Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual
CS332
Parallel Computation

Preparing the scientific material

Prof. Mohamed Arafa
T.A. Hager

Contents

Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Parallel Computation

Lab - 1
Introduction to Parallel Computation

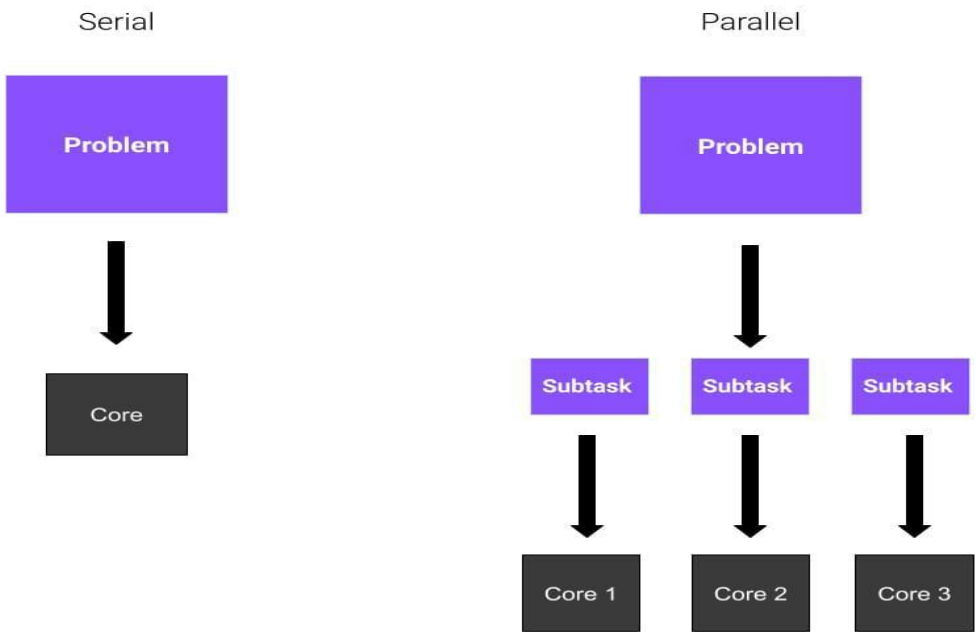
What is PARALLEL PROGRAMMING?

Parallel programming is a programming technique that enables a computer to execute multiple tasks simultaneously. It involves dividing a problem into smaller subproblems and distributing those subproblems across multiple processors or cores, allowing computations to occur in parallel rather than sequentially. This approach improves the efficiency and performance of computing systems, particularly for large-scale or computationally intensive tasks.

WHY PARALLEL PROGRAMMING?

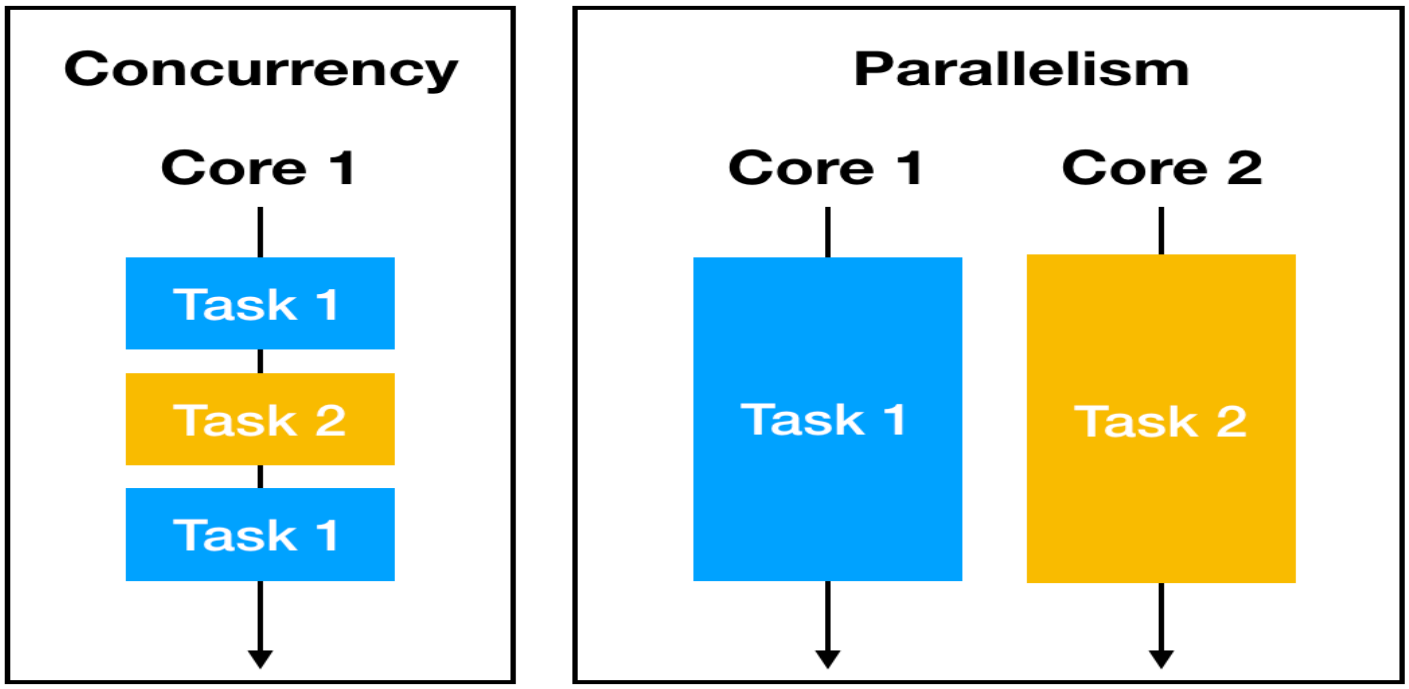
Traditional, or sequential programming, executes instructions one after the other, which can be slow for complex tasks. Parallel programming, on the other hand, leverages modern hardware architectures that support multiple cores or processors to complete tasks more quickly and efficiently. This is particularly useful in fields like:

- **–Scientific computing (simulations, data analysis)**
- **–Machine learning and AI**
- **–Image and video processing**
- **–Gaming and real-time systems**
- **–Large-scale data processing (e.g., in databases or cloud computing)**



