SET-224 /CET-225
Operating Systems

LAB # 07

LAB Title

Understanding Threads
Multithreaded Programming using Python

Assessment of CLO: 04, PLO: 05

Student Name:	
Roll No.	

Semester		Session	

S. No.	Perf. Level / Criteria	Excellent (2.5)	Good (2)	Satisfactory (1.5)	Needs Improvement (0 ~ 1)	Marks Obtained
1	Project Execution & Implementation	Fully functional, optimized, and well-structured.	Minor errors, mostly functional.	Some errors, requires guidance.	Major errors, non-functional, or not Performed.	
2	Results & Debugging Or Troubleshooting	Accurate results with effective debugging Or Troubleshooting.	Mostly correct, some debugging Or Troubleshooting needed.	Partial results, minimal debugging Or Troubleshooting.	Incorrect results, no debugging Or Troubleshooting, or not attempted.	
3	Problem-Solving & Adaptability (VIVA)	Creative approach, efficiently solves challenges.	Adapts well, minor struggles.	Some adaptability, needs guidance.	Lacks innovation or no innovation, unable to solve problems.	
4	Report Quality & Documentation	Clear, structured, with detailed visuals.	Mostly clear, minor gaps.	Some clarity issues, missing details.	Poorly structured, lacks clarity, or not submitted.	
	Total Marks Obtained Out of 10					

Experiment evaluated by

Instructor's Name	Engr.Bushra Aziz	
Date		Signature	

Objective: Understanding the concept of threads and multithreaded programming in python.

Theory:

Thread is the smallest unit of processing. It is scheduled by an OS. In general, it is contained in a process. So, multiple threads can exist within the same process. It shares the resources with the process. Each thread belongs to exactly one process and no thread can exist outside a process.

A thread uses the same address space of a process. A process can have multiple threads. A key difference between processes and threads is that multiple threads share parts of their state. Typically, multiple threads can read from and write to the same memory (no process can directly access the memory of another process). However, each thread still has its own stack of activation records and its own copy of CPU registers, including the stack pointer and the program counter, which together describe the state of the thread's execution. A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads. Processes are independent while thread is within a process. Processes have separate address spaces while threads share their address spaces. Multithreading has some advantages over multiple processes. Threads require less overhead to manage than processes, and intra-process thread communication is less expensive than inter-process communication.

Types of Threads:

User Level thread (ULT) –

User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

Kernel Level Thread (KLT) –

Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Viewing threads of a process/processes on Linux Ps Command

In ps command, "-T" option enables thread views. The following command list all threads created by a process with <pid>.

One of the command for finding <pid> is use ps aux |grep (python-filename)

For example: ps aux | grep thread2.py, where thread2.py is filename in this case.

```
labit@labit:~$ ps aux | grep thread2
labit        17203  0.0  0.2 167668  9088 pts/1    Sl+  09:45   0:00 python3 thread2.py
labit        17207  0.0  0.0   9564  2688 pts/0    S+   09:45   0:00 grep --color=auto thread2
labit@labit:~$ ps -T -p 17203
    PID    SPID TTY          TIME CMD
  17203   17203 pts/1    00:00:00 python3
  17203   17204 pts/1    00:00:00 python3
  17203   17205 pts/1    00:00:00 python3
labit@labit:~$
```

The "SPID" column represents thread IDs, and "CMD" column shows thread names.

Top Command

The top command can show a real-time view of individual threads. To enable thread views in the top output, invoke top with "-H" option. This will list all Linux threads. You can also toggle on or off thread view mode while top is running, by pressing 'H' key.

```
top - 09:48:18 up 50 min,  1 user,  load average: 0.01, 0.09, 0.10
Threads: 547 total,   1 running, 546 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.8 sy,  0.0 ni, 99.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3868.6 total,    625.6 free,   1065.8 used,   2177.2 buff/cache
MiB Swap:   3898.0 total,   3898.0 free,      0.0 used.   2513.0 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
17214 labit     20   0   13884   4608   3456 R   1.6   0.1   0:00.32 top
 1740 labit     20   0 4048836 372852 143160 S   1.3   9.4   0:35.57 gnome-shell
 1755 labit     20   0 4048836 372852 143160 S   0.7   9.4   0:29.38 llvmpipe-1
 2028 labit     20   0  142344  39124  29156 S   0.7   1.0   0:06.60 vmtoolsd
```

*Command-line tools such as ps or top, which display process-level information by default, can be instructed in many ways to display thread-level information.

