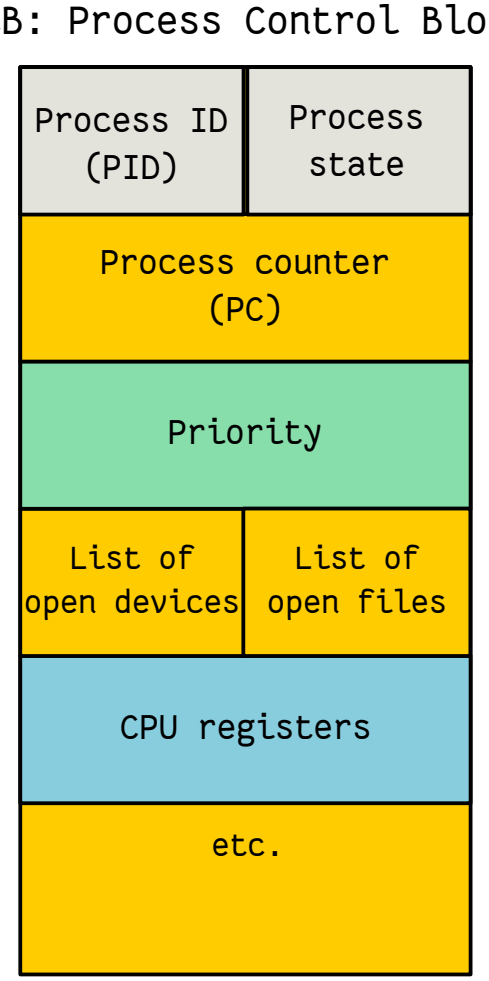
Programs, Processes and Threads

Processes and Programs

A **program** is a sequence of instructions written to perform a specified task with a computer. It is a passive entity, meaning it does not perform any action until it is executed.

A program can be seen as a black box. The instructions of the program will tell to the system how to obtain the desired output from the given input.

A **process**, on the other hand, is what you get when you run a program. Ie. a **process** is an instance of a program that is being executed. It is an active entity, meaning it performs actions when it is running. A process has its own memory space, system resources, and execution context. All the information about the process that the OS needs to manage it is stored in an structure called PCB (Process Control Block)



Every time you run a program the system starts a new process. Each process is independent and isolated from other processes, which means that they do **not share memory or resources** unless explicitly designed to do so.

A **thread** is the smallest unit of execution within a process. A process can have multiple threads, which **share the same memory space and resources of the parent process**. Threads are used to perform tasks concurrently within a process, allowing for more efficient use of system resources.

Concurrent vs Parallel Programming

Concurrent programming means that, give a length of time, multiple tasks can make progress simultaneously. This does **not necessarily** mean that they are **executing at the same time**, but rather that they are being managed in a way that allows them to share resources and time effectively.

For two task (or processes) to be concurrent, one task should start before the other finishes. This means that the tasks can **overlap in time**, but they **do not have to be executing at the same time**.

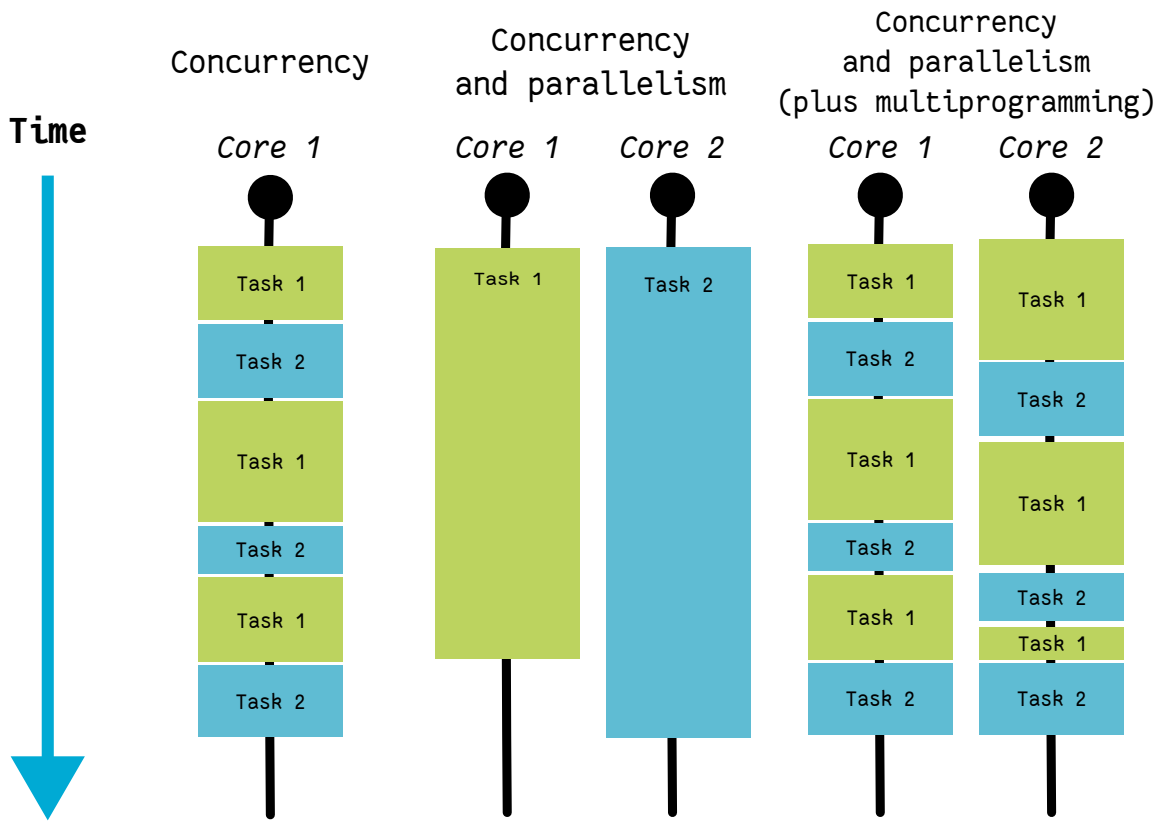
On the other hand, **parallel** programming means that **multiple tasks are executing at the same time**. This requires multiple processors or cores to be available, as each task needs its own dedicated resources to run simultaneously.

