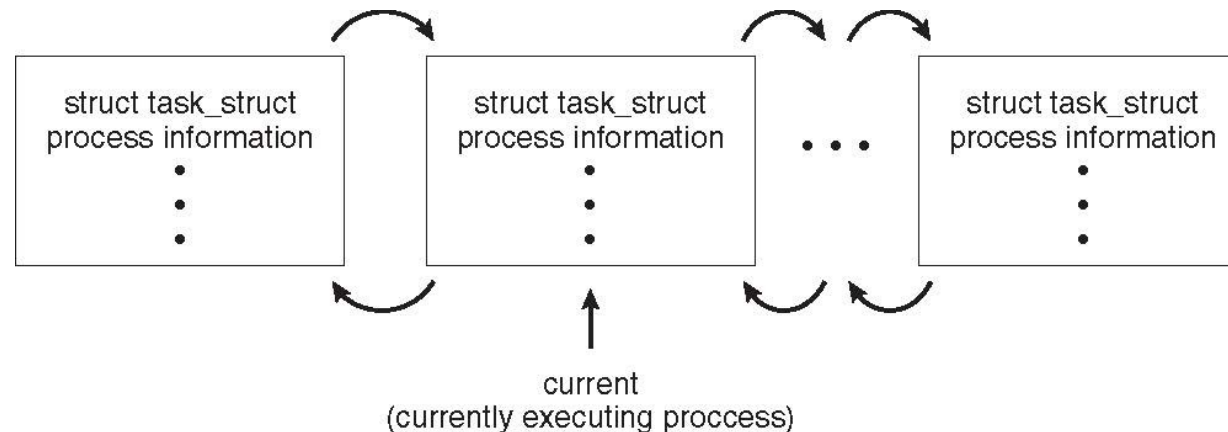
For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time.

The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.

The process control block in the Linux operating system is represented by the **C structure task struct**, which is found in the include file in the kernel source-code directory.

This structure contains all the necessary information for representing a process, including the **state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent.**

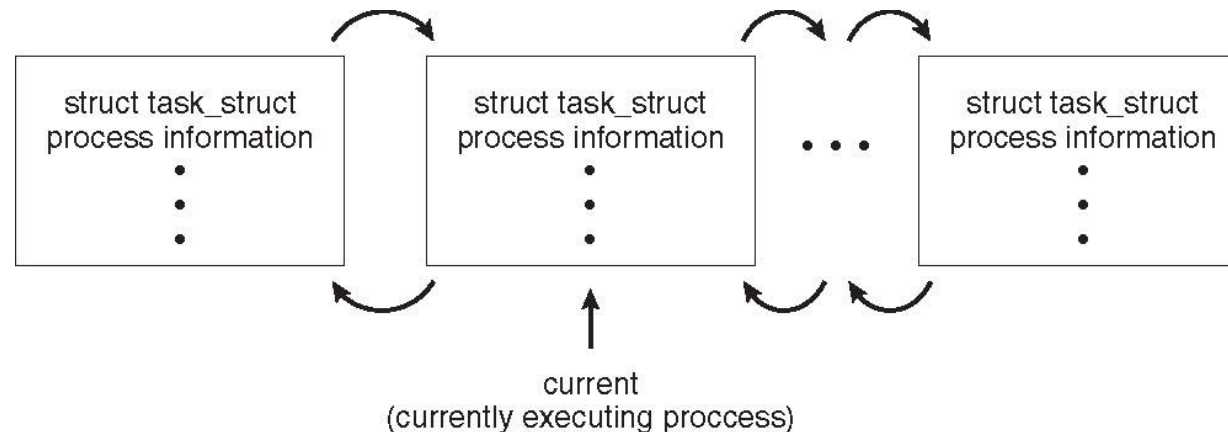


The kernel maintains a pointer— **current**—to the process currently executing on the system.

If **current** is a pointer to the process currently executing, its state is changed with the following:
current->state = new state

The process control block in the Linux operating system is represented by the **C structure task struct**, which is found in the include file in the kernel source-code directory.

