DU1 – Programming processes

Basic Concepts

Programs, processes, and threads

A **program** is the **code** or set of instructions that is stored on your computer to complete a certain task. For example, **chrome.exe** is an executable file containing the set of instructions written so that we can view web pages.

A computer program in its human-readable form is called **source code**. Usually, the source code needs to be translated to machine instructions using its language's **compiler**. The resulting file is called **executable**.

Alternatively, source code may execute within the language's **interpreter**. Java compiles into an intermediate form, which is then executed by a Java interpreter.

There are many types of **programs**, including programs built into the operation system and ones to complete specific tasks. Generally, task-specific programs are called **applications** (or apps).

Programs are not stored on the primary memory in your computer. They are **stored** on a disk or a **secondary memory** on your computer. A program is sometimes referred as a **passive entity,** as it resides in a secondary memory.

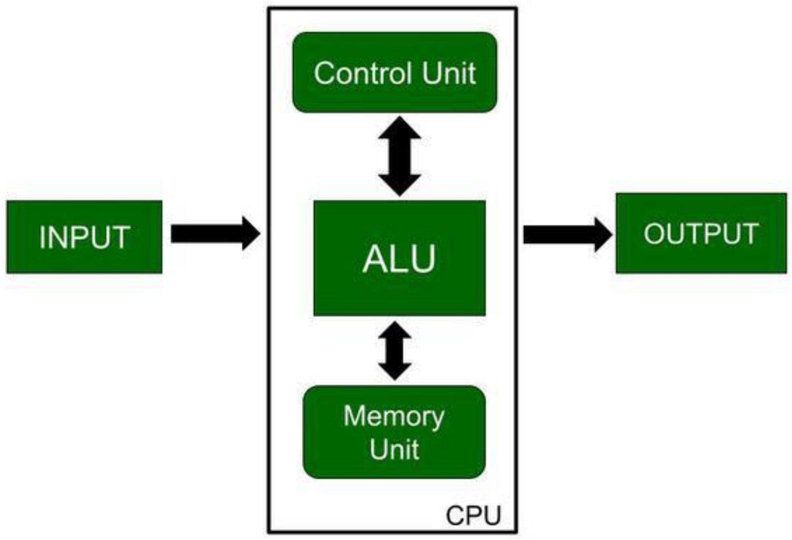
Operating systems today can run multiple **programs** at the same time. For example, you can use your browser (a program) but you can also listen to music on your media player (another program).

While a program is a passive collection of instructions typically stored in a file on disk, a process is the execution of those instructions after being loaded from the disk into memory. A **process** is the instance of a computer program, a **program in execution**. When you double click on the Google Chrome icon on your computer, you start a process which will run the Google Chrome program.

A **process** is sometimes referred as an **active entity** as it resides in the primary memory and leaves the memory if the system is rebooted. Several processes may be related to the same program. For example, you can run multiple instances of a notepad program. Each instance is referred as a process.

A **process** is managed by the **operating system** and has its own memory space, execution state and system resources such as CPU time, I/O devices, and files. The operating system schedules and manages processes to ensure efficient use of system resources and to provide a reliable and responsive computing environment.

Each **process** is managed inside the **primary memory** (RAM) and controlled and executed by using the **microprocessor**.



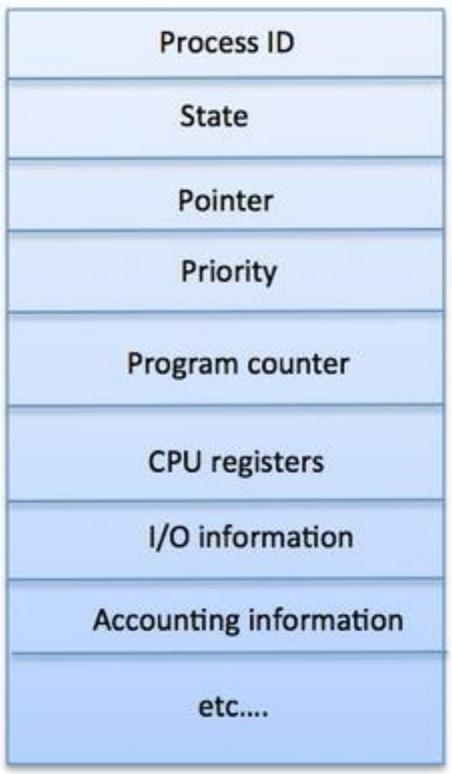
The **two** most **important components** of a process are the **program code** and a **set of data**. The program code is a set of instructions, whereas the set of data is the collection of data that is associated with the code of the program.

When a **process** is in execution, it can be characterised by the following **elements**:

- **Identifier**. Every process has a unique identification, which helps the processor distinguish between the processes. Every process has a **process Id** which acts as an identifier for that process.
- **State**. A process can be in a ready state, waiting for a state, a new state, a running state, and a terminated state.
- **Priority**. There are several processes in the system. But each process must have a priority level, which helps the processor decide which process to respond to first.

