实验一　UNIX/LINUX 及其使用环境
（一）UNIX/LINUX 常用命令简介

实验目的

1、了解 UNIX/LINUX 的命令及使用格式。

2、熟悉 UNIX/LINUX 的常用基本命令。

实验内容

1、通过 WINDOWS 操作系统中的远程登录程序 telnet.exe 登录 LINUX。

2、熟悉 UNIX/LINUX 的常用基本命令如 ls、who、cp、pwd、ps、pstree、top 等。

3、通过 WINDOWS 操作系统中的 FTP 程序，练习 WINDOWS 和 LINUX 之间的文件交换。

实验预备内容

预习附录一《UNIX/LINUX 简介》

实验指导

一、UNIX的登录与退出

1、登录

在DOS环境下用MS提供的telnet程序（也可使用WINDOWS 自带的telnet图形界面程序或多功能的S-Term终端程序），可使PC作为终端（terminal）登录（login）UNIX/LINUX 服务器（UNIX　Server）。

（1）执行格式：

telnet　hostname(主机名)

或　telnet　主机的 IP 地址

例：　telnet　www.yahoo.com

telnet　140.122.77.120

（2）登录步骤

login：　（输入用户名）

password：　（输入口令）

2、退出

在 UNIX 系统提示符$下，输入 logout、exit 或 shutdown 。

例：$ logout

二、UNIX 命令格式

命令　[选项]　[处理对象]

例：ls　-la　mydir

注意：（1）命令一般是小写字串。注意大小写有别；

（2）选项通常以减号（-）再加上一个或数个字符表示，用来选择一个命令的不同操作；

（3）同一行可有数个命令，命令间应以分号隔开；

（4）命令后加上&可使该命令后台（background）执行。

三、常用命令

1、目录操作

和 DOS 相似，UNIX 采用树型目录管理结构，由根目录（/）开始一层层将子目录建下去，各子目录以 / 隔开。用户 login 后，工作目录的位置称为 home directory，由系统管理员设定。‘~’符号代表自己的 home directory，例如 ~/myfile 是指自己 home 目录下 myfile 这个文件。

UNIX 的通配符有三种：’*’ 和 ’?’ 用法与 DOS 相同， ‘-‘ 代表区间内的任一字符，如 test[0-5]即代表 test0，test1，……，test5 的集合。

（1）***显示目录文件 ls***

执行格式： ls [-atFlgR] [name] (name 可为文件名或目录名)

例： ls 显示出当前目录下的文件

ls -a 显示出包含隐藏文件的所有文件

ls -t 按照文件最后修改时间显示文件

ls -l 显示目录下文件的许可权、拥有者、文件大小、修改时间及名称

ls -lg 同上

ls -R 显示出该目录及其子目录下的所有文件

注：更多用法请输入ls --help查看，其它命令的更多用法请输入 命令名 --help 查看.

（2）***建新目录 mkdir***

执行格式： mkdir directory-name

例： mkdir dir1 （新建一名为 dir1 的目录）

（3）***删除目录 rmdir***

执行格式： rmdir directory-name 或 rm directory-name

例： rmdir dir1 删除目录 dir1，但它必须是空目录，否则无法删除

rm -r dir1 删除目录 dir1 及其下所有文件及子目录

rm -rf dir1 不管是否空目录，都删除，而且不给出提示,使用时要小心

（4）***显示当前所在目录 pwd***

执行格式： pwd 显示当前所在目录的全路径

（5）***改变工作目录位置 cd***

执行格式： cd [name]

例： cd 改变目录位置至用户 login 时的 home directory

cd dir1 改变目录位置，至dir1目录

cd .. 改变目录位置，至当前目录的上层目录

cd ../user 改变目录位置，至上一级目录下的user目录

cd /dir-name1/dir-name2 改变目录位置，至绝对路径

cd - 回到进入当前目录前的上一个目录

（6）***查看目录大小 du***

执行格式： du [-s] directory

例： du dir1 显示目录 dir1 及其子目录容量（以 kb 为单位）

du -s dir1 显示目录 dir1 的总容量

（7）***显示环境变量***

echo $HOME 显示 home 目录

echo $PATH 显示可执行文件搜索路径

env 显示所有环境变量(可能很多,最好用"env | more","env | grep PATH"等)

