

计算机操作系统

实验指导



上海大学计算机工程与科学学院

操作系统课程组

二〇一四年八月

计算机操作系统实验（一）目录

第一部分 《计算机操作系统（一）》课程实验.....	1
实验一 Linux 操作系统基本命令 ... 第一、二周	1
实验二 用户界面与 Shell 命令 ... 第三周	6
实验三 进程管理及进程通信..... 第四、五、六周	13
实验四 Linux 进程调度与系统监视 第七、八周	23
实验五 用户与组群管理..... 第九周	31
实验六 Shell 编程..... 第十周	39
第二部分 Linux 命令界面使用.....	41
第三部分 编辑程序 Vi.....	50
第四部分 Shell 程序设计语言	56
第五部分 Linux 编程系统调用.....	66
附录 A 实验报告格式.....	102
附录 B 参考资料	103

第一部分 《操作系统（一）》课程实验

实验一 Linux操作系统基本命令

一. 实验目的

1. 了解Linux运行环境，熟悉交互式分时系统、多用户环境的运行机制。
2. 练习Linux系统命令接口的使用，学会Linux基本命令、后台命令、管道命令等命令的操作要点。

二. 实验准备

复习操作系统中相关的用户接口概念。

查阅Linux中Shell的资料，它既是一个命令解释程序，又是一个程序设计语言。

熟悉本《实验指导》第二部分，从中你可以学会Shell的一般命令。

三. 实验内容

通过终端或虚拟终端，在基于字符的交互界面中进行Shell的基本命令的操作。

四. 实验步骤

1. 参照本《实验指导》第二部分介绍的方式，登录进入Linux命令操作界面。
2. 使用主机终端的用户可以用<Alt+ F1>、< Alt+ F2>、----< Alt+F6>切换屏幕，转换到其它虚拟终端，试着再登录进入系统，以实现多个用户同时登录到同一台计算机。
3. 参照本《实验指导》第二部分介绍的方式及实例，执行以下各类命令，熟悉Linux用户命令接口。
查看信息命令

序号	命令	功能
1	man [命令]	显示联机手册
2	[命令] - -help	显示联机帮助
3	pwd	显示当前目录
4	date	显示系统日期和时间
5	who	查看当前注册到系统的每个用户的信息
6	who am I	显示本用户信息
7	w [选项] [用户名]	显示目前注册的用户及用户正在运行的命令
8	id [用户名]	显示用户名与用户id、组名与组id
9	cal [月] [年]	查看日历
10	env	显示环境变量
11	vmstat 或 top	显示系统状态
12	clear	清除屏幕

操作：

- ① 执行pwd查看当前目录。
- ② 用who am i看看当前用户信息。
- ③ 通过who看看有谁在系统中。
- ④ 用vmstat显示系统状态。

📖 思考：你的用户名、用户标识、组名、组标识是什么？当前你处在系统的哪个位置中？现在有哪些用户和你一块儿共享系统

文件操作命令

序号	命令	功能
1	cat [>] 文件名	显示或创建一个文件
2	more [文件名]	分页浏览文件
3	head [-显示行数] 文件名	显示文件头部
4	tail [+起始行数] 文件名 或 tail [-起始行数] 文件名	显示文件尾部
5	cp [选项] 源文件 目标文件	复制文件
6	ln 文件名 新文件名	文件链接
7	mv [选项] 源文件 目标文件	移动或重命名文件
8	rm [选项] 文件名 目录名	删除文件
9	find 目录 [条件] [操作]	查找文件

提示：先用cat命令建立一个文件，然后用它进行其它目录操作和文件操作。

操作：

- ① 执行

cat > mytext.txt

通过键盘输入一些信息，用ctrl+c结束，建立文件mytext.txt。“>”是一个重定向命令。（可以参见本《实验指导》第二部分介绍的有关重定向内容）

- ② 执行

cat mytext.txt

显示文件内容。

- ③ 执行`

ln mytext.txt mytext2.dat

（建立链接）

cat mytext2.dat

（看到了吗？其中的内容是否与mytext.txt一样？）

④ 执行

```
ls -l mytext?.*
```

显示文件目录，注意i节点号，链接计数。

📖 思考：文件链接是什么意思？有什么作用？

目录操作

序号	命令	功能
1	ls [选项] [文件名---]	列目录 ^{说明①}
2	cd 目录名	改变当前目录
3	mkdir [-m 存取控制模式] 目录名	创建目录 ^{说明②}
4	rmdir 目录名	删除目录

说明：① 列目录操作通过选项设置显示方式。可以参见《实验指导》第二部分介绍的目录操作的内容）

② 若省略存取控制模式，则默认为0755，即文件主有全部权限，同组人和其它人只可读与执行；否则用三位八进制数说明模式。

操作：

① 执行

```
ls -l
```

看看当前目录的内容，请特别注意文件类型、文件的存取控制权限、i节点号、文件属主、文件属组、文件大小、建立日期等信息。

② 执行

```
cd /lib
```

```
ls -l|more
```

看看/lib目录的内容，这里都是系统函数。再看看/etc，这里都是系统配置用的数据文件；/bin中是可执行程序；/home下包括了每个用户主目录。

📖 思考：Linux文件类型有哪几种？文件的存取控制模式如何描述？

修改文件属性

序号	命令	功能
1	chown 用户名 文件名	改变文件的所有者
2	chgrp 组名 文件名	改变文件的组标识
3	chmod 访问模式 文件名 目录名	改变文件权限

操作：

① 执行

`chmod 751 mytext.txt`

(存取控制模式的表示可用八进制或字符表示, 参见本《实验指导》第二部分介绍的目录操作的内容。)

`ls -l mytext.txt`

(查看文件mytext.txt的存取控制权限。)

② 执行

(修改文件所有者为stud090)

`chown stud090 mytext.txt`

📖 思考: 执行了上述操作后, 若想再修该文件, 看能不能执行。为什么?

3. 熟悉进程概念, 进程通信中的软中断信号概念。执行以下进程管理命令。

进程管理命令

序号	命令	功能
1	<code>ps [选项]</code>	报告进程状态
2	<code>kill [-信号] 进程号</code> (传送信号给指定进程) <code>kill -l</code> (显示信号数和信号名表)	传送信号给当前运行的进程
3	<code>wait [n]</code>	等待进程完成
4	<code>sleep n</code>	挂起一段时间

操作:

① 执行

`ps -ef`

根据本《实验指导》第二部分介绍的进程管理命令选项, 查看当前系统中各个进程的信息。特别注意进程号、父进程号、属主等内容。

② 执行本《实验指导》第二部分介绍的wait和sleep命令。

📖 思考: 系统如何管理系统中的多个进程? 进程的家族关系是怎样体现的? 有什么用?

信息传递操作

序号	命令	功能
1	<code>talk 用户名 [终端名]</code>	与其他用户建立对话
2	<code>write 用户名 [终端]</code>	向其他用户发终端信息
3	<code>mesg [y n]</code>	允许或禁止其他用户发信息到本终端
4	<code>wall [信息]</code>	给所有现在登录系统的用户发广播

五. 讨论

1. Linux系统命令很多，在手头资料不全时，如何查看命令格式？
2. Linux系统用什么方式管理多个用户操作？如何管理用户文件，隔离用户空间？用命令及结果举例说明。
3. 用什么方式查看你的进程的管理参数？这些参数怎样体现父子关系？当结束一个父进程后其子进程如何处理？用命令及结果举例说明。
4. Linux 系统“文件”的含义是什么？它的文件有几种类型？如何标识的？
5. Linux 系统的可执行命令主要放在什么地方？找出你的计算机中所有存放系统的可执行命令的目录位置。
6. Linux 系统得设备是如何管理的？在什么地方可以找到描述设备的信息？
7. 画出Linux 根文件系统的框架结构。描述各目录的主要作用。你的用户主目录在哪里？
8. Linux 系统的Shell是什么？请查找这方面的资料，说明不同版本的Shell的特点。
9. 下面每一项说明的是哪类文件。

(1) -rwxrw-r--	(2) /bin	(3) ttyx3	(4) brw-rw-rw-
(5) /etc/passwd	(6) crw-rw-rw	(7) /usr/lib	(8) Linux

实验二 用户界面与 Shell 命令

一 实验要求

- (1) 掌握图形化用户界面和字符界面下使用 Shell 命令的方法。
- (2) 掌握 ls、cd 等 Shell 命令的功能。
- (3) 掌握重定向、管道、通配符、历史记录等的使用方法。
- (4) 掌握手工启动图形化用户界面的设置方法。

二 实验内容

图形化用户界面（GNOME 和 KDE）下用户操作非常简单而直观，但是到目前为止图形化用户界面还不能完成所有的操作任务。

字符界面占用资源少，启动迅速，对于有经验的管理员而言，字符界面下使用 Shell 命令更为直接高效。

Shell 命令是 Linux 操作系统的灵魂，灵活运用 Shell 命令可完成操作系统所有的工作。并且类 UNIX 的操作系统在 Shell 命令方面具有高度相似性。熟悉掌握 Shell 命令，不仅有助于掌握 RHEL Server 5，而且几乎有助于掌握各发行版本的 Linux，甚至 UNIX。

RHEL Server 5 中不仅可在字符界面下使用 Shell 命令，还可以借助于桌面环境下的终端工具使用 Shell 命令。桌面的终端工具中使用 Shell 命令时可显示中文，而字符界面下显示英文。

1. 图形化用户界面下的 Shell 命令操作

【操作要求 1】显示系统时间，并将系统时间修改为 2011 年 9 月 17 日零点。

【操作步骤】

- (1) 启动计算机，以超级用户身份登录图形化用户界面。
- (2) 依次单击顶部面板的「应用程序」菜单=>「附件」=>「终端」，打开桌面环境下的终端工具。
- (3) 输入命令“date”，显示系统的当前日期和时间。
- (4) 输入命令“date 091700002011”，屏幕显示新修改的系统时间。在桌面环境的终端中执行时显示中文提示信息，如图 2-1 所示。

【操作要求 2】切换为普通用户，查看 2011 年 9 月 17 日是星期几。

【操作步骤】

- (1) 前一操作是以超级用户身份进行的，但通常情况下只有在必须使用超级用户权限的时候，才以超级用户身份操作。为提高操作安全性，输入“su - helen”命令切换为普通用户 helen。
- (2) 输入命令“cal 2011”，屏幕上显示出 2011 年的日历，由此可知 2011 年 9 月 17 日是星期日，参见图 2-2。



图 2-1 设置时间



图 2-2 查看日历

【操作要求 3】查看 ls 命令的-s 选项的帮助信息

【操作步骤】

方法一：

- (1) 输入 “man ls” 命令，屏幕显示出手册页中 ls 命令相关帮助信息的第一页，介绍 ls 命令的含义、语法结构以及 -a、-A、-b 和 -B 等选项的意义。
- (2) 使用 PgDn 键、PgUp 键以及上、下方向键找到 -s 选项的说明信息。
- (3) 由此可知，ls 命令的 -s 选项等同于 --size 选项，以文件块为单位显示文件和目录的大小。
- (4) 在屏幕上的 “:” 后输入 “q”，退出 ls 命令的手册页帮助信息。

方法二：

- (1) 输入命令 “ls --help”，屏幕显示中文的帮助信息。
- (2) 拖动滚动条，找到 -s 选项的说明信息，由此可知 ls 命令的 -s 选项等同于 --size 选项，以文件块为单位列出所有文件的大小，如图 2-3 所示。
- (3) 在屏幕上的 “:” 后输入 “q”，退出 ls 命令的手册页帮助信息。



图 2-3查看帮助信息

【操作要求 4】查看/etc 目录下所有文件和子目录的详细信息

【操作步骤】

- (1) 输入命令“cd /etc”，切换到/etc 目录。
- (2) 输入命令“ls -al”，显示/etc 目录下所有文件和子目录的详细信息。

2. 字符界面下的 Shell 命令操作

包括 RHEL Server 5 在内的 Linux 系统都具有虚拟终端。虚拟终端为用户提供多个互不干扰、独立工作的工作界面，并且在不同的工作界面可用不同的用户身份登录。也就是说虽然用户只面对一个显示器，但可以切换到多个虚拟终端，好像在使用多个显示器。

RHEL Server 5 具有 7 个虚拟终端，其中第 1 个~第 6 个为字符界面；而第 7 个为图形化用户界面，必须启动图形化用户界面时才存在。各虚拟终端的切换方法为：

从字符界面的虚拟终端到其他虚拟终端：ALT+F1~ALT+F7

从图形化用户界面到字符界面：CTRL+ALT+F1~CTRL+ALT+F6

【操作要求 1】查看当前目录。

【操作步骤】

- (1) 启动计算机后默认会启动图形化用户界面，按下 CTRL+ALT+F1 键切换到第 1 个虚拟终端。
- (2) 输入一个普通用户的用户名（helen）和口令，登录系统。
字符界面下输入口令时，屏幕上不会出现类似“*”的信息，提高了口令的安全性。
- (3) 输入命令“pwd”，显示当前目录，相关操作参见如下内容。

```
Red Hat Enterprise Linux Server release 5 (Tikanga)
Kernel 2.6.18-8.el5 on an i386
```

```
localhost login : helen
Password :
Last login: Tue Nov 20 09:28:42 on tty1
[helen@localhost ~]$ pwd
/home/helen
```

虚拟终端未登录时显示的第一行信息表示当前使用的 Linux 的发行版本是 Red Hat Enterprise Linux Server，版本号为 5，又名 Tikanga。第二行信息显示 Linux 内核版本是 2.6.18-8.el5，以及本机的 CPU 型号是 i686。（Linux 将 Intel 奔腾以上级别的 CPU 都表示为 i686。）第三行信息显示本机默认的主机名 localhost。

成功登录系统后，还会显示该用户帐号上次登录系统的时间以及登录的终端号。

【操作要求 2】用 cat 命令在用户主目录下创建一名为 f1 的文本文件，内容为：

```
Linux is useful for us all.
You can never imagine how great it is.
```

【操作步骤】

- (1) 输入命令“cat >f1”，屏幕上输入点光标闪烁，依次输入上述内容。
使用 cat 命令进行输入时，不能使用左右上下方向键，只能用退格键（Backspace）来删除光标前一位的字符。并且一旦按下回车键，该行输入的字符就不可修改。
- (2) 上述内容输入后，按 Enter 键，让光标处于输入内容的下一行，按 CTRL+D 键结束输入。
- (3) 要查看文件是否生成，输入命令“ls”即可。
- (4) 输入命令“cat f1”，查看 f1 文件的内容，相关操作参见如下内容。

```
[helen@localhost ~]$ cat >f1
Linux is useful for us all.
You can never imagine how great it is.
[helen@localhost ~]$ ls
Desktop f1
[helen@localhost ~]$ cat f1
Linux is useful for us all.
```

You can never imagine how great it is.

【操作要求 3】向 f1 文件增加以下内容: Why not have a try?

【操作步骤】

- (1) 输入命令“cat >>f1”，屏幕上输入点光标闪烁。
- (2) 输入上述内容后，按 Enter 键，让光标处于输入内容的下一行，按 CTRL+D 键结束输入。
- (3) 输入“cat f1”命令，查看 f1 文件的内容，会发现 f1 文件增加了一行，相关操作参见如下内容。

```
[helen@localhost ~]$ cat >>f1
Why not have a try?
[helen@localhost ~]$ cat f1
Linux is useful for us all.
You can never imagine how great it is.
Why not have a try?
```

Shell 命令中可使用重定向来改变命令的执行。此处使用“>>”符号可向文件结尾处追加内容，而如果使用“>”符号则将覆盖已有的内容。

Shell 命令中常用的重定向符号共三个，如下所示：

- >: 输出重定向，将前一命令执行的结果保存某个文件。如果这个文件不存在，则将创建此文件；如果这个文件已有内容，则将放弃原有内容。
- >>: 附加输出重定向，将前一命令执行的结果追加到某个文件。
- <: 将某个文件交由命令处理。

【操作要求 4】统计 f1 文件的行数，单词数和字符数，并将统计结果存放在 countf1 文件。

【操作步骤】

- (1) 输入命令“wc <f1> countf1”，屏幕上不显示任何信息。
- (2) 输入命令“cat countf1”，查看 countf1 文件的内容，其内容是 f1 文件的行数、单词数和字符数信息，即 f1 文件共有 3 行，19 个词和 87 个字符，相关操作参见如下内容。

```
[helen@localhost ~]$ wc <f1> countf1
[helen@localhost ~]$ cat countf1
3 19 87
```

【操作要求 5】将 f1 和 countf1 文件的合并为 f 文件

【操作步骤】

- (1) 输入命令“cat f1 countf1 >f”，将两个文件合并为一个文件。
- (2) 输入命令“cat f”，查看 f 文件的内容，如下所示。

```
[helen@localhost ~]$ cat f1 countf1 >f
[helen@localhost ~]$ cat f
Linux is useful for us all.
You can never imagine how great it is.
Why not have a try?3 19 87
```

【操作要求 6】分页显示/etc 目录中所有文件和子目录的信息

【操作步骤】

- (1) 输入命令“ls /etc|more”，屏幕显示出“ls /etc”命令输出结果的第一页，屏幕的最后一行上还会出现“--More--”字样，按空格键可查看下一页信息，按 Enter 键可查看下一行信息。
- (2) 浏览过程中按“q”键，可结束分页显示。

管道符号“|”用于连接多个命令，前一命令的输出结果是后一命令的输入。

【操作要求 7】仅显示/etc 目录中前 5 个文件和子目录。

【操作步骤】

输入命令“ls /etc |head -n 5”，屏幕显示出“ls /etc”命令输出结果的前面 5 行，相关操作参见如下内容。

```
[helen@localhost ~]$ ls /etc|head -n 5
```

```
a2ps.cfg
a2ps-site.cfg
acpi
adjtime
aliases
```

【操作要求 7】清除屏幕内容

【操作步骤】

输入命令“clear”，则屏幕内容完全被清除，命令提示符定位在屏幕左上角。

3. 通配符的使用

Shell 命令的通配符包括*、?、[]、-和!，灵活使用通配符可同时引用多个文件方便操作。

*：匹配任意长度的任何字符。

?：匹配一个字符。

[]：表示范围。

-：通常与[]配合使用，起始字符-终止字符构成范围

!：表示不在范围，通常也与[]配合使用。

【操作要求 1】显示/bin/目录中所有以 c 为首字母的文件和目录

【操作步骤】

输入命令“ls /bin/c*”，屏幕将显示/bin 目录中以 c 开头的文件和目录，相关操作参见如下内容。

```
[helen@localhost ~]$ ls /bin/c*
/bin/cat      /bin/chmod    /bin/cp      /bin/csh
/bin/chgrp   /bin/chown    /bin/cpio    /bin/cut
```

【操作要求 2】显示/bin/目录中所有以 c 为首字母，文件名只有 3 个字符的文件和目录

【操作步骤】

(1) 按向上方向键，Shell 命令提示符后出现上一步操作时输入的命令“ls /bin/c*”。

(2) 将其修改为“ls /bin/c??”，按下 Enter 键，屏幕显示/bin 目录中以 c 为首字母，文件名只有 3 个字符的文件和目录，相关操作参见如下内容。

```
[helen@localhost ~]$ ls /bin/c??
/bin/cat      /bin/csh      /bin/cut
```

Shell 可以记录一定数量的已执行过的命令，当用户需要再次执行时，不用再次输入，可以直接调用。使用上下方向键，PgUp 或 PgDown 键，在 Shell 命令提示符后将出现已执行过的命令。直接按 Enter 键就可以再次执行这一命令，也可以对出现的命令行进行编辑，修改为用户所需要的命令后再执行。

【操作要求 3】显示/bin 目录中所有的首字母为 c 或 s 或 h 的文件和目录。

【操作步骤】

输入命令“ls /bin/[csh]*”，屏幕显示/bin 目录中首字母为 c 或 s 或 h 的文件和目录，相关操作参见如下内容。

```
[helen@localhost ~]$ ls /bin/[csh]*
/bin/cat      /bin/chown    /bin/csh      /bin/sed      /bin/sh      /bin/stty
/bin/chgrp   /bin/cp      /bin/cut      /bin/setfont  /bin/sleep   /bin/su
/bin/chmod   /bin/cpio    /bin/hostname /bin/setserial /bin/sort    /bin/sync
```

[csh]*并非表示所有以 csh 开头的文件，而表示是以 c 或 s 或 h 的文件。另外为避免误解，也可以使用[c,s,h]*，达到相同的效果。

【操作要求 4】显示/bin/目录中所有的首字母是 v、w、x、y、z 的文件和目录。

【操作步骤】

输入命令“ls /bin/[!a-u]*”，屏幕显示/bin 目录中首字母是 v~z 的文件和目录，相关操作参见如下内容。

```
[helen@localhost ~]$ ls /bin/[!a-u]*
/bin/vi      /bin/view    /bin/ypdomainname /bin/zcat
```

【操作要求 5】重复上一步操作

【操作步骤】

输入命令“!!”，自动执行上一步操作中使用过的“ls /bin/[!a-e]*”命令, 相关操作参见如下内容。

```
[helen@localhost ~]$ !!  
ls /bin/[!a-u]*  
/bin/vi    /bin/view  /bin/ypdomainname  /bin/zcat
```

用户不仅可利用上下方向键来显示执行过的命令；还可以使用 `history` 命令查看或调用执行过的命令。`history` 命令可查看到已执行命令在历史记录列表中的序号，可使用“`!` 序号”命令调用，而“`!!`”命令则执行最后执行过的那个命令。

【操作要求 6】 查看刚执行过的 5 个命令

【操作步骤】

输入命令“`history 5`”，显示最近执行过的 5 个命令，相关操作参见如下内容。命令编号可能不同。

```
[helen@localhost ~]$ history 5
15 ls /bin/c??
16 ls /bin/[csh]*
17 ls /bin/[^a-u]*
18 ls /bin/[^a-u]*
19 history 5
```

4. 设置手工启动图形化用户界面

图形化用户界面可以在启动 **RHEL Server 5** 时自动启动，也可以在字符界面启动后用“`startx`”命令 手动启动。`/etc/inittab` 文件中运行级别（`initdefault`）的取值决定启动 **RHEL Server 5** 后是否自动启动图形化用户界面。

RHEL Server 5 默认的运行级别为 5，即自动启动图形化用户界面，如果将其修改为 3 则只提供字符界面。

在实际工作中，对于以担任服务器功能为主的 **RHEL Server 5** 主机而言，通常运行级别为 3，这样的话系统资源可几乎完全用于提供服务，而不必消耗在图形界面上。

【操作要求 1】 设置开机不启动图形化用户界面

【操作步骤】

- (1) 按下 `ALT+F7` 键，切换回到图形化用户界面，以超级用户身份登录。
- (2) 依次单击「应用程序」菜单=>「附件」=>「文本编辑器」，打开 `gedit` 文本编辑器。
- (3) 单击工具栏上的「打开」按钮，从「打开文件...」对话框中选择 `/etc` 目录中的 `inittab` 文件。
- (4) 将文件中的“`id: 5: initdefault:` ”所在行的“5”修改为“3”，修改后的文件如图 2-4 所示。



