1. 软链接(会占用磁盘大小，常用于升级) 和 硬链接

(1)建立软链接和硬链接：操作目标文件即为操作源文件

软链接：ln -s 源文件 目标文件 (目标文件的大小 即为源文件的名字字符所占的大小)  
硬链接：ln 源文件 目标文件

**动态库通过软链接很容易升级。** 如原来的版本为add.c，建立软连接link -s add.c link ，当版本升级后为add1.c时，再建立软连接link -s add1.c link ,然而对于程序来说，我们每次加载的依然是link没变，故容易升级。

(2)软链接：

**软链接可以理解成快捷方式**。它和Windows下的快捷方式作用是一样的：**软链接文件的大小和创建时间和源文件的大小不同**。当源文件被删除后，软链接也不会存在了：软链接文件只是维持了从软链接到源文件的指向关系，而不是源文件的内容，大小应为源文件名字的字符所占的大小。软链接文件和源文件的inode值不同。

(3)硬链接：

**硬链接等于cp拷贝加同步更新**：**硬链接文件和源文件的大小和创建时间一样**。硬链接文件的内容和源文件的内容一模一样，且硬链接文件和源文件的inode值相同。

如何区别硬链接和拷贝？

那么cp -p拷贝的文件是不是就相当于硬链接了呢？其实不是，对于源文件的内容有修改，**硬链接文件会同步更新修改，始终保持和源文件的内容相同**，而复制的文件则不可能做到这一点。且拷贝和源文件的inode值不同。

(4)作用和区别：**软链接源文件不可删除 ；硬链接源文件可删除**。

软链接：软链接像快捷方式，方便我们打开源文件。

硬链接：允许一个文件拥有多个有效路径名，这样用户就可以建立硬连接到源文件，可以防止源文件被“误删”。

**2. 常用命令大全：**

(1)Python：是C的忠实粉丝，觉得用C写程序效率太低了，就把C语言的很多接口封装成了命令，发明了python语言。

cat /etc/issue 查看Linux的版本号系统信息 (Ubuntu、CentOS小红帽)。

Linux下按Tab键能自动联想； 文件以 点 . 开头的都是隐藏文件。

**所有配置文件都在/etc目录下； 程序文件放在/home目录下； 日志在/var/log目录下；设备文件在/dev目录：c开头代表字符设备(鼠标键盘打印机等)，b开头代表块设备(磁盘)**

(2)

仅 / 开头代表的是根目录，故输入绝对路径时均是以 / 开头，

如： cd / (到根目录)； ls / (查看根目录)； cd ~ :波浪线~代表家目录，即当前用户下的家目录。 点 . 缺省的，代表当前目录：ls 等价于 ls .

dpkg命令：查看安装包,如查看ssh有没有安上：dpkg -l|grep ssh

(3)

输完一条命令后，**看命令的返回值： echo $? (是shell脚本命令)**

返回值为0即成功，不为0失败。

(4)在root下配置用户：

要想添加一个普通用户user1 ，则可以使用 useradd user1。使用该命令后，系统会在目录“/home”下建立一个名为 user1 的目录。**加 -m才会创建目录,ubuntu下需要 useradd –m test –s /bin/bash**

1) -s的作用是用于指定所使用的脚本解析器 ( /bin/bash脚本解析器 )

2) /home目录：所有普通用户均没有写权限，因为其属于root用户：即为什么配置新用户的时候要加上参数 -m，才有目录显示，因为普通用户无法chmod和创建用户目录。

**scp**: 远程拷贝文件从一个Linux上到另一台Linux上。(桥接模式), **ssh**即用于远程登陆。

Linux上查看有哪些用户的命令：**cat /etc/passwd** (注意：所有配置文件都在/etc目录下)

**which**命令：查看系统命令所在的位置； 即在PATH变量指定的路径中搜索某个系统命令的位置，并返回第一个搜索结果。

(5)**脚本解析器：用于解析脚本命令，如Shell脚本(bash)、Python。**

Linux原生态的脚本是bash脚本： Linux 中的 shell 有很多类型，其中最常用的几种是: Bourne shell (sh)、C shell (csh) 和 Korn shell (ksh)。 Linux 操作系统缺省的 shell 是Bourne Again shell，它是 Bourne shell 的扩展，简称 **Bash**，与 Bourne shell 完全向后兼容，并且在Bourne shell 的基础上增加、增强了很多特性。Bash放在/bin/bash中

(6)

Linux多用户系统，看此时我是哪一个用户的命令：whoami

Windows是单用户系统，同时只能有1个用户登陆。 退出当前用户命令： exit；

查看系统中已经存在的环境变量：env； 查看环境变量的路径:echo $PATH

tree -h : ：以树状图形式列出目录的内容，并显示大小。

Linux输命令时：ctrl+a光标到行首； ctrl+e光标到行尾。

(7)把今天所有输出的命令给导出：history

**脚本语言中 ， 大于号 ＞ 是指：重定向标准输出**

如：history >file\_xuebin.txt，指把所有命令写到file\_xuebin.txt文件中。

(8)配置固定IP，防止IP变了登不上Xshell

**Linux下用ifconfig查看的IP不是固定IP，是每次网关来分配的。**故有时候用Xshell连接不上，就是因为IP变了。所以最好要自己设置好固定IP。

配置 固定 IP：

1、记录下自己的IP ，记录自己的网关 route

2、在Ubuntu上点击上下箭头-》Edit connections-》IPv4 Settings

从dhcp改为 manual，Add上面的IP为固定IP。

掩码 255.255.255.0

DNS server

202.96.134.33

sudo ifconfig ens33 down 关闭网卡

sudo ifconfig ens33 up 启动网卡(即重新进行网络配置)

桥接模式下：Linux和Windows是处于同一个网络中的。

(9)**文件查找： find 起始目录 查找条件 操作**

1)**find /usr/include -name stdio.h** 查找/usr/include目录下名为stdio.h的头文件

2)**find . -name file** 查找当前目录下名为file的文件。

**find . -empty** 查找当前目录下大小为0的空目录或空文件。（如下图）

3)通配符: **\*** 0个或任意多个字符;  **？**只有1个字符;  **[]** :在一范围内的字符

**find . -name “\*.c”** 查找当前目录下的所有 .c文件，因为 ’\*’星号 是特殊字符(通配符)，故要加上引号“”

**find . -name “work\*.c”** 查找当前目录下名字中含有work的所有.c文件

**find . -name “work？”** 查找当前目录下名为file且后有1个字符的名字的文件

**find . -name “fil[a-d][1-2]”** 查找当前目录下名为fil，第四个字符为a~d范围，且第五个字符为1~2范围内。

**find /usr/include/ -name stdio.h|xargs ls -l** 分别输出/usr/include/目录下各个名为stdio.h文件的详细信息。

4)该命令的查找条件可以是一个逻辑运算符not、and、or组成的复合条件。

<1> and：逻辑与，在命令中用-a表示，表示只有当所给的条件都满足时，查找条件才满足。

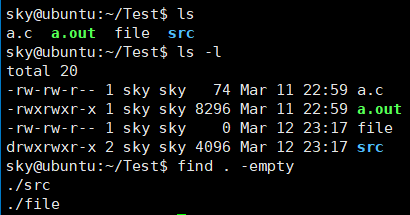
**find . -name file -a -type d** 查找的file文件是一个目录下的文件

**find . -name file -a -type f** 查找的file文件是一个普通文件

<2> or：逻辑或，在命令中用-o表示，表示只要所给的条有一个满足，查找条件就满足。

**find /home/user -name main.cc -o -name main.c**

在“/home/user”目录下查找名字为main.cc 或名字为 main.c的文件。



(10)查看磁盘空间

df -h： 显示整个磁盘的使用情况 (在显示的/dev/sda中)。

du -h： 显示每个文件和目录的磁盘使用空间 (du -h main.c：显示main.c所占磁盘大小)

(11)搜索之前的命令(命令很长，不想往上翻和再一次手动输入）

ctrl+r：找到后想执行 按回车， 不想执行修改命令 按Tab键。

(12)查看文件内容 cat

1)echo后什么都没有写，默认输出\n

2)命令：head –n(行数值) 文件名 ：显示目标文件的前几行。

命令：tail –n(行数值) 文件名 ：显示目标文件的最后几行。

history |head -10 查看历史命令的头10行。

history |tail -10 查看历史命令的最后10行。

**head -10 main.c >file** 将main.c中的前100行内容导入到(重定向)file文件中。

3)单页浏览文件(不可编辑)： more或者less命令: 按回车往下翻，按q退出

4)对文件内容进行排序(根据ASCII码)：sort

5)查看文件的内容是什么类型(判别文件类型)：file ,如：file /usr/bin/gdb

6)统计指定文件中的 行数、字数、字节数： **wc 文件名**

wc -c :统计字节数。

wc -l :统计行数/文件数：如**ls /etc|wc -l :统计etc目录下有多少个文件(行)**

wc -w :统计字数。一个字被定义为由空白、跳格或换行字符分隔的字符串

(13) 搜索文件中的内容：grep

grep God file : 搜索出file文件中出现过的God的行。

find /usr/include/ -name stdio.h|xargs grep FILE

find . -name \*.c|xargs grep main 查找当前目录下所有.c文件中的内容含有main的行。

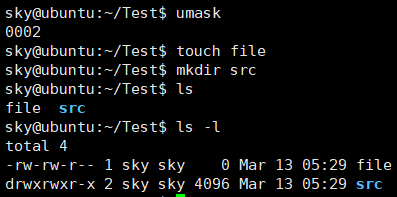
(14)掩码umask ，通过umask命令修改的掩码，只有一次生效 (为了不给某些用户权限)。

umask指文件（0666）或目录（0777）创建时在全部权限中要去掉的一些权限，普通用户缺省时umask的值为002，超级用户为022。002表示创建目录时所有者的权限不去掉， 所属组权限不去掉，其他组权限写属性去掉。

创建一文件以后，普通用户缺省的权限为664 、超级用户644

创建一目录以后，普通用户缺省的权限为775 、超级用户755

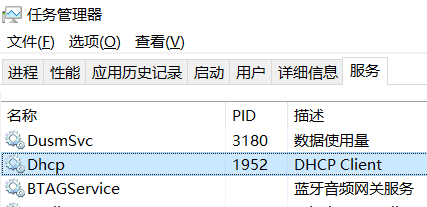
可以通过umask查看默认的缺省的掩码值。通过umask 001修改掩码值。



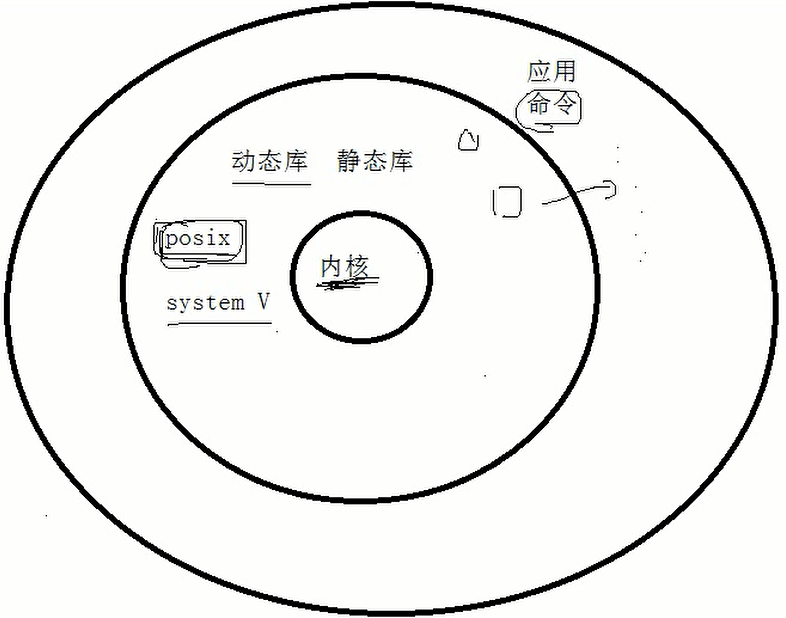
3.压缩、交换机、DHCP

压缩算法只能压缩单个文件。平时压缩多个文件时，是一种错觉：它先把多个文件打成一个包(连续的)，再把这个包进行压缩。

交换机：也是一台电脑，和个人电脑区别是不需要很大的硬盘。交换机里面有一个服务就是DHCP Service服务，用于给网络中每一个拥有DHCP Client的客户端分一个IP (因为DHCP模式，所以能自动获取到IP)。



4.



