Threads

- Introduction
- Single tasking
- Multitasking
- Difference between Process and Threads
- Concurrent Programming and GIL
- Creating Threads
- Thread class methods
- Thread Synchronization
- Communication between threads
- Daemon Threads

Introduction

A thread represents a separate path of execution of a group of statements

In every python program, there is always a thread running internally which is appointed by the PVM to execute the program statements

In Python program, we need to import **'threading'** module which is needed while dealing with threads.

current_thread(): returns an object containing the information about the presently running thread in the program

```
name = current_thread().getname()          # return the name of current thread
```

The 'name' contains the name of the internal class of Python which is **'MainThread'**

We can also access the main thread's information by calling **'main_thread()'** method

Program 1: Program to find the currently running thread in Python Program

```
# Every Python program is run by main thread
import threading

print('Let us find the current thread')

# Find the name of the present thread
print('Currently Running Thread:', threading.current_thread().getName())

# Check if it is main thread or not
if threading.current_thread() == threading.main_thread():
  print('The current thread is the main thread')
```

```
else:
  print('The current thread is not the main thread')
```

```
Let us find the current thread
Currently Running Thread: MainThread
The current thread is the main thread
```

From above program we can conclude that there is always a default thread running called 'MainThread' that executes program statement.

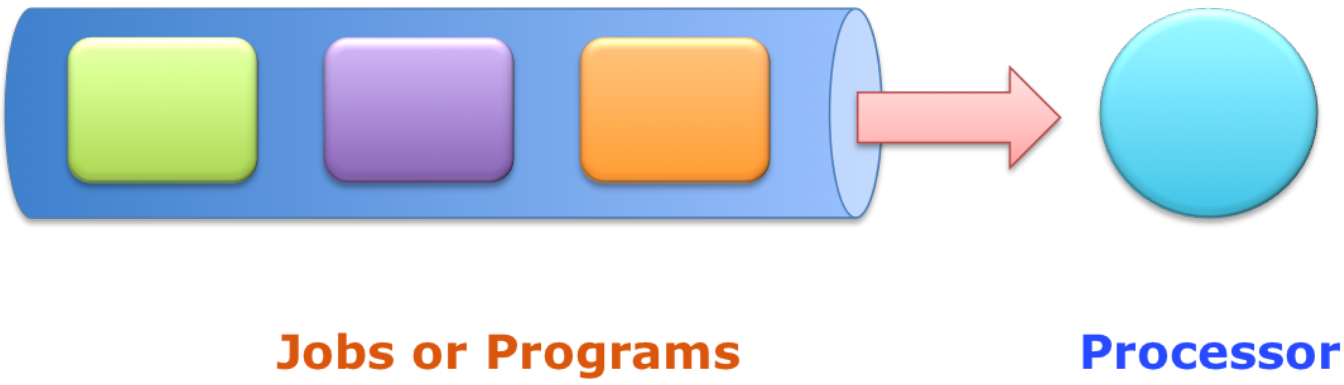
The way the statements are executed is of two types:

- Single Tasking
- Multitasking

Single Tasking

Task: Doing calculations or operations which involves execution of group of statement

Single Tasking: Only one task is given to the processor at a time



Advantage:

> The system where task should be executed one by one without time gap, the above system is preferable

Drawback:

> A lot of processor time is wasted

> Microprocessor has to sit idle without any job for a long time

Multitasking

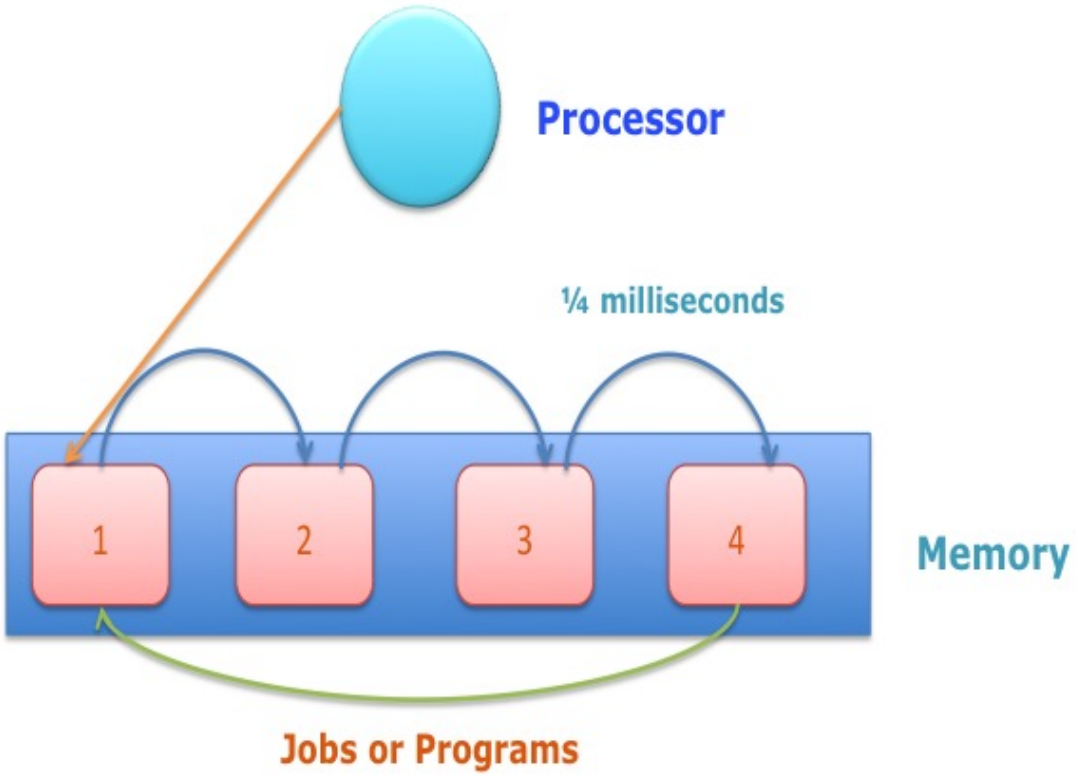
To use processor's time in optimum way, we can give it several jobs at a time. This is called Multitasking

Consider following scenario:

- There are four tasks to be executed.
- The memory is divided into four parts. All the four jobs are loaded into memory
- The processor will divide time unit into 4 equal parts known as time slice. (Here it is 1/4 milliseconds)
- Processor will serve each job for time slice
- After serving all the jobs it will visit first job if it is not completed.
- The cycle repeats until all the jobs get completed