2-4修改 `inittab` 文件

- (5) 单击工具栏上的「保存」按钮，并关闭 `gedit`。
- (6) 单击顶部面板的「系统」菜单=>「关机」，弹出对话框，选择「重新启动」，重新启动计算机。

【操作要求 2】 手工启动图形化用户界面。

【操作步骤】

- (1) 计算机重启后只有字符界面可用，输入用户名和相应的口令后，登录 **Linux** 系统。
- (2) 输入命令“`startx`”，启动图形化用户界面。
- (3) 单击「系统」菜单=>「注销」，弹出对话框，单击「注销」按钮，返回到字符界面。

实验三 进程管理及进程通信

一. 实验目的

利用Linux提供的系统调用设计程序，加深对进程概念的理解。
体会系统进程调度的方法和效果。
了解进程之间的通信方式以及各种通信方式的使用。

二. 实验准备

复习操作系统课程中有关进程、进程控制的概念以及进程通信等内容（包括软中断通信、管道、消息队列、共享内存通信及信号量概念）。
熟悉本《实验指导》第五部分有关进程控制、进程通信的系统调用。它会引导你学会怎样掌握进程控制。
阅读例程中的程序段。

三. 实验方法

用vi 编写c 程序（假定程序文件名为prog1.c）
编译程序

```
$ gcc -o prog1.o prog1.c
```

```
或 $ cc -o prog1.o prog1.c
```

运行

```
$/prog1.o
```

四. 实验内容及步骤

用vi 编写使用系统调用的C 语言程序。

1. 编写程序。显示进程的有关标识（进程标识、组标识、用户标识等）。经过5 秒钟后，执行另一个程序，最后按用户指示（如：Y/N）结束操作。

程序使用的系统调用原型请参见本《实验指导》第五部“Linux 编程系统调用”中“有关进程的系统调用”部分的相关内容。

2. 参考例程1，编写程序。实现父进程创建一个子进程。体会子进程与父进程分别获得不同返回值，进而执行不同的程序段的方法。

例程1：利用fork()创建子进程

```
/* 用fork()系统调用创建子进程的例子*/  
main()  
{  
    int i;  
    if (fork())
```

```

        i=wait();                                /*父进程执行的程序段*/
                                                /* 等待子进程结束*/
    {
        printf("It is parent process.\n");
        printf("The child process,ID number %d, is finished.\n",i);
    }
    else{                                        /*子进程执行的程序段*/
        printf("It is child process.\n");
        sleep(10);
        exit();                                /*向父进程发出结束信号*/
    }
}

```

📖 思考：子进程是如何产生的？ 又是如何结束的？子进程被创建后它的运行环境是怎样建立的？

3. 参考例程2，编写程序。父进程通过循环语句创建若干子进程。探讨进程的家族树以及子进程继承父进程的资源的关系。

例程2：循环调用fork()创建多个子进程。

```

/*建立进程树*/
#include<unistd.h>
main()
{ int i;
  printf( "My pid is %d, my father' s pid is %d\n" ,getpid()
        ,getppid());
  for(i=0; i<3; i++)
    if(fork()==0)
      printf( "%d pid=%d ppid=%d\n" , i,getpid(),getppid());
    else
    { j=wait(0);
      Printf(“ %d:The chile %d is finished.\n” ,getpid(),j);
    }
}

```

📖 思考：① 画出进程的家族树。子进程的运行环境是怎样建立的？反复运行此程序看会有什么情况？解释一下。

② 修改程序，使运行结果呈单分支结构，即每个父进程只产生一个子进程。画出进程树，解释该程序。

4. 参考例程3 编程，使用fork()和exec()等系统调用创建三个子进程。子进程分别启动不同程序，并结束。反复执行该程序，观察运行结果，结束的先后，看是否有不同次序。

例程3:创建子进程并用execvp()系统调用执行程序实验

```
/*创建子进程，子进程启动其它程序*/
#include<stdio.h>
#include<unistd.h>
main()
{
    int child_pid1,child_pid2,child_pid3;
    int pid,status;
    setbuf(stdout,NULL);
    child_pid1=fork();                                /*创建子进程1*/
    if(child_pid1==0)
    { execvp("echo","echo","child process 1",(char *)0); /*子进程1 启动其它程序*/
      perror("exec1 error.\n ");
      exit(1);
    }
    child_pid2=fork();                                /*创建子进程2*/
    if(child_pid2==0)
    { execvp("date","date",(char *)0);                 /*子进程2 启动其它程序*/
      perror("exec2 error.\n ");
      exit(2);
    }
    child_pid3=fork();                                /*创建子进程3*/
    if(child_pid3==0)
    { execvp("ls","ls",(char *)0);                     /*子进程3 启动其它程序*/
      perror("exec3 error.\n ");
      exit(3);
    }
    puts("Parent process is waiting for child process return!");
    while((pid=wait(&status))!=-1)                    /*等待子进程结束*/
    { if(child_pid1==pid)                              /*若子进程1 结束*/
      printf("child process 1 terminated with status %d\n",(status>>8));
      else
      {if(child_pid2==pid)                             /*若子进程2 结束*/
```

```

        printf("child process 2 terminated with status %d\n", (status >> 8));
    else
    { if(child_pid3==pid)                /*若子进程3 结束*/
        printf("child process 3 terminated with status %d\n", (status >> 8));
    }
}
}
puts("All child processes terminated.");
puts("Parent process terminated.");
exit(0);
}

```

📖 思考：子进程运行其它程序后，进程运行环境怎样变化的？反复运行此程序看会有什么情况？解释一下。

5. 参考例程4 编程，验证子进程继承父进程的程序、数据等资源。如用父、子进程修改公共变量和私有变量的处理结果；父、子进程的程序区和数据区的位置。

例程4：观察父、子进程对变量处理的影响

/*创建子进程的实验。子进程继承父进程的资源，修改了公共变量**globa** 和私有变量**vari**。观察变化情况。*/

```

#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int globa=4;
int main()
{
    pid_t pid;
    int vari=5;
    printf("before fork.\n");
    if ((pid=fork())<0)
    {
        printf("fork error.\n");
        exit(0);
    }
}

```

```

else
    if(pid==0)
    {
        globa++;
        vari--;
        printf("Child %d changed the vari and globa.\n",getpid());
    }
    else
        printf("Parent %d did not changed the vari and globa.\n",getpid());
    printf("pid=%d, globa=%d, vari=%d\n",getpid(),globa,vari);
    exit(0);
}

```

📖 思考：子进程被创建后，对父进程的运行环境有影响吗？解释一下。

6. 参照《实验指导》第五部分中“管道操作的系统调用”。复习管道通信概念，参考例程5，编写一个程序。父进程创建两个子进程，父子进程之间利用管道进行通信。要求能显示父进程、子进程各自的信息，体现通信效果。

例程5：管道通信的实验

/*程序建立一个管道fd*/
 /*父进程创建两个子进程P1、P2 */
 /*子进程P1、P2 分别向管道写入信息*/
 /*父进程等待子进程结束，并读出管道中的信息*/

```

#include<stdio.h>
main()
{
    int i,r,j,k,l,p1,p2,fd[2];
    char buf[50],s[50];
    pipe(fd);
    while((p1=fork())!=-1);
    if(p1==0)

```

```

lockf(fd[1],1,0);                                /*子进程1 执行*/
                                                    /*管道写入端加锁*/

printf(buf,"Child process P1 is sending messages! \n");
printf("Child process P1! \n");
write(fd[1],buf,50);                              /*信息写入管道*/
lockf(fd[1],0,0);                                /*管道写入端解锁*/
sleep(5);
j=getpid();
k=getppid();
printf("P1 %d is weakup. My parent process ID is %d.\n",j,k);
exit(0);
}
else
{ while((p2=fork())!=-1);                        /*创建子进程2*/
    if(p2==0)
    {
        lockf(fd[1],1,0);                        /*子进程2 执行*/
                                                    /*管道写入端加锁*/
        sprintf(buf,"Child process P2 is sending messages! \n");
        printf("Child process P2! \n");
        write(fd[1],buf,50);                      /*信息写入管道*/
        lockf(fd[1],0,0);                        /*管道写入端解锁*/
        sleep(5);
        j=getpid();
        k=getppid();
        printf("P2 %d is weakup. My parent process ID is %d.\n",j,k);
        exit(0);
    }
    else
    { l=getpid();
      wait(0);                                    /* 等待被唤醒*/
      if(r=read(fd[0],s,50)==-1)
          printf("Can't read pipe. \n");
      else
          printf("Parent %d: %s \n",l,s);
      wait(0);                                    /* 等待被唤醒*/
      if(r=read(fd[0],s,50)==-1)
          printf("Can't read pipe. \n");
      else
    }
}

```

```

        printf("Parent %d: %s\n",l,s);
        exit(0);
    }
}
}

```

- 📖 思考：①什么是管道？进程如何利用它进行通信的？解释一下实现方法。
 ②修改睡眠时机、睡眠长度，看看会有什么变化。请解释。
 ③加锁、解锁起什么作用？不用它行吗？

7. 编程验证：实现父子进程通过管道进行通信。进一步编程，验证子进程结束，由父进程执行撤消进程的操作。测试父进程先于子进程结束时，系统如何处理“孤儿进程”的。

- 📖 思考：对此作何感想，自己动手试一试？解释一下你的实现方法。

8. 编写两个程序一个是服务者程序，一个是客户程序。执行两个进程之间通过消息机制通信。消息标识**MSGKEY** 可用常量定义，以便双方都可以利用。客户将自己的进程标识（**pid**）通过消息机制发送给服务者进程。服务者进程收到消息后，将自己的进程号和父进程号发送给客户，然后返回。客户收到后显示服务者的**pid** 和**ppid**，结束。以下例程6 基本实现以上功能。这部分内容涉及《实验指导》第五部分中“IPC系统调用”。先熟悉一下，再调试程序。

例程6：消息通信的实验

/*客户进程向服务器进程发出信号，服务器进程接收作出应答，并再向客户返回消息。*/

```

=====
/*服务者程序*/
/*The server receives the message from client,and answer a message*/
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>

```

```

#define MSGKEY 75
struct msgform                                     /*定义消息结构*/
{
    long mtype;
    char mtext[256];
}msg;
int msgqid;
main()
{
    int i,pid,* pint;
    extern cleanup();
    for(i=0;i<20;i++)                               /*设置软中断信号的处理程序*/
        signal(i,cleanup);
    msgqid=msgget(MSGKEY,0777|IPC_CREAT);             /*建立消息队列*/
    for(;;)                                           /*等待接受消息*/
    {
        msgrcv(msgqid,&msg,256,1,0);                 /* 接受消息*/
        pint=(int *)msg.mtext;
        pid=*pint;
        printf("Server :receive from pid %d\n",pid); /*显示消息来源*/
        msg.mtype=pid;
        *pint=getpid();                              /*加入自己的进程标识*/
        msgsnd(msgqid,&msg,sizeof(int),0);            /*发送消息*/
    }
}
cleanup()
{
    msgctl(msgqid,IPC_RMID,0);
    exit();
}

=====
/*客户程序*/
/*The client send a message to server,and receives another message from
server*/
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define MSGKEY 75
struct msgform                                     /*定义消息结构*/
{

```


```

    long mtype;
    char mtext[256];
};
main()
{
    struct msgform msg;
    int msgqid,pid,*pint;
    msgqid=msgget(MSGKEY,0777);           /*建立消息队列*/
    pid=getpid();
    pint=(int *)msg.mtext;
    *pint=pid;
    msg.mtype=1;                           /*定义消息类型*/
    msgsnd(msgqid,&msg,sizeof(int),0);     /*发送消息*/
    msgrcv(msgqid,&msg,256,pid,0);         /*接受从服务器发来的消息*/
    printf("Clint : receive from pid %d\n",* pint);
}

```

实验可以在后台运行服务器进程（命令行后加&），前台运行客户进程或用不同终端运行客户进程。并可通过ps 命令察看后台进程。

用Shell编写一个脚本程序。先在后台启动服务程序，再在前台反复执行多个客户程序，观察系统反映。体会客户/服务器体系结构。实验结束向服务器发出终止服务的信号。

 思考：想一下服务器程序和客户程序的通信还有什么方法可以实现？解释一下你的设想，有兴趣试一试吗。

9. 这部分内容涉及《实验指导》第五部分中“有关信号处理的系统调用”。编程实现软中断信号通信。父进程设定软中断信号处理程序，向子进程发软中断信号。子进程收到信号后执行相应处理程序。

例程7：软中断信号实验

/* 父进程向子进程发送18 号软中断信号后等待。子进程收到信号，执行指定的程序，再将父进程唤醒。*/

```


main()
{
    int i,j,k;
    int func();
    signal(18,func());           /*设置18 号信号的处理程序*/
}

```

```

if(i=fork())                                /*创建子进程*/
{
    j=kill(i,18);                            /*向子进程发送信号*/
    printf("Parent: signal 18 has been sent to child %d,returned %d.\n",i,j);
    k=wait();                                /*父进程被唤醒*/
    printf("After wait %d,Parent %d: finished.\n",k,getpid());
}
else
{
    sleep(10);
    printf("Child %d: A signal from my parent is recived.\n",getpid());
}
/*子进程结束，向父进程发子进程结束信号*/
}
func()                                       /*处理程序*/
{ int m;
    m=getpid();
    printf("I am Process %d: It is signal 18 processing function.\n",m);
}

```

 思考：这就是软中断信号处理，有点儿明白了吧？讨论一下它与硬中断有什么区别？看来还挺管用，好好利用它。

10. 怎么样，试一下吗？用信号量机制编写一个解决生产者—消费者问题的程序，这可是受益匪浅的事。本《实验指导》第五部分有关进程通信的系统调用中介绍了信号量机制的使用。

五. 研究并讨论

1. 讨论Linux 系统进程运行的机制和特点，系统通过什么来管理进程？
2. C 语言中是如何使用Linux 提供的功能的？用程序及运行结果举例说明。
3. 什么是进程？如何产生的？举例说明。
4. 进程控制如何实现的？举例说明。
5. 进程通信方式各有什么特点？用程序及运行结果举例说明。
6. 管道通信如何实现？该通信方式可以用在何处？
7. 什么是软中断？软中断信号通信如何实现？

一 实验要求

- (1) 熟练掌握手工启动前后台作业的方法。
- (2) 熟练掌握进程与作业管理的相关 Shell 命令。
- (3) 掌握 at 调度和 cron 调度的设置方法。
- (4) 了解进行系统性能监视的基本方法。

二 实验内容

1. 作业和进程的基本管理

【操作要求 1】先在前台启动 vi 编辑器并打开 f4 文件，然后挂起，最后在后台启动一个查找 inittab 文件的 find 作业，find 的查找结果保存到 f5。

【操作步骤】

- (1) 以超级用户 (root) 身份登录到 RHEL Server 5 字符界面。
- (2) 输入命令“vi f4”，在前台启动 vi 文本编辑器并打开 f4 文件。
- (3) 按下 Ctrl+Z 组合键，暂时挂起“vi f4”作业，屏幕显示该作业的作业号。
[1]+ stopped vim f4
- (4) 输入命令“find / -name inittab > f5 &”，启动一个后台作业，如下所示。在显示作业号的同时还显示进程号。

```
[root@localhost ~]# find / -name inittab >f5 &  
[2] 2619
```

【操作要求 2】查看当前作业、进程和用户信息，并对作业进行前后台切换。

【操作步骤】

- (1) 输入命令“jobs”，查看当前系统中的所有作业。

```
[root@localhost ~]# jobs  
[1]+  Stopped                  vi 4  
[2]-  Running                  find / -name inittab > f5
```

由此可知“vi f4”作业的作业号为 1，已经停止。“find / -name inittab > f5 &”作业的作业号为 2，正在运行。

- (2) 输入命令“fg 2”，将“find / -name inittab > f5 &”作业切换到前台。屏幕显示出“find / -name inittab > f5”命令，并执行此命令。稍等片刻，作业完成后屏幕再次出现命令提示符。
- (3) 输入命令“cat f5”，查看“find / -name inittab > f5”命令的执行结果。
- (4) 再次输入命令“jobs”，可发现当前系统中的只有一个已停止的作业“vi f4”。
- (5) 输入命令“kill -9 %1”，终止“vi f4”作业。

```
[root@localhost ~]# jobs  
[1]+  Stopped                  vi f4  
[root@localhost ~]# kill -9 %1
```

使用 kill 命令时“-9”选项可强制性中止进程或作业。

- (6) 稍等片刻，输入命令“jobs”，查看到当前没有任何作业。

```
[root@localhost ~]# jobs  
[root@localhost ~]#
```

- (7) 输入命令“ps -l”，查看进程的相关信息，显示出的信息类似如下信息。

```
[root@localhost ~]# ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	2587	2586	0	75	0	-	1450	wait4	tty1	00:00:00	bash
4	R	0	2640	2587	0	77	0	-	819	-	tty1	00:00:00	ps

主要输出项的含义为:

S (State)	进程状态, 其中 R 表示运行状态; S 表示休眠状态; T 表示暂停或终止状态; Z 表示僵死状态。
UID (User ID)	进程启动者的用户 ID。
PID (Process ID)	进程号。
PPID (Parents Process ID)	父进程的进程号。
NI (Nice)	进程的优先级值。
SZ (Size)	进程占用内存空间的大小, 为 KB 为单位。
TTY (Terminal)	进程所在终端的终端号, 其中桌面环境的终端窗口表示为 pts/0, 字符界面的终端号为 tty1~tty6。
TIME	进程已运行的时间。
CMD (Command)	启动该进程的 Shell 命令。

(8) 输入命令 “who -H”, 查看用户信息。

```
[root@localhost ~]# who -H
```

NAME	LINE	TIME	COMMENT
root	tty1	2007-05-25 08:25	
lucy	tty2	2007-05-25 08:36	

主要输出项的含义为:

NAME: 用户名
 LINE: 用户登录的终端号
 TIME: 用户登录的时间。

2. at 进程调度

所谓进程调度就是设定某个指定的作业在固定的时间、或者固定的频率, 或者系统空闲时自动执行的操作。根据作业要求执行的条件不同, 可选中不同的调度方式。

at 调度: 在指定的时间执行一次特定的作业。

batch 调度: 在系统空闲时执行一次特定的作业。

cron 调度: 每到指定的时间就执行特定的作业, 可执行多次。

【操作要求 1】设置一个调度, 要求在 2008 年 1 月 1 日 0 时, 向所有用户发送新年快乐的问候。

【操作步骤】

(1) 超级用户输入命令 “at 00:00 01012008”, 设置 2008 年 1 月 1 日 0 时执行的 at 调度的内容。

at 调度的时间表示方法如下所示:

- (1) HH:MM: 即小时:分钟, 如 09:17, 采用 24 小时计时制。
- (2) 数字 AM/PM: 采用 12 小时计时制, 如 3am。
- (3) MMDDYY 或 MM/DD/YY 或 DD.MM.YY: 指定具体的日期, 必须写在具体时间之后。
- (4) now+时间间隔: 指定距离现在的时间, 时间单位为 minutes (分钟), hours (小时), day (天), week (星期)。
- (5) 具体时间: today (今天)、tomorrow (明天)、midnight (深夜)、noon (中午)、teatime (下午 4 点)、Tuesday (周二)、July 11 (7 月 11 日)。

(2) 屏幕出现 at 调度的命令提示符“at>”，输入“wall Happy New Year!”，向所有用户发送消息。

(3) 光标移动到“at>”提示符的第三行，按下 Ctrl+D 组合键结束输入。根据调度设置的时间，最后显示出作业号和将要运行的时间。

```
[root@localhost ~]# at 00:00 01012008
at>wall Happy New Year!
at><EOT>
job 1 at 2008-01-01 00:00
```

【操作要求 2】 设置一个调度，要求 5 分钟后向所有用户发送系统即将重启的消息，并在 2 分钟后重新启动计算机。

【操作步骤】

- (1) 超级用户输入命令“at now +5 minutes”，设置 5 分钟后执行的 at 调度的内容。
- (2) 屏幕出现 at 调度的命令提示符“at>”，输入“wall please logout; the computer will restart.”，向所有用户发送消息。
- (3) 在“at>”提示符的第二行输入“shutdown -r +2”，系统 2 分钟后将重新启动。
“shutdown -r +2”命令与“reboot +2”命令效果相同，都是在 2 分钟后重新启动。

【操作步骤】

- (1) 超级用户输入命令“at now +5 minutes”，设置 5 分钟后执行的 at 调度的内容。
- (2) 屏幕出现 at 调度的命令提示符“at>”，输入“wall please logout; the computer will restart.”，向所有用户发送消息。
- (3) 在“at>”提示符的第二行输入“shutdown -r +2”，系统 2 分钟后将重新启动。
“shutdown -r +2”命令与“reboot +2”命令效果相同，都是在 2 分钟后重新启动。
- (4) 光标移动到“at>”提示符的第三行，按下 Ctrl+D 组合键结束输入。最后显示作业号和运行时间。

```
[root@localhost ~]# at now+5 minutes
at>wall please logout,the computer will restart
at>shutdown -r +2
at><EOT>
job 2 at 2007-10-13 12:07
```

【操作要求 3】查看所有的 at 调度，并删除 08 年 1 月 1 日执行的调度任务。

【操作步骤】

(1) 输入“atq”命令，查看所有的 at 调度，显示出作业号、将在何时运行以及 at 调度的设定者。

```
[root@localhost ~]# atq
1 at 2008-01-01 00:00 a root
2 at 2007-10-13 12:07 a root
```

(2) 输入“atrm 1”命令删除作业号为 1 的 at 调度，并再次输入“atq”命令查看剩余的所有 at 调度内容。

```
[root@localhost ~]# atrm 1
[root@localhost ~]# atq
2 at 2007-10-13 12:07 a root
```

(3) 5 分钟后系统将自动运行作业号为 2 的 at 调度内容。先向所有用户发送消息，然后再等 2 分钟重新启动。

3. cron 进程调度

cron 调度的内容共有 6 个字段，从左到右依次为分钟、小时、日期、月份、星期和命令，如表 4-1 所示。

表 4-1 cron 文件的格式						
字段	分钟	小时	日期	月份	星期	命令
取值范围	0~59	0~23	01~31	01~12	0~6, 0 为星期天	

在设置 cron 调度时，所有的字段都不能为空，字段之间用空格分开，如果不指定字段内容，则使用“*”符号。

使用“-”符号表示一段时间。如果在日期栏中输入“1-5”则表示每个月前 5 天每天都要执行该命令。

使用“,”符号表示指定的时间。如果在日期栏中输入“5,15,25”则表示每个月的 5 日、15 日和 25 日都要执行该命令。

使用“/”符号表示间隔频率，如果在小时栏中输入“*/2”，表示 某 2 小时执行一次该命令。

【操作要求 1】helen 用户设置 crontab 调度，要求每天上午 8 点 30 份查看系统的进程状态，并将查看结果保存于 ps.log 文件。

【操作步骤】

(1) 以普通用户 helen 登录，并输入命令“crontab -e”，新建一个 crontab 配置文件。

(2) 屏幕出现 vi 编辑器，按下“i”，进入输入模式，输入“30 8 * * * ps >ps.log ”。

(3) 按下 Esc 键退出 vi 的文本输入模式，并按下“:”键切换到最后一行模式，输入“wq”，保存并退出编辑器，显示“crontab: installing new crontab”信息。

(4) 输入命令“crontab -l”，查看 helen 用户的 cron 调度内容。

