Discipline

Parallel Computing: Analysis and Synthesis of Threads and Processes

CS5003

Assignment 3

Sheikh Muhammed Tadeeb (AU19B1014)

❖ Problem Statement:

Work on a segment of coded block covering the following topics:

1. Multiple Thread Creation - Task I
2. Multiple Process Creation - Task II
3. Execute Scheduling - Task III
4. Execute Thread Lifecycle - Task IV
5. Generate and Kill daemon process - Task V

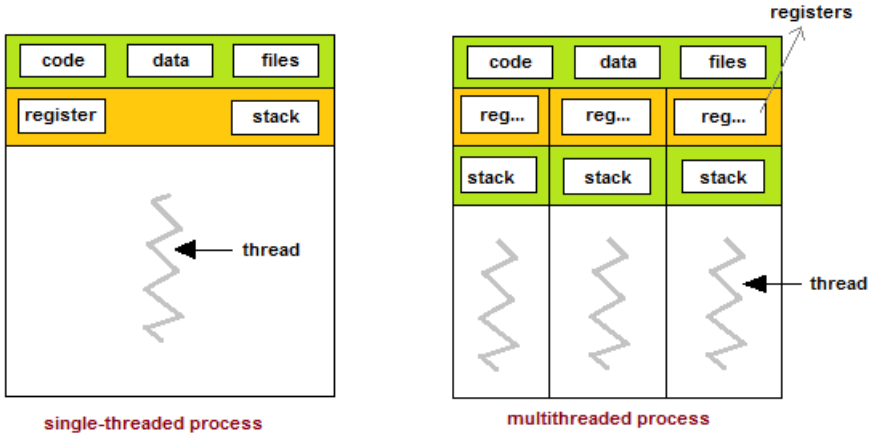
❖ Solution:

1. Multiple Thread Creation:

- *Brief:*

  Multithreading in Python programming is a well-known technique in which multiple threads in a process share their data space with the main thread which makes information sharing and communication within threads easy and efficient. Threads are lighter than processes. Multi threads may execute individually while sharing their process resources. The purpose of multithreading is to run multiple tasks and function cells at the same time.

  Multithreading allows you to break down an application into multiple sub-tasks and run these tasks simultaneously. If you use multithreading properly, your application speed, performance, and rendering can all be improved.



The below code shows the creation of new thread using a function:

- *Code:*

```
def my_function():

    """Assignment Task-I:In this task you are suppose to showcase process
    and thread creation mechanisms logic of coded block need to be
explained add screen snippets of action noticed and observation """
    return None


print("Task1__doc__:")
print(my_function.__doc__)
import os
import threading

# a simple function that disturbs CPU cycles
def cpu_dist():
    while True:
        pass
pid=os.getpid()
print('\n  Process ID: ',pid)#generate pid at this location read os module
doc
print('Thread Count TC: ', threading.active_count())
for th in threading.enumerate():
    print(th)

print('\n Write disturbs according to number of physical cores you
have...')
for i in range(1,13): #add range here based on number of logical processors
you have
    threading.Thread(target=cpu_dist).start()

# display information about this process
print('\n  Process ID: ',pid)#generate pid at this location read os module
doc
print('Thread Count: ', threading.active_count())
for th in threading.enumerate():
    print(th)
```

