OperatingSystems

Introduction

The term operating systems was coined from the “operators” who used to operate switches in the early computer age.

What is an Operating System?

- An Intermediary between hardware and user.
- A program that manages computer hardware.
- The one program running at all times on the computer(kernel).

OS is of Several types:

- mainframe operating systems
- personal computer (PC) operating systems
- operating systems for mobiles platforms

What are the characteristics of these platforms which the OS needs to optimize for?

- Different mode of Inputs,limited resources
- Background processes, Battery(Bigger concern), memory
- Orientation of screen
- Completely different hardware management

★ Over the past 5-6 years all the android mobile markets tried to optimize power in their very new versions.

User Expectations:

- Convenience, easy of use, good performance
- Don’t bother about resource utilization

OS in system:

- OS is a **resource allocator**: Avoids conflict of requests by managing all the resources
- OS is a **control program**: Controls execution of user programs to prevent errors and improper use of the computer.

We have been talking about the resources many times, what are resources actually ?

- Memory & storage
- CPU(processing)
- I/O devices and ports

What does an OS do?

- Process management (Scheduling)
- Memory management (RAM management)
- Storage and I/O management

★ The OS itself is of few gb, of course the entire OS cannot be on the memory at same time, only parts used at some time are being loaded and unloaded.

What is the segmentation fault?

- Every process has a memory segment attached to it, if a process wants to access(R/W) the memory in another process segment we get a segmentation fault.

★ A process is a live entity; a program under execution.
★ OS itself is in the harddisk but when it has to run it has to be in the memory.
★ CPU cannot operate on the harddisk; it can operate only on the memory.
★ System Call is something the OS needs to handlen to create a new process. System calls are software Interrupts.

Computer System Organization

The most important function of an OS is to establish a communication between hardware and software of the device.

Device Controller:

- It is a hardware component that works as a bridge between the device hardware and Operating System(or an application program).

The CPU, device controller, graphics adapter connect through a common bus providing access to shared memory.

It is the job of the OS to manage “who gets what allocation in the memory ” since the CPUs and devices compete for the memory cycles.

What is an Interrupt?

- It is the signal that any hardware or software sends to the CPU to pay attention to it.
- interrupt transfers control to the interrupt service routine through the interrupt vector.
- CPU reads the content in the keyboard buffer through the interrupt vector and puts it in the memory, where it is supposed to be kept.
- It is an important aspect of modern OS design, and hence modern OS design is Interrupt driven.
- Software Interrupts also known as trap/exception are caused either by an error or a user request.

Device Driver:

If at all one of the hardware fails to establish a contact with the OS device driver comes to the rescue and establishes a contact between them.

IVT importance:

While a process is going on, if an interrupt arrives from the i/o controller then the CPU must halt the ongoing process by dumping it into some temporary memory space and load the routine to process the interrupt task and when done again it needs to load back the leftover task which is costly in terms of efficiency.

IVT (Interrupt Vector Table) contains the addresses of all the service routines which makes the routine to process the tasks faster.

Storage Structure:

Main memory –

- only large storage media that the CPU can access directly
- instruction execution
- random access
- Volatile

Secondary storage –

- Cheaper, slower, non-volatile.

★ All the registers,cache, memory are volatile but the secondary storage is non-volatile.SSDs run pretty much on memory but they have a battery backup and hence non-volatile.

- **MultiProgramming**

Multiprogramming is needed for efficiency. Single process cannot keep CPU and I/O devices busy at all times, hence multiple processes are needed to run simultaneously. When it has to wait for I/O, the OS switches to another job. There will be no interactivity in multiprogramming

- **Multitasking (Timesharing)**

CPU switches jobs so frequently that users can interact with each job while it is running
Interactive computing: User interaction via input devices here Response time should be < 1 second.