```
[helen@localhost ~]$ crontab -l
30 8 * * * ps >ps.log
```

- (5) 为立即查看到 `crontab` 调度的结果，切换为超级用户，并适当修改系统时间，如修改为 8 点 29 分。最后退回到 `helen` 用户。

```
[helen@localhost ~]$ su -  
Password:  
[root@localhost ~]# date 11200829  
Tue Nov 20 08:29:00 CST 2007  
[root@localhost ~]# exit  
[helen@localhost ~]$
```

- (6) 等待 1 分钟后，查看 `ps.log` 文件的内容，如果显示出正确内容，那么说明 `crontab` 调度设置成功。

【操作要求 2】`helen` 用户添加设置 `crontab` 调度，要求每三个月的 1 号零时查看正在使用的用户列表。

【操作步骤】

- (1) 再次输入命令 “`crontab -e`”，出现 `vi` 编辑器，按下 “`i`”，屏幕进入文本输入模式。
(2) 在原有内容之后，另起一行，输入 “`0 0 */3 * who >who.log`”。
(3) 最后保存并退出 `vi` 编辑器。
(4) 为立即查看到 `crontab` 调度的结果，切换为超级用户，并适当修改系统时间，如修改为 3 月 31 日 23 点 59 分。最后退回到 `helen` 用户。

```
[helen@localhost ~]$ su -  
Password:  
[root@localhost ~]# date 03312359
```

```
Sat Nov 20 23:59:00 CST 2007  
[root@localhost ~]# exit  
[helen@localhost ~]$
```

- (5) 等待 1 分钟后，查看 `who.log` 文件的内容，如果显示出正确内容，那么说明新增的 `crontab` 调度设置成功。

【操作要求 3】查看 `cron` 调度内容，最后删除此调度。

【操作步骤】

- (1) 输入命令 “`crontab -l`”，查看 `cron` 调度内容。

```
[helen@localhost ~]$ crontab -l
```

```
30 8 * * * ps >ps.log
0 0 * */3 * who >who.log
```

(2) 输入命令“crontab -r”，删除 cron 调度内容。

(3) 再次输入命令“crontab -l”，此时无 cron 调度内容。

```
[helen@localhost ~]$ crontab -r
[helen@localhost ~]$ crontab -l
```

no crontab for helen

4. 系统性能监视

【操作要求 1】利用 Shell 命令监视系统性能

【操作步骤】

(1) 输入命令“top”，屏幕动态显示 CPU 利用率、内存利用率和进程状态等相关信息，

类似图 4-1。

```
top - 21:29:33 up 30 min, 3 users, load average: 0.00, 0.07, 0.00
task: 83 total, 1 running, 0 sleeping, 1 stopped, 0 zombie
Cpu(s): 0.00us, 0.27us, 0.00us, 99.73id, 0.00us, 0.00us, 0.00us, 0.00us
Mem: 51560K total, 43252K used, 82624K free, 46252K buffers
Swap: 111404K total, 0K used, 111404K free, 38624K cached
```

PID	USER	PR	NI	U	ST	RES	SHR	S	CPU	MEM	TIME	COMMAND
2068	helen	15	0	2180	210	708	0	1	0.2	0.00	0.00	top
1	root	15	0	2032	644	540	3	0	0.1	0.00	0.73	init
2	root	RT	0	0	0	0	3	0	0.0	0.00	0.00	migration/0
3	root	39	13	0	0	0	3	0	0.0	0.00	0.01	ksoftirqd/0
4	root	RT	0	0	0	0	3	0	0.0	0.00	0.00	watchdog/0
5	root	10	-5	0	0	0	3	0	0.0	0.00	0.24	events/0
6	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	khelper
7	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	kthread
10	root	10	-5	0	0	0	3	0	0.0	0.00	0.04	kblockd/0
11	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kcpid
72	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	cqevent/0
75	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	khubd
77	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	kseriad
139	root	25	0	0	0	0	3	0	0.0	0.00	0.00	pdflush
140	root	15	0	0	0	0	3	0	0.0	0.00	0.10	pdflush
141	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kswapd0
142	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	syncd
203	root	11	-5	0	0	0	3	0	0.0	0.00	0.00	kpmcswd

图 4-1 按 cpu 使用率显示进程信息

(2) 按下 M 键，所有进程按照内存使用率排列，类似图 4-2。

```
top - 21:30:21 up 31 min, 3 users, load average: 0.00, 0.05, 0.07
task: 83 total, 1 running, 0 sleeping, 1 stopped, 0 zombie
Cpu(s): 0.00us, 1.00us, 0.00us, 98.99id, 0.00us, 0.00us, 0.00us, 0.00us
```

PID	USER	PR	NI	U	ST	RES	SHR	S	CPU	MEM	TIME	COMMAND
2068	helen	15	0	2180	644	540	3	1	0.2	0.00	0.00	top
1	root	15	0	2032	644	540	3	0	0.1	0.00	0.73	init
2	root	RT	0	0	0	0	3	0	0.0	0.00	0.00	migration/0
3	root	39	13	0	0	0	3	0	0.0	0.00	0.01	ksoftirqd/0
4	root	RT	0	0	0	0	3	0	0.0	0.00	0.00	watchdog/0
5	root	10	-5	0	0	0	3	0	0.0	0.00	0.24	events/0
6	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	khelper
7	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	kthread
10	root	10	-5	0	0	0	3	0	0.0	0.00	0.04	kblockd/0
11	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kcpid
72	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	cqevent/0
75	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	khubd
77	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	kseriad
139	root	25	0	0	0	0	3	0	0.0	0.00	0.00	pdflush
140	root	15	0	0	0	0	3	0	0.0	0.00	0.10	pdflush
141	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kswapd0
142	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	syncd
203	root	11	-5	0	0	0	3	0	0.0	0.00	0.00	kpmcswd
206	root	19	-5	0	0	0	3	0	0.0	0.00	0.00	svtsh_0
270	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kmswrd

图 4-2 按内存使用率显示进程信息

(3) 按下 T 键，所有进程按照执行时间排列，类似图 4-3。

```
top - 21:30:56 up 31 min, 3 users, load average: 0.01, 0.05, 0.07
task: 83 total, 1 running, 0 sleeping, 1 stopped, 0 zombie
Cpu(s): 0.00us, 0.00us, 0.00us, 99.99id, 0.00us, 0.00us, 0.00us, 0.00us
```

PID	USER	PR	NI	U	ST	RES	SHR	S	CPU	MEM	TIME	COMMAND
2068	helen	15	0	2180	910	708	0	1	0.2	0.00	0.92	top
2254	root	15	0	1880	544	252	3	0	0.1	0.00	0.25	ps
2377	root	10	0	1920	624	544	0	0	0.1	0.01	0.50	lsd -l /etc/passwd
1	root	15	0	2032	644	540	3	0	0.1	0.00	0.73	init
2	root	RT	0	0	0	0	3	0	0.0	0.00	0.00	migration/0
3	root	39	13	0	0	0	3	0	0.0	0.00	0.01	ksoftirqd/0
4	root	RT	0	0	0	0	3	0	0.0	0.00	0.00	watchdog/0
5	root	10	-5	0	0	0	3	0	0.0	0.00	0.24	events/0
6	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	khelper
7	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	kthread
10	root	10	-5	0	0	0	3	0	0.0	0.00	0.04	kblockd/0
11	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kcpid
72	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	cqevent/0
75	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	khubd
77	root	10	-5	0	0	0	3	0	0.0	0.00	0.00	kseriad
139	root	25	0	0	0	0	3	0	0.0	0.00	0.00	pdflush
140	root	15	0	0	0	0	3	0	0.0	0.00	0.10	pdflush
141	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kswapd0
142	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	syncd
203	root	11	-5	0	0	0	3	0	0.0	0.00	0.00	kpmcswd
206	root	19	-5	0	0	0	3	0	0.0	0.00	0.00	svtsh_0
270	root	20	-5	0	0	0	3	0	0.0	0.00	0.00	kmswrd

图 4-3 按执行时间显示进程信息

(4) 最后按下 P 键，恢复按照 CPU 使用率排列所有进程。

(5) 按下 CTRL+C 组合键结束 top 命令。

【操作要求 2】利用「系统监视器」工具监视 CPU 使用情况。

【操作步骤】

(1) 启动 GNOME 桌面环境，依次单击「系统」菜单=>「管理」=>「系统监视器」，打开「系统监视器」窗口。

(2) 自动显示「资源」选项卡，查看当前 CPU、内存和交换分区、网络历史的使用情况，如图 4-4 所示。



图 4-4 查看资源信息

【操作要求 2】利用「系统监视器」查看当前所有的进程，要求显示出启动进程的用户。

【操作步骤】

(1) 在「系统监视器」窗口单击「进程列表」选项卡，默认显示当前用户启动的所有进程。单击「查看」菜单，选中「所有的进程」单选按钮，并选中「依赖关系」复选框，则显示系统中所有的进程，如图 4-5 所示。



图 4-5 设置查看所有进程



图 4-6 设置进程显示首选项

(2) 单击「编辑」菜单中的「首选项」，弹出「系统监视器首选项」对话框。在「进程」选项卡，选中「进程域」栏的「用户」复选框，要求显示出启动进程的用户，如图 4-6 所示。单击「关闭」按钮，显示进程的各种信息。

【操作要求 3】利用「系统监视器」查看所有的文件系统

【操作步骤】

(1) 在「系统监视器」窗口单击「文件系统」选项卡，显示当前 RHEL Server 5 系统中主要的文件系统。

(2) 单击「编辑」菜单中的「首选项」，弹出「系统监视器首选项」对话框。在「文件系统」选项卡，选中「显示全部文件系统」复选框，要求显示出全部的文件系统，如图 4-7 所示。最后单击「关闭」按钮。

(3) 「文件系统」选项卡显示全部的文件系统的信息，如图 4-8 所示。



图 4-7 设置显示全部文件系统

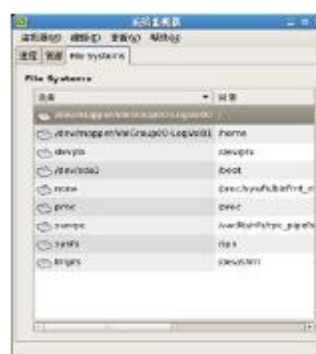


图 4-8 显示文件系统信息

【操作要求 4】利用「系统日志」工具查看系统日志

系统日志文件都保存于/var/log 目录中，包括以下重要的日志文件：

boot.log	记录系统引导的相关信息
cron	记录 cron 调度的执行情况
dmesg	记录内核启动时的信息，主要包括硬件和文件系统的启动信息
maillog	记录邮件服务器的相关信息
messages	记录系统运行过程的相关信息，包括 I/O、网络等
rpm_pkgs	记录已安装的 RPM 软件包信息
secure	记录系统安全信息
Xorg.0.log	记录图形化用户界面的 Xorg 服务器的相关信息

【操作步骤】

超级用户依次单击「系统」菜单=>「管理」=>「系统日志」，打开「系统日志」窗口。可分别查看各类系统日志，如图 4-9 所示。

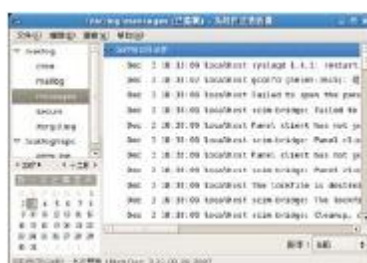


图 4-9 查看系统日志

实验五 用户与组群管理

一 实验要求

- (1) 理解/etc/passwd 和/etc/group 文件的含义。
- (2) 掌握桌面环境下管理用户与组群的方法。
- (3) 掌握利用 Shell 命令管理用户与组群的方法。
- (4) 掌握批量新建用户帐号的步骤和方法。

二 实验内容

1. 桌面环境下管理用户与组群

【操作要求 1】新建两个用户帐号，其用户名为 xuser1 和 xuser2，口令为“e12ut59er”和“wfult28er”。

【操作步骤】

- (1) 以超级用户身份登录 X Window 图形化用户界面，依次单击「系统」菜单=>「管理」=>「用户和组群」，启动「用户管理者」窗口，如图 5-1 所示。



图 5-1 打开用户管理者窗口



图 5-2 添加 xuer1 用户

- (2) 单击工具栏上的「添加用户」按钮，出现「创建新用户」窗口。在「用户名」文本框中输入用户名“xuser1”，在「口令」文本框中输入口令“e12ut59er”，在「确认口令」文本框中再次输入口令“e12ut59er”，如图 5-2 所示，然后单击「确定」按钮，返回「用户管理者」窗口。
- (3) 用同样的方法新建用户 xuser2，完成后「用户管理者」窗口如图 5-3 所示。



图 5-3 完成用户添加

- (4) 依次单击顶部面板的「应用程序」=>「附件」=>「文本编辑器」，启动 gedit 文本编辑器，打开/etc/passwd 和/etc/shadow 文件将发现文件的末尾出现表示 xuser1 和 xuser2 用户帐号的信息。打开/etc/group 和/etc/gshadow 文件将发现文件末尾出现表示 xuser1 和 xuser2 私人组群的信息。
- (5) 按下 CTRL+ALT+F2 组合键切换到第 2 个虚拟终端，输入用户名 xuser2 和相应的口令可登录 Linux 系统，说明新建用户操作已成功。
- (6) 输入“pwd”命令，屏幕显示用户登录后进入用户主目录“/home/xuser2”，操作内

容如下所示。

```
localhost login : xuser2
Password :
[xuser2@localhost ~]$ pwd
/home/xuser2
```

(7) 输入“exit”命令，xuser2 用户退出登录。

(8) 按下 ALT+F7 组合键返回 GNOME 桌面环境。

【操作要求 2】 锁定 xuser2 用户帐号

【操作步骤】

- (1) 在「用户管理者」窗口选中 xuser2 用户帐号，单击工具栏上的「属性」按钮，打开「用户属性」窗口。
- (2) 选中「帐号信息」选项卡让「本地口令被锁」复选框被选中，如图 5-4 所示。单击「确定」按钮，返回「用户管理者」窗口。



图 5-4 锁定 xuer2 用户

- (3) 按下 CTRL+ALT+F2 组合键，再次切换到第 2 个虚拟终端，输入用户名 xuser2 和相应的口令，发现 xuser2 用户无法登录 Linux 系统，说明 xuser2 用户账号的确已被锁定，操作内容如下所示：

```
localhost login : xuser2
Password :
Login incorrect
```

- (4) 按下 ALT+F7 组合键再次返回 GNOME 桌面环境。

【操作要求 3】 删除 xuser2 用户

【操作步骤】

- (1) 在「用户管理者」窗口，单击「编辑」菜单的「首选项」，弹出「首选项」对话框，不选中「隐藏系统用户和组」复选框，如图 5-5 所示，最后单击「关闭」按钮。此时「用户」选项卡中显示包括超级用户和系统用户在内的所有用户，如图 5-6 所示。



图 5-5 取消隐藏系统用户和组



图 5-6 显示所有用户

- (2) 在「搜索过滤器」文本框中输入“x*”并按下 Enter 键，则仅显示以 x 为首字母的用户，如图 5-7 所示。
- (3) 选中 xuser2 用户，单击工具栏上的「删除」按钮，弹出对话框，如图 5-8 所示，单击「是」按钮，返回「用户管理者」窗口，发现 xuser2 用户已被删除。



图 5-7 显示 x 开头的用户



图 5-8 确认删除 xuser2 用户

- (4) 在「搜索过滤器」文本框中输入“*”并按下 Enter 键，则显示所有用户。

【操作要求 4】新建两个组群，分别是 myusers 和 temp。

【操作步骤】

- (1) 在「用户管理者」窗口选中「组群」选项卡，当前显示出所有组群。
- (2) 单击工具栏上的「添加组群」按钮，出现「创建新组群」对话框。在「组群名」文本框中输入“myusers”，如图 5-9 所示，单击「确定」按钮，返回「用户管理者」窗口。
- (3) 用相同的方法新建 temp 组群，完成后「用户管理者」窗口如图 5-10 所示。



图 5-9 新建组群



图 5-10 完成组群新建

【操作要求 5】修改 myusers 组群属性，将 xuser1 和 helen 用户加入 myusers 组群。

【操作步骤】

- (1) 从「组群」选项卡中选择 myusers 组群，单击工具栏上的「属性」按钮，弹出「组群属性」窗口。
- (2) 选择「组群用户」选项卡，选中 helen 和 xuser1 前的复选框，设置 helen 用户和 xuser1 用户的 myusers 组群的成员，如图 5-11 所示。单击「确定」按钮，返回「用户管理者」窗口，如图 5-12 所示。



图 5-11 向组群添加用户



图 5-12 组群及组群成员

【操作要求 6】删除 temp 组群

【操作步骤】

从「组群」选项卡中选择 temp 组群，单击工具栏上的「删除」按钮，出现确认对话框，单击「是」按钮即可。

2. 编辑用户配置文件

【操作要求 1】新建用户配置文件 myusers-profile

【操作步骤】

- (1) 依次单击「系统」菜单=>「管理」=>「用户配置文件编辑器」，打开「User Profile Editor」窗口，如图 5-13 所示。
- (2) 单击「添加」按钮，弹出「Add Profile」窗口，在「Profile name」文本框中输入用户配置文件名“myusers-profile”，如图 5-14 所示。单击「添加」按钮，回到「User Profile Editor」窗口，如图 5-15 所示。



图 5-13 User Profile Editor 窗口



图 5-14 设置用户配置文件名



图 5-15 完成添加用户配置文件名

【操作要求 2】设置 myusers-profile 用户配置文件的内容：应用程序的默认字体为中易宋体 18030，桌面背景为花园。

【操作步骤】

- (1) 在「User Profile Editor」窗口选中“myusers-profile”文件，单击「编辑」按钮，出现「编辑配置文件 myusers-profile」窗口。
「编辑配置文件 myusers-profile」窗口与当前实际的桌面几乎一样，利用这个窗口的系统菜单可设置用户配置文件的实际内容。
- (2) 在「编辑配置用户」窗口中依次单击「系统」菜单=>「首选项」=>「字体」，打开「字体首选项」对话框，如图 5-16 所示。单击应用程序字体的字体列表，出现「拾取字体」对话框（参见图 5-17），从「字体族」选择“中易宋体 18030”，并单击「确定」按钮。



图 5-16 字体首选项



图 5-17 修改字体

- (3) 回到「字体首选项」对话框，如图 5-18 所示，此时窗口中的字体发生变化，单击窗口右上角的关闭按钮，关闭此对话框。
- (4) 在「编辑配置用户」窗口中依次单击「系统」菜单=>「首选项」=>「桌面背景」，打开「桌面背景首选项」对话框，选择“花园”。此时「编辑配置用户」窗口的桌面也发生变化，如图 5-19 所示。最后单击「关闭」按钮。



图 5-18 完成字体设置



图 5-19 完成字体设置

- (5) 单击「编辑配置用户」窗口「配置文件」菜单的「保存」项，保存用户配置文件的修改内容。最后单击「编辑配置用户」窗口右上角的关闭按钮，回到「User Profile Editor」窗口。

【操作要求 3】设置 xuser1 的用户配置文件为 myusers-profile

【操作步骤】

- (1) 在「User Profile Editor」窗口选中 myusers-profile 文件，单击「Users」按钮，出现「配置文件 myusers-profile 的用户」对话框。
- (2) 选中 xuser1 用户的复选框，如图 5-20 所示，最后单击「关闭」按钮。



图 5-20 完成字体设置

- (3) 单击「系统」菜单的「注销」项，超级用户退出 GNOME 桌面环境。
- (4) 以 xuser1 用户登录，并启动 GNOME 桌面环境，查看应用程序的字体和桌面环境。
- GNOME 的用户配置文件编辑功能允许超级用户创建多种用户配置方案，分配给不同类型的普通用户。

3. 利用 Shell 命令管理用户与组群

【操作要求 1】新建一名为 duser 的用户，其口令是“tdd63u2”，主要组群为 myusers。

【操作步骤】

- (1) 按下 CTRL+ALT+F3 组合键，切换到第 3 个虚拟终端，以超级用户身份登录。
- (2) 输入命令“useradd -g myusers duser”，建立新用户 duser，其主要组群是 myusers。

- (3) 为新用户设置口令，输入命令“passwd duser”，根据屏幕提示输入两次口令，最后屏幕提示口令成功设置信息，如下所示。

```
[root@localhost ~]# useradd -g myusers duser
[root@localhost ~]# passwd duser
Changing password for user duser.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

设置用户口令时输入的口令在屏幕上并不显示出来，而输入两次的目的在于确保口令没有输错。

- (4) 输入命令“cat /etc/passwd”，查看 /etc/passwd 文件的内容，发现文件的末尾增加 duser 用户的信息。
- (5) 输入命令“cat /etc/group”，查看 /etc/group 文件的内容，发现文件内容未增加。
- (6) 按下 ALT+F4 组合键，切换到第 4 个虚拟终端，输入 dusr 用户名和口令可登录 Linux 系统。
- (7) 输入“exit”命令，duser 用户退出登录。

【操作要求 2】将 duser 用户设置为不需口令就能登录。

【操作步骤】

- (1) 按下 ALT+F3 组合键，切换到正被超级用户使用的第 3 个虚拟终端。
- (2) 输入命令“passwd -d duser”，如下所示。

```
[root@localhost ~]# passwd -d duser
Removing password for user duser.
passwd: Success
```

在实际应用中此功能必须谨慎使用，因为不需口令的用户帐号一旦被窃可能造成系统入侵的严重后果。

- (3) 按下 ALT+F3 组合键，再次切换到第 3 个虚拟终端，在“Login:”后输入用户名“duser”，按下 Enter 键就直接出现 Shell 命令提示符，说明 duser 用户不需口令即可登录，如下所示。

```
localhost login : duser
Last login : Thu Nov 22 04:06:15 on tty4
[duser@localhost ~] $
```

【操作要求 3】查看 duser 用户的相关信息

【操作步骤】

在第 3 个虚拟终端输入命令“id duser”，显示 duser 用户的用户 ID (UID)、主要组群的名称和 ID (GID)，如下所示。

```
[root@localhost ~]# id duser
uid=502(duser) gid=502(myusers) groups=502(myusers) context=system_u:
system_r:unconfined_t: SystemLow-SystemHigh
```

