**同步IO（阻塞非阻塞）**

一、同步IO 之 Blocking IO

用户进程process在Blocking IO读recvfrom操作的两个阶段都是等待的。**在数据没准备好的时候，process原地等待kernel准备数据。kernel准备好数据后，process继续等待kernel将数据copy到自己的buffer。在kernel完成数据的copy后process才会从recvfrom系统调用中返回。**

二、同步IO 之 NonBlocking IO

process在NonBlocking IO读recvfrom操作的第一个阶段是不会block等待的，如果kernel数据还没准备好，那么recvfrom会立刻返回一个EWOULDBLOCK错误。当kernel准备好数据后，进入处理的第二阶段的时候，process会等待kernel将数据copy到自己的buffer，在kernel完成数据的copy后process才会从recvfrom系统调用中返回。

**异步IO**

异步IO要求process在recvfrom操作的两个处理阶段上都不能等待，也就是process调用recvfrom后立刻返回，kernel自行去准备好数据并将数据从kernel的buffer中copy到process的buffer在通知process读操作完成了，然后process在去处理。遗憾的是，linux的网络IO中是不存在异步IO的，linux的网络IO处理的第二阶段总是阻塞等待数据copy完成的。真正意义上的网络异步IO是Windows下的IOCP（IO完成端口）模型。

**Socket事件的wakeup callback机制**

linux(2.6+)内核的事件wakeup callback机制，这是IO多路复用机制存在的本质。Linux通过socket睡眠队列来管理所有等待socket的某个事件的process，同时通过wakeup机制来异步唤醒整个睡眠队列上等待事件的process，通知process相关事件发生。通常情况，socket的事件发生的时候，其会顺序遍历socket睡眠队列上的每个process节点，调用每个process节点挂载的callback函数。在遍历的过程中，如果遇到某个节点是排他的，那么就终止遍历，总体上会涉及两大逻辑：（1）睡眠等待逻辑；（2）唤醒逻辑。

1. 睡眠等待逻辑：涉及select、poll、epoll\_wait的阻塞等待逻辑

[1]select、poll、epoll\_wait陷入内核，判断监控的socket是否有关心的事件发生了，如果没，则为当前process构建一个wait\_entry节点，然后插入到监控socket的sleep\_list

[2]进入循环的schedule直到关心的事件发生了

[3]关心的事件发生后，将当前process的wait\_entry节点从socket的sleep\_list中删除。

（2）唤醒逻辑：

[1]socket的事件发生了，然后socket顺序遍历其睡眠队列，依次调用每个wait\_entry节点的callback函数

[2]直到完成队列的遍历或遇到某个wait\_entry节点是排他的才停止。

[3]一般情况下callback包含两个逻辑：1.wait\_entry自定义的私有逻辑；2.唤醒的公共逻辑，主要用于将该wait\_en

**epoll和select以及poll**

IO多路复用，就是我们熟知的select、poll、epoll模型。在IO多路复用的时候，process在两个处理阶段都是block住等待的。初看好像IO多路复用没什么用，其实select、poll、epoll的优势在于可以以较少的代价来同时监听处理多个IO。

**Select—1024**

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

当用户process调用select的时候，select会将需要监控的readfds集合拷贝到内核空间（假设监控的仅仅是socket可读），然后遍历自己监控的socket sk，挨个调用sk的poll逻辑以便检查该sk是否有可读事件，遍历完所有的sk后，如果没有任何一个sk可读，那么select会调用schedule\_timeout进入schedule循环，使得process进入睡眠。如果在timeout时间内某个sk上有数据可读了，或者等待timeout了，则调用select的process会被唤醒，接下来select就是遍历监控的sk集合，挨个收集可读事件并返回给用户了。

select存在两个问题：

[1] 被监控的fds需要从用户空间拷贝到内核空间

为了减少数据拷贝带来的性能损坏，内核对被监控的fds集合大小做了限制，并且这个是通过宏控制的，大小不可改变(限制为1024)。

[2] 被监控的fds集合中，只要有一个有数据可读，整个socket集合就会被遍历一次调用sk的poll函数收集可读事件

由于当初的需求是朴素，仅仅关心是否有数据可读这样一个事件，当事件通知来的时候，由于数据的到来是异步的，我们不知道事件来的时候，有多少个被监控的socket有数据可读了，于是，只能挨个遍历每个socket来收集可读事件。

到这里，我们有三个问题需要解决：

（1）被监控的fds集合限制为1024，1024太小了，我们希望能够有个比较大的可监控fds集合 （2）fds集合需要从用户空间拷贝到内核空间的问题，我们希望不需要拷贝

（3）当被监控的fds中某些有数据可读的时候，我们希望通知更加精细一点，就是我们希望能够从通知中得到有可读事件的fds列表，而不是需要遍历整个fds来收集。

**poll—鸡肋**

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

select遗留的三个问题中，问题(1)是用法限制问题，问题(2)和(3)则是性能问题。poll和select非常相似，poll并没着手解决性能问题，poll只是解决了select的问题(1)fds集合大小1024限制问题。下面是poll的函数原型，poll改变了fds集合的描述方式，使用了pollfd结构而不是select的fd\_set结构，使得poll支持的fds集合限制远大于select的1024。poll虽然解决了fds集合大小1024的限制问题，但是，它并没改变大量描述符数组被整体复制于用户态和内核态的地址空间之间，以及个别描述符就绪触发整体描述符集合的遍历的低效问题。poll随着监控的socket集合的增加性能线性下降，poll不适合用于大并发场景。

**通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。**

1 fds集合拷贝问题的解决

对于IO多路复用，有两件事是必须要做的(对于监控可读事件而言)：1. 准备好需要监控的fds集合；2. 探测并返回fds集合中哪些fd可读了。细看select或poll的函数原型，我们会发现，每次调用select或poll都在重复地准备(集中处理)整个需要监控的fds集合。然而对于频繁调用的select或poll而言，fds集合的变化频率要低得多，我们没必要每次都重新准备(集中处理)整个fds集合。

于是，epoll引入了epoll\_ctl系统调用，将高频调用的epoll\_wait和低频的epoll\_ctl隔离开。

同时，epoll\_ctl通过(EPOLL\_CTL\_ADD、EPOLL\_CTL\_MOD、EPOLL\_CTL\_DEL)三个操作来分散对需要监控的fds集合的修改，做到了**有变化才变更**，将select或poll高频、大块内存拷贝(集中处理)变成epoll\_ctl的低频、小块内存的拷贝(分散处理)，**避免了大量的内存拷贝**。

同时，对于高频epoll\_wait的可读就绪的fd集合返回的拷贝问题，epoll通过内核与用户空间mmap(内存映射)同一块内存来解决。mmap将用户空间的一块地址和内核空间的一块地址同时映射到相同的一块物理内存地址（不管是用户空间还是内核空间都是虚拟地址，最终要通过地址映射映射到物理地址），使得这块物理内存对内核和对用户均可见，减少用户态和内核态之间的数据交换。

另外，epoll通过epoll\_ctl来对监控的fds集合来进行增、删、改，那么必须涉及到fd的快速查找问题，于是，一个低时间复杂度的增、删、改、查的数据结构来组织被监控的fds集合是必不可少的了。在linux 2.6.8之前的内核，epoll使用hash来组织fds集合，于是在创建epoll fd的时候，epoll需要初始化hash的大小。于是epoll\_create(int size)有一个参数size，以便内核根据size的大小来分配hash的大小。在linux 2.6.8以后的内核中，epoll使用红黑树来组织监控的fds集合，于是epoll\_create(int size)的参数size实际上已经没有意义了。