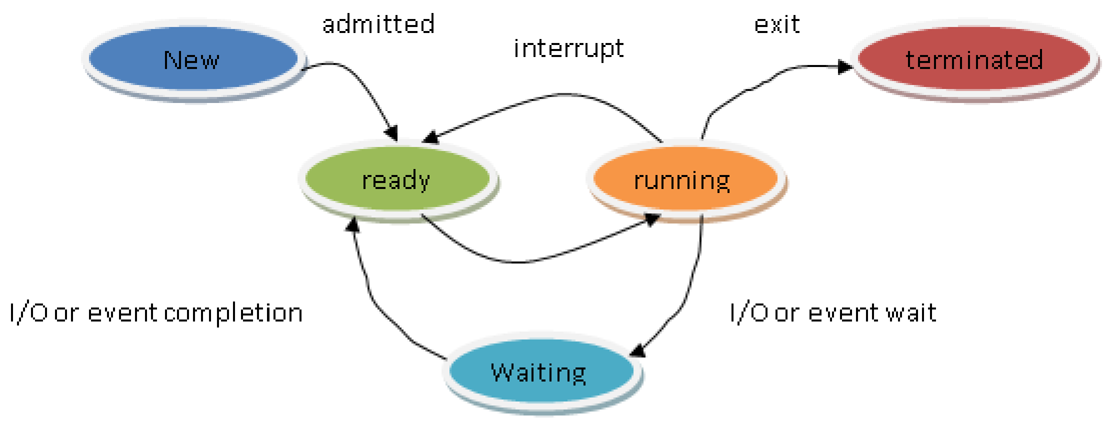
- **Program Counter**. As a process has a program code that has a set of instructions, the program counter helps the processor identify which instruction is to be executed next.
- **Memory pointer**. The memory pointer contains the pointer to the program code and to the data that are associated with the program.
- **Context data**. While the process is running, the processor stores intermediate results in registers, which are the context data.
- **I/O status information**. A process may require the I/O devices for which it makes the I/O requests. It also has information about the files used by the process.
- **Accounting information**. The accounting information has information about the time spent by the process on the processor.
- **Stack.** A stack is a data structure that stores information about the active subroutines of a computer program. It is used as a scratch space for the process. It is distinguished from dynamically allocated memory for the process that is known as the **heap**.

All this information is organised inside a data structure, and it is termed the **Process Control Block** (PCB).



A **Process Control Block** is a data structure maintained by the Operating System for each process. The PCB is identified by an integer process ID (PID). A PCB stores all the information needed to keep track of a process. It is an important tool that helps the OS support multiple processes and provides multiprocessing. It contains enough information so that, if an interruption occurs, the process can start from where it left off afterwards as if nothing had happened.

Process can communicate with each other and can be created, terminated and managed by system calls provided by the operating system.



A **process** can be in one of several **states**. These states are:

- **New**. The process to be executed by the CPU is being created.
- **Running**. The process is running and being executed by the CPU.
- **Ready**. The process is ready to be executed by the CPU, all the necessary resources are available to it.
- **Waiting**. The process is waiting for some resource to become available before it can continue executing.
- **Terminated**. The process has finished executing and has been removed from memory.

Only one process can be in a running state on any processor at a time, while multiple processes may be in a ready or waiting state.

Each process can run **concurrent subtasks** called **threads**. Each process is started with a single thread, often called the **primary thread**, but it can create additional threads from any of its threads.

Several processes may be associated with the same program. A process can create another process (child process).

You might have multiple instances of a single program. In that situation, each instance of that running program is a process. Each process has a separated memory address space. That separated memory address is helpful because it means that a process runs independently and is isolated from other processes. However, processes cannot directly access shared data in other processes. Switching from one process to another requires some amount of time (relatively speaking) for saving and loading registers, memory maps, and other resources.

Having independent processes matters for users. If a single process has a problem, you can close that program and keep using your computer. Practically, that means you can end a malfunctioning program and keep working with minimal disruptions.

Threads are sub-tasks of processes and if synchronized correctly can give the illusion that an application is performing everything at once. Without threads you would have to write one program per task, run them as processes and synchronize them through the operating system.

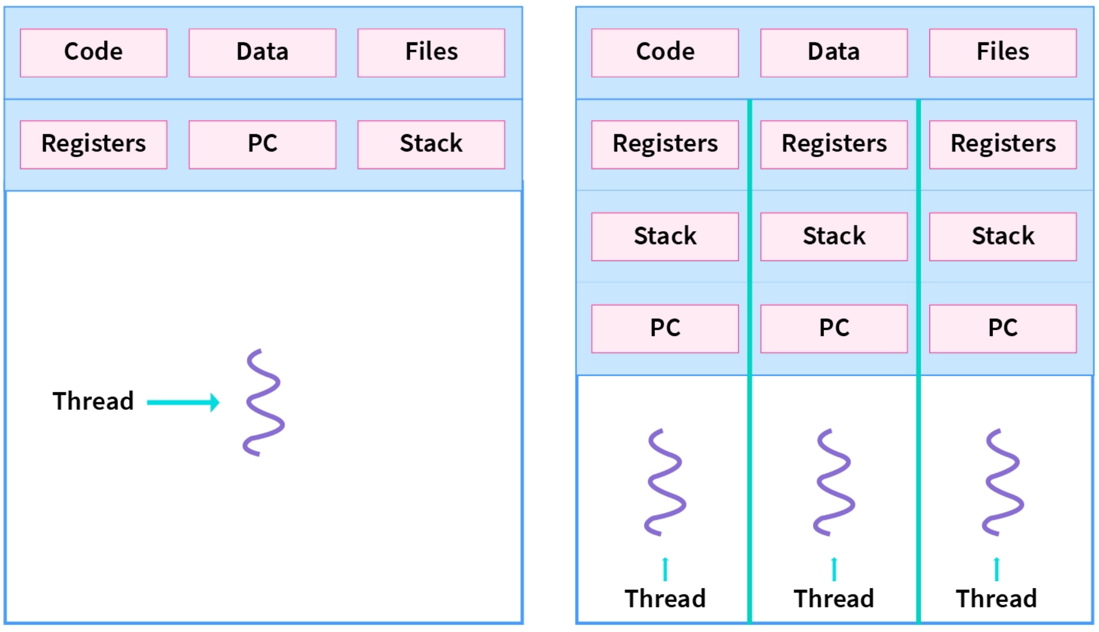
A **thread** of execution is the smallest sequence of instructions that can be independently managed by a scheduler. Threads are small components of a process and multiple treads can run concurrently and share the same code, memory, variables, etc. A thread is an execution unit that has its own program counter, a stack, and a set of registers that reside in a process. Threads cannot exist outside any process. Also, each thread belongs to exactly one process. The information like code segment, files, and data segment can be shared by the different threads.