【操作要求 4】从普通用户 duser 切换为超级用户

【操作步骤】

- (1) 第 4 个虚拟终端当前的 Shell 命令提示符为“\$”，表明当前用户是普通用户。
- (2) 输入命令“ls /root”，屏幕上没有出现/root 目录中文件和子目录的信息，而是出现提示信息，提示当前用户没有查看/root 目录的权限。
- (3) 输入命令“su -”或者是“su - root”，屏幕提示输入口令，此时输入超级用户的

口令，验证成功后 Shell 提示符从 “\$” 变为 “#”，说明已从普通用户转换为超级用户。

- (4) 再次输入命令 “ls /root”，可查看/root 目录中文件和子目录的信息，相关操作如下所示。

```
[duser@localhost ~]$ ls /root
ls : /root : Permission denied
[duser@localhost ~]$ su -
Password :
[root@localhost ~]# ls /root
```

```
anaconda-ks.cfg  Desktop  install.log  install.log.syslog
```

- (5) 输入 “exit” 命令，回到普通用户的工作状态。

- (6) 输入 “exit” 命令，duser 用户退出登录。

【操作要求 5】 一次性删除 duser 用户及其工作目录

【操作步骤】

- (1) 按下 ALT+F3 组合键，切换到正被超级用户使用的第 3 个虚拟终端。

- (2) 输入命令 “userdel -r duser”，删除 duser 用户。

处于登录状态的用户不能删除。如果在新建这个用户时还创建了私人组群，而该私人组群当前又没有其他用户，那么在删除用户的同时也将一并删除这一私人组群。

- (3) 输入命令 “cat /etc/passwd”，查看/etc/passwd 文件的内容，发现 duser 的相关信息已消失。

- (4) 输入命令 “ls /home”，发现 duser 的主目录/home/duser 也不复存在。

【操作要求 6】 新建组群 mygroup

【操作步骤】

- (1) 在超级用户的 Shell 提示符后输入命令 “groupadd mygroup”，建立 mygroup 组群。

- (2) 输入命令 “cat /etc/group”，发现 group 文件的末尾出现 mygroup 组群的信息。

- (3) 输入命令 “cat /etc/gshadow”，发现 gshadow 文件的末尾也出现 mygroup 组群的信息。

【操作要求 7】 将 mygroup 组群改名为 newgroup

【操作步骤】

- (1) 输入命令 “groupmod -n newgroup mygroup”，其中-n 选项表示更改组群的名称。

- (2) 输入命令 “cat /etc/group”，查看组群信息，发现原来 mygroup 所在行的第一项变为 “newgroup”。

【操作要求 8】 删除 newgroup 组群

【操作步骤】

超级用户输入 “groupdel newgroup” 命令，删除 newgroup 组群。

4. 批量新建多个用户帐号

【操作要求】 为全班同学 20 位同学创建用户帐号，用户名为 “s” +学号的组合，其中班级名册中第一位同学的学号为 080101。所有同学都属于 class0801 组群。所有同学的初始口令为 111111。

【操作步骤】

- (1) 以超级用户身份登录，输入命令 “groupadd -g 600 class0801”（假设值为 600

- 的 GID 未被使用)，新建全班同学的组群 class0801，
- (2) 输入命令“vi student”，新建用户信息文件。
 - (3) 按下“i”键，切换为 vi 的文本编辑模式，输入第一行信息：s080101:x:601:600::/home/s080101:/bin/bash”。
 - (4) 按下 ESC 键，切换到命令行模式，拖动鼠标，将整行选中，如图 5-21 所示，然后按下字母键 y 两次。也就是将当前选中的行放到 vi 的暂存区域（类似于 Windows 的剪贴板）。



图 5-21 选中行

- (5) 然后按下字母键 p，就复制一行信息，如图 5-22 所示，重复此操作 19 次，然后部分修改每位同学用户信息不同的地方。



图 5-22 粘贴行

- (6) 最后编辑完成的文件，如图 5-23 所示，为节约篇幅仅显示前 10 位同学信息。最后保存并退出 vi。

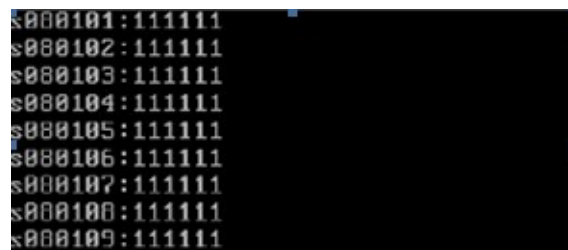


图 5-23 用户信息文件

- (7) 输入命令“vi stu-passwd”，新建用户口令文件。
- (8) 按下“i”键，切换为 vi 的文本编辑模式，输入第一行信息：“s080101:111111”，即所有同学的初始口令为 111111。按下 ESC 键，切换到命令行模式，拖动鼠标，将整行选中，然后按下字母键 y 两次，复制行。
- (9) 连续按 p 键 19 次，就可复制出 19 行信息，然后修改成正确的用户名，如图 5-24 所示。

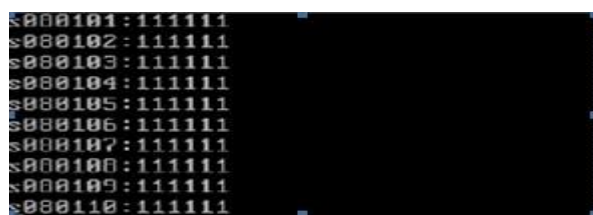


图 5-24 用户口令文件

- (10) 输入命令“newusers < students”，批量新建用户帐号。
- (11) 输入命令“pwunconv”，暂时取消 shadow 加密。
- (12) 输入命令“chpasswd <stu-passwd”，批量新建用户的口令。
- (13) 输入命令“pwconv”，进行 shadow 加密，完成批量创建用户帐号工作。

- (14) 输入命令“cat /etc/passwd”，查看/etc/passwd 文件将发现所有的用户帐号均已建立。
- (15) 可尝试以新建的用户名登录，并应该及时修改用户的口令。

使用此方法批量创建的用户登录时的命令提示符，不是默认的[用户名@localhost ~]\$，而是-bash-3.1\$。如果希望使用默认的命令提示符，可将采用 useradd 命令新建的用户，如 helen 的用户主目录中的.bash_profile 和.bashrc 文件复制到批量创建的用户主目录即可。

实验六 SHELL 编程

一. 实验目的

掌握vi 的三种工作方式，熟悉vi 编辑程序的使用。
学习Shell 程序设计方法。掌握编程要领。

二. 实验准备

复习操作系统课程相关的用户接口概念。
熟悉本《实验指导》第三、四部分。

三. 实验内容

学习使用vi 编辑程序。
编写Shell 程序。
将程序文件设置为可执行文件（用chmod 命令）。
在命令行方式中运行Shell 程序。

四. 实验步骤

1. 按本《实验指导》第三部分的内容。熟悉vi 的三种工作方式。熟悉使用各种编辑功能。

思考：试一试vi 的三种工作方式各用在何时？用什么命令进入插入方式？怎样退出插入方式？文件怎样存盘？注意存盘后的提示信息。

2. 创建和执行Shell 程序

用前面介绍的Vi 或其他文本编辑器编写Shell 程序，并将文件以文本文件方式保存在相应的目录中。

用chmod 将文件的权限设置为可执行模式，如若文件名为shdemo.h,则命令如下：

`$ chmod 755 shdemo.h` (文件主可读、写、执行，同组人和其他人可读和执行)

在提示符后执行Shell 程序：

`$ shdemo.h` (直接键入程序文件名执行)

或 `$ sh shdemo.h` (执行Shell 程序)

或 `$.shdemo.h` (没有设置权限时可用点号引导)

3. 用vi 编写《实验指导》“第四部分Shell 程序设计”中的例1（假设文件名为prog1.h），练习内部变量和位置参数的用法。

用chmod将文件的权限设置为可执行模式，并在提示符后键入命令行：

`$/prog1.` #没有参数

或 `$sh prog1.`

屏幕显示：

Name not provided

在提示符后键入命令行：

```
$/prog1.h Theodore
```

#有一个参数

屏幕显示：

```
Your name is Theodore
```

#引用\$1 参数的效果

4. 进一步修改程序prog1.h，要求显示参数个数、程序名字，并逐个显示参数。
5. 修改例1程序（即上面的 prog1.h），用read命令接受键盘输入。若没有输入显示第一种提示，否则第二种提示。
6. 用vi 编写《实验指导》“第四部分 Shell 程序设计”中的例2、例3，练习字符串比较运算符、数据比较运算符和文件运算符的用法，观察运行结果。
7. 修改例2程序，使在程序运行中能随机输入字符串，然后进行字符串比较。
8. 修改例3程序，使在程序运行中能随机输入文件名，然后进行文件属性判断。
9. 用vi 编写《实验指导》“第四部分 Shell 程序设计”中的例4、例5、例6、例7，掌握控制语句的用法，观察运行结果。
10. 用vi 编写《实验指导》“第四部分 Shell 程序设计”中的例8 及例9掌握条件语句的用法，函数的用法，观察运行结果。
11. 编程，在屏幕上显示用户主目录名（HOME）、命令搜索路径（PATH），并显示由位置参数指定的文件的类型和操作权限。

📖 思考：到此为止你对Shell 有所认识了吧？怎么样？自己再编两个程序：

- ① 做个批处理程序，体会一下批处理概念。
- ② ② 做个菜单，显示系统环境参数。将此程序设置为人人可用。

五. 讨论

1. Linux 的Shell 有什么特点？
2. 怎样进行Shell编程？如何运行？有什么条件？
3. vi 编辑程序有几种工作方式？查找有关的详细资料，熟练掌握屏幕编辑方式、转移命令方式以及末行命令的操作。学习搜索、替换字符、字和行，行的复制、移动，以及在vi中执行Shell命令的方式。
4. 编写一个具有以下功能的Shell程序。
 - (1) 把当前目录下的文件目录信息输出到文件 filedir.txt 中；
 - (2) 在当前目录下建立一个子目录，目录名为 testdir2 ；
 - (3) 把当前目录下的所有扩展名为 c 的文件以原文件名复制到子目录testdir2中；
 - (4) 把子目录中的所有文件的存取权限改为不可读。（提示：用 for 循环控制语句实现，循环的控制列表用 'ls' 产生。）
 - (5) 在把子目录 testdir2 中所有文件的目录信息追加到文件 filedir.txt 中；
 - (6) 把你的用户信息追加到文件 filedir.txt 中；
 - (7) 分屏显示文件 filedir.txt

第二部分 Linux 命令界面使用

一. 登录进入命令行实验环境:

Linux 是多用户操作系统。进入系统的用户都必须有个人帐号和口令。

命令行交互界面, 按以下步骤操作:

- 系统启动, 进入 Linux 环境。
- 登录: 在 Linux 的 “login: ” 提示后输入用户名和口令。
- 登录成功: 普通用户提示符为 \$
超级用户提示符为 #

说明: 若使用的是主机终端, 可以用<Alt+ F1>、< Alt+ F2>、----< Alt+F6>切换屏幕, 转换到其它虚拟终端, 以实现多个用户同时登录到同一台计算机。

利用 TCP/IP 网络连接终端

- 在 Win95/98 “开始” 菜单的 run 对话框中 (或 MSDOS 提示下) 输入:
telnet IP 地址(或域名)
- 在 telnet 窗口中的 “login: ” 提示后输入用户名和口令, 登录进入系统。

注: 在 Windows 的 telnet 窗口中, 勾选 “终端/首选” 的 “VT100” 选项, 可使光标键有效。

二. 退出系统:

在提示符后键入 logout 或按 Ctrl+D。

【说明】 登录后, Linux 将用户置于登录用户的主目录中(通常在/home 中的), 启动一个 Shell 进程与用户交互。Shell 实际上是操作系统与用户的交互接口, 操作系统通过 Shell 接受用户的请求, 建立相应的命令处理进程, 处理用户请求。

三. 基本命令

用户在提示符后可以输入 Linux 命令进行操作。一些简单而基本的 Linux 命令介绍如下: (注意: Linux 命令大小写敏感)

1. 查看信息

(1) 显示联机手册

man [命令]

注: Linux 的所有命令都可以通过 man 查看联机手册。

(2) 显示联机帮助

[命令] - -help

注: 任何命令都可以用 - -help 选项显示帮助信息。

(3) 显示当前目录

pwd

- (4) 显示系统日期和时间

date

- (5) 查看当前注册到系统的每个用户的信息

who

注：结果显示的第一个字段是登录用户的标识符，第二个字段显示的是登录用户使用的终端，第三、四字段显示的是日期，第五字段显示的是时间，若从不同网络远程登录到用户主机上，在第六个字段上显示带地址的计算机名。

- (6) 显示本用户信息

who am i

- (7) 显示目前注册的用户及用户正在运行的命令

w [选项] [用户名]

选项：

-h 不显示起始行

-s 按短格式显示

-l 按长格式显示（默认）

- (8) 显示用户名和用户 id、组名与组 id

id

- (9) 查看日历

cal [月] [年]

- (10) 显示环境变量

env

- (11) 显示系统状态

vmstat 或 top

- (12) 清除屏幕

clear

2. 目录操作

- (1) 列目录

ls [选项] [文件名---]

选项：

-l 以长格式列出文件信息。显示包括以下内容：文件类型及存取控制权限、链

接计数、文件属主、文件属组、文件大小、文件的最后更改日期以及文件名。

-a 列出相应目录下的所有文件。

-c 根据文件的最后更改时间分类显示文件。

-g 列出各个文件的组权限。

-i 在第一列显示 i 节点号。

说明： • 几个选项同时使用时只要用一个“-”引导。

• 文件类型有普通文件（~）、目录文件（d）、块设备特别文件（b）、字符设备特别文件（c）、命名管道文件（p）等。

• “存取控制模式”指对不同用户分配不同的操作权。Linux 文件系统将用户分成三类，即文件主、同组人、其他人。每种人可以行使的操作有三种，即读（r）、写（w）、执行（x）。

• 权限分别用二进制位来置位，“1”表示有权、“0”表示无权。

r	w	x	r	w	x	r	w	x
---	---	---	---	---	---	---	---	---

文件主	同组人	其他人
-----	-----	-----

如用 -rwxr-x--x（八进制为 0751）表示这是一个普通文件，文件主可以读、写、执行，同组人可以读和执行、其他人只能执行。

• r、w、x 对于不同类型的文件有不同的含义。见下表：

例：ls -al (以长格式列出当前目录中所有文件信息)

(2) 改变当前目录

cd 目录名

说明： 目录名”是用户要进入的另一个目录路径。若省略目录名则返回用户的主目录。

例：\$ cd /home (当前目录改为/home)

(3) 创建目录

`mkdir [-m 存取控制模式] 目录名`

说明:

- “存取控制模式”指对不同用户分配不同的操作权。若省略则为文件主有全部权限，同组人和其它人只可读与执行（即 0755）。
- 对要创建目录的父目录，用户应有相应的权限。

(4) 删除目录

`rmdir 目录名`

说明：非空目录必须先删除该目录下的所有子目录和文件。

3. 文件操作

(1) 显示或创建一个文件

`cat [>]文件名`

说明：“文件名”应含有文件名和文件所在的目录名。

若不带“>”符号则显示文件说明所指示的文件的内容。

若带“>”符号则为创建一个新文件。.

(2) 分页浏览文件

`more [文件名]`

说明:

“文件名”应含有文件名和文件所在的目录路径名。

`more` 在显示文件时满一屏自动暂停，待用户按下<SPACEBAR>键后再继续。

此命令省略文件名则可用在管道命令中分页显示命令行的输出。

退出 `more` 可用<ctrl+c>或在“:”提示后按 `q` 命令。

(3) 显示文件头部

`head [-显示行数] 文件名`

(4) 显示文件尾部

`tail [+起始行数] 文件名`

`tail [-起始行数] 文件名`

说明：“+起始行数”表示从文件头开始的行数显示至文件尾；“-起始行数”表示从文件尾开始倒数若干行显示至文件尾。

(5) 复制文件

`cp [选项] 源文件 目标文件`

选项:

`-i` 覆盖现有文件之前给出交互提示，以供用户确认。

`-p` 保留原始文件中的属性。

`-r` 拷贝时保留原始文件的目录结构。

说明：• 几个选项同时使用时只要用一个“-”引导。

- “源文件”和“目标文件”都应含有文件名和文件所在的目录路径名。

(6) 文件链接

`ln 文件名 新文件名`

说明:

这是 UNIX 系统最有特色的命令之一。该命令的作用是用多个（新）文件名与一个文件实体建立链接（也就是一个文件起了多个名字或说多个文件名使用同一个 `i` 节点）。命令执行后，可用 `ls -il` 查看第一列的 `i` 节点号是相同的而且、链接数为大于原来的数字。这就是所谓的“硬链接”。

(7) 移动或重命名文件

`mv` [选项] 源文件 目标文件

选项:

- i 覆盖现有文件之前给出交互提示, 以供用户确认。
- r 拷贝时保留原始文件的目录结构。

说明:

- “源文件”和“目标文件”都应含有文件所在的目录路径名和文件名。
- 若目标文件名是原来不存在的则为重命名。

(8) 删除文件

`rm` [选项] 文件名|目录名

选项:

- f 强制删除, 即忽略各种错误和警告。
- r 递归地进入到子目录并删除文件。
- i 以交互方式删除文件, 在删除前要求用户确认。

说明: 删除文件时“文件名”应含有文件所在的目录路径。

(9) 查找文件

`find` 目录 [条件] [操作]

条件:

- | | |
|-----------------------------|--------------------------------------------------------------|
| <code>-atime</code> [+ -] n | 查找存取时间早于 n 天的 (+) 或最近 n 天的 (-) 文件。 |
| <code>-mtime</code> [+ -] n | 查找修改时间早于 n 天的 (+) 或最近 n 天的 (-) 文件。 |
| <code>-name</code> 文件名 | 查找指定文件名的文件。 |
| <code>-type</code> 字符 | 查找指定字符所表示的文件类型: f 为普通文件、d 为 C 目录、c 为字符设备文件、b 为块设备文件、p 为管道文件。 |

操作:

- | | |
|-----------------------|-----------------------------------|
| <code>-print</code> | 显示找到的文件名及路径。 |
| <code>-exec</code> 命令 | 对找到的文件执行 Shell 命令。 |
| <code>-ok</code> 命令 | 类似于 <code>-exec</code> 但先显示命令和参数。 |

例:

- 显示当前目录中所有以“.c”结尾的文件

```
find -name "*.c" -print
```

- 删除一星期以前的新闻

```
find /usr/news -mtime +7 -exec rm {} \;
```

- 查看当前目录下 2 天以前产生的文件

```
find . -atime +2
```

4. 修改文件属性

(1) 改变文件的所有者

`chown 用户名 文件名`

说明：“文件名”应含有文件名和文件所在的目录名。

“用户名”应为已在系统中具有帐号的用户名。

(2) 改变文件的组标识

`chgrp 组名 文件名`

说明：“文件名”应含有文件名和文件所在的目录名。

“组名”应为已在系统中具有帐号的组。

(3) 改变文件权限

`chmod 访问模式 文件名|目录名`

例：`chmod 751 jkl.o`

设置 `jkl.o` 文件的访问模式为，文件主可读、可写、可执行，同组人可读、可执行，其他人只可执行。

5. 建立和安装软盘文件系统（操作者要有 root 权限）

Linux 允许用户在块设备上建立文件系统。如给计算机增加一个新的磁盘时，就要在这个磁盘上建立一个文件系统。在磁盘上建立了文件系统后，随时都可以将它安装到 Linux 的文件系统中去。此后，用户就可以用访问目录的方法去访问该磁盘，从而扩大了系统可用存储空间。

在磁盘上建立文件系统，必须先对磁盘进行分区。

(1) 建立硬盘分区

`fdisk [设备名]`

说明：

普通用户没有权限，将不能执行此命令。

`fdisk` 命令的子命令可参考联机手册或相应的参考资料。

(2) 软盘格式化

`fdformat /dev/fd0H1440`

(3) 创建软盘上的文件系统

`mkfs -t ext2 /dev/fd0`

说明：

建立软盘文件系统，盘中原来的数据将会被破坏。

(4) 安装软盘文件系统

`mount [选项] 设备 目录`

选项：

`-r` 被安装的文件系统只有读权。

`-w` 被安装的文件系统只有读权和写权。

`-t fs-type` 制定被安装的文件系统的类型。

设备:

系统的设备文件名。如: /dev/fd0。

目录:

通常这个目录是空的, 而且是专门为安装文件系统而设置的, 如: /mnt/fd0。。

例:

```
mount -t ext2 /dev/fd0 /mnt/fd0
```

说明:

①至此凡是要用软盘时都以 /mnt/fd0 表示, 如查看软盘文件目录就用

```
ls -l /mnt/fd0
```

```
cp ./jkl.c /mnt/fd0
```

②未经卸载软盘, 文件系统盘片将无法取出。

一旦建立了文件系统, 就可以把它安装到一个可存取的目录下。

③不带参数的 mount 命令, 将报告系统中已安装的所有文件系统。

(5) 卸载软盘文件系统

```
umount /dev/fd0
```

6. 进程管理

(1) 报告进程状态

```
ps [选项]
```

选项:

-l 长列表

-u 显示用户名和起始时间

-j 按作业格式

-s 按信号格式

-v 按虚拟存储器格式

-a 也显示其他用户进程

-x 显示不带控制终端的进程

-c 列出命令名

-e 显示环境

-w 用宽格式显示

-r 只显示正在运行的进程

-n 为 USER 和 WCHAN 提供数字显示

-t ttyxx 只显示 ttyxx 控制的进程

报告列: PID 进程号

PRI 进程优先级

NI Linux 进程的 nice 值

SIZE 虚拟映象的大小 (文本+数据+栈)

RSS 驻留空间大小

WCHAN 进程等待内核事件名

STAT	
R	进程状态
S	可执行的
D	睡眠
T	不间断睡眠
Z	停止或跟踪
W	僵死
TT	进程没有驻留页
PAGEN	进程的控制 tty
TRS	造成从磁盘读取页的页面出错号
SWAP	文本驻留大小
	交换设备上的 K 字节数

(2) 传送信号给当前运行的进程

kill [-信号] 进程号 (传送信号给指定进程)

kill -l (显示信号数和信号名表)

说明：在用户程序中采用系统调用 kill，也可完成将信号发送给某个进程。

(3) 等待进程完成

wait [n]

说明：Shell 等待进程号为 n 的后台进程终止，并报告终止状态。若缺省 n，则等待所有后台进程终止。

(4) 挂起一段时间

sleep n

说明：n 为秒数。

例：

(sleep 60; command) &

60 秒钟后执行 command 命令，其中 command 表示某个具体的命令名，&表示该命令后台执行。

7. 信息传递

(1) 与其他用户建立对话

talk 用户名 [终端名]

说明：可用 ctrl-d 或 Ctrl-z 或 Ctrl-c 来结束本次通话。

(2) 向其他用户发终端信息

write 用户名 [终端]

说明：命令键入后，用户可在键盘上输入任何想要通信的信息，直到按下 Ctrl-d 为止。

(3) 允许或禁止其他用户发信息到本终端

mesg [y | n]