2 按需遍历就绪的fds集合

通过上面的socket的睡眠队列唤醒逻辑我们知道，socket唤醒睡眠在其睡眠队列的wait\_entry(process)的时候会调用wait\_entry的回调函数callback，并且，我们可以在callback中做任何事情。为了做到**只遍历就绪的fd**，我们需要有个地方来组织那些已经就绪的fd。为此，epoll引入了一个中间层，一个**双向链表(ready\_list)**，一个单独的**睡眠队列(single\_epoll\_wait\_list)**，并且，与select或poll不同的是，epoll的process不需要同时插入到多路复用的socket集合的所有睡眠队列中，相反**process只是插入到中间层的epoll的单独睡眠队列中，process睡眠在epoll的单独队列上，等待事件的发生**。

同时，引入一个中间的wait\_entry\_sk，它与某个socket sk密切相关，wait\_entry\_sk睡眠在sk的睡眠队列上，其callback函数逻辑是将当前sk排入到epoll的ready\_list中，并唤醒epoll的single\_epoll\_wait\_list。而single\_epoll\_wait\_list上睡眠的process的回调函数就明朗了：遍历ready\_list上的所有sk，挨个调用sk的poll函数收集事件，然后唤醒process从epoll\_wait返回。

于是，整个过来可以分为以下几个逻辑：

（1）epoll\_ctl EPOLL\_CTL\_ADD逻辑

[1] 构建睡眠实体wait\_entry\_sk，将当前socket sk关联给wait\_entry\_sk，并设置wait\_entry\_sk的回调函数为epoll\_callback\_sk

[2] 将wait\_entry\_sk排入当前socket sk的睡眠队列上

回调函数epoll\_callback\_sk的逻辑如下：

[1] 将之前关联的sk排入epoll的ready\_list

[2] 然后唤醒epoll的单独睡眠队列single\_epoll\_wait\_list

（2）epoll\_wait逻辑

[1] 构建睡眠实体wait\_entry\_proc，将当前process关联给wait\_entry\_proc，并设置回调函数为epoll\_callback\_proc

[2] 判断epoll的ready\_list是否为空，如果为空，则将wait\_entry\_proc排入epoll的single\_epoll\_wait\_list中，随后进入schedule循环，这会导致调用epoll\_wait的process睡眠。

[3] wait\_entry\_proc被事件唤醒或超时醒来，wait\_entry\_proc将被从single\_epoll\_wait\_list移除掉，然后wait\_entry\_proc执行回调函数epoll\_callback\_proc

回调函数epoll\_callback\_proc的逻辑如下：

[1] 遍历epoll的ready\_list，挨个调用每个sk的poll逻辑收集发生的事件，对于监控可读事件而已，ready\_list上的每个sk都是有数据可读的，这里的遍历必要的(不同于select/poll的遍历，它不管有没数据可读都需要遍历一些来判断，这样就做了很多无用功。)

[2] 将每个sk收集到的事件，通过epoll\_wait传入的events数组回传并唤醒相应的process。

（3）epoll唤醒逻辑 整个epoll的协议栈唤醒逻辑如下(对于可读事件而言)：

[1] 协议数据包到达网卡并被排入socket sk的接收队列

[2] 睡眠在sk的睡眠队列wait\_entry被唤醒，wait\_entry\_sk的回调函数epoll\_callback\_sk被执行

[3] epoll\_callback\_sk将当前sk插入epoll的ready\_list中

[4] 唤醒睡眠在epoll的单独睡眠队列single\_epoll\_wait\_list的wait\_entry，wait\_entry\_proc被唤醒执行回调函数epoll\_callback\_proc

[5] 遍历epoll的ready\_list，挨个调用每个sk的poll逻辑收集发生的事件

[6] 将每个sk收集到的事件，通过epoll\_wait传入的events数组回传并唤醒相应的process。

epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分：

1）调用epoll\_create()建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)

2）调用epoll\_ctl向epoll对象中添加这100万个连接的套接字

3）调用epoll\_wait收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll\_wait的效率也非常高，因为调用epoll\_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

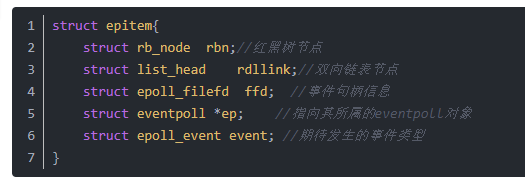
当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，



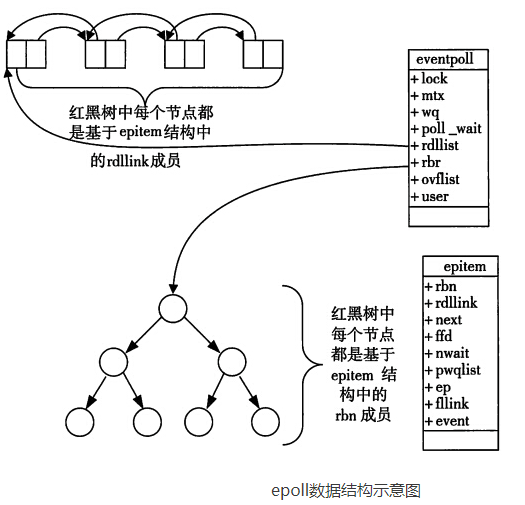
每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll\_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。

在epoll中，对于每一个事件，都会建立一个epitem结构体，



当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。



epoll的用法一句话描述就是：三步曲。

第一步：epoll\_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。

第二步：epoll\_ctl()系统调用。通过此调用向epoll对象中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。

第三部：epoll\_wait()系统调用。通过此调用收集收集在epoll监控中已经发生的事件。

**epoll的水平触发LT和边缘ET触发**

**https://cloud.tencent.com/developer/article/1005481**

    在linux的IO多路复用中有水平触发,边缘触发两种模式,这两种模式的区别如下:

    水平触发:如果文件描述符已经就绪可以非阻塞的执行IO操作了,此时会触发通知.允许在任意时刻重复检测IO的状态,没有必要每次描述符就绪后尽可能多的执行IO.select,poll就属于水平触发.

    边缘触发:如果文件描述符自上次状态改变后有**新**的IO活动到来,此时会触发通知.在收到一个IO事件通知后要尽可能多的执行IO操作,因为如果在一次通知中没有执行完IO那么就需要等到下一次新的IO活动到来才能获取到就绪的描述符.信号驱动式IO就属于边缘触发.

    epoll既可以采用水平触发,也可以采用边缘触发.

举例说明:**一个管道收到了1kb的数据,epoll会立即返回,此时读了512字节数据,然后再次调用epoll.**

**这时如果是水平触发的,epoll会立即返回,因为有数据准备好了.**

**如果是边缘触发的不会立即返回,因为此时虽然有数据可读但是已经触发了一次通知,在这次通知到现在还没有新的数据到来,直到有新的数据到来epoll才会返回,此时老的数据和新的数据都可以读取到(当然是需要这次你尽可能的多读取).**

在epoll\_wait返回socket\_fd可读的时候，我们需要小心处理，要不然会造成死锁和socket饿死现象。典型如listen\_fd返回可读的时候，我们需要不断的accept直到EAGAIN。假设同时有三个请求到达，epoll\_wait返回listen\_fd可读，这个时候，如果仅仅accept一次拿走一个请求去处理，那么就会留下两个请求，如果这个时候一直没有新的请求到达，那么再次调用epoll\_wait是不会通知listen\_fd可读的，于是epoll\_wait只能睡眠到超时才返回，遗留下来的两个请求一直得不到处理，处于饿死状态。

