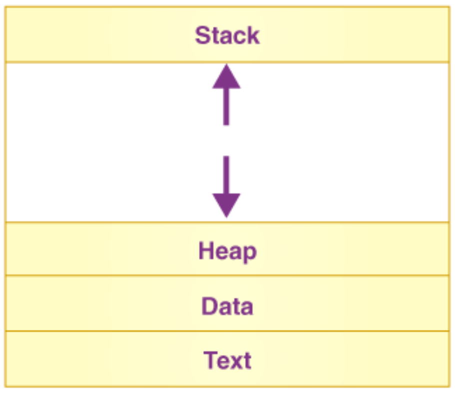
Operating System Notes

Process:

A process is a running program that serves as the foundation for all computation. In contrast to the program, which is often regarded as some ‘passive’ entity, a process is an ‘active’ entity.

A process is essentially running software. The execution of any process must occur in a specific order. A process refers to an entity that helps in representing the fundamental unit of work that must be implemented in any system.

A program can be segregated into four pieces when put into memory to become a process: stack, heap, text, and data. The diagram below depicts a simplified representation of a process in the main memory.



S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.

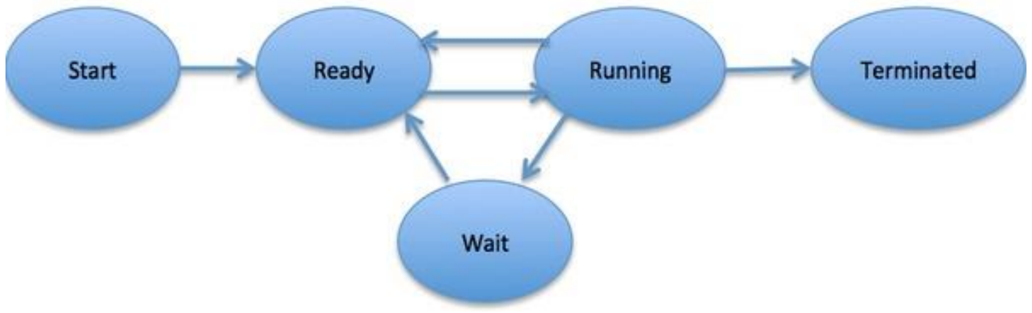
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

Process Life Cycle

When a process executes, it passes through different states. In general, a process can have one of the following five states at a time.

S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.

3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



Process Control Block (PCB)

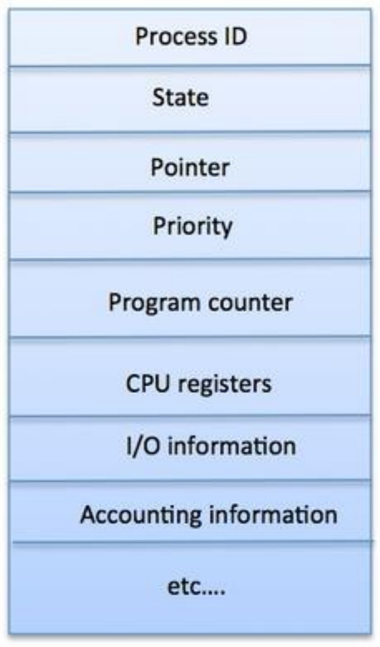
A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Here is a simplified diagram of a PCB −



A scheduler is a program that uses a scheduling algorithm to make choices. The following are characteristics of a good scheduling algorithm:

- For users, response time should be kept to a bare minimum.

- The total number of jobs processed every hour should be as high as possible, implying that a good scheduling system should provide the highest possible throughput.
- The CPU should be used to its full potential.
- Each process should be given an equal amount of CPU time.

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.

For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time.The number of processes currently in memory is known as the **degree of multiprogramming**.

In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

swapping, whose key idea is that sometimes it can be advantageous to

remove a process from memory (and from active contention for the CPU)

and thus reduce the degree of multiprogramming. Later, the process can be

reintroduced into memory, and its execution can be continued where it left off.

Context Switch

When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

Context includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch.**

When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Contextswitch time is pure overhead, because the system does no useful work while switching.

