# 需要用到的Linux命令

* 查看进程树：ps -tree
* 查看父子进程的id关系：ps -ajx
* 显示进程信息：ps -aux

# 多任务编程

定义：通过应用程序使用计算机的多个核心达到同时执行多个任务的目的，一次提高计算机运行效率。

实施方案：多进程、多线程

并行：多个计算机核心在同时处理多个任务

并发：同时处理多个任务，内核在多个任务间不断的切换，但实际一个时间点内核只能处理其中一个任务。

## 进程Process

定义：程序在计算机中的一次运行过程

* 程序：是一个可执行的文件，是静态的占有磁盘空间，不占有计算机的资源
* 进程：是一个动态过程的描述，占有计算机的资源，有一定的声明周期

\*\*\*同一个程序的不同运行过程是不同的进程，占用资源和生命周期都不一样。

进程的创建流程：

1. 用户通过运行程序或者调用接口发起创建进程
2. 操作系统接受用户请求，开始创建进程
3. 操作系统分配计算机资源，确定进程状态，开辟进程空间等操作
4. 操作系统将创建好的进程提供给应用程序使用

进程特性：

1. 进程之间互不影响，各自独立运行
2. 进程是操作系统分配的最小单位
3. 每个进程空间独立，各自占有一定的虚拟内存

### **Cpu时间片**

定义：如果一个进程占有计算机核心，我们说该进程占有计算机cpu时间片。

1. 多个任务之间是争夺cpu的关系
2. 谁占有cpu最终是操作系统决定的

### PCB进程控制块

定义：在内存中开辟的一块空间，用来记录进程的信息

* 进程控制块是操作系统查找识别进程的标志
* PID（Process ID）：在操作系统中每个进程都有一个唯一的ID号用来区别于其他进程。ID号由操作系统自动分配，是一个大于0的整数。
* 父子进程：在系统中除了初始化进程，每个进程都有一个父进程，可能有0个或多个子进程，由此形成父子进程。

### 进程的状态

三态：

* 就绪态：进程具备执行条件，等待系统资源的分配
* 运行态：进程占有CPU时间片正在运行
* 等待态：进程暂时不具备执行条件阻塞等待，让出CPU时间片，满足条件后在就绪

五态

* 新建态：创建一个新进程，获取资源的过程
* 三态
* 终止态：进程执行结束，资源释放回收的过程

子进程关闭被自动处理,以此达到不存在僵尸进程：

signal(SIGCHLD,SIG\_IGN)

关于Linux（ps -aux）显示的进程状态信息（STAT）说明：

* S 等待态【可中断等待】
* D 等待态【不可中断等待】
* T 等待态【暂停状态】
* R 运行态【包含就绪态】
* Z 僵尸进程
* < 高优先级进程
* N 优先级较低
* 1 有子进程的
* S 会话组组长
* + 前台进程

### 进程优先级

定义：决定了一个进程的执行权限，和占有资源的优先程度

### 多进程编程

* 子进程会复制父进程全部代码段，包括fork之前产生的内存空间
* 子进程从fork的下一句开始执行，与父进程互不干扰
* 父子进程的执行顺序是不一定的，父子进程共用一个终端显示
* 父子进程空间独立，操作的都是本空间的内容，互不干扰，拥有各自的PID
* 父子进程会根据fork返回值的差异选择执行不同的代码，所以if结构几乎都是fork的固定搭配。
* Pid,status = os.waitpid(pid,option) pid退出的子进程pid，state退出时的状态
  1. pid -1任意子进程退出都处理
  2. 0阻塞 os.WNOHANG非阻塞

**多进程测试：**

****

### 二级子进程

* 父进程使用wait回收子进程，子进程在创建一个子进程然后关闭，来达成让系统管理子进程的效果

# 进程共享空间

## 管道通信

* 在内存中开辟管道空间，生成管道对象，多个进程使用同一管道进行读写操作
* from multiptocessing import Pipe
* fd1,fd2 = Pipe(duplex = True)
  + 默认表示双向管道，False单向管道
    - 双向2边都可读写
    - 单向fd1只读，fd2只写
* fd.recv()读取写入的内容
* fd.send(msg)写入要读取的内容