ET强制需要在非阻塞模式下使用。在ET模式下，epoll\_wait返回socket\_fd有数据可读，我们必须要读完所有数据才能离开。因为，如果不读完，epoll不会在通知你了，虽然有新的数据到来的时候，会再次通知，但是我们并不知道新数据会不会来，以及什么时候会来。由于在阻塞模式下，我们是无法通过recv/read来探测空数据事件，于是，我们必须采用非阻塞模式，一直read直到EAGAIN。

epoll\_wait原本的语意是：监控并探测socket是否有数据可读(对于读事件而言)。LT模式保留了其原本的语意，只要socket还有数据可读，它就能不断反馈，于是，我们想什么时候读取处理都可以，我们永远有再次poll的机会去探测是否有数据可以处理，这样带来了编程上的很大方便，不容易死锁造成某些socket饿死。相反，ET模式修改了epoll\_wait原本的语意，变成了：监控并探测socket是否有新的数据可读。

最后总结一下，ET和LT模式下epoll\_wait返回的条件

* ET - 对于读操作

[1] 当接收缓冲buffer内待读数据增加的时候时候(**由空变为不空的时候、或者有新的数据进入缓冲buffer**)

[2] 调用epoll\_ctl(EPOLL\_CTL\_MOD)来改变socket\_fd的监控事件，也就是重新mod socket\_fd的EPOLLIN事件，并且接收缓冲buffer内还有数据没读取。(这里不能是EPOLL\_CTL\_ADD的原因是，epoll不允许重复ADD的，除非先DEL了，再ADD) 因为epoll\_ctl(ADD或MOD)会调用sk的poll逻辑来检查是否有关心的事件，如果有，就会将该sk加入到epoll的ready\_list中，下次调用epoll\_wait的时候，就会遍历到该sk，然后会重新收集到关心的事件返回。

* ET - 对于写操作

[1] 发送缓冲buffer内待发送的数据减少的时候(由满状态变为不满状态的时候、或者有部分数据被发出去的时候) [2] 调用epoll\_ctl(EPOLL\_CTL\_MOD)来改变socket\_fd的监控事件，也就是重新mod socket\_fd的EPOLLOUT事件，并且发送缓冲buffer还没满的时候。

* LT - 对于读操作 LT就简单多了，唯一的条件就是，接收缓冲buffer内有可读数据的时候
* LT - 对于写操作 LT就简单多了，唯一的条件就是，发送缓冲buffer还没满的时候

在绝大多少情况下，ET模式并不会比LT模式更为高效，同时，ET模式带来了不好理解的语意，这样容易造成编程上面的复杂逻辑和坑点。因此，建议还是采用LT模式来编程更为舒爽。

**----------------------------**

**创建共享内存**

共享内存：是被多个进程共享的一部分物理内存，共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入了数据，共享这个内存区域的所有进程就可以理解看到其中的内容。

共享内存实现实现分为两个步骤：

一：创建共享内存，使用**shmget**函数

二：映射共享内存，将这段创建的共享内存映射到具体的进程空间去，使用**shmat**函数

创建：

int shmget（key\_t key，int size，int shmflg）

key标识共享内存的键值：0/IPC\_PRIVATE。当key的取值为IPC\_PRIVATE，则函数shmget（）将创建一个新的共享内存；如果key的取值为0，而参数shmflg中又设置IPC\_PRIVATE这个标志，则同样会创建一块新的共享内存。

返回值：如果成功则返回内存标识符，如果失败则返回-1；

映射：

int shmat（int shmid， char \*shmaddr，int flag）

参数：

shmid：shmget函数返回的共享内存存储标识符

flag：决定以后用什么方式来确定映射地址（通常为0）

返回值：

如果成功，则返回共享内存到进程中的地址；如果失败，则返回-1

一个进程不再需要共享内存时，需要把它从进程地址空间中脱离。

脱离函数：

int shmdt（char \*shmaddr）

# pthread\_join和pthread\_detach

每一个线程在任何情况，要么是可结合的状态（joinable），要么是可分离的状态（detached）。   
先将这两个函数的原型列一下   
int pthread\_join(pthread\_t tid, void \*\* pthread\_return);   
int pthread\_detach(pthread\_t tid);   
当我们的线程运行结束后，最后显示的调用被回收。这样就出现两种回收方式。   
1. pthread\_join是一个阻塞函数，调用方会阻塞到pthread\_join所指定的tid的线程结束后才被回收，但是在此之前，调用方是霸占系统资源的。   
2. pthread\_detach，不会阻塞，调用它后，线程运行结束后会自动释放资源。   
PS：一个可结合线程在运行结束后，若没有调用pthread\_join，会进入一个类似zombie process的状态，也就是系统中还有一些资源没有回收。需要pthread\_join来回收这些资源。（这就类似进程操作中的waitpid函数）

**线程**

线程是进程中 执行运算的最小单位，是独立运行和独立调度的基本单位

通常在一个进程中可以包含若干个线程

线程可以利用进程所拥有的资源

由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间**并发**执行的程度。

**线程安全和不安全**

**线程安全**就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

**线程不安全**就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和[单线程](http://baike.baidu.com/view/2554947.htm)运行的结果是一样的，而且其他的[变量](http://baike.baidu.com/view/296689.htm)的值也和预期的是一样的，就是线程安全的。

线程安全问题都是由[全局变量](http://baike.baidu.com/view/261041.htm)及[静态变量](http://baike.baidu.com/view/675642.htm)引起的。

若每个线程中对[全局变量](http://baike.baidu.com/view/261041.htm)、[静态变量](http://baike.baidu.com/view/675642.htm)只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑[线程同步](http://baike.baidu.com/view/6501899.htm)，否则的话就可能影响线程安全。

比如一个 ArrayList 类，在添加一个元素的时候，它可能会有两步来完成：1. 在 Items[Size] 的位置存放此元素；2. 增大 Size 的值。

在[单线程](http://baike.baidu.com/view/2554947.htm)运行的情况下，如果 Size = 0，添加一个元素后，此元素在位置 0，而且 Size=1；

而如果是在[多线程](http://baike.baidu.com/view/65706.htm)情况下，比如有两个线程，线程 A 先将元素存放在位置 0。但是此时 CPU 调度线程A暂停，线程 B 得到运行的机会。线程B也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0 （注意哦，我们假设的是添加一个元素是要两个步骤哦，而线程A仅仅完成了步骤1），所以线程B也将元素存放在位置0。然后线程A和线程B都继续运行，都增加 Size 的值。

那好，我们来看看 ArrayList 的情况，元素实际上只有一个，存放在位置 0，而 Size 却等于 2。这就是“线程不安全”了

**线程池**

线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程(N1)，放入空闲队列中。这些线程都是处于阻塞（Suspended）状态，不消耗CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当N1个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态，线程池自动销毁一部分线程，回收系统资源。   
      基于这种预创建技术，线程池将线程创建和销毁本身所带来的开销分摊到了各个具体的任务上，执行次数越多，每个任务所分担到的线程本身开销则越小，不过我们另外可能需要考虑进去线程之间同步所带来的开销。

**线程池**，简单来说就是有一堆已经创建好的线程（最大数目一定），初始时他们都处于空闲状态，当有新的任务进来，从线程池中取出一个空闲的线程处理任务，然后当任务处理完成之后，该线程被重新放回到线程池中，供其他的任务使用，当线程池中的线程都在处理任务时，就没有空闲线程供使用，此时，若有新的任务产生，只能等待线程池中有线程结束任务空闲才能执行，

**我们为什么要使用线程池呢？**

简单来说就是线程本身存在开销，我们利用多线程来进行任务处理，单线程也不能滥用，无止禁的开新线程会给系统产生大量消耗，而线程本来就是可重用的资源，不需要每次使用时都进行初始化，因此可以采用有限的线程个数处理无限的任务。

