Fork me on GitHub

Concurrency

Execute a shell command

```
# get stdout, stderr, returncode

>>> from subprocess import Popen, PIPE
>>> args = ['time', 'echo', 'hello python']
>>> ret = Popen(args, stdout=PIPE, stderr=PIPE)
>>> out, err = ret.communicate()
>>> out
b'hello python\n'
>>> err
b'        0.00 real         0.00 user         0.00 sys\n'
>>> ret.returncode
0
```

Create a thread via “threading”

```
>>> from threading import Thread
>>> class Worker(Thread):
...   def __init__(self, id):
...     super(Worker, self).__init__()
...     self._id = id
...   def run(self):
...     print("I am worker %d" % self._id)
...
>>> t1 = Worker(1)
>>> t2 = Worker(2)
>>> t1.start(); t2.start()
I am worker 1
I am worker 2

# using function could be more flexible
>>> def Worker(worker_id):
...   print("I am worker %d" % worker_id)
...
>>> from threading import Thread
>>> t1 = Thread(target=Worker, args=(1,))
>>> t2 = Thread(target=Worker, args=(2,))
>>> t1.start()
I am worker 1
I am worker 2
```

Performance Problem - GIL

```
# GIL - Global Interpreter Lock
# see: Understanding the Python GIL
>>> from threading import Thread
>>> def profile(func):
...   def wrapper(*args, **kwargs):
...     import time
...     start = time.time()
...     func(*args, **kwargs)
...     end   = time.time()
...     print(end - start)
...   return wrapper
...
>>> @profile
... def nothread():
```

```
...   fib(35)
...   fib(35)
...
>>> @profile
... def hasthread():
...   t1=Thread(target=fib, args=(35,))
...   t2=Thread(target=fib, args=(35,))
...   t1.start(); t2.start()
...   t1.join(); t2.join()
...
>>> nothread()
9.51164007187
>>> hasthread()
11.3131771088
# !Thread get bad Performance
# since cost on context switch
```

Consumer and Producer

```
# This architecture make concurrency easy
>>> from threading import Thread
>>> from Queue import Queue
>>> from random import random
>>> import time
>>> q = Queue()
>>> def fib(n):
...   if n<=2:
...     return 1
...   return fib(n-1)+fib(n-2)
...
>>> def producer():
...   while True:
...     wt = random()*5
...     time.sleep(wt)
...     q.put((fib,35))
...
>>> def consumer():
...   while True:
...     task,arg = q.get()
...     print(task(arg))
...     q.task_done()
...
>>> t1 = Thread(target=producer)
>>> t2 = Thread(target=consumer)
>>> t1.start();t2.start()
```

Thread Pool Template

```
# producer and consumer architecture
from Queue import Queue
from threading import Thread

class Worker(Thread):
   def __init__(self,queue):
      super(Worker, self).__init__()
      self._q = queue
      self.daemon = True
      self.start()
   def run(self):
      while True:
         f,args,kwargs = self._q.get()
         try:
            print(f(*args, **kwargs))
```

```
            except Exception as e:
                print(e)
            self._q.task_done()

class ThreadPool(object):
    def __init__(self, num_t=5):
        self._q = Queue(num_t)
        # Create Worker Thread
        for _ in range(num_t):
            Worker(self._q)
    def add_task(self,f,*args,**kwargs):
        self._q.put((f, args, kwargs))
    def wait_complete(self):
        self._q.join()

def fib(n):
    if n <= 2:
        return 1
    return fib(n-1)+fib(n-2)

if __name__ == '__main__':
    pool = ThreadPool()
    for _ in range(3):
        pool.add_task(fib,35)
    pool.wait_complete()
```

Using multiprocessing ThreadPool

```
# ThreadPool is not in python doc
>>> from multiprocessing.pool import ThreadPool
>>> pool = ThreadPool(5)
>>> pool.map(lambda x: x**2, range(5))
[0, 1, 4, 9, 16]
```

