# 一、linux基础

## Q：linux常用命令，查看网络状态用什么命令？

A： **cd：**切换当前目录

eg：cd ../path （切换到上层目录中的path目录中，“..”表示上一层目录）

**ls：**查看文件与目录

eg：ls -l （以长数据串的形式列出当前目录下的数据文件和目录）

**grep：**分析一行的信息，提取需要的信息，常和管道命令一起使用

-o只输出文件中匹配到的行

-E使用正则表达式查找

-n输出包含匹配字符串的行数；

eg：ls -l | grep -i file（把ls -l的输出中包含字母file（不区分大小写）的内容输出）

**find：**查找指定目录下满足某些特点的文件

eg：find /home/ljianhui -user ljianhui（在目录/home/ljianhui中找出所有者为ljianhui的文件）

**cp：**复制文件

eg：cp -a file1 file2 （连同文件的所有特性把文件file1复制成文件file2）

**mv：**移动文件、目录、更名

eg：mv file1 file2 file3 dir # 把文件file1、file2、file3移动到目录dir中

**rm：**删除文件、目录

eg：rm -fr dir  强制删除目录dir中的所有文件（-f 强制；-r 递归）

**ps：**将某个时间点的进程运行情况选取下来并输出

eg：ps aux  查看系统所有的进程数据

ps ax  查看不与terminal有关的所有进程

ps -lA # 查看系统所有的进程数据

ps axjf # 查看连同一部分进程树状态

**kill：**向进程发送信号，当发送的是终止运行信号时，终止进程运行，1表示重启被终止的程序、9表示强制中断一个程序的运行，可能会有.filename.swp文件残留、15表示以正常的方式来终止该程序，正常退出就是按应用程序自己的退出流程完成退出，这样就可以清理并释放资源。比如 vim 程序，如果是正常的退出，就会删除掉临时文件\*.swp。15是kill的默认参数，所以可以省略。

eg：kill -15 pid （让进程号为pid的进程正常退出）

**killall：**功能类似kill，但是提供向进程名发信号

eg：killall -SIGHUP syslogd # 重新启动syslogd

**mkdir：**创建目录

eg：mkdir -p-m 750 bin/os\_1 (在当前目录中建立bin和bin下的os\_1目录，权限设置为文件主可读、写、执行，同组用户可读和执行，其他用户无权访问)

**tar：**对文件进行压缩和解压

eg：压缩：tar -jcv -f filename.tar.bz2 要被处理的文件或目录名称

查询：tar -jtv -f filename.tar.bz2

解压：tar -jxv -f filename.tar.bz2 -C 欲解压缩的目录

**cat：**查看文本文件内容，通常与more和less一起使用，从而可以一页页查看数据

eg：cat text | less（查看text文件中的内容）

**chgrp：**改变文件所属用户组

eg：chgrp users -R ./dir（递归地把dir目录下中的所有文件和子目录下所有文件的用户组修改为users）

**chown：**改变文件的所有者，与chgrp命令使用方法相同

eg：chown -R liu /usr/meng //将/usr/meng下的文件递归设置owner为liu

**chmod：**改变文件权限，chmod还可以使用u（user）、g（group）、（other）、a（all）和+（加入）、-（删除）、=（设置）跟rwx搭配来对文件的权限进行更改

eg：chmod g+w file （向file的文件权限中加入用户组可写权限）

**vi：**文本编辑，它接一个或多个文件名作为参数，如果文件存在就打开，如果文件不存在就以该文件名创建一个文件。

eg：vi test.c //打开并编辑test.c文件，如果文件不存在，创建一个。

**gcc：**将C语言的源程序文件，编译成可执行程序

eg：gcc -o test.c test（把源文件test.c编译成可执行程序test）

gcc -S test.c  （把源文件test.c转换为相应的汇编程序源文件test.s）

**time：**用于测算一个命令（即程序）的执行时间

**su：**切换使用者为超级用户

**netstat：**显示路由表、实际的网络连接以及每一个网络接口设备的状态信息

eg：**netstat -a #** 列出所有socket端口

**netstat -at #** 列出所有TCP端口

**netstat -au #** 列出所有UDP端口

**ln ：**创建链接；

eg：ln -s source target //软链接，将文件名为target的文件链接到文source上

ln source target //硬链接，将文件名为target的文件链接到文件source上

**echo：**字符串输出；

eg：echo It is a test //输出文本“It is a test”