This mechanism is known as **'Round Robin Method'**

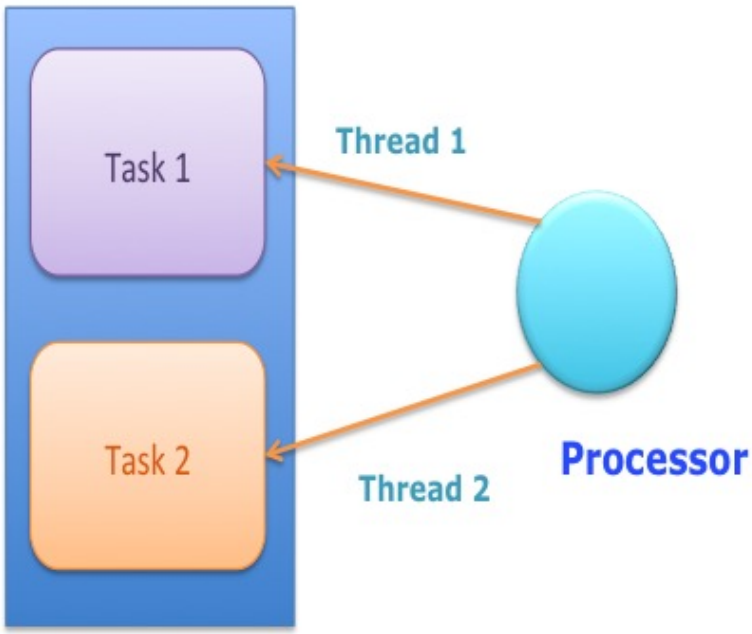
Above scenario gives a feel that all four processes are executing simultaneously. This is called **Multitasking**



Process-based Multitasking

We have only one processor in computer system.

One processor has to execute several tasks means that it has to execute them in a round robin method.



Thread-based Multitasking

Difference between a Process and a Thread

Process	Thread
Represents group of a statement which are executed by PVM using main thread	Represents group of statements within program
Have their own memory and program counter	Does not have their own memory and program counter
Data of one process is isolated from another process	Data of one thread is shared by another thread
Known as Heavy weight process as they are consuming memory and processor time	Known as Light weight Process as occupying very less memory and less time

Concurrent Programming and GIL

Concurrent Programming:

> It is possible to create multiple processes and set them to work simultaneously

> Similarly, multiple threads can be created and set them to work on different parts of the program simultaneously

> Executing the tasks or parts of program simultaneously is called **Concurrent Programming**

Global Interpreter Lock (GIL)

> PVM is **not thread safe** as more than one thread can act on same data simultaneously

> Therefore PVM uses an internal Global Interpreter Lock (GIL) that allows only a single thread to execute at any given moment

> GIL does not allow more than one thread to run at a time

> This is the **obstacle** to write concurrent program in Python

> GIL **will not impose** this restriction of using only one thread at a time in case of **normal python program** that take some input and provide output

> GIL will **impose** this restrictions on the application that involves **heavy amount of CPU processing** or those involving **multiple processors**

Uses of Threads

Threads are useful in **Concurrent Programming** (whenever we want to perform more than one tasks simultaneously)

In following situations threads can be used:

Server Side Programming:

> To serve the multiple clients over network or Internet.

> Threads are used at server side so that they can do multiple jobs at a time

To create game and animation:

> In games we have to perform many actions at same time. Animation is related with moving objects from one place to another

> **Example**: In a game there is situation where flight is moving from left to right and machine gun is firing bullets. So we need two threads to implement those two

> actions

Creating Threads in Python

Python provides '**Thread**' class of **threading** module that is useful to create threads

To create user defined thread, we need to create an object of Thread class.

There are different ways of creating threads in Python as:

- Creating a thread without using a class
- Creating a thread by creating a sub class to Thread Class
- Creating a thread without creating sub class to Thread Class

Creating Thread without using Sub class

The purpose of a thread is to execute a group of statement like a function.

We can create a thread by creating an object of Thread class and pass the function name as target for the thread as:

```
t = Thread(target = functionname, [args = (arg1, arg2, ...)]
```

Here,

> 't' represents objects of Thread class.

> 'target' represents the function on which thread will act

> 'args' represents a tuple of arguments which are passed to the function

Thread can be started by calling start( ) method as

```
t.start()
```

Then thread will jump to the target function and execute the code

---

Program 2: A Python program to create and use it to run a function

```
# Creating a thread without using a class
from threading import *

# create a function
def display():
  print('Hello I am running')
```

```
# Create a thread and run the function for 5 times
for i in range(5):
  t = Thread(target = display)     # Create the Thread and specify the function as target
  t.start()                        # Run the Thread
```

```
Hello I am running
Hello I am running
Hello I am running
Hello I am running
Hello I am running
```

Program 3: Program to pass arguments to a function and execute using thread

Here we are passing a string to display() function at the time of creating a thread

```
# Creating a thread without using a class (Version 2)
from threading import *

# create a function
def display(str):
  print(str)

# Create a thread and run the function for 5 times
for i in range(5):
  t = Thread(target = display, args = ('Hello',))
  t.start()
```

```
Hello
Hello
Hello
Hello
Hello
```

Creating a Thread by creating a Sub class to Thread Class

The Thread class is already created by Language designer.

Step 1: Create sub class of Thread Class

We can make our class as a sub class to Thread class so that all the properties of Thread class are inherited.

This can be done with following statement

```
class MyThread(Thread):
```

Step 2: Override run( ) method of Thread Class

There is run() method defined in the Thread class.We can **override run( ) method** to create our own thread.

Overriding means - Writing our own method with the same name as the run( ) method of the super class

Step 3: To create object of Sub class:

Create the object of class 'MyThread' which has all the properties of Super class

```
t1 = MyThread()
```

Step 4: Execute start( ) method

We can run the thread by calling the start( ) method as

```
t.start()
```

The thread will jump into user defined run() method and start executing the code inside it

Step 5: Wait for completion of execution of Thread

It is better to wait for the completion of thread execution by calling the join method on thread as

```
t.join()
```

This will wait till the thread completely executes the run( ) method

A thread will terminate automatically when it will comes out of the run ( ) method

---

Program 4: Program to create a thread by making our class as sub class to Thread class

```
# Creating our own thread
from threading import Thread

# Create a class as sub class to Thread class
class MyThread(Thread):

  # Override the run() method of Thread class
  def run(self):
    for i in range(1,6):
      print(i)

# Create an instance of MyThread class
t1 = MyThread()

# start running the thread t1
t1.start()

# Wait till the thread completes execution
t1.join()
```

```
1
2
3
4
5
```

Program 5: Program to create a thread that accesses the instance variable of a class

```
# A Thread that accesses the instance variable
from threading import *

# Create a class as sub class to Thread class
class MyThread(Thread):

  # Constructor that calls Thread class constructor
  def __init__(self,str):
    Thread.__init__(self)
    self.str = str

  # override the run() method of Thread class
  def run(self):
    print(self.str)

# Create an instance of MyThread class and pass the string
t1 = MyThread('Hello')

# start running the thread t1
t1.start()

# wait till the thread completes execution
t1.join()
```

```
Hello
```

Creating a Thread without creating Sub Class to Thread class

Step 1: Create object of an Independent class

Create object of a class that is not inherited from Thread class

```
obj = MyThread('Hello')
```

Step 2: Create an object of Thread class and specify the method of independent class as target

The thread is created by creating object of the Thread class.