- *Usage of methods and purpose:*

  So, what we did in the above code,

  1. We defined a function cpu_dist to create a thread.

  2. Then we used the threading module to create a thread that invoked the function as its target.

  3. Then we used start () method to start the Python thread.

- *Output on Windows:*

```
runfile('C:/Users/Sheikh Tadeeb/Downloads/Assignment-III/task1.py',
wdir='C:/Users/Sheik Tadeeb/Downloads/Assignment-III')
Task1__doc__:
Assignment Task-I:In this task you are suppose to showcase process
    and thread creation mechanisms logic of coded block
    need to be explained add screen snippets of action noticed and
observation

  Process ID:  9500
Thread Count TC:  7
<_MainThread(MainThread, started 5276)>
<Thread(Thread-6, started daemon 18348)>
<Heartbeat(Thread-7, started daemon 19900)>
<ControlThread(Thread-5, started daemon 19320)>
<HistorySavingThread(IPythonHistorySavingThread, started 18576)>
<Thread(Thread-8, started 18996)>
<ParentPollerWindows(Thread-4, started daemon 20800)>

 Write disturbs according to number of physical cores you have...

  Process ID:  9500
Thread Count:  19
<_MainThread(MainThread, started 5276)>
<Thread(Thread-6, started daemon 18348)>
<Heartbeat(Thread-7, started daemon 19900)>
<ControlThread(Thread-5, started daemon 19320)>
<HistorySavingThread(IPythonHistorySavingThread, started 18576)>
<Thread(Thread-8, started 18996)>
<ParentPollerWindows(Thread-4, started daemon 20800)>
<Thread(Thread-9, started 19048)>
<Thread(Thread-10, started 20548)>
<Thread(Thread-11, started 2448)>
<Thread(Thread-12, started 19092)>
<Thread(Thread-13, started 12772)>
<Thread(Thread-14, started 4500)>
<Thread(Thread-15, started 9424)>
<Thread(Thread-16, started 21268)>
<Thread(Thread-17, started 9652)>
<Thread(Thread-18, started 15880)>
<Thread(Thread-19, started 12284)>
<Thread(Thread-20, started 18048)>
```

- *Output on Linux:*

```
tadeeb@tadeebsUbuntu:~$ python3 task1_AU19B1009.py
Task1__doc__:
Assignment Task-I:In this task you are suppose to showcase process
    and thread creation mechanisms logic of coded block
```

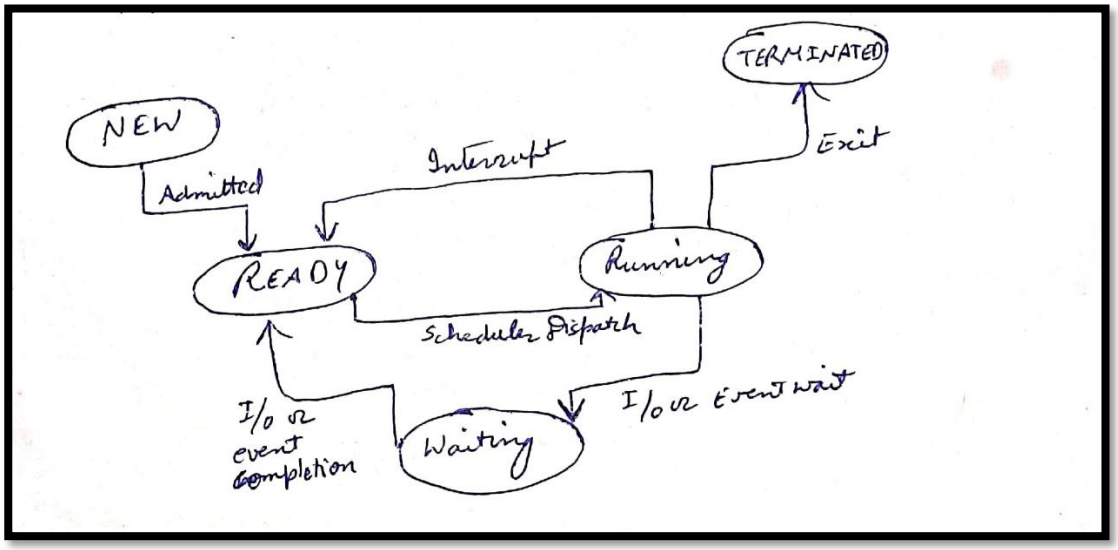
need to be explained add screen snippets of action noticed **and** observation

```
  Process ID:  4621
Thread Count TC:  1
<_MainThread(MainThread, started 140130232043328)>

 Write disturbs according to number of physical cores you have...

  Process ID:  4621
Thread Count:  13
<_MainThread(MainThread, started 140130232043328)>
<Thread(Thread-1, started 140130206680832)>
<Thread(Thread-2, started 140130198288128)>
<Thread(Thread-3, started 140130189895424)>
<Thread(Thread-4, started 140130181502720)>
<Thread(Thread-5, started 140130173110016)>
<Thread(Thread-6, started 140130164717312)>
<Thread(Thread-7, started 140129817589504)>
<Thread(Thread-8, started 140129809196800)>
<Thread(Thread-9, started 140129800804096)>
<Thread(Thread-10, started 140129792411392)>
<Thread(Thread-11, started 140129784018688)>
<Thread(Thread-12, started 140129775625984)>
```

