<https://blog.csdn.net/u011456016/article/details/70803198>

文件是linux中的一个重要概念。在Linux中，一切（几乎一切）都是文件。简单的说，C中基本的的printf（）函数，scanf（）函数，其实都属于文件操作。

对于文件操作，虽然都是通过函数调用的方式实现，却还是能分为两类：系统调用和库函数。

这篇文章将先介绍ｌｉｎｕｘ中文件的概念，系统调用和库函数的概念 ，然后具体的讨论两种方式下的文件操作。

博文的主要内容如下：

1. Linux 中的文件
2. 文件访问－库函数
3. 文件访问－系统调用
4. 库函数
5. 标准 I/O 库
6. ／proc文件系统

**1 Linux中的文件**

## 1.1概念

按照普通的定义，文件不过是一堆数据，在往下说，就是存储器中的０１０１。。。而我们这里讨论的文件有了更广的定义。对于Ｌｉｎｕｘ中的文件，我的理解是：

Ｌｉｎｕｘ中的文件具有的特点是：可通过操作系统或者程序对外提供信息，也能对内输入信息，可以被创建，删除。

Ｌinux中，文件有特别重要的意义，他们为操作系统和设备提供了一个简单而统一的接口。在Ｌｉｎｕｘ中，几乎一切都可以看做是文件 。

这就意味着，普通程序完全可以像使用文件（普通定义）那样使用磁盘文件、串行口、打印机和其他设备。

硬件设备在linux操作系统中也被表示为文件。例如，可以通过如下命令把cd-rom驱动器挂载为一个文件，

#mount -t iso9660 /dev/hdc /mnt/cdrom

#cd /mnt/rom

然后，就能像访问普通文件那样在ｃｄ－ｒｏｍ目录中漫游。

## 1.2操作

和操作一般意义上的文件一样，linux中对文件的操作只需要五个基本的函数：

open、close、read、write和ioctl

通过调用这几个函数就能对ｌｉｎｕｘ中的文件进行读、写等操作。不过，这种操作又分为系统调用和库函数调用。简单的说，系统调用是最直接的方式，

库函数调用最终也是通过系统调用实现的。可认为库函数调用是对系统调出于效率考虑而做出的优化。

库函数调用和系统调用的区别和联系请参看：[linux系统调用和库函数调用的区别](http://www.cnblogs.com/yanlingyin/archive/2012/04/23/2466141.html)

我们用很少的函数就可以对文件和设备进行访问和控制。这些函数就是所谓的系统调用，由操作系统直接提供，他们是通向操作系统本身的接口。

操作系统的核心部分，既内核，其实就是一组设备驱动程序。这是一些对硬件进行控制的接口。

**2 文件访问－系统调用**

通过系统调用来访问文件是最直接的方式。系统调用函数直接作用于操作系统内核的设备驱动程序从而实现文件访问。

## 2.1 文件描述符

在系统中需要处理的文件（读、写操作）需要一个标识，以便在其它地方能识别出这个文件，于是就产生了文件描述符。文件描述符是一些小值整数，简单的说就是

一个文件ＩＤ用于在系统中唯一的标识文件。文件描述符的总数也就是系统可以打开文件的最多个数，这取决于系统的配置情况。

当开始运行程序时，也就是系统开始运行时，它一般会有三个已经打开的文件描述符。他们是：

* ０：标准输入
* １：标准输出
* ２：标准错误

其它文件的文件描述符，在调用文件打开函数ｏｐｅｎ时返回。这就是说，每个设备对应着一个文件描述符。文件描述符由操作系统分配，每次分配最小的。

## 2.2 write系统调用

write，就是把缓冲区的数据写入文件中。注意，这里的文件时广泛意义的文件，比如写入磁盘、写入打印机等等。

Linux 中write（）的函数原型：

size\_t write(int fildes, const void \*buf, size\_t nbytes);