1. **最内层：内核(装载和启动可执行程序),内核对外提供了1000多个系统调用。**

**中间层：大量的静态库、动态库。主要学习的是动态库提供的大量接口，这些接口主要有两个标准：Posix标准、System V标准。**

**最外层：我们写的应用程序便运行在最外层 、Linux命令(前辈已写好)也在最外层。**

BSD协议(苹果电脑就是得到此协议的授权)： system v标准

GPL协议： posix标准

(2)硬件资源，如内存管理、CPU的使用、网络、磁盘等均通过内核来管理。故**不论什么语言都必须要用这1000多个系统调用才能使用到硬件资源**，而内核提供的系统调用比中间层的接口少了很多。 驱动：是运行在内核中！！！

系统调用：是由内核提供的接口，系统调用有1000多个； 对框架的掌握 和 理解

标准C都是系统调用实现的！！！

(3)Linux应用场景：服务器、手机内核、机器人、嵌入式、AR、VR。

5.重点！！！

-l 列出指定目录下所有目录及文件的详细信息。例如 列出“/root/home”目录下的所有子目录及文件，则使用

**ls –l ：每行列出的详细信息依次是：**

**文件类型与权限 硬链接数 文件所有者uid 文件所属组gid 文件大小 最近修改时间 文件名字**

使用ls –l 命令显示的信息中 ，**开头是由10个字母构成的字符串**：指的是

(1)其中**第一个字符表示文件类型**，它可以是下列类型之一：

- ：普通文件

d ： 目录

l ：符号链接

b ：块设备文件

c ：字符设备文件

p ：命名管道

s ：socket文件

(2)**后面的9个字符表示文件的访问权限**，分为3组，每组3位。

第一组表示文件创建者的权限，第二组表示同组用户的权限，第三组表示其他用户的权限。**每一组的三个字符分别表示对文件的 读、写、执行权限**。

各权限如下：r（读）、w（写）、x（执行）、\_（没有设置权限）。每一组可以用一个数字表示，例如 r\_x : 5 , rw\_:6 , r\_\_:4，

那么这三组就可以用3个数字表示，例如rwxr\_xr\_x:755 ， rw\_r\_\_r\_\_:644。

ls输出内容是有颜色的，比如：目录是蓝色，压缩文件是红色的显示，如果没有颜色，可以加上参数--color=never 表示输出没有彩色，而--color=auto 表示自动，--color=always表示始终有颜色。

(3)

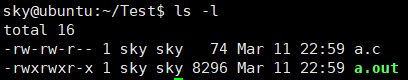
若文件为目录时：其大小仅指目录本身的大小，不包括此目录下的文件大小(区别Windows)

Linux下目录默认所占大小均为4K，即4096字节。

文件名和文件内容是分开存储的，文件名是在目录大小中存着。

(4)

Linux可执行文件不以后缀来区分(区别Windows的exe文件)，要通过ls -l查看权限，即可执行文件有x权限，一般为rwx。如下a.out可执行文件：



6．**重点Vim操作：**是Linux/UNIX 系列OS 中通用的全屏编辑器。在命令状态下：

(1)写内容时：按I从光标所在行的开始处插入文本；

按A从光标所在行的末尾开始添加文本

Vim保存不退出(类似Windows下的ctrl+s) **:w**

(2)删除

x 删除光标处的字符(仅删除一个字符);

D 删除此时光标位置到行尾的文本

(3)移动光标

(小帽子) ^ 光标移动到行首； $ 光标移动到行尾.

Ctrl+f 向下翻一页； Ctrl+b 向上翻一页.

H 光标定位到当前页首； L 光标定位到当前页的最后一行的行首.

w 光标往后移一个字； b 光标往前移一个字.

(4)查找 :/

**:/要查找的字符**，其中有多个要查找的字符时：按n向下移动，按N向上移动。

(5)替换 :s/

:s/God/Ghost 把光标所在行中的第一个God替换成Ghost

:5,12s/God/Ghost 把5 ~ 12行中的第一个God替换成Ghost

:5,12s/God/Ghost/g  **加上g为全部替换掉**

:5,12s/God/Ghost/ig 加上i为字符不区分大小写替换

:**%s/God/Ghost/g**  全部替换

(6)复制y 、粘贴p

[n]yy 从光标所在行开始往下复制n行，[n]代表一个整数

**V横选模式：可视化选择(如仅复制一行中的部分内容)按V选中块，可对块删除d、复制f**

**V竖选模式：用于同时操作多行 (如注释多行)**

1)按ctrl+v 选中要注释的行

2)按大写I (或shift+i) , 再输入要添加的字符 (如注释// 、Tab键来对齐)

3)按esc

4)取消时，ctrl+v选中后，按d或x (如用于取消开头的多个空格、注释等)

(7)vim下查看十六进制 ：

:%!xxd 十六进制

:%!xxd -r 返回文本模式

(8)

gg=G 格式化

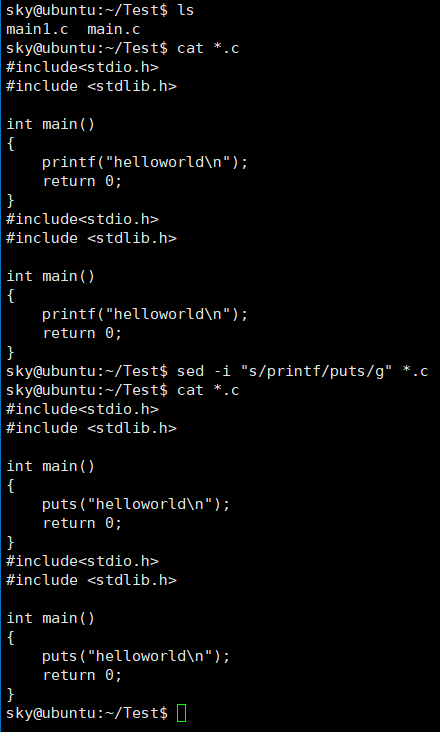
**:set nonu**  取消vim中前面的行号(下次打开仍然有)

在vim中按**ctrl+p** :自动联想

序列化(网络编程)：序列化意为将数据变为字符串

7.用脚本命令sed来替换vim中的内容 (不能进到vim中去)

sed -i “s/printf/puts/g” \*.c 将所有.c文件中的printf替换成puts ：



8.

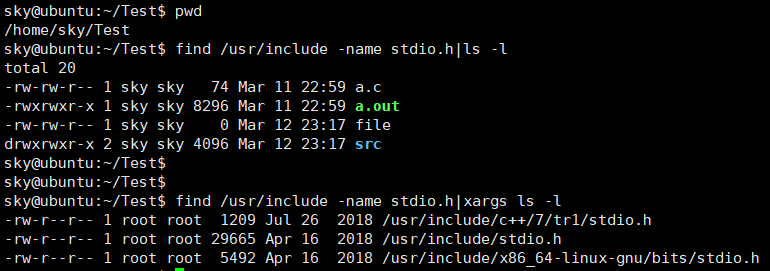
**xargs:代表分别进行(可看作一个for循环)。**

**| 竖杠,代表管道(是重定向的一种)：指前面的输出作为下一个命令的输入。**

**注意：ls -l每次只能看1个目录或文件的信息。**

**find /usr/include/ -name stdio.h|xargs ls -l :若不加xargs则仅仅显示当前目录下的ls -l的详细信息，因为ls -l每次只能看1个目录或文件的信息；**

**要加上xargs才能分别多次输出/usr/include/目录下各个stdio.h文件的详细信息。**



9.

**.vimrc :**每次用vim打开文件时要去读取的配置文件。

**.bashrc :**每次登陆时要去执行的配置文件。

在 .bashrc配置文件中,可以**修改并配置自己的命令：如ll =ls -l** 、

安装Cman 、修改PS1环境变量(修改颜色)

**gcc -c main.c** (用于增量编译)**: 直接生成 .o目标文件**, 实现了预处理、编译、汇编

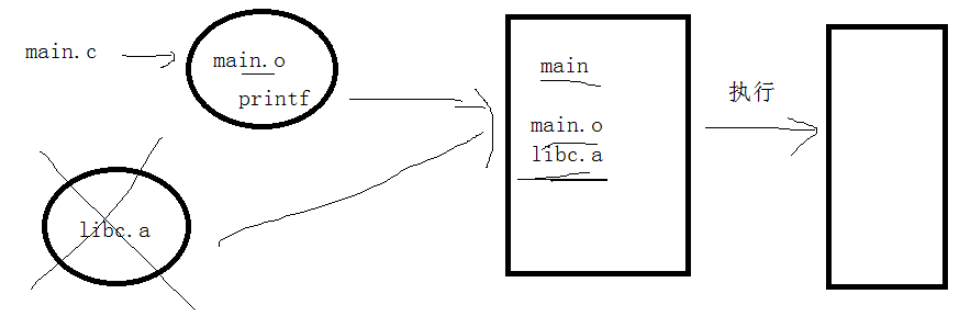
10.动态库和静态库 (重点！！！)

查看可执行程序依赖哪些动态库 命令： **ldd a.out**

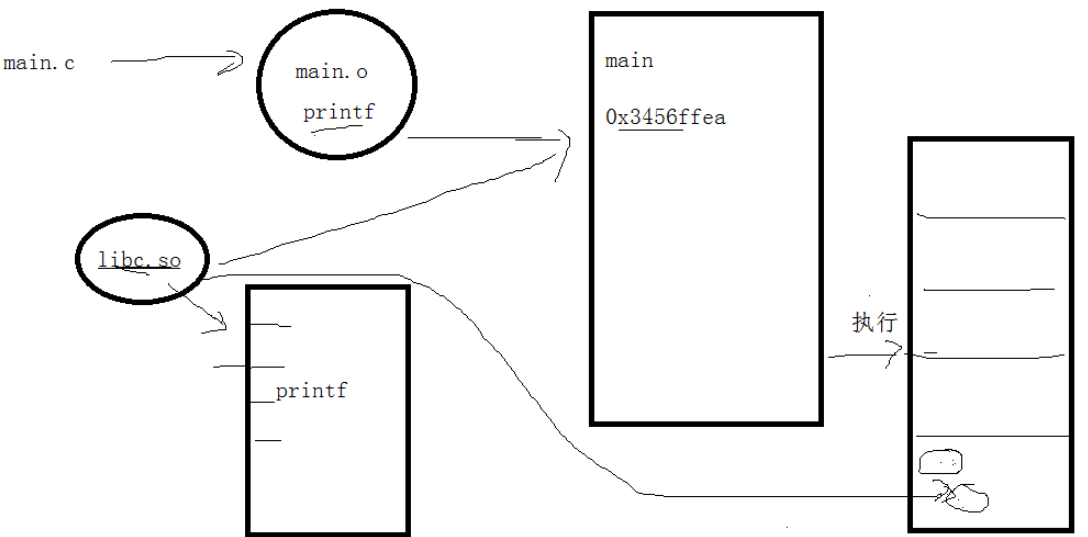
(1)**C库**：**/libc.so动态库**，**且C库是隐式链接**。 (如scanf、printf等函数的实现，均在libc动态库中) gcc编译过程的第四步：链接(硬链接)分为动态链接(动态库)、静态链接(静态库)； 也可分为隐式链接（C库） 、显式链接（自己创建的库、线程库）