Threads are popularly used to improve the application through parallelism. Actually, only one thread is executed at a time by the CPU, but the CPU switches rapidly between the threads to give an illusion that the threads are running on parallel.

Threads are also known as **light-weight processes**.

When a process starts, it receives an assignment of memory and other computing resources. Each thread in the process shares that memory and resources. With single-threaded processes, the process contains one thread.

Threads share the same address space, memory, machine states when they are under the same process. Therefore, there is no protection between them. A simple thing to remember is that processes have a high overhead and don't emphasise sharing resources while threads have a lower overhead and emphasise sharing the same resources.



Single-threaded process **Multithreaded process**

Threads are more suitable for high concurrency applications with shared states like web application servers or GUI apps where the resources can be shared among them. Processes are suitable for applications that require their own resources to be private and can only be accessed by themselves, which are highly due to the need for security purposes.

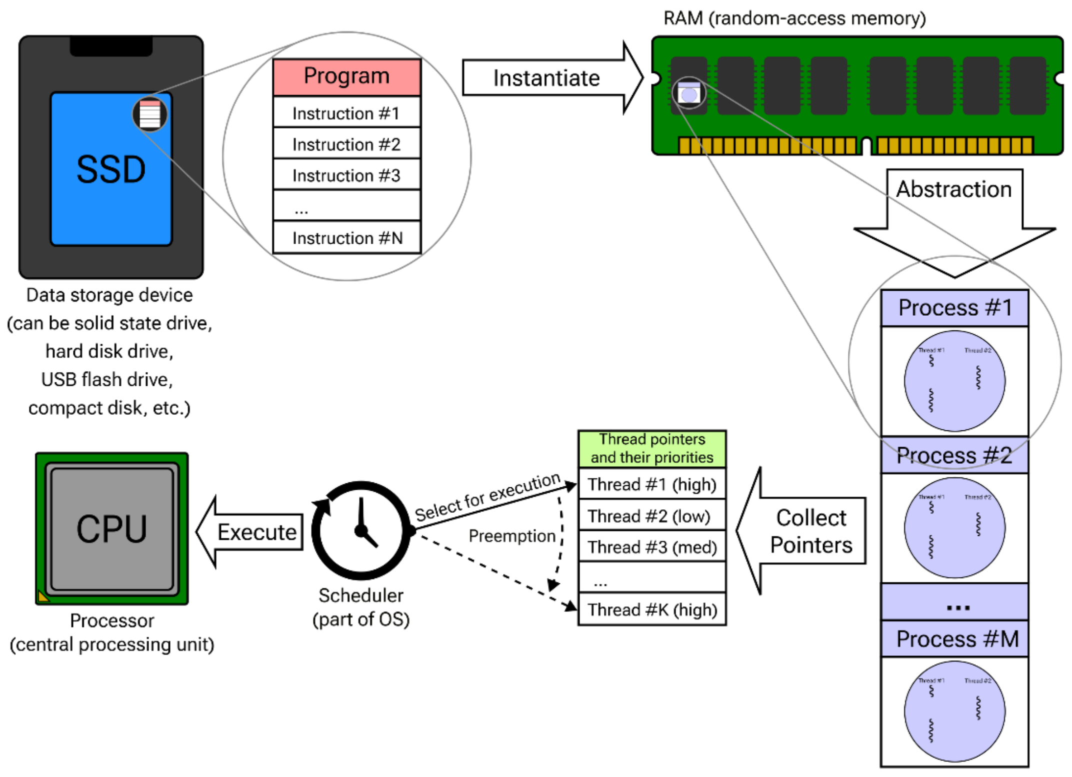
In multi-threaded processes, the process contains more than one thread and the process is accomplishing a number of things at virtually the same time.

The **stack** and the **heap** are the two kinds of **memory** available to a thread or process. Distinguishing between these kinds of memory matters because each **thread** will have its own **stack**. However, **all** the **threads** in a **process** will share the **heap**. The **heap** contains all **objects** created by the application. Objects reside there regardless of which thread created them. Objects on the heap are visible to all threads.

