多线程

CPU 发展史

（Intel）

快速回顾Intel的发展史: https://zhuanlan.zhihu.com/p/373629126

- 2002-2004年，超线程P4处理器 2002年11月14日，英特尔在全新英特尔奔腾4处理器3.06GHz上推出其创新超线程(HT)技术。超线程(HT)技术支持全新级别的高性能台式机，同时快速运行多个计算应用，或为采用多线程的单独软件程序提供更多性能。超线程(HT)技术可将电脑性能提升达25%。除了为台式机用户引入超线程(HT)技术外，英特尔在推出英特尔奔腾4处理器3.06GHZ时达到了一个电脑里程碑。

- 2005-2006年，多线程，2005年4月，英特尔的第一款双核处理器平台包括采用英特尔955X高速芯片组、主频为3.2GHz的英特尔奔腾处理器至尊版840，此款产品的问世标志着一个新时代来临了。双核和多核处理器设计用于在一枚处理器中集成两个或多个完整执行内核，以支持同时管理多项活动。英特尔超线程(HT)技术能够使一个执行内核发挥两枚逻辑处理器的作用，因此与该技术结合使用时，英特尔奔腾处理器至尊版840能够充分利用以前可能被闲置的资源，同时处理四个软件线程。

CPU 轮询

CPU在多个任务之间切换执行， 由于切换的速度非常快，看起来就像同时在进行。

- 一个CPU核心同一时刻只能执行一个任务。
- 多核CPU可以理解为多个单核CPU的集合。

并发与并行

对于一个支持多进程的系统，CPU 会从一个进程快速切换至另一个进程，其间每个进程各运行几十或几百个毫秒。

虽然单核的 CPU在某一个瞬间，只能运行一个进程。但在1秒钟期间，它可能会运行多个进程，这样就产生并行的错觉，实际上这是并发。

多任务编程：

- 多任务编程： 在一个程序中编写多个任务，在程序运行中让多个任务同时执行。
- 实现方法： 多进程编程，多线程编程。
- 好处： 充分利用计算机资源

进程

进程（Process）是指在系统中能独立运行并作为资源分配的基本单位，它是由一组机器指令、数据和堆栈等组成的，是一个能独立运行的实体。当你运行一个程序，你就启动了一个进程。凡是用于完成操作系统的各种功能的进程就是系统进程，而所有由你启动的进程都是用户进程。进程一般有三个状态：就绪状态、执行状态和等待状态.

进程的特征：

- 动态性：进程的实质是程序的一次执行过程，进程是动态产生，动态消亡的。
- 并发性：任何进程都可以同其他进程一起并发执行。
- 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。
- 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。

任何一个进程都有3个基础部分：

- 一个可以运行的程序。
- 这个程序运行所需要的数据。
- 程序执行的状态。

线程

线程（Thread），线程是进程中的一个实体，作为系统调度和分派的基本单位。进程是由一个或多个线程构成的。而线程是进程中的实际运行单位，是独立运行于进程之中的子任务，是操作系统进行运算调度的最小单位。可理解为线程是进程中的一个最小运行单元。

线程的性质：

- 线程是进程内的一个相对独立的可执行的单元。若把进程称为任务的话，那么线程则是应用中的一个子任务的执行。
- 由于线程是被调度的基本单元，而进程不是调度单元。所以，每个进程在创建时，至少需要同时为该进程创建一个线程。即进程中至少要有一个或一个以上的线程，否则该进程无法被调度执行。
- 进程是被分给并拥有资源的基本单元。同一进程内的多个线程共享该进程的资源，但线程并不拥有资源，只是使用他们。
- 线程是操作系统中基本调度单元，因此线程中应包含有调度所需要的必要信息，且在生命周期中有状态的变化。
- 由于共享资源(包括数据和文件)，所以线程间需要通信和同步机制，且需要时线程可以创建其他线程，但线程间不存在父子关系。

多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程的相关概念：

主线程：当一个程序启动时，就有一个进程被操作系统（**OS**）创建，与此同时一个线程也立刻运行，该线程通常叫做程序的主线程（**Main Thread**）。因为它是程序开始时就执行的，如果你需要再创建线程，那么创建的线程就是这个主线程的子线程。