**线程同步互斥：临界区/互斥量/事件/信号量**

加入同步机制主要是为了在多线程程序中，如果需要对某个共享资源C进行同步访问，什么是同步访问，就是A线程访问过程中，B线程不能访问，必须等A线程访问结束后，B线程才能访问，而互斥锁，如果用来对C进行保护， A访问C资源的过程中，B不能访问，A访问结束后，B可以访问，但不一定访问的到，这取决于系统的调度是否给到B，如果没有，A反而被调度到了，那么A就有可能方法到C。反观同步机制，在这种情况下，如果系统没有调度到B，A也是没有可能访问C的，必须等B调度到之后，A才可能重新访问。

互斥锁主要用来保护临界资源，什么是临界资源，就是有可能多个线程都需要访问的数据地址，也有可能是某一段代码，执行这段代码有可能会改变多个线程都需要访问的数据。

线程互斥是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

实现线程同步互斥的四种方式：

1、临界区（Critical Section）：指的是一个访问共用资源（例如：共用设备或是共用存储器）的程序片段，而这些共用资源又无法同时被多个线程访问的特性。适合一个进程内的多线程访问公共区域或代码段时使用。

     2、互斥量 (Mutex)：适合不同进程内多线程访问公共区域或代码段时使用，与临界区相似。

     3、事件（Event）：通过线程间触发事件实现同步互斥

     4、信号量（Semaphore）：与临界区和互斥量不同，可以实现多个线程同时访问公共区域数据，原理与操作系统中PV操作类似，先设置一个访问公共区域的线程最大连接数，每有一个线程访问共享区资源数就减一，直到资源数小于等于零。信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即P(信号变量))和发送（即V(信号变量))信息操作。最简单的信号量是只能取0和1的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。这里主要讨论二进制信号量。

由于信号量只能进行两种操作等待和发送信号，即P(sv)和V(sv),他们的行为是这样的：

P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行

V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而挂起，就给它加1.

举个例子，就是两个进程共享信号量sv，一旦其中一个进程执行了P(sv)操作，它将得到信号量，并可以进入临界区，使sv减1。而第二个进程将被阻止进入临界区，因为当它试图执行P(sv)时，sv为0，它会被挂起以等待第一个进程离开临界区域并执行V(sv)释放信号量，这时第二个进程就可以恢复执行。

实现：

在两个线程关联的函数中加了一句 std::lock\_guard<std::mutex> mtx\_locker(mtx); C++中，通过构造std::mutex的实例来创建互斥元，可通过调用其成员函数lock()和unlock()来实现加锁和解锁，然后这是不推荐的做法，因为这要求程序员在离开函数的每条代码路径上都调用unlock()，包括由于异常所导致的在内。作为替代，标准库提供了std::lock\_guard类模板，实现了互斥元的RAII惯用语法（资源获取即初始化）。该对象在构造时锁定所给的互斥元，析构时解锁该互斥元，从而保证被锁定的互斥元始终被正确解锁。

**多线程并发**

多线程并发指的是在同一个进程中执行多个线程。

多线程是实现并发（并行）的一种手段。线程的划分尺度小于进程，使得多线程程序的并发性高。另外，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。

线程是轻量级的进程，每个线程可以独立的运行不同的指令序列，但是线程不独立的拥有资源，依赖于创建它的进程而存在。也就是说，同一进程中的多个线程共享相同的地址空间，可以访问进程中的大部分数据，**指针和引用**可以在线程间进行传递。这样，同一进程内的多个线程能够很方便的进行数据共享以及通信，也就比进程更适用于并发操作。由于缺少操作系统提供的保护机制，在多线程共享数据及通信时，就需要程序员做更多的工作以保证对共享数据段的操作是以预想的操作顺序进行的，并且要极力的避免**死锁**(deadlock)。

**多线程实现方式**

C++11 的标准类 std::thread 对线程进行了封装，它的声明放在头文件 thread 中，其中声明了线程类 thread, 线程标识符 id，以及名字空间 this\_thread

当 thread 类的功能不能满足我们的要求的时候（比如改变某个线程的优先级），可以通过 thread 类实例的 native\_handle() 返回值作为参数来调用相关的 pthread 函数达到目的。

thread::id 定义了在运行时操作系统内唯一能够标识该线程的标识符，同时其值还能指示所标识的线程的状态，其默认值 (thread::id()) 表示不存在可控的正在执行的线程（即空线程，比如，调用 thead() 生成的没有指定入口函数的线程类实例），当一个线程类实例的 get\_id() 等于默认值的时候，即 get\_id() == thread::id()，表示这个线程类实例处于下述状态之一：

尚未指定运行的任务

线程运行完毕

线程已经被转移 (move) 到另外一个线程类实例

线程已经被分离 (detached)

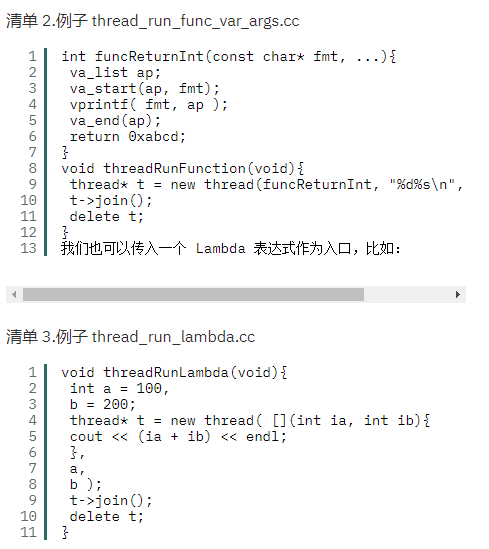
有时候我们需要在线程执行代码里面对当前调用者线程进行操作，针对这种情况，C++11 里面专门定义了一个名字空间 this\_thread，其中包括 ：

get\_id() 函数可用来获取当前调用者线程的 id 。get\_id() 函数实际上是通过调用 pthread\_self() 函数获得调用者线程的标识符

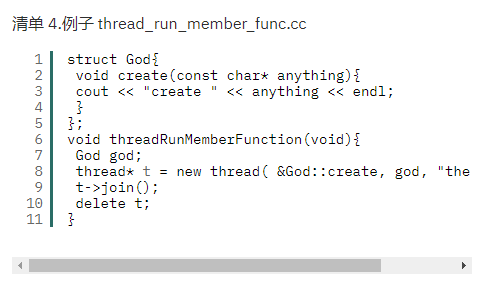
yield() 函数可以用来将调用者线程跳出运行状态，重新交给操作系统进行调度。而 yield() 函数则是通过调用操作系统 API sched\_yield() 进行调度切换。

sleep\_until 和 sleep\_for 函数则可以让调用者线程休眠若干时间。

使用 thread 类创建线程可以使用一个函数作为入口，也可以是其它的 Callable 对象，而且，可以给入口传入任意个数任意类型的参数：



一个类的成员函数也可以作为线程入口：

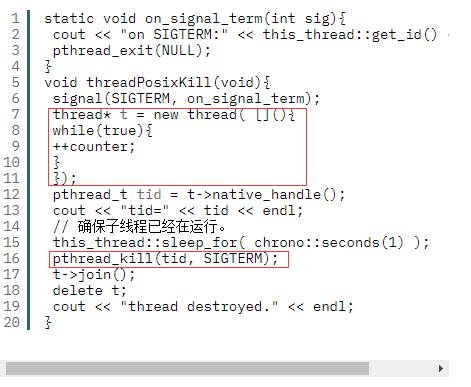


该如何处理这个线程的结束？

一种方式是等待这个线程结束，在一个合适的地方调用 thread 实例的 join() 方法，调用者线程将会一直等待着目标线程的结束，当目标线程结束之后调用者线程继续运行；