## Q：统计一个文件中某个字符串出现的个数

A： a、**grep**

eg：grep -o ‘keyword’ my\_file.txt | wc -l

//直接使用grep -c返回包含指定单词的行数；采用-o返回这样的行，再用wc(word count)进行计数。

b、**awk**

eg：awk -v RS=’keyword’ ‘END {print –NR}’ my\_file.txt

v

## Q：gcc和g++的区别？

A： a、对于后缀为.c的文件，gcc将其当成C程序，g++将其当成C++程序；

b、对于.cpp后缀文件，gcc和g++都会将其当做c++程序；

c、编译阶段，g++会调用gcc，gcc、g++都可以编译c/c++；

d、链接阶段，通常会用g++来完成，如果使用gcc命令，其不会自动和c++程序使用的库连接，需要指定C++链接库。用gcc不会连接C++库

eg：gcc helloworld.cpp -lstdc++ -o helloworld

**gcc编译过程**

a、预处理,生成.i的文件[预处理器cpp] gcc -E

b、将预处理后的文件转换成汇编语言,生成文件.s[编译器egcs] gcc -S

c、有汇编变为目标代码(机器代码)生成.o的文件[汇编器as] gcc -c

d、连接目标代码,生成.out可执行程序[链接器ld] gcc -o

## Q：linux中查看系统资源的命令

A：**top**用来显示当前系统中各项资源情况，包括系统启动时间、登陆用户树、系统中的进程、CPU使用情况、物理内存使用情况。

**ps**用于查看当前系统中的所有进程状态。

ps -L可以查看进程的所有线程信息，线程ID表示未LWP

**free**用于查看系统内存使用情况；

**netstat**查看各网络端口情况。（netstat -a显示所有连线中的socket）

**vmstat**查看CPU使用情况。

**uptime**查看系统负载，包括平均运行的进程数量。

**iostat**查看系统设备的IO负载情况

**lsof**查看当前打开的文件，可用于查看当前端口占用情况

**df**查看磁盘占用情况

## Q：Linux文件读写使用的系统调用？

A：**打开、创建文件open()**

int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

pathname：文件路径；flags：读/写/创建；mode：文件权限；返回文件描述符

**关闭文件close()**

int close(int fd); fd就是open返回的文件名描述符

**读取文件read()** //把指定数目数据读到内存

ssize\_t read(int fd, void \*buf, size\_t count); buf内存首地址 count读取字节数

读取成功，返回读取的字节数；读取函数出错，返回-1；如果达到文件末尾，返回0；

**写文件write()** //将指定数目数据写到文件

ssize\_t write(int fd, const void \*buf, size\_t count); buf数据首地址

**文件偏移lseek()**

off\_t lseek( int fildes, off\_t offset, int whence)；

**建立内存映射函数mmap()**

void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset);

mmap()函数将普通文件映射到进程虚拟地址空间，进程可以像访问普通内存一样对文件进行访问。 mmap()还使得进程之间可以通过映射同一个文件实现共享内存。

**解除内存映射munmap()**

int munmap(void \*start, size\_t length);

https://blog.csdn.net/jinmie0193/article/details/79845316

## Q：mmap函数（内存映射）

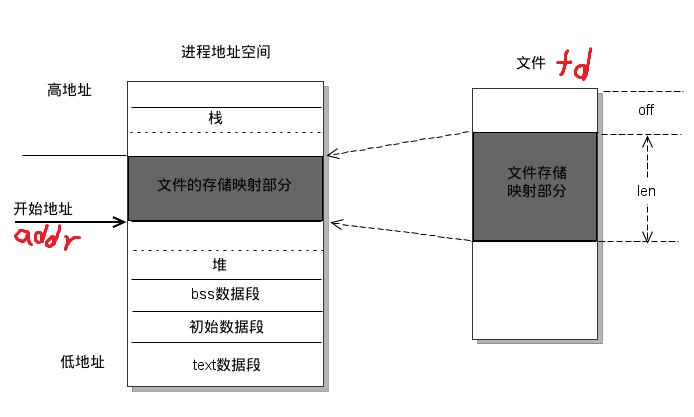
A：a、文件被打开后（open的过程是不能省的，缓存到内核的地址空间），使用mmap，将物理内存上的文件的一部分直接添加到进程的共享内存区，进程可以直接访问该部分内存。（通过read调用，需要陷入内核，由内核将对应部分的文件内容拷贝给进程）

b、多个线程可以通过映射物理内存中文件的同一部分，实现通信；

c、父进程调用mmap后fork子进程，子进程和父进程的共享内存区指向同一文件的同一部分，可以直接通信。

<https://www.cnblogs.com/charlesblc/p/6263665.html>

<http://blog.csdn.net/maverick1990/article/details/48050975>



**常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝。而mmap操控文件，只需要从磁盘到用户主存的一次数据拷贝过程。**说白了，mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。**页缓存处于内核空间。**

## Q：硬链接和软链接

A： 硬链接：若一个 inode 号对应多个文件名，则称这些文件为硬链接。换言之，硬链接就是同一个文件使用了多个别名。不存在实体文件和链接对应，通过在目录中增加一个链接到某inode号码的记录实现硬链接；