- *State Diagram:*



- *Inference:*

  As we can when we didn't created threads only 7 threads were running which were kernel threads but after creating 12 new threads as we have 12 logical processors so to run each thread on respective core, we can see later the thread number got

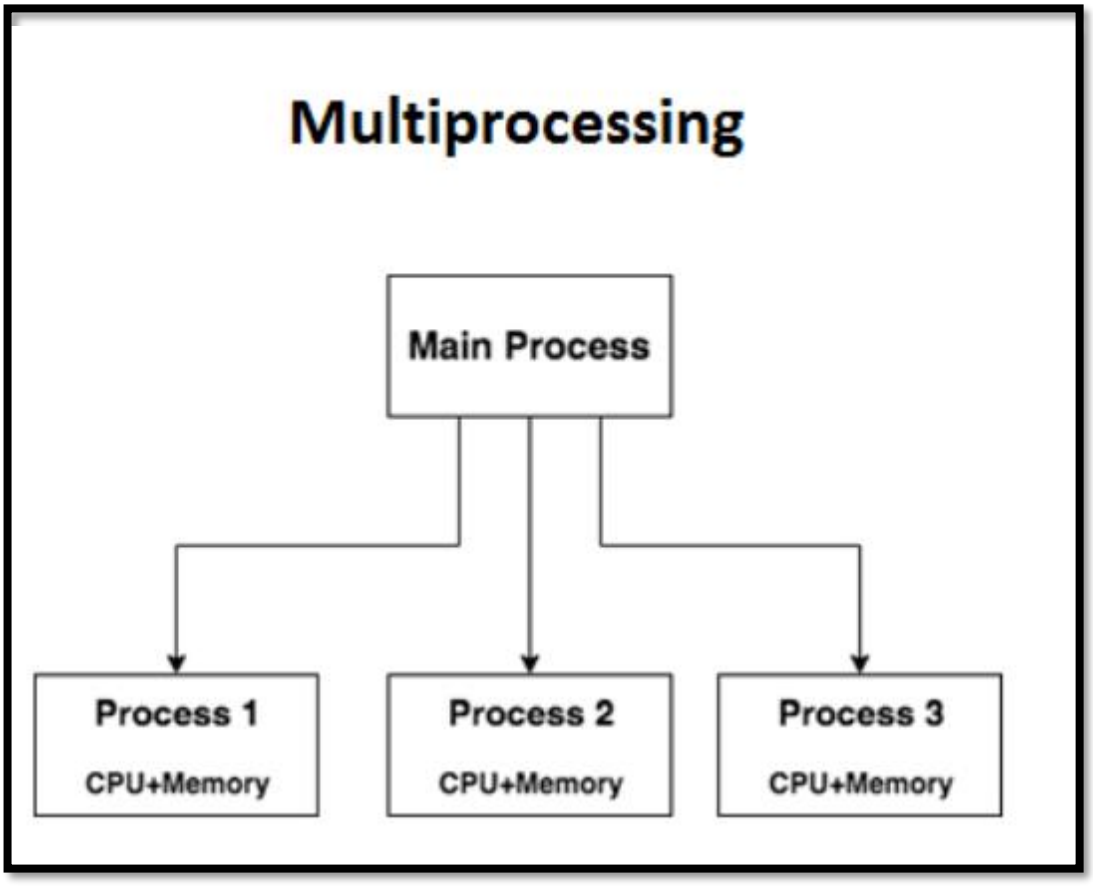
incremented from 7 to 19 i.e., our newly created 12 threads got added with the previous count of threads.

2. Multiple Process Creation:

- *Brief:*

  Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system. Consider a computer system with a single processor. If it is assigned several processes at the same time, it will have to interrupt each task and switch briefly to another, to keep all of the processes going.

  This situation is just like a chef working in a kitchen alone. He has to do several tasks like baking, stirring, kneading dough, etc.