## 消息队列

* 在内存中建立队列模型，进程通过队列存取消息实现通信

## 共享内存Array、Value

* 在内存中开辟一块空间，进程可以写入和读取内容，但是每次写入内容都会覆盖之前内容。
* Array(ctype,data)：
  + ctype列表元素类型
  + data列表数据



# 信号通信signal

|  |  |  |
| --- | --- | --- |
| **信号名称** | **信号含义** | **默认处理方法** |
| **SIGHUP** | **连接断开** |  |
| **SIGINT** | **CTRL+C** | **KeyboardInterrupt** |
| **SIGQUIT** | **CTRL+\** |  |
| **SIGTSTP** | **CTRL+Z** |  |
| **SIGKILL** | **终止一个进程** |  |
| **SIGSTOP** | **暂停一个进程** |  |
| **SIGALRM** | **时钟信号** |  |
| **SIGCHLD** | **子进程状态改变时给父进程发出** |  |

同步执行：按照逐句执行、一步完成在进行下一步

异步执行：在执行过程中利用内核记录延迟发生或者准备处理的事件。这样不影响应用层的持续执行。当事件发生，由内核告知应用层处理。

\*信号是唯一的异步通信方法

* 在父进程中忽略子进程状态改变,子进程退出自动由系统处理，需要在创建进程前使用
  + signal.signal(signal.SIGCHLD,signal.SIG\_IGN)

**信号通信测试：**



## 信号量（信号灯）Semaphore

* 给定一个数量，对多个进程可见，且多个进程都可以操作。进程通过对数量多少的判断执行各自的行为。
* Sem = Semaphore(信号量初始值)
* Sem.acquire()消耗一个信号量
* Sem.release()增加一个信号量
* Sem\_get\_value()获取信号量值

**信号灯测试：**



# 进程的同步互斥

**临界资源：多个进程或者线程都能够操作的共享资源**

**临界区：操作临界资源的代码段**

**同步：同步是一种合作关系，为完成某个任务，多进程或者多线程之间形成一种协调，按照约定或者条件执行操作临界资源。**

**互斥：是一种制约关系，当一个进程或者线程使用临界资源的时候进行上锁处理，另一个进入阻塞等待，直到解锁后才能继续使用**

**进程的同步互斥解决方式（even、Lock）：**

****

# 线程

定义：线程也是一种多任务编程方法，可以利用计算机多核资源完成程序的并发执行。线程又被称为轻量级的进程。

特征：

1. 是计算机多核分配的最小单位
2. 一个进程可以包含多个线程
3. 线程也是一个运行的过程，消耗计算机资源，多个线程共享进程的资源和空间
4. 线程的创建删除消耗的资源都要远远小于进程
5. 多个线程之间执行互不干扰
6. 线程也有自己的特有属性，比如指令集 ID

## threading模块

## 线程通信

通信方法：多个线程共享进程的空间，所以线程间通信使用全局变量完成。

注意事项：线程间使用全局变量往往要同步互斥机制保证通信安全。

**线程声明和通信方式：**

****

**自定义线程：**

****

## Python线程的GIL问题（全局解释器锁）

Python-------- >支持多线程---------- >同步互斥---------- >加锁 ------------->超级锁，给解释器加锁 ---- >解释器同一时刻只能解释一个线程

后果：一个解释器同一个时刻只能解释执行一个线程，所以导致python线程效率低下。但是当遇到IO阻塞时线程会自动让出解释器，因此python线程更加适合高延迟的IO程序并发。

解决方法：

1. 尽量用进程完成开发
2. 不适用c解释器
3. 尽量使用多种方案组合的方式进行并发操作，线程用作高延迟IO

**进程线程速度比较：**



## 进程和线程的区别

1. 都是多任务编程方式，都能够使用计算机的多核资源
2. 进程创建删除消耗的计算机资源比线程多
3. 进程空间独立，数据互不干扰，有专门的IPC（进程的通信）！线程使用全局变量进行通信
4. 一个进程可以有多个线程，多个线程共用进程的资源，线程在资源操作时往往需要同步互斥
5. 进程线程在系统中都有自己特有的属性，ID，代码段，栈区等资源
6. 进程是操作系统分配的最小单位，线程是CPU分配的最小单位