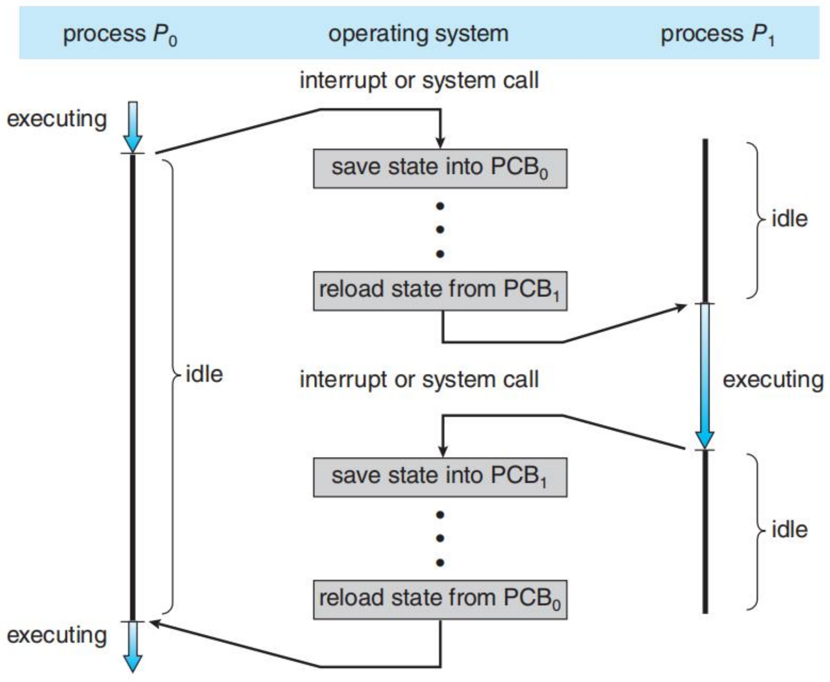


Figure 3.6 Diagram showing context switch from process to process.

The systemd process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots.

In general, when a process creates a child process, that child process will

need certain resources (CPU time, memory, files, I/O devices) to accomplish

its task. A child process may be able to obtain its resources directly from

the operating system, or it may be constrained to a subset of the resources

of the parent process. The parent may have to partition its resources among

its children, or it may be able to share some resources (such as memory or

files) among several of its children. Restricting a child process to a subset of

the parent’s resources prevents any process from overloading the system by

creating too many child processes

The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call). The parent waits for the child process to complete with the wait() system call.

Processes are created in the Windows API using the CreateProcess() function, which is similar to fork() in that a parent creates a new child process. However, whereas fork() has the child process inheriting the address space of its parent, CreateProcess() requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas fork() is passed no parameters, CreateProcess() expects no fewer than ten parameters.

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the wait() system call). All the resources of the process —including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

If a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;

int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the init process as the new parent to orphan processes. (Recall from Section 3.3.1 that init serves as the root of the process hierarchy in UNIX systems.) The init process periodically

invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Android Process Hierarchy

- **Foreground process**—The current process visible on the screen, representing the application the user is currently interacting with.

- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)

- **Service process**—A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)

- **Background process**—A process that may be performing an activity but is not apparent to the user.

- **Empty process**—A process that holds no active components associated with any application

Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is ***independent*** if it does not share data with any other processes executing in the system. A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Cooperating processes require an **interprocess communication** (**IPC**) mechanism that will allow them to exchange data— that is, send data to

and receive data from each other. There are two fundamental models of

interprocess communication: **shared memory** and **message passing**. In the

shared-memory model, a region of memory that is shared by the cooperating

processes is established. Processes can then exchange information by reading

and writing data to the shared region. In the message-passing model,communication takes place by means of messages exchanged between the cooperating processes.