- *Code:*

```
def my_function():

    """ Assignment Task-II:In this task you are suppose to showcase
process
    creation mechanisms (single threaded multiple process) architecture
the coded block
    need to be explained add screen snippets of action noticed and
observation """
    return None

print("Task2__doc__:")
print(my_function.__doc__)

import os
import threading
#import multiprocessing library

def cpu_dist():
    while True:
        pass

print('Hi I am AU19B1014',__name__)
#create entry point to the program at this location use using __name__
== __main__ conditioning
# display information about this process
if __name__ == "__main__":
    print('\n  Process ID: ', os.getpid())#get pid at this point
    print('Thread Count: ', threading.active_count())
    for th in threading.enumerate():
        print(th)

    print('Start disturbance cycle based on your configuration')
    for i in range(1,5): # Decide the range with reason
        threading.Thread(target=cpu_dist).start()

    # display information about this process
    print('\n  Process ID: ', os.getpid())
    print('Thread Count: ', threading.active_count())#get active count
of thread
    for th in threading.enumerate():
        print(th)
```

- *Usage of methods and purpose:*

  In Python, the multiprocessing module includes a very simple and intuitive API for dividing work between multiple processes.

  To import the multiprocessing module, we do:

  import multiprocessing

  To create a process, we create an object of Process class. It takes following arguments:

target: the function to be executed by process.

To start a process, we use *start* method of *Process* class.

In above program, we use *os.getpid()* function to get ID of process running the current target function.

- *Output of Linux:*

```
tadeeb@tadeebsUbuntu:~$ python3 task2_AU19B1009.py
Task2__doc__:
 Assignment Task-II:In this task you are supposed to showcase
process
    creation mechanisms (single threaded multiple process)
architecture the coded block
    need to be explained add screen snippets of action noticed
and observation
Hi I am AU19B1014 __main__

  Process ID:  4873
Thread Count:  1
<_MainThread(MainThread, started 139981898667840)>
Start disturbance cycle based on your configuration

  Process ID:  4873
Thread Count:  7
<_MainThread(MainThread, started 139981898667840)>
<Thread(Thread-1, started 139981872482048)>
<Thread(Thread-2, started 139981864089344)>
<Thread(Thread-3, started 139981855696640)>
<Thread(Thread-4, started 139981641217792)>
<Thread(Thread-5, started 139981632825088)>
<Thread(Thread-6, started 139981624432384)>
```

- *Output of Windows:*

```
runfile('C:/Users/Sheikh Tadeeb/Downloads/task2_AU19B1009.py',
wdir='C:/Users/Sheikh Tadeeb/Downloads')
Task2__doc__:
 Assignment Task-II:In this task you are suppose to showcase
process
    creation mechanisms (single threaded multiple process)
architecture the coded block
    need to be explained add screen snippets of action noticed
and observation
Hi I am AU19B1014 __main__

  Process ID:  18064
Thread Count:  7
<_MainThread(MainThread, started 13456)>
```

```
<Thread(Thread-6, started daemon 24412)>
<Heartbeat(Thread-7, started daemon 22228)>
<ControlThread(Thread-5, started daemon 18324)>
<HistorySavingThread(IPythonHistorySavingThread, started 24032)>
<Thread(Thread-8, started 24472)>
<ParentPollerWindows(Thread-4, started daemon 24120)>
Start disturbance cycle based on your configuration

  Process ID:  18064
Thread Count:  34
<_MainThread(MainThread, started 13456)>
<Thread(Thread-6, started daemon 24412)>
<Heartbeat(Thread-7, started daemon 22228)>
<ControlThread(Thread-5, started daemon 18324)>
<HistorySavingThread(IPythonHistorySavingThread, started 24032)>
<Thread(Thread-8, started 24472)>
<ParentPollerWindows(Thread-4, started daemon 24120)>
<Thread(Thread-9, started 17956)>
<Thread(Thread-10, started 9688)>
<Thread(Thread-11, started 14772)>
<Thread(Thread-12, started 16592)>
<Thread(Thread-13, started 18316)>
<Thread(Thread-14, started 20844)>
<Thread(Thread-15, started 21496)>
<Thread(Thread-16, started 6836)>
<Thread(Thread-17, started 2736)>
<Thread(Thread-18, started 21604)>
<Thread(Thread-19, started 14796)>
<Thread(Thread-20, started 16664)>
<Thread(Thread-21, started 17680)>
<Thread(Thread-22, started 1544)>
<Thread(Thread-23, started 15380)>
<Thread(Thread-24, started 18228)>
<Thread(Thread-25, started 23360)>
<Thread(Thread-26, started 17976)>
<Thread(Thread-27, started 23592)>
<Thread(Thread-28, started 23008)>
<Thread(Thread-29, started 8064)>
<Thread(Thread-30, started 20276)>
<Thread(Thread-31, started 21064)>
<Thread(Thread-32, started 4504)>
<Thread(Thread-33, started 14204)>
<Thread(Thread-34, started 15524)>
<Thread(Thread-35, started 8876)>
```

- *Inference:*

  Two or more processors or CPUs present in the same computer, sharing system bus, memory and I/O is called Multiprocessing System. It allows parallel execution of different processors. Notice that it matches with the process IDs of p1 and p2 which we obtain using pid attribute of Process class.