线程（**Thread**）也叫轻量级进程，是操作系统能够进行运算调度的最小单位，它被包涵在进程之中，是进程中的实际运作单位。线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。

子线程：使用**threading**、**ThreadPoolExecutor**创建的线性均为子线程。

主线程的重要性体现在两方面：**1.**是产生其他子线程的线程；**2.**通常它必须最后完成执行，比如执行各种关闭动作。

Python提供了**threading**模块来实现多线程:**threading.Thread**可以创建线程；**setDaemon(True)**为守护主线程，默认为**False**；**join()**为守护子线程。

线程的生命周期大体可分为**5**种状态:

1. 新建**(NEW)**：新创建了一个线程对象。
2. 可运行**(RUNNABLE)**：线程对象创建后，其他线程**(**比如**main**线程）调用了该对象的**start()**方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取**cpu**的使用权 。
3. 运行**(RUNNING)**：可运行状态**(runnable)**的线程获得了**cpu**时间片（**timeslice**），执行程序代码。
4. 阻塞**(BLOCKED)**：阻塞状态是指线程因为某种原因放弃了**cpu**使用权，也即让出了**cpu timeslice**，暂时停止运行。直到线程进入可运行**(runnable)**状态，才有机会再次获得**cpu timeslice** 转到运行**(running)**状态。
5. 死亡（**dead**）线程**run()**、**main()**方法执行结束，或者因为异常退出了**run()**方法，则该线程结束生命周期

 **image source:** https://xinzhi.wenda.so.com/a/1524033624614917

阻塞又分为三种：

1. 等待阻塞：运行状态的线程执行**wait()**方法，线程会被放在等待队列中，使本线程进入阻塞状态。
2. 同步阻塞：线程在获得**synchronized**同步锁失败，线程被放入锁池中，线程进入同步阻塞。
3. 其他阻塞：调用线程的**sleep()**或者**join()**后，线程会进入阻塞状态，当**sleep**超时或者**join**终止或超时，线程重新转入就绪状态。

原文链接：https://blog.csdn.net/Elephant_King/article/details/122559577

 **image source:** https://blog.csdn.net/ThinkWon/article/details/102021274

守护线程

守护线程是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾收集器线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

守护线程作用是为其他线程提供便利服务，守护线程最典型的应用就是 **GC (垃圾收集器)**。

如果你有兴趣了解**Python**垃圾收集: **https://blog.csdn.net/allway2/article/details/118065742**

Python 中的多线程

In [2]:
```
from time import sleep
import threading

def music(music_name):
    for i in range(2):
        print('Listening {}'.format(music_name))
        sleep(1)
    print('music over')

def game(game_name):
    for i in range(2):
        print('Playing {}'.format(game_name))
        sleep(3)
    print('game over')
```

In [4]:
```
threads = []
t1 = threading.Thread(target=music,args=('Beat it',))
threads.append(t1)
t2 = threading.Thread(target=game,args=('Need for Speed',))
threads.append(t2)

if __name__ == '__main__':
    for t in threads:
        #t.setDaemon(True)
        #t.daemon=True
        t.start()

        t.join()

    print('Main Thread is over')
```

```
Listening Beat it
Listening Beat it
music over
Playing Need for Speed
Playing Need for Speed
game over
Main Thread is over
```

结果（每次可能不一样）

Listening Beat it

Playing Need for Speed

Main Thread is over

music over

Listening Beat it

music over

game over

Playing Need for Speed

game over

Thread.join()

如果**thread**是某个子线程，则调用**thread.join()**的作用是确保**thread**子线程执行完毕后才能执行下一个线程。

In [ ]:
```
if __name__ == '__main__':
    t = threading.Thread(target=music, args=("Poker Face",))
    t.start()
    #t.join()
    print("main thread ended")
```

In [ ]:
```
if __name__ == '__main__':

    threads = []

    for i in range(5):
        # target指定线程要执行的代码，args指定该代码的参数
        t = threading.Thread(target=music, args=("Poker Face" + str(i),))
        threads.append(t)


    for t in threads:
        t.start()
        t.join()

    print("main thread ended")
```