★ If processes don't fit in memory, swapping moves them in and out to run. Virtual memory allows execution of processes larger than physical memory

Operating System Operations

- Boot time: hardware starts in kernel mode
- After loading OS, user applications are started in user mode
- When trap / interrupt occurs, hardware switches from user mode to kernel mode

Computing Environments

- Single-Processor Systems
- Multiprocessors
- Multicore Systems

Multiprocessors:

Multiprocessors are also known as parallel systems. Two or more processors in close communication, sharing the computer bus and sometimes the clock, memory and peripheral devices

These are of Two types:

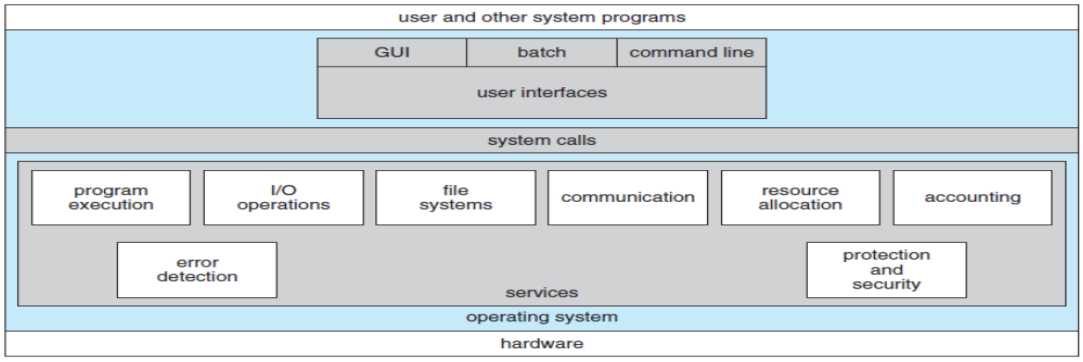
- Asymmetric Multiprocessing – each processor is assigned a specific task, the boss processor controls worker processors.
- Symmetric Multiprocessing – each processor performs all tasks, peers

Multicore Systems :

Involve Symmetric Multiprocessing, include multiple computing cores on a single chip more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication

OS Services

- User Interface
    - CLI or GUI or BPI with different input modalities.
- Program Execution
    - load the program into the memory, run & execution of the program.
- I/O operations
    - Running the program requires I/O which may involve a file or an I/O device.
- File Manipulation
    - Reading, writing files & directories, creating ,searching & deleting them.
- Communications
    - Exchange of information between the processes within a system or a network of systems.
- Error Detection
    - errors may occur in CPU or memory hardware or I/O devices or in user- program; OS should take the appropriate action to ensure correct and consistent computing & Debugging.
- Resource allocation
    - allocating resources for multiple concurrently executing processes
- Accounting
    - keeping track of which users use how much and what kinds of computer resources
- Protection and security
    - Keeping track of “Who owns what information”, “ who has access to what” and protecting accordingly.



CLI:

- Primarily fetches a command from user and executes it

GUI:

- User-friendly interface that involves mostly mouse, keyboard, and monitor.
- Icons represent files, programs, actions, Various mouse buttons over objects in the interface cause various actions (open directory, options, provide info)

Many systems now include both CLI and GUI interfaces

- Microsoft Windows is GUI with CLI “command” shell
- Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME).

Touchscreen Interface:

- Touchscreen devices require new interfaces where mouse is not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry & has Voice commands (doesn’t need GUI)

★ BASH - Bourne Again Shell

★ Before user Interfaces programs are loaded in punched cards or ted drivers or using an array of switches.

★ Kernel is the Core of the Operating System.

★ Batch Processing Interface is an user Interface, where immediate responses are not obtained.

System calls:

- System calls are the interface to the services provided by the OS. These are typically written in high level language(C / C++).
- These are accessed by programs via APIs (Application Programmers Interface).
- Three most common APIs are :
    - Win32 API for Windows
    - POSIX API for POSIX-based systems (including all versions of UNIX, Linux, and Mac OS X)
    - Java API for the Java virtual machine (JVM)

- A number is associated with each system call, System-call interface maintains a table indexed according to these numbers
- The caller need not know anything about how the system call is implemented he Just needs to obey the API and understand what the OS will do as a result of call execution. Most details of the OS interface are hidden from programmers by API.