THREAD CREATION in Python- (Multithreaded Programming):

There are two modules which support the usage of threads in Python3 −

_thread

 Threading

The Threading Module

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module

The threading module has the Thread class that implements threading. The methods provided by the Thread class are as follows −

- run() − The run() method is the entry point for a thread.

- start() − The start() method starts a thread by calling the run method.

- join([time]) − The join() waits for threads to terminate.

- isAlive() − The isAlive() method checks whether a thread is still executing.

- getName() − The getName() method returns the name of a thread.

- setName() − The setName() method sets the name of a thread.

To implement a new thread using the threading module, you have to do the following –

- Import the threading module.

- Define the function for the thread.

- Create a thread object and start it.

- Use join() if you need to wait for the thread(s) to complete.

- Manage multiple threads using loops and lists

Example 1:This program demonstrates multithreading by launching two threads that print different subject names and then pause execution for 30 seconds, allowing both threads to run in parallel.

```
  GNU nano 6.2                                                thread2.py *
import threading
import time
#user-defined function
def subname(name):
 print ("Subject name is",name)
 time.sleep(30)

#creating first thread object i.e t1
t1=threading.Thread(target=subname,args=("Operating Systems",))

#creating second thread object i.e t2
t2=threading.Thread(target=subname,args=("Data Structures",))

#start() method to start thread execution
t1.start()
t2.start()

#join() method to wait for threads to terminate
t1.join()
t2.join()
```

```
labit@labit:~$ nano thread2.py
labit@labit:~$ python3 thread2.py
Subject name is Operating Systems
Subject name is Data Structures
labit@labit:~$
```

Example python program for running multiple threads: This Python program demonstrates multithreading by dynamically creating a separate thread for each subject name in a list, where each thread prints the subject name and simulates a delay using `sleep`, allowing concurrent execution of multiple tasks.

```
GNU nano 6.2                                              threadM.py
import threading
import time
#user-defined function
def subname(sname):
 print ("Subject name is",sname)
 time.sleep(30)

#list to hold all threads
threads = []

#subject-names to pass to the threads
names = ["Operating suystems","Data structures","Computer programming","Embedded Systems"]

#creating thread object for every subject-name and add it to the list
for sname in names:

  t=threading.Thread(target=subname,args=(sname,))
  threads.append(t)


#start() method to start each thread execution
for t in threads:
 t.start()


#join() method to wait for each thread to terminate
for t in threads:
 t.join()
```

```
labit@labit:~$ python3 threadM.py
Subject name is Operating suystems
Subject name is Computer programming
Subject name is Data structures
Subject name is Embedded Systems
```

Limitations and difficulty in using Python for Multithreaded work/tasks:

GIL (global interpreter lock), the mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. Due to the GIL only one thread can be executed at a time. Therefore, the above code is concurrent but not parallel. Multi-Threading in python can be considered as an appropriate model only if you want to run multiple I/O-bound tasks simultaneously.

Using the threading module in Python or any other interpreted language with a GIL can actually result in reduced performance. If your code is performing a CPU bound task, using the threading module will result in a slower execution time. For CPU bound tasks and truly parallel execution, we can use the multiprocessing module instead of multithreading in python

Exercises:

1. Modify Example 1 to display strings via two independent threads:

    thread1: "Hello! StudentName   ",

    thread2: "Student roll no is: "

2. Write a Python program that performs basic arithmetic operations (addition, subtraction, multiplication, and division) using multithreading. Each operation should be handled by a separate, independent thread, allowing the calculations to be executed concurrently.