In [5]:
```
if __name__ == '__main__':

    threads = []

    for i in range(3):
        # target指定线程要执行的代码，args指定该代码的参数
        t = threading.Thread(target=music, args=("Poker Face" + str(i),))
        threads.append(t)


    for t in threads:
        t.start()

    for t in threads:
        t.join()
        #t.join(0.2)

    print("main thread ended")
```

```
Listening Poker Face0
Listening Poker Face1
Listening Poker Face2
Listening Poker Face1
Listening Poker Face0
Listening Poker Face2
music over
music over
music over
main thread ended
```

守护线程

我们看下面这个例子，这里使用**setDaemon(True)**把所有的子线程都变成了主线程的守护线程，因此当主进程结束后，子线程也会随之结束。所以当主线程结束后，整个程序就退出了。

In [6]:
```
import time
import threading


def fun():
    print("start fun")
    time.sleep(5)
    print("end fun")

def main():
    print("main thread")
    t1 = threading.Thread(target=fun,args=())
    t1.daemon = True
    t1.start()
    time.sleep(1)
    print("main thread end")

if __name__ == '__main__':
    main()
```

```
main thread
start fun
main thread end
end fun
```

自定义线程

继承**threading.Thread**来自定义线程类，其本质是重构**Thread**类中的**run**方法

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

class MyThread(threading.Thread):
    def __init__(self, n):
        super(MyThread, self).__init__()  # 重构run函数必须要写
        self.n = n

    def run(self):
        print("task", self.n)
        time.sleep(1)
        print('2s')
        time.sleep(1)
        print('1s')
        time.sleep(1)
        print('0s')
        time.sleep(1)

if __name__ == "__main__":
    t1 = MyThread("t1")
    t2 = MyThread("t2")
    t1.start()
    t2.start()
```

```
task t1
task t2
2s2s

1s
```

```
1s
0s
0s
```

线程池

因为新建线程系统需要分配资源、终止线程系统需要回收资源，所以如果可以重用线程，则可以减去新建/终止的开销以提升性能。同时，使用线程池的语法比自己新建线程执行线程更加简洁。

Python为我们提供了**ThreadPoolExecutor**来实现线程池，此线程池默认子线程守护。它的适应场景为突发性大量请求或需要大量线程完成任务，但实际任务处理时间较短。

In [ ]:
```python
from concurrent.futures import ThreadPoolExecutor
import threading
import time

# 定义一个准备作为线程任务的函数
def action(max):
    my_sum = 0
    for i in range(max):
        print(threading.current_thread().name + '  ' + str(i))
        my_sum += i
    return my_sum

# 创建一个包含2条线程的线程池
pool = ThreadPoolExecutor(max_workers=2)
# 向线程池提交一个task, 50会作为action()函数的参数
future1 = pool.submit(action, 50)
# 向线程池再提交一个task, 100会作为action()函数的参数
future2 = pool.submit(action, 100)
# 判断future1代表的任务是否结束
print('f1 ', future1.done())
time.sleep(3)
# 判断future2代表的任务是否结束
print('f2', future2.done())
# 查看future1代表的任务返回的结果
print(future1.result())
# 查看future2代表的任务返回的结果
print(future2.result())
# 关闭线程池
pool.shutdown()
```

你银行账户有**500**块，花呗欠了**400**，借呗欠了**350**，现在都来要债了。。。。。

1. 花呗线程问银行还有多少钱，银行回答**500**
2. 花呗线程收走**400**
3. 花呗线程结束

4. 借呗线程问银行还有多少钱，银行回答**100**

5. 借呗线程记录余额不足
6. 借呗线程结束

如果。。。

1. 花呗线程问银行还有多少钱，银行回答**500**
2. 花呗线程等待

3. 借呗线程问银行还有多少钱，银行回答**500**

4. 借呗线程收走**350**
5. 借呗线程结束
6. 花呗线程运行
7. 花呗线程收走**400**
8. 花呗线程结束

结果。。。。 余额变成**-250**

线程互斥

我们把一个时间段内只允许一个线程使用的资源称为临界资源，对临界资源的访问，必须互斥的进行。互斥，也称间接制约关系。线程互斥指当一个线程访问某临界资源时，另一个想要访问该临界资源的线程必须等待。当前访问临界资源的线程访问结束，释放该资源之后，另一个线程才能去访问临界资源。锁的功能就是实现线程互斥。

我把线程互斥比作机场安检的过程，因为安检一次只能一个人，所以只允许一个人进行检查。当第一个人被检查时，这时如果第二个人也想被检查，那就必须等第一个人检查完，将锁解开后才能进行，在这期间第二个人就只能在等待。这个过程与代码中使用锁的原理如出一辙，这里的安检过程就是临界资源。

Python 的 **threading** 模块引入了锁。**threading** 模块提供了 **Lock** 类，它有如下方法加锁和释放锁：

acquire()：对 **Lock**加锁，其中**timeout**参数指定加锁多少秒 **release()**：释放锁