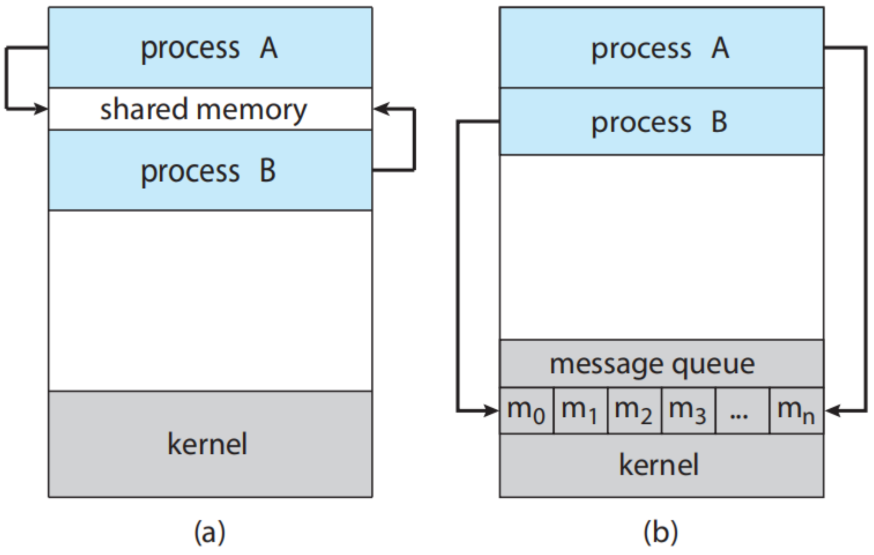


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are typically implemented using system

calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish sharedmemory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Interprocess communication using shared memory requires communicating

processes to establish a region of shared memory. Typically, a shared-memory

region resides in the address space of the process creating the shared-memory

segment. Other processes that wish to communicate using this shared-memory

segment must attach it to their address space. Recall that, normally, the oper

ating 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.

One solution to the producer–consumer problem uses shared memory. To

allow producer and consumer processes to run concurrently, we must have

available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by

the producer and consumer processes. A producer can produce one item while

the consumer is consuming another item. The producer and consumer must be

synchronized, so that the consumer does not try to consume an item that has

not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no prac

tical limit on the size of the buffer. The consumer may have to wait for new

items, but the producer can always produce new items. The **bounded buffer**

assumes a fixed buffer size. In this case, the consumer must wait if the buffer

is empty, and the producer must wait if the buffer is full.

IPC in Shared-Memory Systems

IPC in Message-Passing Systems

This passage is describing different synchronization mechanisms used in inter-process communication through the send() and receive() primitives. In the context of operating systems and concurrent programming, synchronization is crucial to ensure proper communication and coordination between processes.

1. **Blocking Send:**

   - The sending process is blocked until the message is received by the receiving process or stored in the mailbox.

- This ensures that the sender does not proceed further until it is certain that the message has been successfully sent and received by the intended recipient.

2. **Nonblocking Send:**

- The sending process sends the message and immediately resumes its operation without waiting for the receiver to acknowledge the message.

- This allows the sender to continue with its tasks without waiting for the message to be processed by the receiver.

3. **Blocking Receive:**

- The receiving process is blocked until a message is available in the mailbox.

- This ensures that the receiver doesn't proceed until there is a message to be processed.

4. **Nonblocking Receive:**

- The receiver attempts to retrieve a message, but if there is none available, it returns either a valid message or a null value.

- This allows the receiver to check for messages without getting blocked, enabling it to perform other tasks if no message is currently available.

In essence, these options provide flexibility in designing communication mechanisms between processes, allowing developers to choose the level of synchronization that best fits the requirements of their application. The terms "synchronous" and "asynchronous" refer to the timing and coordination of events between processes, with synchronous operations implying a blocking behavior and asynchronous operations indicating nonblocking behavior.

Buffering

Whether communication is direct or indirect, messages exchanged by commu

nicating processes reside in a temporary queue. Basically, such queues can be

implemented in three ways:

• **Zero capacity**. The queue has a maximum length of zero; thus, the link

cannot have any messages waiting in it. In this case, the sender must block

until the recipient receives the message.

• **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages

can reside in it. If the queue is not full when a new message is sent, the

message is placed in the queue (either the message is copied or a pointer

to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender

must block until space is available in the queue.

