操作系统

## 内存分配

CPU调度算法

先来先服务(FCFS)调度**算法**

短作业(进程)优先调度算法SJ(P)F

非抢占式优先权算法

抢占式优先权调度算法

**高相应比算法HRN：**响应比=(等待时间+要求服务时间)/要求服务时间;

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。

**时间片轮转调度RR：**按到达的先后对进程放入队列中，然后给队首进程分配CPU时间片，时间片用完之后计时器发出中断，暂停当前进程并将其放到队列尾部，循环；

**多级反馈队列调度算法：**目前公认较好的调度算法；设置多个就绪队列并为每个队列设置不同的优先级，第一个队列优先级最高，其余依次递减。优先级越高的队列分配的时间片越短，进程到达之后按FCFS放入第一个队列，如果调度执行后没有完成，那么放到第二个队列尾部等待调度，如果第二次调度仍然没有完成，放入第三队列尾部...。只有当前一个队列为空的时候才会去调度下一个队列的进程。

- FIFO算法
  先入先出，即淘汰最早调入的页面。

- OPT(MIN)算法
  选未来最远将使用的页淘汰，是一种最优的方案，可以证明缺页数最小。
  可惜，MIN需要知道将来发生的事，只能在理论中存在，实际不可应用。

- LRU(Least-Recently-Used)算法
  用过去的历史预测将来，选最近最长时间没有使用的页淘汰(也称最近最少使用)。
  LRU准确实现：计数器法，页码栈法。
  由于代价较高，通常不使用准确实现，而是采用近似实现，例如Clock算法。

**首次适应(First Fit)算法：**空闲分区以地址递增的次序链接。分配内存时顺序查找，找到大小能满足要求的第一个空闲分区。

**最佳适应(Best Fit)算法：**空闲分区按容量递增形成分区链，找到第一个能满足要求的空闲分区。

**最坏适应(Worst Fit)算法：**又称最大适应(Largest Fit)算法，空闲分区以容量递减的次序链接。找到第一个能满足要求的空闲分区，也就是挑选出最大的分区。

## inux 共享内存实现

说起共享内存，一般来说会让人想起下面一些方法:

1、多线程。线程之间的内存都是共享的。更确切的说，属于同一进程的线程使用的是同一个地址空间，而不是在不同地址空间之间进行内存共享;

2、父子进程间的内存共享。父进程以MAP_SHARED|MAP_ANONYMOUS选项mmap一块匿名内存，fork之后，其子孙进程之间就能共享这块内存。这种共享内存由于受到进程父子关系的限制，一般较少使用;

3、mmap文件。多个进程mmap到同一个文件，实际上就是大家在共享文件page cache中的内存。不过文件牵涉到磁盘的读写，用来做共享内存显然十分笨重，所以就有了不跟磁盘扯上关系的内存文件，也就是我们这里要讨论的tmpfs和shmem；