参数说明：

fildes：文件描述符，标识了要写入的目标文件。例如:fildes的值为1，就像标准输出写数据，也就是在显示屏上显示数据；如果为 2 ，则想标注错误写数据。

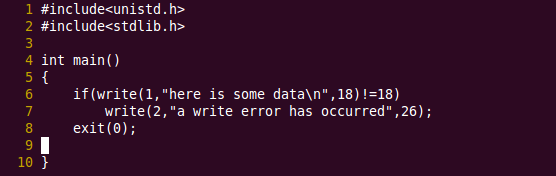
\*buf：待写入的文件，是一个字符串指针。

nbytes：要写入的字符数。

函数返回值：size\_t  返回成功写入文件的字符数。需要指出的是，write可能会报告说他写入的字节比你所要求的少。这并不一定是个错误。在程序中，你需要检查

error已发现错误，然后再次调用write写入剩余的数据。

请看下面的例子：



运行结果：

http://pic002.cnblogs.com/images/2012/348708/2012080322180832.png

这个程序只在标准输出上显示一条消息。

## read系统调用

系统调用read是从文件中读出数据。要读取的文件用文件描述符标识，数据读入一个事先定义好的缓冲区。他返回实际读入的字节数。

Linux中read的函数原型：

size\_t read(int fildes, void \*buf, size\_t nbytes);

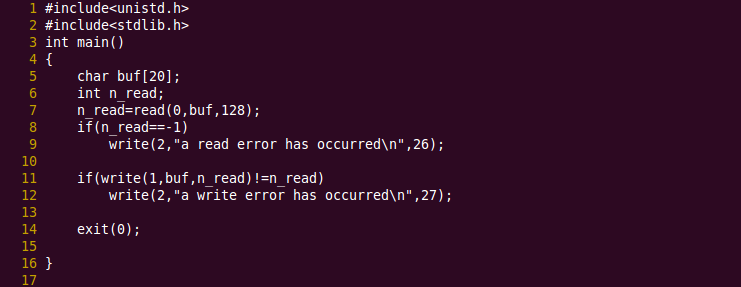
参数说明：

fildes：文件描述符，标识要读取的文件。如果为0，则从标准输入读数据。类似于scanf（）的功能。

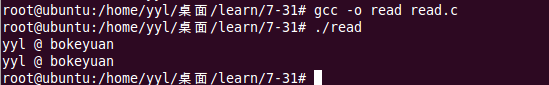
\*buf：缓冲区，用来存储读入的数据。

nbytes：要读取的字符数。

返回值：size\_t返回成功读取的字符数，它可能会小于请求的字节数。



运行结果：



## open系统调用

系统调用open的作用是打开一个文件，并返回这个文件的描述符。

简单地说，open建立了一条到文件或设备的访问路径。如果操作成功，它将返回一个文件描述符，read和write等系统调用使用该文件描述符对文件或

设备进行操作。这个文件描述符是唯一的，他不会和任何其他运行中的进程共享。如果两个程序同时打开一个文件，会得到两个不同的问价描述符。如果

同时对两个文件进行操作，他们各自操作，互补影响，彼此相互覆盖（后写入的覆盖先写入的）为了防止文件按读写冲突，可以使用文件锁的功能。这不是

本次重点，以后介绍。

Linux中open的函数原型有两个：

int open(const char \*path, int oflags);

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

参数说明。

path：准备打开的文件或设备名字。

oflags：指出要打开文件的访问模式。open调用必须指定如下所示的文件访问模式之一：



open调用哈可以在oflags参数中包括下列可选模式的组合（用”按位或“操作）：

* O\_APPEDN: 把写入数据追加在文件的末尾。
* O\_TRUNC: 把文件长度设为零，丢弃以后的内容。
* O\_CREAT: 如果需要，就按参数mode中给出的访问模式创建文件。
* O\_EXCL: 与O\_CREAT一起调用，确保调用者创建出文件。使用这个模式可防止两个程序同时创建一个文件，如果文件已经存在，open调用将失败。

 关于其他可能出现的oflags值，请看考open的调用手册。

mode：

当使用哦、O\_CREAT标志的open来创建文件时，我们必须使用三个参数格式的open调用。第三个参数mode 是几个标志按位OR后得到的。他们是：

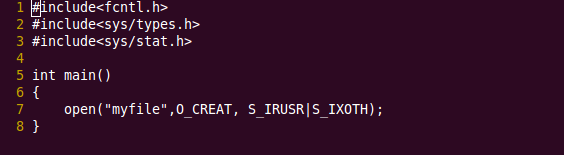
* S\_IRUSR: 读权限，文件属主。
* S\_IWUSR:写权限，文件属主。
* S\_ IXUSR:执行权限，文件属主。
* S\_IRGRP:读权限，文件所属组。
* S\_IWGRP:写权限，文件所属组。

。。。。

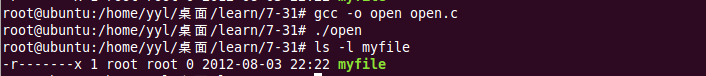
请看下面例子：

open（"myfile", O\_CREAT, S\_IRUSR|S\_IXOTH ;

他的作用是创建一个名为myfile 的文件，文件属主拥有读权限，其他用户拥有执行权限，且只有这些权限。



运行结果：



程序创建了一个名为myfile的文件，文件属主有读权限，其他用户有执行权限，且只有这些权限。

## close系统调用

close系统调用用于“关闭”一个文件，close调用终止一个文件描述符fildes以其文件之间的关联。文件描述符被释放，并能够重新使用。

close成功返回1，出错返回-1.