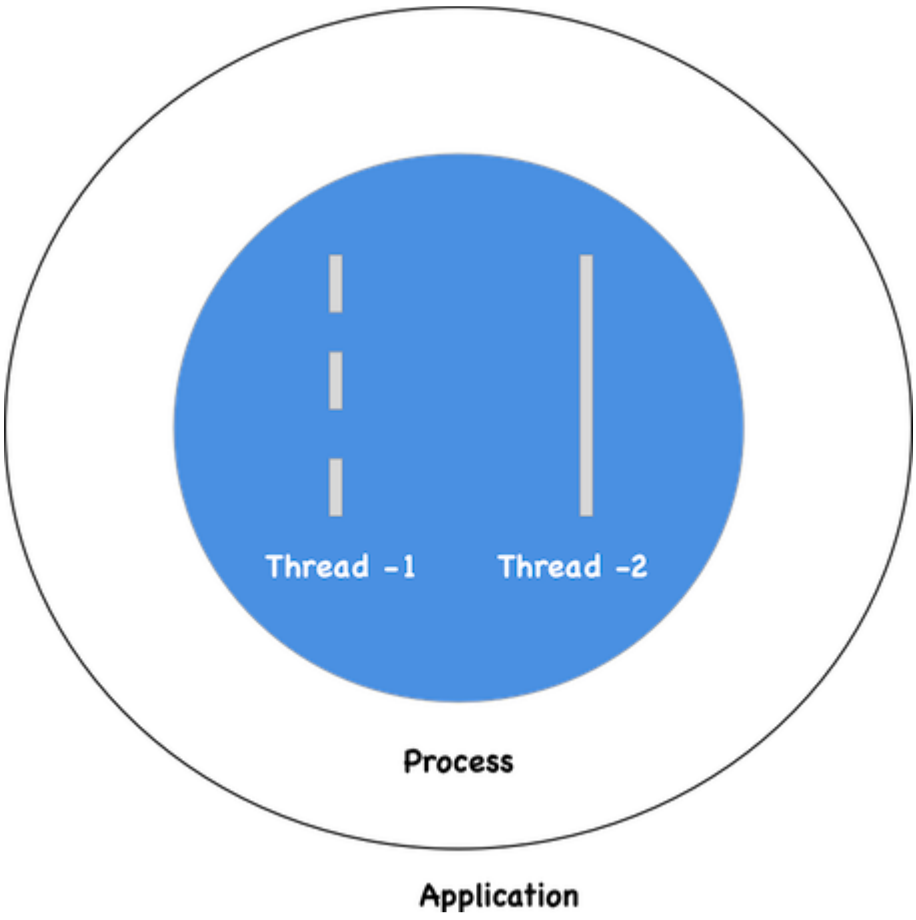
CONCURRENCY VS. PARALLELISM

- **Concurrency: Refers to the ability to handle multiple tasks at the same time, but not necessarily simultaneously. It involves task-switching and overlaps in execution.**
- **Parallelism: Refers to executing multiple tasks at exactly the same time, typically on different processors or cores**



THREADS & PROCESSES

- **Threads are a sequence of execution of code which can be executed independently of one another. It is the smallest unit of tasks that can be executed by an OS. A program can be single threaded or multi-threaded.**
- **A process is an instance of a running program. A program can have multiple processes. A process usually starts with a single thread i.ea primary thread but later down the line of execution it can create multiple threads.**



SYNCHRONOUS AND ASYNCHRONOUS

SYNCHRONOUS

•Imagine you were given to write two letters one to your mom and another to your best friend. You can not at the same time write two letters unless you are a pro ambidextrous.

•In a synchronous programming model, tasks are executed one after another. Each task waits for any previous task to complete and then gets executed.

ASYNCHRONOUS

•Imagine you were given to make a sandwich and wash your clothes in a washing machine. You could put your clothes in the washing machine and without waiting for it to be done, you could go and make the sandwich. Here you performed these two tasks asynchronously.

•In an asynchronous programming model, when one task gets executed, you could switch to a different task without waiting for the previous to get completed.

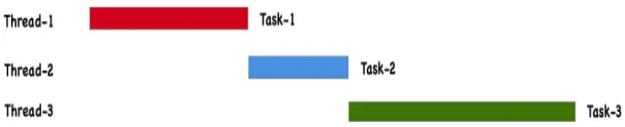
SYNCHRONOUS AND ASYNCHRONOUS IN A SINGLE AND MULTI-THREADED ENVIRONMENT

SYNCHRONOUS

Single Threaded:

Each task gets executed one after another. Each task waits for its previous task to get executed.

Multi-Threaded:



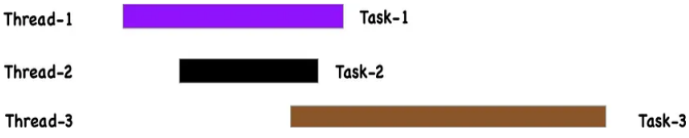
Tasks get executed in different threads but wait for any other executing tasks on any other thread.

ASYNCHRONOUS

Single Threaded:

Tasks start executing without waiting for a different task to finish. At a given time a single task gets executed.

Multi-Threaded:



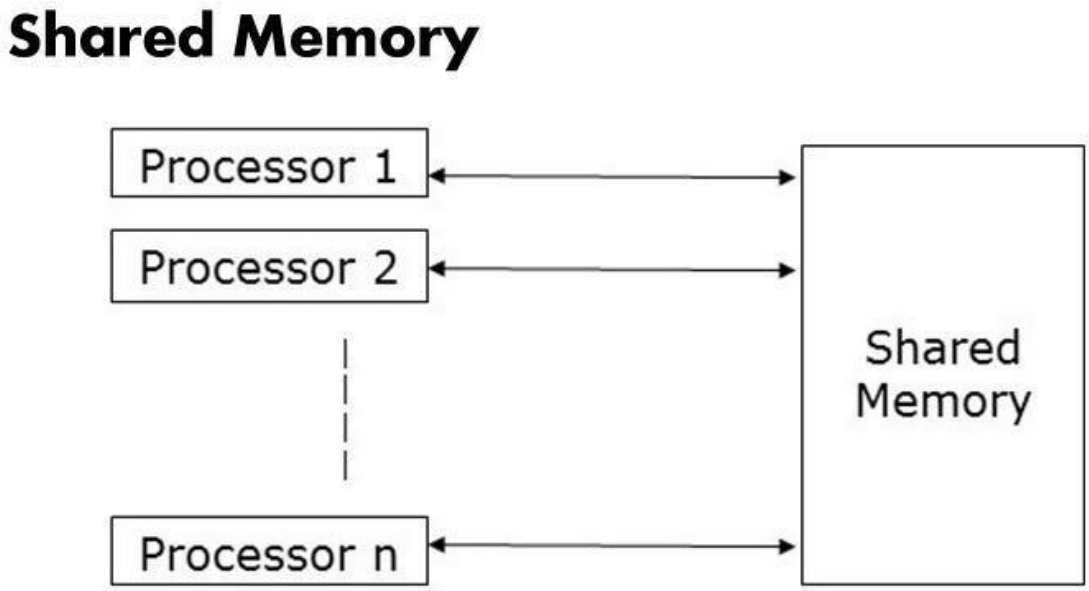
Tasks get executed in different threads without waiting for any tasks and independently finish off their executions.