The method of the class MyThread (Independent class) is passed as target to this thread object

```
t1 = Thread(target = obj.display, args=(1,2))
```

Here t1 is Thread class object

target represents display( ) method of MyThread object 'obj'

'args' represents tuple of values passed to the method

Program 6: Program to create a thread that acts on the objects of a class that is not derived from Thread class

```
# Creating a thread without making sub class to Thread class
from threading import *

# create our own class
class MyThread:

  # Constructor
  def __init__(self,str):
    self.str = str

  # Method
  def display(self, x, y):
    print(self.str)
    print('The arguments are',x,y)

# Create an instance of our own class and store 'Hello' string
obj = MyThread('Hello')

# Create a thread to run display method of obj
t1 = Thread(target = obj.display, args = (1,2))

# Run the thread
t1.start()
```

```
Hello
The arguments are 1 2
```

Thread Class Methods

Here, t represents the Thread class object

Method or Property	Description
t.start( )	• Starts the thread. • It must be called at most once per thread object • It arranges for the object's run() method to be invoked in a separate thread of control
t.join([timeout])	• Waits until the thread terminates or a timeout occurs • 'timeout' is a floating point number specifying a timeout for the operation in seconds (or fraction of seconds)
t.is_alive( )	• Returns True if the thread is alive in the memory and False otherwise • A thread is alive from the moment the start( ) method returns until its run( ) method terminates
t.setName(name)	• Gives a name to the thread
t.getName( )	• Returns name of the thread
t.name	• This is a property that represents the thread's name
t.setDaemon(flag)	• Makes a thread a daemon thread if the flag is True
t.isDaemon( )	• Returns True if the Thread is a daemon thread, otherwise False
t.daemon	• This is a property that takes either True or False to set the thread as daemon or not

Single Tasking using a Thread

A Thread can be employed to execute only one task at a time

Suppose there are three tasks executed by a thread one by one. This is called 'Single Tasking'

Plan for preparation of Tea

Task 1: Boil Milk and Tea Powder for 5 minutes

Task 2: Add sugar and boil for 3 Minutes

Task 3: Filter and Serve

All the three tasks can be put one by one in the main method as

```
def prepareTea(self):
    self.task1()
    self.task2()
    self.task3()
```

We have to make this **main method as the target for the Thread**

In this program, the time is shown with the help of the sleep() function of the *time* module as

```
sleep(seconds)
```

Program 7: Program to show single tasking using a thread that prepares tea

```
# Single tasking using a single thread
from threading import *
from time import *

# Create our own class
class MyThread:

  # a method that performs 3 tasks one by one
  def prepareTea(self):
    self.task1()
    self.task2()
    self.task3()

  def task1(self):
    print('Boil milk and Tea powder for 5 minutes...', end='')
    sleep(6)
    print('Done')

  def task2(self):
    print('Add sugar and boil for 3 minutes...', end='')
    sleep(3)
    print('Done')

  def task3(self):
    print('Filter it and serve ...', end='')
    print('Done')

# create an instance to our class
obj = MyThread()

# create a thread and run prepare Tea method of obj
t = Thread(target=obj.prepareTea)
t.start()

t.join()
```

```
Boil milk and Tea powder for 5 minutes...Done
Add sugar and boil for 3 minutes...Done
Filter it and serve ...Done
```

Multitasking using Multiple Threads

Several tasks are executed at a time. For that purpose we need more than one thread

To perform two tasks we require two threads and attach them these two tasks

Now these two tasks are executed simultaneously.

Using more than one thread is called multithreading which is used in Multitasking

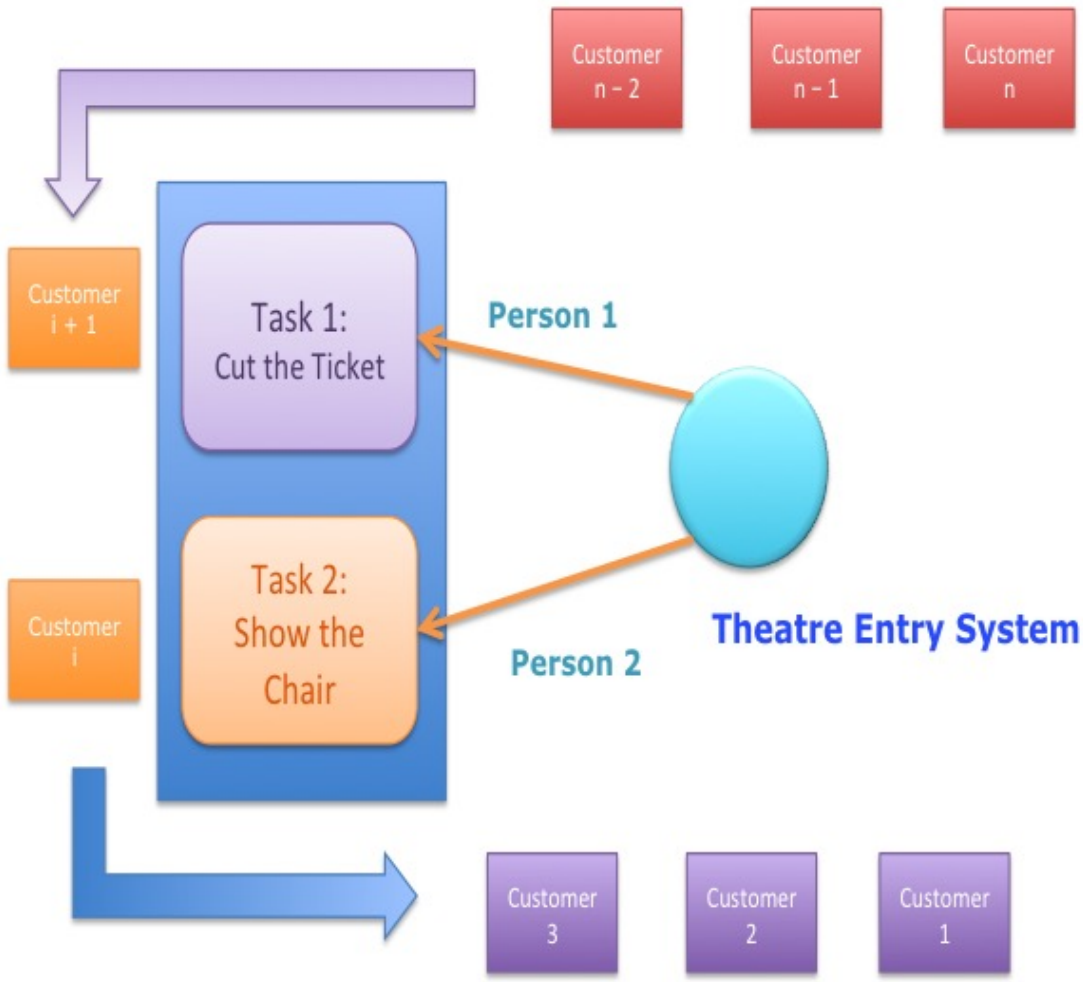
Example:

In Movie Theatre at entry, we have two persons (Two threads)

1. Checking and Cutting the tickets
2. Show the chair

If only one person is doing both the tasks then it will take more time.

