## 进程线程间的通信

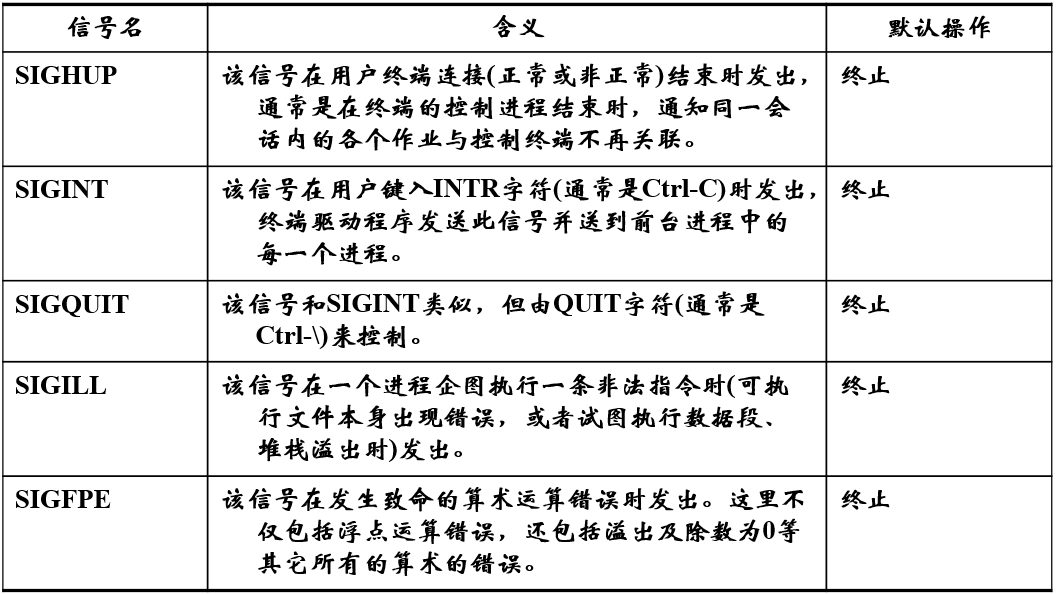
想要有效地使用多个进程，通常要求他们之间有某种通信。进程间通信就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。另外，系统空间是“公共场所”，各进程均可以访问，所以内核也可以提供这样的条件。此外，还有双方都可以访问的外设。在这个意义上，两个进程当然也可以通过磁盘上的普通文件交换信息，或者通过“注册表”或其它数据库中的某些表项和记录交换信息。广义上这也是进程间通信的手段，但是一般都不把这算作“进程间通信”。   
我们所说的进程间通信（IPC，Interprocess communication）是一组编程接口，让程序员能够协调不同的进程，使之能在一个操作系统里同时运行，并相互传递、交换信息。这使得一个程序能够在同一时间里处理许多用户的要求。因为即使只有一个用户发出要求，也可能导致一个操作系统中多个进程的运行，进程之间必须互相通话。IPC接口就提供了这种可能性。每个IPC方法均有它自己的优点和局限性。

### 信号

信号是一个操作系统的特性，它提供了一个途径可以通知进程发生了一个事件并异步处理这个事件。信号可以由系统本身生成，也可以从一个进程发送到另一个进程。由于信号会中断程序的正常控制流，如果在中间接收到信号有些才做可能会产生错误。

信号是在软件层次上对中断机制的一种模拟，是一种异步通信方式。信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程 。

linux下常见信号



信号名: SIGHUP

含义:在用户终端链接结束时发出,通常是在终端的控制进程结束时,通知同一会话内的各个作业与控制终端不再联系

默认操作:终止

信号名:SIGINT

含义:在用户输入 Ctr+C时发出,终端驱动程序发送此信号并送到前台进程中的每一个进程

默认:终止

SIGQUIT: 与SIFINT类似,但由QUIT字符(通常是Ctrl+/)来控制

默认:终止

SIGILLL:在一个进程企图执行一条非法指令时发出

默认:终止

SIGFPE: 在发生致命的运算错误是发生,不仅包括浮点运算错误,还高阔溢出及除数为0等其他所有的算术的错误

默认:终止

SIGKILL:用于立即结束程序的运行,并且不能被阻塞,处理和忽略

默认:终止

SIGALARM:当一个定时器到时的时候发出信号.