（8）***修改环境变量，在bash下用 export,***如：

export PATH=$PATH:/usr/local/bin

想知道 export 的具体用法，可以用 shell 的 help 命令：help export

2、文件操作

（1）***查看文件内容　cat，more***

执行格式：cat filename 或 more filename 或 cat filename | more

例：cat　file1　　以连续显示方式，查看文件 file1 的内容

more　file1　或

cat　file1 | more　以分页方式查看文件 file1 的内容

（2）***删除文件　rm***

执行格式：　rm　filename

例：rm　file?　　删除前四个字符为 file，第五个字符不定的所有文件

rm　f*　　删除以字符 f 开头的所有文件

（3）***复制文件　cp***

执行格式：　cp　[-r]　source　destination

例：　cp　file1　file2　　将 file1 复制成 file2

cp　file1　dir1　　将 file1 复制到目录 dir1

cp　/tmp/file1　　将 file1 复制到当前目录

cp　/tmp/file1　file2　　将 file1 复制到当前目录，名为 file2

cp　–r　dir1　dir2　　(recursive　copy)复制整个目录。

（4）***移动或更名　mv***

执行格式：　mv　source　destination

例：　mv　file1　file2　　将文件 file1 更名为 file2

mv　file1　dir1　　将文件 file1 移到目录 dir1 下

mv　dir1　dir2　　将目录 dir1 更名为 dir2

（5）***比较文件(可以是二进制的)或目录的内容　diff***

执行格式：　diff [-r]　name1　name2　(name1、name2 同为文件或目录)

例：　diff　file1　file2　　比较 file1 与 file2 的不同处

diff　-r　dir1　dir2　　比较 dir1 与 dir2 的不同处

（6）***文件中字符串的查找　grep***

执行格式：　grep　string　file

例：　grep　abc　file1　　查找并列出文件 file 中包含串 abc 的行

（7）*文件或命令的寻找*

执行格式一：whereis　command　显示命令的路径

执行格式二：which　command　显示路径及使用者所定义的别名

执行格式三：whatis　command　显示命令的功能摘要

执行格式四：find　search_path　-name　filename　-print

搜寻指定路径下某文件的路径

例：　find　/　-name　message　-print

在全系统中寻找文件 message 是否存在，若有，则列出所在位置

执行格式五：locate filename

根据系统预先生成的文件/目录数据库(/var/lib/slocate/slocate.db)查找匹配的文件/目录,查找速度很快,如果有刚进行的文件改变而系统未到执行定时更新数据库的时间,可以打入updatedb命令手动更新.

例：locate　msg

列出系统中所有名字中包含字符串 msg 的文件和目录

（8）***建立文件或目录的链接　ln***

例：　ln　source　target1　建立source文件（已存在）的硬链接,命名为target1

ln -s　source　target2　建立source文件的符号链接,命名为target2

3、系统询问与改变权限

（1）***查看系统中的使用者　who***

执行格式：　who

例：　who　列出当前在系统中的用户

who　am　I　查看自己的用户名

（2）***改变自己目前使用系统的帐号　su***

执行格式：　su　username

例：　su　username2

password：　提示您输入新帐号口令

UNIX/LINUX 系统给不同的用户赋予不同的权限。root 用户具有最高的权限，可执行任何操作。系统管理员一般情况下应以普通用户进入系统，若在操作过程中遇到权限不够，再用 su 命令改变到 root 帐号。

（3）***改变文件或目录的访问权　chmod***

执行格式：　chmod　[-R]　mode　name

其中：[-R]为递归处理,将指定目录下所有文件及子目录一并处理

mode用3个八进制数表示，分别表示不同类型用户对文件的读、写、执行允许权(r:read,数值为 4， w:write,数值为 2，x:execute,数值为 1)

mode：　rwx　rwx　rwx

用户类：　拥有者(o)　同组用户(g)　任何用户(w)(a 的情况直接命令方式才可用)