软链接：即符号链接，建立一个独立文件，文件中记录了被链接文件的路径名。

**区别：**

a、软链接对应有实体文件，硬链接没有；

b、软链接可以对文件和目录创建，硬链接不可以（如果子目录硬链接了父目录，则出现目录环）；

c、创建软链接时，文件的链接计数不会增加；硬链接会增加计数。

d、当链接原文件移动路径时，硬链接不受影响；软链接失效。

e、软链接可以跨文件系统创建、硬链接不可以；

|  |  |
| --- | --- |
| å¾ 2. è½¯é¾æ¥çè®¿é® | 元数据 用户数据å¾ 1. éè¿æä»¶åæå¼æä»¶ |

inode即索引节点号

## Q：Linux目录权限

A： a、linux中对使用者分为：用户（文件和目录的所有者）、用户组（同组成员）、其它用户；

b、文件权限分为：4（读）、2（写）、1（执行）；

chmod 654 /test //用户（读写）、用户组（读执行权限）、其它用户（只读）

c、目录中，只有同时具备读权限和执行权限才可以访问目录下的文件。

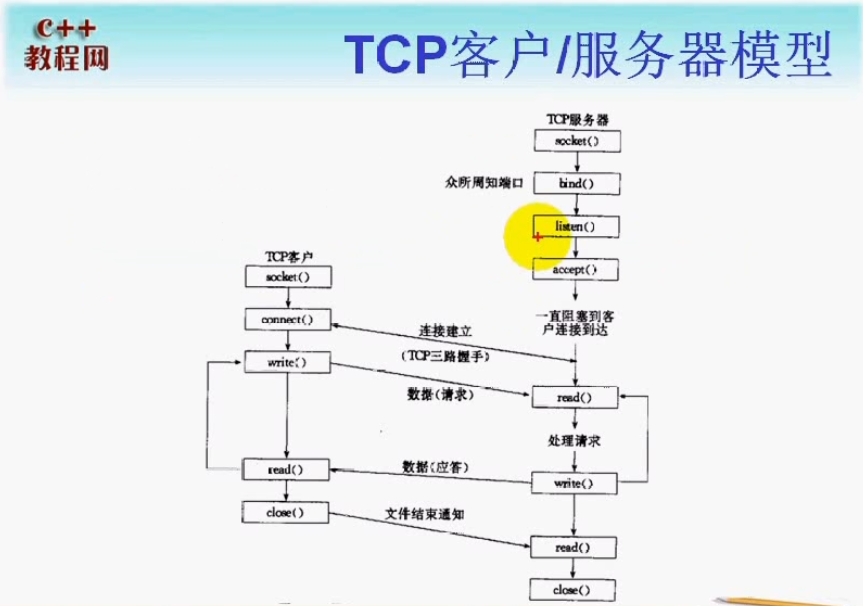
<https://www.cnblogs.com/puloieswind/p/5845638.html>

# 二、linux网络编程

## Q：TCP/UDP网络编程的客户端和服务器模型？

A：伪代码：

## Q: TCP



**//服务器端代码**

int main()

{

int listenfd = Socket(AF\_INET, SOCK\_STREAM,0);

//设置端口号和地址

Bind(listenfd, &servAddr, sizeof(servAddr));//servAddr设置协议族、IP地址、端口号

Listen(listenfd, LISTENQ); //Listen第二个参数是套接字排队的最大连接个数

Signal(SIGCHLD, sig\_chld);

for()

{

connfd = Accept(listenfd, cliAddr, cliLen); //Accept从已连接队列中取出第一个连接，返回连接套接字

if(fock() == 0)

{

Close(listenfd); //在子进程中，不再对listenfd进行读写，但并没有关闭套接字；

while((n = readn(connfd, buf, MAXLINE)) > 0)

{

writen(connfd, buf, n)

}

exit(0);

}

Close(connfd);//父子进程共享listenfd、connfd两个连接描述符，通过引用计数的//方式保证只有最后一个进程退出时，该描述符才被真正销毁，即连接才被断开

}

}

//信号处理函数

int sig\_chld()

{

while((pid = waitpid(-1, stat, WNOHANG)) > 0)

//处理pid子进程

}

**//客户端代码**

int main()

{

int connfd = Socket(AF\_INET, SOCK\_STREAM, 0);

Connect(connfd, servAddr, sizeof(servAddr));

while(Fgets(sendLine, MAXLINE, fp) != NULL)

{

Writen(connfd, sendLine, strLen(sendLine) );

Readn(socket, recvline, MAXLINE);

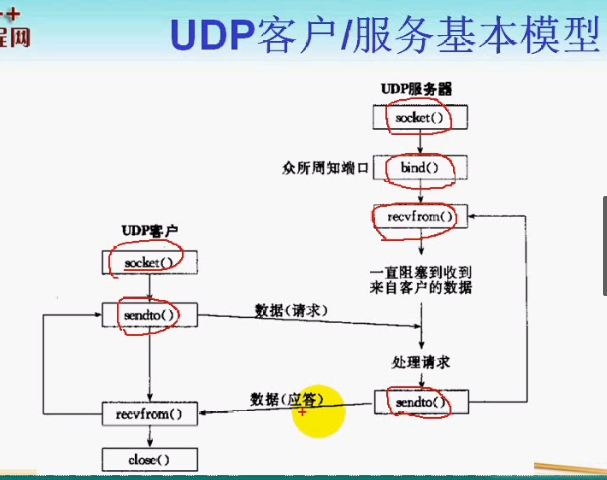
Fputs(recvline, stdout);

}

exit();

}

## Q: UDP



//服务器程序

int main()

{

sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

Bind(sockfd, &servAddr, sizeof(servAddr));

for()

{

n = Recvfrom(sockfd, mesg, MAXLINE, 0, cliAddr, len);

Sendto(sockfd, mesg, n, 0, cliAddr, len);

}

}

//客户端程序

int main()

{

sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

while(Fgets(sendline, MAXLINE, fd))

