Concurrency

A definition of the word *concurrent* from the Webster dictionary is:

> converging, meeting, intersecting, **running together** at one point.

Applying this to programming, we can say that concurrent programming is a programming paradigm that allows multiple processes to run in overlapping time periods. This means that tasks can start, run, and complete in an overlapping manner, rather than sequentially.

Concurrency vs Parallelism

There are mainly three ways to execute multiple tasks.

Single Process

I we only have one CPU (with one core), we can only run one task at a time. If we have a set number of tasks to execute, we can decide to run them sequentially, one after the other. This is called sequential programming.

This is the way the first computers worked. They had a single CPU and could only execute one instruction at a time.

Multiprogramming

Even if we still have a single CPU, we can run multiple tasks by switching between them. This is called multiprogramming. The CPU switches between tasks so quickly that it gives the illusion that the tasks are running simultaneously.

This allows the CPU to be utilized more efficiently, as if a task is waiting for I/O operations to complete, the CPU can switch to another task and continue executing it until the I/O operation is done.

This is done by the operating system, which manages the scheduling of tasks. The OS allocates a small time slice to each task and switches between them rapidly.

Parallelism

As the technology advanced, we started to have multiple CPUs (or multiple cores in a single CPU). This allows us to run multiple tasks truly simultaneously. This is called parallelism.

At the same time, individual cores can also use multiprogramming to run multiple tasks by switching between them.

Pros and Cons of Parallel Processing

Pros:

- Simultaneous execution of tasks.
- Leads to significantly faster execution of tasks.
- Allows for more complex computations.
- Cost-effective as you can substitute a single supercomputer with a cluster of less expensive machines.

Cons:

- Compilers and IDEs for parallel systems are harder to develop.
- Parallel algorithms are more complex to design and implement.
- Energy consumption can be higher.
- More complex data access and communication between tasks.

Distributed Systems

Andrew S. Tanembaum and Marteen Van Stem, in their book *Distributed Systems: Principles and Paradigms* loosely define a distributed system as:

> A distributed system is a collection of independent computers that appears to its users as a single coherent system.

One of the most known examples of distributed systems is the Internet. It is a collection of independent computers (servers, routers, etc.) that work together to provide a coherent system (the Web).

Other of the most known examples of distributed systems are the systems known as *cloud computing*. These systems are made up of multiple computers that work together to provide a service (e.g., Google Drive, Dropbox, etc.). Some examples of cloud computing providers are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

Some of the characteristics of a distributed system are:

- Concurrency: Programs are executed concurrently on multiple computers.
- Existence of a common clock: All computers in the system have a common clock to synchronize their actions via messages.
- Independent failures: Each computer in the system can fail independently of the others.

Pros and Cons of Distributed Systems

Pros:

- Data and resources are shared.
- Can grow on demand.
- Load balancing.
- Can be more reliable.

Cons:

- Increased complexity.
- Require specialized software.
- Complexities related to network communication.
- Security issues.

Critical Section

A critical section is a part of a program that accesses shared resources (like shared data structures or devices) and must not be concurrently accessed by more than one thread of execution.

When multiple threads access a shared resource without proper synchronization, it can lead to inconsistent or incorrect results. To prevent this, we use synchronization mechanisms like mutexes (mutual exclusion), semaphores, or monitors to ensure that only one thread can access the critical section at a time.

Deadlocks

A deadlock can occur when two or more threads need to execute their critical sections, but each thread is waiting for a resource that another thread holds. As a result, none of the threads can proceed, and they are all stuck in a state of waiting.

Race Conditions

Race condition occurs when multiple threads or processes read and write the same variable i.e. they have access to some shared data and they try to change it at the same time. In such a scenario threads are *racing* each other to access/change the data.

A race condition is a situation that may occur inside a **critical section** if the critical section is not properly protected.
This happens when the result of multiple thread execution in a critical section differs according to the order in which the threads execute.
Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper **thread synchronization** using locks or atomic variables can prevent race conditions.

Bernstein's conditions

When we have multiple threads running concurrently, we need to ensure that they can access shared resources safely. To do this, we need to check if two threads can run in parallel without interfering with each other. This is where Bernstein's conditions come into play.

Let's say we have two statements, `S1` and `S2`, that we want to execute in parallel. Each statement has a set of variables that it reads from and a set of variables that it writes to. We can denote these sets as follows:

- `R(S1)` and `R(S2)`: the set of variables read by statements `S1` and `S2`, respectively.
- `W(S1)` and `W(S2)`: the set of variables written by statements `S1` and `S2`, respectively.

We can say that `S1` and `S2` can be executed in parallel if the following conditions are met:

- `R(S1) ∩ W(S2) = ∅` (no variable is read by `S1` and written by `S2`).
- `W(S1) ∩ R(S2) = ∅` (no variable is written by `S1` and read by `S2`).
- `W(S1) ∩ W(S2) = ∅` (no variable is written by both `S1` and `S2`).

Let's see an example to illustrate this:

```
// S1:
a = x + y

// S2:
b = z + 1

// S3:
c = a - b
```