### 使用场景

* 需要创建较多并发，同时任务关联系比较强时一般使用多线程
* 不同的任务模块可能更多用进程
* 使用进程线程需要考虑数据的处理复杂性，比如进程间通信是否方便，同步互斥是否过于复杂。

### 面试问题：

* 进程线程的区别和联系
* 进程间通信方式都知道哪些，有什么特点
* 同步互斥意义是什么？什么情况下用
* 给一个情形，分析下用进程还是用线程，理由
* 一些常见概念：僵尸进程、进程状态、GIL

# 线程池和进程池

* ​Python3.2开始，标准库的concurrent.futures模块，ThreadPoolExecutor和ProcessPoolExecutor两个类，实现了对threading和multiprocessing的进一步抽象（这里主要关注线程池），不仅可以帮我们***自动调度线程***，还可以做到：
  + 主线程可以获取某一个线程（或者任务的）的状态，以及返回值。
  + 当一个线程完成的时候，主线程能够立即知道。
  + 让多线程和多进程的编码接口一致。

## 实例

### 简单使用

* from concurrent.futures import ThreadPoolExecutor
* executor = ThreadPoolExecutor(max\_workers=1)
  + max\_workers最大同时运行的数量
* task1 = executor.submit(fn, \*args, \*\*kwargs)
  + 通过submit函数提交执行的函数到线程池中,submit函数立即返回,不阻塞
  + fn对应的函数
  + 后两个为该函数的参数
  + task1是<Future at 0x7f8008c14f98 state=running>
    - state代表其当前的状态
* task1.done()
  + done用于判定某个任务是否完成
* task2.cancel()
  + cancel用于取消某个任务，该任务还没有放入线程池中才能取消
  + 返回是否成功取消任务
* task1.result()
  + result方法可以获取task的执行结果，是个阻塞函数

### as\_completed

* as\_completed()方法是一个生成器，在没有任务完成的时候，会阻塞，在有某个任务完成的时候，会yield这个任务，就能执行for循环下面的语句，然后继续阻塞住，循环到所有的任务结束。
* from concurrent.futures import ThreadPoolExecutor,as\_completed
* all\_task = [executor.submit(get\_html, url) 循环语句]
  + 批量获取submit函数返回的Future对象
* for future in as\_completed(all\_task):
  + data = future.result()

例： 

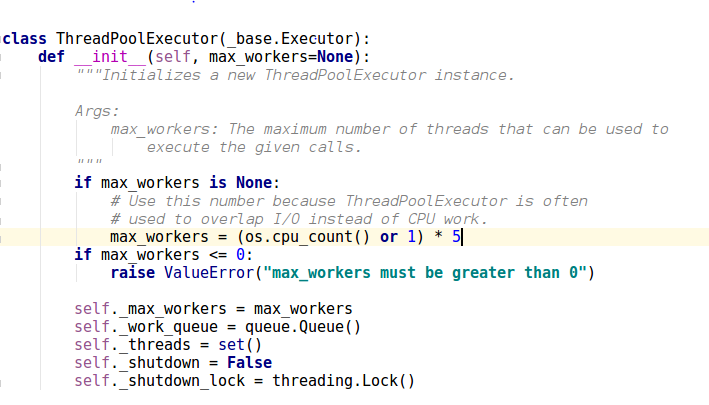
### map

* 使用map方法，无需提前使用submit方法，map方法与python标准库中的map含义相同，都是将序列中的每个元素都执行同一个函数。上面的代码就是对urls的每个元素都执行get\_html函数，并分配各线程池。
* for data in executor.map(get\_html, urls):

### wait

* from concurrent.futuresm import ALL\_COMPLETED,FIRST\_COMPLETED,wait
* wait(all\_task,time, return\_when=FIRST\_COMPLETED)
  + 等待的任务序列、超时时间以及等待条件
  + 等待条件
    - 默认为ALL\_COMPLETED，表明要等待所有的任务都结束。
    - FIRST\_COMPLETED，表示第一个任务完成就停止等待。