{

Sendto(socket, sendline, strlen(sendline), 0, servAddr, sizeof(servAddr));

n = Recvfrom(socket, mes, MAXLINE, 0, NULL, NULL);

Fputs(mes, stdout);

}

}

客户端，第一次sendto的时候绑定客户端地址，所以其实不需要bind

TCP/UDP网络编程对比：

1、大多数TCP服务器是并发的，大多数UDP服务器是迭代的；

2、UDP客户端程序不需要调用Connect函数， 当调用Connect函数后，该套接字只能与Connect关联的客户端套接字通信；

## Q：UDP的connect函数

A：由于UDP无连接，所以UDP传输过程中产生的异步错误（例如对端关闭错误）不会返回到UDP套接口，调用connect就是为了解决这个问题。

a、UDP调用connect不会引起三次握手，仅仅通知内核记录对端的IP和端口；

b、UDP可以重复调用connect，指定新的对端IP和端口；

## Q：僵尸进程、孤儿进程，编程时如何避免产生僵尸进程，如何kill僵尸进程？

A：**僵尸进程和孤儿进程**

僵尸进程：子进程先退出，父进程没有调用wait/waitpid处理它，子进程称为僵尸进程。僵尸进程进程ID仍然保存在系统中，但是不占用任何内存空间。僵尸进程父进程结束后，如果还没回收僵尸进程，僵尸进程变成孤儿进程，被init进程接管，最后由init进程回收其资源。

孤儿进程：父进程先于子进程退出，子进程称为孤儿进程。孤儿进程被init进程收养。孤儿进程退出时，init进程将回收其资源。

所以说，孤儿进程死后不占用系统进程ID，僵尸进程仍然需要占用。

**僵尸进程的避免：**

a、signal（SIGCHLD，SIG\_IGN），设置忽略子进程终止信号。这样子进程终止时，自动释放所有资源。

b、fork两次，子进程fork孙进程后直接退出，孙进程成为孤儿进程，退出时由init进程负责释放其资源。孙进程会继承子进程的资源

c、通过调用wait/waitpid函数。

**僵尸进程的处理：**

a、通过ps命令找到所有进程状态为Z的进程的PID及其父进程的PPID

ps -A -ostat,ppid,pid,cmd | grep -e '^[Zz]'

-A  参数列出所有进程

-o  自定义输出字段 stat（状态）、ppid（进程父id）、pid（进程id）、cmd（命令）

因为状态为z或者Z的进程为僵尸进程，所以我们使用grep抓取stat状态为zZ进程

grep -e '^[Zz]' 是过滤行首为Z或z的行。-e 表示后面的内容为正则表达式，'^[Zz]' 中 ^ 表示行首，[Zz]表示Z或z的任意一个字母。

b、kill子进程；HUP(1)是让进程挂起，睡眠;kill (9)六亲不认的；term(15)正常的退出进程

kill -HUP PID

c、如果子进程无法kill，kill父进程

<https://www.cnblogs.com/dongguolei/p/8086346.html>

<https://www.cnblogs.com/0616--ataozhijia/p/4324946.html>

## Q：wait/waitpid函数

A：pid\_t wait(int \* statloc);

pid\_t waitpid(pid\_t pid,int \*statloc,int options);

wait/waitpid用于处理已终止子进程。两个函数均返回两个值，已终止子进程的ID、子进程终止状态statloc。通过statloc可以判断子进程是正常终止、被信号杀死还是作业控制停止。

**wait/waitpid比较：**

a、waitpid可以处理多个子进程同时终止的情况。

waitpid通过设置options为WNOHANG，提供了一个无阻塞的wait，waitpid在没有子进程终止时会立即返回。那么在子进程终止信号被触发时，通过循环调用waitpid可以处理所有已终止进程。由于wait在子进程尚未终止时会阻塞，所以不能在处理函数中循环调用wait，每次只能处理第一个终止的子进程。

b、设置pid大于零，使waitpid等待指定的进程

c、waitpid支持工作控制，对终止状态（正常终止、异常终止、暂停终止）进行检查。设置options为WUNTRACED。

**waitpid中pid的含义：**

a、pid==-1 等待任何一个子进程；

b、pid >0   等待进程ID与pid值相同的子进程

c、pid==0   等待与调用者进程组ID相同的任意子进程

d、pid<-1   等待进程组ID与pid绝对值相等的任意子进程

<https://blog.csdn.net/kevinhg/article/details/7001719>

## Q：Shutdown和Close

A： a、close关闭描述符时，采用引用计数的方式，只有在线程连接计数值变为0时才关闭套接字。使用shutdown可以不管计数就激发TCP的正常终止过程。

b、close终止读和写两个方向的数据传输。shutdown可以只关闭其中一半。例如，shutdown关闭写端后，仍然可以接收从对方过来的数据，直到对方关闭连接。howto可以设置为SHUT\_RD，SHUT\_WR，SHUT\_RDWR.