#Include<unistd.h>

int close(int fildes);

## ioctl系统调用

ioctl提供了一个用于控制设备及其描述符行为和配置底层服务的接口。终端、文件描述符、甚至磁带机都可以又为他们定义的ioctl，具体

细节可以参考特定设备的使用手册。

下面是ioctl 的函数原型

#include<unistd.h>

int ioctl(int fildes, int cmd,,,,,,);

ioctl对描述符fildes指定的对象执行cmd 参数中所给出的操作。

## 其他和文件管理有关的系统调用

还有许多其他的系统调用能对文件进行操作。

几个常用的如：lseek（）对文件描述符fildes指定文件的读写指针进行设置，也就是说，它可以设置文件的下一个读写位置。

fstat,stat,lstat 是和文件描述符相关的函数操作，这里就不做介绍。

dup，dup2系统调用。dup提供了复制文件描述符的方法，使我们能够通过两个或者更多个不同的文件描述符来访问同一个文件。这可以用于

在文件的不同位置对数据进行读写。

**4 库函数**

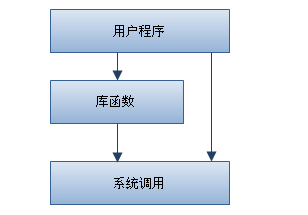
在输入、输出操作中，直接使用系统调用效率会非常底。具体原因有二：

* 系统调用会影响系统性能。与函数调用相比，系统调用的开销大。因为在执行系统调用的时候，要切换到内核代码区执行，然后再返回用户代码。这必然就需要大量的时间开支。一种解决办法是：尽量减少系统调用的次数，让每次系统调用完成尽可能多的 任务。例如每次系统调用写入大量的字符而不是单个字符。
* 硬件会对系统调用一次能读写的数据块做一定的限制。例如，磁带机通常的写操作数据块长度是10k，如果缩写数据不是10k的整数倍，磁带机还是会以10k为单位绕磁带，这就在磁带上留下空隙。

为了提高文件访问操作的效率，并且使得文件操作变得更方便，Linux发行版提供了一系列的标准函数库。他们是一些由函数构成的集合，你可以在自己的程序方便的中使用它们，

去操作文件。提供输出缓冲功能的标准I/O库就是这样的例子。你可以高效的写任意长度的数据块，库函数则在需要的时候安排底层函数调用（系统调用）

也就是说，库函数在用户和系统之间，增加了一个中间层。如下图所示：



库函数是根据实际需要而包装好的系统调用，用户可在程序中方便的使用库函数，如标准I O库（稍后会讲）

**5 标准I/O库**

 标准I/O库及其头文件<stdio.h>为底层I/O系统调用提供了一个通用的接口。这个库现在已经成为ANSI标准C的一部分，而前面所讲的系统调用却不是。

标准I/O库提供了许多复杂功能的函数，用于格式化输出和扫描输入，它还负责满足设备的缓冲需求。

在许多方面，使用标准I/O库和使用底层文件描述符类似。需要先打开一个文件，已建立一个文件访问路径（也就是系统调用中的文件描述符）

在标准I/O库中，与文件描述符对应的叫 流（stream），它被实现为指向结构FILE的指针。

在启动程序时，有三个文件流是自动打开的。他们是：

* stdin: 标准输入
* stdout: 标准输出
* stderr: 标准错误输出

下面会介绍一些常用的I/O库函数：

## 5.1 fopen函数

fopen函数类似于系统调用中的open函数。和open一样，它返回文件的标识符，只是这里叫做流（stream），在库函数里实现为一个指向文件的指针。

如果需要对设备的行为进行明确的控制，最好使用底层系统调用，因为这可以避免使用库函数带来的一些非预期的副作用，如输入/输出缓冲。

函数原型：

#include<stdio.h>

FILE \*fopen(const char \*filename, const char \*mode);