另一个方式是将这个线程分离，由其自己结束，通过调用 thread 实例的 detach() 方法将目标线程置于分离模式。一个线程的 join() 方法与 detach() 方法只能调用一次，不能在调用了 join() 之后又调用 detach()，也不能在调用 detach() 之后又调用 join()，在调用了 join() 或者 detach() 之后，该线程的 id 即被置为默认值（空线程），表示不能继续再对该线程作修改变化。如果没有调用 join() 或者 detach()，那么，在析构的时候，该线程实例将会调用 std::terminate()，这会导致整个进程退出，所以，如果没有特别需要，一般都建议在生成子线程后调用其 join() 方法等待其退出，这样子最起码知道这些子线程在什么时候已经确保结束。

在 C++11 里面没有提供 kill 掉某个线程的能力，只能被动地等待某个线程的自然结束，如果我们要主动停止某个线程的话，可以通过调用 Linux 操作系统提供的 pthread\_kill 函数给目标线程发送信号来实现，示例如下：



**线程间的通信-多生产者多消费者**

（1）wait()方法：

　　 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。

　　 线程调用wait()方法，释放它对锁的拥有权，然后等待另外的线程来通知它（通知的方式是notify()或者notifyAll()方法），这样它才能重新获得锁的拥有权和恢复执行。

　　 要确保调用wait()方法的时候拥有锁，即，wait()方法的调用必须放在synchronized方法或synchronized块中。

（2）notif()方法：

　notify()方法会唤醒一个等待当前对象的锁的线程。唤醒在此对象监视器上等待的单个线程。  
（3）notifAll()方法：

　 notifyAll（）方法会唤醒在此对象监视器上等待的所有线程。 　　  
（4）如果多个线程在等待，它们中的一个将会选择被唤醒。这种选择是随意的，和具体实现有关。（线程等待一个对象的锁是由于调用了wait方法中的一个）  
　notify()方法应该是被拥有对象的锁的线程所调用。  
（5）以上方法都定义在类：Object中  
1.因为，这些方法在操作同步中的线程的时候，都必须标示其所操作线程所持有的锁（被该锁的对象调用），而只有同一个对象监视器下（同一个锁上）的被等待线程，可以被持有该锁的线程唤醒，（无法唤醒不同锁上的线程）  
2.所以，等待和唤醒的必须是同一个对象的监视器①下（同一个锁上）的线程。

而锁可以是任意已近确定的对象， 能被任意对象调用的方法应当定义在 Object类中。  
注：①监视器（锁）：同一个对象的监视器下（同一个锁上）的线程，一次只能执行一个：就是拥有监视器所有权（持有锁）的那一个线程。

https://www.cnblogs.com/Wenxu/p/7979023.html

<https://www.cnblogs.com/haippy/p/3252092.html>

程序需要维护两个计数器，分别是生产者已生产产品的数目和消费者已取走产品的数目。另外也需要保护产品库在多个生产者和多个消费者互斥地访问。

**线程是怎么划分的：用户级/核心级线程和混合方式**

用户级线程:是在用户空间建了一个线程库，这个线程库里提供了一系列的针对线程的操作。这些线程的管理是通过一个Run-time System运行时系统来管理的。它完成的就是这些线程的创建、还有线程数据结构的一些管理工作。Run-time System管理这些线程的数据结构、线程表。这是用户级线程的一个实现。那么对于内核而言，由于你的线程的实现是在用户空间，所以操作系统内核并不知道线程的存在，也就是说它的管理还是以进程为单位来管理，它没有感知线程的存在。因此线程的切换，从一个线程换到另外一个线程不需要操作系统内核的干预，也不需要进入内核来做这件事情，因而比较快。

  核心级线程：内核管理所有的线程。通过API的接口向用户提供一些API的函数，由用户可以创建线程。所以内核既维护了进程的数据结构，也维护了进程里头的各个线程的数据结构。内核里头既管了线程表，也管了进程表。线程的切换就需要内核干预了，因此要进入内核来完成切换的过程。调度也是以线程为单位来进行的。实现了核心级线程机制的典型的操作系统就是Windows。

混合模型：线程的创建呢是在用户空间用线程库来完成的。但是内核儿也要管理线程，也就是说调度是由内核来完成的。那么这个采用这种混合模型实现线程机制的呢是Solaris操作系统。实际上是用户线程通过了一个多路复用来复用多个内核级线程，也就是核外的用户空间的线程通过一个机制和核内的一个内核线程对应起来。那么调度内核这个线程上CPU其实就是调度这个核外的这个线程上CPU。

**-------------------------------------**

**进程**（process）

进程是 资源分配和调度 的基本单位

**第一**，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域，即程序（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

**第二**，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时，它才能成为一个活动的实体，我们称其为进程。

**如何开启进程fork（）**

fork通过复制父进程来创建一个新的进程。这个新创建的进程称为调用fork()函数的子进程，这个调用fork（）的进程称为子进程的父进程。子进程除了以下几点之外就是父进程的一个复制品。   
\*子进程有其唯一的PID；   
\*子进程的父进程PID（PPID）和父进程的PID是相同的；   
\*子进程不继承父进程的内存块；   
\*子进程的资源使用计数器和CPU时间计数器都将被置为空；   
\*子进程挂起信号量的数目初始化为0；   
\*子进程并不继承父进程的信号量调节器；   
\*子进程并不继承父进程的记录锁；   
\*子进程不继承父进程 的时间计数器；   
\*子进程不继承父进程的异步I/O操作和异步I/O操作内容；

**fork创建出来的子进程从哪里开始执行**

加入一个程序有partA+fork+partB组成。那么当父进程执行到fork的同时会创建一个子进程，然后父进程接着执行直到结束。在子进程当中会从fork开始执行，直至结束。   
所涉及的中断调用   
如果Fork成功则在父进程会返回新建立的子进程代码（PID），而在新建立的子进程中则返回0。如果fork失败则直接返回-1。   
getpid()   
取得目前进程的识别码，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。   
getppid（）   
取得目前进程的父进程识别码。

**fork出来的子进程和父进程到底谁先执行**

进程是用于解决多任务的，进程之间的执行顺序是不确定的。   
fork出来的子进程是一个就绪状态，这是他在等待队列中，只要它已抢占到CPU就可以执行。   
如果有哪个操作系统进程之间的执行是有先后顺序的，我想这肯定是一个bug。因为如果有先后顺序，那么这个操作系统该怎么做到程序的并发执行？   
所以我们最终的结论就是：

**子进程和父进程谁先执行决定于谁先从就绪状态抢占到了CPU，并没有确定的先后顺序。**

**Fork/join和线程池**

不使用 Fork-Join 框架时，使用普通的线程池是怎么实现的。

* 我们往一个线程池提交了一个大任务，规定好任务切割的阀值。
* 由池中线程（假设是线程A）执行大任务，发现大任务的大小大于阀值，于是切割成两个子任务，并调用 submit() 提交到线程池，得到返回的子任务的 Future。
* 线程A就调用 返回的 Future 的 get() 方法阻塞等待子任务的执行结果。
* 池中的其他线程（除线程A外，线程A被阻塞）执行两个子任务，然后判断子任务的大小有没有超过阀值，如果超过，则按照步骤2继续切割，否则，才计算并返回结果。

嘿，好像一切都很美好。真的吗？别忘了， **每一个切割任务的线程（如线程A）都被阻塞了，直到其子任务完成，才能继续往下运行** 。如果任务太大了，需要切割多次，那么就会有多个线程被阻塞，性能将会急速下降。更糟糕的是，如果你的线程池的线程数量是有上限的，极可能会造成池中所有线程被阻塞，线程池无法执行任务。

Fork-Join 框架使用了 “工作窃取（work-stealing）”算法。工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。

