26 Synchronization

"Well-oiled threads run smoother..."

Contents

Synchronization

- In a multithreaded application we may be needed to coordinate (synchronize) the activities of the threads running in it.
- The need to coordinate activities of threads will arise in two situations:

  (a) When data or other resources are to be shared amongst threads.
  (b) When we need to carry out communication between threads.

Examples of Sharing Resources

- **Example 1**: Suppose a function has a statement **n = n + 1**. Here value of **n** is read, 1 is added to it and the result is written back. If multiple threads call this function then **n** will be shared amongst these threads. In such a case, if one thread has read **n** and before it updates it another thread may read and update it. Such overlapping accesses and modifications from multiple threads may not increment **n** systematically.
- **Solution to Example 1**: To ensure proper incrementation of **n**, we should ensure that whichever thread gets the time-slot first should complete working with **n**. If in the meanwhile another thread gets the time-slot, it should be made to wait. Only when first thread is done, the other thread should be able to access to **n**.
- **Example 2**: Suppose there are two threads in an application. One thread reads a list of numbers and prints its squares and another reads the list and prints cubes of numbers in it. So both threads are going to share the list. When the threads print the squares and cubes, the output is likely to get mixed up.
- **Solution to Example 2**: To avoid mixing of output we should ensure that whichever thread gets the time-slot first should complete working with the list. If in the meanwhile other thread gets the time-slot, it should be made to wait. Only when first thread is done, the other thread should be able to access the list.

Example of Communication between Threads

- Suppose one thread is generating numbers in an infinite loop and another thread is finding squares of generated numbers. Unless the new number is generated its square cannot be found. So if squaring thread gets the time slot earlier than the generating thread, squaring thread must be made to wait. Also, when square is being generated, new numbers should not get generated. This is necessary otherwise the squaring thread may miss some numbers.

- This is a typical producer-consumer problem, where the number generating thread is the producer and the squaring thread is the consumer.
- Here communication between two threads would be required. When producer thread completes production it should communicate to the squaring thread that it is done with production. When consumer thread completes squaring it should communicate to the producer thread that it is done and producer thread can produce the next number.

Mechanisms for Sharing Resources

- Python's **threading** module provides three mechanisms for sharing resources between threads:

  (a) Lock
  (b) RLock
  (c) Semaphore

- They should be used in following situations:

  - For synchronized access to shared resources - use lock.
  - For nested access to shared resources - use re-entrant lock.
  - For permitting limited number of accesses to a resource - use semaphore.

Lock

- Locks are used to synchronize access to a shared resource. We should first create a **Lock** object. When we need to access the resource we should call **acquire( )**, then use the resource and once done, call **release( )** as shown below:

```
lck = threading.Lock( )
lck.acquire( )
# use the resource
lck.release( )
```

- For each shared resource, a new **Lock** object should be created.
- A lock can be in two states—'Locked' or 'Unlocked'.
- A **Lock** object has two methods—**acquire( )** and **release( )**. If a thread calls **acquire( )** it puts the lock in 'Locked' state if it is currently in 'Unlocked' state and returns. If it is already in 'Locked' state then the call to **acquire( )** blocks the thread (means control doesn't return from **acquire( )**). A call to **release( )** puts the lock in 'Unlocked' state.

RLock

- Sometimes a recursive function may be invoked through multiple threads. In such cases, if we use **Lock** to provide synchronized access to shared variables it would lead to a problem—thread will be blocked when it attempts to acquire the same lock second time.
- This problem can be overcome by using re-entrant Lock or **RLock.** A re-entrant lock only blocks if another thread currently holds the lock. If the current thread tries to acquire a lock that it's already holding, execution continues as usual.
- A lock/rlock acquired by one thread can be released either by same thread or by another thread.
- **release( )** should be called as many times as **acquire( )** is called.
- Following code snippet shows working of normal lock and re-entrant lock.

```
lck = threading.Lock( )
lck.acquire( )
lck.acquire( )          # this will block

rlck = threading.RLock( )
rlck.acquire( )
rlck.acquire( )         # this won't block
```

- A lock/rlock is also known as mutex as it permits mutual exclusive access to a resource.