DATA AND TASK PARALLELISM

Data Parallelism : Involves distributing the same task across multiple data sets, where each processor performs the same operation on different pieces of data. **–Example: Applying the same filter to different parts of an image.**	**Task Parallelism : Involves distributing different tasks or processes across multiple processors.** **Each task may perform different operations, possibly on the same or different data.** **–Example: One processor handles image processing while another handles audio processing in a video editing application.**

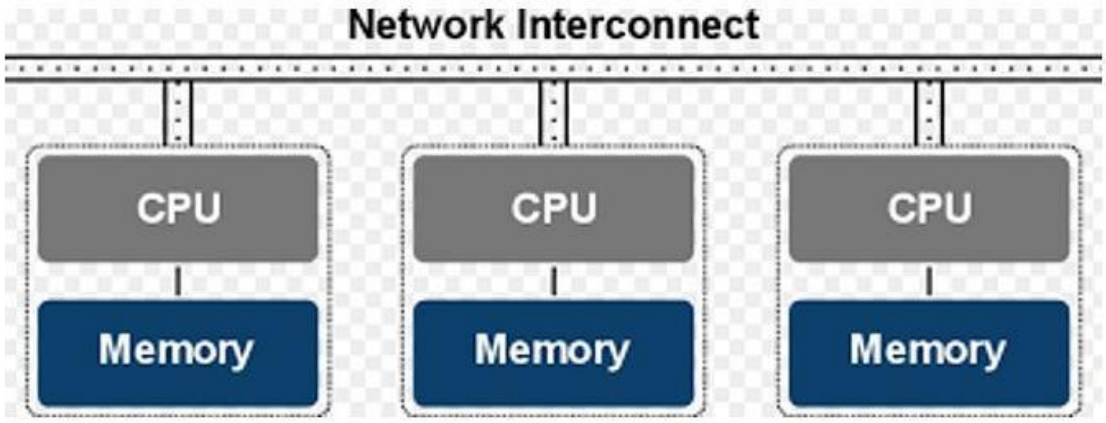
SHARED MEMORY VS. DISTRIBUTED MEMORY

Shared Memory: Multiple processors share a single memory space, and tasks communicate by reading/writing to this shared memory (e.g., on multi-core systems).



SHARED MEMORY VS. DISTRIBUTED MEMORY

Distributed Memory: Each processor has its own private memory, and communication between tasks occurs via message passing (e.g., in cluster computing).



Distributed Memory

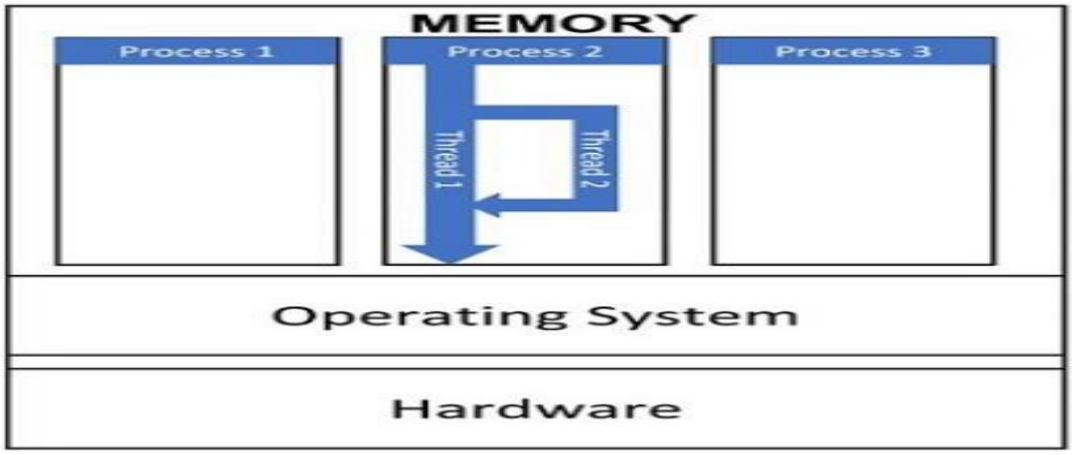
KEY CONCEPTS IN PARALLEL PROGRAMMING

Task Decomposition : The process of dividing a computational problem into smaller tasks or subtasks, which can be executed in parallel. **Synchronization : When multiple tasks access shared resources, it is important to synchronize their access to avoid conflicts or data corruption.Methods like locks, semaphores, and barriers are used to ensure that tasks access shared resources in a controlled manner.**	**Load Balancing : In parallel computing, it's important to ensure that the computational workload is evenly distributed across all processors or cores. Otherwise, some processors may finish their tasks while others are still working, leading to inefficiencies.**

Lab - 2
Multithreading in python

MULTITHREADING IN PYTHON

- **In Python, multithreading allows you to run multiple threads concurrently within a single process, which is also known as thread-based parallelism. This means a program can perform multiple tasks at the same time, enhancing its efficiency and responsiveness.**
- **Multithreading in Python is especially useful for multiple I/O-bound operations, rather than for tasks that require heavy computation.**
- **Generally, a computer program sequentially executes the instructions, from start to the end. Whereas Multithreading divides the main task into more than one sub-task and executes them in an overlapping manner.**
- **However, Python's Global Interpreter Lock (GIL) prevents multiple threads from executing Python bytecodes simultaneously on a single core. So, while multithreading can help with I/O-bound tasks, it doesn't provide significant speed-ups for CPU-bound tasks, where multiprocessing would be more effective.**



THREAD HANDLING MODULES IN PYTHON

Python provides several modules for handling threads and parallel execution.

–threading: High-level module for managing threads, with synchronization tools and easier handling of concurrency.

–thread: Lower-level module for working with threads, providing basic thread operations but no synchronization or advanced management features.

–concurrent.futures: High-level API for managing thread and process pools, making it easier to work with parallelism in Python.

–multiprocessing: For CPU-bound tasks requiring parallelism without the limitations of the GIL, though this involves processes, not threads.

THREAD LIFE CYCLE IN PYTHON

- **In Python, the lifecycle of a thread refers to the various states a thread goes through during its execution. These states are part of the threading module, and understanding the thread lifecycle is important when dealing with concurrency.**
- **Key Methods in Thread Lifecycle:**

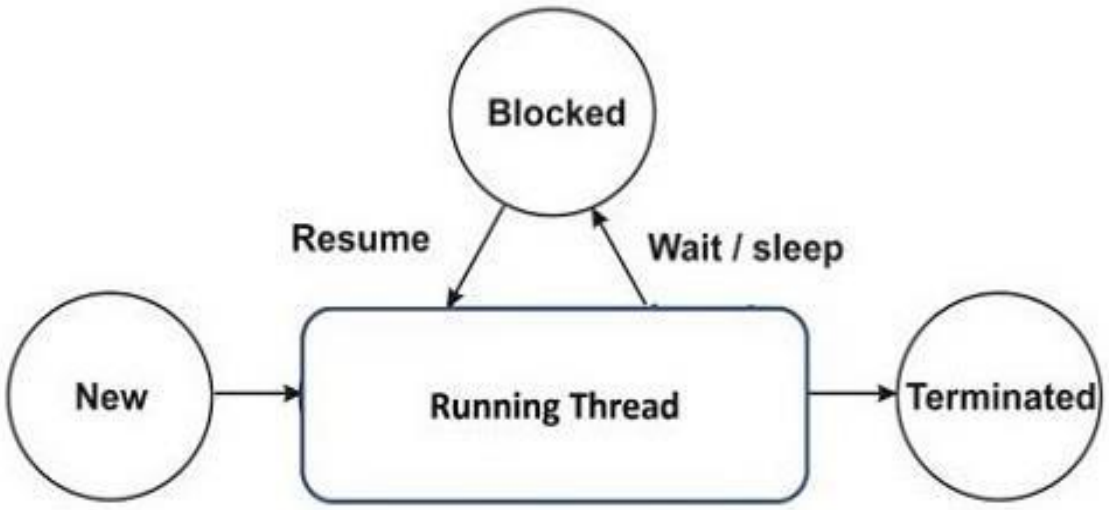
  –start(): Moves the thread from the New state to the Runnable state.