int shutdown（int sockfd， int howto）；

## Q：获取和设置套接字选项

A：**获取、设置方式 socket 用户进程与内核网络协议栈的通讯接口**

a、getsockopt、setsockopt函数

b、fcntl函数

设置套接字为非阻塞型（默认是阻塞）；

fcntl（fd，F\_SETFL，O\_NONBLOCK）；

c、ioctl函数

**getsockopt、setsockopt常用套接字选项**

a、SO\_KEEPALIVE；该选项被设置后，如果一段时间内（默认两小时）整个TCP连接没有数据传输，TCP就会给对方发送一个探测存活的数据包，以此检测对方是否存活，及时清理半开连接。

b、SO\_LINGER；用于设置当调用close函数时，如果发送缓冲区还有数据未发送，是close延迟关闭等待发送完成，还是直接丢弃数据。

c、SO\_REUSEADDR；设置允许多个套接字对一个端口的绑定。例如重启TCP服务器程序时，迅速连接到原端口.设置服务器地址端复用。

d、SO\_RCVBUF；设置UDP接收缓冲区大小

e、SO\_RCVTIMEO；设置接收的超时时间；

f、SO\_SNDTIMEO；设置发送的超时时间；

## Q：recv函数的返回值及其意义？

A： 1、接收成功，返回其实际copy的字节数（读到指定字节数量时，返回len；当读到文件结尾时，返回值小于len）。

2、如果recv在copy时出错，那么它返回-1，并设置相应错误（SOCKET\_ERROR）；

3、返回0表示对方关闭了套接口。

send函数  
int send( SOCKET s,const char\* buf,int len,int flags);

第一个参数指定发送端套接字描述符；  
第二个参数指明一个存放应用程序要发送数据的缓冲区；  
第三个参数指明实际要发送的数据的字节数；

recv函数  
int recv( SOCKET s,char\* buf,int len,int flags);   第一个参数指定接收端套接字描述符；  
第二个参数指明一个缓冲区，该缓冲区用来存放recv函数接收到的数据；  
第三个参数指明buf的长度；

## Q：有哪些网络I/O？阻塞IO和多路复用IO的区别

A： IO的过程分为两个部分：等待数据到达、从内核缓冲区拷贝数据到进程空间（内存）

**阻塞IO**：在IO执行的两个阶段，进程一直被阻塞。一个进程同时只能处理一个IO。

**非阻塞IO**：等待数据到达期间，如果没有数据到达，内核立即返回一个错误，进程不阻塞，继续发送请求。数据到达，从内核拷贝数据期间，进程阻塞。（ 每个进程可以同时处理多个IO请求，但是循环调用请求占用大量CPU）

**信号驱动IO：**进程提前建立SIGIO的信号处理程序，当数据到达时通知线程。所以等待数据到达期间，进程不阻塞。数据到达，内核通过SIGIO信号通知进程，进程调用recv拷贝数据，期间阻塞。（单进程可以处理多个IO，但是信号处理开销较大）

**多路复用IO**：用select、poll、epoll同时对多个IO进行监测。等待数据到达期间，进程阻塞于select、poll、epoll上，期间进程也不能处理其它任务。有数据到达时，系统调用返回，对应的IO再去拷贝数据。（每个进程可以处理多个IO）

**异步IO**：调用aio\_read函数，把IO的两个过程都交给内核，期间进程不阻塞，继续处理其它任务。由内核等待数据到达，然后将数据拷贝到用户内存空间，拷贝完成后再通过信号通知进程。（单个进程处理多个IO）

阻塞IO和多路复用IO比较：

a、都属于同步IO。在等待数据到达过程中，对于阻塞IO，进程阻塞于IO调用；对于非阻塞IO，进程阻塞于select、epoll调用；拷贝数据期间，进程阻塞。

b、阻塞IO单进程只能同时处理一个IO；

非阻塞IO和异步IO比较：

a、都支持单线程对多个IO的监控；

b、非阻塞IO是同步IO，从内核拷贝数据的过程需要进程自己完成；

c、非阻塞IO通过轮询的方式确认数据到达，CPU开销非常大。

同步和异步、阻塞和非阻塞：

a、同步和异步指的是，处理IO时，需不需要一个严格的顺序，在一个调用没有返回结果之前，该调用就不算完成；对应到IO，就是一次IO完成是不是以进程把数据拷贝到进程空间为标志，除异步IO以外的所有IO，最后都需要自己去拷贝数据，所以都是同步IO。

b、阻塞和非阻塞是等待消息时，进程有没有被挂起

<https://blog.csdn.net/sh_c1991/article/details/52351010>

## Q：select/poll/epoll函数的区别？（实现I/O端口复用）

A：**select**

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

传递给 select函数的参数会告诉内核：

•我们所关心的文件描述符

      •对每个描述符，我们所关心的状态。(我们是要想从一个文件描述符中读或者写，还是关注一个描述符中是否出现异常)

      •我们要等待多长时间。(我们可以等待无限长的时间，等待固定的一段时间，或者根本就不等待)

   从 select函数返回后，内核告诉我们一下信息：

      •对我们的要求已经做好准备的描述符的个数

      •对于三种条件哪些描述符已经做好准备.(读，写，异常)