Semaphore

- If we wish to permit access to a resource like network connection or a database server to a limited number of threads we can do so using a semaphore object.
- A semaphore object uses a counter rather than a lock flag. The counter can be set to indicate the number of threads that can acquire the semaphore before blocking occurs.
- Once the counter is set, the counter decreases per **acquire( )** call, and increases per **release( )** call. Blocking occurs only if more than the set number of threads attempt to acquire the semaphore.
- We have to only initialize the counter to the maximum number while creating the semaphore object, and the semaphore implementationl takes care of the rest.

Mechanisms for Inter-thread Communication (ITC)

- Python's **threading** module provides two mechanisms for inter-thread communication:

  (a) Event
  (b) Condition

Event

- An **Event** object is used to communicate between threads. It has an internal flag which threads can set or clear through methods **set( )** and **clear( )**.
- Typical working: If thread 1 calls the method **wait( )**, it will wait (block) if internal flag has not yet been set. Thread 2 will set the flag. Since the flag now stands set, Thread 1 will come out its wait state, perform its work and then clear the flag. This scenario is shown in the following program:

```
def fun1( ) :
    while True :
        # wait for the flag to be set
        ev.wait( )
        # once flag is set by thread 2, do the work in this thread
        ev.clear( )  # clear the flag

def fun2( ) :
    while True :
        # perform some work
        # set the flag
        ev.set( )

ev = Event( )
th1 = threading.Thread(target = fun1)
th2 = threading.Thread(target = fun2)
```

Condition

- A **Condition** object is an advanced version of the **Event** object. It too is used to communicate between threads. It has methods **acquire( )**, **release( )**, **wait( )**, **notify( )** and **notifyAll( )**.
- A **Condition** object internally uses a lock that can be acquired or released using **acquire( )** and **release( )** functions respectively. **acquire( )** blocks if the lock is already in locked state.
- **Condition** object can notify other threads using **notify( )/notifyAll( )** about a change in the state of the program.
- The **wait( )** method releases the lock, and then blocks until it is awakened by a **notify( )** or **notifyAll( )** call for the same **Condition** in another thread. Once awakened, it re-acquires the lock and returns.
- A thread should release a **Condition** once it has completed the related actions, so that other threads can acquire the condition for their purposes.
- Producer Consumer algorithm is a technique for generating requests and processing the pending requests. Producer produces requests, Consumer consumes generated requests. Both work as independent threads.
- **Condition** object can be used to implement a Producer Consumer algorithm as shown below:

```
# Producer thread
cond.acquire( )
# code here to produce one item
cond.notify( )
cond.release( )

# Consumer thread
```

```
cond.acquire( )
while item_is_not_available( ) :
    cond.wait( )
# code here to consume the item
cond.release( )
```

- Working of Producer Consumer problem:
  - Consumer waits while Producer is producing.
  - Once Producer has produced it sends a signal to Consumer.
  - Producer waits while Consumer is consuming.
  - Once Consumer has consumed it sends a signal to Producer.

P</> *Programs*

Problem 26.1

Write a program through which you can prove that in this programming situation synchronization is really required. Then write a program to demonstrate how synchronization can solve the problem.

