多线程与多进程编程
Part I

马栩杰

2016 年 7 月 2 日

讲义 Slides 版见 `https://cdn.rawgit.com/ma-xujie/reveal.js/multithreading-1/index.html`

1 线程与进程

1.1 背景

现代的操作系统都支持多任务，在一个任务执行的过程中可以同时进行另一个任务。实际上经常会有开着 IDE 写代码，同时打开浏览器查文档，同时开着音乐播放器等等一大堆程序同时运行的情况。这种多任务同时运行显然不是因为有多核 CPU 所以每个核上跑一个任务。

操作系统在运行时不仅仅可以同时并行执行很多个独立的程序，而且，与我们自己编译的 C 和 C++ 小黑框程序不同，我们日常使用的很多软件单个程序也在同一时刻执行着很多任务。比如网页浏览器就可以在浏览网页的同时下载东西。

由于有进程和线程的存在，才使得系统可以同时执行多个程序，一个程序可以同时执行多个任务。

1.2 进程

程序是一段指示计算机每一步执行动作的指令序列，单独的程序不能自己运行。

向操作系统发出执行一段程序的命令以后，程序被放入计算机内存运行，就产生了进程。

进程（Process）是计算机中运行程序的实体，一个程序至少产生一个进程，可以产生多个进程。每个进程都有独立的虚拟地址空间。两个不同进程之间的变量所在的内存是相互独立的，指针（C/C++）和全局变量不能共享。

一个进程需要系统分配 CPU 、内存、文件、I/O 设备等资源以后才能工作。

1.3 线程

一个线程（Thread）指的是进程中一个单一顺序的控制流，一个进程中可以同时执行多个线程，每条线程并行执行不同的任务。

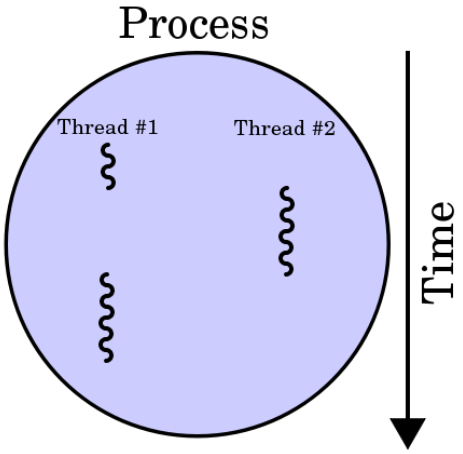
线程被包含在进程之中，是进程中的实际运作单位。

线程是操作系统能够进行运算调度的最小单位。

每个线程都有独立的寄存器和栈空间，但是同一个进程内的各个线程共享虚拟地址，因此一个进程内的所有线程的内存是共享的。

1.4 多任务并行的运行方式

在操作系统中，通常都有很多程序同时运行。每个程序有 ≥ 1 个进程，每个进程中有 ≥ 1 个线程执行。但是，本质上 CPU 的一个核心在某一特定时刻只能执行一个任务。那么问题来了，虽然有这么多线程同时运行，但是使用者却往往感受不到它们相互之间有什么影响，这是怎么做到的？实际上操作系统的做法是让 CPU 以很快的速度在不同的线程之间来回切换，在一段时间内，可以做到每个线程都执行一小段时间。在使用者看来，就是在同时执行很多任务。



1.5 多进程与多线程编程

在写复杂程序的时候，同时执行多个任务是不可避免的。这时就需要在程序中使用多进程/多线程编程。

例如：队式程序在运行时可能会需要同时运行 3D 界面、游戏逻辑、以及调用两个选手的程序，而其中每个模块都需要在一个独立的控制流下运行。

需要在程序中同时进行多个任务的时候，一般有以下 3 种实现方式：

- 多进程
- 多线程
- 多进程 + 多线程

我们以队式 17 为例：

- 进程 1：平台
  - 线程 1：平台主程序
  - 线程 2：逻辑
  - 线程 3：回放生成器
  - 线程 4：选手 1
  - 线程 5：选手 2
  - 线程 6：socket 发送
  - 线程 7：socket 接收
- 进程 2：3D 播放器
  - 线程 8：播放器主程序
  - 线程 9：socket 收发
  - 线程 10：Unreal 启动的各种线程
  - 线程 11：...