  –run(): Represents the thread's task; moves the thread to the Running state.

  –join(): Causes the calling thread to wait for another thread to complete, effectively blocking it until the other thread finishes.

  –sleep(): Pauses the thread for a certain period, temporarily moving it to the Blocked state.

  –is_alive(): Checks whether the thread is still running or has finished.



THREAD LIFE CYCLE IN PYTHON (NEW)

- **import threading**
- **def task():**

–print("Thread task")

- **thread = threading.Thread(target=task) # Thread is created but not started**

THREAD LIFE CYCLE IN PYTHON (RUNNABLE)

- **thread.start() # Thread is now in the Runnable state**

THREAD LIFE CYCLE IN PYTHON (RUNNING)

- **def task():**

–print("Thread running")

- **thread = threading.Thread(target=task)**
- **thread.start() # Thread starts running here**

THREAD LIFE CYCLE IN PYTHON (WAITING/SLEEPING)

•import time

•def task():

–print("Thread sleeping")

–time.sleep(2) # Thread is blocked for 2 seconds

–print("Thread awake")

•thread = threading.Thread(target=task)

•thread.start()

THREAD LIFE CYCLE IN PYTHON (DEAD)

•def task():

–print("Thread has completed its work")

•thread = threading.Thread(target=task)

•thread.start()

•thread.join() # Waits for the thread to finish execution

•print("Thread has terminated")

Lab - 3
Creating a thread in python

CREATING A THREAD IN PYTHON

•Threads in python are an entity within a process that can be scheduled for execution. In simpler words, a thread is a computation process that is to be performed by a computer. It is a sequence of such instructions within a program that can be executed independently of other codes.

•In Python, there are two ways to create a new Thread. We will also be making use of the threading module in Python. Below is a detailed list of those processes:

–Creating Threads with Functions

–Creating Threads by Extending the Thread Class

CREATING THREADS BY EXTENDING THE THREAD CLASS

•You can create threads by using the Thread class from the threading module. In this approach, you can create a thread by simply passing a function to the Thread object. Here are the steps to start a new thread

1.Define a function that you want the thread to execute.

2.Create a Thread object using the Thread class, passing the target function and its arguments.

3.Call the start method on the Thread object to begin execution.

4.Optionally, call the join method to wait for the thread to complete before proceeding.

CREATING THREADS BY EXTENDING THE THREAD CLASS

```
•from threading import Thread
•from time import sleep

•# function to create threads
•def threaded_function(arg):
        •for i in range(arg):
        •print("running")
        •# wait 1 sec in between each thread
        •sleep(1)

•if __name__ == "__main__"
•thread = Thread(target = threaded_function, args= (10, ))
•thread.start()
•thread.join()
•print("thread finished...exiting")
```

CREATING THREADS BY EXTENDING THE THREAD CLASS

•Another approach to creating a thread is by extending the Thread class. This approach involves defining a new class that inherits from Thread and overriding its __init__ and run methods. Here are the steps to start a new thread −

1.Define a new subclass of the Thread class.
2.Override the __init__ method to add additional arguments.
3.Override the run method to implement the thread's behaviour.

•# import the threading module
•import threading

•class thread(threading.Thread):

•def__init__(self, thread_name, thread_ID):
    •threading.Thread.__init__(self)
    •self.thread_name= thread_name
    •self.thread_ID= thread_ID

•# helper function to execute the threads
•def run(self):
    •print(str(self.thread_name) +" "+ str(self.thread_ID));

•thread1 = thread(“Thread 1", 1000)
•thread2 = thread(“Thread 2 ", 2000)
•thread1.start()
•thread2.start()
•print("Exit")

STARTING A THREAD IN PYTHON

•In Python, starting a thread involves using the start() method provided by the Thread class in the threading module. This method initiates the thread's activity and automatically calls its run() method in a separate thread of execution. Meaning that, when you call start() on each thread object (for example., thread1, thread2, thread3) to initiate their execution.

•Python to launch separate threads that concurrently execute the run() method defined in each Thread instance. And the main thread continues its execution after starting the child threads.

•The start() method is fundamental for beginning the execution of a thread. It sets up the thread's environment and schedules it to run. Importantly, it should only be called once per Thread object. If this method is called more than once on the same Thread object, it will raise a RuntimeError.

•from threading import Thread
•from time import sleep
•defmy_function(arg):
•for iin range(arg):
•print("child Thread running", i)
•sleep(0.5)
•thread = Thread(target = my_function, args= (10,))
•thread.start()
•thread.join()
•print("thread finished...exiting")

STARTING A THREAD IN PYTHON

```
•import threading
•import time

•class MyThread(threading.Thread):
•def__init__(self, threadID, name, counter):
        •threading.Thread.__init__(self)
        •self.threadID= threadID
        •self.name = name
        •self.counter = counter

•def run(self):
        •print("Starting " + self.name)
        •print_time(self.name, self.counter)
        •print("Exiting " + self.name)

•def print_time(threadName, counter):
        •while counter:
        •time.sleep(1)
        •print("%s: %s" % (threadName, time.ctime(time.time())))
        •counter -= 1

•# Create new threads
•thread1 = MyThread(1, "Thread-1", 1)
•thread2 = MyThread(2, "Thread-2", 2)
•thread3 = MyThread(3, "Thread-3", 3)
•# Start new Threads
•thread1.start()
•thread3.start()
•print("Exiting Main Thread")
```

Lab - 4
Python -Thread Scheduling

JOINING THE THREADS

•In Python, joining the threads means using the join() method to wait for one thread to finish before moving on to others.
This is useful in multithreaded programming to make sure some threads are completed before starting or continuing with other threads.
By using the join() method, you can make sure that one thread has finished running before another thread or the main program continues.

•To join the threads in Python, you can use the Thread.join() method from the threading module.
Which generally is used to block the calling thread until the thread on which join() was called terminates.
The termination may be either normal, because of an unhandled exception −or until the optional timeout occurs.
You can call join() multiple times. However, if you try to join the current thread or attempts to join a thread before starting it with the start() method, will raise the RuntimeErrorexception.