Program

```
import time
import threading

def fun1( ) :
    print('Entering fun1')
    global g
    g += 1
    #time.sleep(10)
    g -= 1
    print('In fun1 g =', g)
    print('Exiting fun1')

def fun2( ) :
    print('Entering fun2')
    global g
    g += 2
    g -= 2
    print('In fun2 g =', g)
    print('Exiting fun2')

g = 10
th1 = threading.Thread(target = fun1)
th2 = threading.Thread(target = fun2)
th1.start( )
th2.start( )
th1.join( )
th2.join( )
```

Output

```
Entering fun1
In fun1 g = 10
Exiting fun1
Entering fun2
In fun2 g = 10
Exiting fun2
```

If you uncomment the call to **time.sleep( )**, the output changes to:

```
Entering fun1
Entering fun2
In fun2 g = 11
Exiting fun2
In fun1 g = 10
Exiting fun1
```

Tips

- We are using the global variable **g** in **fun1( )** and **fun2(**) which are running in two different threads. As expected, both print the value of **g** as 10, as both increment and decrement it by 1 and 2 respectively.
- If you uncomment the call to **sleep( )** the output becomes inconsistent. **fun1( )** increments the value of **g** to 11, but before it can decrement the incremented value, **fun2( )** gets the time-slot, which increments **g** to 13, decrements it to 11 and prints it. The time-slot again goes to **fun1( )**, which decrements **g** to 10 and prints it.

- The solution to avoid this mismatch is given in the program shown below.

Program

```
import time
import threading

def fun1( ) :
    print('Entering fun1')
    global g
    lck.acquire( )
    g += 1
    g -= 1
    lck.release( )
    print('In fun1 g =', g)
    print('Exiting fun1')

def fun2( ) :
    print('Entering fun2')
    global g
    lck.acquire( )
    g += 2
    g -= 2
    lck.release( )
    print('In fun2 g =', g)
    print('Exiting fun2')

g = 10
lck = threading.Lock( )
th1 = threading.Thread(target = fun1)
th2 = threading.Thread(target = fun2)
th1.start( )
th2.start( )
th1.join( )
th2.join( )
```

Tips

- In main thread we have created a **Lock** object through the call **threading.Lock( )**.

- If **fun1** thread gets the first time-slot, it calls **acquire( )**. This call puts the lock in 'Locked' state and returns. So **fun1** thread can work with g. If midway through its time-slot expires and **fun2** thread gets it, it will also call **acquire( )**, but it will be blocked (control will not return from it) since lock is in 'Locked' state. In the next time-slot **fun1** thread finishes its work and releases the lock (puts the lock in 'Unlocked' state) by calling **release( )**. As a result, **fun2** thread can work with **g** when it gets time-slot.

Problem 26.2

Write a program that calculates the squares and cubes of first 6 odd numbers through functions that are executed in two independent threads. Incorporate a delay of 0.5 seconds after calculation of each square/cube value. Report the time required for execution of the program. Make sure that the output of **squares( )** and **cubes( )** doesn't get mixed up.

Program

```
import time
import threading

def squares(nos, lck) :
    lck.acquire( )
    print('Calculating squares...')
    for n in nos :
        time.sleep(0.5)
        print('n = ', n, ' square =', n * n)
    lck.release( )

def cubes(nos, lck) :
    lck.acquire( )
    print('Calculating cubes...')
    for n in nos :
        time.sleep(0.5)
        print('n = ', n, ' cube =', n * n * n)
    lck.release( )

arr = [1, 3, 5, 7, 9, 11]
startTime = time.time( )
```

```
lck = threading.Lock( )

th1 = threading.Thread(target = squares, args = (arr, lck))
th2 = threading.Thread(target = cubes, args = (arr, lck))
th1.start( )
th2.start( )
th1.join( )
th2.join( )

endTime = time.time( )
print('Time required = ', endTime - startTime, 'sec')
```

Output

```
Calculating squares...
n =  1  square = 1
n =  3  square = 9
n =  5  square = 25
n =  7  square = 49
n =  9  square = 81
n =  11  square = 121
Calculating cubes...
n =  1  cube = 1
n =  3  cube = 27
n =  5  cube = 125
n =  7  cube = 343
n =  9  cube = 729
n =  11  cube = 1331
Time required =  6.001343250274658 sec
```

