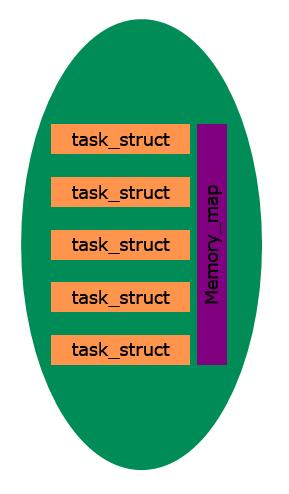
# 课程介绍

* 1. 线程概述;（了解）
* 2. python多线程编程;（掌握）
* 3. 线程的互斥和同步;（掌握）

1. 线程概述
   1. 什么是线程

【维基百科】：

线程（thread）是现代操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

由于进程的地址空间是私有的，因此在进程间上下文切换时，系统开销比较大;

为了提高系统的性能，许多操作系统规范里引入了轻量级进程的概念，也被称为线程;

通常来说，线程就是指共享相同地址空间的多个任务;

他带来的好处是：

* 大大提高了任务的切换效率；
* 避免了额外的TLB&Cache的刷新；
  1. 线程 VS 进程

进程和线程的区别：

* 地址空间：进程具备独立的地址空间，相互不影响，而线程是共享一个进程的地址空间。
* 通信：进程间通信IPC，往往需要操作系统提供相关机制来完成，而线程间可以直接读写进程数据段（如全局变量）来进行通信，线程间通信需要考虑临界资源的互斥访问。
* 调度和切换：线程上下文切换比进程上下文切换要快得多。

另外一种说法：

* 一个程序至少有一个进程,一个进程至少有一个线程.
* 线程的划分尺度小于进程(资源比进程少)，使得多线程程序的并发性高。
* 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率
* 线线程不能够独立执行，必须依存在进程中
  1. 总结和注意事项

线程仍然是被调度任务，这点和进程是一致的，最大的区别就是资源访问的方式。

线程的调度状态和进程的状态是一致的，在linux下，把每个线程还是归为进程ID进行管理。

1. python多线程编程

python的thread模块是比较底层的模块，python的threading模块是对thread做了一些包装的，可以更加方便的被使用，在实际开发中，我们都使用threading模块。

* 1. python线程的产生
* \_thread模块： 提供了底层原始C接口的封装。(py2的叫做thread)
* threading模块：对thread模块进行再次封装，使用起来更方便
  + 1. 原始线程模块的使用（了解）

在Python3中，内置了\_thread模块对底层线程做了封装，简单了解下线程的特点：

* + 主进程结束后，线程会自动消失；
  + 线程依赖进程提供的内存映射资源；
* 在实际开发中，建议都使用threading模块进行相关开发。
* threading模块里有一个Thread类，提供了管理线程产生的功能。
* Thread类的使用方式和Process类是一致的，所以可以对比记忆。
  + 1. 使用Thread类实例化方法

import threading

import time

def worker(tid):

print(f"... {tid}工作开始 线程名称{threading.current\_thread().name}...")

time.sleep(1)

print(f"=== {tid}工作结束 ===")

threads = []

for i in range(5):

t = threading.Thread(target=worker, args=(i,))

threads.append(t)

t.start()

* + 1. 使用Thread继承方法

在子类中，要先调用线程Thread初始化方法，再进行其他属性绑定。

import threading

import time

class TaskThread(threading.Thread):

def \_\_init\_\_(self, flags, name):

super(TaskThread, self).\_\_init\_\_() **# 先做初始化，后绑定**

self.flags = flags

self.name = name

def run(self):

for \_ in range(3):

print("in threading, {}")

time.sleep(1)

def main\_process():

print("主进程...")

t1 = TaskThread(2, 'Sam')

t1.start()

print("创建完成")

if \_\_name\_\_ == '\_\_main\_\_':

main\_process()

* + 1. 线程的名字和ID

threading提供了类方法currentThread()可以返回运行中的线程对象。

每一个线程对象，有name属性来标识线程名字，id属性标识线程的身份标识。

* + 1. 线程间通信

由于线程是共享了进程的内存资源，所以只要在不同的线程中，使用全局名字空间的key值，那么就相当于访问的是同一块的内存。

* + 1. 总结

python中产生线程的方法主要分为实例化和继承法。

实例化的方法：核心是通过传入target和args的方式。

继承法：重写run方法，在\_\_init\_\_中绑定线程代码里需要的参数。

使用threading.Thread后，默认主进程也是线程，主线程结束，不影响其他子线程。

1. 线程的同步和互斥
   1. 什么是临界资源

* 临界资源是指在多任务编程时，每个任务都能访问到的资源，就叫做临界资源。
* 由于多任务调度的随机性，可能出现一个任务在操作，而另外一个任务也在操作，导致每个任务最终访问的效果出现混乱。
* 多线程编程时，共享整个虚拟内存，所以在多线程编程时，往往需要考虑临界资源的互斥和同步访问。主要考虑的内容是：全局变量和代码资源。
* 多进程编程来说，进程的用户空间资源是独立的，临界资源只是内核空间。相对多线程编程来说，不需要考虑那么多临界访问的问题
  + 1. 全局变量访问临界区

from threading import Thread

glb\_var = 0

# 线程1，写数据

def task1\_th():

global glb\_var

for x in range(1000):

glb\_var += 1

print("线程1看到的glb\_var:", glb\_var)

# 线程2，写数据

def task2\_th():

global glb\_var

for x in range(1000):

glb\_var += 1

print("线程2看到的glb\_var:", glb\_var)

# 主线程

def main\_thread():

t1 = Thread(target=task1\_th)

t2 = Thread(target=task2\_th)

t1.start()

t2.start()

t1.join()

t2.join()

print("主线程里看到的glb\_var:", glb\_var)

main\_thread()

