# 课程介绍

* 1、Linux系统编程模型;（理解）
* 2、Linux进程介绍;（理解）
* 3、python下的进程编程;（掌握）
* 4、进程池及进程间通信；（了解）

1. Linux系统编程模型

硬件层

内核层（C）

系统调用层（C）

python原始接口层

python高级接口层

框架

py程序

**内核空间**

**用 户 空 间**

文件描述符

py基本对象

py高级对象

1. 进程概述
   1. 什么是进程

* 进程（process），是计算机中已运行程序的实体。进程曾经是分时系统的基本运作单位。
* 当代多数操作系统，包括Linux2.6以上内核，是面向线程设计的，进程本身不是基本运行单位，而是线程的容器。
  + 1. 程序和进程的区别
* 程序只是一组指令的有序集合，它本身没有任何运行的含义，它只是一个静态的实体。
* 进程是一个动态的实体，它有自己的生命周期。它因创建而产生，因调度而运行，因等待资源或事件而被处于等待状态，因完成任务而被撤消。反映了一个程序在一定的数据集上运行的全部动态过程。
  1. 进程调度

我们平时使用windows时，既可以边聊QQ，边上网页，边听音乐，那么这是怎么做到的那？那么这里就是靠操作系统的进程调度模块来做到的。

时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法。

时间片由操作系统内核的调度程序分配给每个进程。首先，内核会给每个进程分配相等的初始时间片，然后每个进程轮番地执行相应的时间，当所有进程都处于时间片耗尽的状态时，内核会重新为每个进程计算并分配时间片，如此往复。

* 1. 总结

每一个进程都具备2个重要特性：

* 任务的可调度性： task\_struct
* 资源的独立性： memory\_map

task\_struct

memory\_map

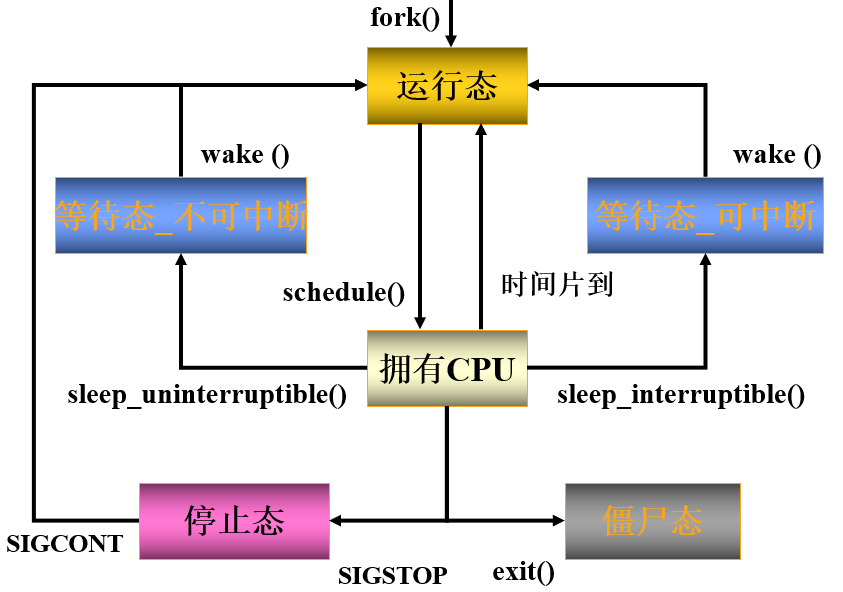
task\_struct

memory\_map

1. Linux进程概述
   1. Linux进程管理命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| ps -aux | 显示进程信息，包括无终端的（x）和针对用户（u）的进程。 |
| ps -ef | -f：按进程信息格式化输出 -e:列出所有。 |
| pstree | 列出进程树状结构。 |
| top | 监视系统中不同的进程所使用的资源。 |
| kill | 向进程发送信号。 |

* 1. Linux进程状态



◇ R ： 运行中或可被运行的状态

◇ S ： 可中断的等待态

◇ D ： 不可中断的等待态

◇ Z ： 僵尸态

◇ T ： 停止态

* 1. Linux下的进程生命周期
     1. fork系统调用

python的os模块封装了常见的系统调用，其中就包括fork，可以在Linux下轻松创建子进程。

fork函数一旦调用成功，系统会产生一个原来进程的克隆，原进程升级为父进程。

import os

os.fork()

print("child process...")

