Module 2.2 Thread

Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

Outline

- Thread
- Thread Type
- Thread Model

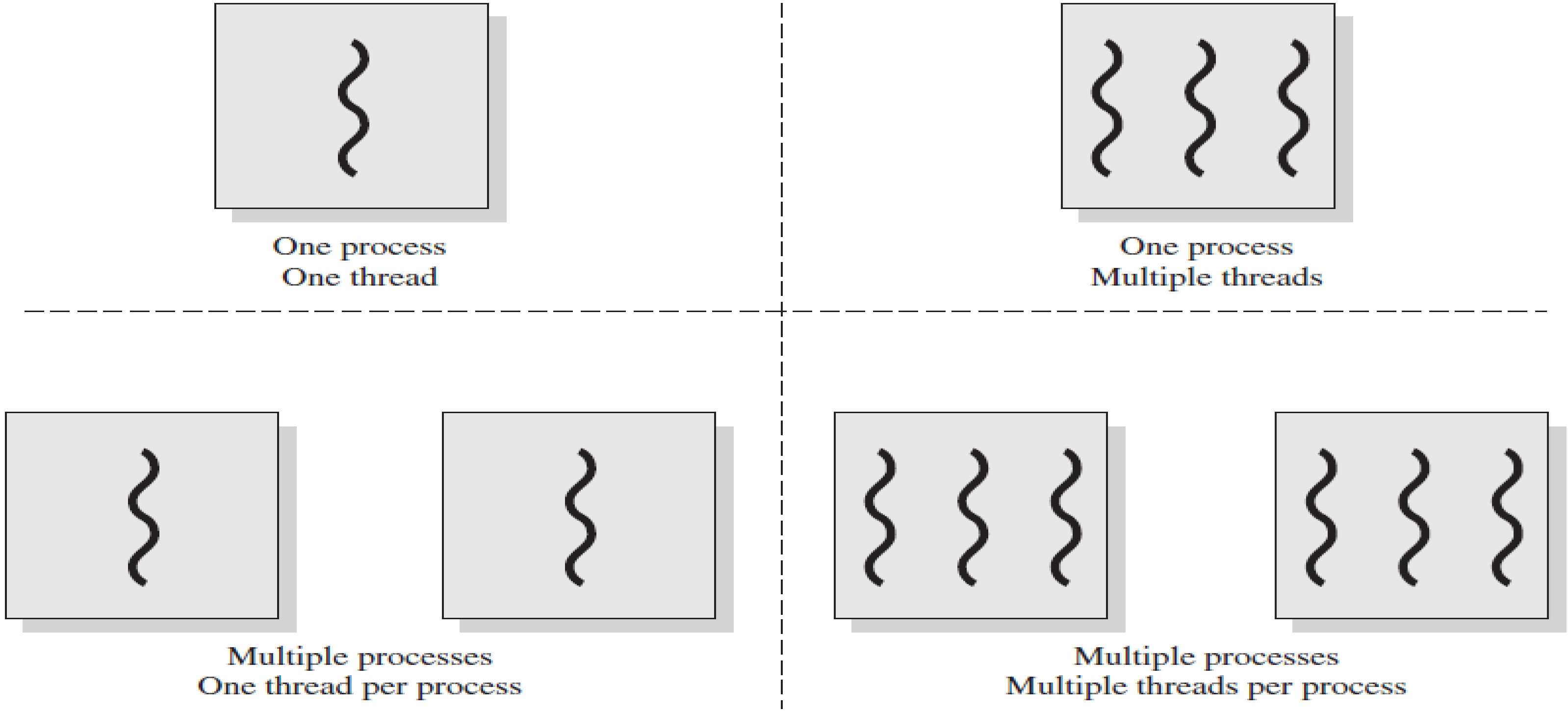
Thread

- Supports Parallelism with multiple threads of execution at a time
- A thread executes sequentially and is interrupt-able so that the processor can turn to another thread
- Does not need entire process context to execute so considered as lightweight.
- Includes the program counter and stack pointer) and its own data area for a stack
- Supports multiple parallel executions

  e.g. Notifications in background while you are using the app
- The idea is to **achieve parallelism by dividing a process into multiple threads.**

Thread

- *Multithreading* refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.