参数介绍：

readfds：select监视的可读文件句柄集合。

writefds: select监视的可写文件句柄集合。

exceptfds：select监视的异常文件句柄集合。

n表示待测试的最大描述符编号加1；通过设置n告知内核只需要检测编号小于n的描述符空间，提高效率。

返回值大于0表示有多少个文件描述符就绪（可以通过这个提高select返回后对文件描述符检测的效率）；返回值等于0，表示超时；返回值小于0，表示出现异常。

timeout超时时间，当timeout为0时，表示select检查完立即返回，不需要等待超时时间到达。可以轮询。

a、最大监控文件描述符限制。受两方面影响：进程可以打开的最大文件描述符数量限制；fd\_set的大小限制，fd\_setsize默认为1024；

b、检测文件描述符就绪的效率低：每轮循环都需要扫描整个fd\_set，集合越大，速度越慢；select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件；

c、每次调用时，都需要将fd\_set数组拷贝到内核空间，返回时，又需要拷贝到用户空间。拷贝开销大。

d、每轮循环后，都需要重置fd\_set的状态；

**poll**

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

基本与select相同，但是poll用链表来记录需要检测的设备文件描述符，无最大检测文件描述符限制。

**epoll**

int epoll\_create(int size)；//创建一个epoll的句柄，内部用哈希表实现，size表示哈希表容量；目前采用epoll\_create1（int flags），内部采用红黑树实现。

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout); 收集在epoll监控中已经发生的事件

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。

a、可以检测的描述符数量无限制；

b、最大优点是，检测效率就不随着描述符数量增加而下降。内部管理有一个就绪链表，通过callback的方式通知描述符就绪，不需要通过轮询。

c、将增加描述符和监听过程分开，通过共享内存的方式实现添加描述符。调用epoll\_wait时就不用拷贝文件描述符到内核空间，省去了大量拷贝过程。

d、epoll返回时，不仅可以通知有IO事件到来，还可以直接定位到是哪个fd就绪，不必遍历整个fd集合。

注意：epoll相对于select，只有在**高并发**（需要监控的文件描述符多），并且同一时间只有**少数socket是活跃的**情况下才有优势。在并发量低、socket都比较活跃时，select和epoll效率差不多。

<https://blog.csdn.net/shenya1314/article/details/73691088>

<http://blog.csdn.net/qq_16836151/article/details/52089404>

<https://blog.csdn.net/sh_c1991/article/details/52351010>

## Q：epoll介绍，即其两种触发模式，哪种效率高？

A： a、**LT电平触发模式**：缺省的工作方式，当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，直到程序处理完所有数据，否则下次调用epoll\_wait时，会再次响应应用程序并通知此事件。 （poll默认的方式）

b、**ET边缘触发模式**：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，epoll\_wait只被触发一次，如果数据没有处理完，下次调用epoll\_wait不会再响应。

**效率：**

如果采用ET模式,那么仅当状态发生变化时才会通知,而采用LT模式类似于原来的select/poll操作,只要还有没有处理的事件就会一直通知. LT模式触发的次数更多；ET模式使用起来更复杂，在数据读写不完全时调用epoll\_wait会永久阻塞，所以需要应用程序自身维护就绪队列，保证数据的读写完全。

* ET模式（边缘触发）**只有数据到来才触发**，**不管缓存区中是否还有数据**，缓冲区剩余未读尽的数据不会导致epoll\_wait返回；
* LT 模式（水平触发，默认）**只要有数据都会触发**，缓冲区剩余未读尽的数据会导致epoll\_wait返回。

## Q：用select实现IO的超时设置

A： 利用select函数在超时后返回0的特点，可以实现对IO的超时设定。包括：

a、read\_timeout;函数封装

b、write\_timeout; 函数封装

c、accept\_timeout; 写套接口可用

d、connect\_timeout;connect完成时，是写套接口可用。

## Q：Direct I/O的使用及优缺点

A：**Direct I/O：**

直接IO是一种实现零缓存的方式，让**用户地址空间和磁盘空间之间直接进行数据传输**，不需要经过内核缓冲区。操作系统层提供的缓存往往会使应用程序在读写数据的时候获得更好的性能，但是对于某些自缓存应用程序（例如数据库管理系统），它们比操作系统更了解其要操作的数据的语义，所以它可以采用更加高效的缓存替换算法。直接 I/O 经常会和异步I/O的方式结合在一起使用。

**Direct I/O的使用**

打开文件的时候设置对文件的访问模式为 O\_DIRECT，接下来使用 read() 或者 write() 系统调用去读写文件的时候使用的是直接I/O方式。

**优点：**

通过减少操作系统内核缓冲区和应用程序地址空间的数据拷贝次数，降低了对文件读取和写入时所带来的 CPU 的使用以及内存带宽的占用；

**缺点：**

a、设置直接 I/O 的开销非常大，直接 I/O 不能提供缓存 I/O 的优势。当内核高速缓冲存储器存有文件的副本时，缓存 I/O 的读操作可以直接从高速缓冲存储器中获取数据，而直接 I/O 的读数据操作会造成磁盘的同步读，这会带来性能上的差异 , 并且导致进程需要较长的时间才能执行完。

b、与直接 I/O 读操作类似的是，直接 I/O 写操作也会导致应用程序关闭缓慢。所以，应用程序使用直接 I/O 进行数据传输的时候通常会和使用异步 I/O 结合使用。