We can say that to achieve concurrency we can follow two paths:

- Through **multiprogramming**: The OS slices the CPU time and assigns those lime slices to the processes. When a process reach it's time limit the OS remove the process from the CPU and gives time to another

process. This way, **over a period of time** the user can see that there are may task running **at the same time**. We don't need more than one CPU to achieve this.
- Through **multiprocessing**: To implement this system we need to have more than one CPU (or cores). The OS sends as many process as it can to as many CPUs (or cores) the system have. So we can literally say that there are more than one process running **at the same time** at any given moment.

Obviously the OS combines both approaches to get the benefits of both techniques:



Every time we have parallelism, we have concurrency, but not every time we have concurrency we have parallelism.

Motivation

The primary motivation for concurrent programming is to improve the responsiveness of a system and the efficiency in the use of hardware resources.

We will list all the improvements that concurrency brings in the following points:

- Optimices the use of CPU and other resources. While a process is performing I/O operations (like reading from a file or waiting for network access), other processes can use the CPU to perform computations. This leads to better utilization of system resources and improved overall performance.
- Provides interactivity to the user. In a concurrent system, the user interface can remain responsive while background tasks are being executed. This is particularly important in applications that require real-time interaction, such as video games or multimedia applications.
- Improves responsiveness. A server, due to concurrency, can handle multiple client requests simultaneously, reducing wait times and improving the overall user experience.
- Allows for a more comprehensive design of complex systems. Complex systems often involve multiple

components that need to interact with each other. Concurrency allows these components to operate independently, making it easier to design and maintain the system.
- Increases security and fault tolerance. In a concurrent system, if one process fails, it does not necessarily affect the other processes. This can improve the overall reliability and security of the system.

On top of this, current technology trends are pushing towards multi-core processors and distributed computing, making concurrency a fundamental aspect of modern software development. If we want to take advantage of the available hardware, we need to design our applications to be concurrent.

Inter-process Communication (IPC)

When we have multiple processes running concurrently, they may need to communicate with each other to share data or coordinate their actions. This is known as **inter-process communication (IPC)**.

Concurrent programming requires mechanisms for IPC to ensure that concurrent processes would be able to synchronize with each other and share data safely.

This communication is dependent on the Operating System (OS) and can be achieved through various methods, such as:

- Message passing: This is commonly used when the processes are being run on different machines. Both processes must share the same protocol to be able to communicate.
- Shared memory: This can only be used when the processes are being run on the same machine. Both processes must have access to the same memory space to be able to communicate.

> **WARNING!**
>
> Although both methods are valid, they have different trade-offs.
> **Sharing memory between processes is more efficient than message passing, but it requires careful synchronization to avoid data corruption.**

Other aspect of communication we need to consider is the way the communication happen:

- Asynchronous communication: In this case, the sender process sends a message and continues its execution without waiting for a response from the receiver process. The receiver process can handle the message at its own pace.
- Synchronous communication: In this case, the sender process sends a message and waits for a response from the receiver process before continuing its execution. The receiver process must handle the message immediately.

Services and Threads

A program is composed by a sequence of instructions that determine a flow of execution. This flow of execution is called a **thread**. The flow of execution depends not only on the instructions of the program, but also on the input data and the environment in which the program is executed.

Taking this into account we can say that a process with only one flow of execution is called a **single-threaded process**. A process with multiple flows of execution that run concurrently is called a **multi-threaded process**.