Compare with “map” performance

```
# pool will get bad result since GIL
import time
from multiprocessing.pool import \
     ThreadPool

pool = ThreadPool(10)
def profile(func):
    def wrapper(*args, **kwargs):
       print(func.__name__)
       s = time.time()
       func(*args, **kwargs)
       e = time.time()
       print("cost: {0}".format(e-s))
    return wrapper

@profile
def pool_map():
    res = pool.map(lambda x:x**2,
                   range(999999))

@profile
def ordinary_map():
    res = map(lambda x:x**2,
              range(999999))

pool_map()
ordinary_map()
```

output:

```
$ python test_threadpool.py
pool_map
cost: 0.562669038773
ordinary_map
cost: 0.38525390625
```

Mutex lock

Simplest synchronization primitive lock

```
>>> from threading import Thread
>>> from threading import Lock
>>> lock = Lock()
>>> def getlock(id):
...   lock.acquire()
...   print("task{0} get".format(id))
...   lock.release()
...
>>> t1=Thread(target=getlock,args=(1,))
>>> t2=Thread(target=getlock,args=(2,))
>>> t1.start();t2.start()
task1 get
task2 get

# using lock manager
>>> def getlock(id):
...   with lock:
...     print("task%d get" % id)
...
>>> t1=Thread(target=getlock,args=(1,))
>>> t2=Thread(target=getlock,args=(2,))
>>> t1.start();t2.start()
task1 get
task2 get
```

Deadlock

Happen when more than one mutex lock.

```
>>> import threading
>>> import time
>>> lock1 = threading.Lock()
>>> lock2 = threading.Lock()
>>> def task1():
...   with lock1:
...     print("get lock1")
...     time.sleep(3)
...     with lock2:
...       print("No deadlock")
...
>>> def task2():
...   with lock2:
...     print("get lock2")
...     with lock1:
...       print("No deadlock")
...
>>> t1=threading.Thread(target=task1)
>>> t2=threading.Thread(target=task2)
>>> t1.start();t2.start()
get lock1
```

```
 get lock2

>>> t1.isAlive()
True
>>> t2.isAlive()
True
```

Implement “Monitor”

Using RLock

```python
# ref: An introduction to Python Concurrency - David Beazley
from threading import Thread
from threading import RLock
import time

class monitor(object):
   lock = RLock()
   def foo(self,tid):
      with monitor.lock:
         print("%d in foo" % tid)
         time.sleep(5)
         self.ker(tid)

   def ker(self,tid):
      with monitor.lock:
         print("%d in ker" % tid)
m = monitor()
def task1(id):
   m.foo(id)

def task2(id):
   m.ker(id)

t1 = Thread(target=task1,args=(1,))
t2 = Thread(target=task2,args=(2,))
t1.start()
t2.start()
t1.join()
t2.join()
```

output:

```
$ python monitor.py
1 in foo
1 in ker
2 in ker
```

Control primitive resources

Using Semaphore

```python
from threading import Thread
from threading import Semaphore
from random    import random
import time

# limit resource to 3
sema = Semaphore(3)
def foo(tid):
    with sema:
        print("%d acquire sema" % tid)
```

```
        wt = random()*5
        time.sleep(wt)
    print("%d release sema" % tid)

threads = []
for _t in range(5):
    t = Thread(target=foo,args=(_t,))
    threads.append(t)
    t.start()
for _t in threads:
    _t.join()
```

output:

```
python semaphore.py
0 acquire sema
1 acquire sema
2 acquire sema
0 release sema
 3 acquire sema
2 release sema
 4 acquire sema
1 release sema
4 release sema
3 release sema
```

Ensure tasks has done

Using ‘event’

```
from threading import Thread
from threading import Event
import time

e = Event()

def worker(id):
   print("%d wait event" % id)
   e.wait()
   print("%d get event set" % id)

t1=Thread(target=worker,args=(1,))
t2=Thread(target=worker,args=(2,))
t3=Thread(target=worker,args=(3,))
t1.start()
t2.start()
t3.start()

# wait sleep task(event) happen
time.sleep(3)
e.set()
```

output:

```
python event.py
1 wait event
2 wait event
3 wait event
2 get event set
 3 get event set
1 get event set
```