Tips

- To ensure that output of **squares( )** doesn't get mixed up with output of **cubes( )** we should ensure that when one is working another should be put on hold.
- In main thread we have created a **Lock** object through the call **threading.Lock( )**. Along with the list, this **Lock** object is shared between **squares( )** and **cubes( )**.
- If **squares** thread gets the first time-slot, it calls **acquire( ).** This call puts the lock in 'Locked' state and returns. So **squares** thread can

start generating and printing squares. If midway through its time-slot expires and **cubes** thread gets it, it will also call **acquire( )**, but it will be blocked (control will not return from it) since lock is in 'Locked' state. In the next time-slot **squares** thread finishes its work and releases the lock (puts the lock in 'Unlocked' state) by calling **release( )**.

- Similar reasoning would hold good if **cubes** thread gets the first time-slot.
- Suppose there were three threads **squares**, **cubes** and **quadruples** and **squares** thread acquires the lock. When it releases the lock which of the two waiting threads will proceed is not defined and may vary across Python implementations.

Problem 26.3

Write a program that prints the following 3 messages through 3 different threads:

```
[What is this life...]
[We have no time...]
[To stand and stare!]
```

Each thread should be passed the relevant message and should print '[', message and ']' through three different **print( )** calls.

Program

```
import time
import threading

def printMsg(msg, lck):
    lck.acquire( )
    print('[', end = '')
    print(msg, end = '')
    time.sleep(0.5)
    print(']')
    lck.release( )

lck = threading.Lock( )
th1 = threading.Thread(target = printMsg,
        args = ('What is this life...', lck))
```

```
th1.start( )
th2 = threading.Thread(target = printMsg,
        args = ('We have no time...', lck))
th2.start( )
th3 = threading.Thread(target = printMsg,
        args = ('To stand and stare!', lck))
th3.start( )

th1.join( )
th2.join( )
th3.join( )
```

Tips

- Three threads are created. In each thread the **printMsg( )** function is executed, but a different message is passed to it in each thread.
- To ensure that '[', message and ']' are printed in the same order in each thread, the activity of the threads is synchronized.
- When one thread acquires a lock, others are blocked until the thread that acquired the lock releases it.

Problem 26.4

Write a program that runs a recursive **print_num( )** function in 2 threads. This function should receive an integer and print all numbers from that number up to 1.

Program

```
import threading

def print_num(n) :
    try :
        rlck.acquire( )
        if n == 0 :
            return
        else :
            t = threading.current_thread( )
            print(t.name, ':', n)
            n -= 1
```

```
                print_num(n)
    finally :
            rlck.release( )

rlck = threading.RLock( )
th1 = threading.Thread(target = print_num, args = (8,))
th1.start( )
th2 = threading.Thread(target = print_num, args = (5,))
th2.start( )
th1.join( )
th2.join( )
```

Output

```
Thread-1 : 8
Thread-1 : 7
Thread-1 : 6
Thread-1 : 5
Thread-1 : 4
Thread-1 : 3
Thread-1 : 2
Thread-1 : 1
Thread-2 : 5
Thread-2 : 4
Thread-2 : 3
Thread-2 : 2
Thread-2 : 1
```

Tips

- Since we are sharing resources in a recursive function we have used **RLock** instead of **Lock**.
- A lock acquired by one thread can be released by another. So we have released the lock in **finally** block for each thread. **finally** block goes to work only when control returns from **print_num( )** last time after completing all recursive calls.
- We have printed name of each thread along with the current value of **n** so that we get an idea of which thread are we working in.

- If we replace **RLock** with **Lock** we will get output from one thread only. This is because one thread will acquire the lock and do some printing. When its' time-slot expires and another thread gets it, it will also call **acquire( )** and would get blocked.
- If you do not use any lock the output from the two threads will get mixed up.

Problem 26.5

Write a program that runs a recursive **factorial( )** function in 2 threads. This function should receive an integer and print all the intermediate products and final product.