说明： 可选 y（允许）或 n（禁止）。

- (4) 给所有现在登录系统的用户发广播

wall [信息]

四. 特殊命令

1. 后台命令 “&”

在 Linux 中 Shell 的前台命令指那些输入命令行后必须得到系统响应，然后才能启动下一条命令的执行。后台命令则不要求系统马上执行，也就是，启动了后台命令后，即使未得到命令响应的情况下仍可以输入启动下一条命令。后台命令的输入只要在命令行后加上后台命令符号 “&” 即可。一般不要求立即响应或运行时间较长的命令可用后台方式运行。

例如：要求系统在空闲时编译程序 prog.c

```
cc prog.c&
```

2. 文件名通配符号 “?” 和 “*”

? 文件名中该字符位置可以对应任何一个字符。

* 文件名中该字符位置开始可以对应任何一个或几个字符。

3. 输入输出重定向符 “<”、“>” 和 “>>”

Linux 系统中通过进程执行一个命令时，系统自动地将键盘输入定义为标识符为 0 的特殊文件且称为标准输入；将屏幕输出定义为标识符为 1 的特殊文件并称为标准输出；将输出到终端显示器的错误信息定义为标识符为 2 的特殊文件称为标准错误信息输出。

在执行命令时，若要改变输入输出文件，可以使用输入输出重定向符号。

“<” 表示输入重定向，如：a<b 表示将文件 b 的内容作为输入送到文件 a；

“>” 表示输出重定向，如：c>d 表示将 c 的内容输出到文件 d，c 可以是命令行结果或文件；

“>>” 也是输出重定向，如：e>>f 表示将 e 的内容顺序追加到文件 f 后半部。

e 可以是命令行结果或文件。若文件 f 不存在，则自动建立并追加。

4. 管道命令 “|”

Linux 中管道命令是一种利用系统缓冲区的方式。这种方式就是把一个命令的输出结果放在系统缓冲区内直接作为下一个命令的输入。管道命令的使用格式是：

```
命令 1 | 命令 2 | 命令 3 ----
```

例：ls -l 列目录的结果送到命令 more 处理后分屏显示。

```
ls -l | more
```

ps 命令将系统进程的状态送到命令 more 处理后分屏显示。

```
ps -a | more
```

Linux 系统的 Shell 还可以让用户使用 Shell 程序设计语言，把命令编制成程序后批处理执行。有关 Shell 程序的设计请参考本实验指导第四部分 “Shell 程序设计语言”；有关程序的编辑方式请参考本实验指导第三部分 “编辑程序 Vi”。

第三部分 编辑程序 Vi

vi 是 Linux 系统中一个传统的全屏幕文本编辑程序。利用它可以编写各种文本文件。在 Linux 的电子邮件程序 mail 中，就可以直接引用 vi 来阅读和编辑信件。

vi 编辑程序中有三种不同的操作方式。当用户正在编辑一个文件时，vi 可能处在插入方式、转义命令方式或末行命令方式中的一种方式下(有的教材将转义命令方式和末行命令方式统称为命令方式)。

插入方式

插入方式可用插入命令(见表 2)的任何一个进入，屏幕下方有一行“-----Insert-----”字样表示。插入时用户输入的任何字符都认为是文本的内容，且可以用退格键来纠正错误。在插入方式中按一下<ESC>，即退出插入方式，进入转义命令方式。

转义命令方式

刚进入 vi 或退出插入方式，即为转义命令方式。这时键入的任何字符转义为特殊功能，如：移动、删除、替换等。大多数转义命令由一个或两个字母组成，操作时没有提示符，而且输入命令不需要按<ENTER>。

末行命令方式

在转义命令方式中，按冒号“:”就进入末行命令方式。屏幕最末一行的行首显示冒号作为命令提示。命令行输入后按<ENTER>开始执行。此时用户可进行文件的全局操作，如：全局查找、替换、文件读、写等。

1. 启动和退出 vi

通常在 Linux 提示符后可用以下命令调用 vi：

```
vi filename
```

这里，filename 必须遵守 Linux 文件命名的规定。若 filename 已经存在，则编辑器将取出该文件，并将光标放置在文件首行的开始处。如果 filename 不存在，则编辑器将以此名产生一个空文件，并将光标放置在屏幕的左上角。

当 vi 被激活时，屏幕上将尽可能多地显示出被编辑文件的内容。屏幕的最下一行显示有关的控制信息。文本末尾以后的空白区的左边第一列将显示符号“~”，表示这是文本输入区。

下面的命令调用 vi 编辑器，但是开始时是一个没有命名的空白文件的顶部：

```
vi
```

当然如果想保存这个文件时，必须给它命名，这一操作将在本节的后面作介绍。

与其他的文字处理程序不同，vi 可以处理一系列文件，例如：

```
vi filename1 filename2
```

在这种情况下，vi 首先处理 filename1，然后处理 filename2。

退出 vi 的最简单的方法是键入命令：

```
: q
```

其中冒号使操作进入末行命令方式，q 是退出命令，它将使用户退回到 Shell 中。如果对文件没有作太多的改动，则 vi 会接受这个命令。但是，如果对文件作了较大的改动而又没有重新存盘，则 vi 将拒绝此命令。在这种情况下需作选择，可将文件写入磁盘使这些改动得到存储这时候用命令：

```
: wq
```

保存并退出。也可键入命令：

```
: q!
```

这个命令意为放弃文件保护功能，强制退出。使用它时一定要考虑好。

2. 写文件

由于 vi 缓冲区的内容是保存在内存中，所以不论有意或无意地退出 Linux 后，这些内容都会丢失。因此有规律地将缓冲区的内容写（存）到磁盘上非常重要，不这样做往往会导致灾难性的后果，而且也没有理由不定期地存盘。存盘过程很简单，只需键入命令：

```
: w
```

命令 w 将文件以当前名字存入磁盘，并覆盖了文件先前的副本。但 w 命令不影响缓冲区的内容。

如果要将文件以不同的名字保存，例如 newfile，需键入命令：

```
: w newfile
```

如果不存在名为 newfile 的文件，vi 编辑器将存入文件并给出文件的大小。如果这个文件已经存在，那么 vi 会通知你，但并不刷新文件的内容。如果想刷新已存在的文件的内容，则可键入命令：

```
: w! newfile
```

同样，感叹号“！”放弃了标准的 vi 防止覆盖文件的保护功能，强制刷新。

3. 转换文件

假如同前面提到的那样，启动 vi 编辑器时若用了多个文件名，如：

```
vi filename1 filename2
```

vi 先让用户编辑第一个文件 filename1，若所作编辑的内容已被存盘，用命令 n 就可以实现文件的转换。键入命令：

```
: n
```

第一次键入 n 命令时，vi 将从编辑 filename1 转换到编辑 filename2。若有两个以上的文件时可以再次用 n 命令来进行转换。

若键入命令：

```
: n!
```

vi 将在不检查缓冲区中的内容是否存盘的情况下完成文件间的转换工作。

一般来说，用户在使用带有“！”的命令时一定要小心。

4. 读文件

命令 `r` 可将文件从盘中读到缓冲区中，这个命令还可用来合并几个文件。例如，假设两个命令文件段 `newscript1` 和 `newscript2` 已分别通过测试，可以用以下的步骤组成命令文件 `newscript`：

```
键入 vi    newscript1
将光标移至最后一行的开始处
键入<ESC>键。
键入 : r newscript2
键入 : w newscript
```

在 `r` 前加入一个行号可读取该行以后的文本。例如，下面的命令：

```
: 20r abc
```

将文件 `abc` 的 20 行以后的内容读到缓冲区中。

5. 文件编辑

以下介绍功能键以及移动光标的命令，使用户能够在 `vi` 中快速地编辑文件。

(1) 功能键

与其他的程序编辑器或文字处理器一样，`vi` 有它自己的功能键以完成各种有用的操作。其中共有两套功能键，一套是转义命令模式的功能键，另一套是输入模式的功能键。

转义命令模式下的功能键包括<ESC>、（或<CTRL-C>）、</>、<?>和<: >。描述如下：

<ESC> 用来取消不完整的命令。

（在大部分终端编上与<CTRL-C>相同）产生一个中断，告诉编辑器停止它正在进行的工作。

</> 在末行命令方式中工作。用来定义一个被查找的字符串，并指示向前查找。

<?> 在末行命令方式中工作。用来定义一个被查找的字符串，并指示反向查找。

<: >进入末行命令方式。用来执行 `ex` 命令，如 `r`（读）和 `w`（写）等。`ex` 是与 `vi` 编辑器有关的程序编辑器。

在输入模式下的功能键包括<ESC>、<ENTER>、<BACKSPACE>、<CTRL-U>和<CTRL-W>。

具体描述如下：

<ESC> 用来改变操作方式，从输入方式进入命令方式。

<ENTER> 用来开始新的一行。

<BACKSPACE>在当前行上将光标后退并删除一个字符。

<CTRL-U> 将光标移至需插入的位置，并开始插入操作。有些系统用<CTRL-X>来代替。

<CTRL-W>或<CTRL-X> 将光标移至上次插入的文本中的第一个字符处。

(2) 移动光标命令

`vi` 中许多移动光标的命令可由不同的键完成，这主要取决于用户的终端类型。表 1 列

出并简要地描述了主要的移动光标的命令。

表 1 移动光标命令

按键	功能描述
L 或 SPACEBAR 或右箭头 (→)	右移光标
H 或 BACKSPACE 或左箭头 (←)	左移光标
J 或 CTRL-N 或 LF 或下箭头 (↓) 或-	移至下一行
K 或 CTRL-P 或上箭头 (↑) 或+	移至上一行
^	移到当前行的起始处
\$	移到当前行的末尾处
N G	移到第 n 行的始端
W或w	右移到下一个词
B或b	左移到当前单词开头
E或e	将光标移至当前单词的末尾
(将光标移到当前段的始端
)	将光标移到当前段的末端
[将光标移到当前节的始端
]	将光标移到当前节的末端
{	将光标移到当前段的前面
}	将光标移到当前段的后面
%	匹配符
H	将光标移到屏幕左上角处
M	将光标移到屏幕中央行的始端
L	将光标移到屏幕最后一行

(3) 文本插入命令

文本插入命令在编辑缓冲区中添加文本。插入点由专门的命令决定。用<ESC>键可退出输入模式。表 2 描述了主要的文本插入命令。

表 2 文本插入命令

用户输入命令	功能描述
i text <ESC>	在光标前插入新文本，ENTER 可重起一行
I text <ESC>	在当前行起始处插入新文本
a text <ESC>	在光标后输入新的文本
A text <ESC>	在当前行末尾输入新的文本
o text <ESC>	在当前行下产生新的一行并输入文本
O text <ESC>	在当前行上产生新的一行并输入文本

注：表中 text 表示用户的文本内容。<ESC>表示输入结束，返回到转义命令方式下。

(4) 文本删除命令

文本删除命令将删除文本中的字符、字、句、行，并在一个“删除文本缓冲区”堆栈中保留该删除的文本，以便恢复。“删除文本缓冲区”堆栈编号为 1~9，即 vi 可保留最后的 9 次删除。删除命令有转义命令方式的也有末行命令方式的。表 3 描述了主要的文本删除命令。

表 3 文本删除命令

用户输入命令	功能描述
X	删除光标处的一个字符
n x	删除光标处的 n 个字符
X	删除光标前面的一个字符
n X	删除光标前面的 n 个字符
Dd	删除整行
D	删除当前行中光标后的全部内容
: n ₁ , n ₂ d	删除第n ₁ 行到第n ₂ 行的内容

(5) 文本修改命令

文本修改命令更改指定的文本，实质上它是删除原有文本，再加上新的文本的过程。

表 4 描述了主要的文本修改命令。

表 4 文本修改命令

用户输入	功能描述
u或 : u	取消最近一次命令的动作
U	取消对当前行所做的修改
. (转义命令点号)	重复最近一次的插入或删除命令
cc text<ESC>	将当前行用指定文本代替
C text <ESC>	将当前行光标后的部分用指定文本代替
r char	替换光标后的字符
R text <ESC>	用 text 一一替换光标后的字符
s text <ESC>	用 text 替换光标后的一个字符

注：表中 text 表示用户的文本内容，char 表示字符。<ESC>表示输入结束。

(6) 查找命令

查找命令可在编辑缓冲区中向前或向后查找指定的内容。它们在缓冲区的起始和结尾之间运行。查找命令可根据要求重复进行或改变查找方向。表 5 描述了主要的查找命令。

表 5 查找命令

用户输入命令	功能描述
/pattern/	向前查找指定的内容 (pattern)
? pattern?	反向查找指定的内容 (pattern)
N	重复查找上次指定的内容
N	反向重复查找上次指定的内容
f char	在当前行查找指定字符
F char	在当前行反向查找指定字符
t char	将光标移到指定字符的前面
T char	将光标反向移到指定字符的前面
;	重复上次字符的查找
,	设置查找方向为上次查找字符的反方向
m letter	标记编辑缓冲区中的位置
' letter	将光标移至标记的行的始端

注：表中 pattern 表示用户查找的文本内容。char 表示字符。letter 表示一个字符的标记。

(7) 复制和移动

vi 中执行“复制和粘贴”的操作、“剪切和粘贴”的操作采用的是复制、剪切(删除)和放置的组合。表 6 列出了相应命令。

表 6 复制和移动

用户输入命令	功能描述
[n]yy	复制光标所在的一行或 n 行
[n]dd	剪切(删除)光标所在的一行或 n 行
P	将复制或剪切的内容放置在光标所在行下
P	将复制或剪切的内容放置在光标所在行上
:n1,n2 co n3	将 n1 到 n2 行的内容复制到 n3 之后
:n1,n2 m n3	将 n1 到 n2 行的内容移到 n3 之后

vi 的命令还有很多，这里不再一一列举，用户可以查看 Linux 的用户参考资料，以求得全面的了解。

第四部分 Shell 程序设计语言

Shell 程序是利用文本编辑程序建立的一系列 Linux 命令和实用程序序列的文本文件。它可以像 Linux 的任何命令一样执行。在执行 Shell 程序时，Linux 一个接一个地解释并执行每个命令。

Shell 是一种很成熟的程序设计语言，跟其他任何语言一样，有自己的语法、变量和控制语句。以下介绍的是 Linux 的 bash 的基本语法。

一. 创建和执行 Shell 程序方法

用户可以用前面介绍的 Vi 或其他文本编辑器编写 Shell 程序(如 shdemo.h)，并将文件以文本格式保存在相应的目录中。

用 chmod 将文件的权限设置为可执行模式，如文件名为 shdemo.h, 则命令如下：

```
chmod 755 shdemo.h
```

(文件主可读、写、执行，同组人和其他人可读和执行)

在 Shell 提示符后直接键入程序文件名执行：

```
$ shdemo.h
或 $ sh shdemo.h
或 $ . shdemo.h
```

(没有设置权限时可用点号引导)

二. 变量

Linux 支持三种类型的变量：环境变量、内部变量、用户变量。在 Shell 程序中变量是非类型性质的，也就是说不必事先指定变量的类型。

环境变量是系统环境的一部分参数，用户不必定义它们，但可以在 Shell 程序中使用它们，其中某些变量还能在 Shell 程序中加以修改。通过命令 env 可以查看环境变量及其值。

内部变量是由系统提供的。它与环境变量不同，用户不能修改它。

用户变量是在编写 Shell 程序时定义的，用户可以在 Shell 程序内使用和修改它。

变量赋值

```
lcount=0
```

(定义变量 lcount，并赋初值为整数 0)

```
myname=Samuel
```

(定义变量 myname，并赋初值为字符串 Samuel)

```
yourname=" Mr. Samuel"
```

(定义变量 yourname，并赋初值为字符串" Mr. Samuel"。字符串中有空格必须用引号括住。)

变量访问

通过变量名前置\$来访问变量。若有变量 lvar，值为 100，将其赋值给 lcount。

```
lcount = $var
```

则 \$echo \$lcount 将显示 100

三. 位置参数

从命令行执行 Shell 程序时可以自带若干参数。Shell 程序中用系统给出的位置参数来说明它们。存放第一个参数的变量名为 1（数字 1），在程序中用 \$1 访问；存放第二个参数的变量名为 2（数字 2），在程序中用 \$2 访问；依此类推。

例 1：设有如下 Shell 程序 myprog1.1，可带一个参数，程序显示这个参数。

```
#Name display program
if [ $# == 0 ]           #若参数个数为 0
then
    echo "Name not provided"
else
    echo "Your name is " $1
fi
```

使程序具有执行权限，并在提示符后键入命令行：

```
$. /myprog1             #没有参数
```

或 \$sh myprog1

屏幕显示：

```
Name not provided
```

在提示符后键入命令行：

```
$. /myprog1 Theodore   #有一个参数
```

屏幕显示：

```
Your name is Theodore  #引用$1 参数的效果
```

四. 内部变量

内部变量是 Linux 提供的一种特殊变量。这类变量由系统设置不能修改，在程序中用来作判定。

常用内部变量是：

\$#	传送给 Shell 程序的位置参数的个数
\$?	最后命令的代码或再 Shell 程序内所调用的 Shell 程序
\$0	Shell 程序的名称
\$*	调用 Shell 程序时所传送的全部参数的字符串

五. 特殊字符

某些字符对 Linux Shell 来说具有特殊含义，若将他们用在变量名或字符串中将产生程序表现不正常。

常用特殊字符有：

\$	引用变量名的开始
	把标准输出的内容传送到下一个命令
#	标记注释开始
&	后台执行命令
?	匹配一个字符
*	匹配多个字符
>	输出重定向操作符
<	输入重定向操作符
`	命令置换（Tab 键上面的那个键）
>>	输出重定向（到文件）操作符
[]	列出字符范围（如：[a-z] 字符 a 到 z，或[a,s]字符 a 或 s。）
.filename	执行 filename 程序（要有执行权）
空格	两个字的间隔

注：为把这些字符作为正常字符使用，必须用转义符（\）来阻止后续字符的特殊含义，即说明该字符保持正常字符的含义。

双引号：字符串中含有嵌入的空格时，可用双引号括起来，让 Shell 作为整体来解释字符串。

例：给 lword 赋值

```
lword=" abc def"
```

单引号：利用单引号括起来的字符串，阻止 Shell 解析变量。

例：给 newword 赋值

```
newword=' Value of var is $var'
```

执行

```
echo $newword
```

时显示

```
Value of var is $var
```

系统并未引用 var

反斜杠（\）：作为转义符，说明后续的字符保持正常字符解释，即无任何特殊含义。

反引号（` Tab 键上面的那个键）：通知 Shell 把反引号括起来的字符串作为命令执行，并将结果放入变量。

例：执行 wc 命令(统计文件 test.txt 的行数)，结果送入变量 lresult

```
lresult=`wc -l|test.txt`
```

六. 操作符

1. 字符串比较操作符

以下操作符可以用来比较两个字符串表达式：

以下操作符可以用来比较两个字符串表达式:

=	比较两个字符串表达式是否相等
!=	比较两个字符串表达式是否不相等
-n	判定字符串长度是否大于零
-z	判定字符串长度是否等于零

例 2: 在一个程序 compare.h 中比较字符串 string1 和 string2。

```
string1=" The First one"
string2=" The second one"
if [ string1 = string2 ]           ( [ ]和=前后加空格)
then
    echo "string1 equal to string2"
else
    echo "string1 not equal to string2"
fi                                  (条件语句结束)
if [ string1 ]
then
    echo "string1 is not empty"
else
    echo "string1 is empty"
fi
if [ -n string2 ]
then
    echo "string2 has a length greater than zero"
else
    echo "string2 has a length equal to zero"
fi
```

设置好执行权后, 运行

```
$ sh compare.h
```

结果显示

```
string1 not equal to string2
string1 is not empty
string2 has a length greater than zero
```

2. 数字比较操作符

以下操作符可以用来比较两个数字表达式:

-eq	等于
-ge	大于等于
-le	小于等于
-ne	不等于
-gt	大于
-lt	小于

3. 文件操作符

以下操作符可以用来构成判断文件的表达式:

-d	判断文件是否为目录
-f	判断文件是否为普通文件
-r	判断文件是否设为可读文件
-w	判断文件是否设为可写文件
-x	判断文件是否设为可执行文件
-s	判断文件是否长度大于等于零

例 3: 设当前目录下有一个文件 filea, 控制权为 (\sim r - x r - x r - x 即 0751),

有一个子目录 cppdir, 控制权为 (d r w x r w x r w x) 程序 compare2.h 中有如下内容:

```
if [ -d cppdir ]
then
    echo "cppdir is a directory"
else
    echo "cppdir is not a directory"
fi
if [ -f filea ]
then
    echo "filea is a regular file"
else
    echo "filea is not a regular file"
fi
if [ -r filea ]
then
    echo "filea has read permissione"
else
    echo "filea dose not have read permissione"
fi
if [ -w filea ]
then
    echo "filea has write permissione"
else
    echo "filea dose not have write permissione"
fi
if [ -x cppdir ]
then
    echo "cppdir has execute permissione"
else
    echo "cppdir dose not have execute permissione"
fi
```

设置好执行权后, 运行

```
$ sh compare2.h
```

结果显示

```
cppdir is a directory
filea is a regular file
filea has read permissione
filea dose not have write permissione
cppdir has execute permissione
```

4. 逻辑操作符

以下操作符可以用来构成逻辑表达式

!	求反(非)
-a	逻辑与 (AND)
-o	逻辑或 (OR)

七. 重复语句

对重复语句内包含的一系列命令重复执行多次。

1. For 语句

For 语句有几种格式:

第一种格式

```
for curvar in list
do
    statements
done
```

对 list 中的每个值需要执行一次 statements 时用这种格式。每次重复时, 把 list 中的每个值赋值给变量 curvar。List 中的值可由多个, 用空格加以分隔。

第二种格式

```
for curvar
do
    statements
done
```

或

```
for curvar in "$@"
do
    statements
done
```

对传给 Shell 的每个位置参数执行一次 statements。每次重复时把位置参数的值赋值给变量 curvar。

例 4: 下面程序将当前目录中的文件(用命令 `ls` 列出), 复制到 `backup` 子目录中建立备份。

```
for filename in `ls`  
do  
    cp $filename backup/$filename  
    if [ $? -ne 0 ]  
        echo "copy $filename failed"  
    fi  
done
```

2. while 语句

`while` 语句可用在当条件为真时, 执行一系列命令。一旦指定的条件判定为假时, 循环立即终止。语句格式:

```
while expression  
do  
    statement  
done
```

例 6: 进行 10 个偶数的相加

```
loopcount = 0  
result = 0  
while [ $ loopcount -lt 10 ]  
do  
    loopcount = `expr $ loopcount +1`  
    result = `$ result + ( $ loopcount * 2)`  
done  
echo "result is $ result"
```

3. until 语句

`until` 语句可用于执行一系列命令, 直到所指定的条件判定为真时才终止。语句格式:

```
until expression  
do  
    statement  
done
```

例 6: 进行 10 个偶数的相加

```
loopcount = 0  
result = 0  
until [ $ loopcount -ge 10 ]  
do  
    loopcount = `expr $ loopcount +1`  
    result = `$ result + ( $ loopcount * 2)`  
done  
echo "result is $ result"
```

