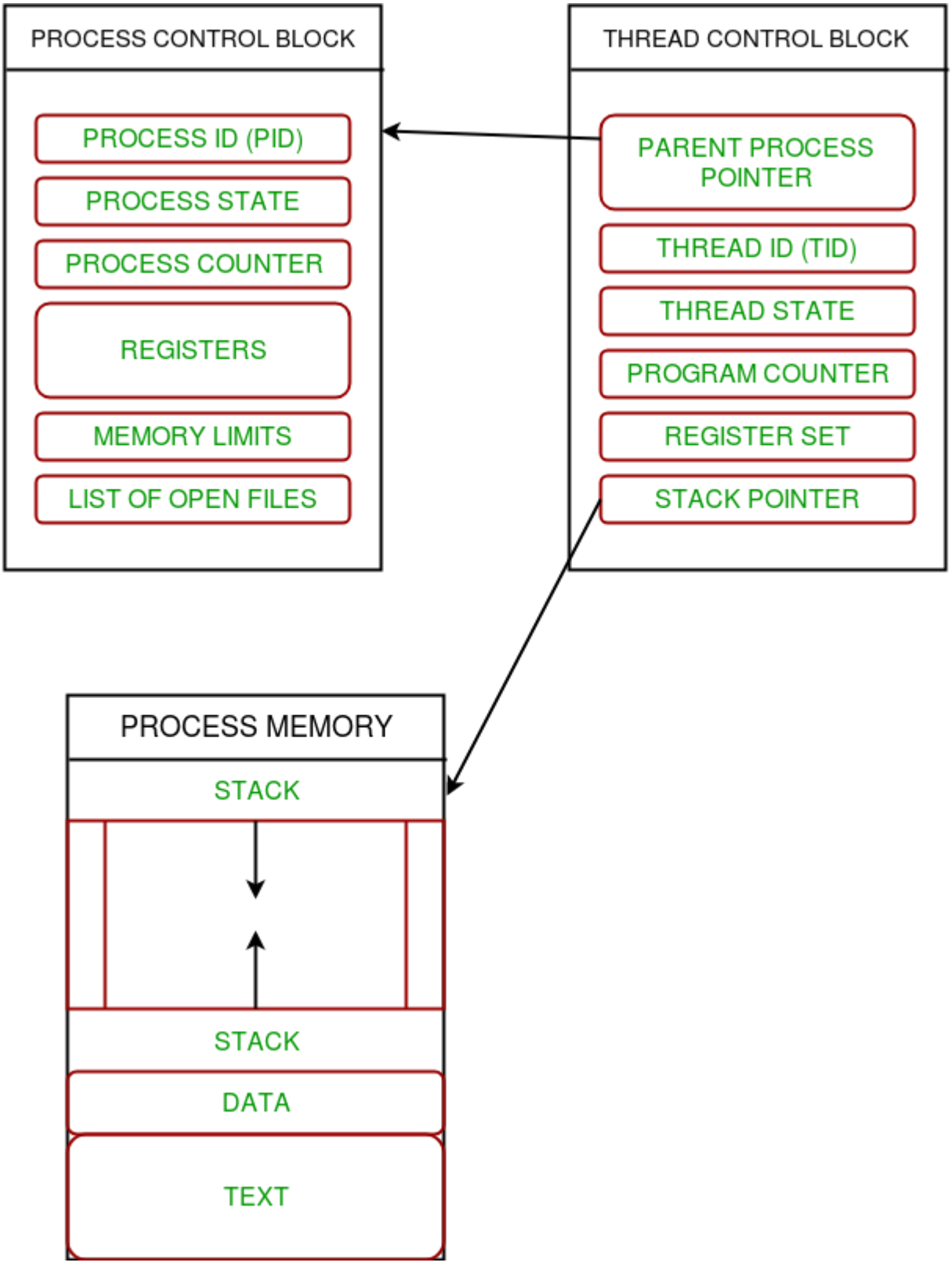
Multithreading in Python -

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System). In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process! A thread contains all this information in a **Thread Control Block (TCB)**

- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter:** a register which stores the address of the instruction currently being executed by thread.
- **Thread state:** can be running, ready, waiting, start or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