Each process runs independently and has its own memory space.

As soon as the execution of target function is finished, the processes get terminated.

3. Execute Scheduling:

- *Brief:*

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

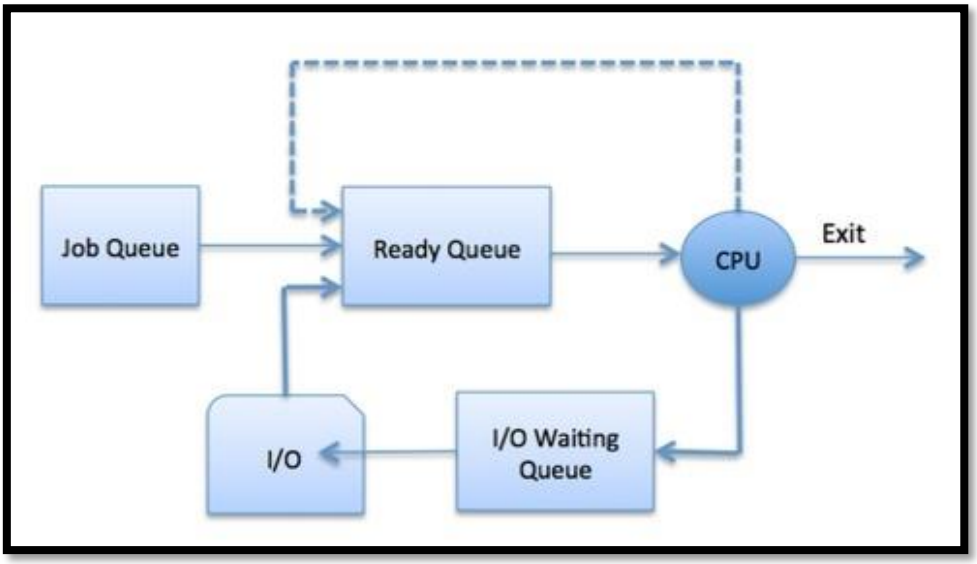
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues:

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

- Queue Diagram:



- Code:

```
def my_function():

   """ Assignment Task-III: Create two threads and run both of them for
specific duration of time
       comment on schedular behaviour and explain the process created
to generate these threads
       You are also required to print the thread ID that should mention
parent and child thread """

print("Task3__doc__:")
print(my_function.__doc__)

def my_function():

    """ Assignment Task-III: Create two threads and run both of them
for specific duration of time
        comment on schedular behaviour and explain the process created
to generate these threads
        You are also required to print the thread ID that should
mention parent and child thread """

print("Task3__doc__:")
print(my_function.__doc__)

#import required dependencies
import threading
import time
speech = True

def speech_chance():
    name = threading.current_thread().getName()
    speech_count = 0                #initialize the variable
    while speech:
        print(name,'Given a Speech ',threading.get_ident())
        print()
        speech_count+=1               #increament the count by one
    print('Count:',speech_count)

#create a entry point for the code use __name__ __main__ block
if __name__ == '__main__':
    threading.Thread(target=speech_chance,
name='parent').start()#assign Target
    threading.Thread(target=speech_chance,name='child').start() #assign
Target
# add timesleep for the thread exection you can use time module
    time.sleep(2)
    speech = False
```

- *Usage of method and purpose:*

  We run many tasks during our day-to-day work that can be automated instead of performed repetitively. A task scheduler allows you to run your task after a particular

period and can be set to perform a task at any given time, be it weekends or daily. Scheduling can be useful to fetch data from a database or to store some data periodically. Scheduling can also be used to train machine learning models as new data comes in.