JOINING THE THREADS

```
•from threading import Thread
•from time import sleep
•def my_function_1(arg):
        •for iin range(arg):
        •print("Child Thread 1 running", i)
        •sleep(0.5)

•def my_function_2(arg):
        •for iin range(arg):
        •print("Child Thread

•# Create thread objects
•thread1 = Thread(target=my_function_1, args=(5,))
•thread2 = Thread(target=my_function_2, args=(3,))

•# Start the first thread and wait for it to complete
•thread1.start()
•thread1.join()

•# Start the second thread and wait for it to complete
•thread2.start()
•thread2.join()
•print("Main thread finished...exiting")
```

JOINING THE THREADS

```
from threading import Thread
from time import sleep

def my_function_1(arg):
    for iin range(arg):
    print("Child Thread 1 running", i)
    sleep(0.5)

def my_function_2(arg):
    for iin range(arg):
    print("Child Thread 2 running", i)
    sleep(0.1)

# Create thread objects
thread1 = Thread(target=my_function_1, args=(5,))
thread2 = Thread(target=my_function_2, args=(3,))

# Start the first thread and wait for 0.2 seconds
thread1.start()
thread1.join(timeout=0.2)

# Start the second thread and wait for it to complete
thread2.start()
thread2.join()
print("Main thread finished...exiting")
```

NAMING THE THREADS

•In Python, naming a thread involves assigning a string as an identifier to the thread object. Thread names in Python are primarily used for identification purposes only and do not affect the thread's behavior or semantics. Multiple threads can share the same name, and names can be specified during the thread's initialization or changed dynamically.

•When you create a thread using threading.Thread() class, you can specify its name using the name parameter. If not provided, Python assigns a default name like the following pattern "Thread-N", where N is a small decimal number. Alternatively, if you specify a target function, the default name format becomes "Thread-N (target_function_name)".

•from threading import Thread
•import threading
•from time import sleep

•def my_function_1(arg):
    •print("This tread name is", threading.current_thread().name)

•# Create thread objects
•thread1 = Thread(target=my_function_1, name='My_thread', args=(2,))
•thread2 = Thread(target=my_function_1, args=(3,))

•print("This tread name is", threading.current_thread().name)

•# Start the first thread and wait for 0.2 seconds
•thread1.start()
•thread1.join()

•# Start the second thread and wait for it to complete
•thread2.start()
•thread2.join()

NAMING THE THREADS

```
•import threading

•def addition_of_numbers(x, y):
        •print("This Thread name is :", threading.current_thread().name)
        •result = x + y

•def cube_number(i):
        •result = i** 3
        •print("This Thread name is :", threading.current_thread().name)

•def basic_function():
        •print("This Thread name is :", threading.current_thread().name)

•# Create threads with custom names
•t1 = threading.Thread(target=addition_of_numbers, name='My_thread', args=(2, 4))
•t3 = threading.Thread(target= basic_function)

•# Start and join threads
•t1.start()
•t1.join()

•t3.name = 'custom_name' # Assigning name after thread creation
•t3.start()
•t3.join()

•print(threading.current_thread().name) # Print main thread's name
```

THREAD SCHEDULING

•Thread scheduling in Python is a process of deciding which thread runs at any given time. In a multi-threaded program, multiple threads are executed independently, allowing for parallel execution of tasks. However, Python does not have built-in support for controlling thread priorities or scheduling policies directly. Instead, it relies on the operating system's thread scheduler.

•Python threads are mapped to native threads of the host operating system, such as POSIX threads (pthreads) on Unix-like systems or Windows threads. The operating system's scheduler manages the execution of these threads, including context switching, thread priorities, and scheduling policies. Python provides basic thread scheduling capabilities through the threading.Timerclass and the schedmodule.

SCHEDULING THREADS USING THE TIMER CLASS

•The Timer class of the Python threading module allows you to schedule a function to be called after a certain amount of time. This class is a subclass of Thread and serves as an example of creating custom threads.

•You start a timer by calling itsstart()method, similar to threads. If needed, you can stop the timer before it begins by using thecancel()method. Note that the actual delay before the action is executed might not match the exact interval specified.

SCHEDULING THREADS USING THE TIMER CLASS

```
•import threading
•import time

•# Define the event function
•def schedule_event(name, start):
        •now = time.time()
        •elapsed = int(now -start)
        •print('Elapsed:', elapsed, 'Name:', name)

•# Start time
•start = time.time()
•print('START:', time.ctime(start))

•# Schedule events using Timer
•t1 = threading.Timer(3, schedule_event, args=('EVENT_1', start))
•t2 = threading.Timer(2, schedule_event, args=('EVENT_2', start))

•# Start the timers
•t1.start()
•t2.start()
•t1.join()
•t2.join()

•# End time
•end = time.time()
•print('End:', time.ctime(end))
```

SCHEDULING THREADS USING THE SCHEDMODULE

•The sched module in Python's standard library provides a way to schedule tasks. It implements a generic event scheduler for running tasks at specific times. It provides similar tools like task scheduler in windows or Linux.

•The scheduler() class is defined in the sched module is used to create a scheduler object. Here is the syntax of the class

–scheduler(timefunc=time.monotonic, delayfunc=time.sleep)

•The methods defined in scheduler class include −

–scheduler.enter(delay, priority, action, argument=(), kwargs={}) −Events can be scheduled to run after a delay, or at a specific time. To schedule them with a delay, enter() method is used.

–scheduler.cancel(event) −Remove the event from the queue. If the event is not an event currently in the queue, this method will raise a ValueError.

–scheduler.run(blocking=True) −Run all scheduled events.

```
•import sched
•import time

•scheduler = sched.scheduler(time.time, time.sleep)

•def schedule_event(name, start):
    •now = time.time()
    •elapsed = int(now -start)
    •print('elapsed=',elapsed, 'name=', name)

•start = time.time()
•print('START:', time.ctime(start))
•scheduler.enter(2, 1, schedule_event, ('EVENT_1', start))
•scheduler.enter(5, 1, schedule_event, ('EVENT_2', start))
•scheduler.run()
•# End time
•end = time.time()
•print('End:', time.ctime(end))
```

Lab - 5
Python - Thread Pool

•A thread pool is a collection of threads that are managed by a pool. Each thread in the pool is called a worker or a worker thread. These threads can be reused to perform multiple tasks, which reduces the burden of creating and destroying threads repeatedly.

•Thread pools control the creation of threads and their life cycle, making them more efficient for handling large numbers of tasks

•. Python does not provide thread pooling directly through the threading module.

•We can implement thread-pools in Python using the following classes −

- •Python ThreadPoolClass
- •Python ThreadPoolExecutorClass

USING PYTHON THREADPOOLCLASS

•The multiprocessing.pool.ThreadPoolclass provides a thread pool interface within the multiprocessing module. It manages a pool of worker threads to which jobs can be submitted for concurrent execution.
•A ThreadPoolobject simplifies the management of multiple threads by handling the creation and distribution of tasks among the worker threads. It shares an interface with the Pool class, originally designed for processes, but has been adjusted to work with threads too.
•ThreadPoolinstances are fully interface-compatible with Pool instances and should be managed either as a context manager or by calling close() and terminate() manually.

```
•from multiprocessing.dummyimport Pool as ThreadPool
•import time
•def square(number):
        •sqr= number * number
        •time.sleep(1)
        •print("Number:{} Square:{}".format(number, sqr))

•def cube(number):
        •cub = number*number*number
        •time.sleep(1)
        •print("Number:{} Cube:{}".format(number, cub))

•numbers = [1, 2, 3, 4, 5]
•pool = ThreadPool(3)
•pool.map(square, numbers)
•pool.map(cube, numbers)
•pool.close()
```

USING PYTHON THREADPOOLEXECUTOR CLASS

•The ThreadPoolExecutor class of the Python the concurrent.futures module provides a high-level interface for asynchronously executing functions using threads. The concurrent.futures module includes Future class and two Executor classes −ThreadPoolExecutor and ProcessPoolExecutor.