```
funcA() {
    lock(锁)    // 要保护的数据的逻辑部分。
    …
    unlock(锁)
}
```

锁

1. 花呗线程获得银行操做锁
2. 花呗线程问银行还有多少钱，银行回答**500**
3. 花呗线程等待
4. 借呗线程问银行还有多少钱，银行回答我被锁了
5. 借呗线程等待
6. 花呗线程运行
7. 花呗线程收走**400**
8. 花呗线程释放银行操做锁
9. 花呗线程结束
10. 借呗线程问银行还有多少钱，银行回答**100**

11. 借呗线程记录余额不足
12. 借呗线程结束

In [ ]:
```
import threading

class Account:
    def __init__(self, card_id, balance):
        # 封装账户ID、账户余额的两个变量
        self.card_id= card_id
        self.balance = balance

def withdraw(account, money):
    # 进行加锁
    lock.acquire()
    # 账户余额大于取钱数目
    if account.balance >= money:
        # 吐出钞票
        print(threading.current_thread().name + "取钱成功！吐出钞票:" + str(money),end=' ')
        # 修改余额
        account.balance -= money
        print("\t余额为: " + str(account.balance))
    else:
        print(threading.current_thread().name + "取钱失败！余额不足")
    # 进行解锁
    lock.release()
# 创建一个账户，银行卡id为8888，存款1000元
acct = Account("8888" , 1000)

# 模拟两个对同一个账户取钱
# 在主线程中创建一把锁
lock = threading.Lock()
threading.Thread(name='窗口A', target=withdraw , args=(acct , 800)).start()
threading.Thread(name='窗口B', target=withdraw , args=(acct , 800)).start()
```

死锁：

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

死锁的规范定义：集合中的每一个进程都在等待只能由本集合中的其他进程才能引发的事件，那么该组进程是死锁的。

In [ ]:
```
import threading

def main():
    lock.acquire()
    print('第一道锁')
    lock.acquire()
    print('第二道锁')
    lock.release()
    lock.release()

if __name__ == '__main__':
    lock = threading.Lock()
    main()
```

我们只是简单的介绍了多线程的概念， 实际大型系统中多线程是非常复杂的。

其他： **https://zhuanlan.zhihu.com/p/158965214 https://zhuanlan.zhihu.com/p/258049386**

超线程： **https://www.intel.cn/content/www/cn/zh/gaming/resources/hyper-threading.html**

https://www.toptal.com/python/beginners-guide-to-concurrency-and-parallelism-in-python

Async/Await (Python 3.5+ only) One of the most requested items in the comments on the original article was for an example using Python 3's asyncio module. Compared to the other examples, there is some new Python syntax that may be new to most people and also some new concepts. An unfortunate additional layer of complexity is caused by Python's built-in urllib module not being asynchronous. We will need to use an async HTTP library to get the full benefits of asyncio. For this, we'll use aiohttp.

中文例子 **https://www.cnblogs.com/heniu/p/12740400.html**

https://zhuanlan.zhihu.com/p/44661314

In [ ]:
```
# The end
```