08-Concurrency-Monitors-Multiprocessing

November 21, 2019

1 Monitors

A monitor is an object that only one thread can execute methods of this object at a time.

In Java Threads there is a `synchronized` modifier making a method a monitor like mutually exclusive.

In python you can use mutexes in the class to achieve that.

```
In [1]: from threading import Lock,Thread,Condition,RLock,Semaphore

        class Monitor:
            def __init__(self):
                self.mutex = RLock()
                self.count = 0

            def increment(self):
                with self.mutex:
                    self.count += 1

            def get(self):
                with self.mutex:
                    return self.count
```

```
In [3]: def f(mon):
            for i in range(10000):
                mon.increment()

        m = Monitor()

        t1 = Thread(target=f, args=(m,))
        t2 = Thread(target=f, args=(m,))
        t3 = Thread(target=f, args=(m,))
        t4 = Thread(target=f, args=(m,))

        t1.start()
```

```
    t2.start()
    t3.start()
    t4.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
    print(m.get())
```

```
40000
```

```
In [1]: class Queue:
            def __init__(self,capacity):
                self.capacity = capacity
                self.queue = []
                self.mutex = RLock()
                self.notempty = Lock(0)

            def empty(self):
                return len(self.queue) == 0

            def full(self):
                return len(self.queue) == self.capacity

            def enqueue(self,val):
                with self.mutex:
                    while self.full():
                        time.sleep(1)
                    self.queue.append(val)
                    if len(self.queue) == 1:
                        self.notempty.release()

            def dequeue(self):
                with self.mutex:
                    while self.empty():
                        self.mutex.release()
                        self.notempty.acquire()
                        self.mutex.acquire()
                    a=self.queue[0]
                    del self.queue[0]
```

2 Producer Consumer

- A queue which is accessed by two (or more) threads. One end a producer thread inserts items, the other end, consumer thread removes and processes the items.
- They work in an infinite loop.
- If queue is empty or full?

- In full and empty cases, they need to check it until queue has an empty slot or has an item respectively. Polling in an infinite loop wastes too much CPU
- Busy waiting is not a good idea:

```
while queue.empty():
    time.sleep(1)  # response time will be slow
    pass
```

- Use synchronization methods semaphores or similar to make other end know that queue is ready (not full or not empty)

3 Condition Variables

- In a monitor, condition variables let threads to signal each other while keeping the monitor semantics (only one thread inside).

```
c = Condition(mutex)
c.wait()
```

`wait` does:

```
c.mutex.release()
# block on condition
# when unblocked:
c.mutex.acquire()
```

Typical usage:

```
c.acquire() # or acquire mutex of c on construction
.....
while actual condition:
    c.wait()
```

The notifier cannot guarantee that the condition holds semantically and notified thread can directly assume condition holds.

`c.notify()` will unblock one of the threads blocking on condition.

`c.notifyAll()` will unblock all of them. However they still wait on the mutex after unblocking. They enter monitor one at a time. asdasd

```
In [7]: import random
        import time

        class PCQueue:
            def __init__(self, capacity=10):
                self.mutex=RLock()
                self.queue = []
                self.capacity = capacity
                self.notempty = Condition(self.mutex)
                self.notfull = Condition(self.mutex)
```

```
    def empty(self):
        with self.mutex:
            return len(self.queue) == 0
    def full(self):
        with self.mutex:
            return len(self.queue) == self.capacity
    def enqueue(self,item):
        with self.mutex:
            while len(self.queue) == self.capacity:
                print("queue is full, waiting")
                self.notfull.wait()

            self.queue.append(item)
            self.notempty.notify()

    def dequeue(self):
        with self.mutex:
            while len(self.queue) == 0:
                print("queue is empty, waiting")
                self.notempty.wait()

            val = self.queue[0]
            del self.queue[0]
            self.notfull.notify()
            return val


def producer(pcq):
    for i in range(30):
        time.sleep(0.15+random.random()*0.15)
        pcq.enqueue(random.randint(0,100))
        print("enqueued")
    print("producer finished")

def consumer(pcq):
    for i in range(30):
        time.sleep(0.05+random.random()*0.2)
        print("dequeued ",pcq.dequeue())
    print("consumer finished")

q = PCQueue()

prod = Thread(target=producer, args=(q,))
cons = Thread(target=consumer, args=(q,))
prod.start()
cons.start()
prod.join()
cons.join()
```

```
queue is empty, waiting
enqueued
dequeued  72
queue is empty, waiting
enqueueddequeued  4

enqueued
dequeued  94
queue is empty, waiting
enqueueddequeued
 11
queue is empty, waiting
enqueued
dequeued  12
queue is empty, waiting
enqueued
dequeued  17
queue is empty, waiting
enqueued
dequeued  76
enqueued
dequeued  96
enqueued
dequeued  87
enqueued
dequeued  81
queue is empty, waiting
enqueued
dequeued  40
queue is empty, waiting
enqueued
dequeued  0
enqueued
dequeued  30
queue is empty, waiting
enqueueddequeued
98
enqueueddequeued
28
queue is empty, waiting
enqueued
dequeued  26
queue is empty, waiting
enqueued
dequeued  73
queue is empty, waiting
enqueued
dequeued  64
```

```
queue is empty, waiting
enqueueddequeued
 36
queue is empty, waiting
dequeued enqueued 71

queue is empty, waiting
enqueued
dequeued  39
queue is empty, waiting
enqueueddequeued
 37
queue is empty, waiting
enqueueddequeued
 86
queue is empty, waiting
enqueued
dequeued  57
queue is empty, waiting
enqueued
dequeued  5
enqueued
dequeued  85
queue is empty, waiting
enqueued
dequeued  40
queue is empty, waiting
enqueued
dequeued  29
queue is empty, waiting
enqueued
dequeued  24
enqueued
producer finished
dequeued  44
consumer finished
```

