Python Talk 10

多线程编程

- 多线程编程可以让程序更加灵活
  - e.g. 一个网络服务器需要处理很多个客户端的请求，但每个请求之间没有很大的关联
- 多线程在某些情况下可以提升程序效率
  - e.g. 一个程序需要处理很多互相独立的图像，此时可以利用多个 CPU 的优势平行处理
- 多线程程序相对容易出错、难以调试
- 为了防止竞争条件，需要用锁、条件变量等进行控制
- Python 的 `threading` 库支持多线程编程

计算一百万个 1 相加

- 单线程做法

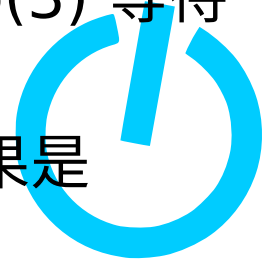
```
s = 0
for i in range(1000000) :
    s += 1
print(s)
```

- 多线程做法
  - 开 10 个线程，共享变量 s；每个线程进行 100000 次加法

多线程程序

```
import time, threading

s = 0

class MyThread(
    threading.Thread) :
  def run(self) :
    global s
    for i in range(100000) :
      s += 1

if __name__ == '__main__' :
  for i in range(10) :
    MyThread().start()
  time.sleep(3)
  print(s)
```

- import：引入 threading 库
- threading.Thread：线程的基类
  - run() 为线程的执行内容
  - 调用 start() 以开始线程
  - 如果调用 run() 就不是多线程调用了
- global 表示引用全局变量
- 通过 time.sleep(3) 等待所有线程结束
- 问题：得到的结果是 1000000 吗?



竞争条件

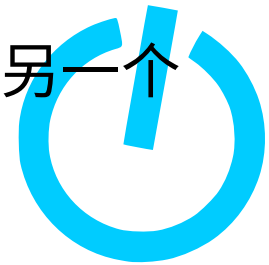
- 看似简单的 `s += 1` 其实包含多个指令

```
load  register,        memory[0x1234]
add   register,        1
store memory[0x1234], register
```

- 翻译成 Python 就是

```
reg = mem[0x1234]    # load
reg += 1             # execute
mem[0x1234] = reg    # store
```

- CPU 可能在任何两个指令中间暂停执行，并切换到另一个线程



竞争条件

- 思考如下左右两个线程的执行先后顺序，时间从上到下

```
# Thread 1                  # Thread 2
                                                  # mem[] = 0
reg1 = memory[]                                   # reg1 = 1
                            reg2 = memory[]       # reg2 = 1
reg1 += 1                                         # reg1 = 2
                            reg2 += 1             # reg2 = 2
mem[] = reg1                                      # mem[] = 2
                            mem[] = reg2          # mem[] = 2
```

- 竞争条件（race condition）的定义：两个线程同时访问一个内存地址，其中至少一个访问是写入
- 竞争条件有很多解决方法
  - 其中最常用的是通过锁控制访问

锁

- 锁（Lock）是一种特殊变量，每个线程需要使用某些资源时获取锁，在用完后释放锁
- 通过各种硬件和软件机制，任何时刻只能有最多一个线程获得锁
  - 如果线程尝试获取被占用的锁，一直等待到锁被释放

```
lock = threading.Lock() # 创建锁
lock.acquire()          # 获取锁
lock.release()          # 释放锁
```

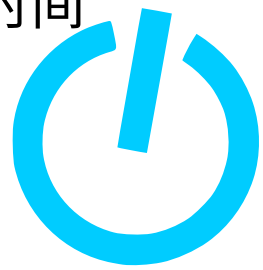
锁示例

```
import time, threading

s = 0
s_lock = threading.Lock()

class MyThread(
    threading.Thread) :
  def run(self) :
    global s, s_lock
    for i in range(100000) :
      s_lock.acquire()
      s += 1
      s_lock.release()

if __name__ == '__main__' :
  for i in range(10) :
    MyThread().start()
  time.sleep(5)
  print(s)
```

- 锁 s_lock 用于管理对 s 的访问
- 访问 s 之前用 acquire 获取锁
- 访问 s 之后用 release 获取锁
- 程序的运算时间有所增长，因此 sleep 从 3 秒变为 5 秒
  - 因为当另一个线程在访问 s 时，当前线程需要等待
  - 如何避免使用 sleep?
  - 真实的程序中无法预测线程执行时间