默认:终止

SIGSTOP:用于暂停一个进程,且不能被阻塞,处理和忽略

默认:暂停进程

SIGTSTP:用于暂停交互进程,用户可以输入Ctrl+Z来发出这个信号

默认:暂停进程

SIGCHLD:子进程改变状态时,父进程会收到这个信号

默认:忽略

SIGABORD:结束进程

默认:终止

#### 常用函数：

os模块:kill()

signal模块:

signal()

alarm()

pause()

### 向进程传递消息

Python中进程见消息传递的方式常用的有Queue、Pipe和Socket的方法，Socket在后面会讲解,

利用multiprocess完成进程间通信的一种简单方法是使用一个Queue来回传递消息。能够用pickle串行化的任何对象都可以通过Queue传递。

pipe管道是比较老牌的进程间通信方式，在Python中也比较常用。

## 进程间的同步与互斥

进程同步是一个操作系统级别的概念,是在多道程序的环境下，存在着不同的制约关系，为了协调这种互相制约的关系，实现资源共享和进程协作，从而避免进程之间的冲突，引入了进程同步。

### 临界资源

在操作系统中，进程是占有资源的最小单位（线程可以访问其所在进程内的所有资源，但线程本身并不占有资源或仅仅占有一点必须资源）。但对于某些资源来说，其在同一时间只能被一个进程所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源。典型的临界资源比如物理上的打印机，或是存在硬盘或内存中被多个进程所共享的一些变量和数据等(如果这类资源不被看成临界资源加以保护，那么很有可能造成丢数据的问题)。对于临界资源的访问，必须是互诉进行。也就是当临界资源被占用时，另一个申请临界资源的进程会被阻塞，直到其所申请的临界资源被释放。而进程内访问临界资源的代码被成为临界区。

对于临界区的访问过程分为四个部分：

1. 进入区:查看临界区是否可访问，如果可以访问，则转到步骤二，否则进程会被阻塞

2. 临界区:在临界区做操作

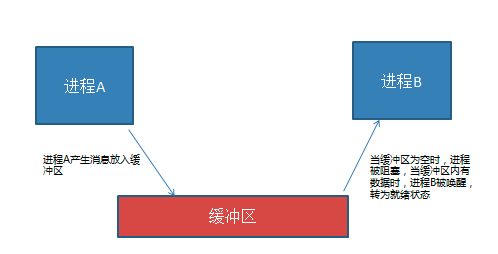
3. 退出区:清除临界区被占用的标志

4. 剩余区：进程与临界区不相关部分的代码

进程同步

进程同步也是进程之间直接的制约关系，是为完成某种任务而建立的两个或多个线程，这个线程需要在某些位置上协调他们的工作次序而等待、传递信息所产生的制约关系。进程间的直接制约关系来源于他们之间的合作。

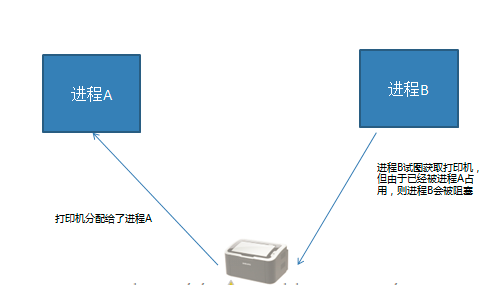
比如说进程A需要从缓冲区读取进程B产生的信息，当缓冲区为空时，进程B因为读取不到信息而被阻塞。而当进程A产生信息放入缓冲区时，进程B才会被唤醒。概念如图所示。



进程互斥

进程互斥是进程之间的间接制约关系。当一个进程进入临界区使用临界资源时，另一个进程必须等待。只有当使用临界资源的进程退出临界区后，这个进程才会解除阻塞状态。

比如进程B需要访问打印机，但此时进程A占有了打印机，进程B会被阻塞，直到进程A释放了打印机资源,进程B才可以继续执行。概念如图所示。



在Python中解决进程间同步互斥的的方法通常有事件，锁，条件变量。

### 事件

Event类提供了一种简单的方法，可以在进程之间传递状态信息。时间可以切换设置和未设置状态。通过使用一个可选的超时值，时间对象的用户可以等待其状态从未设置变为设置。

[Event演示](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/Event_p.py" \t "/home/jason/文档\\x/_blank)

### 锁

如果需要在多个进程见共享一个资源，在这种情况下，可以使用一个Lock来避免访问冲突。

[Lock演示](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/lock_p.py" \t "/home/jason/文档\\x/_blank)

### 条件

Condition对象可以用来同步一个工作流的各个部分，使其中一些部分并行运行，而另外一些顺序运行，即使他们在单独的进程中。

[Cond演示](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/cond.py" \t "/home/jason/文档\\x/_blank)