OS Structures

- Simple Structure/ Monolithic Kernel
- Layered Approach
- Microkernels
- Modules
- Hybrid System

Layered Approach:

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

Micro Kernel:

- user services and kernel services are in separate address spaces
- smaller & slower but more secure and reliable
- extendible, all new services are added to user space
- if a service crashes, working of microkernel is not affected
- Drawback ??? Performance overhead of user space to kernel space communication

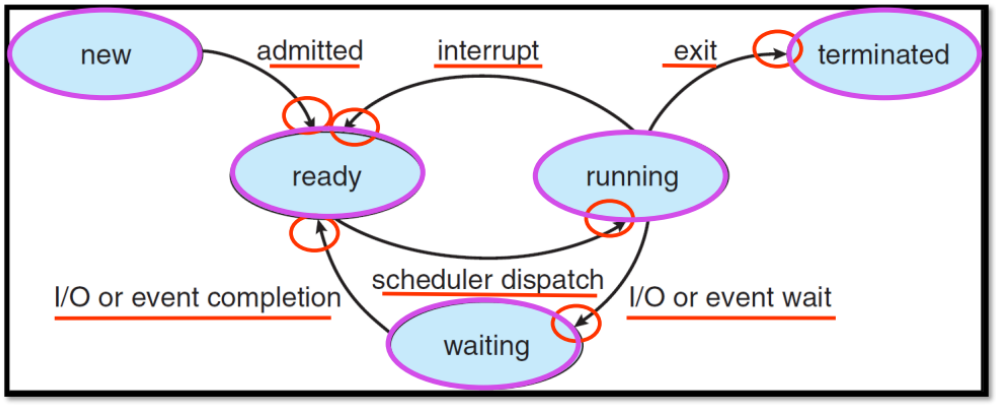
★ With one Processor the OS can execute only one process at a time.

Process Concept

- Process is a program in execution; process execution must progress in sequential fashion. Program becomes a process when an executable file is loaded into memory.
- Process is a live entity whereas the program is a passive entity stored on disk.
- Terms job and process are interchangeable.
- Process itself has multiple parts such as Stack, heap, program counter, text and data section.

States of Process

- **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. (can be scheduled)
- **terminated**: The process has finished execution.



Process scheduling is done by one of the scheduling policies(Schemes)

- **FCFS** : First Come First serve
- **SRTF** : Shortest Remaining Time First
- **SJF** : Shortest Job First
- **CFS**: Completely Fair Scheduler (Linux System)

Design a scheduling system which will maximize the utilization of cpu?
Design a CFS such that all processes have minimum waiting time?

- A process can be scheduled only when it is in the ready state but not in the waiting state because it is waiting for some event to happen.

- When an error occurs the process is forced to terminate; but when an interrupt arises at the running state it asks the cpu to halt the ongoing process and pay attention towards it and hence moves to the ready state.

- Each process will have a process control block. The OS will not have any history of the process that is terminated and it is impossible to get back the terminated process. If we wanna reuse some processes we need to have them in the secondary storage.

Process Creation

1. Parent processes create children processes, which, in turn create other processes, forming a tree of processes.
2. Generally, processes are identified and managed via a process identifier (pid), integer number allocated by the OS.
3. Resource sharing options (CPU time, memory, files, I/O devices) :
    a. Parent and children share all resources
    b. Children share subset of parent's resources
    c. Parent and child share no resources
4. Execution options :
    a. Parent and children execute concurrently
    b. Parent waits until children terminate
5. Address Space :
    a. In Linux, parent and child will have the identical copies of the memory having completely separate existence i.e different memory locations are created with same contents.
    b. In windows, a child has a parent loaded into it.

★ Any process on the linux will clone itself and then execute.
★ There will be no relation between child and parent regarding the states of Process as both have separate existence.
★ The parent and child processes are interlinked but they don't interfere with one another.

fork() - command used to create a child process

- Whenever a parent process is killed before the child process, the child process is re-parented to the main parent process (systemd).