(2)

1) 静态库是目标文件**.a**的归档文件（格式为**lib*name*.a**）。在**编译时便链接静态库**，故链接器便会搜索静态库，并将静态库直接拷贝到该程序的可执行二进制文件到当前文件中。 (可执行程序中已经包含了静态库了，故执行时不需要静态库)



2) 动态库（格式为**lib*name*.so[*.***主版本号***.***次版本号***.***发行号**]**）。在**程序编译时并不会被链接到目标代码中，而是在程序运行时才被链接载入**。(仅执行时才需要拷贝动态库)



(3)静态库与动态库的比较：

**动态库只在执行时才被链接使用**，不是直接编译为可执行文件，并且一个动态库可以被多个程序使用故可称为共享库。(编译后可执行程序内不含有动态库文件，执行时才去加载所用的动态库)

**静态库在编译时就会被链接，并把代码整合到程序中，故程序在执行的时候，便不再需要静态库。** 因此，静态库太占用空间，会使你的程序臃肿并且难以升级，但比较容易部署。而动态库会使你的程序轻便易于升级但难以部署。 (编译后可执行程序内含有静态库文件,执行时删掉静态库仍然可以运行，因为代码在编译时便已经加载进去了)

(4) 对于动态库：

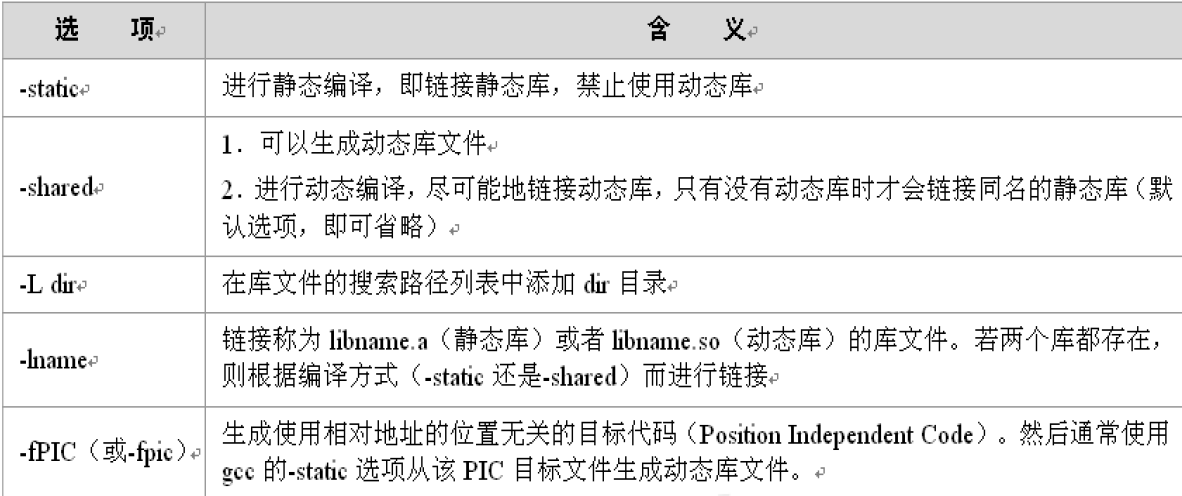
链接之前：在.o文件中printf函数仅仅是一个符号，链接后变为可执行程序，则printf符号便会被替换成一个地址，对应于动态库中printf的相对地址(内容为printf函数的实现)。

动态库、静态库中没有main函数，故没有入口，都是一个一个的API接口，所以只能被链接调用。

可执行程序要被拿到内存中去**执行，即在进程地址空间中的代码段：放了可执行程序a.out和其所需的动态库**。 (记！！！在执行时，才会把动态库拷贝到进程地址空间的代码段中)

(5)**创建动态库、静态库**

若自己写一个函数如add.c接口时，将其链接到静态库或动态库后，就**可以删除此.c文件**了，且删除后仍然可以使用此add接口：因为此函数的实现已经在库中了。



**gcc加上-shared选项，用于生成动态库文件**。 当既有动态库又有静态库时，优先进行动态链接。 只有在没有动态库时才会链接同名字的静态库 (同名是指仅后缀不同，如libc.so、libc.a即为同名的)。

**gcc加上-static选项，用于生成静态库文件**。进行静态链接，禁止使用动态库。

**gcc加上-fpic选项,用于生成 .o目标代码** ，如add.o ，此 .o文件是相对于main.o的地址位置为无关的，即为动态的。

1)创建动态库

$ gcc -fPIC -Wall -c add.c //-c选项：只编译不链接，生成.o目标文件

$ gcc -shared add.o -o libadd.so //生成动态库文件libadd.so

$ sudo cp libadd.so /usr/lib //注册动态库的路径：将库文件拷贝到/lib 或者 /usr/lib下(系统默认搜索库路径模板)：sudo cp lib\*.so usr/lib (**命名库文件时，前面必须加上lib标志**，是库的命名规范)

$ gcc main.c -o main –ladd / /**显式链接名字为add的库文件**，注意区别静态库到此步骤的顺序 (gcc编译的第四步：链接)。

$ ./main

2)创建静态库

$ gcc -c add.c //编译add.c源文件生成add.o目标文件

$ **ar crsv** libadd.a add.o //对目标文件\*.o进行归档，生成lib\*.a静态库文件

//将库文件libadd.a拷贝到/lib或者/usr/lib下（系统默认搜索库路径）

$ sudo cp libadd.a /usr/lib //拷贝到系统路径后再编译(硬链接静态库)

$ gcc main.c -o main –ladd // **-ladd 表示链接库文件libadd.a/.so**

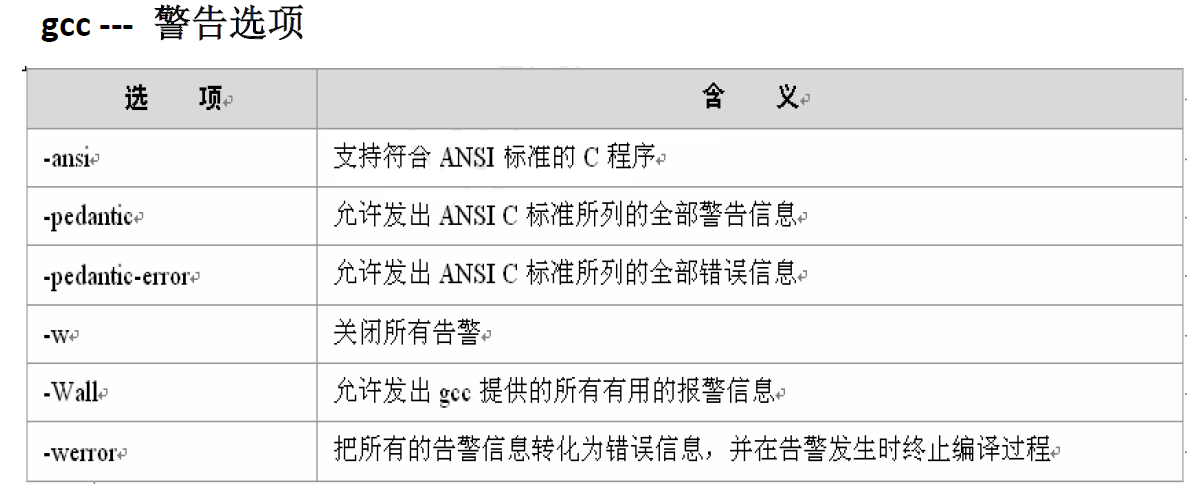
$./main

注意：**gcc编译的第4步链接时，都要加上链接选项：只不过**

**链接库文件分为 隐式链接(不用加选项) 和 显式链接(加-l选项)。**

其中 **C库为隐式链接**， 而 自己创建的库、线程库等 要显式链接，故gcc链接为可执行程序时要加上 -l选项，如上加-ladd

(6)



(7)**头文件和库是什么关系？ (头文件即为声明)**

**头文件是库里所有函数接口、结构体类型等的声明，一个库对应多个头文件。我们也可以把自己写的接口函数放到库(静态库或动态库)中，然后在声明处写成头文件，再在头文件里面写函数的声明。**

11.写程序来**判断是大端还是小端**？

short i=0x1234;

char \*p;

p=(char\*)&i;

if(\*p==0x34)

{

printf("小端");

}else{

printf("大端");

}

12.

注意： **gdb也是一个可执行程序。 为啥a.out执行时要加 ./ ,而gdb执行时不用加？**

因为：直接打a.out搜不到，其要去PATH环境变量中去搜，而当前的可执行程序a.out并没有放到环境变量中去。

**若想不加./就执行a.out: 将pwd当前路径放到环境变量中**即可，即把自己的路径加到环境变量后，还要 :$ 取上之前的环境变量。

如：**如将当前路径 pwd=/home/sky/Test 加入到环境变量：**

**Export PATH=/home/sky/Test:$PATH**

**然后再 echo $PATH查看环境变量,就会发现此路径已经加入到了环境变量中了。**

13. **gdb调试**

**x /4xb &i**  查看内存 //命令x用来检查内存：地址&i后的4个字节、x是16进制、b是以字节为单位来显示。

**bt** ：(BackTrack)查看调用堆栈。

linux中如何刷新输入缓冲区，利用getchar()函数即可。

刷新输出缓冲区可以利用fflush(stdout);

可在程序编译时用 **gcc -g 选项: 打开gdb调试选项** ,gdb是在程序的编译时期运行的 (查看.s文件会发现多了很多微指令，就是gdb的)

常见的调试程序的步骤如下：

gcc –o filename –Wall filename.c –g //编译一定要加**-g**

gdb filename //进入调试

l //显示代码 (list)

b 4 //在第四行设置断点 (break)

r //运行 (run)

n //下一步，且并不会进入函数 (next)

s //表示单步进入函数 (step)

p I //打印变量I (print)

c //运行到最后 (continue)

q //退出 (quit)

14. 程序崩溃了怎么办：用gdb调试core文件

**程序崩溃后会产生一个文件：core文件(即黑匣子)：Segmentation fault(core dumped)**

core的意思是：内存，dump的意思是：扔出来、堆出来。core文件的生成，便是操作系统把程序down掉时的内存的内容扔出来生成的，它可以做为调试程序的参考。core dump又叫核心转储，当程序运行过程中发生异常，程序异常退出时，由操作系统把程序当前的内存状况存储在一个core文件中，叫core dump。

1、**ulimit -c unlimited**  //用于设置core file size为不限制大小

//ulimit资源控制：用于限制用户资源

2、**程序崩溃后，加gcc -g main.c**

3、**再次运行程序 ./a.out ,产生core文件**

4、**gdb a.out core** //崩溃现场回归：执行程序core文件

5、**bt**  //查看崩溃问题，离自己写的代码最近的一行。

注意：(1)第二步可通过ulimit -a来查看是否设置好了。

(2)第四步可通过bt查看程序段错误时的现场。

(3)若现场已经被破坏了，则不能再被core定位。

(4)一般多进程、多线程出了问题，要通过看日志来解决，当问题被缩小到一定范围后再通过gdb来调试core.