With two persons working at same time, the system can efficiently work and could finish the task in less time



Program 8: Program that performs two tasks using two threads simultaneously

```
# Multitasking using two threads
from time import *

class Theatre:
  # constructor that accepts a string
  def __init__(self, str):
    self.str = str

  # a method that repeats for 5 tickets
  def movieshow(self):
    for i in range(1,6):
      print(self.str,":",i)
      sleep(0.1)
```

```
# Create two instance to Theatre class
obj1 = Theatre('Cut Ticket')
obj2 = Theatre('Show Chair')

# Create two threads to run moviesshow()
t1 = Thread(target=obj1.movieshow)
t2 = Thread(target=obj2.movieshow)

# run the threads
t1.start()
t2.start()

# wait to end thread
t1.join()
t2.join()
```

```
     Cut Ticket : 1
     Show Chair : 1
     Cut Ticket : 2
     Show Chair : 2
     Cut Ticket : 3
     Show Chair : 3
     Cut Ticket : 4
     Show Chair : 4
     Cut Ticket : 5
     Show Chair : 5
```

Sometimes the results are absurd and varying in above execution.

We may get the contrary output such as t2 is responding before t1.

It is called as **Race Condition** where threads are not acting in an expected sequence that leads to unreliable output.

Race condition can be eliminated using '**Thread Synchronization**'

Example of Railway Reservation System

No. of births for reservation: 1

Two passengers (Threads) are asking for berth.

Two threads are competing for same resource

Program 9: Program where two threads are acting on the same method to allot a berth from passenger

```
# Multitasking using two threads
from threading import *
from time import *

class Railway:
```

```
  # Constructor that accepts no of available berth
  def __init__(self, available):
    self.available = available

  # A method that reserves berth
  def reserve(self, wanted):
    print('Available no of berths:',self.available)    # Display No of available berths

    if(self.available >= wanted):                      # If condition satisfies then allot
      name = current_thread().getName()                # Find the thread name
      print("%d berths allotted for %s"%(wanted,name)) # Display berth is allotted for the
      sleep(1.5)                                       # make time delay so that ticket is
      self.available -= wanted                         # Decrease the no of available bert

    else:
      print('Sorry, No berths to allot')               # if available < wanted then say so

# Create instance to Railway Class
# Specify only one berth is available
obj = Railway(1)

# Create two threads and specify 1 berth is needed
t1 = Thread(target=obj.reserve, args=(1,))
t2 = Thread(target=obj.reserve, args=(1,))

# Give names to the threads
t1.setName('First Person')
t2.setName('Second Person')

# run the threads
t1.start()
t2.start()

# Wait for completion of execution
t1.join()
t2.join()
```

```
Available no of berths: 1
1 berths allotted for First Person
Available no of berths: 1
1 berths allotted for Second Person
```

The above output is **not correct**

Whenever thread t1 execute method reserve it gets the 'available = 1' therefore it allot berth to first person and waits for printing of ticket.

Before 'available' is made 0 thread t2 executes reserve method and find the value of 'available = 1'. So it also allot the berth to second passenger.

It leads to unreliable result.

The problem arises due to shared resources are accessed simultaneously by multiple threads.

Solution: to allow only one thread to access the shared resource at a time

Thread Synchronization

When a thread is already acting on object preventing any other thread from acting on the same object is called **Thread Synchronization**

Synchronized Object/ Mutex (Mutually Exclusive Lock): The object on which threads are synchronized

Thread synchronization is recommended when multiple threads are acting on same object simultaneously

It is implemented using following techniques:

1. Using Locks
2. Using Semaphores

Locks

It can be used to lock the object on which thread is acting.

When thread enters the object, it locks the object.

When execution gets completed, it will unlock the object and it will comes out of it

We can create lock by creating an object of lock class as

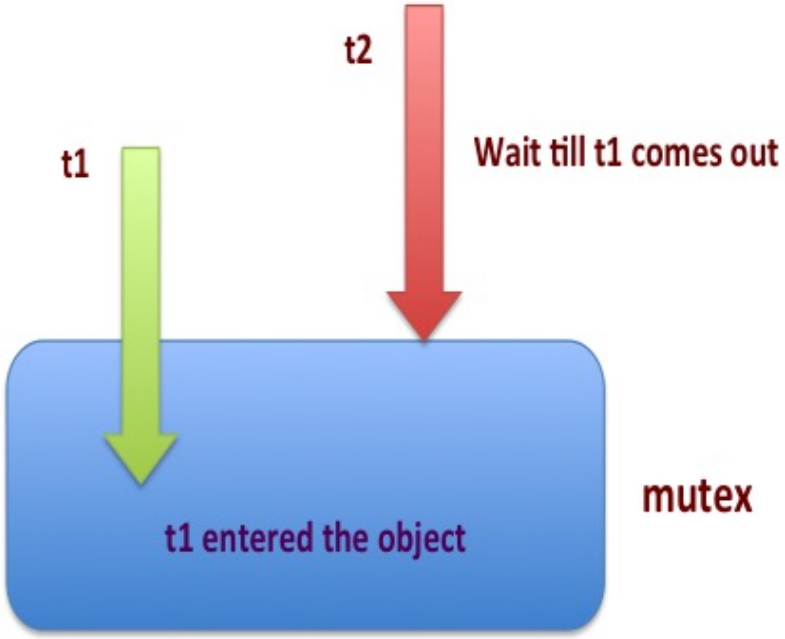
```
l = lock()
```

To lock the current object, we should use acquire() method as

```
l.acquire()
```

To unlock or release the object, we can use release () method as

```
l.release()
```