Thread-safe priority queue

Using ‘condition’

```
import threading
import heapq
import time
import random

class PriorityQueue(object):
    def __init__(self):
        self._q = []
        self._count = 0
        self._cv = threading.Condition()

    def __str__(self):
        return str(self._q)

    def __repr__(self):
        return self._q

    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._q, (-priority,self._count,item))
            self._count += 1
            self._cv.notify()

    def pop(self):
        with self._cv:
            while len(self._q) == 0:
                print("wait...")
                self._cv.wait()
            ret = heapq.heappop(self._q)[-1]
        return ret

priq = PriorityQueue()
def producer():
    while True:
        print(priq.pop())

def consumer():
    while True:
        time.sleep(3)
        print("consumer put value")
        priority = random.random()
        priq.put(priority,priority*10)

for _ in range(3):
    priority = random.random()
    priq.put(priority,priority*10)

t1=threading.Thread(target=producer)
t2=threading.Thread(target=consumer)
t1.start();t2.start()
t1.join();t2.join()
```

output:

```
python3 thread_safe.py
0.6657491871045683
0.5278797439991247
0.20990624606296315
wait...
consumer put value
```

```
0.09123101305407577
wait...
```

Multiprocessing

Solving GIL problem via processes

```
>>> from multiprocessing import Pool
>>> def fib(n):
...     if n <= 2:
...         return 1
...     return fib(n-1) + fib(n-2)
...
>>> def profile(func):
...     def wrapper(*args, **kwargs):
...         import time
...         start = time.time()
...         func(*args, **kwargs)
...         end   = time.time()
...         print(end - start)
...     return wrapper
...
>>> @profile
... def nomultiprocess():
...     map(fib,[35]*5)
...
>>> @profile
... def hasmultiprocess():
...     pool = Pool(5)
...     pool.map(fib,[35]*5)
...
>>> nomultiprocess()
23.8454811573
>>> hasmultiprocess()
13.2433719635
```

Custom multiprocessing map

```
from multiprocessing import Process, Pipe
from itertools import izip

def spawn(f):
    def fun(pipe,x):
        pipe.send(f(x))
        pipe.close()
    return fun

def parmap(f,X):
    pipe=[Pipe() for x in X]
    proc=[Process(target=spawn(f),
          args=(c,x))
          for x,(p,c) in izip(X,pipe)]
    [p.start() for p in proc]
    [p.join() for p in proc]
    return [p.recv() for (p,c) in pipe]

print(parmap(lambda x:x**x,range(1,5)))
```

Graceful way to kill all child processes

```
from __future__ import print_function

import signal
import os
import time

from multiprocessing import Process, Pipe

NUM_PROCESS = 10

def aurora(n):
    while True:
        time.sleep(n)

if __name__ == "__main__":
    procs = [Process(target=aurora, args=(x,))
                for x in range(NUM_PROCESS)]
    try:
        for p in procs:
            p.daemon = True
            p.start()
        [p.join() for p in procs]
    finally:
        for p in procs:
            if not p.is_alive(): continue
            os.kill(p.pid, signal.SIGKILL)
```

Simple round-robin scheduler

```
>>> def fib(n):
...   if n <= 2:
...     return 1
...   return fib(n-1)+fib(n-2)
...
>>> def gen_fib(n):
...   for _ in range(1,n+1):
...     yield fib(_)
...
>>> t=[gen_fib(5),gen_fib(3)]
>>> from collections import deque
>>> tasks = deque()
>>> tasks.extend(t)
>>> def run(tasks):
...   while tasks:
...     try:
...       task = tasks.popleft()
...       print(task.next())
...       tasks.append(task)
...     except StopIteration:
...       print("done")
...
>>> run(tasks)
1
1
1
1
2
2
3
done
5
done
```

Scheduler with blocking function

```
# ref: PyCon 2015 - David Beazley
import socket
from select import select
from collections import deque

tasks  = deque()
r_wait = {}
s_wait = {}

def fib(n):
    if n <= 2:
        return 1
    return fib(n-1)+fib(n-2)

def run():
    while any([tasks,r_wait,s_wait]):
        while not tasks:
            # polling
            rr, sr, _ = select(r_wait, s_wait, {})
            for _ in rr:
                tasks.append(r_wait.pop(_))
            for _ in sr:
                tasks.append(s_wait.pop(_))
        try:
            task = tasks.popleft()
            why, what = task.next()
            if why == 'recv':
                r_wait[what] = task
            elif why == 'send':
                s_wait[what] = task
            else:
                raise RuntimeError
        except StopIteration:
            pass

def fib_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('localhost',5566))
    sock.listen(5)
    while True:
        yield 'recv', sock
        c, a = sock.accept()
        tasks.append(fib_handler(c))

def fib_handler(client):
    while True:
        yield 'recv', client
        req  = client.recv(1024)
        if not req:
            break
        resp = fib(int(req))
        yield 'send', client
        client.send(str(resp)+'\n')
    client.close()

tasks.append(fib_server())
run()
```