This structure contains all the necessary information for representing a process, including the **state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent.**

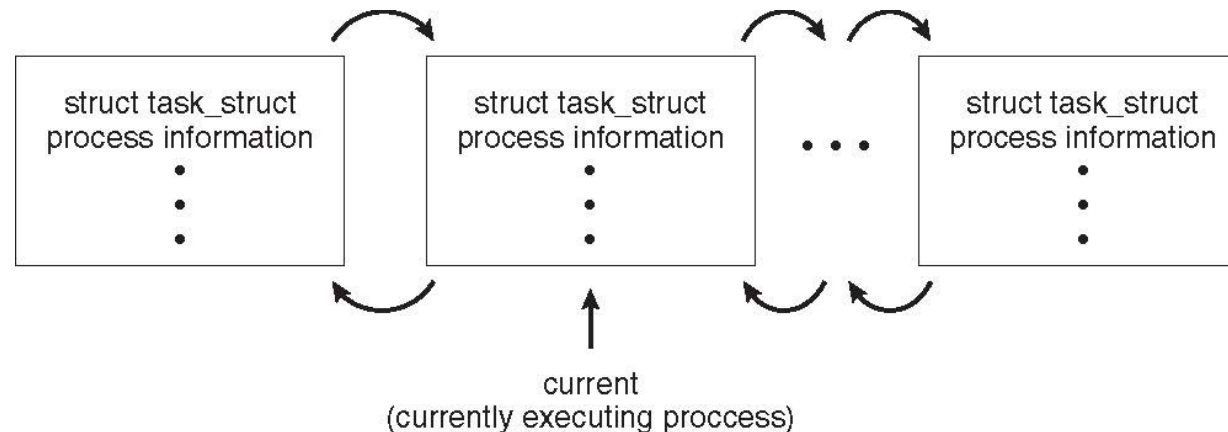


The kernel maintains a pointer— **current**—to the process currently executing on the system.

If **current** is a pointer to the process currently executing, its state is changed with the following:
current->state = new state

The process control block in the Linux operating system is represented by the **C structure task struct**, which is found in the include file in the kernel source-code directory.

This structure contains all the necessary information for representing a process, including the **state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent.**



The kernel maintains a pointer— **current**—to the process currently executing on the system.

If **current** is a pointer to the process currently executing, its state is changed with the following:
current->state = new state

- The objective of **multiprogramming** is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program.
- To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will **never be** more than one running process. If there are more processes, the rest will have **to wait until the CPU** is free and can be rescheduled.
- **Process scheduler** selects among available processes for next execution on CPU

Fork () – System Call is used to create the **separate duplicate process**, the process that is created **is called Child Process** and from which it has been created is called **Parent Process**

After a new child process is created, both processes will execute the next instruction following the fork() system call.

A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains process ID of newly created child process

Each of the Child Process Created will be having the different process ID (PID)

fork() creates a new process by duplicating the calling process, **The new process, referred to as child**, is an exact duplicate of the calling process, referred to as **parent**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // fork();