嗯我们可以看到，队式 17 主要是以多线程的方式实现的...

2 多线程编程

那么接下来先来谈谈多线程的编程实现吧。[1]

2.1 简单的多线程程序

在 Python 中，多线程的实现非常容易。

```
import threading
def foo():
    print('Hello, world!')

t = threading.Thread(target=foo)  # 创建线程
t.start()  # 启动线程
# 然后就可以干别的事情了
```

在 Python 中启动多线程只需要 import 一下 threading 库，创建一个 Thread 对象，然后启动就可以啦。

下面看看 Thread 的参数：

```
threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
```

其中，group 并没有什么卵用，target 是 Thread 运行时调用的函数（如果是 None 的话就什么也不做了），name 是给这个线程起的名字（一个字符串），args 是要传入 target 函数的所有参

[1]以 Python 为例讲解，不熟悉 Python 也可以把它们当作伪代码来看

数（tuple），kwargs 是传入 target 的命名参数（dict），daemon 参数显式指定是否启动为守护线程，daemon = True 则主线程结束时所有子线程也中断结束，如果 deamon = False(默认)，则主线程退出时如果子线程还没有运行完则子线程继续运行。

再看一个例子：

```
import threading
import time

def foo():
    # 可以用 threading.current_thread() 获取当前所在的 Thread 对象
    print('This is thread %s.' % threading.current_thread().name)
    for i in range(5):
        print('hehe :-)')
    time.sleep(1)
    print('Thread %s end.' % threading.current_thread().name)

t = threading.Thread(target=foo, name='hehe')
t.start()

print('This is thread %s.' % threading.current_thread().name)
print('While %s is doing something, I can do something else.' % t.name)
print('Wait until %s end.' % t.name)

t.join()
print('This is thread %s, %s ended.' % (threading.current_thread().name, t.name))
```

运行结果：

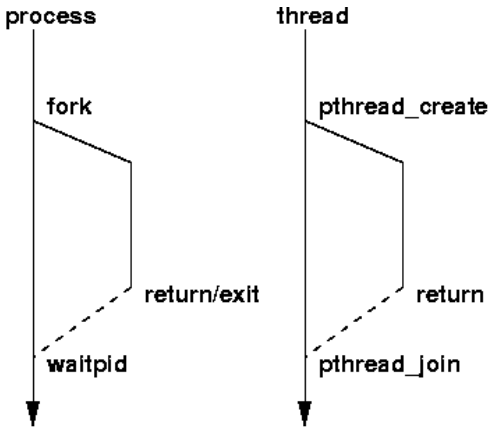
```
This is thread hehe.
hehe :-)
This is thread MainThread.
hehe :-)
While hehe is doing something, I can do something else.
hehe :-)
Wait until hehe end.
hehe :-)
hehe :-)
Thread hehe end.
This is thread MainThread, hehe ended.
```

可以看到在启动了线程 hehe 之后，主线程并没有中断运行，而是开始和 hehe “同时” 运行。当主线程调用 t.join() 时，才开始挂起等待 hehe 运行结束。

Thread 类提供了几个方法：

```
t = threading.Thread(target=foo)
t.start()  # 启动线程，只能调用一次
t.join()  # 阻塞当前所在的线程，等待 t 结束运行
```

对启动新线程的一种理解方式是从当前所在的控制流中分支（fork）出一条控制流，而 join() 就表示在这里与分出去的控制流再合并为顺序执行的一个控制流，也就是挂起当前线程等待另一个线程执行结束。