We have created a method named as speech_chance() which creates threads, so initially we have kept speech count as 0 and created the 2 threads running simultaneously and analysed the CPU scheduling for both parallel threads running and counted the no. of speech run.

In the Main method we have created 2 threads naming as Parent and Child and executed the

- *Output in Windows:*

```
parent Given a Speech  22644

parent Given a Speech  22644

parent
child Given a Speech   21636

child Given a Speech  21636

child Given a Speech  21636
Given a Speech  22644

parent Given a Speech
22644
child Given a Speech  21636

child Given a Speech  21636


child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  21636

child Given a Speech parent 21636

child  Given a Speech  Given a Speech  22644

parent 21636
```

```
child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  21636

child Given a Speech  22644

Given a Speech  21636

child Given a Speech parent Given a Speech  22644

parent  21636

child Given a Speech  21636

child Given a Speech  21636Given a Speech

Count:  22644

Count: 1412
1419
```

- *Output in Linux:*

```
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Child Given a Speech 139758602831616
Child Given a Speech 139758602831616
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Child Given a Speech 139758602831616
Child Given a Speech 139758602831616
Child Given a Speech 139758602831616
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Child Given a Speech 139758602831616
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Child Given a Speech 139758602831616
Child Given a Speech 139758602831616
Child Given a Speech 139758602831616
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Parent Given a Speech 139758611224320
Child Given a Speech 139758602831616
Parent Given a Speech 139758611224320

Speech Count: Child Given a Speech 139758602831616
 Parent   27260

Speech Count:  Child   31969
```

- *Inference:*

  On creating and running the multiple threads on both the linux and windows platform we observed that the scheduling done by the CPU for both the threads speech count has a huge variation as the speech count of both parallel threads in windows comes out to be (Parent:2137, Child:2167) while on Linux (Parent:27260, Child:31969) this variation in terms of platforms as well as the chance given is different.

  This is because of the time we provided was 1 second in both the cases, but it also depends on the no. of CPU's allotted on both platforms, In windows time taken is also less and variation is also not too big but in case of linux we run on virtual machine, CPU allotted to it are 3 and total we have as 12 therefore time take and CPU scheduling is too big in Linux.

  Thus Speech chance is totally depends on how the CPU schedules the parallel threads serves first and also depends on the no. of logical processors of our System.

4. Execute Thread Lifecycle:

- *Brief:*

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

The diagram shown below represent various states of a thread at any instant of time:

