Threads and Multithreading

Concurrency

> Concurrency is the ability of a program to execute multiple tasks at the same time.

Often, a program needs to perform multiple tasks at the same time. For example, a web browser needs to download a file and display a web page at the same time. One option is to have parallel tasks running on different cores of the CPU. This is called parallelism. Another option is to have multiple tasks running on the same core of the CPU. A task starts executing on the CPU. When the task is waiting for some input/output, the CPU switches to another task. This is called concurrency.

Concurrency is crucial in programming for several reasons, and it becomes especially relevant in scenarios where multiple tasks need to be performed simultaneously. Key reasons for embracing concurrency include:

- `Improved Performance`: Concurrency allows for the parallel execution of tasks, leading to improved overall performance. By utilizing multiple resources concurrently, applications can accomplish more work in less time.
- `Responsiveness`: In user-facing applications, concurrency ensures responsiveness by preventing long-running tasks from blocking the user interface. Asynchronous processing or parallel execution keeps the application responsive to user interactions.

- `Resource Utilization`: Efficient concurrency allows the optimal utilization of system resources. Whether it's CPU-bound computations or I/O-bound tasks, concurrency ensures that resources are not idle, contributing to better resource utilization.
- `Scalability`: Concurrency is essential for scalable applications, especially in the context of web servers and distributed systems. Scalable systems can handle increased workloads by efficiently utilizing multiple threads or processes.

Concurrency vs Parallelism

- `Concurrent` - At the same time, but not necessarily at the same instant. It is possible for multiple threads to be at different stages of execution at the same time but not being processed together. A single core CPU can only execute one thread at a time. But it can switch between threads very quickly. This is called context switching. This is how concurrency is achieved. A single core CPU can have concurrency but not parallelism.
- `Parallel` - At the same time and at the same instant. It is possible for multiple threads to be at different stages of execution at the same time and being processed together. A single core CPU cannot achieve parallelism. It can only achieve concurrency. A multi-core CPU can achieve both concurrency and parallelism.

Processes

A process is an independent program that runs in its own memory space. Each process has its own Python interpreter and runs independently of other processes. Processes are well-suited for CPU-bound tasks and can take full advantage of multi-core processors.

When you run a Python program, the interpreter creates a process. This process is assigned a process ID (PID) by the operating system. The process ID is a unique number that is used to identify the process. The process ID is used to perform operations on the process, such as terminating the process.

You can use the `os` module to get the process ID of the current process:

```python
import os

print(os.getpid())
```

The current process is also called the main process. For further independent execution, the main process can create new processes. The main process is called the parent process, and the new processes are called child processes. You can use the `multiprocessing` module to create new processes:

```python
import multiprocessing

def print_process_id():
    print(f"Process ID: {os.getpid()}")

if __name__ == "__main__":
    print_process_id()
    process = multiprocessing.Process(target=print_process_id)
    process.start()
```

Some points to note about the above code:

- The `multiprocessing` module has a `Process` class that can be used to create a new process.
- The `Process` class takes a `target` parameter that is used to specify the function that is executed by the process.
- The `Process` class has a `start` method that is used to start the process. The `start` method calls the function specified by the `target` parameter.

Let's write a program that calculates the sum of two arrays using multiple processes:

```
from multiprocessing import Process

def calculate_sum(data: List[int]):
    return sum(data)

if __name__ == "__main__":
    data_list = [list(range(10)), list(range(10, 20))] # Creating a 2D list of data

    # Creating a Process for summing each list
    processes = [Process(target=calculate_sum, args=(data,)) for data in data_list]

    # Starting and running processes concurrently
    for process in processes:
        process.start()

    # Waiting for all processes to complete
    for process in processes:
        process.join()
```

Here we first created a 2D list containing two lists of integers. Then we created a process for each list. We started all the processes and waited for them to complete. The `join` method is used to wait for a process to complete. The `join` method is a blocking method. It waits until the process is completed and then returns.