The **stack** is a **memory area** that stores **local variables**, **object references**, and information about which **methods** the thread has called.

Since threads share the same address space as the process and other threads within the process, it is easy to communicate between the threads. The disadvantage is that one malfunctioning thread in a process can affect the viability of the process itself.

The following **infographic** explains what happens when a **computer program** is **executed**.



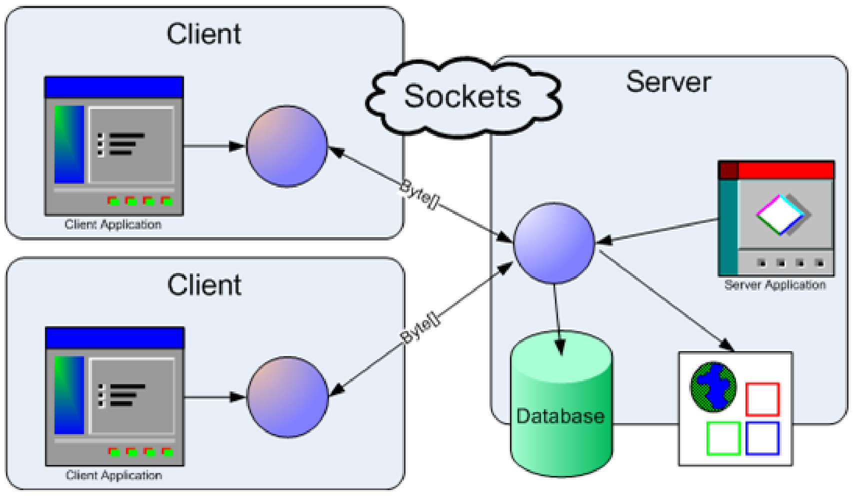
Services

Services are a particular type of processes that provide services to other processes. They run in the background and are not directly used by users.

Other processes, on the other hand, run in the foreground and have a user interface with which users interact.

Services run continuously. They are normally started by the operating system during its start-up and run in the background. They usually provide information about their execution in log files in which they record all events that occur during their execution and details about the actions they perform.

Services can provide services to other processes on the same computer, and to other processes on other computers.



In this case, they typically communicate over a communications network, using standard network protocols, such as TCP or UDP of the TCP/IP protocol family. Such network services are a fundamental component of distributed systems. In this context, server applications, or simply servers, serve other applications, and client applications, or simply clients, use these services. Of course, an application can provide a service to other applications and at the same time use the services provided by another application. That is, it can act as both a client and a server.

Each operating system has its own mechanisms for managing services.

Concurrency and parallelism

Concurrency is the ability of a program to deal (not do) many things at once and is achieved by **multithreading**. Do not confuse concurrency with **parallelism,** which is about doing many things at once. A system is said to be concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously.

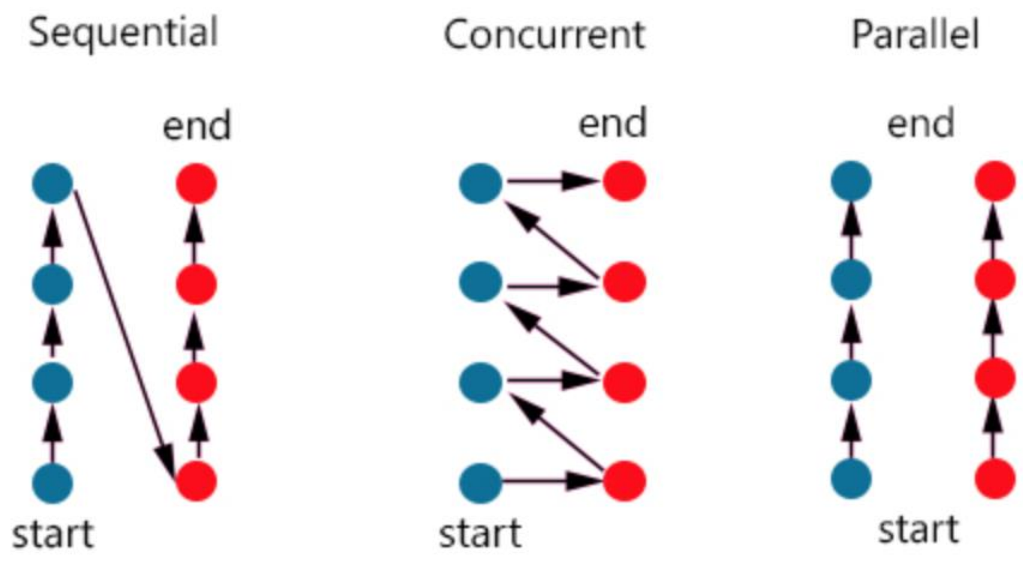
Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

An application can be concurrent but not parallel, so it processes more than one task at the same time, but no two tasks are executing at the same time instant.

An application can be parallel but not concurrent, so it processes multiple sub-tasks of a task in multi-core CPU at the same time.