例：chmod　755　dir1　将目录dir1设定成任何人皆有读取及执行的权利，但只有拥有者可作写修改。其中7=4+2+1, 5=4+1

chmod　700　file1　将 file1 设为只有拥有者可以读、写和执行

chmod　o+x　file2　对 file2，增加拥有者可执行的权利

chmod　g+x　file3　对 file3，增加同组用户可执行的权利

chmod　w-r　file4　对 file4，除去其它使用者可读取的权利

（4）***改变文件或目录拥有者　chown***

执行格式：　chown　[-R]　username　filename

例：　chown　user　file1　将文件 file1 改为 user 所有

chown　:fox　file1　将文件file1改为fox组所有

chown　user:fox　file1　将文件file1改为fox组的user所有

chown　-R　user　dir1　将目录 dir1 及其下所有文件和子目录，改为 user 所有

（5）***检查用户所在组名称　groups***

执行格式：　groups

（6）***改变文件或目录的组拥有权　chgrp***

执行格式：　chgrp　[-R]　groupname　filename

例：　chgrp　vlsi　file1　将文件 file1 改为 vlsi 组所有

chgrp　-R　image　dir1　将目录 dir1 及其下所有文件和子目录,改为 image 组

4、进程操作

（1）***查看系统中目前的进程　ps***

执行格式：　　ps　[-aux]

例：　ps　或 ps　-x　　查看系统中属于自己的 process

　　ps　-au　　查看系统中所有使用者的 process

　　ps　-aux　　查看系统中包含系统内部及所有使用者的 process

　　ps　-aux|grep　apache　　找出系统中运行的所有名称中带有"apache"串的 process

（2）***查看正在 background 中执行的 process***

执行格式：　　jobs

（3）***结束或终止进程　　kill***

执行格式：　kill　[-9]　PID　（PID 为利用 ps 命令所查出的进程 ID）

例：　　kill　-9　456　　终止进程 ID 为 456 的进程

（4）***后台（background）执行 process command 的命令***

执行格式：　command　&　　（在命令后加上 &）

例：　　gcc　file1　&　　在后台编译 file1.c

注意：按下^Z，暂停正在执行的 process。键入"bg"，将所暂停的 process 置入 background 中继续执行。

例：　gcc　file1　&

　　^Z

　　stopped

　　bg

（5）***结束或终止在 background 中的进程　　kill***

执行格式：　　kill　%n

例：　kill　%1　　终止在 background 中的第一个 job

　　kill　%2　　终止在 background 中的第二个 job

（6）***显示系统中程序的执行状态　　top***

例：　　top -q　不断地更新、显示系统程序的执行状态

第一行显示的项目依次为当前时间、系统启动时间、当前系统登录用户数目、平均负载。

第二行为进程情况，依次为进程总数、休眠进程数、运行进程数、僵死进程数、终止进程数。

第三行为CPU状态，依次为用户占用、系统占用、优先进程占用、闲置进程占用。

第四行为内存状态，依次为平均可用内存、已用内存、空闲内存、共享内存、缓存使用内存。

第五行为交换状态，依次为平均可用交换容量、已用容量、闲置容量、高速缓存容量。

PID 每个进程的ID。

PPID 每个进程的父进程ID。

UID 每个进程所有者的UID 。

USER 每个进程所有者的用户名。

PRI 每个进程的优先级别。

NI 该进程的优先级值。

SIZE 该进程的代码大小加上数据大小再加上堆栈空间大小的总数。单位是KB。

TSIZE 该进程的代码大小。对于内核进程这是一个很奇怪的值。

DSIZE 数据和堆栈的大小。

TRS 文本驻留大小。

D 被标记为"不干净"的页项目。

LIB 使用的库页的大小。对于ELF进程没有作用。

RSS 该进程占用的物理内存的总数量，单位是KB。
SHARE 该进程使用共享内存的数量。
STAT 该进程的状态。其中S代表休眠状态；D代表不可中断的休眠状态；R代表运行状态；Z代表僵死状态；T代表停止或跟踪状态。
TIME 该进程自启动以来所占用的总CPU时间。如果进入的是累计模式，那么该时间还包括这个进程子进程所占用的时间。且标题会变成CTIME。
%CPU 该进程自最近一次刷新以来所占用的CPU时间和总时间的百分比。
%MEM 该进程占用的物理内存占总内存的百分比。
COMMAND 该进程的命令名称，如果一行显示不下，则会进行截取。内存中的进程会有一个完整的命令行