15. make工程管理器： Makefile

(1)最简单的Makefile格式：

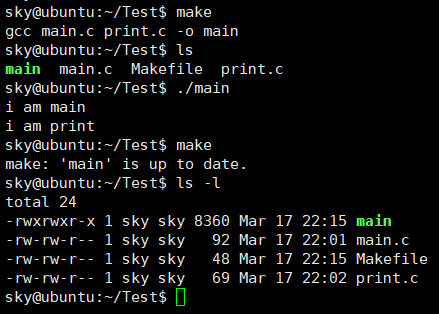
target: dependency\_files //目标项:依赖项

command //第二行前面必须空出一个tab键，而不能为4个空格

例如：1)效果如下图所示

**main:main.c print.c**

**gcc main.c print.c -o main**



2)优化：缩短编译时间，即分开写依赖项，检查各自的目标项和依赖项的时间。

**main:main.o print.o**

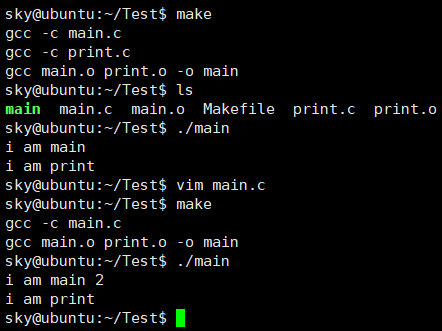
**gcc main.o print.o -o main**

**main.o:main.c**

**gcc -c main.c**

**print.o:print.c**

**gcc -c print.c**



Makefile可实现增量编译：会自动检查文件的更新时间，若发现某个.c文件比.o文件新，则只会将此.c文件变为.o文件，再把所有.o文件链接起来 (比较的是目标项和依赖项的时间)。 若**只改变了main.c，则make只会重新编译main.c后再链接到main, 而print.o并不会发生改变**，是因为：**Makefile只会比较 目标项main.o 和 依赖项main.c 的产生时间，且目标项比依赖项时间旧的话，才会去编译执行下面一条的命令！！！！！**

(2)特殊处理和伪目标：

.PHONY 是 Makefile文件的关键字，表示它后面列表中的目标均为伪目标。

伪目标：指其后面没有依赖项，即为假的目标项。伪目标通常用在清理文件、强制重新编译等情况下。

**main:main.o print.o**

**gcc main.o print.o -o main**

**main.o:main.c**

**gcc -c main.c**

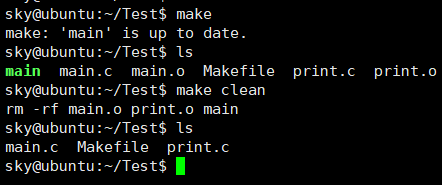
**print.o:print.c**

**gcc -c print.c**

**.PHONY:clean //此句可以省略，只要clean后面没东西，就默认是伪目标。**

**clean:**

**rm -rf main.o print.o main**



如果不用**系统默认的文件名Makefile**，而是用户随便起的一个名字，

如：#vim Makefile11 ,则make 后面必须要加上 -f Makefile11 ，如：

#make –f Makefile11 clean //表示执行clean: 开始的命令段

#make –f Makefile11 main //表示执行main: 开始的命令段

(3)通用性很强的Makefile :变量、函数与规则

随着软件项目的变大、变复杂，源文件也越来越多，如果采用前面的方式写makefile文件，将会使makefile也变得复杂而难于维护。

通过make支持的变量定义、规则和内置函数，可以写出通用性较强的makefile文件，使得同一个makefile文件能适应不同的项目: 多个.c编译成一个可执行程序a.out

变量分为：用户自定义变量，预定义变量（**CFLAGS**），自动变量，环境变量。

1)自定义变量

**变量：用来代替一个文本字符串**，**定义变量(自定义变量)**的2种方法：

变量名=变量值 递规变量展开(几个变量共享一个值) //不常用

变量名:=变量值 简单变量展开(类似于C++的赋值) //**用冒号等号 := 进行赋值**

**使用变量**的格式 **$(变量名)**

例如：

**OBJS:=main.o print.o //定义变量OBJS(自定义变量)，并赋值为main.o print.o**

**ELF:=main //定义变量EXE(自定义变量)，并赋值为main**

**$(ELF):$(OBJS)**

**gcc $(OBJS) -o $(ELF)**

**main.o:main.c**

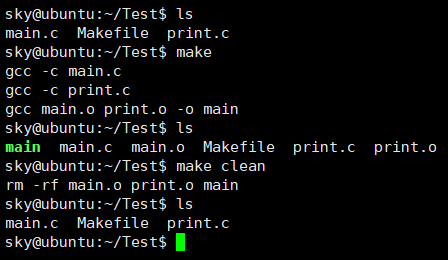
**gcc -c main.c**

**print.o:print.c**

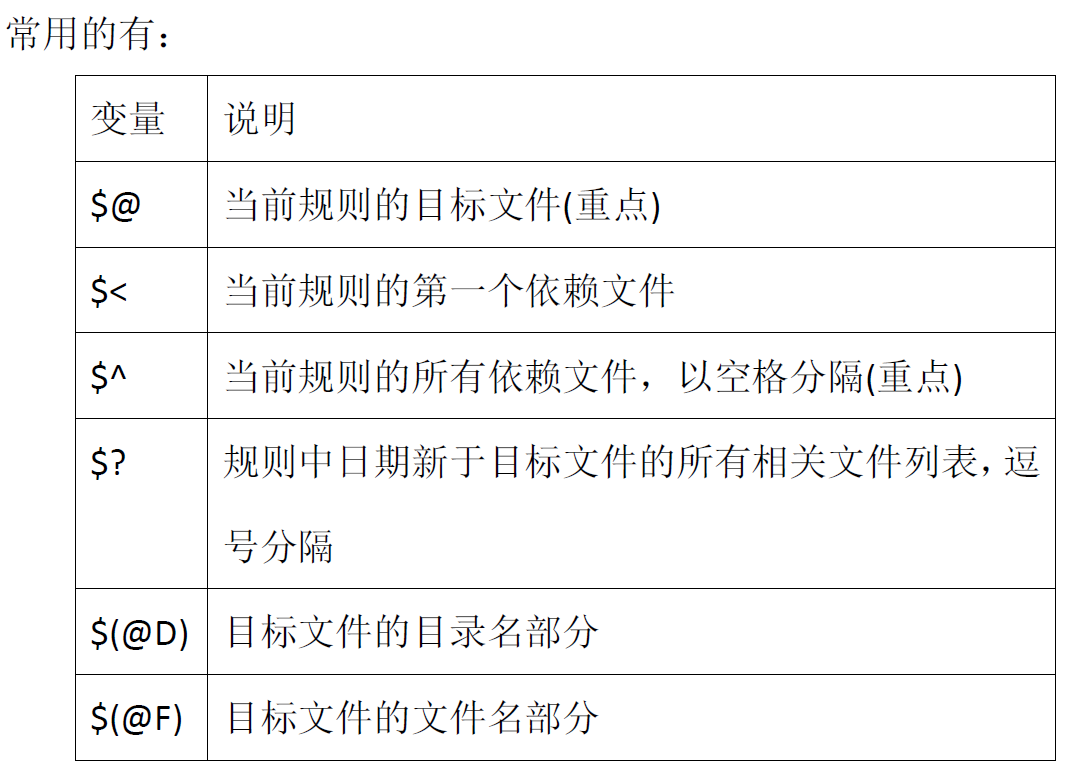
**gcc -c print.c**

**clean:**

**rm -rf $(OBJS) $(ELF)**



2)自动变量：指在使用的时候，自动用特定的值替换。



主要掌握 第一个$@ 和 第三个$^ ， 例如：

**OBJS:=main.o print.o**

**ELF:=main**

**$(ELF):$(OBJS)**

**gcc $^ -o $@ //仅指当前规则( $(ELF):$(OBJS) )下的所有依赖文件和目标文件**

**main.o:main.c**

**gcc -c $^ //$^仅指当前规则( main.o:main.c )下的所有依赖文件(可以不写)**

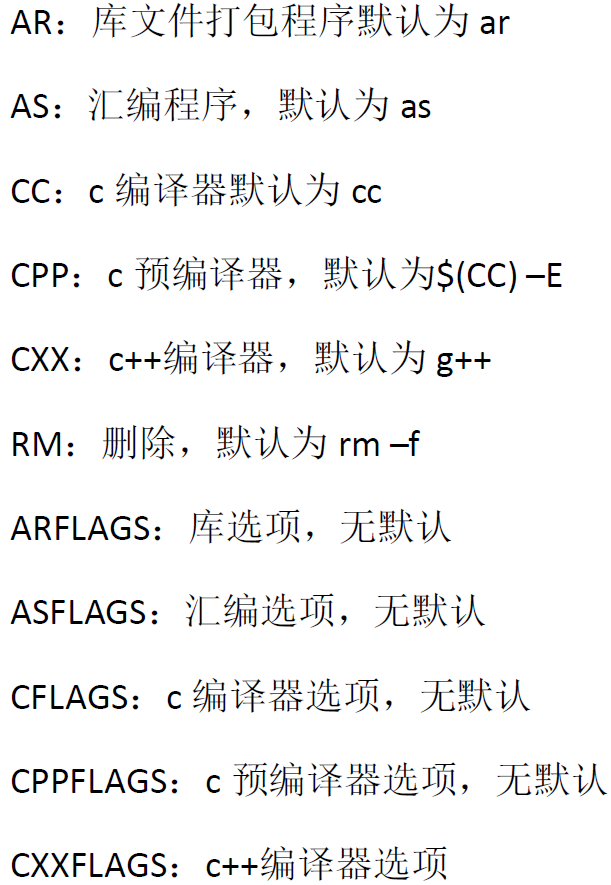
**print.o:print.c**

**gcc -c $^ //$^仅指当前规则( print.o:print.c )下的所有依赖文件(可以不写)**

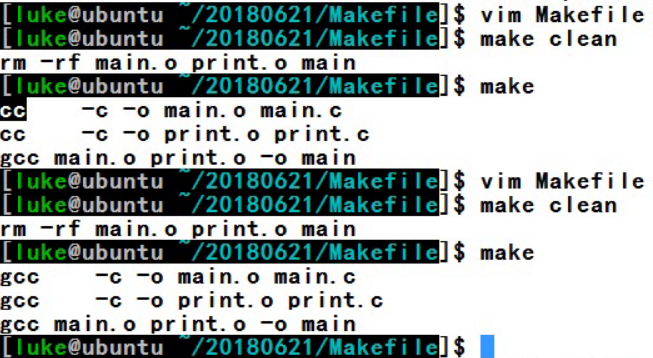
**clean:**

**rm -rf $(OBJS) $(ELF)**

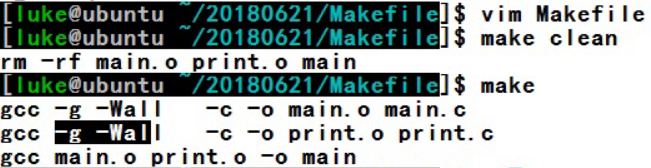
3)预定义变量：内部事先定义好的变量，但是它的值是固定的，并且有些值是为空的：变量名不可改变，只能给其赋值。作用是将隐式规则里默认的值给替换掉。



例如：将cc改变为gcc



gcc加上-g -Wall选项：



**OBJS:=main.o print.o**

**ELF:=main**

**CC:=gcc**

**CFLAGS:=-g -Wall**

**$(ELF):$(OBJS)**

**gcc $^ -o $@ //隐含规则：main.o和print.o 可以缺省不写。**

**clean:**

**rm -rf $(OBJS) $(ELF)**

隐含规则： **\*.o文件 自动依赖\*.c或\*.cc文件，所以可以省略**。

这是因为**当依赖的文件为.o文件时，Makefile会自动把对应名字.c文件变为.o文件**

4) 规则分为：普通规则，隐含规则，模式规则

普通规则：给1)中的变量赋值。 隐含规则：如上makefile中可省略.o文件。

模式规则：通过匹配模式找字符串，**%百分号表示匹配1或多个任意字符串。**

**%.o: %.c**  是指:任何目标文件的依赖文件，是与目标文件同名的并且扩展名为.c的文件

5)函数：

1. wildcard 搜索当前目录下的文件名，展开成一列所有符合由其参数描述的文件名，文件间以空格间隔。**SOURCES = $(wildcard \*.c)**，指定义一个变量SOURCES，并**把当前目录下所有.c文件存入变量 SOURCES** 中。