4. repeat 语句

`repeat` 语句用来执行一个只要重复固定次数的语句。

例: 显示连字符 60 次。

```
Repeat 60 echo '- '
```


5. select 语句

select 语句可以用来生成一个菜单列表。Select 语句格式如下：

```
select item in itemlist
do
    statements
done
```

例 7：执行以下语句时系统生成一个显示数字的菜单，其中 1 表示 continue，2 表示 finish。用户选择 1，则 item 中含有 continue；选择 2，则 item 中含有 finish。

```
select item in continue finish
do
    if [ $ item = "finish" ]
        break
    fi
done
```

6. shift 语句

shift 语句用来从左到右处理位置参数的，位置参数用\$1、\$2、\$3……来标识。
语句格式：

```
shift
或
shift number
```

前一种格式每执行一次 shift，位置参数向左移动一个位置，即原来\$1 丢失，由原来的\$2 替换\$1，由原来的\$3 替换\$2，以此类推。

后一种格式向左移动 number 个位置。

八. 条件语句

1. if 语句

if 语句通过判断逻辑表达式来确定执行哪些语句。语句格式如下：

```
if [ expression ] then
    statements1
else
    statements2
```

```

fi                                #if 语句结束
也可以在 if 语句中嵌套另一个 if 语句。格式如下：
    if [ expression ] then
        statements1
    elif [ expression ] then
        statements2
    else
        statements3
    fi

```

2. case 语句

case 语句用来执行依赖于离散值或匹配指定变量的值语句。语句格式如下：

```

case str in
    str1|str2)
        statements1;;
    str3|str4)
        statements2;;
    *)
        statements3;;
esac                                #case 语句结束

```

每个条件可指定若干离散值（用 ‘|’ 分隔），只要能够匹配，就执行后续的语句。当其它条件均不能匹配，则最后条件是*，表示可以匹配任何值。每个条件下必须用双分号 ‘;;’ 来结束语句，否则会继续执行下个条件下的语句。

例 8：假如给出月份数字作为参数编写程序 displaymonth.h

```

case $1 in
    01|1) echo "Month is January" ;;
    02|2) echo "Month is February" ;;
    03|3) echo "Month is March" ;;
    04|4) echo "Month is April" ;;
    05|5) echo "Month is May" ;;
    06|6) echo "Month is June" ;;
    07|7) echo "Month is July" ;;
    08|8) echo "Month is August" ;;
    09|9) echo "Month is September" ;;
    010|10) echo "Month is October" ;;
    011|11) echo "Month is November" ;;
    012|12) echo "Month is December" ;;
    *) echo "Invalid parameter" ;;
esac

```

九. 杂项语句

1. break 语句

break 语句用来终止重复执行的循环。这些循环语句可以是 for、until 或 repeat 命令。

2. exit 语句

exit 语句用来退出 Shell 程序。在 exit 之后可以有选择地利用一个数字。如果当前的 Shell 程序被另一个 Shell 程序调用，那么调用程序可用这个返回参数检查并作相应判定。

十. 函数

如同其它编程语言一样，Shell 程序也支持函数。函数是执行特殊过程的部件，可以在 Shell 程序中被多次调用。

定义函数的格式：

```
func ( ) {  
    statements  
}
```

调用函数的格式：

```
func param1 param2 param3
```

func 是函数名，param1、param2、param3 是可选的参数。也可以把参数作为字符串来传送，例如 \$@。函数可以分析参数，就好象它们是传送给 Shell 程序的位置参数。

例 9：上面显示月份的程序改成函数如下

```
displaymonth ( ) {                                #定义函数  
    case $1 in  
        01|1) echo "Month is January" ;;  
        02|2) echo "Month is February" ;;  
        03|3) echo "Month is March" ;;  
        04|4) echo "Month is April" ;;  
        05|5) echo "Month is May" ;;  
        06|6) echo "Month is June" ;;  
        07|7) echo "Month is July" ;;  
        08|8) echo "Month is August" ;;  
        09|9) echo "Month is September" ;;  
        010|10) echo "Month is October" ;;  
        011|11) echo "Month is November" ;;  
        012|12) echo "Month is December" ;;  
        *) echo "Invalid parameter" ;;  
    esac  
}  
displaymonth 8                                    #调用函数  
displaymonth 12  
程序将显示  
    Month is August  
    Month is December
```

第五部分 Linux 编程系统调用

Linux 系统的另一种用户接口是程序员用的编程接口，即系统调用。系统调用的目的是使用户可以使用操作系统提供的有关进程控制、文件系统、输入输出系统、设备管理、通信及存储管理等方面的功能，而不必涉及系统内部结构和硬件细节，大大减少用户程序设计和编程难度。

Linux 的系统调用以标准实用子程序形式提供给用户在编程中使用。一般使用系统调用应注意以下三点：

- 函数所在的头文件
- 函数调用格式，包括参数格式、类型。
- 返回值格式、类型。

一. 有关进程管理的系统调用

1. 创建子进程

(1) 创建子进程 `int fork()`

函数调用包含文件：

```
#include<unistd.h>
```

调用格式：

```
n=fork()
```

该系统调用返回值可以是 $n=0$ 、 $n>0$ 、 $n=-1$ 。

如果执行成功，`fork()` 返回给父进程的是子进程的标识符 (pid)；返还给子进程的是 0；创建失败返回 -1。

$n=0$ 表示获得此参数的进程将执行子进程的程序段（由系统调度确定）。

$n>0$ 表示获得此参数的进程将执行父进程的程序段（由系统调度确定），该参数即为子进程的标识符 (pid)。

$n<0$ 表示创建失败。

(2) 创建线程 `clone()`

函数调用包含文件：

```
#include<sched.h>
```

函数调用原形：

```
int clone(int (*fn)(void *arg), void *child_stack, int flags, void *arg)
```

`clone()` 允许子进程和父进程共享一些执行的上下文环境。主要用于多线程编程。

其中 : `int (*fn)(void *arg)` 子进程执行的函数

`oid *child_stack` 子进程使用的堆栈

int flags	父进程和子进程共享的资源
CLONE_VM	使用相同内存
CLONE_FS	共享相同文件系统
CLONE_FILES	使用相同的文件描述表
CLONE_SIGHAND	共享信号处理函数表
CLONE_PID	子进程的 PID 与父进程的 PID 相同

2. 进程的有关标识 ID

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

- 获得进程标识 `int getpid();`
- 获得父进程标识 `int getppid();`
- 获得进程真实用户标识 `int getuid();`
- 获得进程真实组标识 `int getgid();`
- 获得进程有效用户标识 `int geteuid();`
- 获得进程有效组标识 `int getegid();`

这些系统调用返回值是整型的标识数。

3. 进程优先数

- 获取进程优先数 `getpriority()`
- 改变进程优先数 `nice()`

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

```
int nice(int inc);
```

其中: `inc` 为优先数增量。一般用户只能设为正数(降低优先级)。超级用户才可提高进程优先级。

返回值: 成功返回 0; 否则返回-1。

4. 进程等待

- 等待子进程结束 `int wait();`
- 等待指定的子进程结束 `int waitpid();`

函数调用包含文件:

```
#include<sys/type.h>
```

```
#include<sys/wait.h>
```

函数调用原形:

```
pid_t wait(int *statloc);
```

```
pid_t waitpid(pid_t pid, int *statloc, int options)
```

`wait()` 和 `waitpid()` 的区别在于后者较灵活性: `wait()` 挂起调用进程直至有

一个进程结束；而 `waitpid()` 可以不挂起立即返回，或是指定等待直到某个特定进程终止。

其中：`statloc` 是整型指针。如果它不为空，子进程的终止状态字就存放在该参数指示的内存位置。若要忽略终止状态字，可用一个空指针。

`pid` <-1 等待进程组 `id` 等于 `pid` 的绝对值的子进程。
-1 等待任何子进程（这种情况相当于 `wait()` 函数）。
0 等待进程组 `id` 与父进程组 `id` 相同的子进程。
>0 等待进程 `id` 等于 `pid` 的进程。

`Options` 用来控制 `waitpid` 运行的参数。可以取 0 或一些常数。如：`WNOHANG` 表示若无子进程终止状态字，`waitpid` 不挂起，则返回 0。
`WUNTRACED` 与作业控制有关。

- 睡眠 `sleep(int n);`

其中：`n` 为睡眠的时间（秒）。时间一到恢复运行。

5. 用 `exec` 函数运行程序

创建的新进程可以通过 `exec()` 等函数运行一个新程序。`Exec` 函数有 6 种不同的调用形式，都可以启动新程序的运行，只是参数不同。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
int execl(const char *pathname, const char *arg0, ...);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *pathname, const char *arg0, ..., char *const envp[]);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, char *const arg0, ...);
int execvp(const char *filename, char *const argv[]);
```

这些函数中 `execlp` 和 `execvp` 可执行程序是当作 Shell 脚本程序来由 `/bin/sh` 解释执行。

一个进程调用 `exec` 函数执行另一个程序后，这个进程就完全被新程序代替。由于并没有产生新进程所以进程标识号不改变，除此之外旧进程的其它信息，代码段、数据段、栈段等均被新程序的信息所代替。新程序从自己的 `main()` 函数开始运行。

6. 结束进程

在 Linux 中，有三种正常结束进程的方法，两种异常终止的方法。

正常结束：

- 在 `main` 函数中调用 `return`。相当于调用 `exit`。
- 调用 `exit` 函数。
- 调用 `_exit` 函数。

异常终止：

- 调用 `abort`。产生一个 `SIGABRT` 信号。

- 进程收到特定信号。这个信号可以是进程自己产生，也可以来自其它进程和内核。

函数调用包含文件：

```
#include<stdlib.h>
```

函数调用原形：

```
void exit(status);
```

或

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
void _exit(int status);
```

参数：

status 为退出的状态。

返回值：

0 正常

非0 错误

7. 程序中执行命令行

Linux 中允许在程序中执行 Shell 命令行。只要使用 system() 函数就可以了。

函数调用包含文件：

```
#include<stdlib.h>
```

函数调用原形：

```
int system(const char *cmdstring);
```

这个函数是用 fork, exec, waitpid 三个系统调用实现的。若 cmdstring 非空，函数返回值由这三个函数结果确定。

调用格式：

```
system(“data>file”);          #将日期送到文件 file 中
```

或

```
status=system(“who”);          #显示目前登录到系统中的人
```

8. 暂停当前进程的运行

pause() 函数的功能暂停当前进程运行，直到接收了某一信号为止。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
int pause(void);
```

返回值：失败为-1。

二. 有关信号处理的系统调用

信号

在 Linux 系统中, 针对不同的软硬件情况, 核心程序会发送不同的信号来通知进程某个事件的发生。信号也可以由某些进程发出。但是如何处理这个信号, 就要由进程本身来解决。

进程收到信号后, 处置方法有以下几种:

- (1) 采用系统默认的行动 (一般系统暂停进程的执行)
- (2) 执行一个处理函数, 此函数可由用户设定。
- (3) 忽略此信号。
- (4) 暂时搁置该信号

Linux 支持的信号包括 POSIX.1 中的信号以及其它信号, 共三十余种。

1. 设置处理信号的函数 `signal()`

用户可设置一个处理某信号的函数, 以便进程接收到此信号后执行。Linux 中用户可自定义的信号为 16、17 号。

函数调用包含文件:

```
#include<signal.h>
```

函数调用原形:

```
void signal(int signum,void *func(int));
```

其中: `signum` 为信号值。Func 为要执行的函数指针名。

2. 发送信号给某进程 `kill()`

`kill()` 系统调用可以把某个信号发送给一个或一组进程。接到该信号的进程依照事先设定的程序来处置。

函数调用包含文件:

```
#include<sys/types.h>
```

```
#include<signal.h>
```

函数调用原形:

```
int kill(pid_t pid,int sig);
```

其中: `pid` 的取值有 4 种

- (1) `pid` 大于 0 时, 是要送往的进程标识符。
- (2) `pid` 等于 0 时, 信号送往所有与调用进程同一用户组的进程。
- (3) `pid` 等于-1 时, 若调用进程是超级用户的, 则信号送往发送进程本身以及所有正在执行的中的程序。
- (4) `pid` 等于-1 时, 若调用进程不是超级用户的, 则信号送往所有有效用户标识符与发送进程的真实用户标识符相同的进程, 但发送者自己除外, 即用来发送信号给某一用户下的所有进程。

`Sig` 是准备发送的信号的代码。

3. 特定时间送出 `SIGALRM` 信号

`alarm()` 系统调用使核心程序在特定时间送一个 `SIGALRM` 信号给调用它的进程。此

函数用来限制系统调用的执行时间。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
unsigned int alarm(unsigned int seconds);
```

其中：seconds 是指定系统核心在多少秒后发送 SIGALRM 信号。

4. 暂停执行 pause()

pause () 系统调用的功能暂停当前进程运行，直到接收了某一信号为止。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
int pause(void);
```

返回值：失败为-1。

三. 管道操作的系统调用

管道通信是进程的一种通信方式。Linux 中两个进程可以通过管道来传递消息。管道在逻辑上被看作管道文件，在物理上则由文件系统的高速缓冲区构成。管道用 pipe() 系统调用建立。发送信息的进程用 write() 把信息写入管道，接收进程用 read() 从管道中读取信息。

1. 建立管道 pipe()

pipe() 系统调用可以建立一条非命名同步通信管道。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
int pipe(int fildes[2]);
```

其中：fildes[2] 是两个文件描述字放在一个数组中，指向一个管道文件的 i 节点(文件描述字可由用户自定)。fildes[0] 是为读建立的文件描述字，fildes[1] 是为写建立的文件描述字。

返回值：执行成功返回值是 0；执行失败返回值是-1 。

2. 发送信息 write()

发送信息的进程用 write() 把信息写入管道。

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
int write(fildes[1],buf, size);
```

函数将缓冲区 buf 中，长度为 size 的消息写入管道的写入端 fildes[1]。

返回值：执行成功返回值是 0；执行失败返回值是-1。

3. 接收信息 read()

接收信息的进程用 read() 把信息从管道中读出。

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
int read(fildes[0],buf,size);
```

函数将长度为 size 的信息从管道的输出端 fildes[0] 读出，送到缓冲区 buf 中。

返回值：执行成功返回值是 0；执行失败返回值是-1。

4. 进程间互斥lockf()

lockf() 的功能是将指定文件的指定区域进行加锁和解锁，以解决临界资源的共享问题。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
int lockf(files,founction,size);
```

其中：files 文件描述符。

function 是锁定和解锁模式，1 为锁定，0 为解锁。

Size 是锁定解锁的字节数，0 表示从当前位置到文件尾。

注：有关管道的系统调用实例，可参见附录 A 中有关“管道通信的实验”的源程序。

四. IPC 系统调用

Linux 系统也支持 UNIX System V 的进程通信机制 IPC，它负责完成进程间大量的数据传送。这三种机制分别是：消息队列、信号量及共享内存。

System V 的进程通信机制中每个对象都和一个引用标识符相联系。如果进程要访问 IPC 对象，则需要在系统中申请这个唯一的引用标识符。这里 IPC 对象就是指某个消息队列、某个信号量或者某个共享内存段。也就是说，假若要访问某个共享内存段，就需要给这个内存段指定标识符，并通过该标识符完成相关操作。

进程通信机制的每个对象都有一个 icp_perm 结构与之对应。这个结构记录了对象的一些信息，如所有者、创建者、权限等。它定义在<sys/ipc.h>头文件中。

```
Struct ipc_perm{
    Uid_t      uid;          /*所有者的有效用户 ID*/
    Gid_t      gid;          /*所有者的有效组 ID*/
    Uid_t      cuid;        /*创建者的有效用户 ID*/
    Gid_t      cgid;        /*创建者的有效组 ID*/
    Mode_t     mode;        /*访问权限*/
    Ulong      seq;          /*应用序号*/
    Key_t      key;         /*访问键*/
}
```

1. 消息队列通信机制

消息队列是一条由消息连接而成的链表，它保存在内核中，通过消息队列的引用标识符（也叫队列标识符）来访问。

每个消息队列都有一个 `msqid_ds` 结构与之对应，这个结构用来保存消息队列的当前状态的参数。

```
struct msqid_ds{
    struct ipc_perm msg_perm;    /*消息队列的对应的 ipc_perm 的指针*/
    struct msg      *msg_first; /*消息队列的第一个消息的指针*/
    struct msg      *msg_last; /*消息队列的最末一个消息的指针*/
    ulong           msg_ctypes; /*当前队列中的字节总数*/
    ulong           msg_qnum;   /*当前队列中的消息的个数*/
    ulong           msg_qbytes; /*队列中的可存放的最大字节数*/
    pid_t           msg_lspid; /*最近执行 msgsnd() 的进程号 */
    pid_t           msg_lrpid; /*最近执行 msgrcv() 的进程号 */
    time_t          msg_stime; /*最近执行 msgsnd() 的时间*/
    time_t          msg_rtime; /*最近执行 msgrcv() 的时间*/
    time_t          msg_ctime; /*最近一次消息改变的时间*/
};
```

(1) 创建和打开消息队列 `msgget()`

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

函数调用原形：

```
int msgget(key_t key, int flag);
```

其中：key 可以取值为 `IPC_PRIVATE` 以保证返回一个空表项，或被设置成一个还不存在的表项号。这时只要 `IPC_PRIVATE&flag` 为真，就可以返回新表项的描述字。

flag 由操作允许权和控制命令相“或”得到。如：`IPC_CREATE|0777` 等。

返回值：成功返回队列标识符。失败返回-1。

调用格式： `int msgqid;`

```
msgqid=msgget(MSGKEY, 0777)    /*建立消息队列，MSGKEY 是事先定
                                义的常量*/
```

(2) 向消息队列发送消息 `msgsnd()`

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

函数调用原形:

```
int msgsnd(int msgqid, const void *ptr, size_t nbytes, int flag);
```

其中: msgqid 消息队列标识符 (即 msgget() 的返回值)。

*ptr 指向用户存储区的指针。用来存储要发送的消息。

Nbytes 发送消息长度, 以字节记。

Flag 可以取 0 或 IPC_NOWAIT。一旦设置 IPC_NOWAIT, 在队列满的情况下 msgsnd() 返回出错。

返回值: 正确 0; 错误-1。

(3) 从消息队列接收消息 msgrcv()

函数调用包含文件:

```
#include<sys/types.h>
```

```
#include<sys/ipc.h>
```

```
#include<sys/msg.h>
```

函数调用原形:

```
int msgrcv(int msgqid, void *ptr, size_t nbytes, long type, int flag);
```

其中: msgqid 消息队列标识符 (即 msgget() 的返回值)。

*ptr 指向用户存储区的指针。用来存储接收来的消息。

nbytes 接收消息数据的长度, 以字节记。消息中数据长度超过这个值, 就根据 flag 来处理, 要么截断, 要么出错返回。

type 指定要接收消息队列的哪条信息。分三种情况:

Type=0 返回第一条消息。

Type>0 返回消息队列中类型域等于这个值的第一个消息。

Type<0 返回消息队列中类型域小于等于这个值的绝对值的消息中, 类型域最小的第一个消息。

flag 有两位与接收消息有关。倘若该队列中无消息, 如果 IPC_NOWAIT 置位, 在指定 type 无效时 msgrcv 将不等待, 直接带错返回。若设置了 MSG_NOERROR, 若消息数据长度大于 nbytes 时, 消息数据被截断。

返回值: 正确返回 接收的消息数据的长度; 错误-1。

```
调用格式:  int count;                                /*接收字节数*/
            count=msgrcv(qid, qbuf, length, type, 0)  /*从消息队列 qid 接收
                                                         length 个数据, 放在 qbuf 中
                                                         */
```

(4) 消息队列的控制操作 msgctl()

函数调用包含文件:

```
#include<sys/types.h>
```

```
#include<sys/ipc.h>
```

```
#include<sys/msg.h>
```

函数调用原形:

```
int msgctl(int msgqid, int cmd, struct msqid_ds *buf);
```

其中: msgqid 消息队列标识符。

cmd 是本函数执行的控制操作。可以取下面三种值:

IPC_STAT 取消息队列对应的 msqid_ds 放在 buf 指定的空间。

IPC_SET 把消息队列对应的 msqid_ds 中的一些域设置成 buf 中的值。

(只有有效用户和超级用户才能执行这条操作。)

IPC_RMID 将消息队列从系统中删除。(只有有效用户和超级用户才能执行这条操作。)

注: 系统调用实例, 可参见附录 A 中有关“消息通信实验”的源程序。

2. 共享内存通信机制

共享内存是允许多个进程共享一块内存, 由此达到交换信息的通信机制。通常共享内存段由一个进程创建, 接下来的读写操作就可以由多个进程参加, 进行信息传递。

每个共享内存段对应一个 shmid_ds 结构。定义如下:

```
struct shmid_ds{
    struct ipc_perm shm_perm;    /*指向对应 shmid_ds 结构*/
    int shm_segsz; /*共享内存段大小字节数*/
    ushort shm_lkcnt; /*共享内存段被锁定的时间*/
    pid_t shm_cpid; /*创建该共享内存段的进程号*/
    pid_t shm_lpid; /*最近一次调用 shmop() 的进程号*/
    ulong shm_nattch; /*当前把该共享内存段附加到地址空间的进程数*/
    time_t shm_atime; /*最近一次附加操作的时间*/
    time_t shm_dtime; /*最近一次分离操作的时间*/
    time_t shm_ctime; /*最近一次改变的时间*/
};
```

(1) 创建和打开共享内存段函数 shmget()

函数调用包含文件:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

函数调用原形:

```
int shmget(key_t key, int size, int flag);
```

其中: key 是共享内存段标识符。

size 是共享内存段的最小尺寸。创建时必须指定, 打开时可以忽略。

Flag 由操作允许权和控制命令相“或”得到。

返回值：正确返回共享内存段的标识符 `shmid`；错误返回-1。

(2) 将共享内存段附加到进程地址空间函数 `shmat()`

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

函数调用原形：

```
int *shmat(int shmid, void *addr, int flag);
```

其中：`shmid` 是共享内存段的标识符。

`addr` 是附加的内存地址。它和 `flag` 共同设置。

`addr` 为 0，系统查找进程空间，将共享内存段附加在第一块有效内存上。

`addr` 非 0，若 `flag` 中 `SHM_RND` 没置位，共享内存段附加在 `addr` 地址上。

若 `flag` 中 `SHM_RND` 已置位，共享内存段附加在

`addr - (addr mod SHMLAB)` 指定的地址上。

返回值：正确返回共享内存段的指针；错误返回-1。

(3) 将共享内存段从进程地址空间断开函数 `shmdt()`

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

函数调用原形：

```
int shmdt(void *addr);
```

其中：`addr` 是附加函数 `shmat()` 的返回值。

返回值：正确返回 0；错误返回-1。

(4) 共享内存机制控制操作函数 `shmctl()`

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

函数调用原形：