•The concurrent.futures.Future class is responsible for handling asynchronous execution of any callable such as a function. To obtain a Future object, you should call the submit() method on any Executor object. It should not be created directly by its constructor

•result(timeout=None): This method returns the value returned by the call. If the call hasn't yet completed, then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a TimeoutError will be raised. If timeout is not specified, there is no limit to the wait time.

•cancel(): This method, attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return a Boolean value False. Otherwise the call will be cancelled and the method returns True.

•cancelled(): Returns True if the call was successfully cancelled.

•running(): Returns True if the call is currently being executed and cannot be cancelled.

•done(): Returns True if the call was successfully cancelled or finished running.

USING PYTHON THREADPOOLEXECUTOR CLASS

```
•from concurrent.futures import ThreadPoolExecutor
•from time import sleep

•def square(numbers):
        •for val in numbers:
        •ret = val*val
        •sleep(1)
        •print("Number:{} Square:{}".format(val, ret))

•def cube(numbers):
        •for val in numbers:
        •ret = val*val*val
        •sleep(1)
        •print("Number:{} Cube:{}".format(val, ret))


•if __name__ == '__main__':
•numbers = [1,2,3,4,5]
•executor = ThreadPoolExecutor(4)

•thread1 = executor.submit(square, (numbers))
•thread2 = executor.submit(cube, (numbers))

•print("Thread 1 executed ? :",thread1.done())
•print("Thread 2 executed ? :",thread2.done())
•sleep(2)

•print("Thread 1 executed ? :",thread1.done())
•print("Thread 2 executed ? :",thread2.done())
```

PYTHON -MAIN THREAD

•In Python, the main thread is the initial thread that starts when the Python interpreter is executed. It is the default thread within a Python process, responsible for managing the program and creating additional threads. Every Python program has at least one thread of execution called the main thread.

•The main thread by default is a non-daemon thread.

•The main thread will exit whenever it has finished executing all the code in your script that is not started in a separate thread. For instance, when you start a new thread using start() method, the main thread will continue to execute the remaining code in the script until it reaches the end and then exit.

•Accessing the Main Thread

- –threading.current_thread(): This function returns a threading.Thread instance representing the current thread.
- –threading.main_thread(): Returns a threading.Thread instance representing the main thread.

PYTHON -MAIN THREAD

```
•import threading
•import time

•def func(x):
        •time.sleep(x)
        •if not threading.current_thread() is threading.main_thread():
                •print('threading.current_thread() not threading.main_thread()')

•t = threading.Thread(target=func, args=(0.5,))
•t.start()
•print(threading.main_thread())
•print("Main thread finished")
```

```
•import threading
•import time
•def func(x):
        •print('Current Thread Details:',threading.current_thread())
        •for n in range(x):
                •print('Internal Thread Running', n)
        •print('Internal Thread Finished...')

•t = threading.Thread(target=func, args=(6,))
•t.start()
•for i in range(3):
        •print('Main Thread Running',i)
•print("Main Thread Finished...")
```

MAIN THREAD WAITING FOR OTHER THREADS

•To ensure that the main thread waits for all other threads to finish, you can join the threads using the join() method. By using the join() method, you can control the execution flow and ensure that the main thread properly waits for all other threads to complete their tasks before exiting. This helps in managing the lifecycle of threads in a multi-threaded Python program effectively.

```
•from threading import Thread
•from time import sleep

•def my_function_1():
        •print("Worker 1 started")
        •sleep(1)
        •print("Worker 1 done")

•def my_function_2(main_thread):
        •print("Worker 2 waiting for Worker 1 to finish")
        •main_thread.join()
        •print("Worker 2 started")
        •sleep(1)
        •print("Worker 2 done")

•worker1 = Thread(target=my_function_1)
•worker2 = Thread(target=my_function_2, args=(worker1,))
•worker1.start()
•worker2.start()
•for num in range(6):
        •print("Main thread is still working on task", num)
        •sleep(0.60)
•worker1.join()
•print("Main thread Completed")
```

Lab - 6
Python - Thread Priority

•In Python, currently thread priority is not directly supported by the threading module. unlike Java, Python does not support thread priorities, thread groups, or certain thread control mechanisms like destroying, stopping, suspending, resuming, or interrupting threads.

•Even thought Python threads are designed simple and is loosely based on Java's threading model. This is because of Python's Global Interpreter Lock (GIL), which manages Python threads.

•However, you can simulate priority-based behavior using techniques such as sleep durations, custom scheduling logic within threads or using the additional module which manages task priorities.

PYTHON -THREAD PRIORITY SETTING THE THREAD PRIORITY USING SLEEP()

•You can simulate thread priority by introducing delays or using other mechanisms to control the execution order of threads. One common approach to simulate thread priority is by adjusting the sleep duration of your threads.

•Threads with a lower priority sleep longer, and threads with a high priority sleep shorter.

```
•import threading
•import time

•class DummyThread(threading.Thread):
        •def__init__(self, name, priority):
                •threading.Thread.__init__(self)
                •self.name = name
                •self.priority= priority
        •def run(self):
                •name = self.name
                •time.sleep(1.0 * self.priority)
                •print(f"{name} thread with priority {self.priority} is running")

•# Creating threads with different priorities
•t1 = DummyThread(name='Thread-1', priority=4)
•t2 = DummyThread(name='Thread-2', priority=1)

•# Starting the threads
•t1.start()
•t2.start()

•# Waiting for both threads to complete
•t1.join()
•t2.join()
•print('All Threads are executed')
```

PYTHON -THREAD PRIORITYPRIORITIZING PYTHON THREADS USING THE QUEUE MODULE

•The queue module in Python's standard library is useful in threaded programming when information must be exchanged safely between multiple threads. The Priority Queue class in this module implements all the required locking semantics.

•With a priority queue, the entries are kept sorted (using the heapq module) and the lowest valued entry is retrieved first.

•The Queue objects have following methods to control the Queue −

- –**get()**−The get() removes and returns an item from the queue.
- –**put()**−The put adds item to a queue.
- –**qsize()**−The qsize() returns the number of items that are currently in the queue.
- –**empty()**−The empty( ) returns True if queue is empty; otherwise, False.
- –**full()**−the full() returns True if queue is full; otherwise, False.