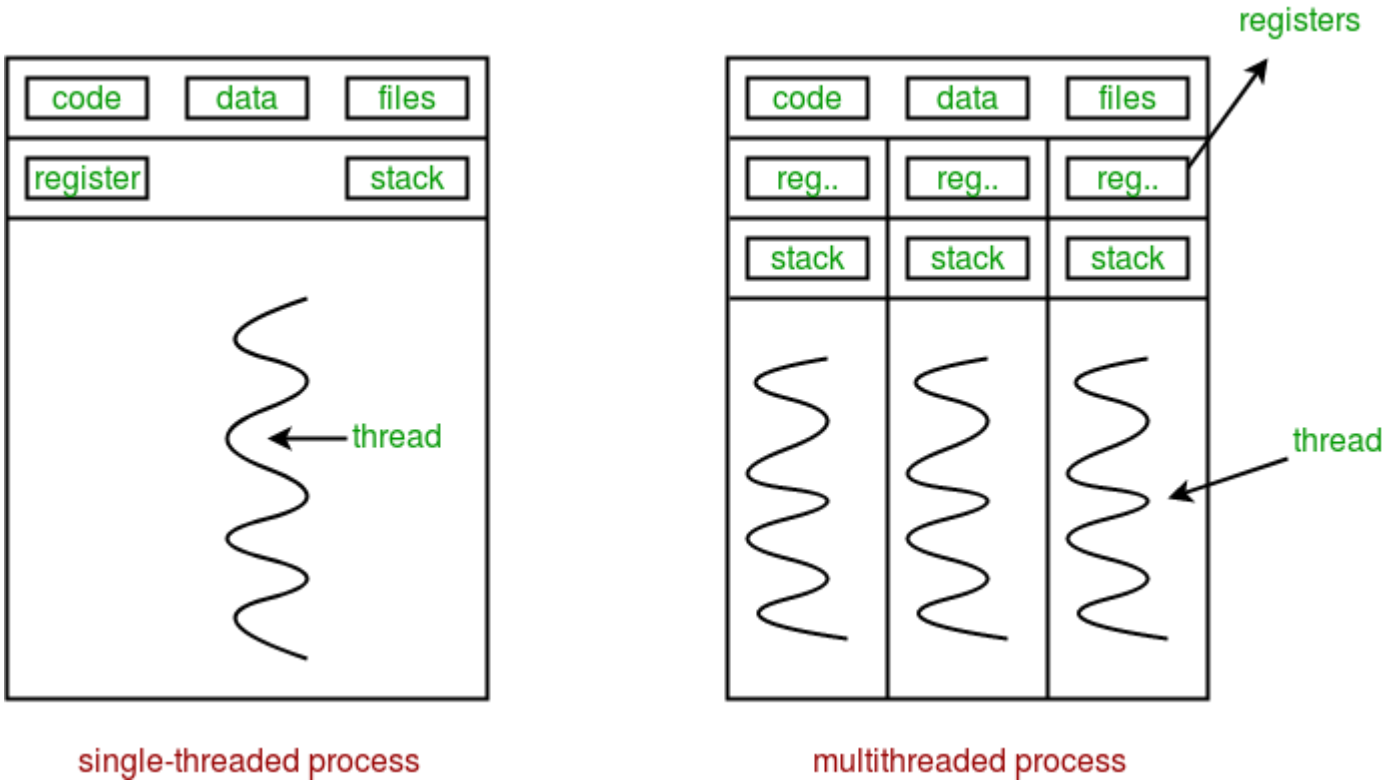
Consider the diagram below to understand the relationship between the process and its thread:



Multi-threading: Multiple threads can exist within one process where:

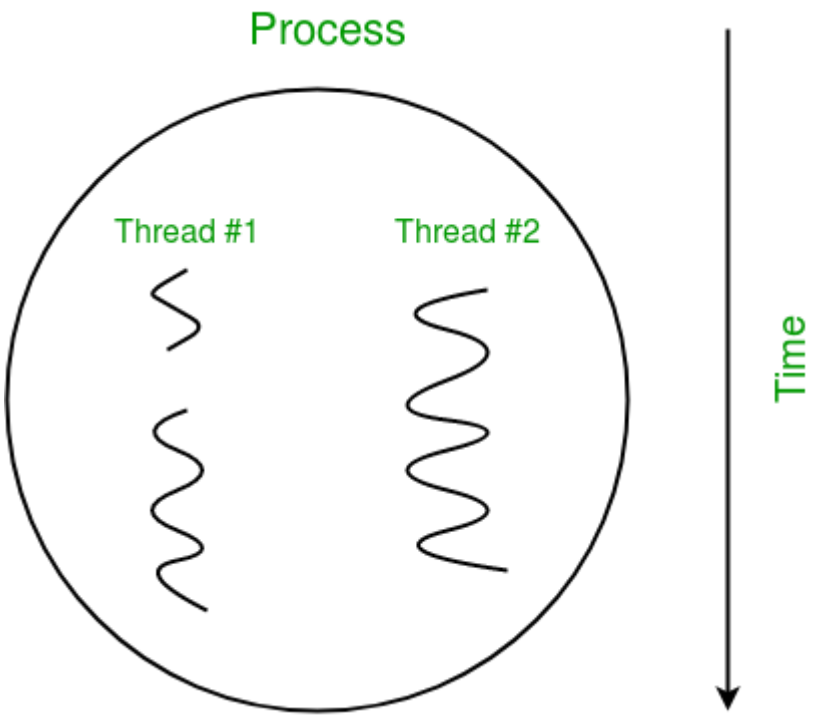
- Each thread contains its own **register set** and **local variables (stored in stack)**.
- All threads of a process share **global variables (stored in heap)** and the **program code**.