参数说明：

\*filename：打开文件的文件名

\*mode：打开的方式

     r 以只读方式打开文件，该文件必须存在。  
　　r+ 以可读写方式打开文件，该文件必须存在。  
　　rb+ 读写打开一个二进制文件，允许读数据。  
　　rw+ 读写打开一个文本文件，允许读和写。  
　　w 打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件  
　　w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。

fopen在成功是返回一个非空的FILE \*指针。失败返回NULL

## 5.2 fread/fwrite函数

fread函数从文件流中读取数据，对应于系统调用中的read；fwrite函数从文件流中写数据，对应于系统调用中的write

函数原型：

#include<stdio.h>

size\_t  fread(void \*ptr, size\_t size, size\_t nitems, FILE \*stream);

参数说明：

\*ptr 要读取数据的缓冲区，也就是要存放读取数据的地方。

size：指定每个数据记录的长度。

nitems： 计数，给出要传输的记录个数。

返回值：成功读取到数据缓冲区的记录个数，当到达文件尾时，他的返回值可能会消耗与nitems，甚至可以是0

size\_t  fwrite(const coid \*ptr, size\_t size , size\_t nitimes, FILE \*stream);

他从指定的数据缓冲区ptr中把数据写入文件流，返回成功写入的记录个数。

## 5.3 fclose函数

fclose函数关闭指定的文件流stream，这个操作会使所有未写出的数据都写出。因为stdio库函数会对数据进行缓冲，所有调用fclose函数是很重要的。

如果程序需要确保数据已经全部写出，就应该调用fclose函数。虽然程序正常结束时，也会自动的调用fclose函数，但这样就不能检测出调用fclose所产生的错误了。

函数原型如下：

#include<stdio,h>

int fclose(FILE \*stream);

## 5.4 fflush函数

fflush函数的作用是把文件流中所有未写出的数据全部写出。 处于效率考虑，在使用库函数的时候会使用数据缓冲区，当缓冲区满的时候才进行写操作。使用fflush函数

可以将缓冲区的数据全部写出，而不关心缓冲区是否满。fclose的执行隐含调用了fflush函数，所以不必再fclose执行之前调用fflush。

函数原型：

#include<stdio.h>

int fflush(FILE \*stream);

## 5.4 使用库函数进行文件读写

1,程序的实现步骤，打开一个文件，读取其中的内容，然后打开文件read.txt，读取其中的内容，然后打开data.txt文件，以追加的方式向其中写入在ｒｅａｄ.txt中读取的指定字节个数的内容，然后写成功后，关闭两个文件。

程序如下：

**[html]** [view plain](https://blog.csdn.net/u011456016/article/details/70803198) [copy](https://blog.csdn.net/u011456016/article/details/70803198)

1. #include**<stdlib.h>**
2. #include**<stdio.h>**
3. #include**<unistd.h>**
4. #include**<error.h>**
5. #include**<string.h>**
7. #if 0
9. #define READ\_FILE\_PATH "~/data.txt"
10. #define WRITE\_FILE\_PATH "~/write.txt"
11. #define LEN 5
12. int main(int argc, char \*argv[])
13. {
14. int len = 0;
15. FILE \*fd1 = NULL;
16. FILE \*fd2 = NULL;
17. char ptr[100] = {0};
19. fd1 = fopen(READ\_FILE\_PATH,"rw+");
20. if(fd1 == NULL) {
21. printf("open read file failed\n");
22. return -1;
23. }
25. //fd2 = fopen(WRITE\_FILE\_PATH,"rw+");
26. fd2 = fopen(WRITE\_FILE\_PATH,"ab");//追加
27. if(fd2 == NULL) {
28. printf("open write file failed\n");
29. return -1;
30. }
32. len = fread(ptr,1,LEN,fd1);
33. if(len != LEN) {
34. printf("read is error len1 = %d \n",len);
35. return -1;
36. }
38. len = fwrite(ptr,1,LEN,fd2);
39. if(len != LEN) {
40. printf("write is error len2 = %d \n",len);
41. return -1;
42. }
43. fclose(fd1);
44. fclose(fd2);
46. printf("read and write success\n");
47. return 1;
48. }
49. #endif

2,一次读取read.txt中的每一行，然后一行写入data.txt文件中，直到读取到read.txt文件的结尾.