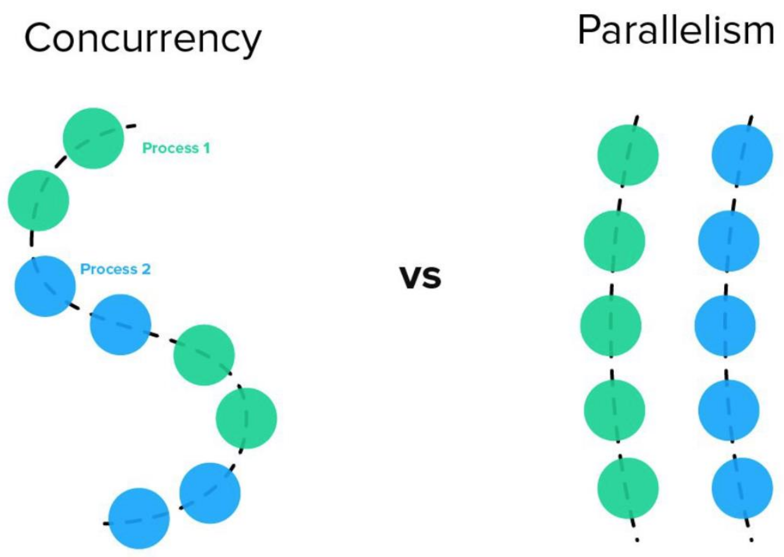
An application can be neither parallel nor concurrent, so it processes all tasks one at a time or sequentially.

An application can be both parallel and concurrent, so it processes multiple tasks concurrently in multi-core CPU at the same time.

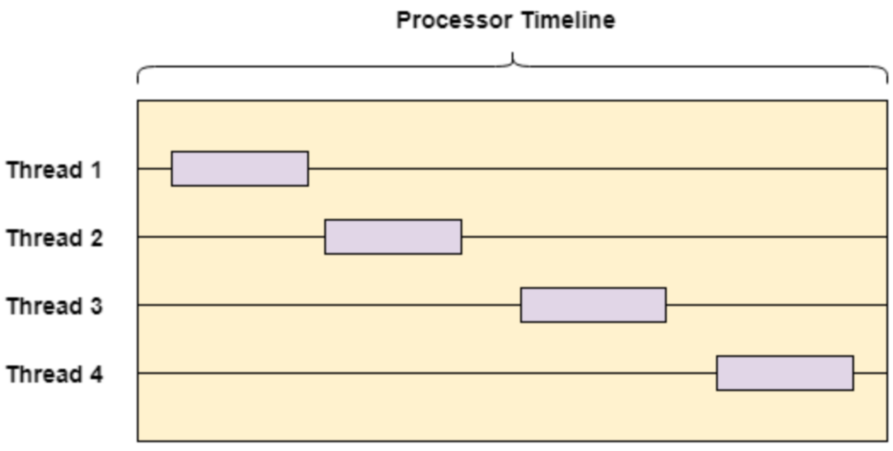


Concurrency is a property of a program at design level where two or more tasks can be in progress simultaneously and is related to how an application handles multiple tasks it works on. It may process one task at a time (sequentially) or work on multiple tasks at the same time (concurrently). Two or more tasks may start, run, and complete in overlapping time periods. However, it does not necessarily mean they will ever be running at the same instant.

Parallelism is a runtime property where two or more tasks are executed simultaneously. An application may process a given task serially from start to end or split it into subtasks to run them in parallel.

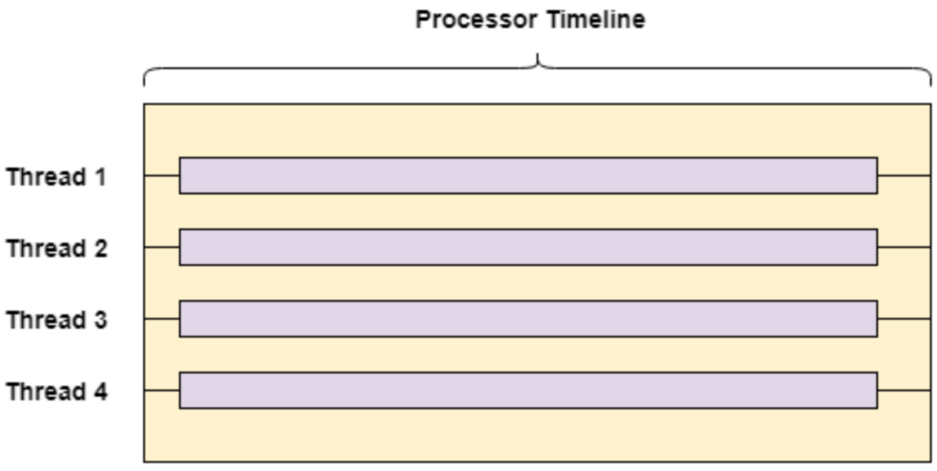


When concurrency is implemented, speed is increased by overlapping I/O activities of one process with CPU process of another process. Multiple processes progress simultaneously. Although the CPU executes only one thread at a time, these threads can be switched on and off as needed, so that no thread is fully completed before another is scheduled i.e. all threads run concurrently.



Concurrency

Parallelism is devised for increasing computational speed by using multiple processors. It is a technique of simultaneously executing different tasks at the same instant by involving several independent computing processing units or computing devices which are parallelly operating and performing tasks to increase computational speed-up and **improve throughput**. Parallelism results in overlapping of CPU and I/O activities in one process with CPU and I/O activities of another process.