2. 字符串替换函数：**$(patsubst要查找的子串,替换后的目标子串，源字符串)**。

意为：将源字符串(以空格分隔)中的所有要查找的子串替换成目标子串。

如OBJS = $(patsubst %.c,%.o,$(SOURCES)) ,即是把SOURCES中'.c' 替换为'.o' 。

3. $(addprefix 前缀，源字符串)函数把第二个参数列表的每一项前缀上第一个参数值

1. **通用Makefile : 多个.c编译成一个可执行程序。**

**SOURCES:=$(wildcard \*.c)**

**OBJS:=$(patsubst %.c,%.o,$(SOURCES)) //返回值再赋给OBJS**

**ELF:=main**

**CC:=gcc**

**CFLAGS:=-Wall**

**$(ELF):$(OBJS)**

**gcc $^ -o $@**

**clean:**

**rm -rf $(OBJS) $(ELF)**

1. **通用Makefile :编译成多个可执行程序。采用for循环编译多个目标文件。**

**SRCS:=$(wildcard \*.c)**

**TARGETS:=$(SRCS:%.c=%)**

**CC:=gcc**

**CFLAGS:=-Wall**

**all:$(TARGETS)**

**@for i in $(TARGETS);do gcc -o $${i} $${i}.c;done**

**clean:**

**rm $(TARGETS)**

16.Linux下的文件操作(Linux下一切皆文件)

(1)文件分为普通文件，目录文件，链接文件和设备文件等。

**- ：普通文件**: 也称磁盘文件，并且能够进行随机的数据存储(能够自由seek定位到某一个位置)；dirent结构体中，普通文件的类型：8

**d ：目录文件**: 它包含了保存在目录中文件列表的简单文件。dirent结构体中，目录的文件类型是：4

**l ：符号链接文件**: 类似于Windows 的快捷方式，指包含到达另一个文件路径的文件。

设备文件分为块设备和字符设备：提供了大多数物理设备的接口(搞嵌入式的要重点研究)

**b ：块设备文件**: 必须以一定大小的块来读出或者写入数据，块设备包括CD-ROM、RAM 驱动器和磁盘驱动器等。一般用于**存储数据**。

**c ：字符设备文件**：一次只能读出和写入一个字节的数据，包括调制解调器、终端屏幕、打印机、声卡以及鼠标。一般用于**传输数据**。

**p ：命名管道文件**: 是一个从一端发送数据，另一端接收数据的数据通道；

**s ：socket文件**：在Linux 中,套接字也可以当作文件来进行处理。基于文件指针的文件操作函数是ANSI 标准函数库的一部分。

(2)



Windows下文件打开方式有12种模式，Linux下只有6种模式。

因为Linux下的r+和rb+是一样的，且写入换行\n，存储的仍是\n (1个字节)。

其中 **ab和ab+ 为追加模式**

在此两种模式下，无论文件读写点定位到何处，在**写数据时都将是在文件末尾添加**，所以比较适合于**多进程写同一个文件**的情况下保证数据的完整性。(但现在更多都是并发操作内存，而不是并发操作磁盘，即内存写好后，再一并放入磁盘：故ab+渐渐退出了时代)

注意 ab+ 与其他模式的区别：仅在写文件时

即每次一旦往文件内写入时，位置指针会先偏移到文件尾后(操作磁盘)，再去写

17.**目录操作(目录指针dir)**: 不在C的标准库中，只能在Linux下使用。 注意：目录和文件在磁盘上的操作原理并不一样！！！故需要学习一套额外的接口。

1) 改变目录或文件的访问权限

#include <sys/stat.h>

**int chmod(const char\* path, mode\_t mode);**

path路径参数指定的文件, 被修改为具有mode 参数给出的访问权限。

2) 获取、改变当前目录：

#include <unistd.h> //头文件

**char \*getcwd(char \*buf, size\_t size);** //获取当前目录，pwd 命令的封装(env可查)

**int chdir(const char \*path);** //修改当前目录，即切换目录，cd 命令的封装。(其改变的是当前进程的路径！！！)

实现原理：

getcwd()函数：将当前的工作目录绝对路径复制到 buf所指的内存空间中，参数size为 buf的空间大小. 在调用此函数时，buf 所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数size 的大小，则返回值NULL， errno 的值则为ERANGE。

倘若参数**buf 为 NULL**，getcwd()会依参数size 大小自动配置内存(使用malloc())，如果参数**size 也为0**，则getcwd()会依工作目录绝对路径的字符串程度来决定所配置的内存大小，进程可以在使用完此字符串后自动利用free()来释放此空间。所以**常用的形式：getcwd(NULL, 0);**  注意：由中间层接口申请的空间，不需要我们自己去free。

chdir()函数：用来将当前的工作目录改变成以参数path 所指的目录。

3)创建和删除目录：

#include <sys/stat.h> #include <sys/types.h> #include <unistd.h>

**int mkdir(const char \*pathname, mode\_t mode);** //创建目录,mode 是目录权限

**int rmdir(const char \*pathname);** //删除目录

4)获取目录信息： #include <sys/types.h> ；#include <dirent.h>

**DIR \*opendir(const char \*name);** //打开参数name 指定的目录，并**返回目录指针dir**，即DIR\*形态的目录流，接下来对目录的读取和搜索都要使用此返回值。函数失败则返回NULL。

**struct dirent \*readdir(DIR \*dir);** //读取目录的信息，并返回一个结构体指针，该指针保存了目录的相关信息。有错误发生或者读取到目录文件尾则返回NULL。注意：**目录之间地址不是连续的，会自动偏移到下一个地址(因为有dirent结构体中d\_off)**

**void rewinddir(DIR \*dir);** //重新定位到目录文件的头部

**void seekdir(DIR \*dir,off\_t offset);** //类似标准C的fseek(),即设置目录流当前的读取位置，再调用readdir函数时，便可从此新位置开始读取。 参数offset代表距离目录文件开头的偏移量：**此offset一定要是telldir()返回来的值**。

**off\_t telldir(DIR \*dir);**  //返回目录流当前的读取位置,类似标准C的ftell()

**int closedir(DIR \*dir);** //关闭目录文件

5) readdir的返回值类型，**dirent 结构体**如下：

struct dirent{

ino\_t d\_ino; // 此目录进入点的inode

off\_t d\_off; //目录开头到进入点的位移:磁盘偏移 (不是连续的)

unsigned short d\_reclen; //目录名的长度(仅指下面**文件名本身的长度**):**可变长结构体**

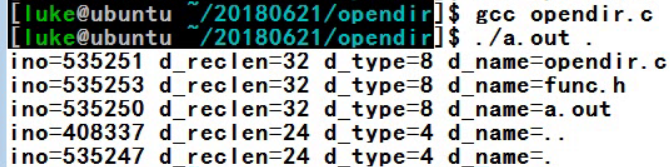
unsigned char d\_type; //所指的文件类型，注意：**打印时要用%d** (因为Linux下文件

最多有16种类型，而char能存储数字范围-127 ~ 126，此处的char存的是整型数)

char d\_name[256]; //文件名 (可知文件名是存储在目录中的！！！)

}; //stat中并没有存储文件名，**文件名是存储在目录中的(即dirent中)**。

目录信息如下图：



(1)目录为什么不用连续的存储结构,而是用指针来记录偏移？

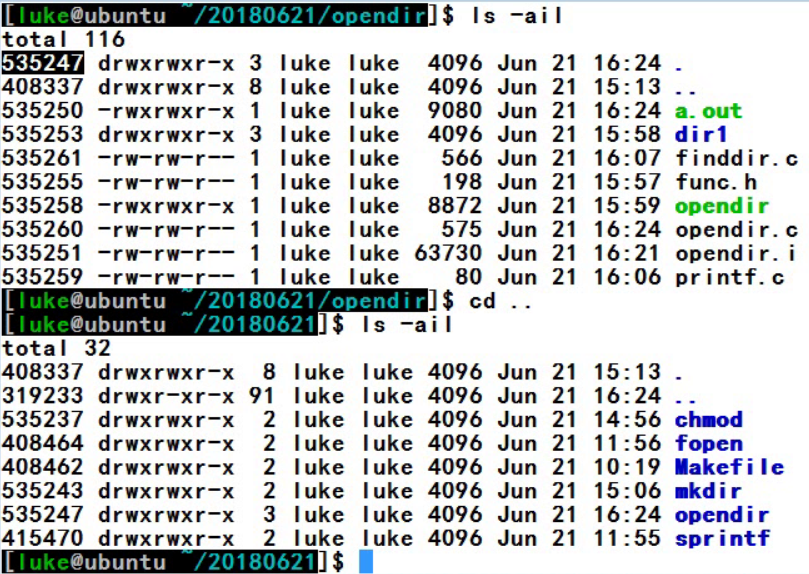
防止碎片化太多(两个连续的空间之间的缝隙大小，并不能放不下新增的文件，故成了碎片)，不利于增删，为了更高效地利用空间。

(2)给文件名opendir.c 、func.h 、a.out开辟的空间是32字节，给隐藏文件开辟空间是24字节。但是文件名并没有那么长，为什么还要开辟这么大的空间？？？

因为并不是只开辟刚好能容下文件名长度大小的空间，而是冗余设计，空间换时间，dirent结构体中的成员d\_reclen是变长结构体。

(3)为什么能通过cd .和cd ..到达当前目录和上一级目录？

是因为记录了d\_ino的信息**，查看d\_ino命令：ls -i**：



**磁盘上存储的每一份文件，都对应着唯一的inode**

**信息**。如上图，上一级目录中的opendir目录和当前 “.”目录的**d\_ino值相同，说明它俩在磁盘上指向同一份文件**。

图中每行第三项的数目：硬链接数目：即有此数目个文件名，对应到第一项的inode上。

如上图 ，535247有3个硬链接，即对应3个文件名：

当前目录的“**.**” ； 上一级目录的“**opendir**” ； 当前目录的子目录dir1中的 “**..**”

5) 对目录实现深度优先遍历、广度优先遍历

**可通过 %\*s 来控制空格数**

例如：printf(“%\*s%s\n”,5,” ”,”hello”); //在hello前面打印5个空格

(1)**深度优先遍历**：类似tree命令的效果 (通过递归)

#include "func.h"

void printdir(char \*path,int width)

{

DIR \*dir;

dir=opendir(path);

if(!dir)

{

perror("opendir");

return;

}

struct dirent \*p;

char buf[512]={0};

while((p=readdir(dir))!=NULL)

{

if(!strcmp(p->d\_name,".")||!strcmp(p->d\_name,".."))

{ //若目录文件为"."或者"..",则什么也不做

}else{

printf("%\*s%s\n",width,"",p->d\_name); //可用%\*s来控制空格数

if(p->d\_type==4) //若是非目录文件则不递归，目录文件类型为4

{

sprintf(buf,"%s%s%s",path,"/",p->d\_name); //路径(字符串)拼接：sprintf

printdir(buf,width+4);

}

}

}

closedir(dir);

}

int main(int argc,char\* argv[])

{

args\_check(argc,2);

puts(argv[1]);

printdir(argv[1],4);

return 0;

}

1. **广度优先遍历：**层序遍历,让每个文件作为一个结点，建立链表，再去遍历链表、

建立链表。(类似通过链表进行二叉树的层次建树)

#include"func.h"

typedef struct file

{

char name[512];

char path[512]; //记录此节点的所在路径(关键！！！)

unsigned char file\_type;

struct file \*next;

}node,\*p\_node;

void bfs(char \*path,p\_node \*ptail)

{

DIR \*dir;

struct dirent \*p;

p\_node pnew=NULL;

dir=opendir(path);

if(dir==NULL)

{

perror("opendir");

return;

}

while((p=readdir(dir))!=NULL)

{

if(!strcmp(p->d\_name,".")||!strcmp(p->d\_name,".."))