* + 1. 代码区访问临界资源

import threading

value1 = 0

value2 = 0

def writer\_worker():

global value1

global value2

count = 0

while True:

count += 1

value1 = count

value2 = count

def check\_worker():

while True:

if value2 != value1:

print(f"+++++v1:{value1}++v2:{value2}+++")

if \_\_name\_\_ == '\_\_main\_\_':

t1 = threading.Thread(target=writer\_worker, name="写线程")

t2 = threading.Thread(target=check\_worker, name="读线程")

t1.start()

t2.start()

* + 1. 如何解决访问临界资源
* 使用python内置的原子结构设计临界资源。
* list、dict的操作方法
* queue.Queue的队列结构
* 使用Lock（锁）、Semaphore（信号量）。
* Lock对象（最常用的）
* Event对象（事件对象）
* Timer对象（定时器对象）
  1. 线程的Lock对象
* 锁机制是操作系统底层提供的一种互斥访问的机制。
* 当一个任务获取到未锁定状态的锁时，任务继续运行；
* 当一个任务获取到以锁定状态的锁时，任务将被放入等待队列，直到这把锁被释放时被唤醒。

threading提供了一个Lock类，来实现了这种锁机制。

* + 1. Lock使用方法

1、实例化一个锁对象。

2、在使用临界资源前加锁acquire()操作

3、使用完成后，释放锁release()操作

修改代码为：

import threading

value1 = 0

value2 = 0

v\_lock = None

def writer\_worker():

global value1

global value2

count = 0

while True:

count += 1

v\_lock.acquire()

value1 = count

value2 = count

v\_lock.release()

def check\_worker():

while True:

v\_lock.acquire()

if value2 != value1:

print(f"+++++v1:{value1}++v2:{value2}+++")

v\_lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

t1 = threading.Thread(target=writer\_worker, name="写线程")

t2 = threading.Thread(target=check\_worker, name="读线程")

v\_lock = threading.Lock()

t1.start()

t2.start()

* + 1. Lock的Context Managers

每次lock后都是加锁，解锁，有时候可能疏忽，会导致死锁的发生，幸好python提供了with关键字。后面的代码都可以改为：

with lock:

临界资源

* 1. 线程的同步

互斥解决的是避免同时访问资源的，而同步是指每个任务执行的先后关系。

举例：

一个线程获取用户输入字符串，一个线程统计用户字符串的数字个数，要求只有在用户输入后，才开始统计。

import threading

user\_words = ""

def user\_get(lock1, lock2):

global user\_words

while True:

lock1.acquire()

user\_words = input("请输入聊天内容：")

lock2.release()

if user\_words.lower() == 'quit':

break

print("用户退出...")

def statistics\_data(lock1, lock2):

count = 0

while True:

lock2.acquire()

for x in user\_words:

if x.isdigit():

count += 1

print(f"用户输入了{count}个数字")

count = 0

lock1.release()

def main\_process():

lock1 = threading.Lock()

lock2 = threading.Lock()

t1 = threading.Thread(target=user\_get, args=(lock1, lock2), name="输入线程")

t2 = threading.Thread(target=statistics\_data, args=(lock1, lock2), name="统计线程")

# t2.setDaemon(True)

lock2.acquire()

t2.start()

t1.start()

t1.join()

print("游戏结束")

if \_\_name\_\_ == '\_\_main\_\_':

main\_process()

1. 课程总结
   1. 重点
      * 1. 进程和线程的区别
        2. 互斥访问锁机制的工作方式
        3. threading产生线程的方式
        4. 如何使用互斥锁
   2. 难点
      * 1. 互斥和同步的区别
        2. 如何实现一个同步访问的多任务系统
2. 课后练习
   * + 1. 复习实例化产生线程和类继承方式产生线程的方法和重写代码。
       2. 复习代码区临界访问的2个例子代码。
       3. 用类继承方式重写代码临界区访问的例子。
       4. 模拟一个输入单词检错的demo，用户输入语句后，判断输入单词是否有误。（扩展）
3. 面试题
   * + 1. 描述线程和进程的区别
       2. 什么是线程安全，什么是线程不安全
       3. 什么是临界资源，如何解决临界资源的同步访问？
4. 扩展知识或课外阅读推荐（可选）
   1. 守护线程

在python中，threading会管理所有子线程，只有全部退出后，解释器才会退出。

但如果我们有一种需求，这个线程永远都不退出，而是依赖其他子线程结束后才结束，这个时候需要使用线程守护属性。比如一个统计用户输入内容，当用户放弃输入后，应该结束整个进程。

线程默认是非守护属性的，要设置为守护属性，调用setDaemon()即可。

import threading

import time

def daemon():

print(f"{threading.current\_thread().getName()} 守护线程运行...")

time.sleep(2)

print(f"{threading.current\_thread().getName()} 守护线程退出...")

d = threading.Thread(name='daemon', target=daemon)

d.setDaemon(True)

def non\_daemon():

print(f"{threading.current\_thread().getName()} 非守护线程运行...")

time.sleep(1)

print(f"{threading.current\_thread().getName()} 非守护线程退出...")

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

如果非要等到非守护线程结束才退出，就可以使用join方法进行等待。

* 1. 线程同步补充
     1. Event事件

这是一种最简单的任务同步机制，一个任务等待一个信号，另外一个任务等待满足后，发送一个事件，等待任务开始运行。

set() : 设置信号

clear(): 清除信号

wait(): 等待信号的发生

见参考代码：event同步统计信息.py

* + 1. 定时器

threading提供了Timer类，作为定时器。

Timer(interval, function, args=None, kwargs=None)

interval:定时的时间，秒为单位

function: 定时后执行的任务

* 1. GIL