Thread

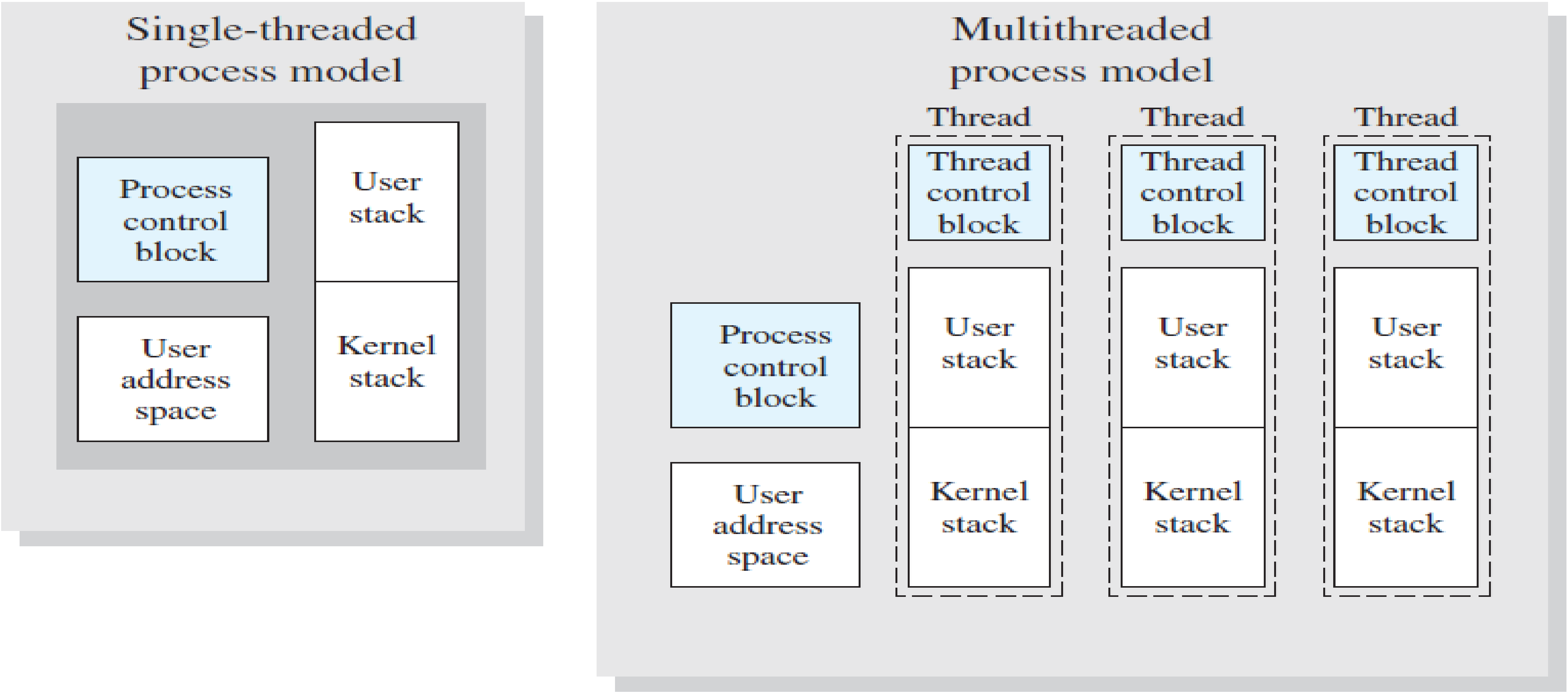
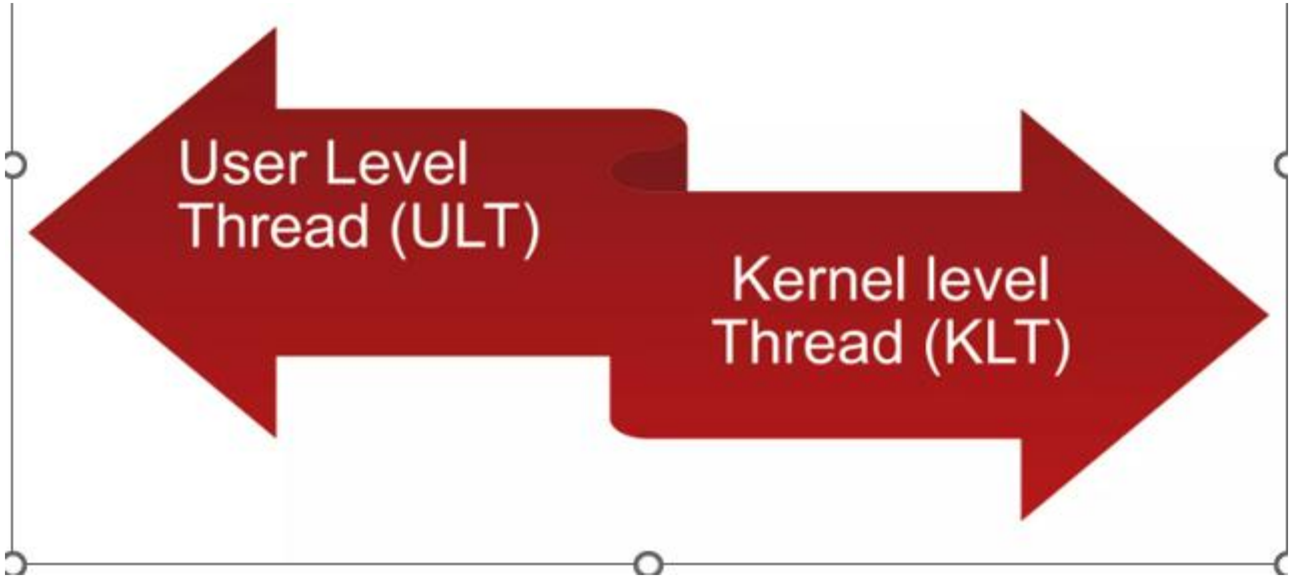