思考为什么会打印2次child process…

* 程序执行到os.fork()时，操作系统会创建一个新的进程（子进程），然后复制父进程的所有信息到子进程中。
* 父进程和子进程都会从fork()函数中得到一个返回值，在子进程中这个值一定是0，而父进程中是子进程的id号。
  + 1. getpid和getppid

这2个函数见名知意，一个是获取当前进程的ID号，一个是获取父进程ID号。

import os

rpid = os.fork()

if rpid<0:

print("fork调用失败。")

elif rpid == 0:

print("我是子进程（%s），我的父进程是（%s）"%(os.getpid(),os.getppid()))

x+=1

else:

print("我是父进程（%s），我的子进程是（%s）"%(os.getpid(),rpid))

print("父子进程都可以执行这里的代码")

* + 1. 孤儿进程

子进程还未运行完成，父进程就结束运行退出，留下的子进程就是孤儿进程。父进程死掉后，孤儿进程会被别的进程收养，通常是init进程（pid为1）。

后面看到的守护进程，实际就是孤儿进程。

* + 1. 僵尸进程

子进程运行完成，但是父进程迟迟没有进行回收，此时子进程实际上并没有退出，其仍然占用着系统资源，这样的子进程称为僵尸进程。

因为僵尸进程的资源一直未被回收，造成了系统资源的浪费，过多的僵尸进程将造成系统性能下降，所以应避免出现僵尸进程。

僵尸进程的状态是Z。

* + 1. 进程资源的回收

Linux系统提供了一个叫做wait的系统调用，来回收子进程的资源。wait不能指定等待哪个进程回收资源，而waitid是可以的。

wait函数返回了一个元组，第一个代表是回收了哪个进程ID号，第二个是子进程的退出状态。

* + 1. 守护进程

守护进程(daemon)是一类在后台运行的特殊进程，用于执行特定的系统任务。

* 守护进程在系统引导的时候启动，并且一直运行直到系统关闭。
* 只在需要的时候才启动，完成任务后就自动结束。

Linux下的服务程序，大部分都是守护进程，比如：

sshd、nginx、apache等

1. python多进程编程

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于Windows没有fork调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块。

* 1. Process类

multiprocessing模块提供了一个Process类来代表一个进程对象。

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Process的函数申明：

**Process([group [, target [, name [, args [, kwargs]]]]])**

* + target：表示这个进程实例所调用对象；
  + args：表示调用对象的位置参数元组；
  + kwargs：表示调用对象的关键字参数字典；
  + name：为当前进程实例的别名；
  + group：大多数情况下用不到；

**Process类常用方法：**

is\_alive()：判断进程实例是否还在执行；

join([timeout])：是否等待进程实例执行结束，或等待多少秒；

start()：启动进程实例（创建子进程）；

run()：如果没有给定target参数，对这个对象调用start()方法时，就将执行对象中的run()方法；

terminate()：不管任务是否完成，立即终止；

**Process类常用属性：**

name：当前进程实例别名，默认为Process-N，N为从1开始递增的整数；

pid：当前进程实例的PID值；

* 1. 实例化使用Process

通过实例化Process类，注意调用start方法

from multiprocessing import Process

import os

from time import sleep

# 子进程要执行的代码

def run\_proc(name, age, \*\*kwargs):

for i in range(10):

print('子进程运行中，name= %s,age=%d ,pid=%d...' % (name, age,os.getpid()))

print(kwargs)

sleep(0.5)

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

print('父进程 %d.' % os.getpid())

p = Process(target=run\_proc, args=('test',18), kwargs={"m":20})

print('子进程将要执行')

p.start()

sleep(1)

p.terminate()

p.join()

print('子进程已结束')

* 1. 类继承方式使用Process

创建新的进程还能够使用类的方式，可以自定义一个类，继承Process类，每次实例化这个类的时候，就等同于实例化一个进程对象。

from multiprocessing import Process

import time

import os

#继承Process类

class Process\_Class(Process):

#因为Process类本身也有\_\_init\_\_方法，这个子类相当于重写了这个方法，

#但这样就会带来一个问题，我们并没有完全的初始化一个Process类，所以就不能使用从这个类继承的一些方法和属性，

#最好的方法就是将继承类本身传递给Process.\_\_init\_\_方法，完成这些初始化操作

def \_\_init\_\_(self,interval):