Fork-Join 框架中的工作窃取算法的**优点**可以总结为以下两点：

* 线程是不会因为等待某个子任务的完成或者没有内部任务要执行而被阻塞等待、挂起，而是会扫描所有的队列，窃取任务，直到所有队列都为空时，才会被挂起。 就如上面所说的。
* Fork-Join 框架在多CPU的环境下，能提供很好的并行性能。在使用普通线程池的情况下，当CPU不再是性能瓶颈时，能并行地运行多个线程，然而却因为要互斥访问一个任务队列而导致性能提高不上去。而 Fork-Join 框架为每个线程为维护着一个内部任务队列，以及一个全局的任务队列，而且任务队列都是双向队列，可从首尾两端来获取任务，极大地减少了竞争的可能性，提高并行的性能。

**进程控制块pcb**

为了描述和管理进程的运行，在OS的核心专门定义了一个数据结构。其作为进程实体的一部分，记录了OS所需的，用于描述进程当前的情况以及进程运行的消息。

作用：

1：作为独立运行基本单位的标志。系统是通过PCB感知进程的存在。

2:能实现间断性运行的方式。多到程序运行的情况下程序是走走停停，间断运行的，当程序移交CPU的时候必须保留现场，调用时要恢复现场，PCB中保留了中断程序的现场信息，

3：提供进程调度所需的信息；PCB中提供了进程的状态信息，还有其它信息任如（进程的优先级，进程的等待时间，以及和执行时间）

4：实现与其他进程的同步与信息：进程同步机制是用于实现各进程间的协调运行，采用信号量机制时，要求每个进程都设置相应的同于同步的信号量。在PCB中还具有实现进程通信区域或通信队列指针等。

进程控制块中的信息

1：进程标识符（PID）：用于唯一标识一个进程一个进程通常有两种标识符1：外部与内部标识符。

2：处理机状态：处理机的状态信息，也称为处理机的上下文，当进程切换时，处理机的状态信息必须保存在相应的PCB中，以便在该进程重新执行时从断点处开始。

3：进程调度信息：在OS进行进程调度的时候，必须了解进程的状态以及进程的调度信息。如（进程的状态，进程的优先级，其他消息（进程调度算法），事件（引起阻塞原因））

4：进程的控制信息：包括1：程序和数据地址，2：进程同步和通信机制，3：资源抢占清单4：连接指针。

**多进程并发**

多进程并发是将一个应用程序划分为多个独立的进程（每个进程只有一个线程），这些独立的进程间可以互相通信，共同完成任务。由于操作系统对进程提供了大量的保护机制，以避免一个进程修改了另一个进程的数据，使用多进程比多线程更容易写出**安全**的代码。由于多个进程并发完成同一个任务时，不可避免的是：操作同一个数据和进程间的相互通信，上述的两个缺点也就决定了多进程的并发不是一个好的选择。

缺点：

（1）**进程间的通信**，无论是使用信号、套接字，还是文件、管道等方式，其使用要么比较复杂，要么就是速度较慢或者两者兼而有之。

（2）运行多个进程的开销很大，操作系统要分配很多的资源来对这些进程进行管理。

**进程间通信方式IPC**

IPC的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket、Streams等。其中 Socket和Streams支持不同主机上的两个进程IPC。

1.匿名管道（pipe亲缘关系的进程通信）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2.命名管道FIFO（mkfifo/mknod）：命名也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

3.消息队列MessageQueue：是基于消息的、用无亲缘关系的进程间通信，主要函数：msgget、msgsend、msgrecv、msgctl。消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。  
4. 共享存储SharedMemory：是进程间通信速度最快的，所以用经常是集合信号量或互斥锁来实现同步，shmget、shmat、shmdt、shmctl。共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。  
5. 信号量Semaphore：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。  
6. **套接字**Socket：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。  
7. 信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

**进程的几种状态run/sleep/D/T/Z**

（1）run（运行状态）：正在运行的进程或在等待队列中对待的进程，等待的进程只要一得到cpu就可以运行

（2）Sleep（可中断休眠状态）：相当于阻塞或在等待的状态

（3）D（不可中断休眠状态）：在磁盘上的进程

（4）T（停止状态）：这中状态无法直观的看见，因为是进程停止后就释放了资源，所以不会留在linux中

（5）Z（僵尸状态）：子进程先与父进程结束，但父进程没有调用wait或waitpid来回收子进程的资源，所以子进程就成了僵尸进程，如果父进程结束后任然没有回收子进程的资源，那么1号进程将回收

**进程状态**:**就绪/运行/阻塞**

就绪状态其实就是获取了出cpu外的所有资源，只要处理器分配资源就可以马上执行。就绪状态有排队序列什么的，排队原则不再赘述。

运行态就是获得了处理器分配的资源，程序开始执行。

阻塞态，当程序条件不够时候，需要等待条件满足时候才能执行，如等待i/o操作时候，此刻的状态就叫阻塞态。

进程状态转换

①就绪→执行：调度

②执行→等待：等待某个事件发生而睡眠

③等待→就绪：因等待的事件发生而唤醒

④执行→就绪：时间片用完或出现高优先

**--------------------------------------**

**Linux系统启动流程**

BIOS自检-->从BIOS中读取启动顺序-->读取MBR中的bootloader-->加载内核-->读取伪根-->读取根文件中的init



**BIOS**

BIOS(Basic Input/Output System)，基本输入输出系统，该系统存储于主板的ROM芯片上，计算机在开机时，会最先读取该系统，然后会有一个加电自检过程，这个过程其实就是**检查CPU和内存，计算机最基本的组成单元(控制器、运算器和存储器)，还会检查其他硬件**，若没有异常就开始加载BIOS程序到内存当中。

详细的BIOS功能，这边就不说了，BIOS主要的一个功能就是存储了磁盘的启动顺序，BIOS会按照启动顺序去查找第一个磁盘头的MBR信息，并加载和执行MBR中的Bootloader程序，若第一个磁盘不存在MBR，则会继续查找第二个磁盘(PS：启动顺序可以在BIOS的界面中进行设置)，一旦BootLoader程序被检测并加载内存中，BIOS就将控制权交接给了BootLoader程序。

**MBR**

　　MBR(Master Boot Record)，主引导记录，MBR存储于磁盘的头部，大小为512bytes，其中，446bytes用于存储BootLoader程序，64bytes用于存储分区表信息，最后2bytes用于MBR的有效性检查。

**GRUB**

　　GRUB(Grand Unified Bootloader)，多系统启动程序，其执行过程可分为三个步骤：

　　　　Stage1：这个其实就是MBR，它的主要工作就是查找并加载第二段Bootloader程序(stage2)，但系统在没启动时，MBR根本找不到文件系统，也就找不到stage2所存放的位置，因此，就有了stage1\_5

　　　　Stage1\_5：该步骤就是为了识别文件系统

　　　　Stage2：GRUB程序会根据/boot/grub/grub.conf文件**查找Kernel**的信息，然后开始**加载**Kernel程序，当Kernel程序被检测并在加载到内存中，GRUB就将控制权交接给了Kernel程序。

　　　　PS：实际上这个步骤/boot还没被挂载，GRUB直接识别grub所在磁盘的文件系统，所以实际上应该是/grub/grub.conf文件

**Kernel**

　　Kernel，内核，Kernel是Linux系统最主要的程序，实际上，Kernel的文件很小，只保留了最基本的模块，并以压缩的文件形式存储在硬盘中，当GRUB将Kernel读进内存，内存开始解压缩内核文件。讲内核启动，应该先讲下initrd这个文件，

　　initrd(Initial RAM Disk)，它在stage2这个步骤就被拷贝到了内存中，这个文件是在安装系统时产生的，是一个临时的根文件系统(rootfs)。因为Kernel为了精简，只保留了最基本的模块，因此，Kernel上并没有各种硬件的驱动程序，也就无法识rootfs所在的设备，故产生了initrd这个文件，该文件装载了必要的驱动模块，当Kernel启动时，可以从initrd文件中装载驱动模块，直到挂载真正的rootfs，然后将initrd从内存中移除。