```
int shmctl(int shmid, int cmd, shm_id_ds *buf);
```

其中：`shmid` 共享内存段标识符。

`cmd` 是本函数执行的控制操作。可以取下面任一种值：

- `IPC_STAT` 取共享内存段对应的 `shm_id_ds` 放在 `buf` 指定的空间。
- `IPC_SET` 把共享内存段对应的 `shm_id_ds` 中的一些域设置成 `buf` 中的值。（只有有效用户和超级用户才能执行这条操作。）
- `IPC_RMID` 将共享内存段从系统中删除。内存段标识符立即删除。但

删除操作并不立即执行，这是因为还有其它进程将内存段附加在自己的地址空间中。一旦这些进程结束或断开连接，这块内存将会从系统中删除。（只有有效用户和超级用户才能执行这条操作。）

- SHM_LOCK 锁住共享内存段。（只有超级用户才能执行这条操作。）
- SHM_UNLOCK 释放共享内存段。（只有超级用户才能执行这条操作。）

3. 信号量机制

信号量是一个整形计数器，用来控制多个进程对共享资源的访问。共享资源分为两类，一类是互斥共享，即任一时刻只允许一个进程访问资源；另一类是同步共享，可有若干进程相互协作来操作资源。

Linux 系统中对信号量操作按以下步骤进行：

1. 进程将信号量的值减 1，并检测这个控制资源的信号量的值。
2. 如果信号量的值是零或正数，就可以使用这个资源。表示它正在使用这个资源的某单元。
3. 如果信号量的值小于零，那么进程进入睡眠状态，直到信号量的值重新大于零时被唤醒。
4. 当一个进程对某个共享资源操作结束时，就将控制这个资源的信号量加 1。如果有某些进程正在等待该资源而处于睡眠状态，就将其唤醒。

为了正确地实现信号量机制，检测和增减信号量的值应该是原语操作。所以信号量一般是在内核中实现的。

每个信号量都与一个结构相联系。该结构定义在头文件<sys/sem.h>中。具体描述如下：

```
struct{
    struct ipc_perm sem_perm;    /*指向与信号量对应的 ipc_perm*/
    struct sem sem_base;        /*指向第一个信号量的位置的指针*/
    ushort      sem_nsems;      /*集合中信号量的个数*/
    time_t      sem_otime;      /*最近一次调用 semop() 的时间*/
    time_t      sem_ctime;      /*最近一次改变的时间*/
};
```

其中：sem 结构记录的是单一信号的信息，内容如下

```
struct sem{
    ushort semval;              /*信号量的值*/
    pid_t  sempid;              /*最近一次执行操作的进程的进程号*/
    ushort semncnt;             /*等待信号量增长，使用资源的进程数*/
    ushort semzcnt;             /*等待信号量减少到零的进程数*/
};
```

(1) 创建和打开信号量集的函数 semget()

函数调用包含文件:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
```

函数调用原形:

```
int semget(key_t key, int nsems, int flag);
```

其中: 参数 key, flag 的含义和用途与消息队列的函数 msgget() 中的相应参数相同。

Nsems 指信号集中应该创建的信号个数。如果是打开信号集, 该参数忽略。

返回值: 正确返回信号标识符; 错误返回-1。

(2) 信号集的原语操作函数 semop()

函数调用包含文件:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
```

函数调用原形:

```
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

其中: semid 为信号集的标识符。Semoparray 为 sembuf 结构的数组。Nops 为数组中元素的个数。

Sembuf 结构中记录了对信号集的操作, 具体如下:

```
Struct sembuf{
    Ushort sem_num; /*要处理的信号量在信号集中的序号*/
    Short   sem_op; /*要执行的操作, 可以取正、负或零*/
    Short   sem_flag; /*操作标记可用 IPC_NOWAIT 或 SEM_UNDO*/
};
```

(3) 信号集控制函数 semctl()

函数调用包含文件:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
```

函数调用原形:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

其中: semid 信号集的标识符。Semnum 指向信号集中特定的信号量, 控制操作就是在该信号量上实施的。arg 是个联合, 内容如下:

```
union semun{
    int      val; /*cmd 为 SEYVAL 时, 这是信号量要被设置的值*/
    struct semid_ds *buf; /*指向 semid_ds 结构的指针, cmd 应为 IPC_STAT 或 IPC_SET*/
    ushort   *array; /*cmd 为 GETALL 或 SETALL 时, 存储信号量的值*/
};
返回值: 所有 GET 命令 (除 GETALL 外) 返回对应的值; 其它情况返回零。
```


五. 文件系统的系统调用

1. Linux 的文件系统的特点

Linux 的最大特点之一是它支持许多不同的文件系统。这使它很容易地与其他操作系统共存于同一台机器上。Linux 已经能够支持三十几种不同的文件系统, 包括 ext、ext2、umsdos、msdos、proc、smb、ncp、iso9660、sysv、hpfs、affs、ufs、ntfs 等。

Linux 的文件系统的另一个特点是将 I/O 子系统结合到文件系统中。Linux 将对外围设备的 I/O 操作设计成与文件的 I/O 操作完全一样, 因此无论处理什么设备, 操作上都和文件的输入输出一样。所有设备都放在文件系统的/dev 目录下, 这样使设备的输入输出有完全相同的接口, 从而使操作和应用程序开发都很方便。

Linux 内核的虚拟文件系统 (VFS) 使系统能支持多个不同的文件系统, 为此虚拟文件系统维护了一些描述整个虚拟文件系统信息和真实文件系统信息的数据结构。每一个 mount (装载) 的文件系统都由一个 VFS 超级块来说明。VFS 超级块包含如下信息:

- | | |
|--------------|-------------------------------------------------------------------------|
| 1) 设备。 | 文件系统所在的块设备号。 |
| 2) I 节点指针。 | 包括 mounted I 节点指针 (指向装载的文件系统中第一个 I 节点) 和 covered I 节点指针 (所挂接的目录的 I 节点)。 |
| 3) 数据块大小。 | 以字节为本文件系统的块的大小。 |
| 4) 超级块操作例程。 | 指向本文件系统所支持的一些超级块例程的指针。 |
| 5) 文件系统类型。 | 指向已经 mount 的文件系统的 file_system_type 数据结构的指针。 |
| 6) 特定文件系统的指针 | 指向本文件系统所需信息的指针。 |

2. 文件的分类

Linux 的文件除了正规文件、目录文件以外, 还有其他几种文件类型。
包括:

- 普通文件
- 目录文件
- 硬链接文件
- 符号链接文件
- 套接字文件
- 有名管道文件
- 字符设备文件
- 块设备文件

其中：

普通文件 是分为文本文件和二进制文件，用户可以自己处理数据的逻辑边界。

用 `ls -l` 列出其属性，最高位用字符“-”表示。

目录文件 是由目录列表组成的有结构的记录式文件。用 `ls -l` 列出其属性，最高位用字符“d”表示。

硬链接文件 用 `ln` 命令可以产生硬链接文件。用 `ls -l` 列出其属性，最高位用字符“1”表示。硬连接文件共享 i 节点，不能跨文件系统存在。

符号链接文件 与硬链接文件不同，符号文件指向一个文件，操作系统将它看作一个特殊文件。可以用 `ln -s` 命令产生。

套接字文件 套接字供 Linux 系统与其他机器联网时使用，一般用在网络端口上。文件系统利用套接字文件进行进程间通信。用 `ls -l` 列出文件列表时，套接字文件权限前第一个字母为 s。

有名管道文件 有名管道文件也是通过文件系统进行通信的一种方法。命令 `mknod` 可用来创建一个有名管道。用 `ls -l` 列出文件列表时，有名管道文件权限前第一个字母为 p。

字符设备文件 设备文件一般存放在 `/dev` 目录中。字符设备文件通过文件系统提供了一种与设备驱动程序通信的方法。每次的通信量是一个字符。用 `ls -l` 列出文件列表时，字符设备文件权限前第一个字母为 c。每个字符设备还包括两个数值，代表进行通信的主设备和从设备。

块设备文件 块设备文件与字符设备文件一样存放在 `/dev` 目录中，用于同驱动程序通信。块设备也包含主从设备号。进行通信时一次传送一个数据块。用 `ls -l` 列出文件列表时，块设备文件权限前第一个字母为 b。

3. 文件描述字

当进程成功地打开一个文件时，操作系统通过系统调用返回一个文件描述字给调用它的进程。这个文件描述字是提供给其它函数或系统调用对文件读写的接口。系统内部用文件描述字来识别文件。每个进程至多同时使用 20 个文件描述字（0~19），其中 0、1、2 分别自动产生指定给 `stdin`、`stdout`、`stderr`，因此打开的第一个文件描述字是 3，接下来是 4，以此类推。

4. inode 信息

inode (index node) 是 Unix 和 Linux 描述文件信息的数据结构。每一个文件（包括普通文件、目录文件、设备文件、……）都有一个 inode 与之对应。对文件的访问通过 inode 来实现。同时系统还提供了以下一组读取 inode 信息的系统调用。

• **inode** 结构包含以下信息:

- 1) 设备信息: 文件使用的设备。
- 2) 状态 (mode) 信息: 包含文件的类型及访问权限。
- 3) 所有者信息: 文件所有者的用户 ID、文件所有者所在的组 ID。
- 4) 连接信息 (link): 指向本 inode 的连接文件的个数。
- 5) 文件的大小 (size): 文件的字节数。
- 6) 时间戳 (timestamps): 上次文件被访问的时间 (access time)、上次文件被修改的时间 (mod time)。
- 7) 数据块 (datablocks): 含指向文件占用的磁盘数据块的指针。前 12 个是直接指针, 后 3 个分别指向一级间接块、二级间接块、三级间接块。

• 可供进程访问的属性

为了访问 inode, 可供进程使用的属性被复制到 stat 结构中。stat 结构在 <sys/stat.h> 中定义。主要的域有:

dev_t	st_dev	/*设备号*/
ino_t	st_ino	/*索引节点号*/
umode_t	st_mode	/*文件类型和权限*/
nlink_t	st_link	/*链接计数*/
uid_t	st_uid	/*文件所有者的标识号 owner ID*/
gid_t	st_gid	/*文件所有者的组标识号 group ID*/
dev_t	st_rdev	/*设备文件的设备号*/
off_t	st_size	/*以字节为单位的文件的容量*/
time_t	st_atime	/*最后一次访问的时间*/
time_t	st_mtime	/*最后一次修改的时间*/
time_t	st_ctime	/*最后一次状态改变的时间*/
long	st_blksize	/*文件系统 I/O 首选的块的大小*/
long	st_blocks	/*实际分配的块的个数*/

• 有关 **inode** 的系统调用

系统还提供了以下一组读取 inode 信息的系统调用。stat 结构中的信息可以利用三个系统调用 fstat()、stat() 或 lstat() 之一来访问。

函数调用包含文件:

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

函数调用原形:

```
int fstat(int fd, struct stat *sbuf);
```

或

```
int stat(char *pathname, struct stat *sbuf);
```

或

```
int lstat(char *pathname , stat *sbuf);
```

其中：

fd	指向打开文件描述信息的文件描述符。
Pathname	文件的路径名。
Sbuf	是 stat 结构的指针。

正确返回参数：0；

错误返回参数：-1；

说明：

如果第一个参数是文件的路径名而不是文件描述符，则 stat() 调用 和 fstat() 调用是相同的。

如果文件是一个符号链接，lstat() 调用将给出链接文件自身的状态细节，stat() 调用将依照链接给出所指向的文件的信息。

注：系统调用实例，可参见附录 A 中有关“获得 inode 信息实验”的源程序。

每个文件系统都有最大设备号和最小设备号。设备号的数据类型是 dev_t 。系统提供了两个宏 major 和 minor 可以访问设备文件的最大设备号和最小设备号。

inode 结构中的设备参数除了 st_dev 还有 st_rdev。其中 st_dev 值对系统中每个文件名来说是指含有该文件名和对应的 inode 的文件系统的设备号；而 st_rdev 这个值只有字符设备文件和块设备文件才有，它表示了实际设备的设备号。

注：系统调用及宏操作实例，可参见附录 A 中有关“设备文件的设备号信息”的源程序。

5. 文件操作的系统调用

(1) 打开文件的系统调用 open()

打开文件和创建文件的方法是 open()。

函数调用包含文件：

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

函数调用原形：

```
int open(const char *pathname, int flag);
```

或

```
int open(const char *pathname, int flag,mode_t mode);
```

其中：

pathname	是待打开或创建的文件名。
Flag	规定如何打开文件。包括：
O_RDONLY	只读方式打开。
O_WRONLY	只写方式打开。
O_RDWR	读写方式打开。
O_APPEND	写文件时赋加在文件末尾。
O_CREAT	若文件不存在，则创建该文件。
O_EXCL	若使用 O_CREAT 时，文件已存在，则产生出错。
O_TRUNC	若文件已存在，写入之前先删除原有数据。
Mode	用于 flag 为 O_CREAT 时指定新文件的所有者、文件的用户组以及系

统中其他用户的访问权限位。包括：

S_IRUSR

S_IWUSR

S_IXUSR	文件所有者的读权限位
S_IRGRP	文件所有者的写权限位
S_IWGRP	文件所有者的执行权限位
S_IXGRP	文件用户组的读权限位
S_IROTH	文件用户组的写权限位
S_IWOTH	文件用户组的执行权限位
S_IXOTH	文件其他用户的读权限位
	文件其他用户的写权限位
	文件其他用户的执行权限位

访问权限位按位逻辑加组合：

S_IRWXU	定义为(S_IRUSR S_IWUSR S_IXUSR)
S_IRWXG	定义为(S_IRGRP S_IWGRP S_IXGRP)
S_IRWXO	定义为(S_IROTH S_IWOTH S_IXOTH)

可以用以下常量值设置 set_uid 和 set_gid：

S_ISUID	设置 set_uid
S_ISGID	设置 set_gid

正确返回参数：文件描述符；

错误返回参数：-1；

例：

```
open("afile", O_RDONLY);          /* 以只读方式打开文件 afile */
open("bfile", O_RDWR | O_TRUNC);  /* 以读写方式打开文件 bfile，长度置为
                                   0 */
open("cfile", O_WRONLY | O_CREAT | O_EXCL, S_IRWXU | S_IXGRP);
/* 创建文件 cfile 用于写操作，其权限设置为 rwx--x--x */
```

(2) 检测文件是否访问过 access()

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
int access(char *pathname, int mode);
```

其中：

pathname	是待检测的文件名。
Mode	是包含在 unistd.h 文件中的值之一：
R_OK	检测调用进程是否有过读操作
W_OK	检测调用进程是否有过写操作
X_OK	检测调用进程是否有过执行操作
F_OK	检测指定的文件是否存在

正确返回参数：1；

错误返回参数：0；

(3) 创建新文件的系统调用

```
creat()
```

新文件可以用 creat() 创建。

函数调用包含文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

函数调用原形：

```
int creat(const char *pathname, mode_t mode);
```

该函数与下面函数等价：

```
open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

返回：如果创建正确则返回文件描述符，文件的打开方式为“只写”；否则返回 -1。

说明：创建文件只用“只写”方式打开。如果创建的文件既要写，又要读，则必须连续使用 creat、close、open。也可以用以下 open 函数：

```
open(pathname, O_RDWR|O_CREAT|O_TRUNC, mode);
```

(4) 读文件的系统调用 read()

一旦用 O_RDONLY 或 O_RDWR 方式打开或建立了文件，就可以用 read() 系统调用从该文件读取字节了。

函数调用包含文件：

```
#include <unistd.h>
```

函数调用原形：

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

正确返回参数：0 或字节数；

错误返回参数：-1；

其中：

fd 是想要读的文件的文件描述符。

Buf 是指向内存的指针。read() 将从文件中读取的字节存放到这个内存块

Nbyte 从该文件复制到内存的字节个数。

说明：read() 操作从文件的当前位置开始，该位置由包含在相应的文件描述中的偏移值(offset)给出。每一次读操作结束，偏移值被加上刚复制的字节数，以便下一次的 read() 操作。

(5) 写文件的系统调用 write()

用 write() 系统调用，可以将数据写到文件中。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

正确返回参数：0 或字节数；

错误返回参数：-1；

其中：

fd 是想要写的文件的文件描述符。

buf 是指向内存的指针。write() 从 buf 所指的内存块中将 nbytes 个字节写到 fd 所指示的文件中。

Nbytes 从内存复制到文件的字节个数。

说明：

每次写操作，都是将字节写到文件描述中偏移值指示的位置。写操作结束偏移值自动地加上实际写入的字节数，为下一次 write() 做准备。

如果文件是用 O_APPEND 方式打开的，则 write() 作用之前偏移值会自动移到文件结束位置。

(6) 关闭文件的系统调用 close()

用 close() 释放打开的文件描述符。

函数调用包含文件：

```
#include <unistd.h>
```

函数调用原形:

```
int close(int fd);
```

其中:

fd 为文件描述符。

正确返回参数: 0;

错误返回参数: -1;

每次打开一个文件时, 文件描述中的引用计数加一。用 close() 关闭该文件时, 文件描述中的引用计数减一。直到某次调用 close() 时, 使引用计数值为 0 了, 系统才释放文件描述符及描述信息。

(7) 设置当前读写位置 lseek()

Linux 系统中可用 lseek() 将当前文件的偏移值移到相关的位置, 设置下一次 read() 或 write() 访问的位置。

函数调用包含文件:

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

函数调用原形:

```
off_t lseek(int fd, off_t offset, int whence);
```

其中:

fd 为文件描述符。

Offset 是一个相对值, 被加到基地址上, 给出新的偏移值。

Whence 指示偏移值的计算点。可以选择下列三个值之一:

SEEK_SET 从文件的开始处计算偏移值。

SEEK_CUR 从文件的当前偏移处开始处计算偏移值。

SEEK_END 从文件的结束处计算偏移值。

6. 文件共享

Linux 系统中支持不同进程共享打开的文件。内核中使用三种数据结构描述打开的文件以及文件的共享关系。

1) 进程打开文件表

每个进程的进程表中都有一个打开文件表，该表的每一个表项记录了一个进程打开的文件。每个进程最多可打开 20 个文件。进程打开文件表的表项主要包括：

- 文件描述符标志

- 指向系统打开文件表中相应表项的指针

2) 系统打开文件表

系统打开文件表记录系统已打开的文件，每个表项对应了一个打开的文件，主要包括：

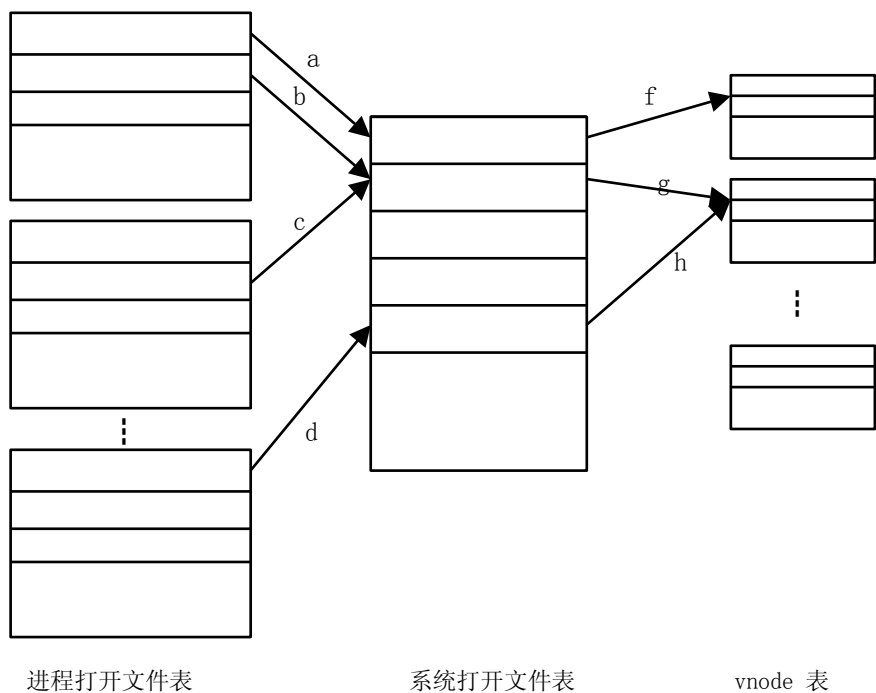
- 文件打开状态标志

- 文件当前偏移地址

- 打开文件的内存 vnode 表入口

3) 内存中的索引节点表 vnode

是 Linux 中每一个打开的文件在内存中的索引节点表。主要包含了该文件的 inode 信息。当文件打开时，该 inode 信息从磁盘中读出，以便文件的信息可以被进程使用。



- 图中：
- a、b 为同一个进程打开的不同文件，指向系统打开文件表中的各自的表项。
 - d 和 b、c 为不同进程打开的文件，指向系统打开文件表中的各自的表项。
 - b、c 为父进程和子进程的文件共享，或通过 `dup()` 系统调用产生的文件共享，所以指向系统打开文件中表中的同一个表项。用系统打开文件表项的共享计数指示共享进程数。
 - f、g、h 是系统打开文件表中的不同表项指向 vnode 的指针。其中 g、h 是不同进程以同一文件名或不同文件名方式打开的指向同一个文件的指针，用 vnode 中的共享计数指示。每关闭一个文件，计数减一，直到计数为零，将内存 inode 复制回磁盘。

(1) 复制文件描述字的系统调用 `dup()`

一个文件的描述符可以利用以下方式复制，所得到的文件描述符可以传递给其他进程从而达到文件的共享。

函数调用包含文件：

```
#include <unistd.h>
```

函数调用原形：

```
int dup(int fd);
```

或 `int dup2(int fd, int fd2);`

返回参数：最小的可用文件描述符。

另一个复制文件描述符的方式是 `fcntl()` 系统调用。

`dup(fd)` 等同于系统调用

```
fcntl(fd, F_DUPFD, 0)
```

```
dup2(fd, fd2) 等同于系统调用
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

(2) 链接文件的系统调用 `link()`

当文件被首次创建时，从给定的路径名自动地产生一个文件的链接。所以文件的一个目录项称为一个链接（link）。此后，对这个文件的硬链接可以通过 `link()` 系统调用产生。

每建立一个文件链接，文件的索引节点(inode)中的链接计数加一。每删除一个链接文件，链接计数减一，直到计数为零，系统回收该索引节点(inode)以及分配给该文件的磁盘数据块。

函数调用包含文件：

```
#include <unistd.h>
```

函数调用原形：

```
int link(char *origfile, char *linkname);
```

正确返回参数：0

错误返回参数：-1

其中：

<code>origfile</code>	是已有的文件
<code>linkname</code>	是新创建的目录链接

说明：

要建立硬链接文件，用户必须对 `origfile` 有读的权限，对 `linkname` 所在的目录有写和执行的权限。只有超级用户有权创建对目录的硬链接。

(3) 移去链接文件的系统调用 `unlink()`

这个系统调用移去指定的文件硬链接并将它的索引节点的链接计数减一。如果索引节点的链接计数已经变成零，则索引节点和文件数据块全部释放。

执行此操作，必须具有对含有该文件目录的目录的可写和可执行权限。

函数调用包含文件：

```
#include <unistd.h>
```

函数调用原形：

```
int unlink(char *pathname);
```

正确返回参数：0

错误返回参数：-1

(4) 建立符号文件的系统调用 `symlink()`

符号链接比硬连接更灵活。它可以链接位于另一个文件系统的文件，几乎所有与文件相关的系统调用，都会自动引用符号链接所指向的文件，除了 `chown()`, `lstat()`, `readlink()`, `rename()`, `unlink()`。