{

}

else

{

pnew=(p\_node)calloc(1,sizeof(node));

strcpy(pnew->name,p->d\_name);

strcpy(pnew->path,path);

pnew->file\_type=p->d\_type;

(\*ptail)->next=pnew;

\*ptail=pnew;

}

}

}

void list\_create(char \*path)

{

DIR \*dir;

char buf[600]={0};

struct dirent \*p;

p\_node phead=NULL,ptail=NULL,pnew=NULL,pcur=NULL;

dir=opendir(path);

if(dir==NULL)

{

perror("opendir");

return;

}

while((p=readdir(dir))!=NULL)

{

if(!strcmp(p->d\_name,".")||!strcmp(p->d\_name,".."))

{

}

else

{

pnew=(p\_node)calloc(1,sizeof(node));

strcpy(pnew->name,p->d\_name);

strcpy(pnew->path,path);

pnew->file\_type=p->d\_type;

if(phead==NULL)

{

phead=pnew;

ptail=pnew;

}

else

{

ptail->next=pnew;

ptail=pnew;

}

}

}

pcur=phead;

while(pcur)

{

printf("%s\n",pcur->name);

if(pcur->file\_type==4)

{

sprintf(buf,"%s%s%s",pcur->path,"/",pcur->name);

bfs(buf,&ptail);

}

pcur=pcur->next;

}

}

int main(int argc,char \*argv[])

{

puts(argv[1]);

check\_args(argc,2);

list\_create(argv[1]);

return 0;

}

19. 获取文件状态信息： **int stat(const char \*pathname, struct stat \*buf);**

注意: (1)pathname路径可以是任何文件，也包括目录，其均有stat信息。

且inode和stat是一一对应的，一个inode只会存一个stat信息。

(2)**当buf为指针形式时：意为其指向的空间是要由我们自己来申请malloc空间的，而不是由此接口来负责申请空间，术语也叫传出参数。** 文件状态信息会放到buf中。

(3)文件内容、文件状态信息、目录信息：分开在磁盘的3个不同的地方存储的。

(4)**stat存的是inode，而并没有存储文件名；文件名是存储在目录中的(即dirent中)**。

**stat结构体**的定义为：

struct stat 16

{

dev\_t st\_dev; // 如果是设备，返回设备表述符，否则为0

ino\_t st\_ino; // inode值

mode\_t st\_mode; // 文件的类型，**用十六进制 %x来打印**：其中16进制数的

**低9位是权限、最高的4位是文件类型**(文件类型只有0 ~ 15)

nlink\_t st\_nlink; // 硬链接数

uid\_t st\_uid; // 属主ID

gid\_t st\_gid; // 组ID

dev\_t st\_rdev; // 设备类型

off\_t st\_size; // 文件大小，字节表示

blksize\_t st\_blksize; // 块大小

blkcnt\_t st\_blocks; // 块数

time\_t st\_atime; // 最后访问时间

time\_t st\_mtime; // 最后修改时间

time\_t st\_ctime; // 最后权限修改时间

};

20.**man接口时要加上参数,且一般参数为2的 man 2 ：都是直接封装的系统调用，很少在系统调用之上做功能性的修改。**

如：**man chmod: man的是命令** ，此命令封装了chmod() 接口函数。

**man 2 chmod: man的是接口**。

chmod 777 file 777是10进制、 **0666** 是8进制

**如何看接口是否为直接封装的系统调用？** 看此**接口有没有操作到硬件上的信息**。

如chmod改变文件的权限，而此文件是存在磁盘(硬件)上的。

在系统上改变文件权限都用chomd()接口，而不会去用chmod命令！！！

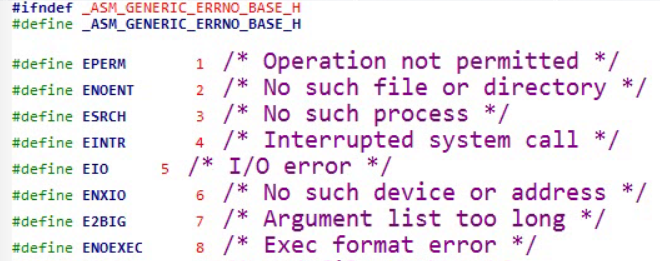
这是因为命令就相当于在C语言中嵌套脚本：会降低性能。

**目录信息：dirent**信息(通过**readdir()** )； **文件状态信息：inode**信息(通过**stat()** )

21. perror实现原理：

当函数执行失败时，便会**把失败的原因写到一个全局变量errno(整型数)**中去。然后perror会去读取errno错误码，再打印出错误码对应 #define的错误原因。

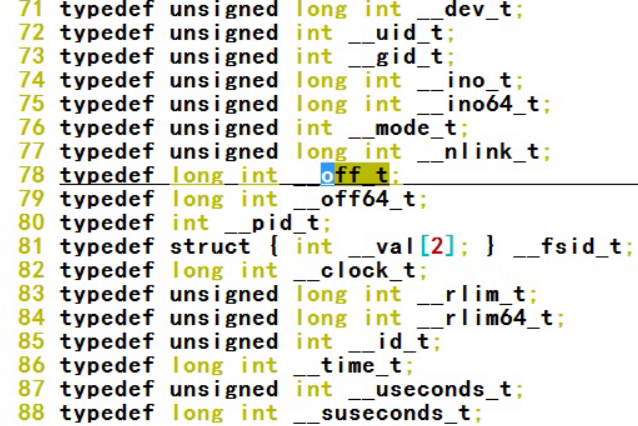
如下图：



22.

1)学生管理系统技巧：可把**perror写成一个宏判断**，这样就可以用goto了(若把perror写成一个接口，则不能使用goto了，因为其不在同一个 .c中,故没法goto到end)。

2)**查看typedef的数据类型：可通过gcc -E main.c -o main.i再用vim查看 .i编译文件来看**。如下图可知：off\_t 是long int类型。



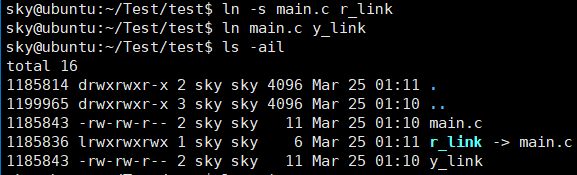
3) **重点！！！！**区别软链接和硬链接：**软链接源文件不可删除 ；硬链接源文件可删除**

**软链接文件的大小和创建时间和源文件的大小不同**；

**硬链接文件和源文件的大小和创建时间一样**；

软链接和源文件的inode值不同；

硬链接和源文件的inode值相同，且它们的 **硬链接数(即引用计数)**相同，且均对应到磁盘上的同一个位置。**只有inode的引用计数(硬链接数)为0时，inode对应的磁盘空间才可以被其他人使用。**

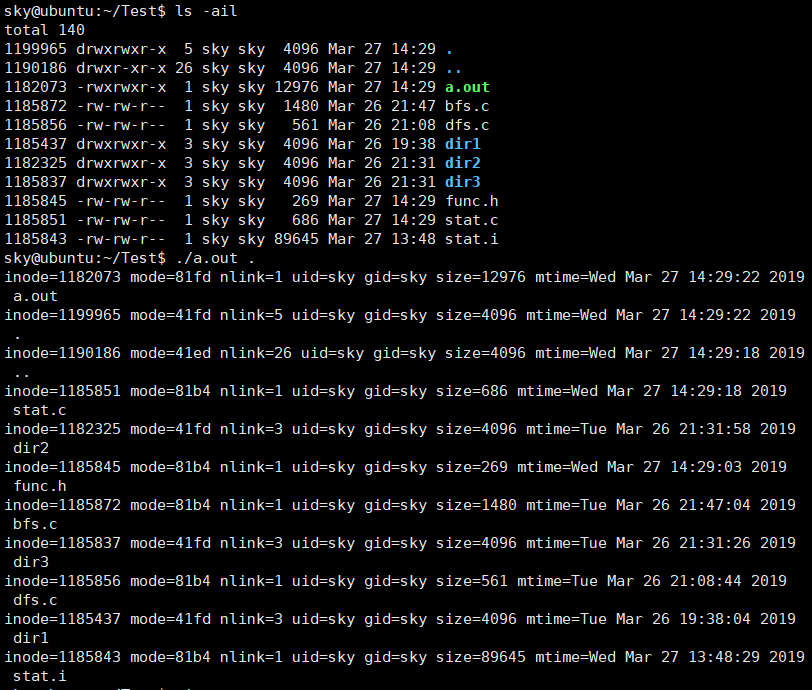


软链接和硬链接的更新之间的区别 ？？？

软链接：操作软链接文件，**相当于操作源文件**。

硬链接：inode值相同，对任一个文件的修改，**都是修改inode对应的磁盘信息**，故能实现同步更新。

23.自己实现 ls -ail



(1) **struct passwd \*getpwuid(uid\_t uid)**;一般用于转换uid(整型数)为文件所有者(字符)

(2) **struct group \*getgrgid(gid\_t gid)**;一般用于转换gid(整型数)为文件所属组(字符)

(3) **char \*ctime(const time\_t \*timep)**; 用于把1970年1月1日到此刻的秒数(长整型)转换为日历时间(字符)，并且会打印换行。

(4)stat结构体成员st\_mode：

mode\_t st\_mode; 要用十六进制 %x来打印：其中16进制数的低9位是权限，最高的4位是文件类型(文件类型只有0 ~ 15)，如上图 a.out文件。

上图a.out的文件类型为：81fd ,二进制为：1000 0001 1111 1101,其中

高4位：1000为文件类型，十进制为8，代表是普通文件。

低9位：1 1111 1101，即111 111 101：rwx rwx r-x,即775权限。

24. 文件的删除、格式化原理：仅删除了 dirent 和 inode(引用计数)信息。

Linux的文件系统是ext4系统。 Why删除一份文件很快，拷贝一份文件很慢？？？

删除文件：若删除的是目录：仅删除了目录的dirent信息；

若删除的是普通文件：仅删除了文件的inode信息和在目录中存的dirent信息。

此时硬链接数为0了，就看不到对应到磁盘上的内容了，但不是真正的磁盘删除。只有当新文件放入磁盘时，才会真正覆盖掉对应的文件内容。

故市面上的恢复工具：都是利用本身dirent、inode信息都有的备份，来恢复链接。

25. MMAP 文件映射(零copy):大数据框架使用的主要手段

(1) #include <sys/mman.h>: 将文件映射到内存(不经过内核态) 磁盘文件的高效操作。

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

//返回的指针，即为映射到堆内存的起始地址值。 注意：要强转为自己的类型。

//在**文件很大**，且**读写频繁**的情况下用mmap，可以高效操作文件。

mmap参数：

**addr**: 要映射的堆内存的地址,一般写NULL，由操作系统来分配。

**length**： 映射的大小。最好是4K的整数倍，因为内存都是以叶来管理的。

**prot**： 内存的权限：读、写 ：PROT\_READ(读) 、PROT\_WRITE(写)： 一般是 按位或|

**flags**: 是否可以共享，即映射进来的数据能否被其他进程访问(每个进程都有各自的进程地址空间0 ~ 4G)：MAP\_SHARED(可共享)、MAP\_PRIVATE(不可共享)

**fd**： 文件描述符，即要知道操作哪一个文件。

**offset**： 偏移的大小(必须是4K的整数倍)，即文件从哪里开始进行映射 (可实现：只映射文件的一部分)。 注意mmap偏移只能用offset参数来设置，不能用lseek，因为lseek是偏移缓冲区的，而mmap并不经过缓冲区；若文件不偏移：从头开始映射，offset设为0