PYTHON -THREAD PRIORITYPRIORITIZING PYTHON THREADS USING THE QUEUE MODULE

•from time import sleep
•from random import random, randint
•from threading import Thread
•from queue import PriorityQueue

•queue = PriorityQueue()

```
•def producer(queue):
        •print('Producer: Running')
        •for i in range(5):
                •# create item with priority
                •value = random()
                •priority = randint(0, 5)
                •item = (priority, value)
                •queue.put(item)

•# wait for all items to be processed
•queue.join()
•queue.put(None)
•print('Producer: Done')

•def consumer(queue):
        •print('Consumer: Running')
        •while True:
                •# get a unit of work
                •item = queue.get()
                •if item is None:
                        •break
                •sleep(item[1])
                •print(item)
                •queue.task_done()
        •print('Consumer: Done')

•producer = Thread(target=producer, args=(queue,))
•producer.start()
•consumer = Thread(target=consumer, args=(queue,))
•consumer.start()
•producer.join()
•consumer.join()
```

Lab - 7
DAEMON THREAD

•Sometimes, it is necessary to execute a task in the background. A special type of thread is used for background tasks, called a daemon thread. In other words, daemon threads execute tasks in the background. These threads handle non-critical tasks that may be useful to the application but do not hamper it if they fail or are canceled mid-operation.

•Also, a daemon thread will not have control over when it is terminated. The program will terminate once all non-daemon threads finish, even if there are daemon threads still running at that point of time.

Daemon	Non-daemon
A process will exit if only daemon threads are running (or if no threads are running).	A process will not exit if at least one non-daemon thread is running.
Daemon threads are used for background tasks.	Non-daemon threads are used for critical tasks.
Daemon threads are terminated abruptly.	Non-daemon threads run to completion.

DAEMON THREAD

```
•import threading
•from time import sleep

•# function to be executed in a new thread
•def run():
        •# get the current thread
        •thread = threading.current_thread()
        •# is it a daemon thread?
        •print(f'Daemon thread: {thread.daemon}')


•# Create a new thread and set it as daemon

•thread = threading.Thread(target=run, daemon=True)

•# start the thread
•thread.start()
•print('Is Main Thread is Daemon thread:', threading.current_thread().daemon)

•# Block for a short time to allow the daemon thread to run
•sleep(0.5)
```

DAEMON THREAD

```
•import threading
•from time import sleep

•# function to be executed in a new thread
•def run():
        •# get the current thread
        •thread = threading.current_thread()
        •# is it a daemon thread?
        •print(f'Daemonthread: {thread.daemon}')

•# Create a new thread
•thread = threading.Thread(target=run)

•# Using the daemon property set the thread as daemon before starting the thread
•thread.daemon= True

•# start the thread
•thread.start()
•print('Is Main Thread is Daemon thread:', threading.current_thread().daemon)

•# Block for a short time to allow the daemon thread to run
•sleep(0.5)
```

Lab - 8
Synchronizing Threads

PYTHON -SYNCHRONIZING THREADS

In Python, when multiple threads are working concurrently with shared resources, it's important to synchronize their access to maintain data integrity and program correctness. Synchronizing threads in python can be achieved using various synchronization primitives provided by the threading module, such as locks, conditions, semaphores, and barriers to control access to shared resources and coordinate the execution of multiple threads.

THREAD SYNCHRONIZATION USING LOCKS

•The lock object in the Python's threading module provide the simplest synchronization primitive. They allow threads to acquire and release locks around critical sections of code, ensuring that only one thread can execute the protected code at a time.

•A new lock is created by calling the Lock() method, which returns a lock object. The lock can be acquired using the acquire(blocking) method, which force the threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock and released using the release() method.

THREAD SYNCHRONIZATION USING LOCKS

```
•import threading
•counter = 10
•def increment(theLock, N):
        •global counter
        •for i in range(N):
                •theLock.acquire()
                •counter += 1
                •theLock.release()

•lock = threading.Lock()
•t1 = threading.Thread(
•t2 = threading.Thread(target=increment, args=[lock, 10])
•t3 = threading.Thread(target=increment, args=[lock, 4])

•t1.start()
•t2.start()
•t3.start()

•# Wait for all threads to complete

•for thread in (t1, t2, t3):
        •thread.join()

•print("All threads have completed")
•print("The Final Counter Value:", counter)
```

CONDITION OBJECTS FOR SYNCHRONIZING PYTHON THREADS

•Condition variables enable threads to wait until notified by another thread. They are useful for providing communication between the threads. The wait() method is used to block a thread until it is notified by another thread through notify() or notify_all().

```
•import threading
•counter = 0
•# Consumer function
•def consumer(cv):
        •global counter
        •with cv:
                •print("Consumer is waiting")
                •cv.wait() # Wait until notified by increment
                •print("Consumer has been notified. Current Counter value:", counter)

•# increment function
•def increment(cv, N):
        •global counter
        •with cv:
                •print("increment is producing items")
                •for i in range(1, N + 1):
                        •counter += i# Increment counter by i

•# Notify the consumer
•cv.notify()
•print("Increment has finished")
```

CONDITION OBJECTS FOR SYNCHRONIZING PYTHON THREADS

•# Create a Condition object
•cv = threading.Condition()
•# Create and start threads
•consumer_thread= threading.Thread(target=consumer, args=[cv])
•increment_thread= threading.Thread(target=increment, args=[cv, 5])
•consumer_thread.start()
•increment_thread.start()
•consumer_thread.join()
•increment_thread.join()
•print("The Final Counter Value:", counter)

SYNCHRONIZING THREADS USING THE JOIN() METHOD

•The join() method in Python's threading module is used to wait until all threads have completed their execution. This is a straightforward way to synchronize the main thread with the completion of other threads.

CONDITION OBJECTS FOR SYNCHRONIZING PYTHON THREADS

```
•import threading
•import time

•class MyThread(threading.Thread):
        •def__init__(self, threadID, name, counter):
                •threading.Thread.__init__(self)
                •self.threadID= threadID
                •self.name = name
                •self.counter= counter

        •def run(self):
                •print("Starting " + self.name)
                •print_time(self.name, self.counter, 3)

        •def print_time(threadName, delay, counter):
                •while counter:
                  •time.sleep(delay)
                •print("%s: %s" % (threadName, time.ctime(time.time())))
                •counter -= 1

•threads = []
•# Create new threads
•thread1 = MyThread(1, "Thread-1", 1)
•thread2 = MyThread(2, "Thread-2", 2)

•# Start the new Threads
•thread1.start()
•thread2.start()

•# Join the threads
•thread1.join()
•thread2.join()
•print("Exiting Main Thread")
```

ADDITIONAL SYNCHRONIZATION PRIMITIVES

•Using RLocks(threading.RLock) (Reentrant Lock) allows the same thread to acquire the lock multiple times. It's useful when a thread recursively calls functions that acquire the same.

•Using Events (threading.Event) allows threads to wait until a specific condition is met.

•Using Barriers (threading.Barrier) allows multiple threads to wait until they have all reached a certain point.

PYTHON -INTER-THREAD COMMUNICATION

•Inter-Thread Communication refers to the process of enabling communication and synchronization between threads within a Python multi-threaded program.