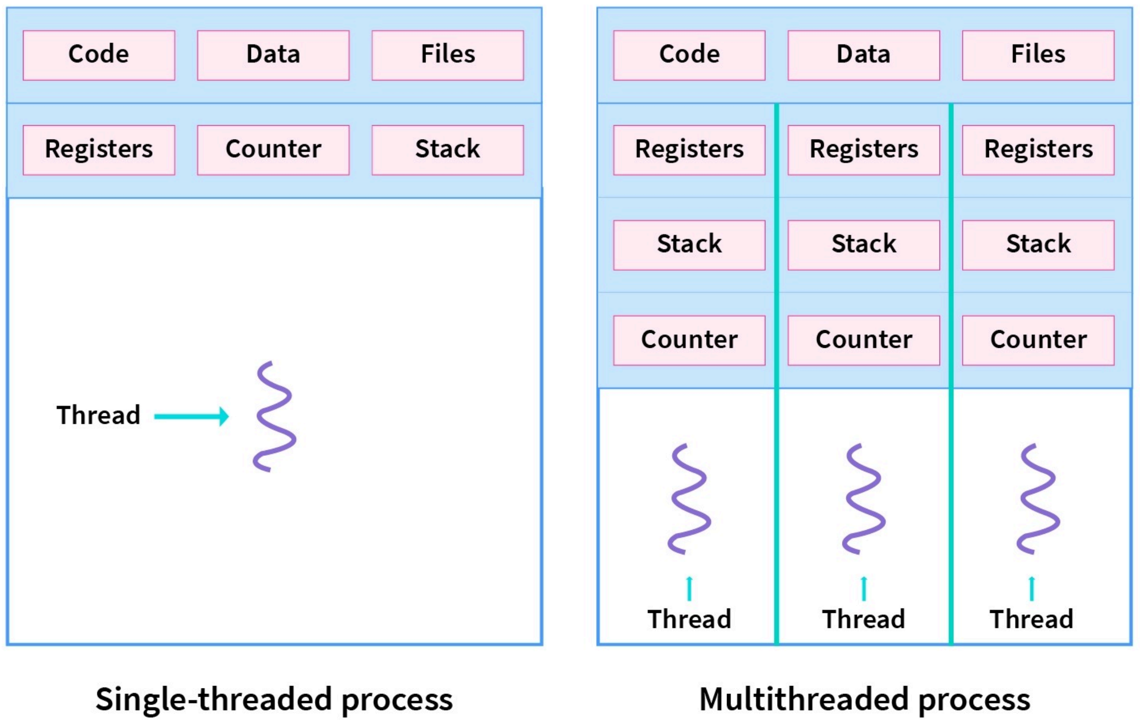
Threads

> A thread is a lightweight process. It is a unit of execution within a process. A process can have multiple threads. Each thread has its own program counter, stack, and registers. Threads share the same address space. This means that all threads in a process can access the same memory. This is different from processes where each process has its own address space.

Often, a process needs to perform multiple tasks at the same time. For example, a web browser needs to download a file and display a web page at the same time. Creating a new process for each task is expensive. This is because creating a new process requires a lot of resources.

Threads are used to solve this problem. Threads are used to perform multiple tasks within a process. This is done by sharing the same address space. This means that all threads in a process can access the same memory. This is different from processes where each process has its own address space.

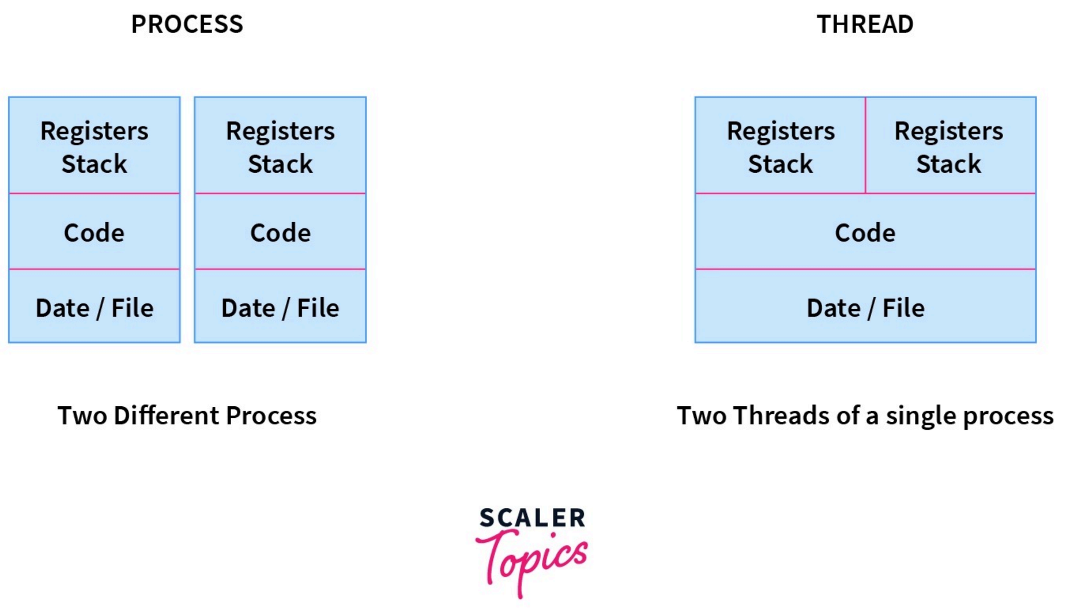
Thread is a sequential flow of tasks within a process. Threads in OS can be of the same or different types. Threads are used to increase the performance of the applications.
Each thread has its own program counter, stack, and set of registers. But the threads of a single process might share the same code and data/file. Threads are also termed as lightweight processes as they share common resources.



SCALER Topics

Thread vs Process

Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can read, write or change another thread’s data)
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.



Using threads in Python

In Python, the threading module provides a way to work with threads. There are two main ways to create a thread:

1. Instantiating the `Thread` class
2. Extending the `Thread` class

Instantiating the `Thread` class

You can create threads using the `Thread` function directly from the `threading` module. This approach is more suitable when the thread's logic is defined in a target function.

```python
from threading import Thread

def print():
    # Code to be executed in the new thread
    print("Thread function is running")

# Creating a thread using the Thread function
my_thread = Thread(target=print)

# Starting the thread
my_thread.start()

# Waiting for the thread to complete (optional)
my_thread.join()
```

- We first created a thread using the `Thread` function. The `Thread` function takes a `target` parameter that is used to specify the function that is executed by the thread.
- Then we started the thread using the `start` method. The `start` method calls the function specified by the `target` parameter.
- The `join` method is used to wait for a thread to complete. The `join` method is a blocking method. It waits until the thread is completed and then returns.

Extending the `Thread` class

You can also create threads by extending the `Thread` class. This approach is more suitable when you have complex logic or require a reusable thread class.

```python
from threading import Thread

class MyThread(Thread):

    def run(self):
        # Code to be executed in the new thread
        print("Thread function is running")

# Creating a thread using the MyThread class
my_thread = MyThread()

# Starting the thread
my_thread.start()

# Waiting for the thread to complete (optional)
my_thread.join()
```

Reading List

- Web Browser architecture