**Kernel会以只读方式挂载根文件系统**，当根文件系统被挂载后，**开始装载第一个进程(用户空间的进程)，执行/sbin/init，之后就将控制权交接给了init程序。**

**Init**

　　init，初始化，顾名思义，该程序就是进行OS初始化操作，实际上是根据/etc/inittab(定义了系统默认运行级别)设定的动作进行脚本的执行，第一个被执行的脚本为/etc/rc.d/rc.sysinit，这个是真正的OS初始化脚本，简单讲下这个脚本的任务(可以去看看实际脚本，看看都做了什么)：

　　1、激活udev和selinux；2、根据/etc/sysctl.conf文件，来设定内核参数；3、设定系统时钟；4、装载硬盘映射；5、启用交换分区；6、设置主机名；7、根文件系统检测，并以读写方式重新挂载根文件系统；8、激活RAID和LVM设备；9、启用磁盘配额；10、根据/etc/fstab，检查并挂载其他文件系统；11、清理过期的锁和PID文件

　　执行完后，根据配置的启动级别，执行对应目录底下的脚本，最后执行/etc/rc.d/rc.local这个脚本，至此，系统启动完成。

**Runlevel**

　　runlevel，运行级别，不同的级别会启动的服务不一样，**init会根据定义的级别去执行相应目录下的脚本**，Linux的启动级别分为以下几种

　　0：关机模式

　　1：单一用户模式(直接以管理员身份进入)

　　2：多用户模式（无网络）

　　3：多用户模式（命令行）

　　4：保留

　　5：多用户模式（图形界面）

　　6：重启

　　在不同的运行级别下，/etc/rc.d/rc这个脚本会分别执行不同目录下的脚本

* Run level 0 – /etc/rc.d/rc0.d/
* Run level 1 – /etc/rc.d/rc1.d/
* Run level 2 – /etc/rc.d/rc2.d/
* Run level 3 – /etc/rc.d/rc3.d/
* Run level 4 – /etc/rc.d/rc4.d/
* Run level 5 – /etc/rc.d/rc5.d/
* Run level 6 – /etc/rc.d/rc6.d/

　　这些目录下的脚本只有K\*和S\*开头的文件，K开头的文件为开机需要执行关闭的服务，S开头的文件为开机需要执行开启的服务。

**进程和线程的区别**

资源分配方面：

（1）进程是资源的分配和调度的一个独立单元，而线程是CPU调度的基本单元。真正在处理机上运行的是线程。

（2）同一个进程中可以包括多个线程，至少包括一个线程。线程共享整个进程的资源（寄存器、堆栈、上下文），同一进程中的多个线程共享代码段(代码和常量)，数据段(全局变量和静态变量)，扩展段(堆存储)。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。

（3）进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以**多进程**的程序要比**多线程**的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

创建与销毁方面：

（4）进程的创建调用fork或者vfork，而线程的创建调用pthread\_create，进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束

（5）线程是轻量级的进程，它的创建和销毁所需要的时间比进程小很多，所有操作系统中的执行功能都是创建线程去完成的

（6）线程有自己的私有属性TCB，线程id，寄存器、硬件上下文，而进程也有自己的私有属性进程控制块**PCB**，这些私有属性是不被共享的，用来标示一个进程或一个线程的标志

执行过程方面：

（5）线程中执行时一般都要进行**同步和互斥**，因为他们共享同一进程的所有资源。不同进程的线程间要利用**消息通信**的办法实现同步。

（7）线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP(多核处理机)机器上运行，而进程则可以跨机器迁移。

（8）执行过程中：每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

（9）从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。

**并行和并发的区别**

https://www.cnblogs.com/wangguchangqing/p/6134635.html

并行是指两个或多个独立的操作同时进行。并发是在一个时间段内执行多个操作。在单核时代，多个线程是并发的，在一个时间段内轮流执行；在多核时代，多个线程可以实现真正的并行，在多核上真正独立的并行执行。例如现在常见的4核4线程可以并行4个线程；4核8线程则使用了超线程技术，把一个物理核模拟为2个逻辑核心，可以并行8个线程。

**死锁**

相互等待资源而产生的一种僵持状态，如果没有外力的干预将一直持续这个状态

产生的原因：系统资源不足、相互竞争资源、请求资源顺序不当

死锁的必要条件：互斥、不可抢占、循环等待、请求与保持

如何处理死锁：因为互斥是不可改变的，所以只能破坏其他三个条件中的一个来解除死锁，方法：剥夺资源、杀死其中一个线程

造成死锁必须达成的4个条件（原因）：

1. 互斥条件：一个资源每次只能被一个线程使用。
2. 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：线程已获得的资源，在未使用完之前，不能强行剥夺。
4. 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

**多线程写日志，涉及到单例模式，异步写**

**中断**

中断是CPU处理外部突发事件的一个重要技术。它能使CPU在运行过程中对外部事件发出的中断请求及时地进行处理，处理完成后又立即返回断点，继续进行CPU原来的工作。引起中断的原因或者说发出中断请求的来源叫做中断源。 根据中断源的不同，可以把中断分为硬件中断和软件中断两大类，而硬件中断又可以分为外部中断和内部中断两类

**硬中断软中断**

(1) 硬中断

由与系统相连的外设(比如网卡、硬盘)自动产生的。主要是用来通知操作系统系统外设状态的变化。比如当网卡收到数据包的时候，就会发出一个中断。我们通常所说的中断指的是硬中断(hardirq)。

(2) 软中断

为了满足实时系统的要求，中断处理应该是越快越好。linux为了实现这个特点，当中断发生的时候，硬中断处理那些短时间就可以完成的工作，而将那些处理事件比较长的工作，放到中断之后来完成，也就是软中断(softirq)来完成。

**Windows内存管理方式：段/页/段页**

1、段存储：

**逻辑空间**分为若干个段，每个段定义了一组有完整**逻辑意义**的信息（如主程序Main（））

**内存空间**为每个段分配一个连续的分区

　　段的长度由相应的逻辑信息组的长度决定，因而各段长度不等，引入分段存储管理方式的目的主要是为了满足用户（程序员）在编程和使用上多方面的要求。

　　将程序分页时，页的大小是固定的，只根据页面大小大小死生生的将程序切割开；而分段时比较灵活，只有一段程序有了完整的意义才将这一段切割开。（例如将一个人每隔50厘米切割一段，即为分页；而将一个人分割为头部、身体、腿部（有完整逻辑意义）三段，即为分段）