**int munmap(void \*addr, size\_t length);** //解除映射(类似free() )

注意：addr必须为mmap返回的指针。

length大小必须和mmap中的length参数设置的大小一样。 成功返回0，失败返回-1。

msync：可以将mmap映射地址的 一部分内存 先回写到磁盘上（而解除映射必须全部回写）

1. 和read、write区别：**只能操作固定大小的文件**,其没有改变文件大小的功能。

但比read、write效率更高，用于：磁盘文件内存映射(并不经过缓冲区)。

1) read、write: 多了一次搬移，读取是先从磁盘读到内核的输入缓冲区中，再到用户态自己定义的栈中； 写入也是从用户态写到内核的输出缓冲区，再写到磁盘。注意：**用户态(栈、堆、数据段) 和 内核态(内核)之间搬移数据，花费的开销是很大的。**

2) **Mmap(零拷贝：磁盘文件映射)**：会直接把数据从磁盘搬移到内存中，即内核态 和 用户态 之间没有发生拷贝，即零拷贝。 mmap会把要操作的磁盘上的文件的其大小，在堆内存上映射同样大小的空间，并把这个空间的地址返回回来， **操作这一段的堆内存就相当于操作对应的磁盘空间一样**。等操作完后，再**munmap解除映射，就会把内存上的这段信息回写到对应的磁盘上面**。

3） mmap通常映射的大文件：如优酷看电影，很多人要去读磁盘（其实读的是内存）

（1）先把磁盘文件mmap映射到内存中，再通过共享内存，多个进程访问这个内存，故有很高的效率。 （2）不会真正把1G电影全都搬移到内存中，而是lazy模式：即真正要去读哪一段时，再由DMA驱动器把数据搬移到内存中。

(3)由DMA(磁盘上的芯片)磁盘驱动器来进行映射，要映射多少CPU都提前分配好了，映射过去的空间大小都是固定的，所以无法改变文件大小。

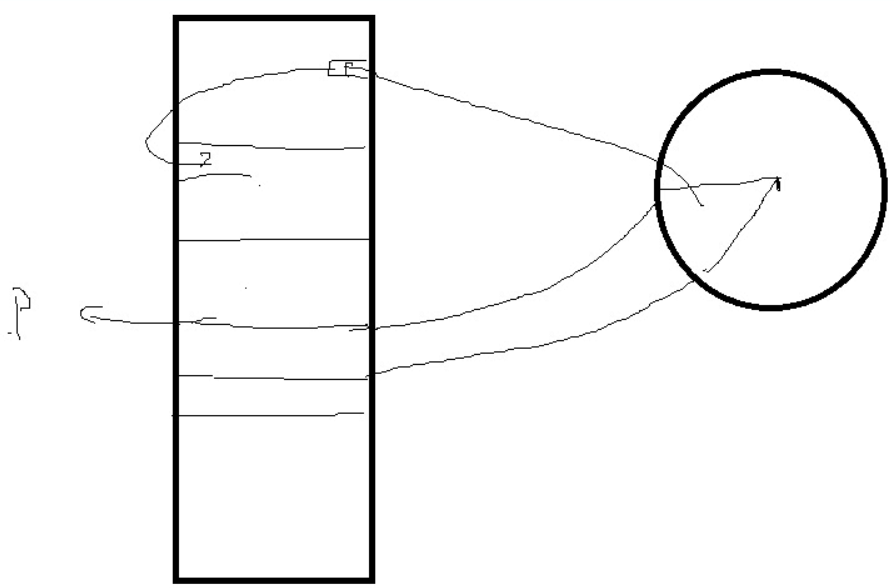
注意：mmap映射的都是虚拟地址空间。

(4)如下图：**mmap与read、write的区别**， mmap不经过缓冲区。向文件写内容时，文件必须是有大小的。因为mmap映射的文件必须有大小，且大小必须是4K的整数倍，所以学的ftruncate、lseek(实现文件空洞)就是为mmap做铺垫，高效操作文件用的。

（5）mmap内存映射的实现过程，总的来说可以分为三个阶段：

（1）进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域； （2）调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系； （3）进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝；

<https://blog.csdn.net/Holy_666/article/details/86532671>



26. Linux特有的：管道文件, **只能用于本机的进程间通信**，不能用于网络之间通信！！！

(1)

创建管道文件命令：**mkfifo**  //文件类型以p开头

管道文件是用于通信的，故不能用vim打开。

(2)

管道是内核中的一段缓冲区：ulimit -a 可以查看pipe size：缓冲区(管道)的大小。

**缓冲区为空时，读端会卡住，等着写端写入数据。**

**管道是内核中的一段缓冲区(管道类似队列：先进先出)，两个进程都同时可见此缓冲区。**

(3)

管道文件open打开时，必须以 只读 或者 只写 的方式(半双工的)，不能用可读可写方式。

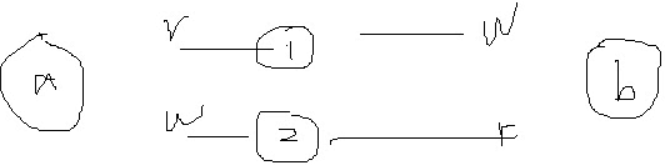
通信方式：3种

全双工：两端可以任意一端到另一端。如TCP通信。

半双工：一段时间内，数据只能从一端到另外一端。 如管道通信。

单工：固定了读写端，只能从一端到另外一端。

(4)双方都能通信：**建立两条管道，要防止死锁(管道两端方式 均是读 或 均是写)**



一条管道： a以读的方式打开管道1，以写的方式打开管道2

另一条管道： b以写的方式打开管道1，以读的方式打开管道2

课上演示的，只能a来写b去读，再b去写a去读，如此反复...并不能在a、b之间任意读写，会发生阻塞，要想实现任意读写：用多路复用select。

27. I/O 多路复用select（**阻塞**函数）： 可实现两边任意读写。

(1) I/O 多路转接模型：在这种模型下，如果请求的I/O 操作阻塞 (且其不是真正阻塞I/O， 而是让其中的一个函数等待)，则在这期间， I/O 还可以去进行其他的操作，如select。

**select系统调用：可以去监控哪一个 描述符可读了(即描述符对应的缓冲区中有数据了)； 单个进程的select默认能监控的描述符个数最大为：1024个，即select不能去监听大量的文件描述符 （每次select都在内核中轮询描述符）**

1. #include <sys/select.h> #include <sys/time.h>

**int select(int maxfd, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptionset, const struct timeval \*timeout);**

**成功返回**：**就绪描述符的正数目(ret>0)**；  **失败返回**：**-1**； **超时返回**：**0**

参数解释：

**maxfd**： 最大的文件描述符（Why其值应为**最大的文件描述符的值+1**）:解释在下面

**readset**： 内核读操作的描述符集合

**writeset**： 内核写操作的描述符集合

**exceptionset**：内核异常操作的描述符集合

**timeout**： 是传入传出参数，表示等待描述符就绪需要多少时间：

(1) NULL 代表永远等下去 (2)一个固定值代表等待的固定时间

(3) 0 代表根本不等待，检查描述字之后立即返回

1. fd\_set集合：是个**位数组**，存2个描述符：标准输入、管道读端。(select：阻塞函数)

其中readset、writeset、exceptionset 都是fd\_set 集合：**传入传出参数**

该集合的相关操作如下：

**void FD\_ZERO(fd\_set \*fdset);**  //将所有fd 清零

**void FD\_SET(int fd, fd\_set \*fdset);**  //增加一个fd

**void FD\_CLR(int fd, fd\_set \*fdset);**  //删除一个fd

**int FD\_ISSET(int fd, fd\_set \*fdset);** //判断一个fd 是否设置好了 (常用来：**判断传出参数的可读描述符 是否已经设置好了**)

y

**传入传出参数（value-result） rdset： 传入时需要放入数据，传出时数据已经被修改了**。 （传入一个值，会有所修改，再传出来）

注意：一般来说，在使用select 函数之前，首先要使用 FD\_ZERO 和 FD\_SET 来初始化文件描述符集(清空、设置要监控的描述符)，再传入给select。 在使用select 函数时，可循环使用FD\_ISSET 测试描述符集，在执行完相关文件描述符之后，使用FD\_CLR 来清除描述符集。 模板如下：

**fd\_set rdset;**  //定义集合(文件描述符集合：传入传出参数)

**FD\_ZERO(&rdset);** //先把所有文件描述符清空

**FD\_SET(0,&rdset);** //设置要监控的描述符：此处为 0(标准输入) 和 fdr(管道读端)

**FD\_SET(fdr,&rdset);** //设置完后，便将其取地址传入给select(还要实现传出：取地址)

//**传入监控**的描述符； **传出可读**的描述符

1. select的内部模拟实现：**rdset传入到select中**（**会在内核中轮询**）

以下函数并不用自己写，仅仅是模拟select中传出的过程：

**fd\_set tmpset;**

**memcpy(&tmpset,rdset,sizeof(fd\_set));**  //rdset传入后，将其 内存拷贝到tmpset

**FD\_ZERO(rdset);** //把传入的rdset清空。 清空后，select在内核中开始轮巡(故参数**maxfd为最大的文件描述符的值+1，来保证在内核中轮巡的范围**)，并去发现哪个描述符可读。

**FD\_SET(0, rdset);**

**FD\_SET(fdr,rdset)；**  //**发现可读的描述符后，通过FD\_SET将其重新写到rdset中，再传出去(因为是取地址可传出)，故参数rdset是 传入传出参数(指针)。**

注意：（1）传出的参数是： 在传入的描述符中，**此时哪个描述符可读了，将其传出来**。

（2） 可用 **FD\_ISSET()判断**：可读描述符是否写到了rdset中。

(6) 退出机制

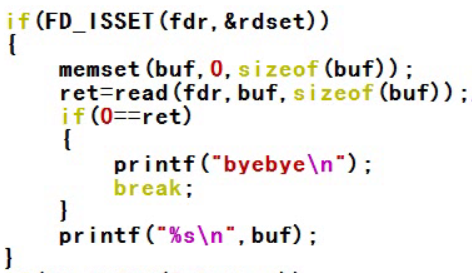
**一端ctrl+c断开后，其在内核中对应的描述符fdr 就会被 设为可读状态**，于是select就会一直监控到可读状态，即select返回值为1，能进入到if判断内( if(ret>0) )， 但是在if判断内，read去读fdr的时候却啥都没有，便会一直疯狂打印换行。

通过：**对端断开后，read的返回值为0，来写退出机制**： if(ret==0) break;

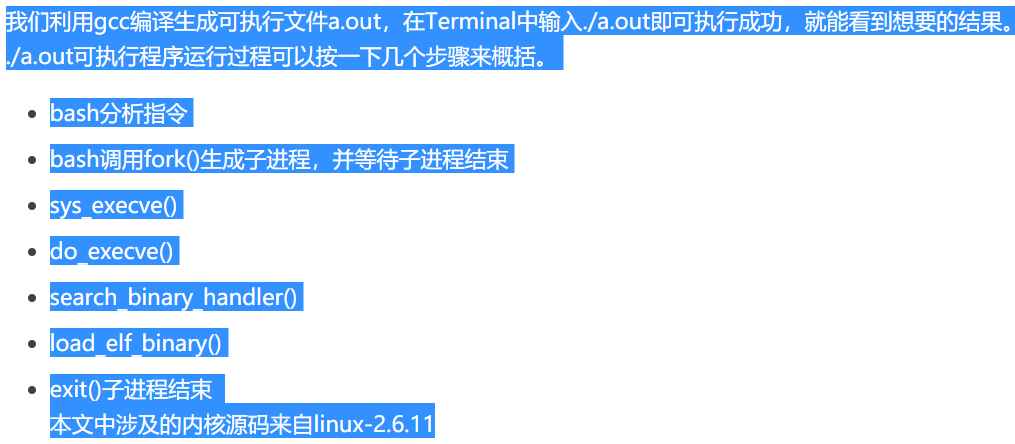
read()：1)读取成功：返回实际读取到的字节数；

2)失败错误：返回-1；

3)已到达文件尾 或 **无可读取的数据：返回0** (来写退出机制)