Thread Synchronization

Program 10: Python program displaying thread synchronization using locks

```
# Thread synchronization using locks
from threading import *
from time import *

class Railway:

  # Constructor that accepts no of available berth
  def __init__(self, available):
    self.available = available
    self.l = Lock()           # Create lock object

  # A method that reserves berth
  def reserve(self, wanted):

    self.l.acquire()                                    # Lock the current object ***
    print('Available no of berths:',self.available)     # Display No of available berths

    if(self.available >= wanted):                       # If condition satisfies then allot
      name = current_thread().getName()                 # Find the thread name
      print("%d berths allotted for %s"%(wanted,name)) # Display berth is allotted for the
      sleep(1)                                         # make time delay so that ticket is p
      self.available -= wanted                          # Decrease the no of available bert
```

```
    else:
      print('Sorry, No berths to allot')              # if available < wanted then say so

    self.l.release()                                  # Task is completed, release the lo


# Create instance to Railway Class
# Specify only one berth is available
obj = Railway(1)

# Create two threads and specify 1 berth is needed
t1 = Thread(target=obj.reserve, args=(1,))
t2 = Thread(target=obj.reserve, args=(1,))

# Give names to the threads
t1.setName('First Person')
t2.setName('Second Person')

# run the threads
t1.start()
t2.start()

t1.join()
t2.join()
```

```
Available no of berths: 1
1 berths allotted for First Person
Available no of berths: 0
Sorry, No berths to allot
```

Semaphores

It is an object that provides synchronization based on the counter.

A semaphore is created as an object of Semaphore class as:

```
l = Semaphore(Countervalue)   # Default Countervalue is 1
```

If the counter value is not given then default value of counter is 1

When the acquire() method is called, the counter gets decremented by 1.

When release( ) method is called, the counter gets incremented by 1.

These methods are used in the following format:

```
l.acquire()   # Make counter 0 and then lock
# Code that is locked by semaphore
l.release()
```

Deadlock of Threads

With the use of Mutually Exclusive locks, there ia possibility of Deadlock

There are two locks each for train and compartment.

There is the possibility that the two locks used by each thread in the following format:

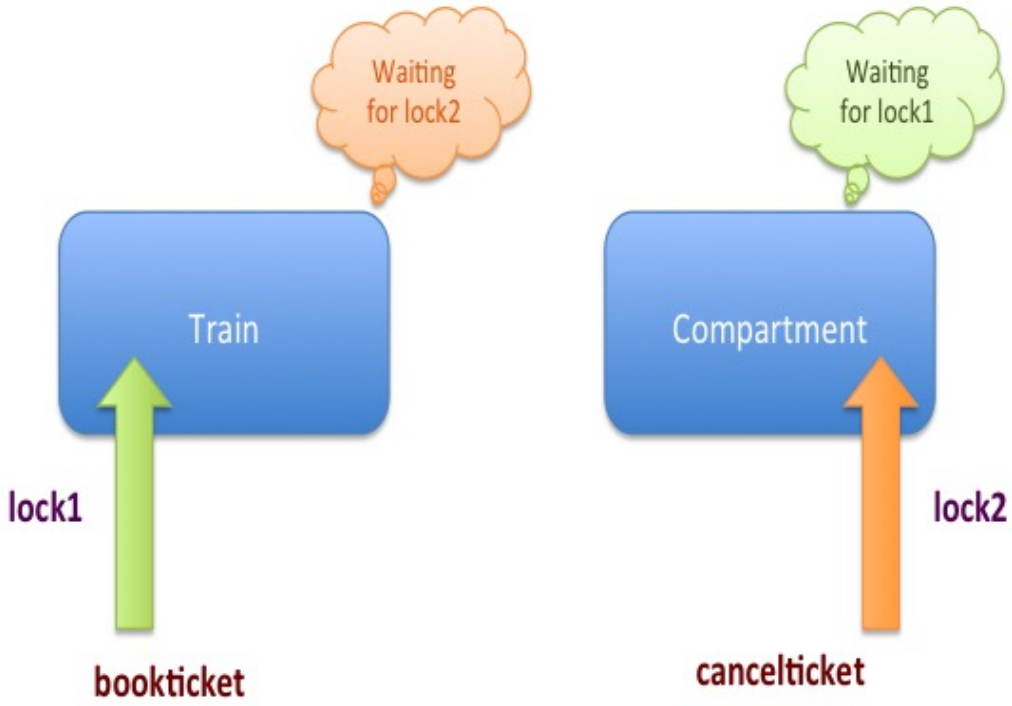
```
# Bookticket Thread
lock-1:
lock on train
    lock-2:
    lock on compartment

# Cancellation Thread
lock-2:
lock on compartment
    lock-1:
    lock on train
```

In above situation 'bookticket' thread will lock train and it will wait for compartment to be released

And cancellation thread will lock compartment and it will wait for Train.

So both the threads are waiting for each other forever.

This situation is called as 'Deadlock'



Deadlock of Threads

Program 11: Program to show deadlock of threads due to locks on objects

```
# Dead lock of Threads
from threading import *
from time import *

# Take two locks
l1 = Lock()
l2 = Lock()

# Create a function for booking a ticket
def bookticket():
  l1.acquire()
  print('Bookticket Locked Train')
  print('Bookticket wants to lock on compartment')
  sleep(1.5)
  l2.acquire()
  print('Bookticket locked compartment')
  l2.release()
  l1.release()
  print('Booking Ticket done ...')


# Create a function for cancelling a ticket
```

```
    def cancelticket():
      l2.acquire()
      print('Cancelticket Locked compartment')
      print('Cancelticket wants to lock on train')
      sleep(1.5)
      l1.acquire()
      print('Cancelticket locked Train')
      l1.release()
      l2.release()
      print('Cancellation of Ticket is done ...')


    # Create two threads and run them
    t1 = Thread(target=bookticket)
    t2 = Thread(target=cancelticket)

    t1.start()
    t2.start()
```