A process can be:

- **Sequential** (single-threaded): It has only one thread of execution. This means that the process can only execute one instruction at a time.

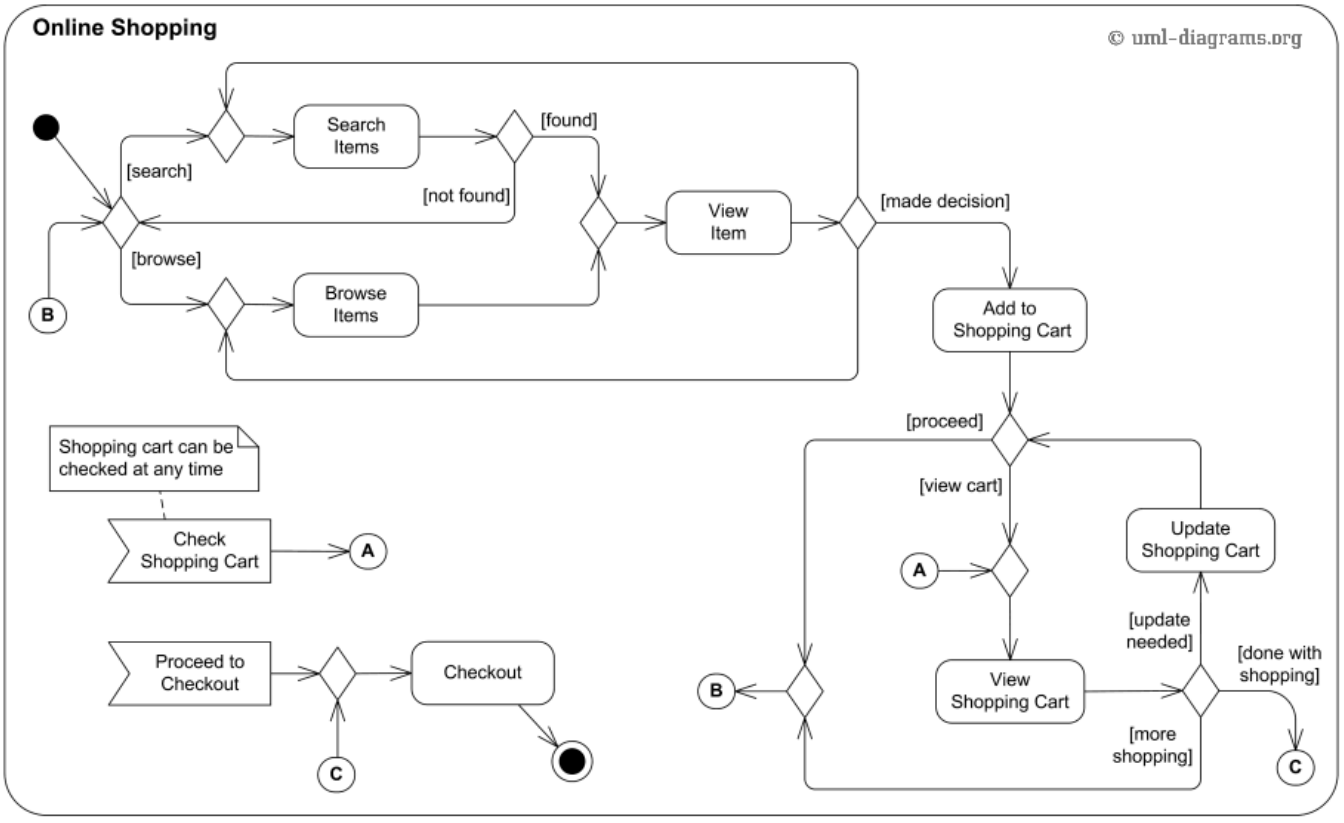
- **Concurrent** (multi-threaded): It has multiple threads of execution. This means that the process can execute multiple instructions at the same time.

Single-threaded process

A single-threaded process can be described as a sequence of instructions that are executed one after the other. The process starts at the beginning of the instruction sequence and executes each instruction in order until it reaches the end of the sequence. This does not mean that there is only one path of execution, as the process can have conditional statements and loops that can change the flow of execution. But it does mean that, for the same input, the process will always follow the same path of execution.

Every instruction is executed one after the other, and the process can only do one thing at a time. If the process needs to wait for an I/O operation to complete, it will block and wait until the operation is finished before continuing with the next instruction.