Process Termination

- When a process finishes the last statement the process resources are deallocated by the OS. The process gets deleted by the system call **exit().**

- Sometimes Parent may terminate the child process because:
  - The child may have exceeded the resources limit.
  - Task assigned to a child is not required.
  - Parent is exiting (Some OS doesn't allow a child to continue if its parent dies).
- cascading termination :
  - If a process terminates then all its children, grand children must be terminated and this termination is initiated by OS

- The parent may wait for the child class to terminate using **wait()** system call so that it comes to know which of its child process are dead as the call returns status information and the pid of the terminated process

Zombie and Orphan Process

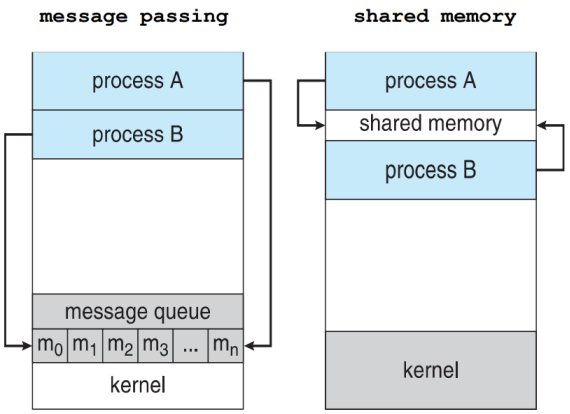
If there is no parent waiting (did not invoke wait() till then) for a process then the process is a **zombie process.** (Parent process has not yielded the child process).

For a process if the parent process is terminated, then the process is an **orphan process.**

Check slides

Interprocess Communication

- Processes within a system are independent or co-operative.
- Independent processes are the ones which do not communicate with each other & cooperating processes are the ones which communicate with each other.
- There are two ways of IPC :
    - Shared Memory
    - Message Passing



Shared Memory:

- In General the memory is private to the respective processes, In shared memory communication, an area of memory is explicitly assigned in common to both the processes that wish to communicate.
- The communication is under the control of the users processes not the operating system

Message passing

- Here processes communicate with each other without sharing address space or variables. The message size is either fixed or variable
    - This can be done using **pipes or message queues**.
- There exists a message queue in the kernel where one process writes into it and another process reads from it

- If processes P and Q wish to communicate, they need to establish a communication link between them exchange messages via send() / receive()
    - send (P, message) – send a message to process P
    - receive(Q, message) – receive a message from process Q

Direct Communication:

- Links are established automatically, the processes just need to know each other's identity
- A link is associated with exactly one pair of communicating processes; similarly there can be a single link between a pair of processes.

Indirect Communication:

- Messages are received from or directed into mailboxes(ports) , links get established only if the processes share a common mailbox.
- Many processes share the same link and there can be many links between a pair of processes.

Synchronization

Let there be two processes P1 and P2, P1 has write instruction and P2 has read instruction. So first P1 needs to be executed first and then P2 to have a proper communication between the processes.

Synchronization makes sure that P1 is executed before P2.

Message passing may be blocking or non-blocking

- Blocking send
    - The sender is blocked until the message passed is received by the receiving process
- Blocking Receive
    - The receiving process is blocked until the message is available from the sending process

- Non-Blocking send

    - The sender sends the message and continues to execute
- Non-Blocking Receive
    - The receiving process continues to execute even if it didn't receive the message, in other words it reads a null message as well.

Buffering

During the message passing /exchange the message needs to reside temporarily in some queue.

- Zero capacity
    - no messages are queued, link can't have any waiting messages, sender must block until receiver receives message
- Bounded capacity
    - queue is of finite length of n messages, sender need not block but has to wait if the queue is full.
- Unbounded capacity–
    - infinite length queue, sender never blocks

Pipe

A pipe has a read end and a write end; processes read(receive) at read end and write(send) at write end.

Let there be two processes P1 and P2, P1 has write instruction w1 and then a read instruction r1 and P2 has read instruction r2 and then a write instruction w2.