output: (bash 1)

```
$ nc loalhost 5566
20
```

```
6765
```

output: (bash 2)

```
$ nc localhost 5566
10
55
```

PoolExecutor

```
# python2.x is module futures on PyPI
# new in Python3.2
>>> from concurrent.futures import \
...     ThreadPoolExecutor
>>> def fib(n):
...     if n<=2:
...         return 1
...     return fib(n-1) + fib(n-2)
...
>>> with ThreadPoolExecutor(3) as e:
...     res= e.map(fib,[1,2,3,4,5])
...     for _ in res:
...         print(_, end=' ')
...
1 1 2 3 5 >>>
# result is generator?!
>>> with ThreadPoolExecutor(3) as e:
...   res = e.map(fib, [1,2,3])
...   inspect.isgenerator(res)
...
True

# demo GIL
from concurrent import futures
import time

def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)

def thread():
    s = time.time()
    with futures.ThreadPoolExecutor(2) as e:
        res = e.map(fib, [35]*2)
        for _ in res:
            print(_)
    e = time.time()
    print("thread cost: {}".format(e-s))

def process():
    s = time.time()
    with futures.ProcessPoolExecutor(2) as e:
        res = e.map(fib, [35]*2)
        for _ in res:
            print(_)
    e = time.time()
    print("pocess cost: {}".format(e-s))


# bash> python3 -i test.py
>>> thread()
9227465
9227465
```

```
thread cost: 12.550225019454956
>>> process()
9227465
9227465
pocess cost: 5.538189888000488
```

How to use `ThreadPoolExecutor`?

```
from concurrent.futures import ThreadPoolExecutor

def fib(n):
    if n <= 2:
        return 1
    return fib(n - 1) + fib(n - 2)

with ThreadPoolExecutor(max_workers=3) as ex:
    futs = []
    for x in range(3):
        futs.append(ex.submit(fib, 30+x))

    res = [fut.result() for fut in futs]

print(res)
```

output:

```
$ python3 thread_pool_ex.py
[832040, 1346269, 2178309]
```

What does “with ThreadPoolExecutor” work?

```
from concurrent import futures

def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)

with futures.ThreadPoolExecutor(3) as e:
    fut = e.submit(fib, 30)
    res = fut.result()
    print(res)

# equal to

e = futures.ThreadPoolExecutor(3)
fut = e.submit(fib, 30)
fut.result()
e.shutdown(wait=True)
print(res)
```

output:

```
$ python3 thread_pool_exec.py
832040
832040
```

Future Object

```python
# future: deferred computation
# add_done_callback
from concurrent import futures

def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)

def handler(future):
    res = future.result()
    print("res: {}".format(res))

def thread_v1():
    with futures.ThreadPoolExecutor(3) as e:
        for _ in range(3):
            f = e.submit(fib, 30+_)
            f.add_done_callback(handler)
    print("end")

def thread_v2():
    to_do = []
    with futures.ThreadPoolExecutor(3) as e:
        for _ in range(3):
            fut = e.submit(fib, 30+_)
            to_do.append(fut)
        for _f in futures.as_completed(to_do):
            res = _f.result()
            print("res: {}".format(res))
    print("end")
```

output:

```
$ python3 -i fut.py
>>> thread_v1()
res: 832040
res: 1346269
res: 2178309
end
>>> thread_v2()
res: 832040
res: 1346269
res: 2178309
end
```

Future error handling

```python
from concurrent import futures

def spam():
    raise RuntimeError

def handler(future):
    print("callback handler")
    try:
        res = future.result()
    except RuntimeError:
        print("get RuntimeError")

def thread_spam():
    with futures.ThreadPoolExecutor(2) as e:
        f = e.submit(spam)
        f.add_done_callback(handler)
```

output:

```
$ python -i fut_err.py
>>> thread_spam()
callback handler
get RuntimeError
```