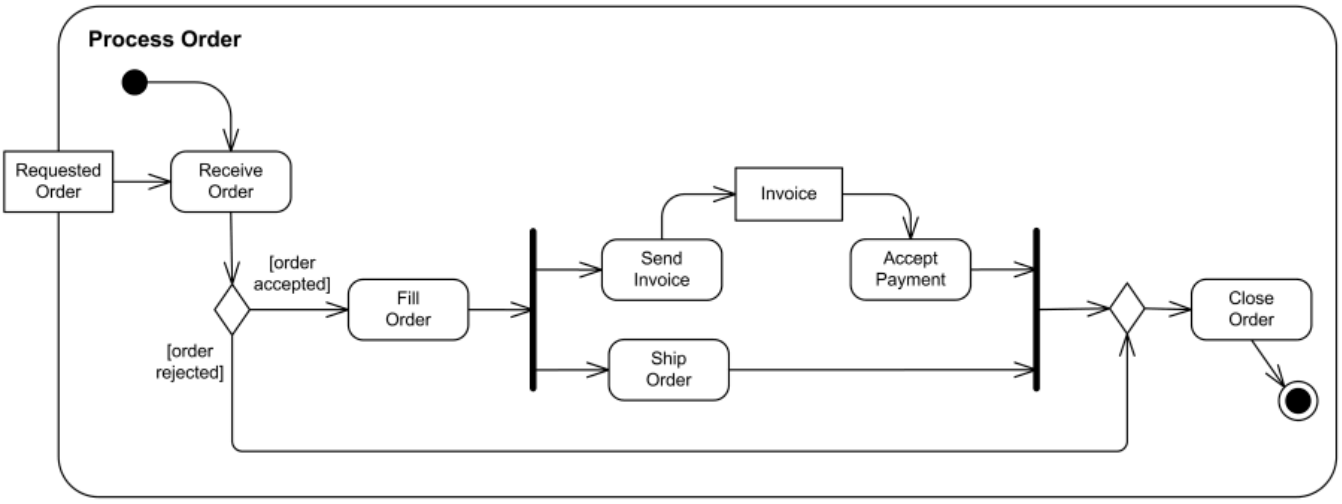
If we focus on the code that defines a single-threaded process, we can see that it is relatively simple. The code is executed in a linear fashion, and there are no concerns about synchronization or communication between threads. This means that it's easier to test for correctness, as there are fewer variables to consider. If the code is simple enough, we can use *white box testing* techniques to ensure that the code behaves as expected.



Multi-threaded process

In a multi-threaded process, there are multiple threads of execution that can run concurrently. Each thread has its own flow of execution, and they can execute instructions independently of each other. This means that the process can do multiple things at the same time. We can see this as a process composed by multiple single-threaded processes that share the same memory space and resources. Each of this *mini-processes* will be known as a thread.

In this kind of programs there is the need to implement **mechanisms for synchronization and communication** between threads. Will be times when two threads will need to access the same resource, memory for example, and we need to ensure that this access is done in a safe way to avoid data corruption. Other times one thread will need to wait for another thread to complete a task before it can continue its execution and we need that thread to be able to notify the waiting thread when it has completed the task.



Multi-threaded processes lead to indeterminism as, for the same input, the program can give different outputs depending on the order in which the threads are executed. This makes it harder to test for correctness, as there are more variables to consider. That the output can change depending on the order of execution of the threads does not mean that the program is incorrect, but it does mean that we need to be more careful when testing it.

Let's see this with an example:

This program creates two new threads that will each print 5 numbers, one through five, one number per second. The main thread will wait for both threads to finish before exiting.

```
package main

import (
 "fmt" // Package for formatted I/O.
 "time" // Package for measuring and displaying time (includes Sleep function).
)

// We define a function that print numbers from 1 to 5 (one number each second) and we _give
it_ a name to be able to distinguish between the two threads.
func printNumbers(name string) {
 for i := 1; i <= 5 ; i++ {
  fmt.Printf("%s: %d\n", name, i)
  time.Sleep(1 * time.Second)
 }
}

func main() {
    // To launch a new thread, we use the `go` keyword followed by the function call.

    // This tow threads will run concurrently, so the output will be interleaved. The output
won't be deterministic, as it will depend on the order in which the threads are executed by
the Go runtime.
```

```
    go printNumbers("Thread 1")
    go printNumbers("Thread 2")

    // Prevent main from exiting
    select {}
}
```

Services

A service is a process which runs in the background and performs a specific task. Services are typically used to provide functionality to other processes or to the system as a whole. They can be started automatically when the system boots up, and they can run continuously in the background without user intervention.

Services do not usually interact directly with the user, but they can provide functionality that is used by other processes or applications. For example, a web server is a service that provides web pages to users when they request them via a web browser.

Threads vs processes

The main difference between a process and a thread is that the first is managed by the OS while the second is managed by it's parent process.

The OS is responsible of spawn new processes. And then it will assign them time and resources, control the communication between processes, etc.

The entity responsible to spawn new threads is a process. With threads is the parent process the entity in charge to manage the threads: synchronization, communication, etc.

With independence of this arrangement the OS can send different threads to different CPUs / cores.