Avoiding Deadlocks in a Program

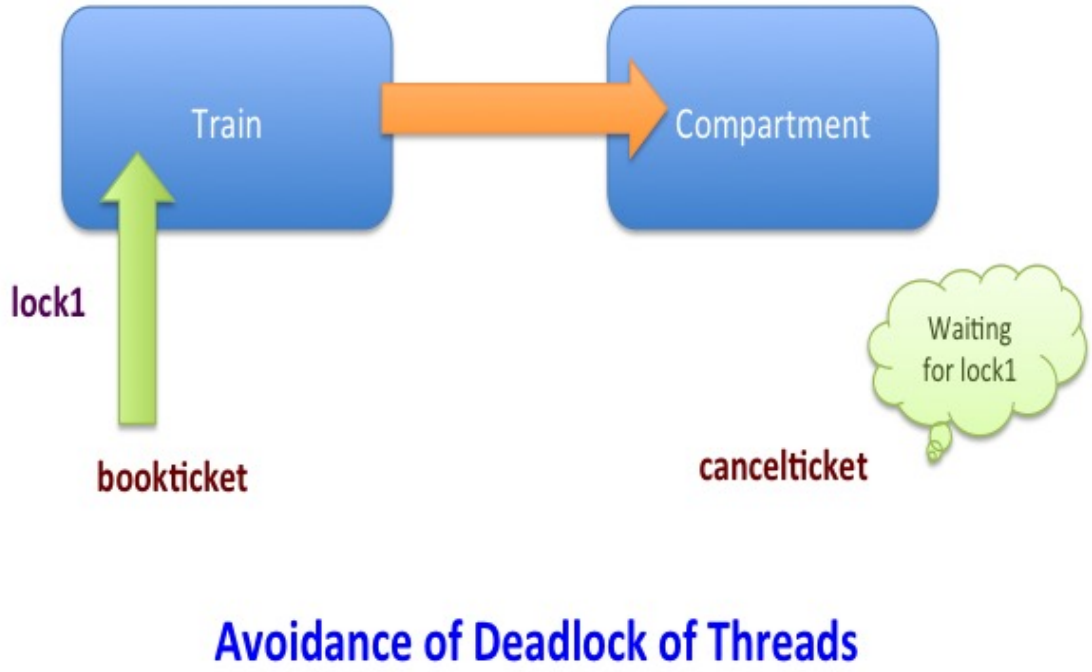
There is no specific solution for the problem of deadlocks.

It depends on the logic used by the programmer.

The programmer should design program such that it does not form any deadlock.

In the previous program, if we use locks in following format then deadlock can be avoided

```
 # Bookticket Thread
 lock-1:
 lock on train
     lock-2:
     lock on compartment

 # Cancellation Thread
 lock-1:
 lock on compartment
     lock-2:
     lock on train
```



Program 12: A python program with good logic to avoid deadlock

```
# Dead lock of Threads
from threading import *
from time import *

# Take two locks
l1 = Lock()
l2 = Lock()

# Create a function for booking a ticket
def bookticket():
  l1.acquire()
  print('Bookticket Locked Train')
  print('Bookticket wants to lock on compartment')
  sleep(1.5)
  l2.acquire()
  print('Bookticket locked compartment')
  l2.release()
  l1.release()
  print('Booking Ticket done ...')


# Create a function for cancelling a ticket
```

```
def cancelticket():
  l1.acquire()
  print('Cancelticket Locked compartment')
  print('Cancelticket wants to lock on train')
  sleep(1.5)
  l2.acquire()
  print('Cancelticket locked Train')
  l2.release()
  l1.release()
  print('Cancellation of Ticket is done ...')


# Create two threads and run them
t1 = Thread(target=bookticket)
t2 = Thread(target=cancelticket)

t1.start()
t2.start()

t1.join()
t2.join()
```

Communication between Threads

In some cases, communication between thread is required.

Example: **Producer-Consumer problem**

> Consumer is waiting for a Producer to produce the data

> When Producer completes the production of the data, then Consumer should take the data and use it

> The Consumer does not know how much time it will take to produce the data

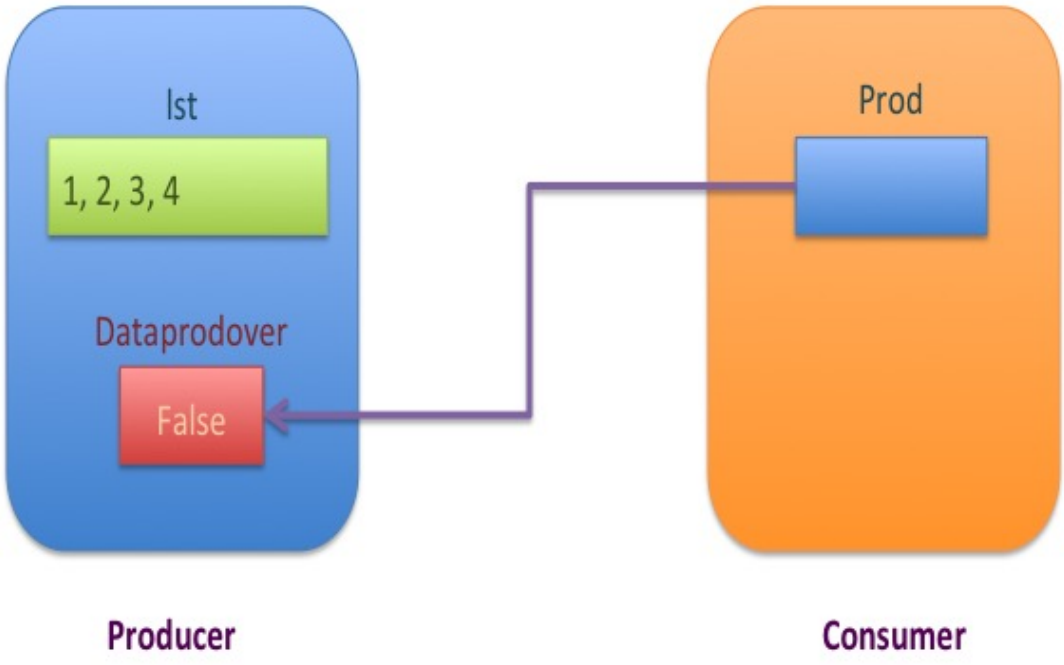
How to inform Consumer that Data is ready??

> Possible Solution: Create one **boolean variable** which is initially set to False.

> When Data is ready, Producer can set this variable to 'True'.

> Consumer is checking the value of Boolean variable in fixed interval of time. It will wait until the variable is False.

> When Consumer gets the value as True, it will start consuming the data



Communication between Producer and Consumer Thread

Program 13: A Python program where Producer and Consumer thread communicate with each other through a boolean type variable

```
# thread communication
from threading import *
from time import *

# Create Producer class
class Producer:
  def __init__(self):
    self.lst = []
    self.dataprodover = False

  def produce(self):
    # create 1 to 10 items and add to the list
    for i in range(1,11):
      self.lst.append(i)
      sleep(1)
      print('Item produced...')

    self.dataprodover = True    # Inform consumer that the data production is over
    print('Production done')

# Create Consumer class
```

```
class Consumer:
  def __init__(self,prod):
    self.prod = prod

  def consume(self):

    # sleep for 100 ms as long as dataprodover is False
    while self.prod.dataprodover == False:
      print('consumer waiting...')
      sleep(1)

    # display the content of list when data production is over
    print(self.prod.lst)

# Create Producer object
p = Producer()

# Create Consumer object and pass Producer object
c = Consumer(p)

# create producer and consumer threads
t1 = Thread(target=p.produce)
t2 = Thread(target=c.consume)

# run the threads
t1.start()
t2.start()

t1.join()
t2.join()
```

```
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
consumer waiting...
consumer waiting...
Item produced...
consumer waiting...
Item produced...
Production done
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This is not the efficient way of communication

Consumer is waiting for fixed interval of time if boolean variable is False. Let us say the waiting time is 100 ms.

Now if Consumer thread goes to sleep state and at the next time frame data production is ready then there are chances of delay of 99 miliseconds.

So this is waste of time and performance is degraded

Thread Communication using notify() and wait() Methods

The Condition class of *threading* module is useful to improve speed of communication between the threads.

The condition class object is called condition variable and is created as