Figure 4.2 **Single Threaded and Multithreaded Process Models**

Benefits

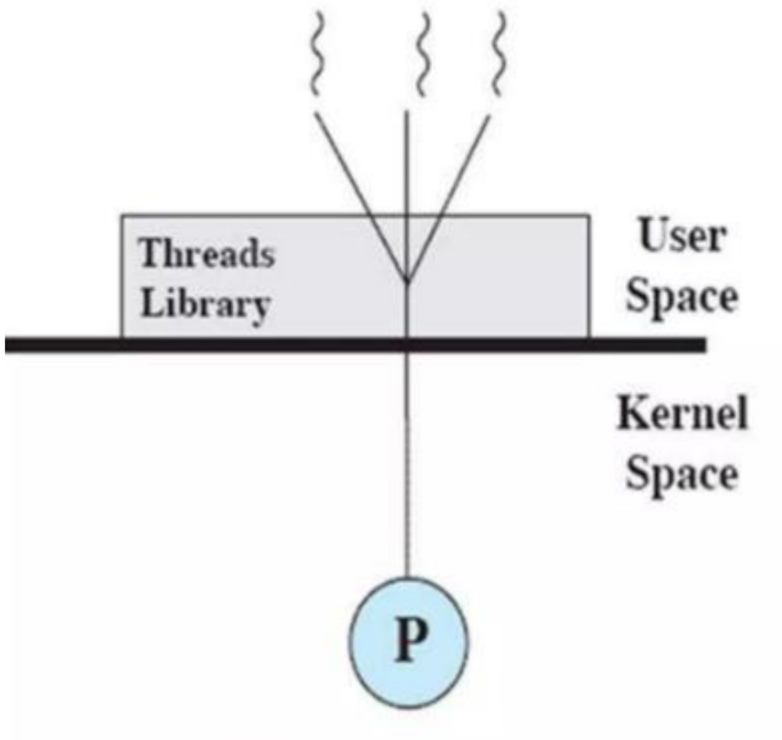
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Types of Thread



User Level Thread (ULT)

- User-level threads are implemented and managed by the user and the kernel is not aware of it.
- User-level threads are implemented using user-level libraries and the OS does not recognize these threads.
- User-level thread is faster to create and manage compared to kernel-level thread.
- Context switching in user-level threads is faster.
- If one user-level thread performs a blocking operation then the entire process gets blocked. Eg: POSIX threads, Java threads, etc.



