Multi Threading

Multi Tasking:

Executing several tasks simultaneously is the concept of multitasking.

There are 2 types of Multi Tasking
1. **Process based Multi Tasking**
2. **Thread based Multi Tasking**

1. Process based Multi Tasking:

Executing several tasks simmultaneously where each task is a seperate independent process is called process based multi tasking.

Eg: while typing python program in the editor we can listen mp3 audio songs from the same system. At the same time we can download a file from the internet. All these taks are executing simultaneously and independent of each other. Hence it is process based multi tasking.

This type of multi tasking is best suitable at operating system level.

2. Thread based MultiTasking:

Executing several tasks simultaneously where each task is a seperate independent part of the same program, is called Thread based multi tasking, and each independent part is called a Thread.

This type of multi tasking is best suitable at programmatic level.

Note: Whether it is process based or thread based, the main advantage of multi tasking is to improve performance of the system by reducing response time.

The main important application areas of multi threading are:
1. To implement Multimedia graphics
2. To develop animations
3. To develop video games
4. To develop web and application servers
etc...

Note: Where ever a group of independent jobs are available, then it is highly recommended to execute simultaneously instead of executing one by one.For such type of cases we should go for Multi Threading.

Python provides one inbuilt module "threading" to provide support for developing threads. Hence developing multi threaded Programs is very easy in python.

Every Python Program by default contains one thread which is nothing but MainThread.

Q.Program to print name of current executing thread:

```
1) import threading
2) print("Current Executing Thread:",threading.current_thread().getName())
```

o/p: Current Executing Thread: MainThread

Note: threading module contains function current_thread() which returns the current executing Thread object. On this object if we call getName() method then we will get current executing thread name.

The ways of Creating Thread in Python:

We can create a thread in Python by using 3 ways
1. Creating a Thread without using any class
2. Creating a Thread by extending Thread class
3. Creating a Thread without extending Thread class

1. Creating a Thread without using any class:

```
1) from threading import *
2) def display():
3)     for i in range(1,11):
4)         print("Child Thread")
5) t=Thread(target=display) #creating Thread object
6) t.start() #starting of Thread
7) for i in range(1,11):
8)     print("Main Thread")
```

If multiple threads present in our program, then we cannot expect execution order and hence we cannot expect exact output for the multi threaded programs. B'z of this we cannot provide exact output for the above program.It is varied from machine to machine and run to run.

Note: Thread is a pre defined class present in threading module which can be used to create our own Threads.

2. Creating a Thread by extending Thread class

We have to create child class for Thread class. In that child class we have to override run() method with our required job. Whenever we call start() method then automatically run() method will be executed and performs our job.

```
1) from threading import *
2) class MyThread(Thread):
```

```
3)     def run(self):
4)         for i in range(10):
5)             print("Child Thread-1")
6)  t=MyThread()
7)  t.start()
8)  for i in range(10):
9)     print("Main Thread-1")
```

3. Creating a Thread without extending Thread class:

```
1)  from threading import *
2)  class Test:
3)     def display(self):
4)         for i in range(10):
5)             print("Child Thread-2")
6)  obj=Test()
7)  t=Thread(target=obj.display)
8)  t.start()
9)  for i in range(10):
10)    print("Main Thread-2")
```

Without multi threading:

```
1)  from threading import *
2)  import time
3)  def doubles(numbers):
4)     for n in numbers:
5)         time.sleep(1)
6)         print("Double:",2*n)
7)  def squares(numbers):
8)     for n in numbers:
9)         time.sleep(1)
10)        print("Square:",n*n)
11) numbers=[1,2,3,4,5,6]
12) begintime=time.time()
13) doubles(numbers)
14) squares(numbers)
15) print("The total time taken:",time.time()-begintime)
```

With multithreading:

```
1)  from threading import *
2)  import time
3)  def doubles(numbers):
4)     for n in numbers:
5)         time.sleep(1)
6)         print("Double:",2*n)
7)  def squares(numbers):
8)     for n in numbers:
```

```
9)        time.sleep(1)
10)       print("Square:",n*n)
11)
12) numbers=[1,2,3,4,5,6]
13) begintime=time.time()
14) t1=Thread(target=doubles,args=(numbers,))
15) t2=Thread(target=squares,args=(numbers,))
16) t1.start()
17) t2.start()
18) t1.join()
19) t2.join()
20) print("The total time taken:",time.time()-begintime)
```

Setting and Getting Name of a Thread:

Every thread in python has name. It may be default name generated by Python or Customized Name provided by programmer.