如果不想永远等下去的话也可以在 join 中输入参数 `t.join(timeout=1)`，表示等待 1s。在这种情况下需要用 `t.is_alive()` 检查等待的线程是否真的在 timeout 内结束运行。

Python 不提供直接获得 thread 所运行函数返回值的方法，因此如果想得到多线程运算的结果的话，就需要采用其他的方法。

2.2 线程间的内存共享

之前提到，同一个进程内的多个线程是共享虚拟地址的，如果在程序中有一个所有函数都可以访问的全局变量的话，那么是不是就可以将函数的返回值写进全局变量中呢?

对于仅有一个线程写这个变量的情况下，没错。但是在实践中，有时会出现多个线程同时要读写一个变量的状况，这时候情况就变得有些复杂..

看这样一段代码:

```
import threading
import time

cnt = 0

def foo():
    global cnt
    for i in range(100000):
        cnt = cnt + 1

threads = []
for i in range(5):
```

```
    threads.append(threading.Thread(target=foo))
    threads[-1].start()

for t in threads:
    t.join()

print(cnt)
```

理论上，程序的运行结果应该是输出 500000。但是实际运行时，cnt 最终的值是不确定的[2]。

因为 cnt 值的修改并不是一个原子操作，在一个线程修改 cnt 值的过程中，这个线程可能会中断，切换到另一个线程，而另一个线程也要修改 cnt 的值，两个线程对这块内存的访问就会发生冲突，使结果无法预料[3]。

2.3 互斥锁

为了解决对公有内存区域访问出现冲突的问题，保证共享内存操作时数据的完整性，引入了互斥锁（Mutex）的概念。[4] [5]

使用互斥锁[6]可以防止出现多个线程同时对一块数据进行读写的情况。

```
cnt = 0
mutex = threading.Lock()  # 创建锁

def foo():
    global cnt
    for i in range(100000):
        mutex.acquire()  # 获取锁
        try:
            cnt = cnt + 1
        finally:
            mutex.release() # 释放锁
```

注意互斥锁的使用一定是要先获取后释放的，如果不释放锁就会导致此段程序无法再次执行。

当一个线程尝试运行带有互斥锁的代码段时，如果没有其他线程获得这个锁，那么这个线程就可以获得锁，并进入该段代码。否则，这个线程会暂停运行，直到该段代码的锁被释放。

互斥锁可以保证一个线程执行一段代码时不会有其他线程同时执行同一段代码。

用 with 语句来写更加优雅：

```
cnt = 0
mutex = threading.Lock()

```

[2] 试试看

[3] http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014319282381876cd506abbc94eb5916192364506fa5

[4] 互斥是指某一资源同时只允许一个访问者对其进行访问。

[5] 同步是指在互斥的基础上通过其它机制实现访问者对资源的有序访问。

[6] https://zh.wikipedia.org/wiki/%E4%BA%92%E6%96%A5%E9%94%81

```
def foo():
    global cnt
    for i in range(100000):
      with mutex:
        cnt = cnt + 1
```

如果有两个线程都持有锁，并且互相希望访问对方加锁的代码块的话，就会出现死锁的情况。在这种情况下，如果没有外力干预，这两个线程就会永远暂停下去。需要避免出现这种情况。

如果希望一段代码可以同时被多个线程访问，但是又想限制访问的线程数量的话，可以使用信号量（Semaphore，Python: `threading.Semaphore`）来限制同时访问一段代码的总线程数。

2.4 线程安全队列

队列（Queue）是一种非常好的多线程同步机制。Python 提供的 Queue[7] 是线程安全的队列，也就是说在使用 Python Queue 的时候不用考虑互斥锁的问题。

```
import queue
import random
import threading
import time

q = queue.Queue()


def foo():
    time.sleep(random.random())
    q.put('Hello! This is %s' % threading.current_thread().name)  # 向队列里放东西

for i in range(10):
    t = threading.Thread(target=foo)
    t.start()

for i in range(10):
    s = q.get()  # 从队列中取东西
    print(s)
```