Process.\_\_init\_\_(self)

self.interval = interval

#重写了Process类的run()方法

def run(self):

print("子进程(%s) 开始执行，父进程为（%s）"%(os.getpid(),os.getppid()))

t\_start = time.time()

time.sleep(self.interval)

t\_stop = time.time()

print("(%s)执行结束，耗时%0.2f秒"%(os.getpid(),t\_stop-t\_start))

if \_\_name\_\_=="\_\_main\_\_":

t\_start = time.time()

print("当前程序进程(%s)"%os.getpid())

p1 = Process\_Class(2)

#对一个不包含target属性的Process类执行start()方法，就会运行这个类中的run()方法，所以这里会执行p1.run()

p1.start()

p1.join()

t\_stop = time.time()

print("(%s)执行结束，耗时%0.2f"%(os.getpid(),t\_stop-t\_start))

1. 进程池

当需要创建的子进程数量不多时，可以直接利用multiprocessing中的Process动态成生多个进程，但如果是上百甚至上千个目标，手动的去创建进程的工作量巨大，此时就可以用到multiprocessing模块提供的Pool方法。

初始化Pool时，可以指定一个最大进程数，当有新的请求提交到Pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到指定的最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行，请看下面的实例：

from multiprocessing import Pool

import os,time,random

def worker(msg):

t\_start = time.time()

print("%s开始执行,进程号为%d"%(msg,os.getpid()))

#random.random()随机生成0~1之间的浮点数

time.sleep(random.random()\*2)

t\_stop = time.time()

print(msg,"执行完毕，耗时%0.2f"%(t\_stop-t\_start))

po=Pool(3) #定义一个进程池，最大进程数3

for i in range(0,10):

#Pool.apply\_async(要调用的目标,(传递给目标的参数元祖,))

#每次循环将会用空闲出来的子进程去调用目标

po.apply\_async(worker,(i,))

print("----start----")

po.close() #关闭进程池，关闭后po不再接收新的请求

po.join() #等待po中所有子进程执行完成，必须放在close语句之后

print("-----end-----")

**multiprocessing.Pool常用函数解析：**

apply\_async(func[, args[, kwds]]) ：使用非阻塞方式调用func（并行执行，堵塞方式必须等待上一个进程退出才能执行下一个进程），args为传递给func的参数列表，kwds为传递给func的关键字参数列表；

apply(func[, args[, kwds]])：使用阻塞方式调用func

close()：关闭Pool，使其不再接受新的任务；

terminate()：不管任务是否完成，立即终止；

join()：主进程阻塞，等待子进程的退出， 必须在close或terminate之后使用；

1. 进程间通信
   1. Linux进程间通信的方式

进程间通信（IPC，Inter-Process Communication），指至少两个进程或线程间传送数据或信号的一些技术或方法。

Linux系统提供的底层进程间通信的方式有：

管道，分为有名管道和无名管道；

信号；

systemV下的共享内存，消息队列，信号量；

BSD下的socket；

其中socket是我们现在网络通信的基础。

* 1. Queue的使用

Queue实际是一个队列的数据模型在多进程模块的一个抽象。

初始化Queue()对象时（例如：q=Queue()），若括号中没有指定最大可接收的消息数量，或数量为负值，那么就代表可接受的消息数量没有上限（直到内存的尽头）

Queue.qsize()： 返回当前队列包含的消息数量；

Queue.empty()： 如果队列为空，返回True，反之False ；

Queue.full()： 如果队列满了，返回True,反之False；

Queue.get([block[, timeout]])：获取队列中的一条消息，然后将其从列队中移除，block默认值为True；

1）如果block使用默认值，且没有设置timeout（单位秒），消息列队如果为空，此时程序将被阻塞（停在读取状态），直到从消息列队读到消息为止，如果设置了timeout ，则会等待timeout秒，若还没读取到任何消息，则抛出"Queue.Empty"异常；

2）如果block值为False，消息列队如果为空，则会立刻抛出"Queue.Empty"异常；

Queue.get\_nowait()：相当Queue.get(False)；

Queue.put(item,[block[, timeout]])：将item消息写入队列，block默认值为True；

1）如果block使用默认值，且没有设置timeout（单位秒），消息列队如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息列队腾出空间为止，如果设置了timeout，则会等待timeout秒，若还没空间，则抛出"Queue.Full"异常；

2）如果block值为False，消息列队如果没有空间可写入，则会立刻抛出"Queue.Full"异常；

Queue.put\_nowait(item)：相当Queue.put(item, False)；

* 1. 进程池中的Queue（了解）

如果要使用Pool创建进程，就需要使用multiprocessing.Manager()中的Queue()，而不是multiprocessing.Queue()，否则会得到一条如下的错误信息：

RuntimeError: Queue objects should only be shared between processes through inheritance.