## python中的多线程编程

Python 提供了几个用于多线程编程的模块，包括 thread, threading 和 Queue 等。thread和threading模块允许程员创建和管理线程。thread模块提供了基本的线程和锁的支持，而threading提供了更高级别，功能更强的线程管理的功能。Queue模块允许用户创建一个可以用于多个线程之间共享数据的队列数据结构。我们将分别介绍这几个模块，并给出一些例子和中等大小的应用。   
出于以下几点考虑，我们不建议您使用 thread 模块。首先，更高级别的 threading 模块更为先进， 对线程的支持更为完善， 而且使用 thread 模块里的属性有可能会与 threading 出现冲突。 其次，低级别的 thread模块的同步原语很少（实际上只有一个） ，而 threading 模块则有很多。另一个不要使用thread原因是，对于你的进程什么时候应该结束完全没有控制，当主线程结束时，所有的线程都会被强制结束掉，没有警告也不会有正常的清除工作。我们之前说过，至少threading模块能确保重要的子线程退出后进程才退出。   
不过，出于对学习Python和线程的兴趣，你可以尝试使用thread，但是在最新的Python3版本中已经不再支持thread模块了。

### threading——管理并发操作

#### threading 模块对象

[Thread演示1](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/thread.py" \t "/home/jason/文档\\x/_blank)

[Thread演示2](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/thread1.py" \t "/home/jason/文档\\x/_blank)

[Thread演示3](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/player.py" \t "/home/jason/文档\\x/_blank)

[Thread演示4](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/player1.py" \t "/home/jason/文档\\x/_blank)

## 同步与互斥机制

### 信号传递

尽管使用多线程的目的是并发地运行单个操作，不过有时也需要在两个或多个线程中同步操作。时间对象是实现线程见安全通信的一种简单方法。Event管理一个内部标志，调用者可以用set()和clear()方法控制这个标志。其他线程可以使用wait()暂停，直到设置这个标志，其想过就是在允许继续之前阻塞线程的运行。

[线程信号演示1](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/event1.py" \t "/home/jason/文档\\x/_blank)

[线程信号演示2](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/event.py" \t "/home/jason/文档\\x/_blank)

### 线程锁

除了同步线程操作之外，还有一点很重要，要能构控制对共享资源的访问，从而避免破坏或丢失数据。Python的内置数据结构（列表、字典等等）是线程安全的，这是Python使用源字节码来管理这些数据结构的一个副作用。Python中实现的其他数据结构或更简单的类型则没有这个保护。要保证同时安全地访问一个对象，可以使用一个Lock对象。

[线程锁演示1](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/lock1.py" \t "/home/jason/文档\\x/_blank)

[线程锁演示2](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/lock.py" \t "/home/jason/文档\\x/_blank)

## 生产者消费者模式

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

[生产者消费者模式](http://nts.newbieol.com/read?src=/static/k7/Python/class-013/src/producer_consumer_mode.py" \t "/home/jason/文档\\x/_blank)

## 新的技术

进程有一个问题，就是进程间切换耗费计算机资源非常大，而且申请一个新的进程的成本也非常高。所以后来就有了线程，它生成的成本和切换的消耗都比进程要低很多，而且线程间通信也非常方便，但是线程也有问题。线程的问题是：1.存在线程安全问题，出了问题非常不易定位。2.进程内部有线程数目的限制。3.随着并发量的增加，线程生成和切换的成本也变得昂贵。解决并发还有一个方案是IO多路复用，它的效率确实非常高，但是代码复杂度也非常高：它把一个流程打散成一个个的节点，散落在多个地方，对开发和维护非常不利。

在python中引入了一个协程的概念，我们简短介绍下同学们将来工作中可能会遇到，协程是如何解决这些问题的：

协程的生成成本更低。其实就是一块内存，记录之前的调用的栈信息。你甚至可以通过控制函数调用的层次来进一步降低协程的大小。要生成一个协程，只需要申请一块内存并赋值。

切换更快。基本是就是内存的拷贝的速度。

没有线程安全问题。一个进程内可以同时存在多个协程，但是只有一个协程是激活的，而且协程的激活和休眠时程序员通过编程来控制，而不是内核来控制的。这样就没有了线程安全问题。

可读性更好。相对于IO多路复用来说，你调用的服务接口或者IO接口是异步的，但是你的代码是流畅（顺序）的，并没有被异步和回调打乱。协程也是异步的，但是它会把异步的事件和回调封装起来，形成类似远程调用接口。

### Python的协程实现：

yield可以实现协程。另外，还有很多第三方的版本，比如greenlet。

### 协程可以用来做什么？

1. 描述逻辑：我主要把协程用来描述逻辑。一个流程可能需要调用多个接口，其中很多接口是异步的。这样描述起来会困难一点。用线程是可以解决部分问题，但是复杂度提升。

2. 提高并发：主要应用在IO密集型应用中。gevent就是在greenlet基础之上的一个处理并发的框架，和上面的区别是，这里的事件及接口是IO接口。

### 协程缺陷

无法使用多核。不过可以通过 进程 + 协程 来解决。