## 进程间通信方式

管道 ：半双工 单向 只能父子进程

FIFO ：去掉父子进程限制

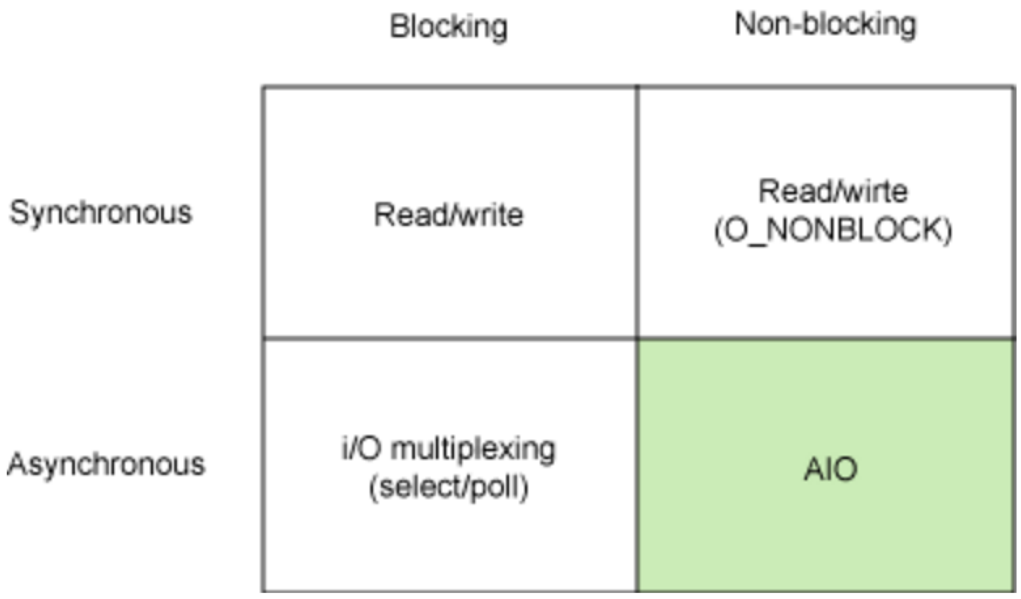
消息队列：独立于读写进程之外，避免了 FIFO 的同步阻塞问题，可以有选择性接收信息

信号量：一个计数器，用于为多个进程提供对共享数据对象的访问。

共享内存 ：允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。需要使用信号量用来同步对共享存储的访问。多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。

套接字：可用于不同机器间的进程通信

## 五种io模型




# 同步

### 同步阻塞 IO（blocking IO）

同步阻塞 IO 模型是最常用的一个模型，也是最简单的模型。在linux中，默认情况下所有的socket都是blocking
kernel就开始了IO的
第一个阶段：准备数据（对于网络IO来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的UDP包。这个时候kernel就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。
第二个阶段：当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。
**优点：**

1. 能够及时返回数据，无延迟；

2. 对内核开发者来说这是省事了；

**进程阻塞挂起不消耗CPU资源，及时响应每个操作**；
实现难度低、开发应用较容易；
适用并发量小的网络应用开发；
**不适用并发量大的应用**：因为一个请求IO会阻塞进程，所以，得为每请求分配一个处理进程（线程）以及时响应，系统开销大。

**缺点：**

1. 对用户来说处于等待就要付出性能的代价了;

## 同步非阻塞 IO（nonblocking IO）

同步非阻塞就是 “每隔一会儿瞄一眼进度条” 的轮询（polling）方式。在这种模型中，设备是以非阻塞的形式打开的。这意味着 IO 操作不会立即完成，read 操作可能会返回一个错误代码，说明这个命令不能立即满足（EAGAIN 或 EWOULDBLOCK）。
也就是说非阻塞的recvform系统调用调用之后，进程并没有被阻塞，内核马上返回给进程，如果数据还没准备好，此时会返回一个error。进程在返回之后，可以干点别的事情，然后再发起recvform系统调用。重复上面的过程，循环往复的进行recvform系统调用。这个过程通常被称之为轮询。轮询检查内核数据，直到数据准备好，再拷贝数据到进程，进行数据处理。**需要注意，拷贝数据整个过程，进程仍然是属于阻塞的状态。**

### 同步非阻塞方式相比同步阻塞方式：

优点：能够在等待任务完成的时间里干其他活了（包括提交其他任务，也就是 “后台” 可以有多个任务在同时执行）。
缺点：任务完成的响应延迟增大了，因为每过一段时间才去轮询一次read操作，而任务可能在两次轮询之间的任意时间完成。这会导致整体数据吞吐量的降低。

## IO多路复用（IO多路复用在阻塞到select阶段时，用户进程是主动等待并调用select函数获取数据就绪状态消息，并且其进程状态为阻塞。所以，把IO多路复用归为同步阻塞模式。）

如果轮询不是进程的用户态，而是有人帮忙就好了。那么这就是所谓的 “IO 多路复用”。UNIX/Linux 下的 select、poll、epoll 就是干这个的（epoll 比 poll、select 效率高，做的事情是一样的）。
IO多路复用有两个特别的系统调用select、poll、epoll函数。select调用是内核级别的，select轮询相对非阻塞的轮询的区别在于---前者可以等待多个socket，能实现同时对多个IO端口进行监听，当其中任何一个socket的数据准好了，就能返回进行可读，然后进程再进行recvform系统调用，将数据由内核拷贝到用户进程，当然这个过程是阻塞的。