The ideal flow is **w1 -> r2 ->w2->r1**. If p1 is executed first and then p2 then flow becomes **w1->r1->r2->w2** which is of no use hence, Synchronization is very important in pipes

Half-duplex communication involves single pipe whereas full-duplex communication involves more than a pipe each of them has one way message passing

Ordinary Pipe:

It is an unidirectional pipe that allows one way communication. It can not be accessed outside the process that created it hence it cannot be used over a network.

It requires a parent-child relation between the communicating processes. Parent creates the pipe and creates a child process using fork() to establish a communication with it, the Child process inherits the pipe from the parent process like any other file/command.

The ordinary pipe ceases to exist if any one of the two processes gets terminated.

To have two way communication using ordinary pipe 2 pipes are required

Named Pipe

It is a bidirectional pipe and hence more powerful than the ordinary pipe. It can be used over a network of processes.

There is no need of parent child relationship between the communicating processes

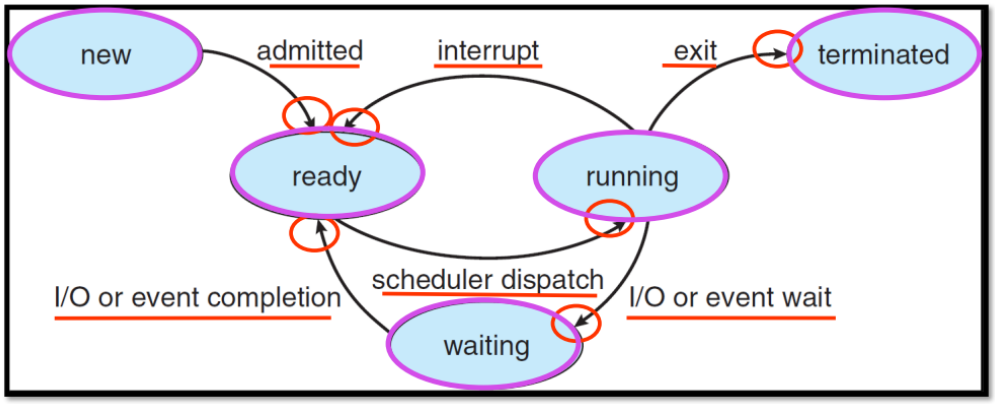
Wide through the concepts

fork()

: 0 - pid of child process

: >0 - pid of parent process

: <0 - error



New : Process created and is on hard disk

Ready : process moved into main memory

Running: The processor gets allocated and the processes get executed

Waiting/Blocked : while executing the process may wait for certain event to occur

Terminated: Once the given task is completed the processes terminates

Shared memory usage is difficult due to less protection of data, mandatory synchronization of processes, usage of pointers.

In unix every directory/IO devices/pipes are treated as a file

File descriptor(fd) : A number that uniquely identifies a file

For a pipe :

If fd = 0 -> read end
If fd = 1 -> write end

Pipe returns either 0 or -1; 0:success, -1: failure

read(fd,buffer,size) write(fd,buff, size)
We never know the order in which the processes get executed, unless or until system calls such as sleep(), wait() are used.

Hence, it's important for a process to close its read end before writing and close its write end before reading.

Since there is a parent-child relationship b/w processes using ordinary pipe they facilitate Intermachine communication only; because the parent-child processes run on the same processor.

Message Queue

msgget()
msgsnd()
msgrcv()
msgctl()

Used for short message (Reply/request/error provoking) passing. The sender and receiver work independently and do not wait/bother about each other.