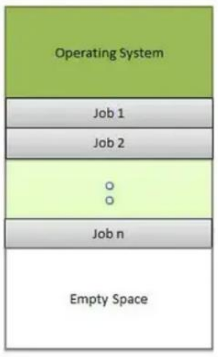
Parallelism

Context Switching

A context switch is the process of **storing** and **restoring** the **state** of a **process** or **thread** so that the execution can be resumed from the same point later. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system.

Multiprogramming

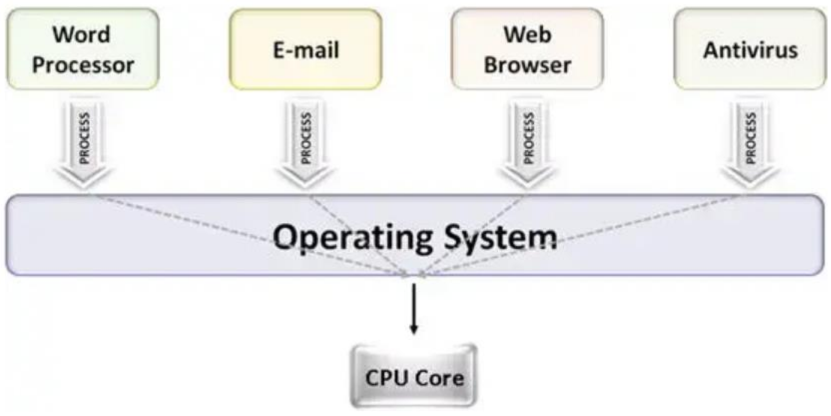
Multiprogramming is the ability of an operating system to execute more than one program on a single processor machine. More than one task/program/job/process can reside in the main memory at one point of time. A computer running Word Processor and a Chrome browser simultaneously is an example of multiprogramming.



Memory layout for Multiprogramming System

Multitasking

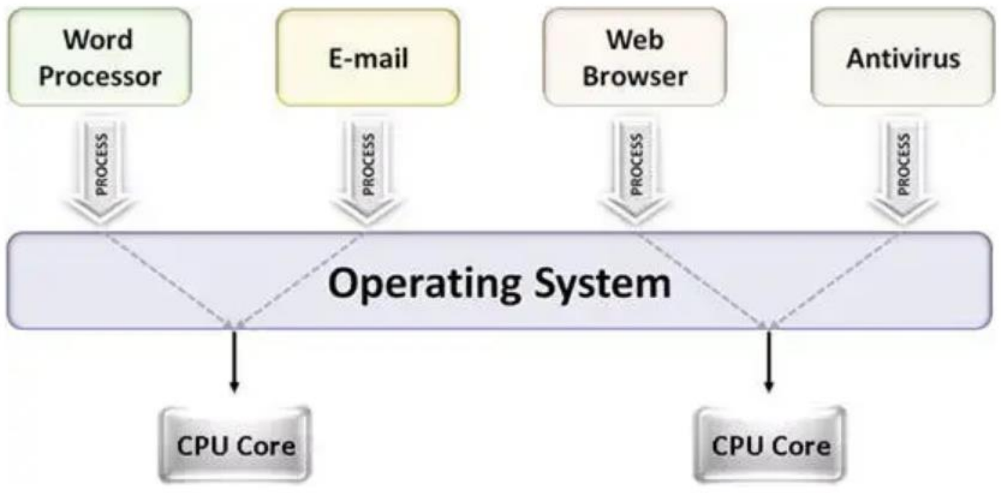
Multitasking is the ability of an operating system to run more than one task simultaneously on a single-processor machine. A single processor machine cannot run more than one task at the same time. In fact, the CPU switches from one task to the next task so quickly that it appears as if all the tasks are running at the same time.



Multitasking System

Multiprocessing

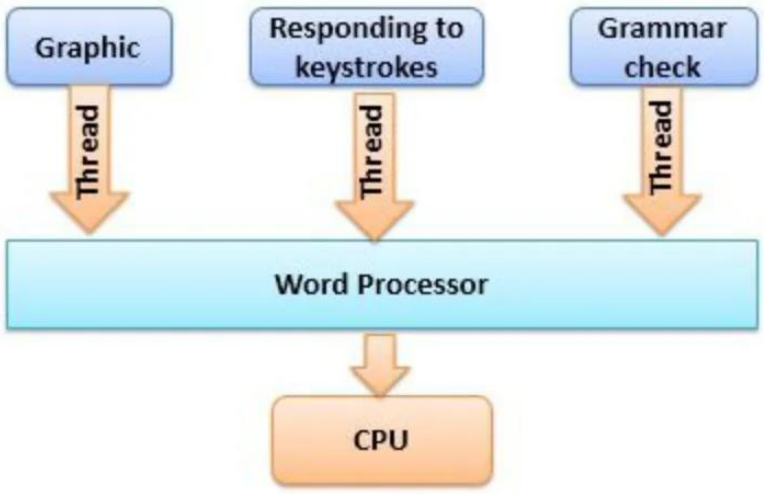
Multiprocessing is the ability of an operating system to execute more than one process simultaneously on a multiprocessor machine. In this, a computer uses more than one CPU at a time.