Program

```
import threading

def factorial(n) :
    try :
        rlck.acquire( )
        if n == 0 :
            return 1
        else :
            p = n * factorial(n - 1)
            print(f'{n}! = {p}')
        return p
    finally :
        rlck.release( )

rlck = threading.RLock( )
th1 = threading.Thread(target = factorial, args = (5,))
th1.start( )
th2 = threading.Thread(target = factorial, args = (8,))
th2.start( )
th1.join( )
th2.join( )
```

Output

```
1 != 1
```

```
2 != 2
3 != 6
4 != 24
5 != 120
1 != 1
2 != 2
3 != 6
4 != 24
5 != 120
6 != 720
7 != 5040
8 != 40320
```

Tips

- Since we are sharing resources in a recursive function we have used **RLock** instead of **Lock**.
- A lock acquired by one thread can be released by another. So we have released the lock in **finally** block for each thread. **finally** block goes to work only when control returns from **factorial( )** last time after completing all recursive calls.
- If we replace **RLock** with **Lock** we will get output from one thread only. This is because one thread will acquire the lock and do some calculation and printing. When its' time-slot expires and other thread gets it, it will also call **acquire( )** and would get blocked.
- If we do not use any lock the output from the two threads will get mixed up.

Problem 26.6

Write a program that defines a function **fun( )** that prints a message that it receives infinite times. Limit the number of threads that can invoke **fun( )** to 3. If 4th thread tries to invoke **fun( )**, it should not get invoked.

Program

```
import threading

def fun(msg) :
    s.acquire( )
    t = threading.current_thread( )
    while True :
        print(t.name, ':', msg)
    s.release( )

s = threading.BoundedSemaphore(3)
th1 = threading.Thread(target = fun, args = ('Hello',))
th2 = threading.Thread(target = fun, args = ('Hi',))
th3 = threading.Thread(target = fun, args = ('Welcome',))
th4 = threading.Thread(target = fun, args = ('ByeBye',))
th1.start( )
th2.start( )
th3.start( )
th4.start( )
th1.join( )
th2.join( )
th3.join( )
th4.join( )
```

Output

```
Thread-2 : Hi
Thread-1 : Hello
Thread-2 : Hi
Thread-1 : Hello
Thread-2 : Hi
Thread-3 : Welcome
Thread-1 : Hello
Thread-2 : Hi
Thread-3 : Welcome
Thread-3 : Welcome
Thread-3 : Welcome
...
```

Tips

- From the output it is evident that the 4th thread could not invoke **fun( )**.

Problem 26.7

Write a program that runs functions **fun1( )** and **fun2( )** in two different threads. Using an event object, function **fun1( )** should wait for **fun2( )** to signal it at random intervals that its wait is over. On receiving the signal, **fun1( )** should report the time and clear the event flag.

Program

```
import threading
import random
import time

def fun1(ev, n) :
    for i in range(n) :
        print(i + 1, 'Waiting for the flag to be set...')
        ev.wait( )
        print('Wait complete at:', time.ctime( ))
        ev.clear( )
        print( )

def fun2(ev, n):
    for i in range(n):
        time.sleep(random.randrange(2, 5))
        ev.set( )

ev = threading.Event( )
th = [ ]
num = random.randrange(4, 8)
th.append(threading.Thread(target = fun1, args = (ev, num)))
th[-1].start( )
th.append(threading.Thread(target = fun2, args = (ev, num)))
th[-1].start( )
for t in th :
    t.join( )
print('All done!!')
```

Output

```
1 Waiting for the flag to be set...
Wait complete at: Sat Nov  2 11:03:43 2019
```

```
2 Waiting for the flag to be set...
Wait complete at: Sat Nov  2 11:03:45 2019

3 Waiting for the flag to be set...
Wait complete at: Sat Nov  2 11:03:48 2019

4 Waiting for the flag to be set...
Wait complete at: Sat Nov  2 11:03:52 2019

5 Waiting for the flag to be set...
Wait complete at: Sat Nov  2 11:03:54 2019

All done!!
```

Tips

- Note how the thread array is maintained using the index value '-1' to refer to the last thread added to the array.