In [8]:
```
prod = Thread(target=producer, args=(q,))
prod2 = Thread(target=producer, args=(q,))
cons = Thread(target=consumer, args=(q,))
cons2 = Thread(target=consumer, args=(q,))
prod.start()
cons.start()
prod2.start()
cons2.start()

prod.join()
```

```
        cons.join()
        prod2.join()
        cons2.join()
```

```
queue is empty, waiting
enqueued
dequeued  60
queue is empty, waiting
enqueued
dequeued  70
queue is empty, waiting
queue is empty, waiting
enqueued
dequeued  18
enqueued
dequeued  31
enqueued
dequeued  94
enqueued
enqueued
dequeued  38
dequeued  85
queue is empty, waiting
queue is empty, waiting
enqueued
dequeued  49
enqueueddequeued
18
enqueued
dequeued  71
queue is empty, waiting
enqueueddequeued
 69
enqueued
dequeued  47
queue is empty, waiting
queue is empty, waiting
enqueued
dequeued  94
dequeued enqueued 43

queue is empty, waiting
enqueueddequeued enqueueddequeued

9341

queue is empty, waiting
enqueued
```

```
dequeued  61
queue is empty, waiting
enqueueddequeued
 19
queue is empty, waiting
queue is empty, waiting
enqueueddequeued
 82
enqueueddequeued
 5
queue is empty, waiting
queue is empty, waiting
dequeued enqueued
 6
enqueued
dequeued  45
queue is empty, waiting
queue is empty, waiting
enqueueddequeued
 92
enqueueddequeued
97
queue is empty, waiting
enqueueddequeued  queue is empty, waitingenqueued
8


dequeued  40
queue is empty, waiting
queue is empty, waiting
enqueueddequeued
 88
enqueued
dequeued  82
queue is empty, waiting
dequeued enqueued
96
enqueued
dequeued  100
enqueued
dequeued  7
queue is empty, waiting
enqueueddequeued
 64
queue is empty, waiting
enqueued
dequeued  91
queue is empty, waiting
```

```
enqueued
dequeued  29
queue is empty, waiting
enqueueddequeued  71

enqueued
dequeued  22
queue is empty, waiting
enqueueddequeued  64

queue is empty, waiting
enqueued
dequeued  81
queue is empty, waiting
enqueued
dequeued  84
queue is empty, waiting
enqueued
dequeued  3
enqueued
dequeued  44
queue is empty, waiting
queue is empty, waiting
dequeued enqueued
30
queue is empty, waitingenqueueddequeued

 59
queue is empty, waiting
enqueued
dequeued  89
enqueueddequeued
 8
queue is empty, waiting
enqueued
dequeued  27
queue is empty, waiting
queue is empty, waiting
enqueueddequeued
 21
enqueueddequeued  queue is empty, waiting
2

queue is empty, waiting
enqueued
dequeued  53
enqueueddequeued  46
```

```
queue is empty, waiting
enqueueddequeued queue is empty, waiting

99
enqueueddequeued
 23
queue is empty, waiting
enqueued
dequeued  90
queue is empty, waiting
enqueueddequeued
56
queue is empty, waiting
queue is empty, waiting
enqueued
dequeued  88
enqueueddequeued
 42
queue is empty, waiting
enqueued
dequeued  14
enqueueddequeued queue is empty, waiting

74
consumer finished
enqueued
dequeued producer finished
43
enqueued
producer finished
dequeued  48
consumer finished
```

4 multiprocessing.Queue and Queue modules

`from multiprocessing import ..., Queue`
`from threading import ....     import Queue`
A synchronized object in shared memory. All blocking/unblocking is already implemented.

```
In [9]: from multiprocessing import Queue,Process

        q = Queue(10)

        def producer(q):
            for i in range(100):
                q.put(i)
```

```python
def consumer(q):
    for i in range(100):
        item = q.get()
        print(item)

prod=Process(target=producer, args=(q,))
cons=Process(target=consumer, args=(q,))
prod.start()
cons.start()
prod.join()
cons.join()
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

```
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

5 Process/Thread Pools

- `[i1, ... , iN]` items and apply `f()` to all of them in parallel to get `[f(i1), f(i2), ... f(N)]` as a result.
- creating `N` threads/process looks logical but resources are limited. `N == 4` it is ok but if `N == 10000`?.
- Instead create `M` processes and compute in groups of `M`.

```
In [11]: from multiprocessing import Pool

         pool = Pool(8)

         def f(i):
             time.sleep(0.2+0.3*random.random())
             return i*i

         g = pool.map(f, [i for i in range(100)])
         print(g)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441,
```

Implement your own pool?