按"ctrl+c"停止查看

（7）***以树状图显示执行的程序 pstree***

例: pstree -h 列出进程树并高亮标出当前执行的程序

（8）***监视虚拟内存 vmstat***

vmstat对系统的虚拟内存、进程、CPU活动进行监视，同时它也对磁盘和forks和vforks操作的个数进行汇总。

不足是：vmstat不能对某个进程进行深入分析，它仅是一对系统的整体情况进行分析。

例如：[angel@home /angel]# vmstat

```
   procs           memory               swap      io      system      cpu
 r  b  w   swpd   free   buff  cache   si   so   bi   bo   in   cs  us  sy  id
 0  0  0   7180   1852  56092  48400    0    0    6    5   24    8   0   0  18
```

其中：

Procs

r: 等待运行的进程数 b: 处在非中断睡眠状态的进程数 w: 被交换出去的可运行的进程数。

Memory

swpd: 虚拟内存使用情况，单位：KB free: 空闲的内存，单位KB

buff: 被用来做为缓存的内存数，单位：KB

Swap

si: 从磁盘交换到内存的交换页数量，单位：KB/秒 so: 从内存交换到磁盘的交换页数量，单位：KB/秒

IO

bi: 发送到块设备的块数，单位：块/秒 bo: 从块设备接收到的块数，单位：块/秒

System

in: 每秒的中断数，包括时钟中断 cs: 每秒的环境（上下文）切换次数

CPU 按 CPU 的总使用百分比来显示

us: CPU 使用时间 sy: CPU 系统使用时间 id: 闲置时间

（9）***分析共享内存、信号量和消息队列 ipcs***

例如：[angel@home /angel]# ipcs

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00280267 0          root       644        1048576    1
0x61715f01 1          root       666        32000      33
```