    print("Hello world!\n");
    return 0;
}
```

Total Number of Processes = 2^n

Output

Without fork()

Hello world

With fork()

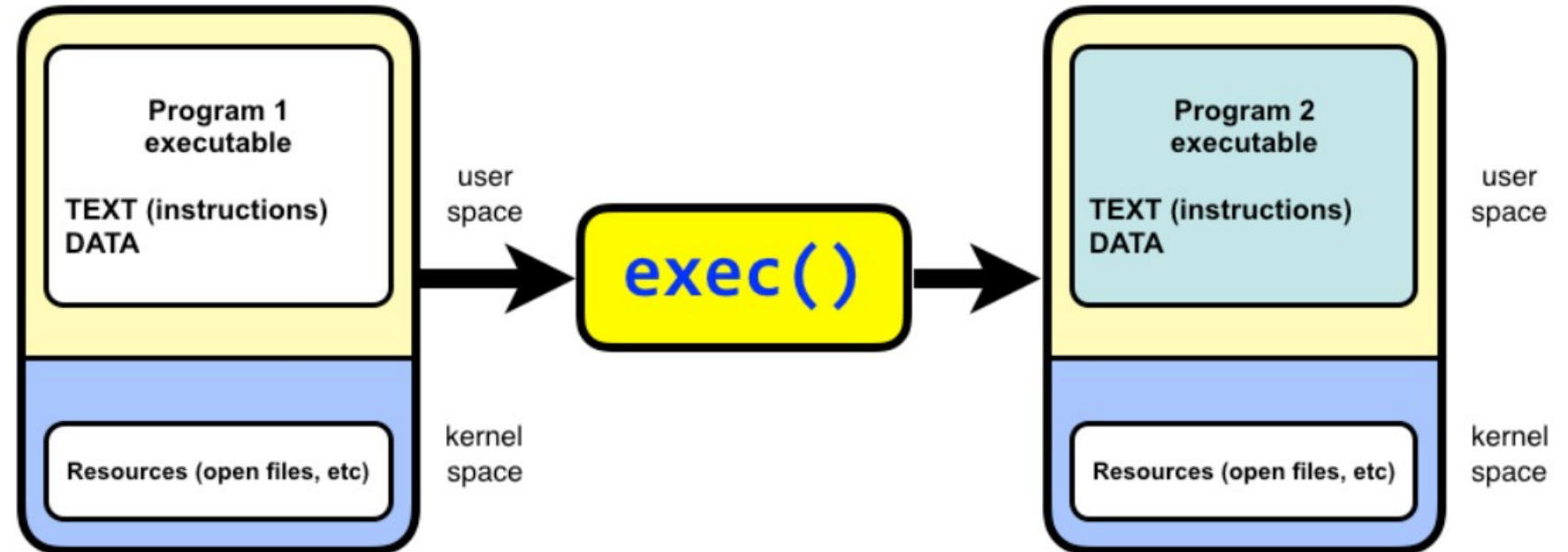
Hello world

Hello world

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.

Process id PID is not changed,



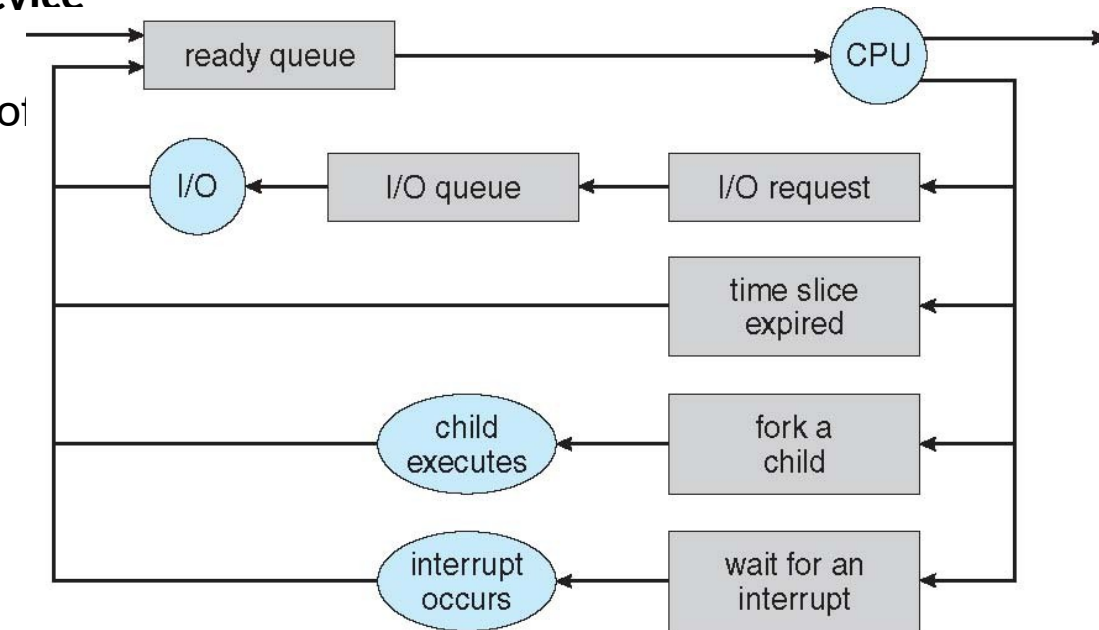
Job queue – List of all the process that are present in the system. The Job queue resides in Secondary Memory

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

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

Time Slice Expired : Process will execute for certain period of Time and go backs to ready queue

Processes migrate among the various queues



- A process migrates among the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion
- The primary distinction between these two schedulers lies **in frequency of execution**

Long-term scheduler (or **Job scheduler**) – selects which processes should be brought into the ready queue from Job queue