We can get and set name of thread by using the following Thread class methods.

t.getName() ➔ Returns Name of Thread
t.setName(newName) ➔ To set our own name

Note: Every Thread has implicit variable "name" to represent name of Thread.

Eg:

```
1) from threading import *
2) print(current_thread().getName())
3) current_thread().setName("Pawan Kalyan")
4) print(current_thread().getName())
5) print(current_thread().name)
```

Output:
MainThread
Pawan Kalyan
Pawan Kalyan

Thread Identification Number (ident):

For every thread internally a unique identification number is available. We can access this id by using implicit variable "ident"

```
1) from threading import *
2) def test():
3)     print("Child Thread")
4) t=Thread(target=test)
```

```
5) t.start()
6) print("Main Thread Identification Number:",current_thread().ident)
7) print("Child Thread Identification Number:",t.ident)
```

Output:

```
Child Thread
Main Thread Identification Number: 2492
Child Thread Identification Number: 2768
```

active_count():

This function returns the number of active threads currently running.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)     print(current_thread().getName(),"...started")
5)     time.sleep(3)
6)     print(current_thread().getName(),"...ended")
7) print("The Number of active Threads:",active_count())
8) t1=Thread(target=display,name="ChildThread1")
9) t2=Thread(target=display,name="ChildThread2")
10) t3=Thread(target=display,name="ChildThread3")
11) t1.start()
12) t2.start()
13) t3.start()
14) print("The Number of active Threads:",active_count())
15) time.sleep(5)
16) print("The Number of active Threads:",active_count())
```

Output:

```
D:\python_classes>py test.py
The Number of active Threads: 1
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
The Number of active Threads: 4
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread3 ...ended
The Number of active Threads: 1
```

enumerate() function:

This function returns a list of all active threads currently running.

Eg:

```
1)  from threading import *
2)  import time
3)  def display():
4)      print(current_thread().getName(),"...started")
5)      time.sleep(3)
6)      print(current_thread().getName(),"...ended")
7)  t1=Thread(target=display,name="ChildThread1")
8)  t2=Thread(target=display,name="ChildThread2")
9)  t3=Thread(target=display,name="ChildThread3")
10) t1.start()
11) t2.start()
12) t3.start()
13) l=enumerate()
14) for t in l:
15)     print("Thread Name:",t.name)
16) time.sleep(5)
17) l=enumerate()
18) for t in l:
19)     print("Thread Name:",t.name)
```

Output:

```
D:\python_classes>py test.py
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
Thread Name: MainThread
Thread Name: ChildThread1
Thread Name: ChildThread2
Thread Name: ChildThread3
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread3 ...ended
Thread Name: MainThread
```

isAlive():

isAlive() method checks whether a thread is still executing or not.

Eg:

```
1)  from threading import *
2)  import time
3)  def display():
4)      print(current_thread().getName(),"...started")
5)      time.sleep(3)
6)      print(current_thread().getName(),"...ended")
7)  t1=Thread(target=display,name="ChildThread1")
8)  t2=Thread(target=display,name="ChildThread2")
```

```
9)  t1.start()
10) t2.start()
11)
12) print(t1.name,"is Alive :",t1.isAlive())
13) print(t2.name,"is Alive :",t2.isAlive())
14) time.sleep(5)
15) print(t1.name,"is Alive :",t1.isAlive())
16) print(t2.name,"is Alive :",t2.isAlive())
```

Output:

```
D:\python_classes>py test.py
ChildThread1 ...started
ChildThread2 ...started
ChildThread1 is Alive : True
ChildThread2 is Alive : True
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread1 is Alive : False
ChildThread2 is Alive : False
```

join() method:

If a thread wants to wait until completing some other thread then we should go for join() method.

Eg:

```
1)  from threading import *
2)  import time
3)  def display():
4)      for i in range(10):
5)          print("Seetha Thread")
6)          time.sleep(2)
7)
8)  t=Thread(target=display)
9)  t.start()
10) t.join()#This Line executed by Main Thread
11) for i in range(10):
12)     print("Rama Thread")
```

In the above example Main Thread waited until completing child thread. In this case output is:

```
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
```

Seetha Thread
Seetha Thread
Seetha Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread

Note: We can call join() method with time period also.

```
t.join(seconds)
```

In this case thread will wait only specified amount of time.

Eg:

```
1)  from threading import *
2)  import time
3)  def display():
4)      for i in range(10):
5)          print("Seetha Thread")
6)          time.sleep(2)
7)
8)  t=Thread(target=display)
9)  t.start()
10) t.join(5)#This Line executed by Main Thread
11) for i in range(10):
12)     print("Rama Thread")
```

In this case Main Thread waited only 5 seconds.

Output:

Seetha Thread
Seetha Thread
Seetha Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread

Rama Thread
Rama Thread
Rama Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread

Summary of all methods related to threading module and Thread

Daemon Threads:

The threads which are running in the background are called Daemon Threads.

The main objective of Daemon Threads is to provide support for Non Daemon Threads(like main thread)

Eg: Garbage Collector

Whenever Main Thread runs with low memory, immediately PVM runs Garbage Collector to destroy useless objects and to provide free memory,so that Main Thread can continue its execution without having any memory problems.

We can check whether thread is Daemon or not by using t.isDaemon() method of Thread class or by using daemon property.

Eg:

```
1) from threading import *
2) print(current_thread().isDaemon()) #False
3) print(current_thread().daemon) #False
```

We can change Daemon nature by using setDaemon() method of Thread class.

```
t.setDaemon(True)
```

But we can use this method before starting of Thread.i.e once thread started,we cannot change its Daemon nature,otherwise we will get
RuntimeException:cannot set daemon status of active thread

Eg:

```
1) from threading import *
2) print(current_thread().isDaemon())
```

```
3) current_thread().setDaemon(True)
```

RuntimeError: cannot set daemon status of active thread

Default Nature:

By default Main Thread is always non-daemon.But for the remaining threads Daemon nature will be inherited from parent to child.i.e if the Parent Thread is Daemon then child thread is also Daemon and if the Parent Thread is Non Daemon then ChildThread is also Non Daemon.

Eg:

```
1) from threading import *
2) def job():
3)     print("Child Thread")
4) t=Thread(target=job)
5) print(t.isDaemon())#False
6) t.setDaemon(True)
7) print(t.isDaemon()) #True
```

Note: Main Thread is always Non-Daemon and we cannot change its Daemon Nature b'z it is already started at the beginning only.

Whenever the last Non-Daemon Thread terminates automatically all Daemon Threads will be terminated.

Eg:

```
1) from threading import *
2) import time
3) def job():
4)     for i in range(10):
5)         print("Lazy Thread")
6)         time.sleep(2)
7)
8) t=Thread(target=job)
9) #t.setDaemon(True)===>Line-1
10) t.start()
11) time.sleep(5)
12) print("End Of Main Thread")
```

In the above program if we comment Line-1 then both Main Thread and Child Threads are Non Daemon and hence both will be executed until their completion.
In this case output is:

```
Lazy Thread
Lazy Thread
Lazy Thread
```

End Of Main Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread

If we are not commenting Line-1 then Main Thread is Non-Daemon and Child Thread is Daemon. Hence whenever MainThread terminates automatically child thread will be terminated. In this case output is

Lazy Thread
Lazy Thread
Lazy Thread
End of Main Thread

Synchronization:

If multiple threads are executing simultaneously then there may be a chance of data inconsistency problems.

Eg:

```
1)  from threading import *
2)  import time
3)  def wish(name):
4)      for i in range(10):
5)          print("Good Evening:",end='')
6)          time.sleep(2)
7)          print(name)
8)  t1=Thread(target=wish,args=("Dhoni",))
9)  t2=Thread(target=wish,args=("Yuvraj",))
10) t1.start()
11) t2.start()
```

Output:

```
 Good Evening:Good Evening:Yuvraj
Dhoni
Good Evening:Good Evening:Yuvraj
Dhoni
....
```

We are getting irregular output b'z both threads are executing simultaneously wish() function.

To overcome this problem we should go for synchronization.

In synchronization the threads will be executed one by one so that we can overcome data inconsistency problems.

Synchronization means at a time only one Thread

The main application areas of synchronization are
1. Online Reservation system
2. Funds Transfer from joint accounts
etc

In Python, we can implement synchronization by using the following
1. Lock
2. RLock
3. Semaphore

Synchronization By using Lock concept:

Locks are the most fundamental synchronization mechanism provided by threading module.
We can create Lock object as follows
l=Lock()

The Lock object can be hold by only one thread at a time.If any other thread required the same lock then it will wait until thread releases lock.(similar to common wash rooms,public telephone booth etc)

A Thread can acquire the lock by using acquire() method.
l.acquire()

A Thread can release the lock by using release() method.
l.release()

Note: To call release() method compulsory thread should be owner of that lock.i.e thread should has the lock already,otherwise we will get Runtime Exception saying
RuntimeError: release unlocked lock

Eg:

```
1) from threading import *
2) l=Lock()
3) #l.acquire() ==>1
4) l.release()
```

If we are commenting line-1 then we will get
RuntimeError: release unlocked lock

Eg:

```
1) from threading import *
```

```
2)  import time
3)  l=Lock()
4)  def wish(name):
5)      l.acquire()
6)      for i in range(10):
7)          print("Good Evening:",end='')
8)          time.sleep(2)
9)          print(name)
10)     l.release()
11)
12) t1=Thread(target=wish,args=("Dhoni",))
13) t2=Thread(target=wish,args=("Yuvraj",))
14) t3=Thread(target=wish,args=("Kohli",))
15) t1.start()
16) t2.start()
17) t3.start()
```

In the above program at a time only one thread is allowed to execute wish() method and hence we will get regular output.

Problem with Simple Lock:

The standard Lock object does not care which thread is currently holding that lock.If the lock is held and any thread attempts to acquire lock, then it will be blocked,even the same thread is already holding that lock.

Eg:

```
1)  from threading import *
2)  l=Lock()
3)  print("Main Thread trying to acquire Lock")
4)  l.acquire()
5)  print("Main Thread trying to acquire Lock Again")
6)  l.acquire()
```

Output:

```
D:\python_classes>py test.py
Main Thread trying to acquire Lock
Main Thread trying to acquire Lock Again
--
```

In the above Program main thread will be blocked b'z it is trying to acquire the lock second time.

Note: To kill the blocking thread from windows command prompt we have to use ctrl+break. Here ctrl+C won't work.

If the Thread calls recursive functions or nested access to resources,then the thread may trying to acquire the same lock again and again,which may block our thread.

Hence Traditional Locking mechanism won't work for executing recursive functions.

To overcome this problem, we should go for RLock(Reentrant Lock). Reentrant means the thread can acquire the same lock again and again.If the lock is held by other threads then only the thread will be blocked.
Reentrant facility is available only for owner thread but not for other threads.

Eg:

```
1) from threading import *
2) l=RLock()
3) print("Main Thread trying to acquire Lock")
4) l.acquire()
5) print("Main Thread trying to acquire Lock Again")
6) l.acquire()
```

In this case Main Thread won't be Locked b'z thread can acquire the lock any number of times.

This RLock keeps track of recursion level and hence for every acquire() call compulsory release() call should be available. i.e the number of acquire() calls and release() calls should be matched then only lock will be released.

Eg:

```
l=RLock()
l.acquire()
l.acquire()
l.release()
l.release()
```

After 2 release() calls only the Lock will be released.

Note:

1. Only owner thread can acquire the lock multiple times
2. The number of acquire() calls and release() calls should be matched.

Demo Program for synchronization by using RLock:

```
1) from threading import *
2) import time
3) l=RLock()
4) def factorial(n):
5)     l.acquire()
6)     if n==0:
7)         result=1
8)     else:
9)         result=n*factorial(n-1)
10)    l.release()
11)    return result
```

```
12)
13) def results(n):
14)     print("The Factorial of",n,"is:",factorial(n))
15)
16) t1=Thread(target=results,args=(5,))
17) t2=Thread(target=results,args=(9,))
18) t1.start()
19) t2.start()
```

Output:
The Factorial of 5 is: 120
The Factorial of 9 is: 362880

In the above program instead of RLock if we use normal Lock then the thread will be blocked.

Difference between Lock and RLock:

table

Lock:

1. Lock object can be acquired by only one thread at a time.Even owner thread also cannot acquire multiple times.
2. Not suitable to execute recursive functions and nested access calls
3. In this case Lock object will takes care only Locked or unlocked and it never takes care about owner thread and recursion level.

RLock:

1. RLock object can be acquired by only one thread at a time, but owner thread can acquire same lock object multiple times.
2. Best suitable to execute recursive functions and nested access calls
3. In this case RLock object will takes care whether Locked or unlocked and owner thread information, recursiion level.

Synchronization by using Semaphore:

In the case of Lock and RLock,at a time only one thread is allowed to execute.

Sometimes our requirement is at a time a particular number of threads are allowed to access(like at a time 10 memebers are allowed to access database server,4 members are allowed to access Network connection etc).To handle this requirement we cannot use Lock and RLock concepts and we should go for Semaphore concept.

Semaphore can be used to limit the access to the shared resources with limited capacity.

Semaphore is advanced Synchronization Mechanism.

We can create Semaphore object as follows.

s=Semaphore(counter)

Here counter represents the maximum number of threads are allowed to access simultaneously. The default value of counter is 1.

Whenever thread executes acquire() method,then the counter value will be decremented by 1 and if thread executes release() method then the counter value will be incremented by 1.

i.e for every acquire() call counter value will be decremented and for every release() call counter value will be incremented.

Case-1: s=Semaphore()

In this case counter value is 1 and at a time only one thread is allowed to access. It is exactly same as Lock concept.

Case-2: s=Semaphore(3)

In this case Semaphore object can be accessed by 3 threads at a time.The remaining threads have to wait until releasing the semaphore.

Eg:

```
1)  from threading import *
2)  import time
3)  s=Semaphore(2)
4)  def wish(name):
5)      s.acquire()
6)      for i in range(10):
7)          print("Good Evening:",end='')
8)          time.sleep(2)
9)          print(name)
10)     s.release()
11)
12) t1=Thread(target=wish,args=("Dhoni",))
13) t2=Thread(target=wish,args=("Yuvraj",))
14) t3=Thread(target=wish,args=("Kohli",))
15) t4=Thread(target=wish,args=("Rohit",))
16) t5=Thread(target=wish,args=("Pandya",))
17) t1.start()
18) t2.start()
19) t3.start()
20) t4.start()
21) t5.start()
```

In the above program at a time 2 threads are allowed to access semaphore and hence 2 threads are allowed to execute wish() function.

BoundedSemaphore:

Normal Semaphore is an unlimited semaphore which allows us to call release() method any number of times to increment counter.The number of release() calls can exceed the number of acquire() calls also.

Eg:

```
1) from threading import *
2) s=Semaphore(2)
3) s.acquire()
4) s.acquire()
5) s.release()
6) s.release()
7) s.release()
8) s.release()
9) print("End")
```

It is valid because in normal semaphore we can call release() any number of times.

BounedSemaphore is exactly same as Semaphore except that the number of release() calls should not exceed the number of acquire() calls,otherwise we will get

ValueError: Semaphore released too many times

Eg:

```
1) from threading import *
2) s=BoundedSemaphore(2)
3) s.acquire()
4) s.acquire()
5) s.release()
6) s.release()
7) s.release()
8) s.release()
9) print("End")
```

ValueError: Semaphore released too many times
It is invalid b'z the number of release() calls should not exceed the number of acquire() calls in BoundedSemaphore.

Note: To prevent simple programming mistakes, it is recommended to use BoundedSemaphore over normal Semaphore.

Difference between Lock and Semaphore:

At a time Lock object can be acquired by only one thread, but Semaphore object can be acquired by fixed number of threads specified by counter value.

Conclusion:

The main advantage of synchronization is we can overcome data inconsistency problems.But the main disadvantage of synchronization is it increases waiting time of threads and creates performance problems. Hence if there is no specific requirement then it is not recommended to use synchronization.

Inter Thread Communication:

Some times as the part of programming requirement, threads are required to communicate with each other. This concept is nothing but interthread communication.

Eg: After producing items Producer thread has to communicate with Consumer thread to notify about new item.Then consumer thread can consume that new item.

In Python, we can implement interthread communication by using the following ways

1. Event
2. Condition
3. Queue
etc

Interthread communication by using Event Objects:

Event object is the simplest communication mechanism between the threads. One thread signals an event and other thereds wait for it.

We can create Event object as follows...

```
event = threading.Event()
```

Event manages an internal flag that can set() or clear()
Threads can wait until event set.

Methods of Event class:

1.set() ➔ internal flag value will become True and it represents GREEN signal for all waiting threads.

2. clear() ➔ inernal flag value will become False and it represents RED signal for all waiting threads.

3. isSet() ➔ This method can be used whether the event is set or not

4. wait()|wait(seconds) ➔ Thread can wait until event is set

Pseudo Code:

```
event = threading.Event()

#consumer thread has to wait until event is set
event.wait()

#producer thread can set or clear event
event.set()
event.clear()
```

Demo Program-1:

```
1)  from threading import *
2)  import time
3)  def producer():
4)      time.sleep(5)
5)      print("Producer thread producing items:")
6)      print("Producer thread giving notification by setting event")
7)      event.set()
8)  def consumer():
9)      print("Consumer thread is waiting for updation")
10)     event.wait()
11)     print("Consumer thread got notification and consuming items")
12)
13) event=Event()
14) t1=Thread(target=producer)
15) t2=Thread(target=consumer)
16) t1.start()
17) t2.start()
```

Output:

```
Consumer thread is waiting for updation
Producer thread producing items
Producer thread giving notification by setting event
Consumer thread got notification and consuming items
```

Demo Program-2:

```
1)   from threading import *
2)  import time
3)  def trafficpolice():
4)      while True:
5)          time.sleep(10)
6)          print("Traffic Police Giving GREEN Signal")
```

```
7)        event.set()
8)        time.sleep(20)
9)        print("Traffic Police Giving RED Signal")
10)       event.clear()
11) def driver():
12)    num=0
13)    while True:
14)       print("Drivers waiting for GREEN Signal")
15)       event.wait()
16)       print("Traffic Signal is GREEN...Vehicles can move")
17)       while event.isSet():
18)          num=num+1
19)          print("Vehicle No:",num,"Crossing the Signal")
20)          time.sleep(2)
21)       print("Traffic Signal is  RED...Drivers have to wait")
22) event=Event()
23) t1=Thread(target=trafficpolice)
24) t2=Thread(target=driver)
25) t1.start()
26) t2.start()
```

In the above program driver thread has to wait until Trafficpolice thread sets event.ie until giving GREEN signal.Once Traffic police thread sets event(giving GREEN signal),vehicles can cross the signal.Once traffic police thread clears event (giving RED Signal)then the driver thread has to wait.

Interthread communication by using Condition Object:

Condition is the more advanced version of Event object for interthread communication.A condition represents some kind of state change in the application like producing item or consuming item. Threads can wait for that condition and threads can be notified once condition happend.i.e Condition object allows one or more threads to wait until notified by another thread.

Condition is always associated with a lock (ReentrantLock).
A condition has acquire() and release() methods that call the corresponding methods of the associated lock.

We can create Condition object as follows

```
condition = threading.Condition()
```

Methods of Condition:

1. acquire() ➔ To acquire Condition object before producing or consuming items.i.e thread acquiring internal lock.

2. release() ➔ To release Condition object after producing or consuming items. i.e thread releases internal lock

3. wait()|wait(time) ➔ To wait until getting Notification or time expired

4. notify() ➔ To give notification for one waiting thread

5. notifyAll() ➔ To give notification for all waiting threads

Case Study:

The producing thread needs to acquire the Condition before producing item to the resource and notifying the consumers.