Context switching

1. Save the context of old process
2. Select a new process among many processes
3. Load the new process

4. (After it's done) Retrieves the context of old process

Switching of cpu from one process to another is done due to the following reasons

1. Time overdue
2. Priority execution
3. Interrupt blockage
4. During Spooning (parent process waits for the child to terminate)

Context of a process includes Program controller, stack pointer, registers, open files, IO devices, priority queues. It is stored in PCB(program control block)

Spooning is the method of creating a child process

Process scheduling:

Selects the next process to be executed on the cpu

- Job queue : set of all processes in the system(hard disk-secondary memory)
- Ready queue: set of all processes in the main memory
- Device queue: set of all processes waiting for an I/O device

During execution a process migrates from one queue to another and they are implemented in the form of linked lists. Each queue will have its own scheduler

- Long-term/Job scheduler
- Short-term scheduler
- Medium-term scheduler
- I/O scheduler

Long-term scheduler decides the degree of multiprogramming it is slower than the short-term scheduler as it is implemented on a hard disk

Short-term scheduler/ CPU scheduler is also known as dispatcher. It is fast, selects which process should be executed next and allocates CPU
It controls the ready queue (implemented when a process is brought into ready queue or if any interrupt occurs).

Mid-term scheduler:
Swapping of processes is done by this scheduler; it removes a process temporarily and stores it on a hard-disk.

- Swaps out the lower priority processes into secondary memory.
- page faulting processes are swapped out.
- processes consuming large amounts of memory are swapped out in order to free up main memory for other processes.

Threads:

- An alternative to a process
- Threads are used for good responsiveness & better performance to the user.

EX: Consider the myntra web application, every time a client accesses the myntra home page a child process gets created wrt to the user's search. When the no.of clients increases the no.of child processes to be created also increases which load up the server and leads to a server down condition.

Here instead of processes, if threads are used then the web application facilitates smooth usage and good responsiveness to its clients because threads are light headed compared to processes.

One single process can have multiple threads of execution, but one thread can belong to only one process. Threads share the code section, data section, other OS resources as well except stack and registers. Stack and register are unique to every thread.

STP (Single thread processing) vs MTP(Multi thread processing)

At STP, Let P0 forkes to process P1 and P2 Let address space for P0 be 1kb then it's obvious that P1 and P2 have 1kb address space. So a total of 3kb space is allocated here.

At MTP, let P0 creates two threads T1 and T2 and assume that 0.8 kb will be code section, data section and other resources and stack register constitute 0.2 kb then total address space consumed will be 1.4kb since threads share the 0.8kb with other threads as well as the process.

We observed that amt of address space required by the threads are very less hence threads are light headed compared to process

Advantages of threads:

We do not require any explicit technique (like IPC) for threads because threads are dependent on each other, they share resources as they are referring to the same Address space unlike processes.

Context switching in processes is time consuming compared to thread switching, as most of the resources are shared, threads easily switch from one to other.

Single threaded processes must run non the same processor regardless of how many processors are available but threads can run on multiple processors enhancing scalability.

SIMILARITIES	DIFFERENCES
Threads create children just like processes	Unlike threads, processes are independent of each other
Like processes, threads within a process run sequentially	Unlike processes, threads can access every address space in the given task
Like processes, only one thread can be active at a time on a single cpu	
Like processes, when one thread is blocked other threads run as usual.	

A multiprocessor system consists of different chips, each one have a processing element

In a multicore system on a single chip we find multiple processors

Multicore systems with multithreading improves concurrency, because communication overhead in multicore systems is less than in multiprocessor systems.

Now-a-days every computer system is a multicore system and the OS kernels are multi-threaded.

Parallelism: Used in a multicore or multiprocessor environment, here multiple tasks are executed on multiple cores or multiple processors

Concurrency: Used in a single core or single processor environment, here multiple tasks are executed in a time interlinked manner. It provides pseudo parallelism.

Concurrent execution on single-core system:
Here there is one processor and it is capable of executing one thread at a time hence threads execute in sequential manner : T1,T2,T3,T4,T1,T2….

Parallelism on a multi-core system:
Here there is more than one core hence T1 and T2 can execute parallely. Threads that are dependent with each other can't execute parallely.

C1: T1,T3,T1,T2….
C2: T2,T4,T3,T1…

In Data Parallelism data gets disturbed to multiple cores to execute the same operation on each subset.

Ex: Sum of 10 elements of the array. On a multicore system the data can be separated as 5,5 elements which run on two core's executing the summation operation.