Multiprocessing System

Multithreading

Multithreading is the ability of an operating system to execute the different parts of a program called threads at the same time.



Multithreading

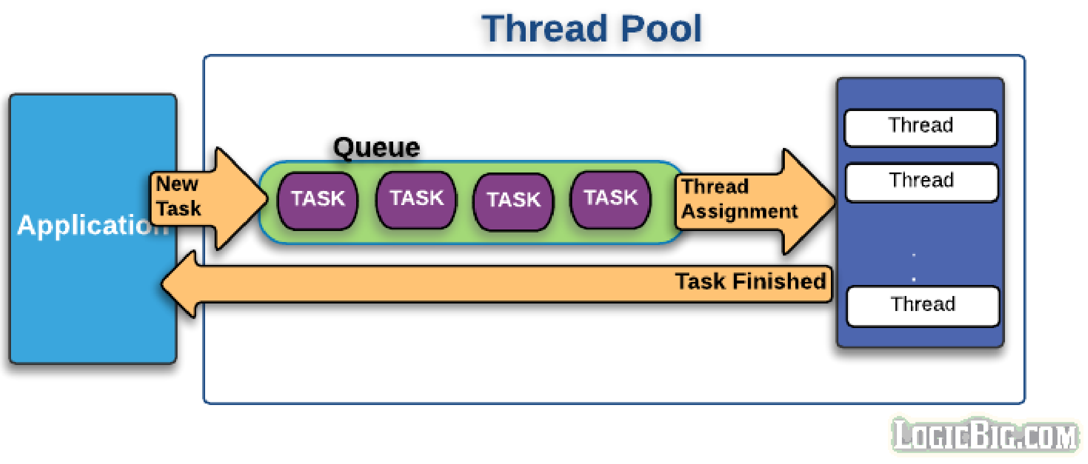
Distributed programming

Distributed programming allows the development of programs and applications that run simultaneously on multiple interconnected computing devices. This approach allows for better resource utilisation, promotes fault tolerance and facilitates network operations. Distributed Programming is a method of software design and implementation that allows multiple computers to work together to solve a common task efficiently. This approach makes it possible to exploit the power of multiple computing resources and improve the performance and reliability of a system.

Thread Pools

Thread pools allow decoupling the dispatching and execution of tasks. You have the option to expose the configuration of an executor while deploying an application or to switch from one executor to another seamlessly.

A thread pool consists of homogenous worker threads that are assigned to execute tasks. Once a worker thread completes a task, it is returned to the pool. Typically, thread pools are linked to a queue from which tasks are dequeued for execution by the worker threads.

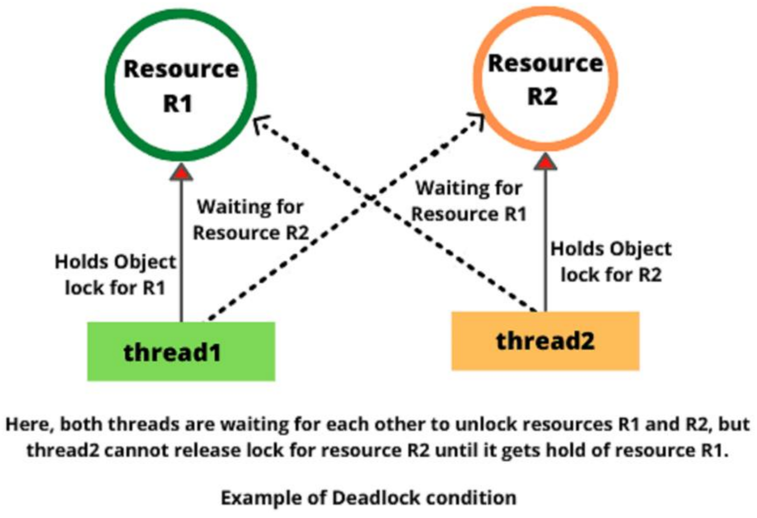


A thread pool can adjust to the size of the threads it contains. A thread poolcan also replace a thread if it dies due to an unexpected exception. The use of a thread pool immediately alleviates the evils of manual thread creation. Locks are a very important feature that makes multithreading possible. Locks are a synchronization technique used to limit access to a resource in a multithreaded environment. A good example of a lock is a mutex.

Issues Involved with Multiple Threads

Deadlock

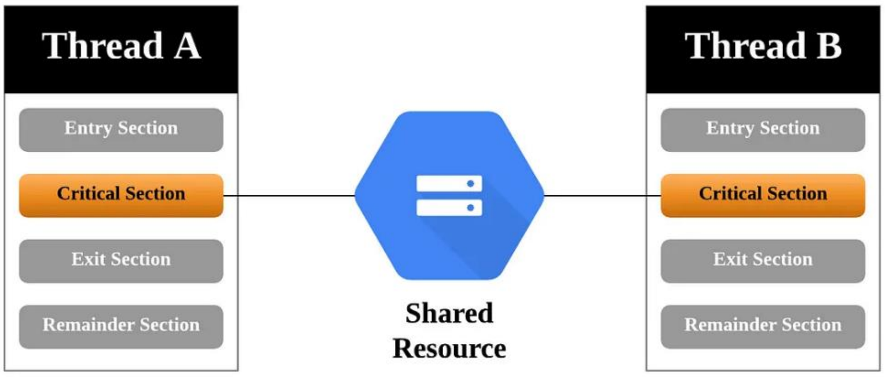
Deadlocks happen when two or more threads can’t make any progress because the resource required by the first thread is held by the second and the resource required by the second thread is held by the first.