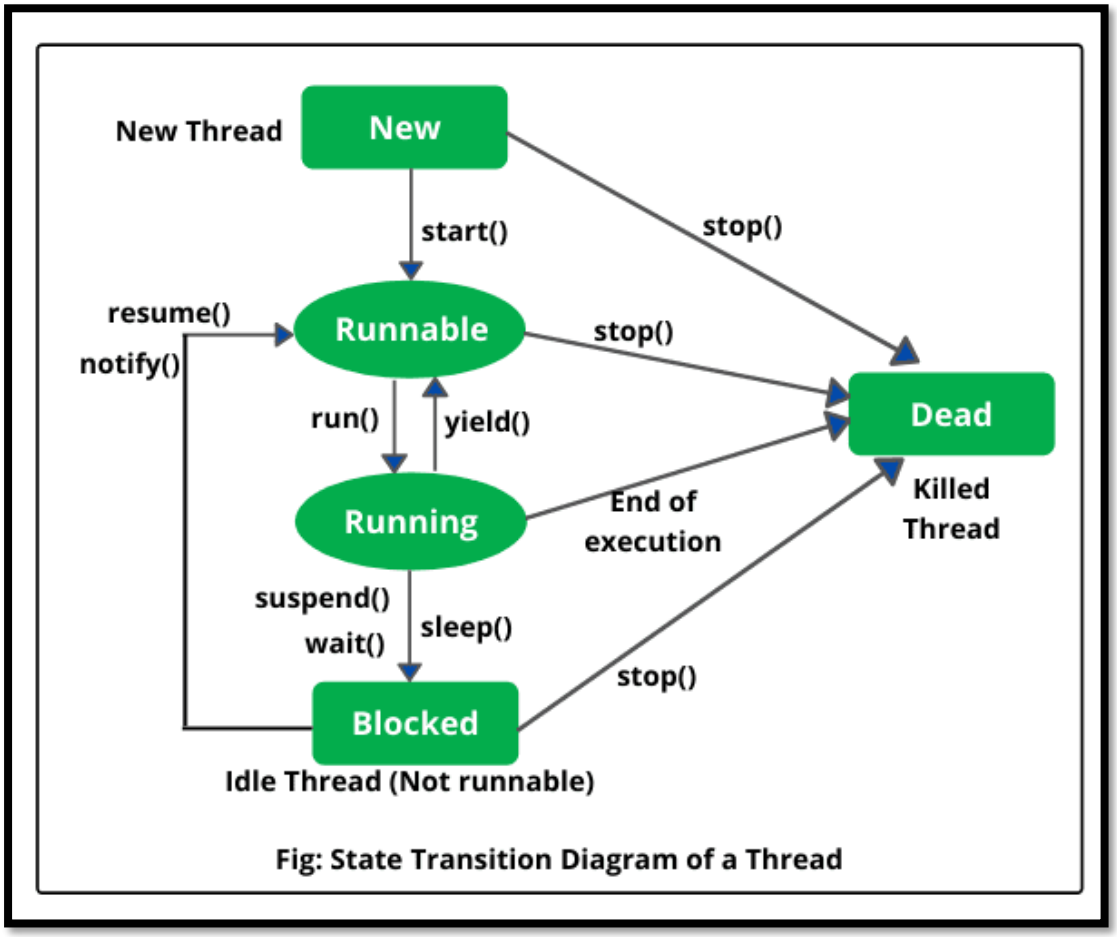


Fig: State Transition Diagram of a Thread

- *Life Cycle of a thread:*

1. New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

2. Runnable State: A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

3. Blocked/Waiting state: When a thread is temporarily inactive, then it's in one of the following states:
    - Blocked
    - Waiting

    For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

4. Timed Waiting: A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

5. Terminated State: A thread terminates because of either of the following reasons:
    - Because it exists normally. This happens when the code of thread has entirely executed by the program.
    - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

    A thread that lies in a terminated state does no longer consumes any cycles of CPU.

- *Code:*

```
def my_function():

    """ Assignment Task-IV:  Two threads presenting thread lifecycle
         Sequence of thread responce is as follows;

        Nipun started & requesting Anirudhs role
        Nipun tells anirudh to start.
        Anirudh started & waiting forr Nipuns turn...
        Nipun continues his role.
        Nipun patiently waits Anirudh to finish &join
        Anirudh completed his role
```

```
        Nipun & Anirudh are both done their play cheers CS5003
    """
print("Task4__doc__:")
print(my_function.__doc__)


#import required dependencies
import threading
import time

class speekeranirudh(threading.Thread):
    def __init__(self):
        super().__init__()
        print("Nipun tells aniruddh to start")

    def run(self):
        print('Anirudh started & waiting for Nipuns turn...')
#add sleeping time for this thread you can use time module
        time.sleep(2)
        print('Anirudh done with his role')

# main thread a.k.a. Nipun
if __name__ == '__main__':
    print("Nipun started and requesting Aniruddh's role")
    anirudh = speekeranirudh()
    print('')

#add start method on child thread
    anirudh.start()
    print('Nipun continues his role')
    time.sleep(0.5)

    print('Nipun patiently waits for Anirudh to finish and join')
#use join method to join threads
    anirudh.join()
    print('Nipun and Anirudh are both done with their roles, cheers
CS5003')
```

- *Usage and purpose:*

  Life Cycle of Thread is the state transition of a thread that starts from its birth and ends on its death. When an instance of a thread is created and is executed by calling the start() method of the Thread class, the thread goes into a runnable state.

  In our program we have 2 threads to be called that are Nipun and Anirudh in which both the thread cycle is analysed which will be running first and then first calling the other thread to join (requesting) and both of them work by one in waiting state other in running and upon successful completion they gets terminated.