```
cv = Condition()
```

This class contains methods which are used in thread communication.

> **notify( ) :** To inform the consumer thread that the data production is completed

> **notify_all( ) :** To inform all the waiting consumer threads at once that the production is over

These methods are used by producer thread

The consumer thread need not to check boolean variable for getting status of data production.

It can wait till it it gets notification from notify() and notify_all() method

> **wait():** Consumer can wait using this method which terminates when Producer invokes notify() or notify_all() methods

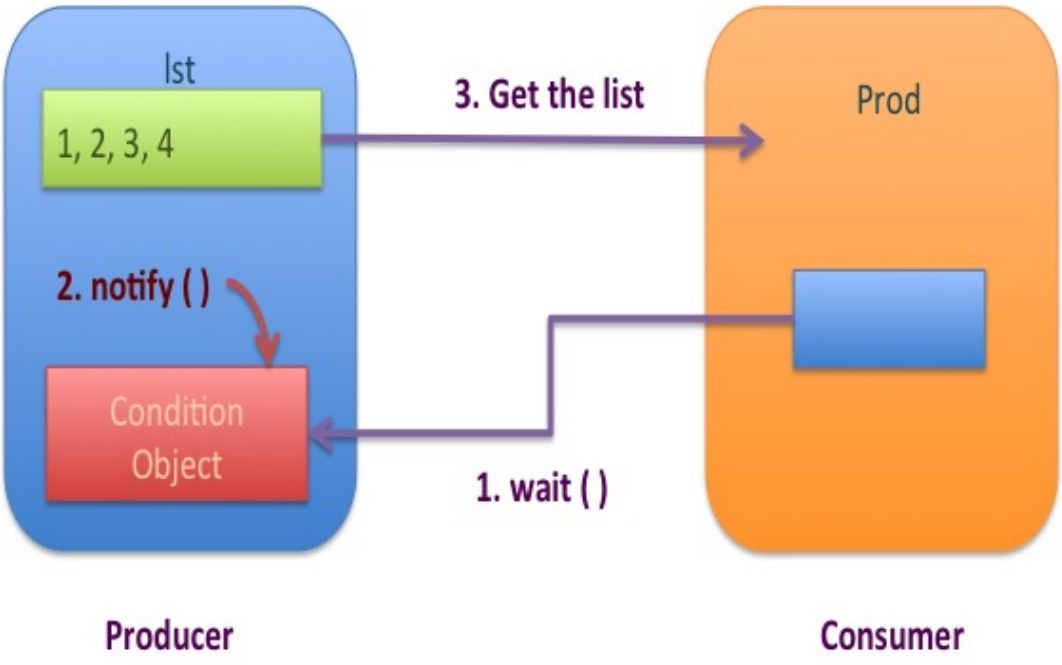
It is written with following syntax:

```
cv.wait(timeout=0)
```

Above code will wait till notification. As soon as notification is received, it will start receiving the data as timeout value is zero

Note:

The methods notify(), notify_all(), wait() should be used in between the acquire() and release() methods which are useful to lock and unlock conditional variables



Producer – Consumer communication using Condition object

Program 14: A Python program where communication is done through notify( ) and wait() method of condition object

```
# thread communication using Condition object
from threading import *
from time import *

# Create Producer class
class Producer:
  def __init__(self):
    self.lst = []
    self.cv = Condition()

  def produce(self):
    # Lock the condition object
    self.cv.acquire()

    # create 1 to 10 items and add to the list
    for i in range(1,11):
      self.lst.append(i)
      sleep(1)
      print('Item produced...')

    # Inform consumer that the data production is over
```

```
        self.cv.notify()
        print('Production done')

       # Release the lock
        self.cv.release()


# Create Consumer class
class Consumer:
  def __init__(self,prod):
    self.prod = prod

  def consume(self):

    # get lock on condition object
    self.prod.cv.acquire()

    # wait only for 0 seconds after production is over
    self.prod.cv.wait(timeout=0)

    # Release the lock
    self.prod.cv.release()

    # display the content of list when data production is over
    print(self.prod.lst)

# Create Producer object
p = Producer()

# Create Consumer object and pass Producer object
c = Consumer(p)

# create producer and consumer threads
t1 = Thread(target=p.produce)
t2 = Thread(target=c.consume)

# run the threads
t1.start()
t2.start()

t1.join()
t2.join()
```

Thread Communication using a Queue

The Queue class of queue model is useful to create a queue that holds the data produced by producer.

The data can be taken from the queue and utilized by Consumer

Queue is FIFO structure (First in First Out)

Advantages:

1. It is useful when there are several producers want to communicate with server
2. Queues are thread safe so no need of any lock

To create Queue object

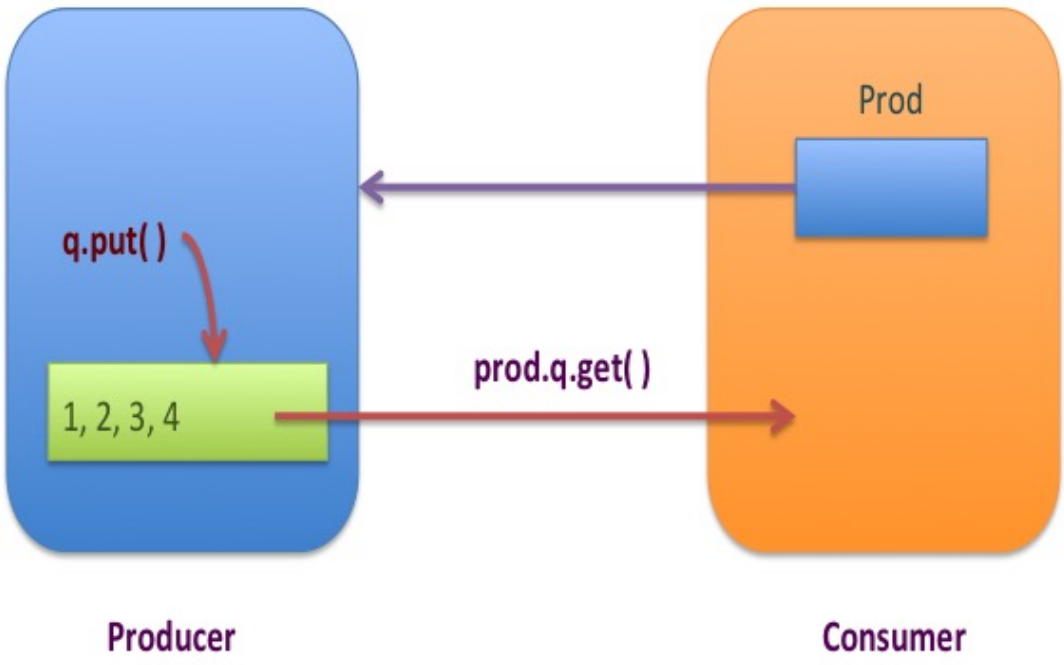
```
q = Queue()
```

To insert an item i into the queue 'q'. The method is used as