```
0x00000000 2          nobody     600        92164      11         dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status
0x00280269 0          root       666        14
0x61715f02 257        root       777        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
```

（10）*监视用户空间程序发出的全部系统调用　strace*

strace　还能显示调用的参数，以及用符号方式表示的返回值。

strace 从内核中接收信息，所以一个程序无论是否按调试方式编译(gcc -g)或是否被去掉了调试信息，都可以被跟踪。

执行格式：　strace　[-tTeo]　executable-program-name

-t ：　用来显示调用发生的时间

-T ：　显示调用花费的时间

-e ：　限定被跟踪的调用类型

-o ：　将输出重定向到一个文件中

类似命令：ltrace　[-fiS]　executable-program-name

5、通信类

（1）***本地工作站与 UNIX 服务器间的文件传输　　ftp***

执行格式：　　ftp　主机名

或　ftp　主机的 IP 地址

后续执行步骤：

name：	输入帐号
password：	输入密码
ftp>help	显示 ftp 可使用的所有命令
ftp>lcd dir1	改变本地机当前目录为 dir1
ftp>get　file1	将 UNIX 服务器文件 file1 拷到本地机
ftp>put　file2	将本地文件 file2，拷到 UNIX 服务器
ftp>!ls	显示本地机当前目录下所有文件
ftp>!pwd	显示本地机当前所在目录下所有文件
ftp>ls	显示 UNIX 服务器当前目录下所有文件
ftp>dir	显示服务器当前目录下所有文件(略同于 UNIX 的　ls -l 指令)
ftp>pwd	显示 UNIX 服务器当前所有目录位置
ftp>cd　dir1	更改 UNIX 服务器的目录至 dir1 下
ftp>mget　*.c	将服务器中 .c　文件拷到本地机中
ftp>mput　*.txt	将所有　.txt　文件拷贝到服务器
ftp>quit	结束 ftp 工作
ftp>bye	结束 ftp 工作

（2）***检查与 UNIX 服务器连接是否正常　　ping***

执行格式：　　ping　hostname

或 ping IP-Address

例： ping 127.1.1.1

（3）***将文件当做 E-mail 的内容送出 mail***

执行格式：mail -s “Subjict-string” username@address < filename

例： mail -s “program” user <file.c

功能：将 file.c 当做 mail 的内容，送至 user，subject name 为 program

（4）***传送 E-mail 给本地 UNIX 服务器上的用户 mail***

执行格式： mail username

（5）***读取信件 mail***

执行格式： mail

（6）***列出套接字使用情况*** *socklist*

（7）***查看网络连接*** *netstat*

6、I/O 命令

（1）***管道（pipe-line）的使用***

执行格式： command1 | command2

功能：将 command1 的执行结果送到 command2 作为输入

例： ls -R | more 以分页方式列出当前目录文件及子目录名称

cat file1 | more 以分页方式，列出 file1 的内容

（2）***标准输入控制***

执行格式： command-line < file 将 file 作为 command-line 的输入

例： mail -s “mail test” user@iis.sinica.edu.tw<file1

功能：将文件 file1 当作信件内容，subject 名称为 mail test 送给收信人

（3）***标准输出控制***

执行格式一： command > filename

功能：将 command 的执行结果送至指定的 filename 中

例： ls -l >list 将执行”ls -l” 的结果写入文件 list 中

执行格式二： command >!filename

功能：同上，若 filename 文件已存在，则强迫重写

执行格式三： command >&filename

功能：将 command 执行所产生的任何信息写入 filename

执行格式四：command >> filename

功能：将 command 的执行结果，附加（append）到 filename

执行格式五：command>>&filename

功能：将 command 执行所产生的任何信息附加于 filename 中

7、其它常用命令

（1）***命令在线帮助 man***

执行格式： man command

例： man ls 查询 ls 这个指令的用法

（2）***设定命令记录表长度 history***

执行格式一： set history = n

例： set history=40

功能：设定命令记录表长度为 40（可记载执行过的前面 40 个命令）

执行格式二： history 查看命令记录表的内容