## Q：零拷贝是什么？

A：Linux中传统的 I/O 过程中，产生的数据传输通常需要在缓冲区中进行多次的拷贝操作。零拷贝就是一种减少数据拷贝次数的技术。

**零拷贝技术的目标**：

a、避免操作系统内核缓冲区之间进行数据拷贝操作。

b、避免操作系统内核和用户应用程序地址空间这两者之间进行数据拷贝操作。

c、用户应用程序可以避开操作系统直接访问硬件存储。

d、数据传输尽量让 DMA 来做。正常是数据进缓存区，满了通知cpu写入内存，DMA直接将磁盘缓冲区内容缓存在自己这里，读取结束通知cpu，减少了cpu开支。

**零拷贝技术**：

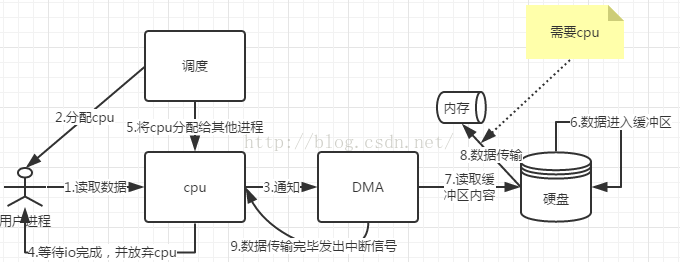
a、直接 I/O：对于这种数据传输方式来说，应用程序可以直接访问硬件存储，操作系统内核只是辅助数据传输：这类零拷贝技术针对的是操作系统内核并不需要对数据进行直接处理的情况，数据可以在应用程序地址空间的缓冲区和磁盘之间直接进行传输，完全不需要 Linux 操作系统内核提供的页缓存的支持。

b、使用mmap（三次拷贝，四次切换）、sendfile（三次拷贝、两次切换）等代替传统的read/write操作。https://blog.csdn.net/u013096088/article/details/79122671

c、**写时复制技术**：如果有多个应用程序需要同时访问同一块数据，那么可以为这些应用程序分配指向这块数据的指针，当其中一个应用程序需要对自己的这份数据进行修改的时候，就需要将数据真正地拷贝到该应用程序的地址空间中去。

## Q：DMA是什么？

A： DMA（Direct Memory Access）是一种无需CPU的参与就可以让外设和系统内存之间进行双向数据传输的硬件机制。使用DMA可以使系统CPU从实际的I/O数据传输过程中摆脱出来，从而大大提高系统的吞吐率。



## Q：守护进程

A：守护进程：它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。它不需要用户输入就能运行而且提供某种服务，不是对整个系统就是对某个用户程序提供服务。Linux系统的大多数服务器就是通过守护进程实现的。常见的守护进程包括系统日志进程syslogd、 web服务器httpd、邮件服务器sendmail和数据库服务器mysqld等。

**守护进程的特点：**

a、孤儿进程；

b、以root权限运行，使用保留端口号（1-1024）

c、非交互式程序，没有控制终端，没有任何输出。

**创建守护进程的步骤：**

a、在父进程中执行fork并exit推出；（保证进程不是进程组组长，否则不能调用setsid）

b、在子进程中调用setsid函数创建新的会话，子进程成为会话组长，并与控制终端脱离；

c、重新fork，使进程不再成为会话组长，从而禁止进程重新打开控制终端；

d、在子进程中调用chdir函数，让根目录 ”/” 成为子进程的工作目录；

e、在子进程中关闭任何不需要的文件描述符

f、重设文件创建掩模，设置进程的umask为0；

**常用的守护进程：**

inetd超级服务器、FTP服务器、Web服务器httpd、邮件服务器sendmail、系统日志服务syslogd。

<http://www.cnblogs.com/mickole/p/3188321.html>

## Q：设想有一个守护进程管理的服务挂掉了怎么拉起来？

A： 重启对应的socket

## Q：inetd超级服务器

A：**为什么需要inetd守护进程：**

a、FTP、Telnet等守护进程的启动代码几乎相同；

b、大部分守护进程不需要工作。

**inetd工作原理：**

a、通过inetd处理普通守护进程的启动细节；

b、单个进程为多个服务等待客户请求，只有在相关请求到达时，才创建一个子进程进行处理，处理完就关闭请求，从而**减少系统中的进程数量**。

**inetd启动流程：**

a、在启动阶段，读入/etc/inetd.conf文件并给该文件中指定的每个服务创建一个适当类型的套接口。

b、为每个套接口调用bind，指定捆绑相应服务器的众所周知端口和通配IP地址。

c、对于每个TCP套接口，调用listen以接受外来的连接请求。对于数据报套接口则不执行本步骤。

d、创建完毕所有套接口后，调用select等待其中任何一个套接口变为可读。

e、当select返回指出某个套接口已可读之后，如果该套接口是一个TCP套接口，而且其服务器的wait-flag值为nowait，那就调用accept接受这个新连接。

f、inetd守护进程调用fork派生进程，并由子进程处理服务请求。

g、子进程关闭除要处理套接字描述符之外的所有描述符；

h、子进程调用exec载入具体的服务程序。

**inetd适用场合：**

inetd只适用于较少启动的服务（例如FTP、Telnet等），对于服务密集型服务器，比如邮件服务器和web服务器，普通方式下，处理每个连接的开销是一个fock，但是使用inetd方式，开销是一个fork和一个exec调用，综合开销可能更大。