Resource R1
Resource R2
Waiting for Resource R2
Waiting for Resource R1
Holds Object lock for R1
Holds Object lock for R2
thread1
thread2
Here, both threads are waiting for each other to unlock resources R1 and R2, but thread2 cannot release lock for resource R2 until it gets hold of resource R1.

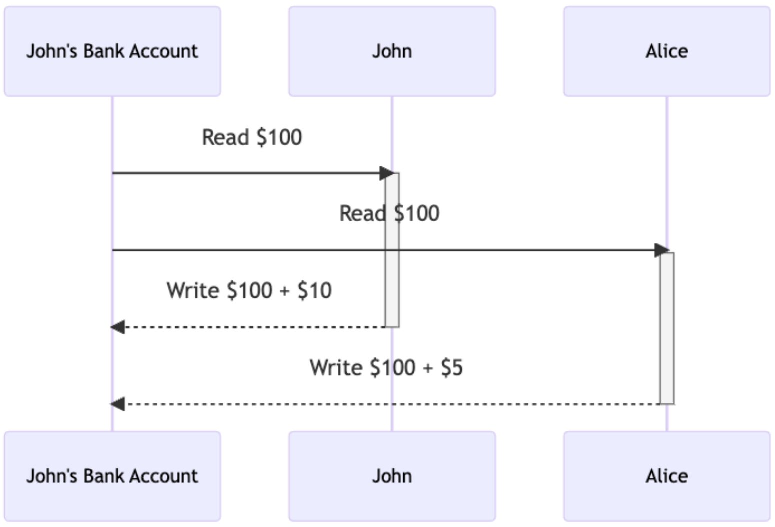
Example of Deadlock condition

Race conditions

Critical section is any piece of code that has the possibility of being executed concurrently by more than one thread of the application and exposes any shared data or resources used by the application for access.



Race conditions happen when threads run through critical sections without thread synchronization. The threads "race" through the critical section to write or read shared resources and depending on the order in which threads finish the "race", the program output changes.

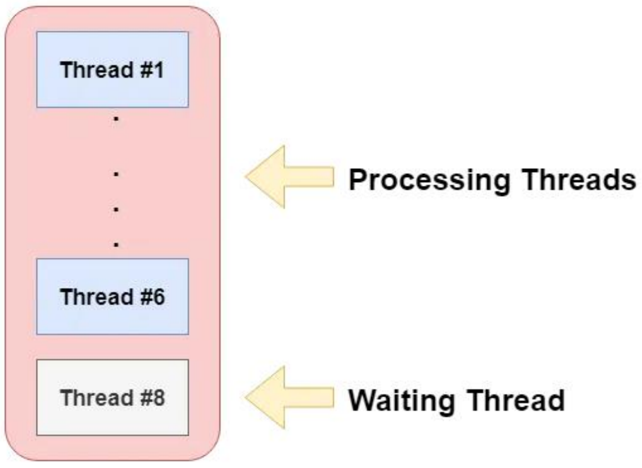


In a race condition, threads access shared resources or program variables that might be worked on by other threads at the same time causing the application data to be inconsistent.

Starvation

Other than a deadlock, an application thread can also experience starvation, where it never gets CPU time or access to shared resources because other "greedy" threads hog the resources.

Starvation occurs if a process is indefinitely postponed.



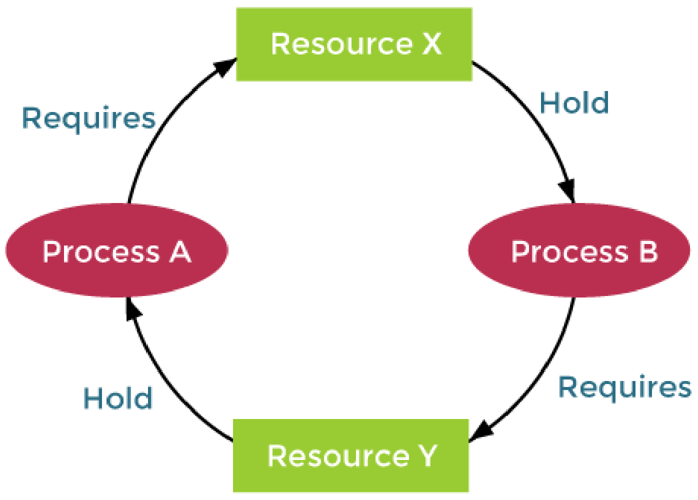
Starvation

Livelock

A livelock happens when two threads keep taking actions in response to the other thread instead of making any progress.

The best analogy is to think of two people trying to cross each other in a hallway. Ann moves to the left to let Helen pass, and Helen moves to her right to let Ann pass.

Both block each other now. Ann sees she's now blocking Helen and moves to her right and Helen moves to her left, seeing she's blocking Ann. They never cross each other and keep blocking each other. This scenario is an example of a livelock.


Resource X
Requires
Hold
Process A
Process B
Hold
Requires
Resource Y