• **Unbounded capacity**. The queue's length is potentially infinite; thus, any

number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no

buffering. The other cases are referred to as systems with automatic buffering

Threads

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Types of Parallelism (new)

In theory there are two different ways to parallelize the workload:

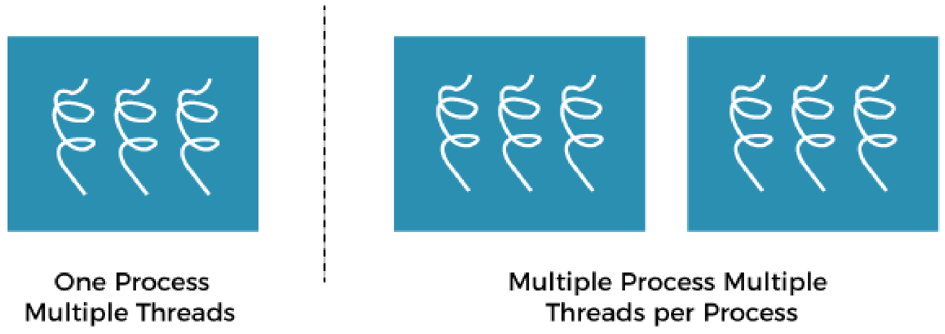
Data parallelism divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.

Task parallelism divides the different tasks to be performed among the different cores and performs them simultaneously.

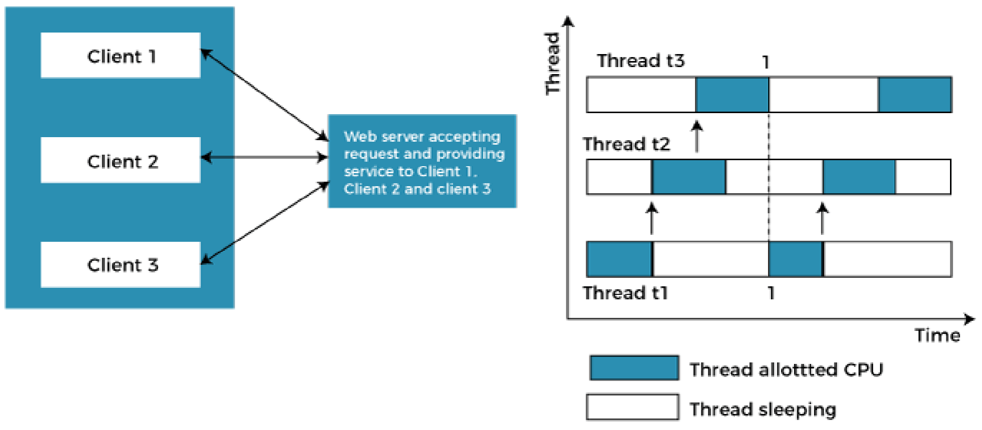
In practice no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination.

Multithreading Model:

Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. With the use of multithreading, multitasking can be achieved.



The main drawback of single threading systems is that only one task can be performed at a time, so to overcome the drawback of this single threading, there is multithreading that allows multiple tasks to be performed.For example:



In the above example, client1, client2, and client3 are accessing the web server without any waiting. In multithreading, several tasks can run at the same time.

In an operating system, threads are divided into the user-level thread and the Kernel-level thread. User-level threads handled independent form above the kernel and thereby managed without any kernel support. On the opposite hand, the operating system directly manages the kernel-level threads. Nevertheless, there must be a form of relationship between user-level and kernel-level threads.

There exists three established multithreading models classifying these relationships are:

Many to one multithreading model

One to one multithreading model

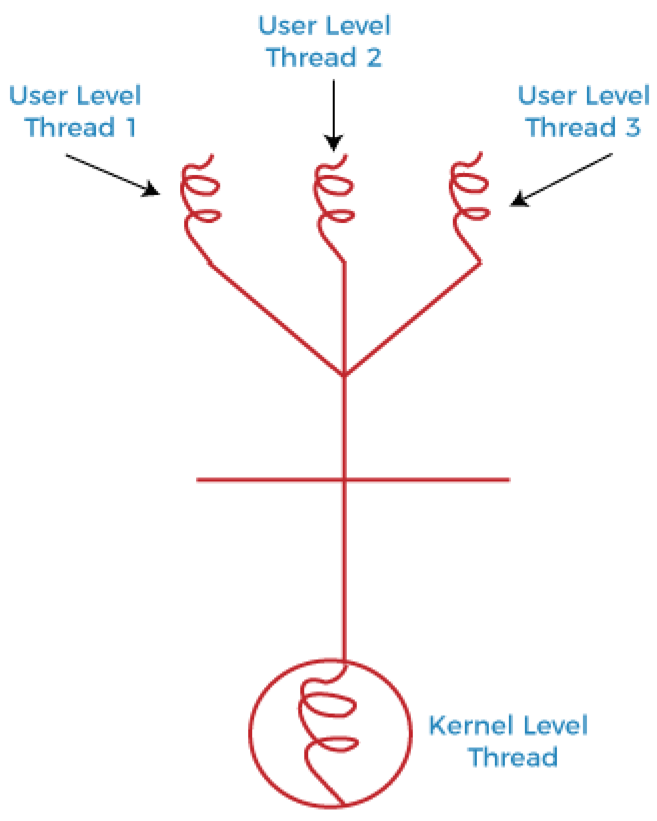
Many to Many multithreading models

Many to one multi-threading model:

The many to one model maps many user levels threads to one kernel thread. This type of relationship facilitates an effective context-switching environment, easily implemented even on the simple kernel with no thread support.

The disadvantage of this model is that since there is only one kernel-level thread schedule at any given time, this model cannot take advantage of the hardware acceleration offered by multi-threaded processes or multi-processor systems. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time.

The thread management is done on the user level so it is more efficient.

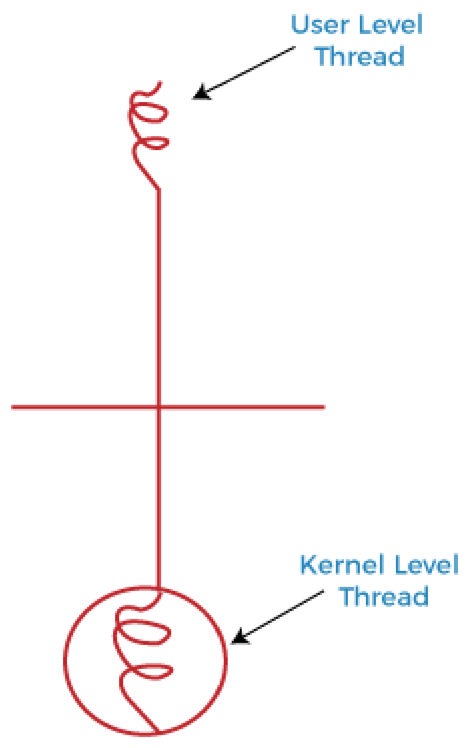


Many-to-one model

In the above figure, the many to one model associates all user-level threads to single kernel-level threads.

One to one multi-threading model:

The one-to-one model maps a single user-level thread to a single kernel-level thread. This type of relationship facilitates the running of multiple threads in parallel. As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won't be blocked.However, this benefit comes with its drawback. The generation of every new user thread must include creating a corresponding kernel thread causing an overhead, which can hinder the performance of the parent process. Windows series and Linux operating systems try to tackle this problem by limiting the growth of the thread count.

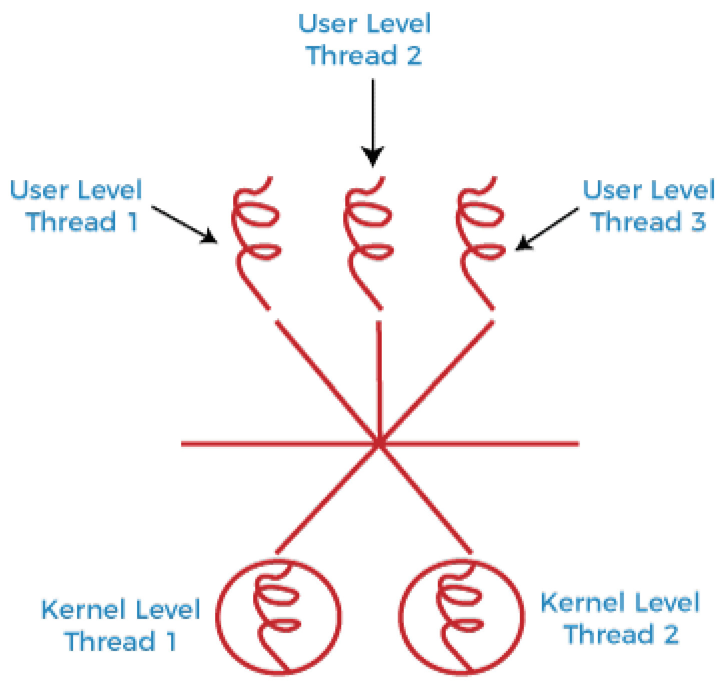


One-to-one model

In the above figure, one model associates that one user-level thread to a single kernel-level thread.