Long-term scheduler is invoked infrequently (seconds, minutes) \Rightarrow (may be slow)

The long-term scheduler controls the **degree of multiprogramming** (the average rate of process creation must be equal to the average departure rate of processes leaving the system)

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) \Rightarrow (must be fast)

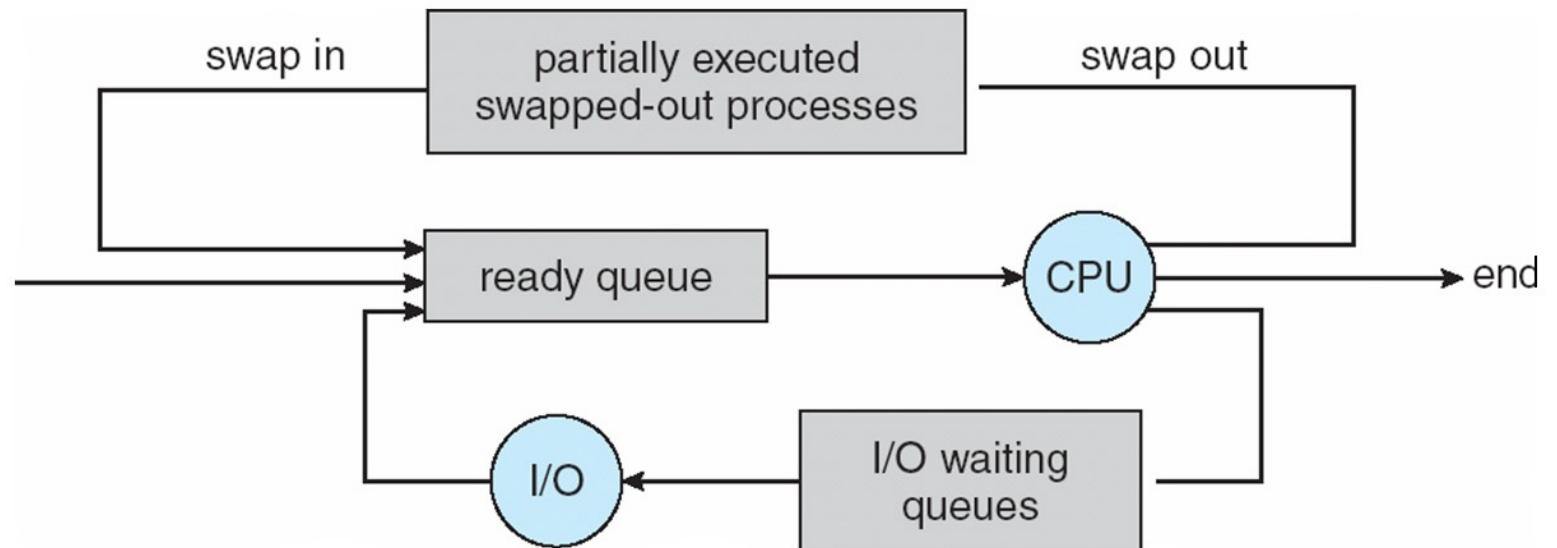
Processes can be described as either:

I/O-bound process – spends more time doing I/O than computations

CPU-bound process – spends more time doing computations

Long-term scheduler strives for good **process mix**

- **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



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**

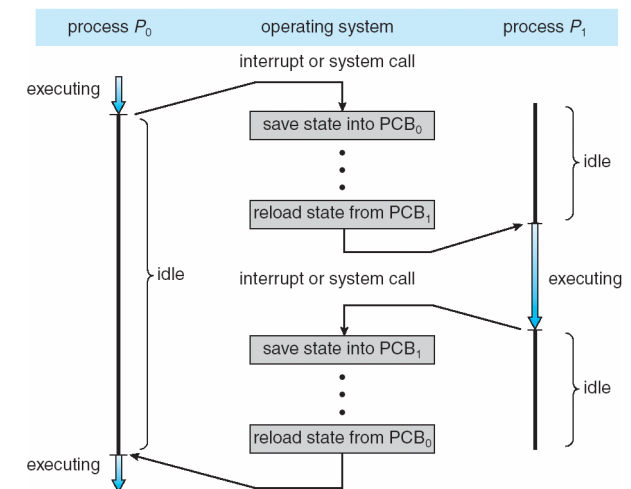
Context of a process represented in the PCB

Context-switch time is overhead; the system does no useful work while switching

- Context-switch time is pure overhead, because the system does no useful work while switching.

Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions

- Time dependent on hardware support



Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended

Due to screen real estate, user interface limits iOS provides for a

- Single **foreground** process- controlled via user interface (appearing on the display)

- Multiple **background** processes- in memory, running, but not on the display, and with limits

- Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Android runs foreground and background, with fewer limits

- Background process uses a **service** to perform tasks

- Service can keep running even if background process is suspended

- Service has no user interface, small memory use

Operations on Processes (Process Creation)

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

Generally, process identified and managed via a **process identifier (pid)**

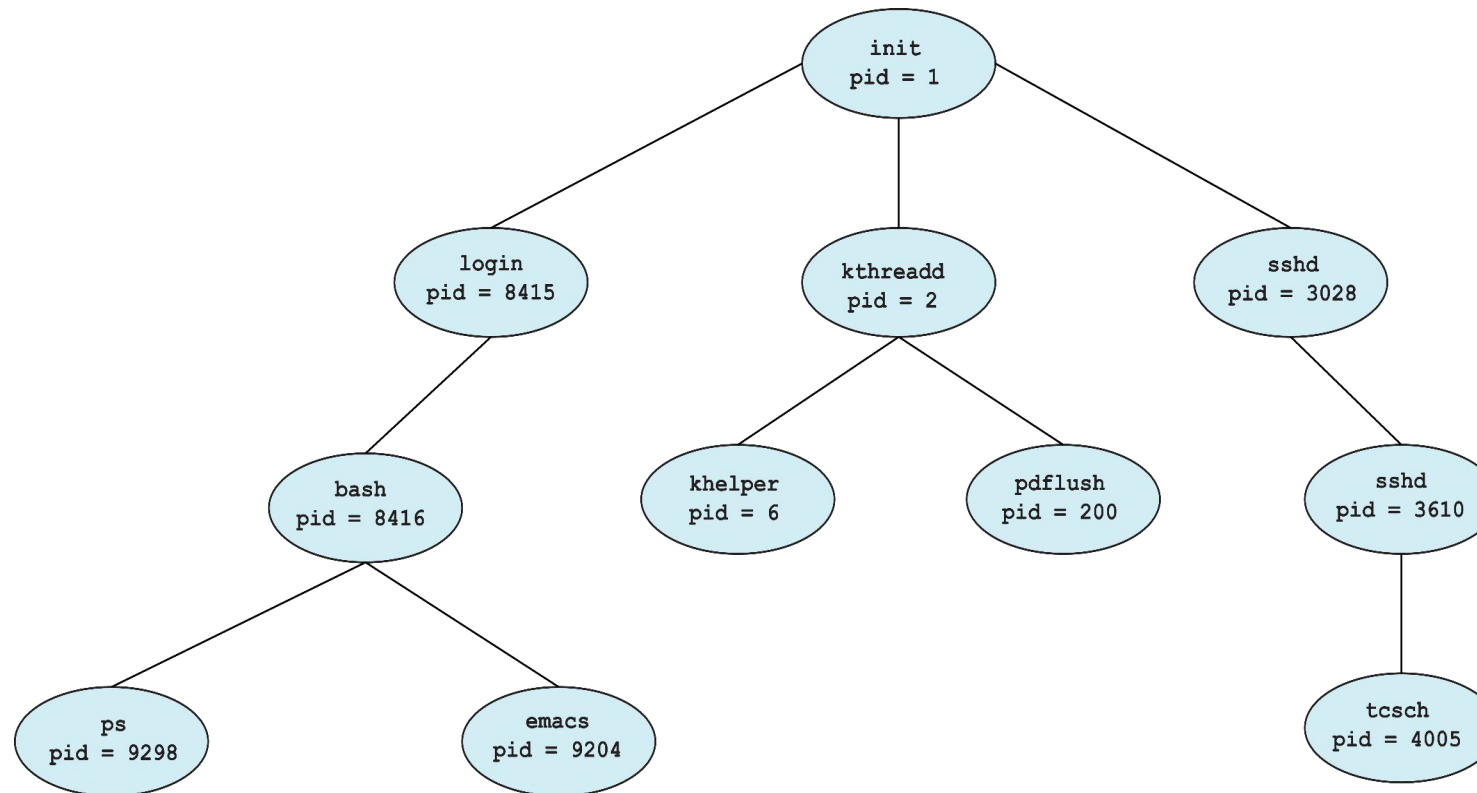
Resource sharing options

1. Parent and children share all resources
- 2 Children share subset of parent's resources