- *Output on Windows:*

```
runfile('C:/Users/Sheikh Tadeeb/Downloads/task4_AU19B1009.py',
wdir='C:/Users/Sheikh Tadeeb/Downloads')
Task4__doc__:
 Assignment Task-IV:  Two threads presenting thread lifecycle
        Sequence of thread responce is as follows;

       Nipun started & requesting Anirudhs role
       Nipun tells anirudh to start.
       Anirudh started & waiting forr Nipuns turn...
       Nipun continues his role.
       Nipun patiently waits Anirudh to finish &join
       Anirudh completed his role
       Nipun & Anirudh are both done their play cheers CS5003

Nipun started and requesting Aniruddh's role
Nipun tells aniruddh to start

Anirudh started & waiting for Nipuns turn...Nipun continues his role

Nipun patiently waits for Anirudh to finish and join
Anirudh done with his role
Nipun and Anirudh are both done with their roles, cheers CS5003
```

- *Output on Linux:*

```
tadeeb@tadeebsUbuntu:~$ python3 task4_AU19B1014.py
Task4__doc__:
 Assignment Task-IV:  Two threads presenting thread lifecycle
        Sequence of thread responce is as follows;

       Nipun started & requesting Anirudhs role
       Nipun tells anirudh to start.
       Anirudh started & waiting forr Nipuns turn...
       Nipun continues his role.
       Nipun patiently waits Anirudh to finish &join
       Anirudh completed his role
       Nipun & Anirudh are both done their play cheers CS5003

Nipun started and requesting Aniruddh's role
Nipun tells aniruddh to start

Anirudh started & waiting for Nipuns turn...
Nipun continues his role
Nipun patiently waits for Anirudh to finish and join
Anirudh done with his role
Nipun and Anirudh are both done with their roles, cheers CS5003
```

- *Inference:*

  On executing the code by modifying some content we get the thread lifecycle and their sequence response with it. Here the Nipun thread will get initiated first and then it requests another thread role i.e.(Anirudh). Nipun tells Anirudh to start, once Anirudh started it waits for Nipun turns to resume back, while Nipun is in waiting state, once Nipun continues his role it waits patiently for Anirudh to finish and join back on the Nipun's call, on join back Anirudh and Nipun both done their role and gets terminated.

5. Generate and Kill daemon process:

- *Brief:*

  Student1 cleans a hostel room meanwhile student2 Cooks a food analogy. The system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

- *Code:*

```
def my_function():

        """ Assignment -III Task V: student1 cleans a hostel room
meanwhile student2 Cooks a food analogy """
print("Task5__doc__:")
print(my_function.__doc__)

import time
import threading

def room_cleaner():
        while True:
            print('Cleaning Started')
            time.sleep(1)


if __name__ == '__main__':
    somesh = threading.Thread(target=room_cleaner)# add target child
thread function defination
#add daemon method to child thread

    somesh.setDaemon(True)
```

```
    somesh.start()

    time.sleep(1)
#add sequence of print statements by parent thread on the count of 0.8
seconds
    print('student2 is done with cooking!') #This is the last
instruction to be executed
```

- Usage and purpose:

  Well, when have to terminate the first process running after the second process gets terminated, we require daemon to kill that first thread.

  A daemon process is a background process that is not under the direct control of the user. This process is usually started when the system is bootstrapped and is terminated with the system shut down. Usually, the parent process of the daemon process is the init process. Daemons are processes that are often started when 1st process does not gets terminated.

  In this task we have room_cleaner() method which prints 1st method till 1 sec.

  In the main method 2nd thread is been initiated while in parallel with 1st thread and daemon is being initiated their role and gets terminated.

- Output of Windows:

```
runfile('C:/Users/Sheikh Tadeeb/Downloads/task5_AU19B1009.py',
wdir='C:/Users/Sheikh Tadeeb/Downloads')
Task5__doc__:
 Assignment -III Task V: student1 cleans a hostel room meanwhile
student2 Cooks a food analogy
Cleaning Started
student2 is done with cooking!Cleaning Started

Cleaning Started
Cleaning Started
Cleaning Started… it will continue
```

- Output of Linux:

```
tadeeb@tadeebsUbuntu:~$ python3 task5_AU19B1009.py
Task5__doc__:
```

```
Assignment -III Task V: student1 cleans a hostel room meanwhile
student2 Cooks
a food analogy
hostel is been cleaning
student2 is done with cooking!
```

- Inference:

  On running the code on both the platforms Linux and windows the 1st thread doesn't get terminated in windows platform on still running the daemon while in Linux the 1st thread gets terminated on the successful completion of 2nd thread this is just because windows doesn't understood the termination process as once it gets run it consumes the memory space while in Linux it gets terminated once it gets completed.