•Generally, threads in Python share the same memory space within a process, which allows them to exchange data and coordinate their activities through shared variables, objects, and specialized synchronization mechanisms provided by the threading module.

•To facilitate inter-thread communication, the threading module provides various synchronization primitives like, Locks, Events, Conditions, and Semaphores objects.

PYTHON -INTER-THREAD COMMUNICATION(THE EVENT OBJECT)

•An Event object manages the state of an internal flag so that threads can wait or set. Event object provides methods to control the state of this flag, allowing threads to synchronize their activities based on shared conditions.

•The flag is initially false and becomes true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

–is_set(): Return True if and only if the internal flag is true.
–set(): Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call wait() once the flag is true will not block at all.
–clear(): Reset the internal flag to false. Subsequently, threads calling wait() will block until set() is called to set the internal flag to true again.
–wait(timeout=None): Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls set() to set the flag to true, or until the optional timeout occurs. When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds.

•from threading import Event, Thread
•import time

•terminate = False

•def signal_state():
    •global terminate
    •while not terminate:
        •time.sleep(0.5)
        •print("Traffic Police Giving GREEN Signal")
        •event.set()
        •time.sleep(1)
        •print("Traffic Police Giving RED Signal")
        •event.clear()

```
•def traffic_flow():
        •global terminate
        •num= 0
        •while num< 10 and not terminate:
            •print("Waiting for GREEN Signal")
            •event.wait()
            •print("GREEN Signal ... Traffic can move")
            •while event.is_set() and not terminate:
              •num+= 1
              •print("Vehicle No:", num," Crossing the Signal")
              •time.sleep(1)
              •print("RED Signal ... Traffic has to wait")


•event = Event()
•t1 = Thread(target=signal_state)
•t2 = Thread(target=traffic_flow)
•t1.start()
•t2.start()

•# Terminate the threads after some time
•time.sleep(5)
•terminate = True

•# join all threads to complete
•t1.join()
•t2.join()
•print("Exiting Main Thread")
```

Lab - 9
Thread Communication & Deadlock

PYTHON -INTER-THREAD COMMUNICATION(THE CONDITION OBJECT)

•The Condition object in Python's threading module provides a more advanced synchronization mechanism. It allows threads to wait for a notification from another thread before proceeding.
•The Condition object are always associated with a lock and provide mechanisms for signaling between threads.

–acquire(*args): Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.
–release(): Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.
–wait(timeout=None): This method releases the underlying lock, and then blocks until it is awakened by a notify() or notify_all() call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.
–wait_for(predicate, timeout=None): This utility method may call wait() repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to False if the method timed out.
–notify(n=1): This method wakes up at most n of the threads waiting for the condition variable; it is a no-op if no threads are waiting.
–notify_all(): Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a RuntimeErroris raised.

PYTHON -INTER-THREAD COMMUNICATION(THE CONDITION OBJECT)

```
•from threading import Condition, Thread
•import time

•c = Condition()

•def thread_a():
        •print("Thread A started")
        •with c:
                •print("Thread A waiting for permission...")
                •c.wait()
                •print("Thread A got permission!")
        •print("Thread A finished")


•def thread_b():
        •print("Thread B started")
        •with c:
                •time.sleep(2)
                •print("Notifying Thread A...")
                •c.notify()
        •print("Thread B finished")

•Thread(target=thread_a).start()
•Thread(target=thread_b).start()
```

PYTHON -THREAD DEADLOCK

•A deadlock may be described as a concurrency failure mode. It is a situation in a program where one or more threads wait for a condition that never occurs. As a result, the threads are unable to progress and the program is stuck or frozen and must be terminated manually.

•Deadlock situation may arise in many ways in your concurrent program. Deadlocks are never not developed intentionally, instead, they are in fact a side effect or bug in the code.

•Common causes of thread deadlocks are listed below −

- –A thread that attempts to acquire the same mutexlock twice.
- –Threads that wait on each other (e.g. A waits on B, B waits on A).
- –When a thread that fails to release a resource such as lock, semaphore, condition, event, etc.
- –Threads that acquire mutexlocks in different orders (e.g. fail to perform lock ordering).

HOW TO AVOID DEADLOCKS IN PYTHON THREADS?

•When multiple threads in a multi-threaded application attempt to access the same resource, such as performing read/write operations on the same file, it can lead to data inconsistency. Therefore, it is important to synchronize concurrent access to resources by using locking mechanisms.

•The Python threading module provides a simple-to-implement locking mechanism to synchronize threads. You can create a new lock object by calling the Lock() class, which initializes the lock in an unlocked state.

LOCKING MECHANISM WITH THE LOCK OBJECT

•An object of the Lock class has two possible states −locked or unlocked, initially in unlocked state when first created. A lock doesn't belong to any particular thread.

•The Lock class defines acquire() and release() method

•The acquire() method of the Lock class changes the lock's state from unlocked to locked. It returns immediately unless the optional blocking argument is set to True, in which case it waits until the lock is acquired.

–Lock.acquire(blocking, timeout)

•The release() method when the state is locked, this method in another thread changes it to unlocked. This can be called from any thread, not only the thread which has acquired the lock.

–Lock.release()

•The release() method should only be called in the locked state. If an attempt is made to release an unlocked lock, a RuntimeErrorwill be raised.

LOCKING MECHANISM WITH THE LOCK OBJECT

```
•from threading import Thread, Lock
•import time
•lock=Lock()
•threads=[]
•class myThread(Thread):
        •def__init__(self,name):
            •Thread.__init__(self)
            •self.name=name
        •def run(self):
            •lock.acquire()
            •synchronized(self.name)
            •lock.release()

•def synchronized(threadName):
        •print ("{} has acquired lock and is running synchronized method".format(threadName))
        •counter=5
        •while counter:
            •print ('**', end='')
            •time.sleep(2)
            •counter=counter-1
        •print('\nlockreleased for', threadName)

•t1=myThread('Thread1')
•t2=myThread('Thread2')

•t1.start()
•threads.append(t1)
•t2.start()
•threads.append(t2)

•for t in threads:
            •t.join()
•print ("end of main thread")
```

SEMAPHORE OBJECT FOR SYNCHRONIZATION

•In addition to locks, Python threading module supports semaphores, which offering another synchronization technique. It is one of the oldest synchronization techniques invented by a well-known computer scientist, Edsger W. Dijkstra.
•The basic concept of semaphore is to use an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().
•The Semaphore class in threading module defines acquire() and release() methods.

```
•from threading import *
•import time

•# creating thread instance where count = 3
•lock = Semaphore(4)
•# creating instance
•def synchronized(name):
        •# calling acquire method
        •lock.acquire()
        •for n in range(3):
            •print('Hello! ', end = '')
            •time.sleep(1)
            •print( name)
            •# calling release method
            •lock.release()

•# creating multiple thread
•thread_1 = Thread(target = synchronized , args= ('Thread 1',))
•thread_2 = Thread(target = synchronized , args= ('Thread 2',))
•thread_3 = Thread(target = synchronized , args= ('Thread 3',))

•# calling the threads
•thread_1.start()
•thread_2.start()
•thread_3.start()
```