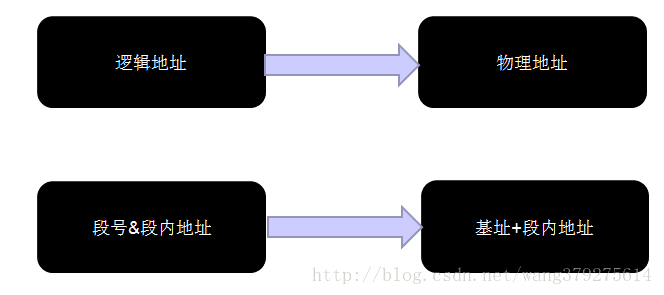
   分两部分：段号、位移量（段内地址）



段内地址的位数可以决定段的大小

逻辑地址=段号&段内地址（&号是连接符号，是将段号作为逻辑地址的最高位）

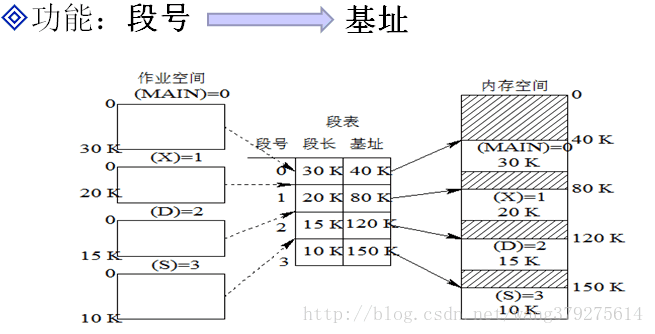
如下图所示：（物理地址=基址+段内地址）（**注意为+号，而不是&号**）



由上图可知若想求物理地址，只需求出**基址**即可：

如何求**基址**呢？段表来帮你

段表：



　　求基址的过程与页式存储中求块号的过程原理相同，这里需要注意的是，物理地址是基址**+**段内地址，而不是基址**&**段内地址，由逻辑地址得到段号、段内地址，再根据段号和段表求出基址，再由基址+段内地址即可得物理地址。

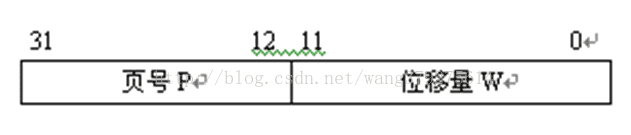
2、页存储

**逻辑空间**等分为页；并从0开始编号

**内存空间**等分为块，与页面大小相同；从0开始编号

分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。

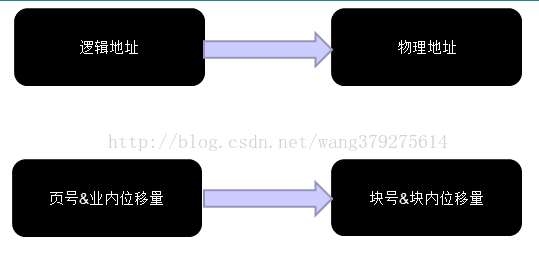
       分两部分：页号、位移量（业内地址）

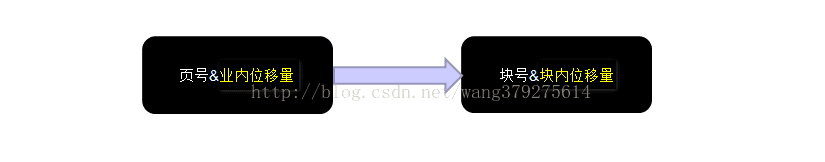


       业内地址的位数可以决定页的大小（如上图每页大小为4K）。

　　逻辑地址=页号&位移量（&号是连接符号，是将页号作为逻辑地址的最高位）

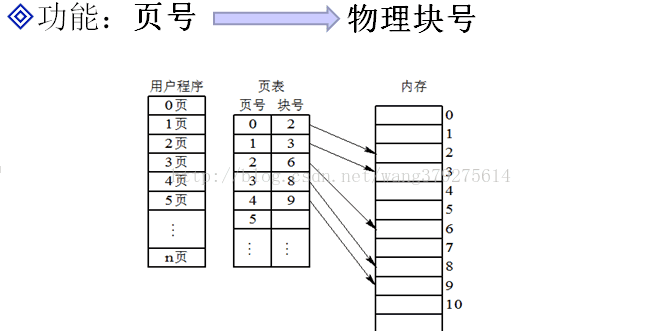
　　如下图所示：（物理地址=块号&块内地址）



　　因为块的大小=页的大小，所以块内位移量=页内位移量，所以只需求出块号即可：

如何求块号呢？页表来帮你

页表：



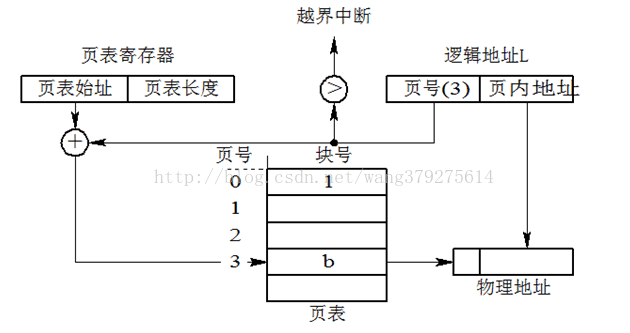
给定一个逻辑地址和页面大小，如何计算物理地址？

1)       根据页面大小可计算出页内地址的位数

2)       页内地址位数结合逻辑地址计算出页内地址（即，块内地址）和页号

3)       页号结合页表，即可得出块号

4)       块号&块内地址即可得出物理地址



请看上图，给出逻辑地址的页号和页内地址，开始进行地址变换：

1)       在被调进程的PCB中取出页表始址和页表大小，装入页表寄存器

2)       页号与页表寄存器的页表长度比较，若页号大于等于页表长度，发生地址越界中断，停止调用，否则继续

3)       由页号结合页表始址求出块号

4)       块号&页内地址，即得物理地址

3、段页存储

**什么是虚拟内存**

将进程部分装入内存中，从而能实现一个很大的程序能在一个比它小的内存中运行，它的主要实现是靠程序的换进换出来实现的，因为内存中0~3G是用户使用，3~4G才是内存使用，通过映射来实现，来进行逻辑地址到物理地址的映射。

**虚拟地址、逻辑地址、线性地址、物理地址的区别**

分段机制把一个逻辑地址转换为线性地址；分页机制把一个线性地址转换为物理地址。

（1）虚拟地址：虚拟内存映射出来的地址

（2）逻辑地址：程序的段加偏移量形成的，C/C++程序中取地址求出来的地址就是逻辑地址

（3）线性地址：是逻辑地址到物理地址的中间层，只有启动分页机制的时候才有线性地址，如果没有分页机制，那么线性地址就是物理地址

（4）物理地址：是内存中实实在在存在的硬件地址，

逻辑地址（启动分段）--》线性地址（启动分页）--》物理地址

**OS调度算法**

https://www.iteblog.com/archives/251.html

[一、先来先服务和短作业(进程)优先调度算法](https://www.iteblog.com/archives/251.html#i)

[二、高优先权优先调度算法](https://www.iteblog.com/archives/251.html#i-2)

[1．优先权调度算法的类型](https://www.iteblog.com/archives/251.html#1-2)

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

1) 非抢占式优先权算法

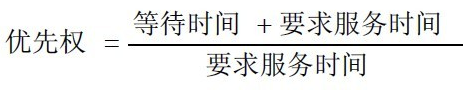
在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

[2．高响应比优先调度算法](https://www.iteblog.com/archives/251.html#2-2)

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：



(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

[三、基于时间片的轮转调度算法](https://www.iteblog.com/archives/251.html#i-3)

[时间片轮转法](https://www.iteblog.com/archives/251.html#1-3)

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度

时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

[多级反馈队列调度算法](https://www.iteblog.com/archives/251.html#2-3)

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

s

如果面对几百上千个并发，这样的情况将导致怎样的后果。

乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

一个典型的倚赖数据库的悲观锁调用：select \* from account where name=”Erica” for update这条 sql 语句锁定了 account 表中所有符合检索条件（ name=”Erica” ）的记录。本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。Hibernate 的悲观锁，也是基于数据库的锁机制实现。注意，只有在查询开始之前（也就是 Hiberate 生成 SQL 之前）设定加锁，才会真正通过数据库的锁机制进行加锁处理，否则，数据已经通过不包含 for update子句的 Select SQL 加载进来，所谓数据库加锁也就无从谈起。