6 Deadlock and Dining Philosophers

```
In [2]: from threading import *
        from time import *
        from random import *
```

```
STARTED = 0
THINKING = 1
HUNGRY = 2
EATING = 3
EXITTED = 4

stmess = "0?-*X"


class Philosopher(Thread):
    def __init__(self,id,forks,states, updated):
        Thread.__init__(self)
        self.id = id
        self.left = forks[0]
        self.right = forks[1]
        self.states = states
        self.states[id] = STARTED
        self.term = False
        self.updated = updated
    def terminate(self):
        self.term = True
    def run(self):
        for i in range(10):
            if self.term:
                break
#             print self.id," is thinking"
            with self.updated:
                self.states[self.id] = THINKING
                self.updated.notify()

            sleep(random()*1)
            with self.updated:
                self.states[self.id] = HUNGRY
                self.updated.notify()
            if self.id % 2 == 0:
                self.left.acquire()
                self.right.acquire()
            else:
                self.right.acquire()
                self.left.acquire()
            with self.updated:
                self.states[self.id] = EATING
                self.updated.notify()
#             print self.id," is eating"
            sleep(random()*4)
            self.left.release()
            self.right.release()
        with self.updated:
```

```python
                self.states[self.id] = EXITTED

print("Enter number of philosopher: ", end='')
n = int(input())

forks = [Lock() for i in range(n)]

phils = []

states = [0 for i in range(n)]

updated = Condition()

for i in range(n):
    phils.append( Philosopher(i,(forks[i],forks[(i+1)%n]),states, updated) )


for phil in phils:
    phil.start()

while True:
    eflag = True
    for i in range(n):
        if states[i] != EXITTED:
            eflag = False
        print(stmess[states[i]],end='')
    print()
    if eflag:
        break
    try:
        with updated:
            updated.wait()
    except KeyboardInterrupt:
        for phil in phils:
            phil.terminate()

for phil in phils:
    phil.join()
```

```
Enter number of philosopher: 4
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-2-7f73b699911b> in <module>()
```

```
         68
         69 for phil in phils:
    ---> 70     phil.start()
         71
         72 while True:


    /usr/lib/python3.5/threading.py in start(self)
    842             _limbo[self] = self
    843         try:
--> 844             _start_new_thread(self._bootstrap, ())
    845         except Exception:
    846             with _active_limbo_lock:


    RuntimeError: can't start new thread
```

7 Synchronizing/Watching a Thread/Process

- Have a condition variable for synchronization.
- Send it to Thread/Process
- In the watcher wait for it
- When the model/state changes in thread/process, notify the condition variable.

Call a function in a thread asynchronously (assume there are multiple threads, join() only joins one of them):

```
def f(x):
    return x*x

def call(c):
    with c[3]:
        c[0] = c[1](c[2])
        c[3].notify()

c = Condition()
# (result, function, input, condition)
result=[None, f, 15, c]
t = Thread(target=call, args=(result))
with c:
    c.wait()
```

```
In [12]: from threading import Thread,Condition,Lock
         import time

         class AsyncCall(Thread):
             def __init__(self,func,args):
```

```
            super().__init__()
            self.func = func
            self.args=args
            self.cond=Condition()
            self.ready = False
            self.start()
        def run(self):
            self.value = self.func(self.args)
            with self.cond:
                self.ready = True
                self.cond.notifyAll()
        def wait(self):
            with self.cond:
                while not self.ready:
                    self.cond.wait()

    def f(x):
        time.sleep(3)
        return x*x

    c = AsyncCall(f,10000)
    print("I can do usefull stuff here...")
    c.wait()
    print(c.value)
```

```
I can do usefull stuff here...
100000000
```

8 Concurrency Overview

- Watch race conditions! Use locks/semaphores to protect them
- Watch deadlocks. Be careful when holding a lock and try to acquire another.
- Never make assumptions about timing!. Timing of a thread becoming ready, calling some heavy function. OS/PL scheduler can behave undeterministically.
- Never busy wait!
- Use monitors and condition variables when you need higher level abstractions of synchronization. a monitor queue for producer consumer
- Be careful about if your data is shared or not! multiprocessing: not shared, use Value/Array/Queue threading: all globals and **object** parameters are shared
- Be careful about Global Interpreter Lock: If task is I/O intensive threading should work. but if it has cpu intensive mostly -> no parallelism. threading: lightweight, shared variables default, easy to manage but worse parallelism multiprocessing: parallel, but more expensive, needs explicit shared variables

- If a process/thread has behavior, implement as a derived class

```
class myclass(Process):   or class myclass(Thread)
```

  call `super().__init__()` in constructor override `run()` method. if a simple function, just start it.

- If only synchronization is required, your classed can be anything, implement a monitor