28.

bash外部可执行文件：

如果是外部可执行文件,bash采用默认的方式: 当我们输入./a.out 并摁下回车的时候, bash会调用系统调用fork一个子进程, 在子进程中执行a.out, bash会一直等待,直到子进程执行结束。

29. ssh非对称加密：(只给别人公钥，私钥自己留着)

非对称加密算法需要两个密钥：公钥 和 私钥。 公钥与私钥是一对：如果**用公钥对数据进行加密**，只有用**对应的私钥才能解密**。 因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。 非对称加密算法实现机密信息交换的基本过程是：甲方生成一对，密钥并将公钥公开，乙方使用该密钥(即甲方的公钥) 对机密信息进行加密后再发送给甲方；甲方再用自己私钥对加密后的信息进行解密； 甲方想要回复乙方时正好相反，使用乙方的公钥对数据进行加密，同理，乙方使用自己的私钥来进行解密。

另一方面，甲方可以使用自己的私钥对机密信息进行签名后再发送给乙方；乙方再用甲方的公钥对甲方发送回来的数据进行验签。

**只能用甲方的私钥来解密 由其公钥加密后的任何信息**。 非对称加密算法的保密性比较好，它消除了最终用户交换密钥的需要。

(1) root用户的home目录在：/root (根目录下的root目录)

在/root 目录下有 .ssh目录，里面存的：公钥、私钥、authorized\_keys文件。

(2) cd /root/.ssh/ 目录：

**ssh-keygen**命令： 来产生一对公钥、私钥 (均只有1行，是一行很长的字符串)

30. 可以封装一个命令用于登陆另一台Linux(桥接模式)：

**将本机的公钥，通过scp 拷贝到对方**的 **.ssh目录下**的**authorized\_keys**文件里即可，以后用命令登陆另一台Linux，就不用再输密码了。

1. 如下图，用kssh命令封装了ssh命令：



（2）chmod +x kssh ，再给此kssh命令可执行性权限。

（3）注意放公钥时：要把自己的公钥以**追加方式**放进去: **>>** ，来防止覆盖掉文件中别人原有的公钥。



31. scp 远程copy文件命令（不同Linux间拷贝数据：桥接模式）

**scp filename username@ip:path**

filename：文件名称 、 username：要copy到的目标主机的用户名、

ip：目标主机IP 、 path：目标主机路径



32. 工作时，敲得**命令很长时，要学会封装命令**： 如封装 ps -elf|grep

（1）**vim kps** ,再在文件中写： 要grep哪个进程，就$几 (如**$1**：要传递的第一个参数)

如下图在VIM中写：



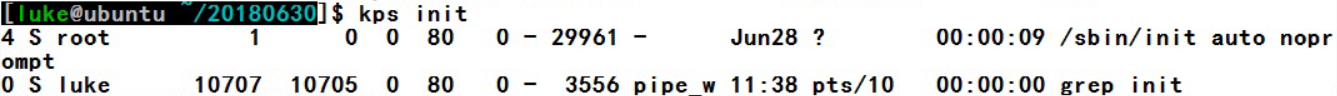
如：封装find、grep组合查找命令，要传3个参数时：用3个$



（2）**chmod +x kps**  //给命令可执行权限。

（3）**sudo cp kps /usr/bin** //此目录存放命令的默认路径，可用echo $PATH查看

（4）如下图，此时： 封装的 kps init命令 即等价于 ps -elf|grep init



33.

现代大家常用的操作系统实际是“**分时多任务操作系统**”，因此也有另一种“实时操作系统”，实时操作系统多用于工控，汽车，航空，航天，这里不做展开。 所谓分时操作系统，就是cpu运行时间被切分成很短时间的碎片，你的app在一直运行只是一种“假象”，实际操作系统不断切换分配cpu时间片给不同的程序运行，从而实现了多任务，当然，一定程度牺牲了你的程序的实时性（比如别的程序cpu时间占用过高，你的程序就可能很卡）。

这是管理模式，只有OS让你执行时你才能执行，到了时间就无情剥夺你占用CPU时间的权力（可以通过操作系统提供的方式提高优先级）。现代多核CPU也是由操作系统统一管理的，APP基本没有选择权力（可以通过操作系统提供的方式绑定）。计算机硬件是由操作系统OS管理的，你可以理解exe或者app，是由操作系统控制的一个动态模块。因此一个app的加载运行，申请资源，最后退出清理全部必须由OS进行调配。

内存是非常宝贵的（相对外存储器），为了适应同时运行更多app，提高内存利用率，扩大app可用“内存”，让app更容易开发，操作系统**内存地址都是要进行抽象的**，抽象可以理解为一层虚拟，在操作系统里叫内存分页管理，**APP操作内存实际使用的是虚拟后的逻辑内存地址**。无论你是几百M物理内存，虚拟后的内存对每个APP来看都是同等大小的，一般至少4G（大小一般由操作系统和cpu的架构决定）。

为了安全，稳定性，兼容新，涉及硬件底层的必须通过API代理由内核和驱动执行，各种I/O设备（显卡显示器，硬盘，网络，键盘鼠标…）的地址和内存映射，不同系统也不一样，这个是由IO硬件和驱动开发者设计的。

OS作为硬件上第一层软件，拥有全部的硬件的访问和控制权，且自己是唯一的控制者。

34.

**驱动** 和 **固件**(firmware)都是代码，前者为软件服务，后者为硬件服务。

硬件厂商希望自己的硬件能被更多的软件厂商使用，所以就需要在硬件之上做一些封装，让自己的硬件操纵起来更容易，这个时候就要有firmware这种东西了，它简化了软件与硬件的交互。 但是为什么不把fimware做的很完美，做的不需要驱动支持呢？因为有不同的操作系统，而不同的操作系统，对于操作硬件的方式完全不同。 硬件厂商一方面为了自己的硬件能被软件更简单的使用，就需要写firmware，而另一方面为了兼容各种操作系统，又不能把firmware写的太死，必须预留足够的余地让软件自由发挥——软件的自由发挥就是驱动。不同操作系统的驱动是不能兼容的，原因就是驱动是为操作系统服务的，这就是驱动存在的意义——适应系统需要。

驱动 driver 指的提供编程api接口，用以操控某一硬件/软件产品的应用程序。

驱动程序全称设备驱动程序，是添加到[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192)中的特殊程序，其中包含有关硬件设备的信息，这些信息能够使计算机与相应的设备进行通信。驱动程序是硬件厂商根据操作系统编写的配置文件，可以说没有驱动程序，计算机中的硬件就无法工作。

35. 虚拟机本质上也是运行在操作系统上的应用软件，只不过这个应用软件会独立的运行另外一个操作系统。

如今比较流行的接入Internet的方式是：**主机通过局域网组织在一起**，**然后再通过交换机和 Internet相连接**。 这样一来就出现了如何区分具体用户，防止盗用的问题。**由于IP只是逻辑上标识，任何人都随意修改，因此不能用来标识用户；而 MAC地址则不然，它是固化在网卡里面的。**基于MAC地址的这种特点，局域网采用了MAC地址来标识具体用户的方法。注意：具体实现：在交换机内部通过“表”的方式把MAC地址和IP地址一一对应，也就是所说的IP、MAC绑定。

IP 地址的主机标识号，可以使路由器能够把二层帧封装的分组，传送到网络上的一台特定的主机，使**主机号与 MAC 地址进行映射**，关键在于使用**子网掩码来确定**或者获取远程主机的网络地址信息。

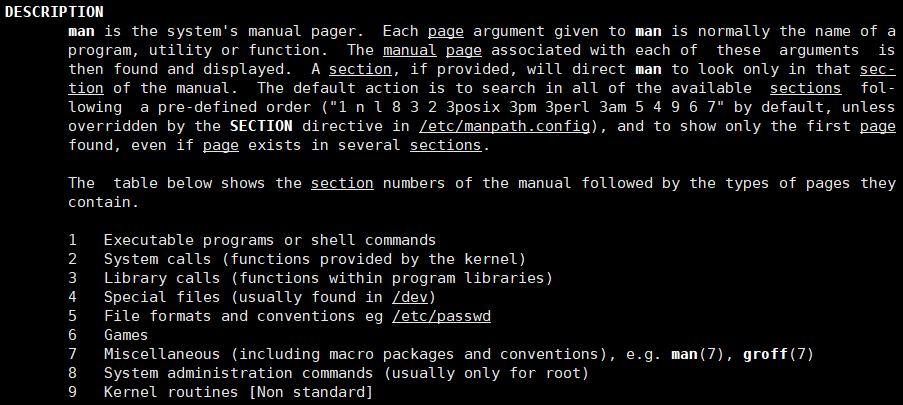
通常一块网卡只设置一个IP地址，其实可以配置多个IP地址；一台路由器会配置两个以上的网卡，故路由器可设置两个以上的IP地址。

在没有连接路由器的情况下，即使共享相同的物理[网段](https://baike.baidu.com/item/%E7%BD%91%E6%AE%B5)，只要网络号不同就无法进行通讯 ,必须通过具有[路由](https://baike.baidu.com/item/%E8%B7%AF%E7%94%B1" \t "_blank)功能的设备进行转发。

广播：广播的包可以被**同一数据链路上**（即网络标识相同）所有主机和路由器接收。一般为本地广播。

36. Linux man指令后面的数字的含义：

man 1: Standard commands （标准命令）  
man 2: System calls （系统调用）  
man 3: Library functions （库函数）  
man 4: Special devices （设备说明）  
man 5: File formats （文件格式）  
man 6: Games and toys （游戏和娱乐）  
man 7: Miscellaneous （杂项）  
man 8: Administrative Commands （管理员命令）  
man 9: 其他（Linux特定的）， 用来存放内核例行程序的文档。



37.

注意：**memset()**是按字节对内存块进行初始化，对于**int数组**：只能初始化为**0**和**-1**（即把每一个字节放成0x00或0xFF），而不能初始化为其他值(除非该值高字节和低字节相同)。 因为一个字节是0xff是-1, 4个字节0xffffffff仍是-1。

38.

要理解inode，就先要理解文件的存储方式。我们都知道，文件是存储在硬盘上的。硬盘的最小单位是扇区，每个扇区的大小为512字节。

如果系统在读取硬盘数据的时候按扇区一个一个来读取，那效率就太低了，而是一次连续性读取多个扇区，所以设计者又将多个扇区整合成一个块（block），所以，块就是文件存取的最小单位。一个块的大小为4k。

我们现在已经有了块的概念，文件数据就是存放在块中。但光有数据还是不行啊？为了方便管理文件，我们还需要文件的元信息，比如文件的属性，创建时间，权限，所占的块大小，数量等等。这些信息就是inode信息。所以硬盘在分区的时候会分为两个区域，一个区域存放数据，一个区域存放inode信息，每一个文件都有一个对应的inode。

39.

read/write并没有进行 真正的接收和发送数据操作， 他们需要进行用户态和内核态的切换，**真正的接收、发送操作是由内核的TCP协议栈完成**的。

read、write: 多了一次搬移，读取是先从磁盘读到内核的输入缓冲区中，再到用户态自己定义的栈中； 写入也是从用户态写到内核的输出缓冲区，再写到磁盘。注意：用户态(栈、堆、数据段) 和 内核态(内核)之间搬移数据，花费的开销是很大的。