### 源码分析

* cocurrent.future模块中的future的意思是未来对象，可以把它理解为一个在未来完成的操作，这是异步编程的基础 。在线程池submit()之后，返回的就是这个future对象，返回的时候任务并没有完成，但会在将来完成。也可以称之为task的返回容器，这个里面会存储task的结果和状态。
* init方法中主要重要的就是任务队列和线程集合，在其他方法中需要使用到。
* 
* submit中有两个重要的对象，\_base.Future()和\_WorkItem()对象，\_WorkItem()对象负责运行任务和对future对象进行设置，最后会将future对象返回，可以看到整个过程是立即返回的，没有阻塞。
* 这个方法的含义很好理解，主要是创建指定的线程数。但是实现上有点难以理解，比如线程执行函数中的weakref.ref，涉及到了弱引用等概念，留待以后理解。
* \_WorkItem对象的职责就是执行任务和设置结果。这里面主要复杂的还是self.future.set\_result(result)。
* 这是线程池创建线程时指定的函数入口，主要是从队列中依次取出task执行，但是函数的第一个参数还不是很明白。留待以后

# 服务器模型

服务器目标：处理速度更快，并发量更高，安全性更强

硬件：更高的配置，更好的集成分布技术，更好的网络优化和网络安全技术

软件：占用资源更少，运行更稳定，算法更优良，安全性更好，并发性更高，更容易扩展

## 循环模型

定义：循环接收客户端请求，处理请求。同一个时刻只能处理一个请求，处理完毕后再处理下一个

优点：实现简单，占用资源少

缺点：无法同时处理多个客户端任务

适用情况：处理的任务可以短时间完成，不需要建立并发，更适合udp适用

## 并发模型

定义：能够同时处理多个客户端请求

### IO并发

定义：IO多路复用

优点：资源消耗少，IO处理速度快

缺点：不能适用CPU密集型程序

### 多进程/多线程并发

定义：为每个客户端创建单独的进程/线程执行请求

优点：每个客户端可以长期占有服务器运行程序，能够使用多核资源，可以处理IO或者cpu运算

缺点：消耗系统资源高

**多进程并发服务器模型：**

****

## FTP文件服务器

* 服务端和客户端2部分，要求启动一个服务端，可以同时处理多个客户端请求

1.可以查看服务端文件库中所有的普通文件

2.从客户端可以下载文件库的文件到本地

3.可以将本地文件上传到服务器文件库

4.退出

# 集成模块

* + StreamRequestHandler 流式套接字请求处理
  + DatagramRequestHandler 数据报套接字请求处理
  + TCPServer TCP服务器
  + UDPServer UDP服务器
  + ForkingTCPServer TCP进程服务器
  + ForkingUDPServer UDP进程服务器
  + ThreadingTCPServer TCP线程服务器
  + ThreadingUDPServer UDP线程服务器
  + ForkingMixIn 创建多进程
  + ThreadingMixIn 创建多线程

**集成模块的测试：**



## 协程基础

定义：线程，微线程。协程的本质是一个单线程程序，所以线程不能够使用计算机多核资源。

作用：能够高效的完成并发任务，占用较少的资源。因此协程的并发量较高。

原理：通过记录应用层的上下文栈区，实现在运行中进行上下文跳转，达到可以选择性的运行想要运行的部分，以此提高程序的运行效率。

优点：消耗资源少，无须切换开销，无需同步互斥，IO并发性好

缺点：无法利用计算机多核

yield------------------- >协程实现的基本关键字

### greenlet



### gevent



**线程网页服务器的测试：**



# PIP的使用

作用：管理python的标准第三方库中第三方软件包

安装指令：sudo apt-get install python3-pip

常用命令：(pip3不行就用pip)

* 安装软件： sudo pip3 install [package]

如：sudo pip3 install ssh

* 查看当前python软件包： pip3 list
* 搜索某个名字的python包： pip3 search [name]
* 查看软件包信息： pip3 show [package]
* 升级软件包： pip3 install --upgrade [package]
* 卸载软件包： pip3 uninstall [package]
* 导出软件包环境： pip3 freeze > requirements.txt
* 根据文档自动安装： pip3 install -r requirements.txt

# PDB调试方法

import pdb

功能：断点设置、单步执行、查看代码、查看函数、追踪变量等…

命令：

* b break 设置断电
* c continue 继续执行
* n next 单步执行
* s step 单步执行，可以进入函数内部
* l list 查看代码段
* pp 查看某个变量值
* help 帮助

进入pdb调试模式：pdb.set\_trace()

在命令行进入pdb调试模式运行：python3 -m pdb [.py文件]