select或poll调用之后，会阻塞进程，与blocking IO阻塞不同在于，此时的select不是等到socket数据全部到达再处理, 而是有了一部分数据就会调用用户进程来处理。如何知道有一部分数据到达了呢? 监视的事情交给了内核，内核负责数据到达的处理。也可以理解为"非阻塞"吧。
I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时（注意不是全部数据可读或可写），才真正调用I/O操作函数。

IO multiplexing就是我们说的select，poll，epoll，有些地方也称这种IO方式为event driven IO。select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select，poll，epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。
当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

性能对比：
因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。
所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。（select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）
在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。所以**IO多路复用是阻塞在select，epoll这样的系统调用之上，而没有阻塞在真正的I/O系统调用如recvfrom之上。**

优点：
I/O多路复用技术通过把多个I/O的阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降底了系统的维护工作量，节省了系统资源

## 从整个IO过程来看，他们都是顺序执行的，因此可以归为同步模型(synchronous)。都是进程主动等待且向内核检查状态。【此句很重要！！！】

高并发的程序一般使用同步非阻塞方式而非多线程 + 同步阻塞方式。要理解这一点，首先要扯到并发和并行的区别。比如去某部门办事需要依次去几个窗口，办事大厅里的人数就是并发数，而窗口个数就是并行度。也就是说并发数是指同时进行的任务数（如同时服务的 HTTP 请求），而并行数是可以同时工作的物理资源数量（如 CPU 核数）。通过合理调度任务的不同阶段，并发数可以远远大于并行度，这就是区区几个 CPU 可以支持上万个用户并发请求的奥秘。在这种高并发的情况下，为每个任务（用户请求）创建一个进程或线程的开销非常大。而同步非阻塞方式可以把多个 IO 请求丢到后台去，这就可以在一个进程里服务大量的并发 IO 请求。

# 异步

**信号驱动式IO（signal-driven IO）**

信号驱动式I/O：首先我们允许Socket进行信号驱动IO,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据

**异步非阻塞 IO（asynchronous IO）**

相对于同步IO，异步IO不是顺序执行。用户进程进行aio_read系统调用之后，无论内核数据是否准备好，都会直接返回给用户进程，然后用户态进程可以去做别的事情。等到socket数据准备好了，内核直接复制数据给进程，然后从内核向进程发送通知。IO两个阶段，进程都是非阻塞的。

异步IO与信号驱动式IO的主要区别是：信号驱动式IO是由内核通知我们何时启动一个IO操作，而异步IO是由内核通知我们IO操作何时完成。

## 同步与异步区别

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes. —— 同步IO操作导致进程阻塞，直到IO操作完成。
- An asynchronous I/O operation does not cause the requesting process to be blocked. —— 异步IO操作不导致进程阻塞。

1. **支持一个进程所能打开的最大连接数**

| | |
|---|---|
| select | 单个进程所能打开的最大连接数有FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32*32，同理64位机器上FD_SETSIZE为32*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接 |

2. **FD剧增后带来的IO效率问题**

| | |
|---|---|
| select | 因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。 |
| poll | 同上 |
| epoll | 因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。 |

3. **消息传递方式**

| | |
|---|---|
| select | 内核需要将消息传递到用户空间，都需要内核拷贝动作 |
| poll | 同上 |
| epoll | epoll通过内核和用户空间共享一块内存来实现的。 |

**总结：**

**select的几大缺点：**

**（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大**

**（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大**

**（3）select支持的文件描述符数量太小了，默认是1024**

**2 poll实现**

poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd_set结构，其他的都差不多。

epoll优点：

**基本原理：**

epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就绪状态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll_wait便可以收到通知。

epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

**epoll的优点：**

1. 没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）。

2. 效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

3. 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

**综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点：**

1. 表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

2. select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善。