1. 课程总结
   1. 重点
      * 1. Linux的系统编程模型，了解文件描述符，系统调用这些概念。
        2. Linux进程状态的切换，进程在阻塞态、运行态的切换。
        3. Linux下的进程逻辑结构，孤儿进程、守护进程、僵尸进程的概念。
        4. python下的Process类的两种使用方法。
        5. multiprocess中的Queue的使用。
   2. 难点
      * 1. 多任务切换的思维。
        2. 进程间通信的方法。
        3. 进程池的概念。
2. 课后练习
3. 复习进程的概念，回答如下问题：

Linux下采用什么命令查看进程信息？S代表什么状态，R代表什么状态，Z代表什么状态？

什么是孤儿进程，什么是僵尸进程，如何避免僵尸进程，什么是守护进程？

1. 复习课堂代码：

写出创建子进程的实例化方法和类继承方法；

利用Queue实现2个进程的数据传递；

1. 面试题
   * + 1. 什么是进程？你怎么理解进程？
       2. 简述Linux进程的状态切换？
2. 扩展知识或课外阅读推荐（可选）
   1. 进程的资源
      1. 虚拟地址 vs 物理地址

每台电脑配置的内存都可能不同，那么程序员在写程序时，难道要为每台设备都重写一遍程序代码吗？显然是不现实的。现代操作系统，采用了一个叫做虚拟内存的概念，程序员只关心操作系统虚拟内存的地址，而虚拟地址和实际物理地址是由操作系统中一个叫做MMU的内存管理模块来进行映射。

对于32位的操作系统，他所能提供的虚拟地址空间，最大是4G。

1G

**用户态**

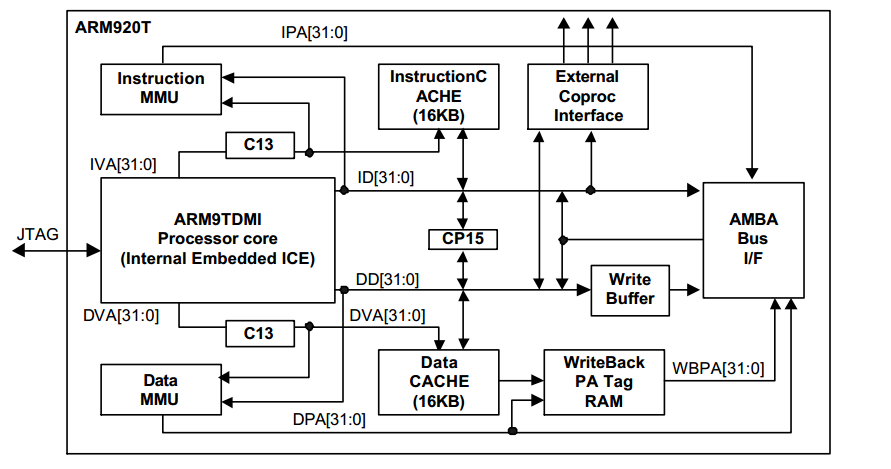
**内核态**

虚拟地址

物理内存

* + 1. CPU结构决定多任务

了解了虚拟地址和物理地址后，那么每个进程到底是怎么做到独立运行的那，下面看一副arm的920t内核的模型图来分析下：



* 1. Pool下的map方法

利用map方法，可以使用多核的方式，快速计算出需要的数据，并以列表形式给出。

from multiprocessing import Pool

import time

# 子进程

def run(arg):

# arg: 子进程运行的参数

time.sleep(1)

return arg\*\*2

def main():

test\_data = [1, 2, 3, 4]

print("# 顺序执行：")

start\_time = time.time()

for d in test\_data:

run(d)

end\_time = time.time()

print("执行时间：", int(end\_time - start\_time))

print("并行执行：")

pool = Pool(3)

start\_time = time.time()

res = pool.map(run, test\_data)

pool.close()

pool.join()

end\_time = time.time()

print("执行时间：", int(end\_time - start\_time))

print(res)

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

main()