条件变量

- 实现线程之间的信号传递
- 问题：线程 A 希望读取变量 X，但是需要在线程 B 写入变量之后进行。X 的访问通过 X_lock 进行限制，通过 X_ready 标明是否已经写入。
- 实现 1
  - 线程 A 通过一个循环，不断获取 X_lock 并通过 X_ready 判断是否可以读取 X
  - 问题：如果 X 一直没有被写入，A 会进行不停的无用计算
- 实现 2
  - A 的每次循环后用 `time.sleep` 等待一个很短的时间
  - 问题：虽然等待很短，但还是有延迟
- 实现 3
  - 线程 B 在写入后“通知”线程 A

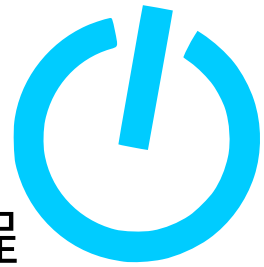


条件变量

- 条件变量允许线程传递“某个事件发生”的信息
- 条件变量需要和锁共同使用，当一个线程使用条件变量时必须已获得对应的锁

```
cv = threading.Condition(lock)
lock.acquire()          # 获取 lock 后才能使用 cv
cv.wait()               # 等待一个事件的发生
cv.notify()             # 通知一个等待的线程
cv.notify_all()         # 通知所有等待的线程
lock.release()
```

- `cv.wait()` 返回时不保证是被 `notify` 的
  - 因此应该用一个 `while` 循环重新判断是否等待
- `cv.wait()` 时会释放 `lock`，否则会阻塞其它线程



条件变量示例

- 线程 A 读取 X，线程 B 写入 X

```
X = 0
X_lock = threading.Lock()
X_ready = False
X_cv = threading.Condition(X_lock)
class ThreadA(threading.Thread) :
    def run(self) :
        X_lock.acquire()
        while not X_ready :
            X_cv.wait()
        print(X)
        X_lock.release()
class ThreadB(threading.Thread) :
    def run(self) :
        X_lock.acquire()
        X = 1234
        X_cv.notify()
        X_lock.release()
```

with

- Python 中 with 有多种用途，在锁中可以用于获取一个锁

```
lock = threading.Lock()
with lock :
    # 锁被获取
    # Do something
# 锁被释放
```

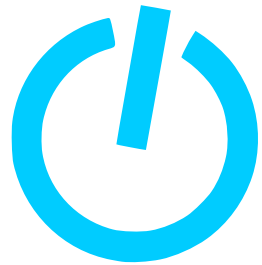
- 等价于

```
lock = threading.Lock()
lock.acquire()
# 锁被获取
# Do something
lock.release()
# 锁被释放
```

条件变量示例 2

- 让主线程等待所有线程退出

```
r = 0
r_lock = threading.Lock()
r_cv = threading.Condition(r_lock)

class MyThread(threading.Thread) :
    def run(self) :
        global r, r_lock, r_cv
        ...
        with r_lock :
            r -= 1
            r_cv.notify()

if __name__ == '__main__' :
    for i in range(10) :
        with r_lock :
            r += 1
        MyThread().start()
    with r_lock :
        while r :
            r_cv.wait()
```



练习

1. 实现一个线程安全的队列
   - 队列的长度限制为 n
   - 对于 `enqueue(elem)` 操作，如果队列不到 n 个元素，将 `elem` 插入队列；如果队列已经有 n 个元素，阻塞至队列长度变小
   - 对于 `dequeue()` 操作，如果队列中有元素，返回队列中最早被插入的元素；否则阻塞至队列长度变大
2. 实现一个主要是读的读写锁
   - 读写锁允许很多个读锁定线程，但是不可以多个写阻塞线程
   - 读和写锁定不能同时存在
   - 读锁定和解锁应该尽可能被优化
   - 写不能被长时间阻塞（当没有读时写不能被阻塞）
   - 方法：`read_acquire()`, `read_release()`, `write_acquire()`, `write_release()`

参考资料

- Operating Systems - Principles & Practice, Second Edition, by Thomas Anderson and Mike Dahlin
- https://docs.python.org/3/library/threading.html

感谢参加此次活动