如果在队列为空的时候尝试调用 `get()` 的话，当前线程会暂停运行，一直等到队列非空 get 到东西为止。

queue 模块也提供了优先队列（PriorityQueue）和 LIFO 队列 (LifoQueue)，需要时可以调用。

2.5 多核处理器与 Python GIL

再回顾一下之前提到的内容：线程是操作系统进行运算调度的基本单位，一个线程是一个线性的控制流。

[7] https://docs.python.org/3/library/queue.html

我们使用多线程的目的很多时候是为了让可以同时进行的多个工作真正实现同时进行从而提高程序运行效率。

对于一个计算密集的多线程任务，在多核处理器上，我们可以让每个线程在一个独立的核心上运行，那么在 N 核处理器上任务就可以用单线程执行的 1/N 的时间完成。

是这样吗?

对绝大多数常见的语言而言，这个认识是正确的。然而，对 Python 来说，在多核处理器上运行一个多线程程序却往往不能达到预期的性能提升。

其原因是在 Python 的 CPython 实现中，引入了全局锁 GIL。

"In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces."

GIL 的存在使得 Python 的代码在同一时刻只能被同一个进程内的一个线程执行，在性能上强行将程序变成单线程，限制了 Python 多线程的效率。实际上在运行中，一个 Python 程序的所有线程加起来也只能使用总计一个 CPU 核的计算资源。[8]

2.6 C++ 的多线程

C++ 11 在 STL 中也提供了多线程编程接口[9]。使用头文件 `#include<thread>` ，然后创建线程:

```
void foo(string str) {
    cout << str << endl;
}

int main() {
    std::thread t(foo, "Hello, world!");  // 创建线程对象的同时即开始运行
    // Do something else
    t.join();  // 等待线程结束
}
```

与处处受 GIL 限制的 Python 不同，C++ 的多线程是真正的多线程。在多核 CPU 上运行的多线程 C++ 程序可以充分利用计算机资源[10]。

使用互斥锁需要 `#include<mutex>`

```
std::mutex mtx;
mtx.lock();
// Do something
mtx.unlock();
```

但是，C++ 的 queue 并不是线程安全的。如果想用 queue 来进行线程间同步的话需要手动对改变 queue 内容的操作加锁。

[8] https://cenalulu.github.io/python/gil-in-python/

[9] http://www.cplusplus.com/reference/thread/thread/

[10] 写一个多线程的 while(true); 循环就可以将系统资源耗尽..

作业-多线程爬虫

之前我写过一个获取豆瓣电影相关推荐的小程序.. 请把附件代码 dbrt.py 改为多线程实现。

提示：可以开若干个线程，让每个线程自行去队列中领取任务并执行。Python 的 list 不是线程安全的。

注意：访问页面的频率不要过高，连续访问的时间也不要过长。

如果提示需要安装 BeautifulSoup 的话，需要执行下面的命令之一：

```
apt-get install python3-bs4  # Ubuntu/Debian
pip3 install beautifulsoup4  # *nix
easy_install3 beautifulsoup4  # 如果用 easy_install 而不是 pip
python -m pip install beautifulsoup4  # Windows 可能是这样？
```

如果没有安装 pip 或 easy_install[11]，可以尝试在`https://www.crummy.com/software/BeautifulSoup/bs4/download/4.0/`下载最新版本的 BeautifulSoup，解压后手动在目录下执行

```
python ./setup.py install
```

BeautifulSoup 文档[12] `https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html`

下周六（7 月 9 日）晚 19:00 前将代码[13]发到邮箱 `mxj14@mails.tsinghua.edu.cn`。

[11] 强烈建议安装！

[12] 当然也可以彻底无视它...

[13] 如果实在不熟悉 Python 可以用伪代码描述思路。