Many to Many multi-threading model:

In this type of model, there are several user-level threads and several kernel-level threads. The number of kernel threads created depends upon a particular application. The developer can create as many threads at both levels but may not be the same. The many to many model is a compromise between the other two models. In this model, if any thread makes a blocking system call, the kernel can schedule other user threads to another kernel thread for execution. Also, with the introduction of multiple threads, complexity is not present as in the previous models. Though this model allows the creation of multiple kernel threads, true concurrency cannot be achieved by this model. This is because the kernel can schedule only one process at a time.



Many-to-Many model

Many to many versions of the multithreading model associate several user-level threads to the same or much less variety of kernel-level threads in the above figure.

The benefits of multithreaded programming can be broken down into four

major categories:

1. Responsiveness. Multithreading an interactive application may allow a

program to continue running even if part of it is blocked or is perform

ing a lengthy operation, thereby increasing responsiveness to the user.

This quality is especially useful in designing user interfaces.

2.Resource sharing: Resources can be shared between all threads within a process, such as code, data, and files. Note: The stack and register cannot be shared between threads.The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. Economy. Allocating memory and resources for process creation is

costly. Because threads share the resources of the process to which they

belong, it is more economical to create and context-switch threads.

4. Scalability. The benefits of multithreading can be even greater in a mul

tiprocessor architecture, where threads may be running in parallel on

different processing cores.

Programming Challenges

For application programmers, there are five areas where multi-core chips present new challenges:

1. Identifying tasks - Examining applications to find activities that can be performed concurrently.
2. Balance - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
3. Data splitting - To prevent the threads from interfering with one another.
4. Data dependency - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
5. Testing and debugging - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

CPU utilization - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

6. Throughput - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
7. Turnaround time - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
8. Waiting time - How much time processes spend in the ready queue waiting their turn to get on the CPU.
9. (Load average - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)
10. Response time - The time taken in an interactive program from the issuance of a command to the commence of a response to that command.

Convoy Effect in FCFS

FCFS may suffer from the convoy effect if the burst time of the first job is the highest among all. As in the real life, if a convoy is passing through the road then the other persons may get blocked until it passes completely. This can be simulated in the Operating System also.

If the CPU gets the processes of the higher burst time at the front end of the ready queue then the processes of lower burst time may get blocked which means they may never get the CPU if the job in the execution has a very high burst time. This is called convoy effect or starvation.

CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer.Arrival Time: Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU execution.

Turn Around Time: Time Difference between completion time and arrival time.

Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time

Preemptive Scheduling

Preemptive scheduling is used when a process switches from the running state to the ready state or from the waiting state to the ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Non-Preemptive Scheduling

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.

1. First Come First Serve:

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

2. Shortest Job First(SJF):

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed.

Characteristics of SJF:

1. Shortest Job first has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.

2. It is associated with each task as a unit of time to complete.

3. It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

3. Priority Scheduling:

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works based on the priority of a process. In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first. In the case of any conflict, that is, where there is more than one process with equal value, then the most important CPU planning algorithm works on the basis of the FCFS (First Come First Serve) algorithm.

Characteristics of Priority Scheduling:

1. Schedules tasks based on priority.

2. When the higher priority work arrives and a task with less priority is executing, the higher priority proess will takes the place of the less priority proess and the later is suspended until the execution is complete.

Advantages of Priority Scheduling:

1. The average waiting time is less than FCFS

2. Less complex

Disadvantages of Priority Scheduling:

One of the most common demerits of the Preemptive priority CPU scheduling algorithm is the Starvation Problem. This is the problem in which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem.