**[html]** [view plain](https://blog.csdn.net/u011456016/article/details/70803198) [copy](https://blog.csdn.net/u011456016/article/details/70803198)

1. #if 1
3. #define READ\_FILE\_PATH "/home/zhuwg/Desktop/code-exe/Written\_Questions/file\_operation/data.txt"
4. #define WRITE\_FILE\_PATH "/home/zhuwg/Desktop/code-exe/Written\_Questions/file\_operation/write.txt"
5. #define LEN 5
6. #define MAX\_LEN 1024
7. int main(int argc, char \*argv[])
8. {
9. int len = 0;
10. FILE \*fd1 = NULL;
11. FILE \*fd2 = NULL;
12. char ptr[100] = {0};
13. char p[MAX\_LEN] = {0};
14. char q[MAX\_LEN] = {0};
16. fd1 = fopen(READ\_FILE\_PATH,"rw+");
17. if(fd1 == NULL) {
18. printf("open read file failed\n");
19. return -1;
20. }
22. //fd2 = fopen(WRITE\_FILE\_PATH,"rw+");
23. fd2 = fopen(WRITE\_FILE\_PATH,"ab");//追加
24. if(fd2 == NULL) {
25. printf("open write file failed\n");
26. return -1;
27. }
28. /\*
29. len = fread(ptr,1,LEN,fd1);
30. if(len != LEN) {
31. printf("read is error len1 = %d \n",len);
32. return -1;
33. }
34. \*/
35. while( !feof(fd1) )
36. {
37. memset(p,0,MAX\_LEN);
38. memset(q,0,MAX\_LEN);
39. fgets(p,MAX\_LEN,fd1);
40. //sprintf(q,"%s\n",p); //sprintf() and fprintf()
41. fputs(p,fd2);
42. }
43. /\*
44. len = fwrite(ptr,1,LEN,fd2);
45. if(len != LEN) {
46. printf("write is error len2 = %d \n",len);
47. return -1;
48. }
49. \*/
50. fclose(fd1);
51. fclose(fd2);
53. printf("read and write success\n");
54. return 1;
55. }

**6 /proc文件系统**

 Linux将一切看做文件，硬件设备在文件系统中也有相应的条目。/dev目录中的文件使用底层系统调用这样一种特殊方式来访问硬件。

/proc/devices与/dev的区别：

（１）

Ａ：　proc目录是一个虚拟文件系统，可以为[**Linux**](http://lib.csdn.net/base/linux)用户空间和内核空间提供交互

　　它只存在于内存中，而不占实际的flash或硬盘空间

Ｂ：　/proc/devices/里的设备是加载驱动程序时生成的

Ｃ：　/dev/下的设备是通过创建设备节点生成的，用户通过此设备节点来访问内核里的驱动

（２）

/proc/devices/中的设备是通过insmod加载到内核的，它可产生一个major供mknod作为 参数。   
/dev/\*.\* 是通过mknod加上去的，格式:mknod device1 c/b major minor 如：mknod dr1 c 254 0，用户通过此设备名来访问你的驱动。

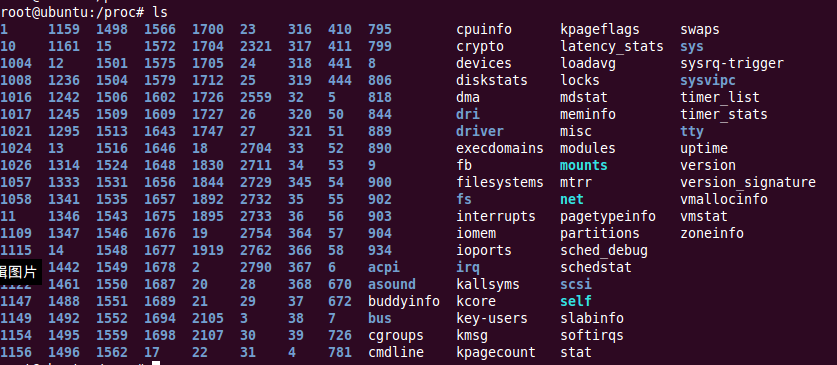
设备 文件 ,设备编号  #ll  -a /dev  在每一行都可以看到设备文件、设备编号（主、次）   
对于每种硬件设备，系统内核有相应的设备驱动程序负责对它的处理。而在Unix 中，使用设备文件的方式来表示硬件设备，每种设备驱动程序都被抽象 为设备文件的形式，这样就给应用程序一个一致的文件界面，方便应用程序和操作系统之间的通信。   
  