Consider the diagram below to understand how multiple threads exist in memory:



single-threaded process multithreaded process

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.
In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed as **context switching**. In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallelly (this is termed as **multitasking**).




```
# Python program to illustrate the concept
# of threading
# importing the threading module
import threading
def print_cube(num):
        print("Cube: {}" .format(num * num * num))
def print_square(num):
        print("Square: {}" .format(num * num))
if __name__ =="__main__":
        t1 = threading.Thread(target=print_square, args=(10,))
        t2 = threading.Thread(target=print_cube, args=(10,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
        print("Done!")
```

```
# Python program to illustrate the concept of threading

import threading

import os

def task1():

print("Task 1 assigned to thread: {}".format(threading.current_thread().name))

print("ID of process running task 1: {}".format(os.getpid()))

def task2():

print("Task 2 assigned to thread: {}".format(threading.current_thread().name))

print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

print("ID of process running main program: {}".format(os.getpid()))

print("Main thread name: {}".format(threading.current_thread().name))

t1 = threading.Thread(target=task1, name='t1')

t2 = threading.Thread(target=task2, name='t2')

t1.start()

t2.start()

t1.join()

t2.join()
```

Using a thread pool:

A thread pool is a collection of threads that are created in advance and can be reused to execute multiple tasks. The concurrent.futures module in Python provides a ThreadPoolExecutor class that makes it easy to create and manage a thread pool.

In this example, we define a function worker that will run in a thread. We create a ThreadPoolExecutor with a maximum of 2 worker threads. We then submit two tasks to the pool using the submit method. The pool manages the execution of the tasks in its worker threads. We use the shutdown method to wait for all tasks to complete before the main thread continues.

. Multithreading can help you make your programs more efficient and responsive.

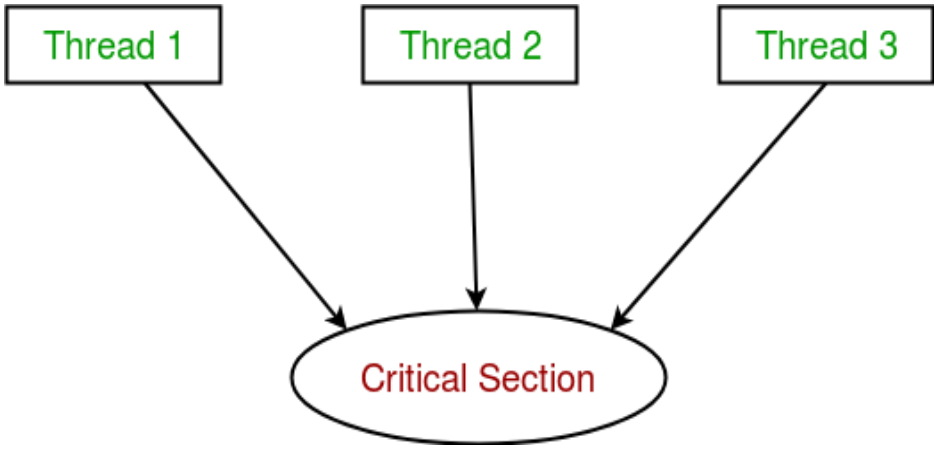
```
import concurrent.futures

def worker():
        print("Worker thread running")
        pool = concurrent.futures.ThreadPoolExecutor(max_workers=2)
        pool.submit(worker)
        pool.submit(worker)
        pool.shutdown(wait=True)
        print("Main thread continuing to run")
```

Synchronization between threads

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as **critical section**. Critical section refers to the parts of the program where the shared resource is accessed.

For example, in the diagram below, 3 threads try to access shared resource or critical section at the same time.



Concurrent accesses to shared resource can lead to **race condition**.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

```
import threading

x = 0

def increment():
    global x
     x += 1

def thread_task():
                        for _ in range(100000):
                                    increment()
```

```
def main_task():
                                global x
                                x = 0
                                t1 = threading.Thread(target=thread_task)
                                t2 = threading.Thread(target=thread_task)
                                t1.start()
                                t2.start()
                                t1.join()
                                t2.join()


if __name__ == "__main__":
                                for i in range(10):
                                    main_task()
                                    print("Iteration {0}: x = {1}".format(i,x))
```

Using Locks

threading module provides a **Lock** class to deal with the race conditions. Lock is implemented using a **Semaphore** object provided by the Operating System.
A semaphore is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Lock class provides following methods:

- **acquire([blocking]) :** To acquire a lock. A lock can be blocking or non-blocking.
    - When invoked with the blocking argument set to **True** (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return **True**.
    - When invoked with the blocking argument set to **False**, thread execution is not blocked. If lock is unlocked, then set it to locked and return **True** else return **False** immediately.
- **release() :** To release a lock.
    - When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.
    - If lock is already unlocked, a **ThreadError** is raised.

```
import threading

x = 0
```

```
def increment():
                        global x
                        x += 1
def thread_task(lock):
                        for _ in range(100000):
                         lock.acquire()
                         increment()
                         lock.release()
def main_task():
                        global x
                        x = 0
                        lock = threading.Lock()
                        t1 = threading.Thread(target=thread_task, args=(lock,))
                        t2 = threading.Thread(target=thread_task, args=(lock,))
                        t1.start()
                        t2.start()
                        t1.join()
                        t2.join()
if __name__ == "__main__":
                        for i in range(10):
                         main_task()
                         print("Iteration {0}: x = {1}".format(i,x))
```