4. Round robin:

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Characteristics of Round robin:

1. It’s simple, easy to use, and starvation-free as all processes get the balanced CPU allocation.

2. One of the most widely used methods in CPU scheduling as a core.

3. It is considered preemptive as the processes are given to the CPU for a very limited time.

Advantages of Round robin:

1. Round robin seems to be fair as every process gets an equal share of CPU.

2. The newly created process is added to the end of the ready queue.

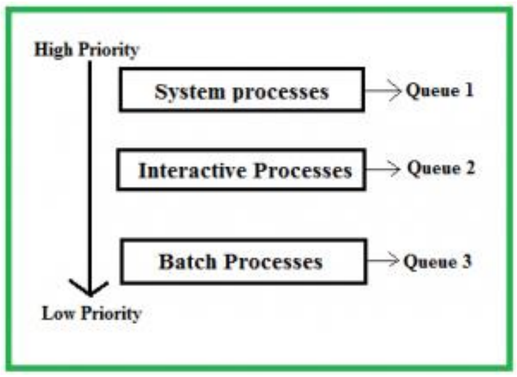
5.Multiple Queue Scheduling:

Processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a foreground (interactive) process and a background (batch) process. These two classes have different scheduling needs. For this kind of situation Multilevel Queue Scheduling is used.

Advantages of multilevel queue scheduling:

The main merit of the multilevel queue is that it has a low scheduling overhead.

Multilevel queue scheduling partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.



10. Multilevel Feedback Queue Scheduling:

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling is like Multilevel Queue Scheduling but in this process can move between the queues. And thus, much more efficient than multilevel queue scheduling.

Characteristics of Multilevel Feedback Queue Scheduling:

1. In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system, and processes are not allowed to move between queues.

2. As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead, But on the other hand disadvantage of being inflexible.

Advantages of Multilevel feedback queue scheduling:

1. It is more flexible

2. It allows different processes to move between different queues

Disadvantages of Multilevel feedback queue scheduling:

1. It also produces CPU overheads

2. It is the most complex algorithm.

Load balancing attempts to keep the workload evenly distributed across all processors in

an SMP system. It is important to note that load balancing is typically necessary

only on systems where each processor has its own private ready queue of eligi

ble threads to execute. On systems with a common run queue, load balancing

is unnecessary, because once a processor becomes idle, it immediately extracts

a runnable thread from the common ready queue.

There are two general approaches to load balancing: push migration and

pull migration. With **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) threads from overloaded to idle or less-busy

processors. **Pull migration** occurs when an idle processor pulls a waiting task

from a busy processor. Push and pull migration need not be mutually exclusive

and are, in fact, often implemented in parallel on load-balancing systems.

S.NO	Process	Thread
1.	Process means any program is in execution.	Thread means a segment of a process.
2.	The process takes more time to terminate.	The thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	The process is isolated.	Threads share memory.
8.	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
9.	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
10.	If one process is blocked then it will not affect the execution of other processes	If a user-level thread is blocked, then all other user-level threads are blocked.
11.	The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
12.	Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
13.	A system call is involved in it.	No system call is involved, it is created using APIs.
14.	The process does not share data with each other.	Threads share data with each other.

What is the Operating System Structure?

The operating system structure refers to the way in which the various components of an operating system are organized and interconnected. There are several different approaches to operating system structure, each with its own advantages and disadvantages.

An operating system has a complex structure, so we need a well-defined structure to assist us in applying it to our unique requirements. Just as we break down a big problem into smaller, easier-to-solve subproblems, designing an operating system in parts is a simpler approach to do it. And each section is an Operating System component. The approach of interconnecting and integrating multiple operating system components into the kernel can be described as an operating system structure

Hybrid kernel is a kernel architecture based on a combination of microkernel and monolithic kernel architecture used in computer operating systems. This kernel approach combines the speed and simpler design of monolithic kernel with the modularity and execution safety of microkernel.

A Hybrid kernel is a combination of both Monolithic and Microkernel architecture. It combines the advantages of both and tries to overcome the disadvantages of each. A Hybrid kernel has a larger kernel than a microkernel but smaller than a monolithic kernel.

Operating System:

• An operating system is a program which manages all the computer hardwares.

• It provides the base for application program and acts as an intermediary between a user and the computer hardware.