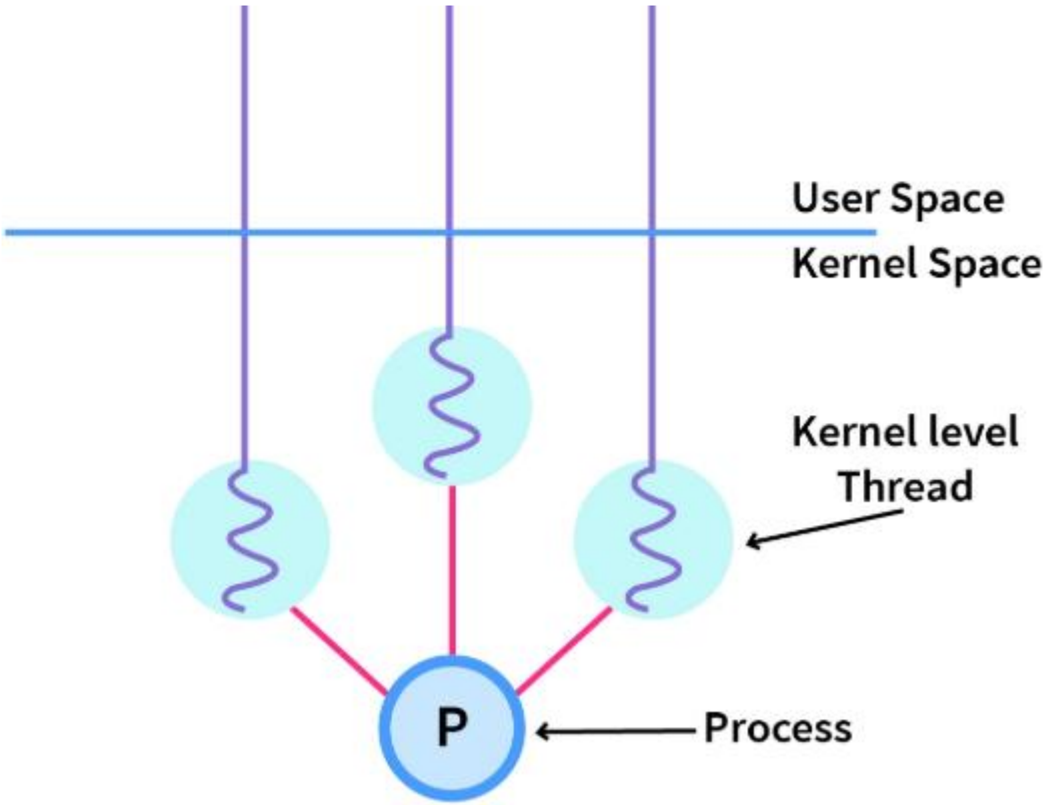
(a) Pure user-level

User Level Thread

- Advantage
  - Greater flexibility and control: User-level threads provide more control over thread management, as the thread library is part of the application. This allows for more customization and control over thread scheduling.
  - Portability: User-level threads can be more easily ported to different operating systems, as the thread library is part of the application.
- Disadvantage
  - Lower performance: User-level threads rely on the application to manage thread scheduling, which can be less efficient than kernel-level thread scheduling. This can result in lower performance for multithreaded applications.
  - Limited parallelism: User-level threads are limited to a single processor, as the application has no control over thread scheduling on other processors.

Kernel Level Thread

- Kernel level threads are implemented and managed by the OS.
- Kernel level threads are implemented using system calls and Kernel level threads are recognized by the OS.
- Kernel-level threads are slower to create and manage compared to user-level threads.
- Context switching in a kernel-level thread is slower.
- Even if one kernel-level thread performs a blocking operation, it does not affect other threads. Eg: Window Solaris.



Kernel Level Thread

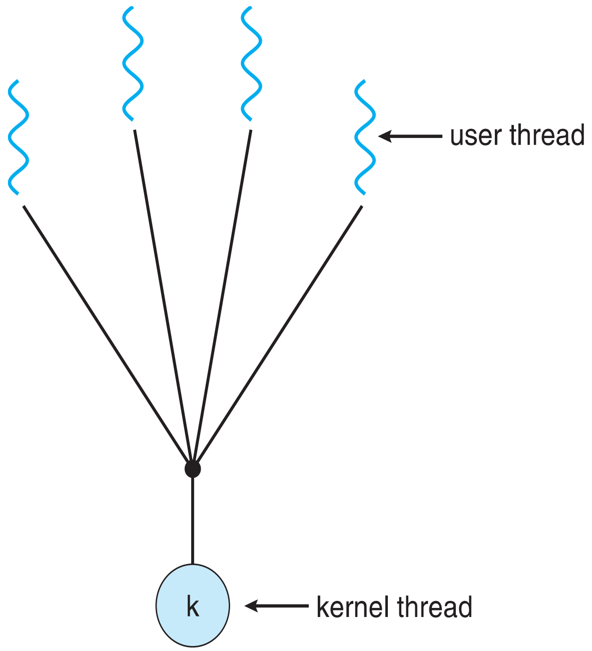
- Advantage
  - Better performance: Kernel-level threads are managed by the operating system, which can schedule threads more efficiently. This can result in better performance for multithreaded applications.
  - Greater parallelism: Kernel-level threads can be scheduled on multiple processors, which allows for greater parallelism and better use of available resources.
- Disadvantage
  - Less flexibility and control: Kernel-level threads are managed by the operating system, which provides less flexibility and control over thread management compared to user-level threads.
  - Less portability: Kernel-level threads are more tightly coupled to the operating system, which can make them less portable to different operating systems.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