用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程执行的程序完全替换为新程序，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段。

## Q：read、readn、recv

A： a、read函数可以从任何文件io中读取数据，recv只能用于从套接口读取数据；

b、read函数读取数据时，可能会由于缓冲区已满，导致读不到指定大小的数据，所以通常需要在while循环中读数据；readn可以避免这个问题。

c、recv相对于read，多了第四个参数flags，当其为0时，功能和read相同。当设置为MSG\_WAITALL时，功能同readn。另外，还可以设置为MSG\_OOB接收紧急指针中的数据。设置为MSG\_PEEK可以读取数据，但是不从缓冲区中移走接收到的数据。

read 原则：

        数据在不超过指定的长度的时候有多少读多少，没有数据则会一直等待。所以一般情况下：我们读取数据都需要采用循环读的方式读取数据，因为一次read 完毕不能保证读到我们需要长度的数据，read 完一次需要判断读到的数据长度再决定是否还需要再次读取。

recv 原则：  
        recv 中有一个MSG\_WAITALL 的参数:  
                recv(sockfd, buff, buff\_size, MSG\_WAITALL),  
        正常情况下recv 是会等待直到读取到buff\_size 长度的数据，但是这里的WAITALL 也只是尽量读全，在有中断的情况下recv 还是可能会被打断，造成没有读完指定的buff\_size的长度。所以即使是采用recv + WAITALL 参数还是要考虑是否需要循环读取的问题，在实验中对于多数情况下recv (使用了MSG\_WAITALL)还是可以读完buff\_size，所以相应的性能会比直接read 进行循环读要好一些。

# 二、linux多线程编程

## Q：线程的创建、退出、连接

A： 线程创建：

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);

线程退出：

void pthread\_exit(void \*retval);

线程连接：

int pthread\_join(pthread\_t thread, void \*\*retval);

## Q：让线程退出的方式：

A： a、进程退出，调用exit()；

b、其它线程调用pthread\_cancel请求退出；

c、线程的入口函数return、main函数return

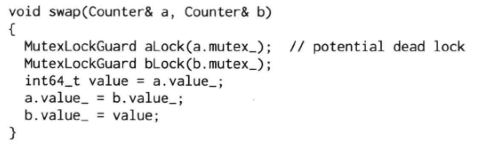
d、线程调用pthread\_exit

## Q：多线程安全问题

A：**用mutex保证线程安全的局限性：**

a、mutex只能用于同步本class数据成员的读写操作，不能保护安全析构；因为析构函数已经超出了成员变量的生命期；

b、使用mutex可能出现死锁，如下图，如果线程A执行swap(a,b)，线程B执行swap(b,a)，可能死锁。



## Q：gdb的基本使用方法？

A： 使用gcc -g生产可以供gdb调试的.out文件；

**常用命令：**

gdb file //启动gdb，打开file文件

（gdb）l //从第一行开始列出源码；

（gdb） //直接回车表示重复上一次命令；

（gdb）b 10/fun //在第10行/fun函数入口位置增加断点；

（gdb）info b //查看断点信息

（gdb）r //运行程序

（gdb）c //继续运行

（gdb）n //单步执行、next（vs中的F10）

（gdb）s //单步跟踪进入（vs中的F11）step

（gdb）p i //打印参数i的值print

（gdb）bt //查看函数堆栈

（gdb）q //跳出gdb

（gdb）b 13 if i == 8 //条件断点，当条件满足时，程序运行停止

（gdb）watch i //设置观察点

<https://blog.csdn.net/haoel/article/details/2879>

<https://blog.csdn.net/tzshlyt/article/details/53668885>

## Q：可重入函数和不可重入函数

A：在多任务系统当中，在任务执行期间捕捉到信号并对其进行处理时，进程正在执行的指令序列就被信号处理程序临时中断。如果从信号处理程序返回，则继续执行进程断点处的正常指令序列，从重新恢复到断点重新执行的过程中，函数所依赖的环境没有发生改变，就说这个函数是可重入的，反之就是不可重入的。

不可重入函数：不同任务调用函数时，可能修改其它任务调用这个函数的数据。不可重入函数被视为不安全函数；

可重入函数：一个被多个任务调用的过程，在调用时不会互相干扰。

**以下函数不可重入：**

a、函数体内使用了静态的数据结构；

b、函数体内调用了malloc或者free函数；（使用了全局内存分配表）

c、函数体中调用了标准IO函数；（使用了全局变量stdout）标准io库很多实现都以不可重入的方式使用全局数据结构。

## Q：无锁编程

A： 使用CAS和内存屏障

//以上为整理过的面经

1. 硬链接和软连接区别
2. kill用法，某个进程杀不掉的原因（进入内核态，忽略kill信号）
3. linux用过的命令
4. 系统管理命令（如查看内存使用、网络情况）
5. 管道的使用 |
6. grep的使用，一定要掌握，每次都会问在文件中查找
7. shell脚本
8. find命令
9. awk使用