In task parallelism tasks get disturbed to multiple cores in such a way that thread gets different tasks irrespective of data being operated on.

Generally, processes run in user mode & whenever it needs service from the Operating system it switches to kernel mode. We can have multiple threads in user mode, kernel mode.

User threads have no kernel support hence there will be no involvement of OS during thread switching, no interrupt is raised during thread switching, the kernel knows nothing about the user threads.

Threads implemented at kernel level are kernel threads. These are completely managed by the OS and the users have no control over them.

Invoking a function in the thread library at user level results in local function calls in user space but at kernel level results in a system call.

1:1 -

Whenever a new user thread is created it's corresponding kernel thread gets created.

Parallel Processing is allowed; if a thread is blocked it's corresponding kernel thread gets blocked and remaining threads work as usual

No.of threads are limited since implementation of kernel threads is expensive

M:1-

Only one kernel thread is created for all the user threads and hence all threads at kernel level are mapped to one kernel thread

It doesn't allow Parallel Processing, the entire process gets blocked if one thread gets blocked.

M:M-

No.of threads are not limited but no.of kernel threads available are machine specific.

Parallel Processing is allowed; if a thread is blocked it's corresponding kernel thread gets blocked and remaining threads work as usual

Two level Model:

Similar to the M:M model but here one user thread gets bound to one kernel thread for certain.

Fork and exec

If a process having multiple threads is forked what happens ?

1. Duplicates all the threads
2. Duplicates the calling thread

exec replaces the current process with all its threads with new process

If a fork is done followed by an exec then duplicating the calling thread is better compared to duplicating all the threads.

The compiler checks whether fork is followed by a exec or not and then selects the optimal version for fork execution

Signal Handling

To notify something to process the OS sends a signal. A Signal must be handled by the signal handler available at the process.

A process receives a signal when it makes a request to the OS or if any unwanted event occurred.

Signal may be received synchronously or asynchronously depends on the source it has sent

Synchronous signal	**ASynchronous signal**
Caused due to the executing process	Caused by an event external to a running process
Errors like : Illegal memory access Div by Zero	Errors like : Ctrl+C Time explication
Signal gets delivered to the process/thread involved in causing the signal	Running process gets the signal asynchronously

Every signal has a Default signal handler which is run by the kernel and is responsible for handling all the signals received.

It can terminate the process, may ignore the signal, dump the core(halting), may continue the process.

A user-defined signal handler overrides the default signal handler.

Method of delivering a signal depends on signal type:
Synchronous signals are delivered to the thread causing the signal whereas some asynchronous signals are sent to all the threads pertaining to the process in execution.

In Unix, most multithreaded processes allocate a thread to specify the type of signal it accepts.

Thread Cancellation: Forcing the thread to terminate before it actually terminates.

The thread that is supposed to be canceled is known as target thread.

Asynchronous cancellation:
The moment the process decides to cancel a thread, One of the threads cancels the target thread immediately without bothering about the consequences.

The OS can't reclaim the resources when asynchronous cancellation occurs.

Deferred Cancellation:
Target thread is periodically checked if it should be canceled or not

Sometimes the target thread might be in the middle of data updation , or it may have all the resources pertaining to the task etc.

With deferred cancellation the target thread can finish its job and then it gets canceled.

Invoking thread cancellation requests cancellation but actual cancellation depends on how target thread takes the cancellation request

If the thread is in disable state then cancellation remains pending until it gets enabled , cleanup handler needs to be invoked

Cpu scheduling:

Degree of multiprogramming : Number of process that are present in the main memory

Preemption: Removal of cpu control over a process

Preemption and cpu scheduling go hand in hand i.e preemption of one process results in scheduling of another process

Main aim of multiprogramming is to have some process running on the processor so that cpu utilization is maximized

Multiprogramming and time slicing together provide best response to the user

Generally cpu burst and I/O burst occur alternately, if a process has more cpu bursts then it is cpu bound and if a process has more I/O bursts then its I/O bound.

System with best performance has good constituents of I/O bound processes and CPU bound Processes.