07-Concurrency-Threading

November 21, 2019

1 Concurrent Programming (cntd.)

1.1 threading

- `Thread` is the main class `Lock, RLock, Semaphore, Condition` are basic synchronization classes

In contrast to `multiprocessing`, threads work in the same memory environment. All global variables and parameters passed by reference to thread function are shared.

Methods and data types are conveniently similar.

Following is the ping/pong example with threading. Note the value of the counter. Also note the race condition in print(). Threads even share the same buffer for the file descriptors.

```
In [46]: import threading as th
```

```
In [ ]: counter = 10
        def ping(name,memut,othmut):
            global counter
            for i in range(0,5):
                # wait until my turn
                memut.acquire()
                print(name,counter)
                # tell other end it is its turn
                othmut.release()
                counter += 1


        imut,omut = th.Lock(), th.Lock()

        pip = th.Thread(target=ping, args=("ping",imut,omut))
        pop = th.Thread(target=ping, args=("pong",omut,imut))

        # make sure only one (ping enters first)
        omut.acquire()
        pip.start()
        pop.start()
        pip.join()
        pop.join()
```

```
        omut.release()
        print("in main thread: {}".format(counter))
```

1.2 A Race Condition Example

```
In [51]: ''' Race condition of threads.
             uncomment mut to use a mutex.
             You need long executing threads with
             multi-line critical regions to observe it.'''
         mut = th.Lock()

         def f(lst):
             for i in range(0,50000):
                 # the following is the critical region
                 with mut:
                 mut.acquire()
                 tmp = lst[0]
                 tmp += 1
                 lst[0] = tmp
                 lst[0] += 1
                 mut.release()


         arr = [0]

         threads = [th.Thread(target=f, args=(arr,)) for i in range(0,10)]

         print(arr)
         for t in threads:
             t.start()

         for t in threads:
             t.join()

         print(arr)


          File "<ipython-input-51-00e3ec59c55a>", line 11
        mut.acquire()
          ^
    IndentationError: expected an indented block
```

1.3 Shared Variables in `multiprocessing`

Shared globals and heap is provided by its nature in `threading`. In multiprocessing standard variables except synchronization primitives are isolated in each process. In order to share values

`Value` and `Array` classes provide shared variables with atomic access that multiple process can use.

- `Value` constructor needs a type specifier argument either a `ctypes` type or a character representing it: > `'i'` equivalent to `c_int` > > `'c'` equivalent to `c_char` > > `'f'` equivalent to `c_float` > > `'d'` equivalent to `c_double` >

  Depending on the parameter a shared variable with given type is created.

- Use `value` member of the `Value` objects to get/update their value.

- `Array` creates homogeneous list of values. Constructor gets a type specifier and either a size or an iterator: > `Array('d',10)` > > `Array('i',(0,0,1,0,0,1,0,0,1))`

```
In [40]: from multiprocessing import Value,Array

         a = Value('i',3)   # a shared integer
         a.value += 1
         print(a)

         v = Array('d',10)  # a shared array of doubles
         for i in range(len(v)):
             v[i] = i                    # set and get like a list
         v[:] = [0,3,4,5,2,3,4,5,1,9]    # can be updated with slices
         print(list(v))                  # is iteratable

         m = Array('c', b'Hello World')   # a character buffer (size of Hello World)
         m.value = b'hello world'   # it will overflow with larger strings
         print(m.value)
```

```
<Synchronized wrapper for c_int(4)>
[0.0, 3.0, 4.0, 5.0, 2.0, 3.0, 4.0, 5.0, 1.0, 9.0]
b'hello world'
```

1.4 Lock versus RLock

`RLock` is called a recursive lock. Basically it is a lock only holder can release. In `Lock`, any thread can release a lock like in our ping/pong example above. In recursive lock, only thread/process that acquired it can release it.

```
In [ ]: a, b = th.Lock(), th.RLock()

        def f():
            a.acquire()
            # This will give error
            b.release()

        f()
```

1.5 Using `with` with synchronization

- `with` provides a semantic context for a data type derived from context manager
- `with` block calls `_enter_` and `_exit_` methods of the context.
- This way, statement blocks that are valid in the context can be defined as:

```
with open("myfile","r") as fd:
    line = fd.readline()
    while line:
        print(line, end='')
        line = fd.readline()
print('finished')
```

- context manager for a file object automatically closes the file when block is terminated.
- similarly database connections can use `with`

```
with sqlite3.connect('myfile.sqlite3') as cur:
    cur.execute('....')
    ...
    ....

# here connection is closed
```

Synchronization classes `Lock, RLock, Semaphore, Condition` can be used similarly:

```
a=Lock()
with a:
    y = x + 1
    x = y
```

Is equivalent to:

```
a=Lock()
try:
    a.acquire()
    y = x + 1
    x = y
finally:
    a.release()
```

Exceptions are handled properly. This form emphasizes the critical regions better.

2 Barriers

- A barrier is a meeting point for all threads. For example each thread makes some initializations and when and only when all of them is ready, the algorithm can start.

```
In [3]: import random
        import time
```

```
class MyBarrier:
    def __init__(self, number = 2):
        self.number = number
        self.mutex = th.RLock()
        self.go = th.Lock()
        self.go.acquire()
        self.current = 0

    def arrived(self):
        self.mutex.acquire()
        self.current += 1
        print("arrived {}th".format(self.current))
        if self.current == self.number:
            self.current -= 1
            self.mutex.release()
            self.go.release()
        else:
            self.mutex.release()
            self.go.acquire()
            self.mutex.acquire()
            self.current -= 1
            if self.current > 0:
                self.go.release()
            self.mutex.release()
        print("left {}".format(self.current))

def barriertest(barrier):
    time.sleep(1 + random.random()*3)
    barrier.arrived()
    time.sleep(1 + random.random()*5)
    print("completed")
    # use same barrier for completion
    barrier.arrived()

bar = MyBarrier(4)

t1 = th.Thread(target=barriertest, args=(bar,))
t2 = th.Thread(target=barriertest, args=(bar,))
t3 = th.Thread(target=barriertest, args=(bar,))
t4 = th.Thread(target=barriertest, args=(bar,))

t1.start()
t2.start()
t3.start()
t4.start()

t1.join()
```

```
        t2.join()
        t3.join()
        t4.join()
        print("all terminated")
```

```
arrived 1th
arrived 2th
arrived 3th
arrived 4th
left 2left 3

left 1
left 0
completed
arrived 1th
completed
arrived 2th
completed
arrived 3th
completedleft 2left 1


left 0arrived 4thall terminated


left 3
```