```
q.put(i)
```

The consumer uses the get() methods

```
x = prod.q.get()
```



Producer – Consumer communication using a Queue

Program 15: A Python program that uses a queue in thread communication

```
# thread communication using Queue
from threading import *
from time import *
```

```
from queue import *

# Create Producer class
class Producer:
  def __init__(self):
    self.q = Queue()

  def produce(self):

    # create 1 to 10 items and add to the list
    for i in range(1,11):
      print('Producing item...',i)
      self.q.put(i)
      sleep(1)



# Create Consumer class
class Consumer:
  def __init__(self,prod):
    self.prod = prod

  def consume(self):

    # receive 1 to 10 items from the queue
    for i in range(1,11):
      print('Receiving item ',self.prod.q.get(i))

# Create Producer object
p = Producer()

# Create Consumer object and pass Producer object
c = Consumer(p)

# create producer and consumer threads
t1 = Thread(target=p.produce)
t2 = Thread(target=c.consume)

# run the threads
t1.start()
t2.start()

t1.join()
t2.join()
```

```
Producing item... 1
Receiving item  1
Producing item... 2
Receiving item  2
Producing item... 3
Receiving item  3
Producing item... 4
Receiving item  4
Producing item... 5
Receiving item  5
Producing item... 6
```

```
Receiving item  6
Producing item... 7
Receiving item  7
Producing item... 8
Receiving item  8
Producing item... 9
Receiving item  9
Producing item... 10
Receiving item  10
```

Daemon Thread

Some of the threads need to be run continuously in memory are called as **Daemon Threads**

Example:

> **Internet Server:** Need to cater clients continuously

> **Garbage Collector:** Runs continuously to delete unused variables and objects while Python Program is being executed

Generally they are used to perform background tasks.

Example:

> A Daemon thread may send data to the printer in the background while the user is working with other application

To make a thread daemon thread, we can use the setDaemon(True) method or assign True to daemon property

---

Program 16: A Python program to understand the creation of daemon thread

Here, we are going to create function **display( )** that displays number once in a second, starting from 1 to 5

The function is executed by **normal thread** 't'

There is another function **display_time()** that displays system date and time in every 2 seconds

This function is executed by another thread 'd' which is made **daemon thread**

```
# creating daemon thread
from threading import *
from time import *

# To display numbers from 1 to 5 every second
def display():
  for i in range(5):
    print('Normal thread:',end='')
    print(i+1)
    sleep(1)
```

```
# To display date and time once in every 2 seconds
def display_time():
  while True:
    print('Daemon Thread:',end='')
    print(ctime())
    sleep(2)

# create a normal thread and attach it to display() and run it
t = Thread(target=display)
t.start()

# create another thread and attach it to display_time()
d = Thread(target=display_time)

# make the thread daemon
d.daemon = True

# run the daemon thread
d.start()

# Wait till Normal thread completes its execution
t.join()

# For Normal Execution output will end when Thread t is ended

# If you run the program with statement [d.join()] uncommented
# then it wait for daemon thread to finish (Runs forever)
# You have to stop program manually
d.join()
```

```
Daemon Thread:Fri Apr 30 06:50:12 2021
Normal thread:1
Daemon Thread:Fri Apr 30 06:50:12 2021
Normal thread:2
Daemon Thread:Fri Apr 30 06:50:14 2021
Normal thread:3
Daemon Thread:Fri Apr 30 06:50:14 2021
Normal thread:4
Daemon Thread:Fri Apr 30 06:50:16 2021
Normal thread:5
Daemon Thread:Fri Apr 30 06:50:16 2021
Daemon Thread:Fri Apr 30 06:50:18 2021
Daemon Thread:Fri Apr 30 06:50:18 2021
Daemon Thread:Fri Apr 30 06:50:20 2021
Daemon Thread:Fri Apr 30 06:50:20 2021
Daemon Thread:Fri Apr 30 06:50:22 2021
Daemon Thread:Fri Apr 30 06:50:22 2021
Daemon Thread:Fri Apr 30 06:50:24 2021
Daemon Thread:Fri Apr 30 06:50:24 2021
Daemon Thread:Fri Apr 30 06:50:26 2021
Daemon Thread:Fri Apr 30 06:50:26 2021
Daemon Thread:Fri Apr 30 06:50:28 2021
Daemon Thread:Fri Apr 30 06:50:28 2021
Daemon Thread:Fri Apr 30 06:50:30 2021
Daemon Thread:Fri Apr 30 06:50:30 2021
Daemon Thread:Fri Apr 30 06:50:32 2021
Daemon Thread:Fri Apr 30 06:50:32 2021
Daemon Thread:Fri Apr 30 06:50:34 2021
Daemon Thread:Fri Apr 30 06:50:34 2021
Daemon Thread:Fri Apr 30 06:50:36 2021
Daemon Thread:Fri Apr 30 06:50:36 2021
Daemon Thread:Fri Apr 30 06:50:38 2021
Daemon Thread:Fri Apr 30 06:50:38 2021
```

In the above program when normal thread terminates, the main thread also terminates and hence program output is ended.

But suppose if you wait till daemon thread finishes its execution then program will run forever.

You need to stop the program manually in this case.

You can teest this by **uncommenting** last statement **d.join()**

```
<ipython-input-32-4c6fb3ae8c41> in <module>()
     38 # then it wait for daemon thread to finish (Runs forever)
     39 # You have to stop program manually
---> 40 d.join()
```

1 frames

```
/usr/lib/python3.7/threading.py in _wait_for_tstate_lock(self, block, timeout)
   1058         if lock is None:  # already determined that the C code is done
   1059             assert self._is_stopped
-> 1060         elif lock.acquire(block, timeout):
   1061             lock.release()
   1062             self._stop()

KeyboardInterrupt:
```

SEARCH STACK OVERFLOW