Problem 26.8

Write a program that implements a Producer - Consumer algorithm. The producer thread should generate random numbers in the range 10 to 20. The consumer thread should print the square of the random number produced by the producer thread.

Program

```
import threading
import random
import queue
import time

def producer( ) :
    for i in range(5) :
        time.sleep(random.randrange(2, 5))
        cond.acquire( )
        num = random.randrange(10, 20)
        print('Generated number =', num)
        q.append(num)
        cond.notify( )
        cond.release( )

def consumer( ) :
    for i in range(5) :
        cond.acquire( )
        while True:
            if len(q) :
                num = q.pop( )
                break
            cond.wait( )

        print('Its square =', num * num)
        cond.release( )

cond = threading.Condition( )
q = [ ]
th1 = threading.Thread(target = producer)
th2 = threading.Thread(target = consumer)
th1.start( )
th2.start( )
th1.join( )
th2.join( )
print('All done!!')
```

Output

```
Generated number = 14
Its square = 196
Generated number = 10
Its square = 100
Generated number = 13
Its square = 169
Generated number = 15
Its square = 225
Generated number = 10
Its square = 100
All done!!
```

Tips

- Examine the program for the following possibilities and satisfy yourself that it works as per expectation in all situations:

- Producer gets a time-slot before Consumer
- Producer gets time-slot when Consumer is consuming
- Producer finishes producing before its time-slot expires
- Consumer gets a time-slot after Producer
- Consumer finishes before its time-slot expires
- Consumer gets a time-slot before Producer
- Consumer gets time-slot when Producer is busy

Exercises

[A] State whether the following statements are True or False:

(a) All multi-threaded applications should use synchronization.

(b) If 3 threads are going to read from a shared list it is necessary to synchronize their activities.

(c) A Lock acquired by one thread can be released by either the same thread or any other thread running in the application.

(d) If Lock is used in reentrant code then the thread is likely to get blocked during the second call.

(e) Lock and RLock work like a Mutex.

(f) A thread will wait on an Event object unless its internal flag is cleared.

(g) A Condition object internally uses a lock.

(h) While using RLock we must ensure that we call **release( )** as many times as the number of calls to **acquire( )**.

(i) Using Lock we can control the maximum number of threads that can access a resource.

(j) There is no difference between the synchronization objects Event and Condition.

(k) If in a Python program one thread reads a document and another thread writes to the same document then the two threads should be synchronized.

(l) If in a Python program one thread copies a document and another thread displays progress bar then the two threads should be synchronized.

(m) If in a Python program one thread lets you type a document and another thread performs spellcheck on the same document then the two threads should be synchronized.

(n) If in a Python program one thread can scan a document for viruses and another thread can pause or stop the scan then the two threads should be synchronized.

[B] Answer the following questions:

(a) Which synchronization mechanisms are used for sharing resources amongst multiple threads?

(b) Which synchronization objects are used for inter-thread communication in a multi-threaded application?

(c) What is the difference between a Lock and RLock?

(d) What is the purpose of the Semaphore synchronization primitive?

(e) Write a program that has three threads in it. The first thread should produce random numbers in the range 1 to 20, the second thread should display the square of the number generated by first thread on the screen, and the third thread should write cube of number generated by first thread into a file.

(f) Suppose one thread is producing numbers from **1** to **n** and another thread is printing the produced numbers. Comment on the output that we are likely to get.

(g) What will happen if thread **t1** waits for thread **t2** to finish and thread **t2** waits for **t1** to finish?

[C] Match the following pairs:

a. RLock	1. limits no. of threads accessing a resource
b. Event	2. useful in sharing resource in reentrant code
c. Semaphore	3. useful for inter-thread communication
d. Condition	4. signals waiting threads on change in state
e. Lock	5. useful in sharing resource among threads