习惯上，所有的设备文件 都放置在/dev 目录下。 

/proc文件系统，可以看做是一个特殊的文件系统，在这个系统中，每个文件都对应一个独立的硬件，所以用户可以通过proc文件系统像访问文件一样来访问硬件设备。

该文件系统通常表现为/proc 目录。该目录中包含了许多特殊文件以允许对驱动和内核信息进行高层访问。

如果你想知道CPU的信息，内核版本信息等，就可以通过proc文件系统。

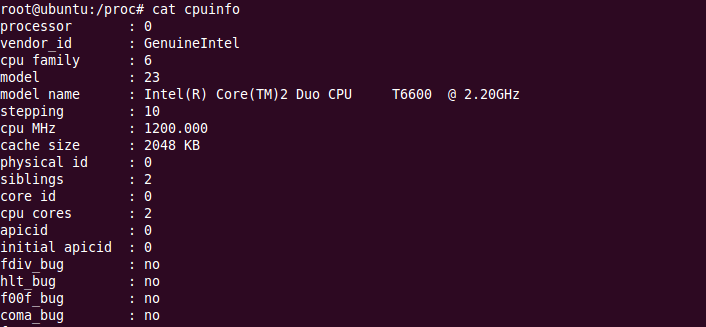
/proc目录中的文件会随系统的不同而不同。我的电脑上的/proc 中的文件如下所示：



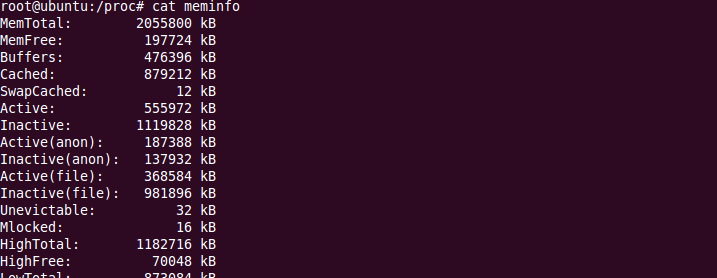
 在多数情况下，直接读取这些文件就可以获得状态信息。

## 6.1 访问设备信息

例如，获取CPU的信息：



内存使用信息(只显示里局部～)：

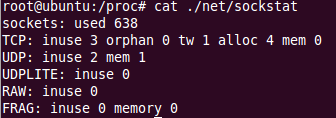


每次读这些文件的内容时，他们所提供的信息都会及时更新。所以再读一次meminfo文件会得到不同的结果。

由特定内核函数给出的更多信息可以在proc目录的子目录中查到。

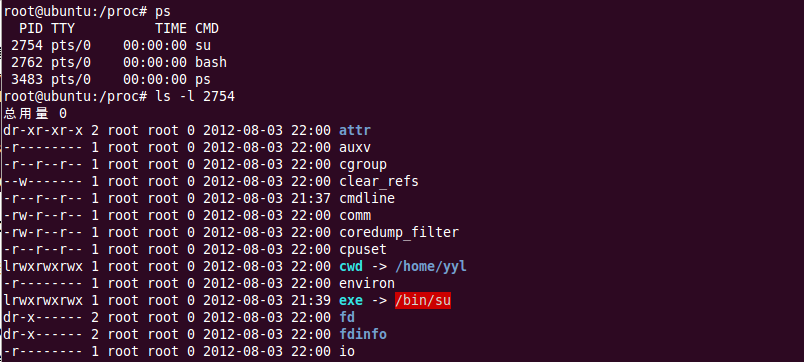
## 6.2 查看内核函数给出的信息

例如：查看网络套接字的使用统计：



## 6.3通过proc查看进程信息

用ps 命令可得到当前正在运行的进程，每个进程在proc中都有相应的信息文件，通过查看这个文件，可以得知进程相关的信息：



进程2754的当前工作目录是：/hme/yyl

程序 /bin/su正在运行，还有其他信息此处不再说明;

## 修改proc文件系统内容

例如，系统中所有运行的程序同时打开的文件总数是Linux内核的一个参数。

http://pic002.cnblogs.com/images/2012/348708/2012080321564119.png

如果我们想要增大这个歌值，则可通过写同一个文件来实现。

注意：对proc的写操作要注意权限问题，在修改时要小心，不适当的值可能会影响到系统的一运行。

http://pic002.cnblogs.com/images/2012/348708/2012080322011330.png

原文地址：

http://www.cnblogs.com/yanlingyin/archive/2012/08/04/2617209.html