• The operating system has two objectives such as:

# Firstly, an operating system controls the computer's hardware.

# The second objective is to provide an interactive interface to the user and interpret

commands so that it can communicate with the hardware.

System Calls:

System calls provide the interface between a process & the OS. These are usually available in the form of assembly language instruction. Some systems allow system calls to be made directly from a high level language program like C, BCPL and PERL etc. systems calls occur in different ways depending on the computer in use. System calls can be roughly grouped into 5 major categories:

1. Process Control:

• End, abort: A running program needs to be able to has its execution either normally (end) or abnormally (abort).

• Load, execute:A process or job executing one program may want to load and executes another program.

• Create Process, terminate process: There is a system call specifying for the purpose of creating a new process or job (create process or submit job). We may want to terminate a job or process that we created (terminates process, if we find that it is incorrect or no longer needed).

• Get process attributes, set process attributes: If we create a new job or process we should able to control its execution. This control requires the ability to determine & reset the attributes of a job or processes (get process attributes, set process attributes).

• Wait time: After creating new jobs or processes, we may need to wait for them to finish their execution (wait time).

• Wait event, signal event: We may wait for a specific event to occur (wait event). The jobs or processes then signal when that event has occurred (signal event).

2. File Manipulation:

• Create file, delete file: We first need to be able to create & delete files. Both the system calls

require the name of the file & some of its attributes.

• Open file, close file: Once the file is created, we need to open it & use it. We close the file

when we are no longer using it.

• Read, write, reposition file: After opening, we may also read, write or reposition the file

(rewind or skip to the end of the file).

• Get file attributes, set file attributes: For either files or directories, we need to be able to

determine the values of various attributes & reset them if necessary. Two system calls get file

attribute & set file attributes are required for their purpose.

3. Device Management:

• Request device, release device: If there are multiple users of the system, we first request

the device. After we finished with the device, we must release it.

• Read, write, reposition: Once the device has been requested & allocated to us, we can read,

write & reposition the device.

4. Information maintenance:

• Get time or date, set time or date:Most systems have a system call to return the current

date & time or set the current date & time.

• Get system data, set system data: Other system calls may return information about the

system like number of current users, version number of OS, amount of free memory etc.

• Get process attributes, set process attributes: The OS keeps information about all its

processes & there are system calls to access this information.

5. Communication: There are two modes of communication such as:

• Message passing model: Information is exchanged through an inter process

communication facility provided by operating system. Each computer in a network has a

name by which it is known. Similarly, each process has a process name which is translated to

an equivalent identifier by which the OS can refer to it. The get hostid and get processed

systems calls to do this translation. These identifiers are then passed to the general purpose open & close calls provided by the file system or to specific open connection system call. The

recipient process must give its permission for communication to take place with an accept

connection call. The source of the communication known as client & receiver known as

server exchange messages by read message & write message system calls. The close

connection call terminates the connection.

• Shared memory model: processes use map memory system calls to access regions of

memory owned by other processes. They exchange information by reading & writing data in

the shared areas. The processes ensure that they are not writing to the same location

simultaneously.

Operating System Services

An operating system provides an environment for the execution of the program. It provides some

services to the programs. The various services provided by an operating system are as follows:

• Program Execution: The system must be able to load a program into memory and to run

that program. The program must be able to terminate this execution either normally or

abnormally.

• I/O Operation: A running program may require I/O. This I/O may involve a file or a I/O device for specific device. Some special function can be desired. Therefore the operating

system must provide a means to do I/O.

• File System Manipulation: The programs need to create and delete files by name and read

and write files. Therefore the operating system must maintain each and every files correctly.

• Communication: The communication is implemented via shared memory or by the

technique of message passing in which packets of information are moved between the

processes by the operating system.

• Error detection: The operating system should take the appropriate actions for the

occurrences of any type like arithmetic overflow, access to the illegal memory location and

too large user CPU time.

• Research Allocation: When multiple users are logged on to the system the resources must

be allocated to each of them. For current distribution of the resource among the various

processes the operating system uses the CPU scheduling run times which determine which

process will be allocated with the resource.

• Accounting: The operating system keep track of which users use how many and which kind

of computer resources.

• Protection: The operating system is responsible for both hardware as well as software protection. The operating system protects the information stored in a multiuser computer

system.