◆ 系统调用 `symlink()` 可以用来建立符号链接。

函数调用包含文件:

```
#include <unistd.h>
```

函数调用原形:

```
int symlink(const char *origfile, const char *linkname);
```

说明:

调用成功 linkname 将称为指向 origfile 的符号链接。

- ◆ 要确定符号链接文件所指向的目标文件可以使用 readlink() 系统调用。

函数调用包含文件:

```
#include <unistd.h>
```

函数调用原形:

```
int readlink(const char *linkname, const char *buf, size_t bufSize);
```

说明:

函数会在 buf 指向的缓冲区返回符号链接文件所指向的目标文件。成功时函数返回文件名的长度，失败时返回 -1。

7. 创建设备文件和管道的系统调用

(1) 创建特殊文件 mknode()

mknode() 函数可以用来，包括有名管道、字符设备、块设备等。

函数调用包含文件:

```
#include<sys/types.h>
```

```
#include<sys/stat.h>
```

```
#include<fcntl.h>
```

```
#include<unistd.h>
```

函数调用原形:

```
int mknode(const char *pathname, mode_t mode, dev_t dev);
```

其中:

pathname 指定该设备的名字。

Mode 指定文件类型和存取权限。文件类型可以是，S_IFIFO(管道文件), S_IFBLK (块文件), S_IFCHR (字符设备)。

Dev 指示新创建的设备的主设备号和从设备号。若是 S_IFIFO 文件次项参数不用)

(2) 用来创建无名管道 Pipe() 函数

无名管道用于父进程和子进程或兄弟进程之间的通信。

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

```
int pipe(int fds[2]);
```

说明:

该系统调用在 `fds` 中返回两个文件描述符, `fds[0]`用于只读, `fd[1]`用于只写。

8. 文件目录结构

(1) 获得当前目录 `getcwd()`、`get_current_dir_name()`、`getwd()`

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

```
char *getcwd(char *buf, size_t size);
```

或 `char *get_current_dir_name(void);`

或 `char *getwd(char *buf);`

说明: 函数 `getcwd()`把当前目录的绝对路径名拷贝到 `buf` 指示的缓冲区中。`Buf` 的大小为 `size` 个字节。当 `buf` 为 `NULL` 时, `getcwd()` 会调用 `malloc()` 分配所需的缓冲区。

程序中应负责用 `free()` 将缓冲区释放

(2) 设置当前目录 `chdir()`、`fchdir()`

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

```
int chdir(const char *path);
```

或 `int fchdir(int fd);`

其中:

`path` 希望成为当前目录的目录

`fd` 文件描述符

成功返回:0

错误返回:	<code>ENOTDIR</code>	所指定的路径中有错
	<code>EACCESS</code>	对目录没有执行权
	<code>EBADF</code>	<code>fd</code> 不是有效的文件描述符

(3) 改变根目录 `chroot()`

系统只有一个根目录, 但是每个进程可以有自己的根目录。这样可以防止一些不安全的进程存取整个文件系统。改变了根目录, 并不改变进程的当前目录。进程仍可以通过相对当前目录的路径存取其它目录的文件。

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

```
int chroot(const char *path);
```

其中:

path 是作为根目录的路径

(4) 创建目录 mkdir()

函数调用包含文件:

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

函数调用原形:

```
int mkdir(const char *dirname, mode_t mode);
```

其中:

dirname 指定的要创建的目录名。

ode 该目录的存取权限。最终的目录还受到 umask 的影响，
即 mode & umask。

说明: 如果 dirname 已经存在或 dirname 中某部分无效。函数返回-1。

新创建的目录的 uid 是进程的有效 uid。gid 的设置或者按系统安装的父目录的 gid，或者进程的有效 gid 将成为新目录的 gid。

(5) 删除目录 rmdir()

函数调用包含文件:

```
#include<unistd.h>
```

函数调用原形:

```
int rmdir(const char *dirname);
```

其中:

dirname 指定的要删除的目录名。

说明: 只有空目录才能被删除，否则 rmdir()将返回 -1，同时 errno 等于 ENOTEMPTY。

(6) 目录访问 opendir()、readdir()、rewinddir()、closedir()

应用程序经常需要了解包含在一个目录中的文件的信息。Linux 可以同时支持多种文件系统。为了使应用程序不涉及具体文件系统的目录格式，Linux 提供了一组系统调用函数帮助应用程序按照一种抽象方式处理目录。

φ 为了打开一个目录，可以使用 opendir() 系统调用。

函数调用包含文件:

```
#include<sys/types.h>
#include<diret.h>
```

函数调用原形:

```
DIR *opendir(const char *pathname);
```

正确返回: 目录指针。

错误返回: 0;

这个系统调用打开指定的目录。如果成功，返回一个目录指针。该目录指针被传递给有关的系统调用 `readdir()` 和 `closedir()`。

- ❖ `readdir()` 系统调用返回一个指向 `dirent` 结构的指针。该结构含有指定目录下的一个文件链接的内容，`ino_t` 号和文件名。重复调用 `readdir()` 可以在目录中顺序访问所有的链接，直到最后返回一个 0 值。

函数调用包含文件：

```
#include<sys/types.h>
#include<dirent.h>
```

函数调用原形：

```
struct dirent *readdir(DIR *dp);
```

正确返回： `dirent` 的指针。

错误返回： 0；

`dirent` 结构如下：

```
struct dirent {
    ino_t    d_ino;        /*inode 节点号*/
    char     d_name[NAME_MAX+1] ;    /*文件名*/
}
```

- ❖ 要从头读取打开的目录的内容，可以使用 `rewinddir()` 系统调用。它将复位 `dirent` 结构的数据，这样下次再用 `readdir()` 将返回该目录的第一项。

- ❖ `closedir()` 系统调用关闭打开的目录。

函数调用包含文件：

```
#include<sys/types.h>
#include<dirent.h>
```

函数调用原形：

```
int closedir(DIR *dp);
```

正确返回： 0；

错误返回： -1；

9. 改变文件属性

- (1) 改变文件名称 `rename()`

一个文件或目录可以由 `rename()` 系统调用修改名称。

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
#include<stdio.h>
```

```
int rename(const char *oldname, const char *newname) ;
```

正确返回： 0；

错误返回： -1；

说明：用户应对包含 oldname 和 newname 的目录具有可写的权限和可执行的权限。

(2) 改变文件读取权限 `chmod()`、`fchmod()`

用于改变文件权限位的系统调用 `chmod()`、`fchmod()` 可以改变文件所有者、用户组和其它人的读、写和执行位。

`chmod()` 用来改变给定的 `pathname` 的文件权限位，而 `fchmod()` 用来改变给定的 `fd` 的文件权限位。

函数调用包含文件：

```
#include<sys/types.h>
```

```
#include<sys/stat.h>
```

函数调用原形：

```
int chmod(char *pathname, mode_t mode) ;
```

或

```
int fchmod(int fd, mode_t mode) ;
```

正确返回：0；

错误返回：-1；

如：`fchmod(fd, S_ISUID|S_IRWXU|S_IXGRP|S_IXOTH)`；

等同于 `fchmod(fd, 04711)`；

文件和目录的默认模式是由 `umask` 来设置的，`umask` 用数字定义其值。用户进入系统时，该值被放在文件系统中，创建的文件若未指定模式，就用 `umask` 来自动设置其许可。当改变一个文件的权限位时，所规定的位，将自动地由当前的 `umask` 的值按以下公式加以修改。

$$\text{Mode} \& (\sim \text{umask})$$

`Umask` 的值也可以用 `umask()` 设置。

函数调用包含文件：

```
#include<unistd.h>
```

函数调用原形：

```
mode_t umask(mode_t mask);
```

正确返回：0；

错误返回：-1；

这个函数用新的 `mask` 的值代替 `umask` 的当前值。

(3) 改变文件的所有者 `chown()`、`fchown()`

`chown()` 和 `fchown()` 系统调用用来改变文件所有者识别号或它的用户组识别号。

函数调用包含文件：

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

函数调用原形：

```
int fchown(int fd, uid_t owner, gid_t group) ;  
int chown(const char *pathname, uid_t owner, gid_t group) ;
```

正确返回: 0;

错误返回: -1;

fchown() 给 fd 所指定的文件描述符赋予新的所有者识别号和组织识别号。chown() 给路径名为 pathname 的文件赋予新的所有者识别号和组织识别号。

Linux 中只有 root 帐号可以使用 chown() 和 fchown() 系统调用。

(4) 改变或设置打开文件的属性 fcntl()

fcntl() 系统调用用于更改正在使用中的文件的属性, 如复制文件描述字、设定 close_on_exec 标志等。

函数调用包含文件:

```
#include<unistd.h>  
#include<fcntl.h>
```

函数调用原形:

```
int fcntl(int fd, int cmd) ;  
或 int fcntl(int fd, int cmd, long arg) ;  
或 int fcntl(int fd, int cmd, struct flock *lock) ;  
常用第二种格式。
```

其中:

fd 文件的描述字。

cmd 为命令, 共 5 种格式

- 1) F_DUPFD 拷贝 fd 到 arg, 如果需要首先关闭 fd。
- 2) F_GETFD 读取并返回 close_on_exec 标志。如果低位字节为 0, 执行 exec() 之后, fd 指向的文件仍保持在打开状态; 反之, 在执行 exec() 之后, 该文件会自动关闭。
- 3) F_SETFD 设置 close_on_exec 标志为 arg 指定的值。
- 4) F_GETFL 读取并返回已打开文件的标志状态。
- 5) F_SETFL 设置由 arg 指定的标志。可设置的标志有 O_APPEND、O_NOBLOCK 和 O_ASYNC。标志设置通过按位 OR 来进行, 因此其它标志不受影响。

六. 标准输入输出

标准输入输出的系统调用都存在于标准输入输出库中。标准输入输出库是 C 语言编写的, 所以不光应用于 Linux, 也可以应用于其它系统。而基于文件系统的输入输出与操作系统关系很大, 不能应用于其它系统。基于文件系统的输入输出都是围绕文件描述符展开的, 打开一个文件, 返回的是文件描述符, 并被应用于输入输出操作。而标准输入输出是围绕“流”(stream)进行的, 当调用标准输入输出创建一个文件时, 就将一个“流”和文件关联在一起了。

1. 标准输入输出的基本操作

在 Linux 系统中，文件和设备都被认为是数据流。在对流进行操作前，需要将它打开。操作完成后，需要通过操作系统清空缓冲区、保存数据。最后应关闭“流”。

只有一个流与某文件或设备关联起来，才可以对这个流进行各种操作，这就是流的打开操作。打开操作成功的话，系统将返回一个 FILE 结构的指针，以后的操作都可以借助这个指针，通过调用系统函数来完成了。

执行程序时自动打开了三个流，它们是标准输入、标准输出和标准错误输出。相应的 FILE 结构指针为 stdin、stdout、stderr。它们和 STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO 文件描述符对应的文件是相同的。

(1) 流的打开操作 fopen()、freopen()、fdopen()

以下三个系统调用都可以打开一个标准输入输出流。

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
FILE *fopen(const char *pathname, const char *type);  
或 FILE *freopen(const char *pathname, const char *type, FILE *fp);  
或 FILE *fdopen(int fildes, const char *type);
```

正确返回：文件结构的指针；

错误返回：空指针；

这三个系统调用的不同之处在于：

 fopen 打开一个特定的文件，文件名由 pathname 指出。

 freopen 在特定的流上打开一个特定的文件。它先关闭 FILE *fp 指定的已打开的流，再打开 pathname 指定的流。一般用于打开特定文件代替标准输入流、标准输出流、标准错误输出流。

 fdopen 将一个流对应到某个已打开的文件上，fildes 就是这个文件的描述符。已打开文件的模式应与 type 定义的模式相同。这个已打开文件一般是管道文件或网络通信管道，这些文件无法通过标准输入输出函数打开，只能通过特定设备函数得到文件描述符，再用 fdopen 函数将一个流与这个文件关联起来。

三个系统调用中的 char *type 表示了流的打开模式：

type	读打开	写打开	文件长度截为零	新建文件	开始读写位置
“r” 或 “rb”	是	否	否	否	文件开头
“r+” 或 “r+b” 或 “rb+”	是	是	否	否	文件开头
“w” 或 “wb”	否	是	是	是	文件开头
“w+” 或 “w+b” 或 “wb+”	是	是	是	是	文件开头
“a” 或 “ab”	否	是	否	是	文件末尾
“a+” 或 “a+b” 或 “ab+”	是	是	否	是	文件末尾

其中“r”为允许读操作、“w”为允许写操作、“a”为允许从文件尾继续写操作、“b”为允许标准输入输出系统区分文本文件和二进制文件。

(2) 流的清洗操作 `fflush()`、`fpurge()`

流的清洗操作是指将输入输出缓冲区的内容写入一个文件或刷新缓冲区。以下两个系统调用都可以实现清洗。

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
int fflush(FILE *fp);
```

或 `int fpurge(FILE *fp);`

正确返回：0；

错误返回：EOF；

其中 `FILE *fp` 就是已经打开的流。它们的区别是：

`fflush` 将缓冲区内容保存在磁盘上，并清空缓冲区。

`fpurge` 将缓冲区中所有数据清除掉。

(3) 流的关闭操作 `fclose()`

用户在应用程序打开的流必须在程序结束时关闭。否则可能造成数据丢失。

流的关闭操作用以下系统调用：

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
int fclose(FILE *fp);
```

正确返回：0；

错误返回：EOF；

(4) 流缓冲区属性的设定 `setbuf()`、`setbuffer()`、`setlinebuf()`、`setvbuf()`

流缓冲区的属性包括大小和类型。缓冲区类型有以下三种：

全缓冲：缓冲区满时才真正执行输入输出，即将缓冲区内容保存到磁盘文件或输出到终端上。

行缓冲：缓冲区中输入一个换行符时，才真正执行输入输出。这种缓冲允许一次一个字符地在终端上输出。一般与终端设备相连的缓冲采用这种方式。

无缓冲：一接受到字符，就执行输入输出。错误标准输出都采用这种方式。

用户打开流时系统就为其设置了默认的缓冲属性。用户也可以利用以下系统函数设定自己想要的缓冲属性。

函数调用包含文件：

```
#include<stdio.h>
```

函数调用原形：

```
int setbuf(FILE *fp, char *buf);
```

或 `int setbuffer(FILE *fp, char *buf, size_t size);`
或 `int setlinebuf(FILE *fp);`
或 `int setvbuf(FILE *fp, char *buf, int mode, size_t size);`

正确返回:0;

错误返回:非 0;

其中:

<code>FILE *fp</code>	已打开的流。
<code>char *buf</code>	用户自己设定的缓冲区。
<code>size_t size</code>	缓冲区大小。
<code>int mode</code>	流德类型, 可以取 <code>_IOFBF</code> 、 <code>_IOLBT</code> 、 <code>_IONBF</code> 分别代表全缓冲、行缓冲和无缓冲。

说明: 调用这些函数时, 必须先打开流, 而且最好还没有进行其它操作。因为其它操作是与缓冲区的性质密切相关的。

(5) 流的定位操作 `ftell()`、`fseek()`、`rewind()`、`fgetpos()`、`fsetpos()`

用户打开流以后, 每次读写操作, 读写指针都会移动一个单元。对于块设备, 用户可以通过系统调用了解指针所处的位置, 并可以将指针移到任何一个位置。完成流定位的系统调用有以下几个。

φ 对于二进制文件, 读写指针从文件开头, 以字节为单位开始计数。通过 `ftell()` 函数可以得到这个值。

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
long ftell(FILE *fp);
```

正确返回:读写指针的位置;

错误返回:-1;

φ `fseek()` 函数用来定位二进制文件。但必须指定偏移量的大小 `offset` 和偏移量的使用方法 `whence`。Whence 可以取值 `SEEK_SET` (从文件头开始计算)、`SEEK_CUR` (从文件当前位置计算)、`SEEK_END` (从文件结尾开始计算)。

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
long fseek(FILE *fp, long offset, int whence);
```

正确返回:0;

错误返回:非零;

φ `rewind()` 函数用来将读写指针移到文件的开头。系统默认这个系统调用总是成功的, 所以没有返回参数。

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
void rewind(FILE *fp);
```

无返回值;

- φ 用户可以使用抽象的数据结构 `fpos_t` 来存放读写指针的位置。`fgetpos()` 可以得到读写指针的位置。

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
int fgetpos(FILE *fp, fpos_t *pos);
```

正确返回:0;

错误返回:非零;

- φ `fsetpos()` 可以定位读写指针的位置。

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
int fsetpos(FILE *fp, const fpos_t *pos);
```

正确返回:0;

错误返回:非零;

2. 非格式化输入输出操作

非格式化输入输出分为: 字符型输入输出、一行型输入输出、直接型输入输出。

字符型输入输出操作

(1) 字符型输入 `getc()`、`fgetc()`、`getchar()`

字符型输入输出每次可以读写一个字符。标准输入输出函数会自动处理缓冲区问题。

字符型输入的操作有以下三个函数调用:

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
int getc(FILE *fp);
```

或 `int fgetc(FILE *fp);`

或 `int getchar(void);`

文件末尾或出错返回: EOF (C 语言中值为 -1);

其它情况返回: 下一个要读入的字符;

说明:

`getchar()` 的作用相当于调用函数 `getc(stdin)`, 将标准输入流作为参数传进去。

函数 `getc` 可以当作宏, 而 `fgetc` 只能作为普通函数调用。

(2) 字符型输出 `putc()`、`fputc()`、`putchar()`

字符型输出的操作与输入函数是对应的, 有以下三个函数调用:

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
int putc(int c, FILE *fp);
```

或 `int fputc(int c, FILE *fp);`

或 `int putchar(int c);`

正确返回: 字符 `c`;

错误返回: `EOF`;

说明:

`putchar(c)` 的作用相当于调用函数 `putc(c, stdout)`, 将标准输出作为参数传进去。

函数 `putc` 可以当作宏, 而 `fputc` 只能作为普通函数调用。

(3) 判断文件尾和文件出错 `ferror()`、`feof()`

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
int ferror(FILE *fp);
```

```
int feof(FILE *fp);
```

正确返回: 非零;

错误返回: 0;

(4) 将字符推回缓冲区 `ungetc()`

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
int ungetc(int c, FILE *fp);
```

正确返回: 字符 `c`;

错误返回: `EOF`;

一行型输入输出操作

使用这种方式, 每次可以读写一行。系统提供如下两个函数处理一行的输入。

(1) 一行型输入 `fgets()`、`gets()`

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
char* fgets(char *buf, int n, FILE *fp);
```

```
char* gets(char *buf);
```

其中:

buf 接受输入的缓冲区的地址

fp 流结构的指针

正确返回: buf;

错误返回: 空指针;

说明:

fgets 需要明确接受的字节数 n。读完一行, 将这一行的内容包括行结尾标志, 一起保存在 buf 中, 用空字符 0 来结束。因此, 这一行内容包括行结尾在内, 不能超过 n-1 个。如果超过 n-1 个字符, 系统接受这一行的一部分, 其余在下次调用 fgets 时读入。

gets 是从标准输入设备输入的。

(2) 一行型输出 fputs()、puts()

函数调用包含文件:

```
#include<stdio.h>
```

函数调用原形:

```
char* fputs(const char *str, FILE *fp);
```

或 char* puts(const char *str);

其中:

char *str 要输出的字符串

FILE *fp 流结构的指针

正确返回: 非负数;

错误返回: EOF;

说明:

fputs 把以字符 0 结尾的字符串输出到指定的流文件中, 但结尾的字符 0 并不输出。

puts 把字符 0 结尾的字符串输出标准输出设备上, 结尾的字符 0 也不输出, 最后加一个换行符。

例: 使用行输入输出的程序。打开命令行指定的流文件 (命令行参数用 argv[i] 表示, 其中 i 为参数序号), 从文件中输入一行, 在屏幕上输出一行。

```
#include<sys/types.h>
```

```
#include<stdio.h>
```

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

```

{
    char s[1024];
    FILE *fp;
    if((fp=fopen(argv[1], "r")) != (FILE*)0)    /*打开流文件，判是否错*/
    {
        while((fgets(s, 1024, fp)) != (char *)0) /*从文件输入一行*/
            puts(s);                            /*再在屏幕输出一行*/
    }
    else
    {
        fprintf(stderr, "file open error.\n");
        exit(1);
    }
    exit(0);
}

```

直接型输入输出操作

字符型输入输出操作在二进制文件的处理中。输入输出以结构为单位，一次可以处理若干个记录。

(1) 直接型输入操作 fread ()

函数调用包含文件：

#include<stdio.h>

函数调用原形：

size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);

其中：

void *ptr 指向若干结构的指针
size_t siz 结构大小（一般可用 sizeof() 获得）
nobj 要处理的结构的个数
FILE *fp 流结构的指针

正确返回：实际读出的数目

(2) 直接型输出操作 fwrite ()

函数调用包含文件：

#include<stdio.h>

函数调用原形：

size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *fp);

其中：

void *ptr 指向若干结构的指针
size_t siz 结构大小（一般可用 sizeof() 获得）
nobj 要处理的结构的个数
FILE *fp 流结构的指针

正确返回：实际写入的数目

附录 A 实验报告格式

《计算机操作系统》实验报告

实验题目：进程控制 -----

姓名： 学号： 实验日期：

实验环境：

实验目的：

实验内容：

操作过程：

结果：

体会：

附录：（源程序）

实验报告以A4纸打印，共约3千字(除附录外)

第一行标题：
宋体、二号、加粗、
居中

题目栏内及其余小标题：
黑体、三号、加粗

正文内容：
宋体、五号、

附录 B 参考资料

计算机操作系统教（第三版）	张尧学、史美林	清华大学出版社
操作系统教程—原理和实例分析	孟静	高等教育出版社
现代操作系统（第三版）	塔嫩鲍姆 著、陈向群等译	机械工业出版社
计算机操作系统教程	陆松年	电子工业出版社
Red Hat Linux 大全	David Pitts 等姚彦忠、赵小杰等译	机械工业出版社
操作系统及实验教程	李善平、郑扣根	机械工业出版社
Linux 上的 C 编程	怀石工作室	中国电力出版社
Linux 管理员指南	何田、宋建平、李舒、阎瑞雪	清华大学出版社
Linux 编程指南	徐严明等	科学出版社
Linux C 高级程序员指南	毛曙福	国防工业出版社
Linux 技术内幕	Moshe Bar 著、王超译	清华大学出版社
Linux 基础及应用习题分析与实验指导	谢蓉	中国铁道出版社