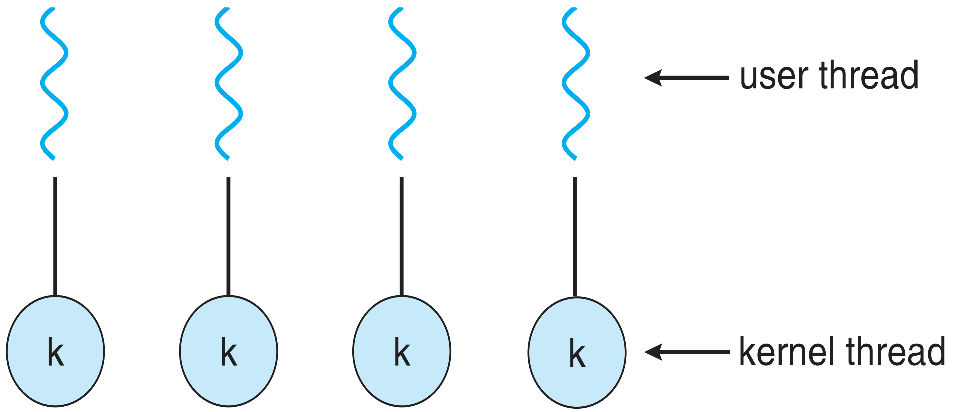
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



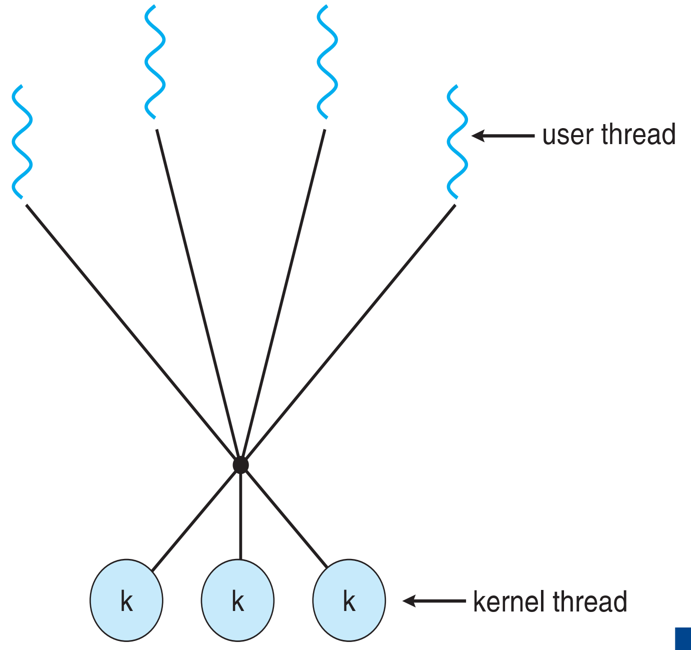
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Thread

- Thread libraries provide programmers with API for the creation and management of threads.
- Three types of Thread
  - **POSIX Pitheads** may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - **Win32 threads** are provided as a kernel-level library on Windows systems.
  - **Java threads**: Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system.

Thread

- Create
- Join
- Terminate

Create, Start and Join

- A thread can be created using the **Thread class** provided by the threading module. Using this class, you can create an instance of the Thread and then start it using the **.start()** method.

Create & Start

```python
import threading

# Creating Target Function
def num_gen(num):
    for n in range(num):
        print("Thread: ", n)

# Main Code of the Program
if __name__ == "__main__":
    print("Statement: Creating and Starting a Thread.")
    thread = threading.Thread(target=num_gen, args=(3,))
    thread.start()
    print("Statement: Thread Execution Finished.")
```

- 1st execution

```
Statement: Creating and Starting a Thread.
Thread:  0
Statement: Thread Execution Finished.
Thread:  1
Thread:  2
```

- 2nd execution

```
Statement: Creating and Starting a Thread.
Thread:  0
Thread:  1
Statement: Thread Execution Finished.
Thread:  2
```

Join() Method

- The join() method is used in that situation, it doesn't let e**xecute the code further until the current thread terminates**.

```python
import threading

# Creating Target Function
def num_gen(num):
    for n in range(num):
        print("Thread: ", n)

# Main Code of the Program
if __name__ == "__main__":
    print("Statement: Creating and Starting a Thread.")
    thread = threading.Thread(target=num_gen, args=(3,))
    thread.start()
    thread.join()
    print("Statement: Thread Execution Finished.")
```

```
Statement: Creating and Starting a Thread.
Thread:  0
Thread:  1
Thread:  2
Statement: Thread Execution Finished.
```

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.

Question ?