3. Parent and child may not share the resources, child may directly take the resources from operating system

Execution options

1. Parent and children **execute concurrently**
2. Parent waits until children terminate

Operations on Processes (Process Creation)



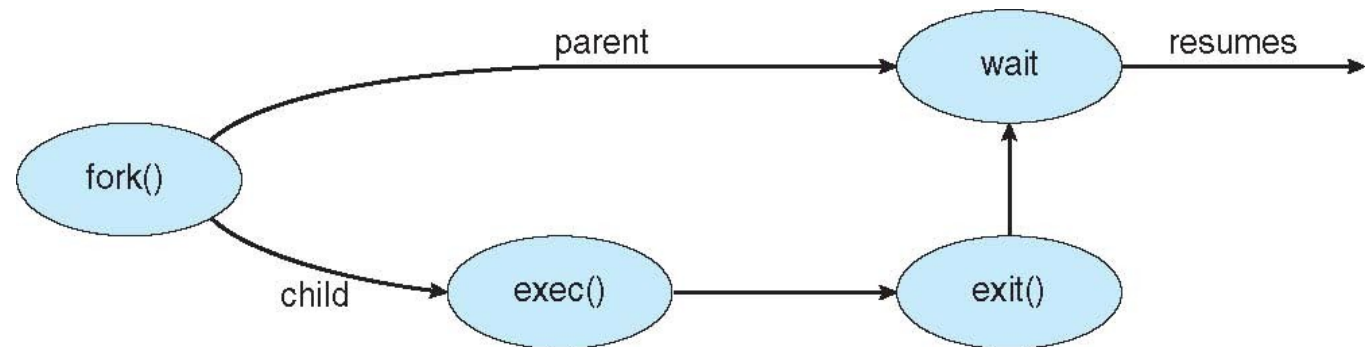
Address space

1. Child duplicate of parent (Basically Child contains all the instructions as parent)
2. Child has a program loaded into it (New Process can be loaded into the child)

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 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

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

process may return a status value (typically an integer) to its parent process (via the wait() system call)

```
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**

Introduction What Operating System do, Operating System structure, Operating system Operations.

System Structures Operating system services, System Calls, Types of System calls

Process Management Process concept, Process scheduling, Operations on processes