```
#Producer Thread
...generate item..
condition.acquire()
...add item to the resource...
condition.notify()#signal that a new item is available(notifyAll())
condition.release()
```

The Consumer must acquire the Condition and then it can consume items from the resource

```
#Consumer Thread
condition.acquire()
condition.wait()
consume item
condition.release()
```

Demo Program-1:

```
1)  from threading import *
2)  def consume(c):
3)      c.acquire()
4)      print("Consumer waiting for updation")
5)      c.wait()
6)      print("Consumer got notification & consuming the item")
7)      c.release()
8)
9)  def produce(c):
10)     c.acquire()
11)     print("Producer Producing Items")
12)     print("Producer giving Notification")
13)     c.notify()
14)     c.release()
15)
16) c=Condition()
17) t1=Thread(target=consume,args=(c,))
18) t2=Thread(target=produce,args=(c,))
19) t1.start()
20) t2.start()
```

Output:

```
Consumer waiting for updation
Producer Producing Items
Producer giving Notification
Consumer got notification & consuming the item
```

Demo Program-2:

```
1)  from threading import *
2)  import time
3)  import random
4)  items=[]
5)  def produce(c):
6)      while True:
7)          c.acquire()
8)          item=random.randint(1,100)
9)          print("Producer Producing Item:",item)
10)         items.append(item)
11)         print("Producer giving Notification")
12)         c.notify()
13)         c.release()
14)         time.sleep(5)
15)
16) def consume(c):
17)     while True:
18)         c.acquire()
19)         print("Consumer waiting for updation")
20)         c.wait()
21)         print("Consumer consumed the item",items.pop())
22)         c.release()
23)         time.sleep(5)
24)
25) c=Condition()
26) t1=Thread(target=consume,args=(c,))
27) t2=Thread(target=produce,args=(c,))
28) t1.start()
29) t2.start()
```

Output:

```
Consumer waiting for updation
Producer Producing Item: 49
Producer giving Notification
Consumer consumed the item 49
.....
```

In the above program Consumer thread expecting updation and hence it is responsible to call wait() method on Condition object.
Producer thread performing updation and hence it is responsible to call notify() or notifyAll() on Condition object.

Interthread communication by using Queue:

Queues Concept is the most enhanced Mechanism for interthread communication and to share data between threads.

Queue internally has Condition and that Condition has Lock.Hence whenever we are using Queue we are not required to worry about Synchronization.

If we want to use Queues first we should import queue module.

import queue

We can create Queue object as follows

```
q = queue.Queue()
```

Important Methods of Queue:

1. put(): Put an item into the queue.
2. get(): Remove and return an item from the queue.

Producer Thread uses put() method to insert data in the queue. Internally this method has logic to acquire the lock before inserting data into queue. After inserting data lock will be released automatically.

put() method also checks whether the queue is full or not and if queue is full then the Producer thread will entered in to waiting state by calling wait() method internally.

Consumer Thread uses get() method to remove and get data from the queue. Internally this method has logic to acquire the lock before removing data from the queue.Once removal completed then the lock will be released automatically.

If the queue is empty then consumer thread will entered into waiting state by calling wait() method internally.Once queue updated with data then the thread will be notified automatically.

Note:
The queue module takes care of locking for us which is a great advantage.

Eg:

```
1) from threading import *
2) import time
3) import random
4) import queue
5) def produce(q):
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
    while True:
        item=random.randint(1,100)
        print("Producer Producing Item:",item)
        q.put(item)
        print("Producer giving Notification")
        time.sleep(5)
def consume(q):
    while True:
        print("Consumer waiting for updation")
        print("Consumer consumed the item:",q.get())
        time.sleep(5)

q=queue.Queue()
t1=Thread(target=consume,args=(q,))
t2=Thread(target=produce,args=(q,))
t1.start()
t2.start()
```

Output:
Consumer waiting for updation
Producer Producing Item: 58
Producer giving Notification
Consumer consumed the item: 58

Types of Queues:

Python Supports 3 Types of Queues.

1. FIFO Queue:

q = queue.Queue()

This is Default Behaviour. In which order we put items in the queue, in the same order the items will come out (FIFO-First In First Out).

Eg:

```
import queue
q=queue.Queue()
q.put(10)
q.put(5)
q.put(20)
q.put(15)
while not q.empty():
    print(q.get(),end='  ')
```

Output: 10 5 20 15

2. LIFO Queue:

The removal will be happend in the reverse order of insertion(Last In First Out)

Eg:

```
1)  import queue
2)  q=queue.LifoQueue()
3)  q.put(10)
4)  q.put(5)
5)  q.put(20)
6)  q.put(15)
7)  while not q.empty():
8)      print(q.get(),end='  ')
```

Output: 15 20 5 10

3. Priority Queue:

The elements will be inserted based on some priority order.

```
1)  import queue
2)  q=queue.PriorityQueue()
3)  q.put(10)
4)  q.put(5)
5)  q.put(20)
6)  q.put(15)
7)  while not q.empty():
8)      print(q.get(),end='  ')
```

Output: 5 10 15 20

Eg 2: **If the data is non-numeric, then we have to provide our data in the form of tuple.**

(x,y)

x is priority
y is our element

```
1)  import queue
2)  q=queue.PriorityQueue()
3)  q.put((1,"AAA"))
4)  q.put((3,"CCC"))
5)  q.put((2,"BBB"))
6)  q.put((4,"DDD"))
7)  while not q.empty():
8)      print(q.get()[1],end='  ')
```

Output: AAA BBB CCC DDD

Good Programming Practices with usage of Locks:

Case-1:

It is highly recommended to write code of releasing locks inside finally block.The advantage is lock will be released always whether exception raised or not raised and whether handled or not handled.

```
l=threading.Lock()
l.acquire()
try:
        perform required safe operations
finally:
        l.release()
```

Demo Program:

```
1)  from threading import *
2)  import time
3)  l=Lock()
4)  def wish(name):
5)      l.acquire()
6)      try:
7)          for i in range(10):
8)              print("Good Evening:",end='')
9)              time.sleep(2)
10)             print(name)
11)     finally:
12)         l.release()
13)
14) t1=Thread(target=wish,args=("Dhoni",))
15) t2=Thread(target=wish,args=("Yuvraj",))
16) t3=Thread(target=wish,args=("Kohli",))
17) t1.start()
18) t2.start()
19) t3.start()
```

Case-2:

It is highly recommended to acquire lock by using with statement. The main advantage of with statement is the lock will be released automatically once control reaches end of with block and we are not required to release explicitly.

This is exactly same as usage of with statement for files.

Example for File:

```
with open('demo.txt','w') as f:
    f.write("Hello...")
```

Example for Lock:

```
lock=threading.Lock()
with lock:
        perform required safe operations
        lock will be released automatically
```

Demo Program:

```
1)  from threading import *
2)  import time
3)  lock=Lock()
4)  def wish(name):
5)      with lock:
6)          for i in range(10):
7)              print("Good Evening:",end='')
8)              time.sleep(2)
9)              print(name)
10) t1=Thread(target=wish,args=("Dhoni",))
11) t2=Thread(target=wish,args=("Yuvraj",))
12) t3=Thread(target=wish,args=("Kohli",))
13) t1.start()
14) t2.start()
15) t3.start()
```

Q. What is the advantage of using with statement to acquire a lock in threading?

Lock will be released automatically once control reaches end of with block and We are not required to release explicitly.

Note: We can use with statement in multithreading for the following cases:

1. Lock
2. RLock
3. Semaphore
4. Condition