（3）***显示说明*** *info*

执行格式： info command-name

例:　　info　gcc
功能:　　查看gcc的说明,按上下箭头选定菜单,回车进入,"u"键返回上级菜单.
　　info不加参数则进入最上一级菜单.
四、用cat 命令查看 /proc 动态文件系统目录下的文件,辨识其中的系统信息.
例如: cat interrupts　列出当前中断占用情况
　　cat ioports　　列出设备的硬件IO占用情况
　　cat pci　　列出pci设备的情况

（二）LINUX 下 C 语言使用、编译与调试实验

实验目的

1、复习 C 语言程序基本知识
2、练习并掌握 UNIX/LINUX 提供的 vi 编辑器来编译 C 程序
3、学会利用 gcc、gdb 编译、调试 C 程序

实验内容

1、用 vi 编写一个简单的、显示"Hello,World!"的 C 程序，用 gcc 编译并观察编译后的结果
2、利用 gdb 调试该程序
3、运行生成的可执行文件。

实验指导

一、C 语言使用简介

LINUX 中包含了很多软件开发工具。它们中的很多是用于 C 和 C++应用程序开发的。

C 是一种能在 UNIX 的早期就被广泛使用的通用编程语言。它最早是由 Bell 实验室的 Dennis Ritchie 为了 UNIX 的辅助开发而写的，从此 C 就成为世界上使用最广泛的计算机语言。

C 能在编程领域里得到如此广泛支持的原因有：

（1）它是一种非常通用的语言，并且它的语法和函数库在不同的平台上都是统一的，对开发者非常有吸引力；

（2）用 C 写的程序执行速度很快；

（3）C 是所有版本 UNIX 上的系统语言；

二、文件编辑器 vi

vi 是在 UNIX 上被广泛使用的中英文编辑软件。vi 是 visual editor 的缩写，是 UNIX 提供给用户的一个窗口化编辑环境。

进入 vi，直接执行 vi 编辑程序即可。

例：$ vi test.c

显示屏幕出现 vi 的编辑窗口，同时 vi 会将文件复制一份至缓冲区（buffer）。vi 先对缓冲区的文件进行编辑，保留在磁盘中的文件则不变。编辑完成后，使用者可决定是否要取代原有的文件。

1、vi 的工作模式

vi 提供二种工作模式：插入模式（insert mode）和命令模式（command mode）。使用者开始进入 vi 后，即处在命令模式下，此时键入的任何字符皆被视为命令，可进行删除、修改、存盘等操作。要输入信息，应转换到输入模式。

（1）命令模式下存盘、退出 vi 方式

在命令模式下，可选用下列指令离开 vi：

：w filename	将缓冲区内的资料写入磁盘中，但并不离开 vi
：w　filename	将缓冲区内的资料写入文件 file 中，但并不离开 vi
：q	离开 vi，若文件被修改过，则要被要求确认是否放弃修改的内容，此指令可与：w 配合使用
：q!	离开 vi，并放弃刚在缓冲区内编辑的内容
：wq filename	将缓冲区内的资料写入磁盘中，并离开 vi
：ZZ	同 wq
：x	同 wq

（2）命令模式下光标的移动

H	左移一个字符
J	下移一个字符
K	上移一个字符
L	右移一个字符
0	移至该行的首
$	移至该行的末
^	移至该行的第一个字符处
H	移至窗口的第一列
M	移至窗口中间那一列
L	移至窗口的最后一列
G	移至该文件的最后一列
W, W	下一个单词（W 忽略标点）
B, B	上一个单词（B 忽略标点）
+	移至下一列的第一个字符处
-	移至上一列的第一个字符处
(	移至该句首
)	移至该句末
{	移至该段首
}	移至该段末
Ng	移至该文件的第 n 列
n+	移至光标所在位置之后第 n 列
n-	移至光标所在位置之前第 n 列

（3）进入插入模式

在命令模式输入以下命令即可进入 vi 插入模式：

a(append)	在光标之后加入资料
A	在该行之末加入资料
i(insert)	在光标之前加入资料
I	在该行之首加入资料
o(open)	新增一行于该行之下，供输入资料用
O	新增一行于该行之上，供输入资料用

（4）命令模式下文本修改命令

dd	删除当前光标所在行

x	删除当前光标字符
X	删除当前光标之前字符
U	撤消
.	重做
f	查找
s	替换,例如:将文件中的所有"FOX"换成"duck", 用":s/FOX/duck/g"
ESC	离开输入模式

（5）插入模式返回命令模式

在插入模式下，按 ESC 键可切换到命令模式。

更多用法见 info vi

三、GNU C 编译器

LINUX 上可用的 C 编译器是 GNU C 编译器，它建立在自由软件基金会编程许可证的基础上，因此可以自由发布。

LINUX 上的 GNU C 编译器（GCC）是一个全功能的 ANCI C 兼容编译器，而一般 UNIX（如 SCO UNIX）用的编译器是 CC。下面介绍 GCC 和一些 GCC 编译器最常用的选项。

1、使用 GCC

格式： gcc [options] [filenames]

filenames 说明要编译和连接的源文件和/或目标文件

options 指定的编译过程中的具体操作

2、GCC 常用选项

GCC 有超过 100 个的编译选项可用，这些选项中的许多可能永远都不会用到，但一些主要的选项将会频繁使用。

当不用任何选项编译一个程序时，GCC 将建立（假定编译成功）一个名为 a.out 的可执行文件。例如，

gcc test.c

编译成功后，在当前目录下产生了一个 a.out 文件。

可用-o 选项为即将产生的可执行文件指定一个文件名来代替 a.out。例如：

gcc –o count count.c

此时得到的可执行文件就不再是 a.out，而是 count。

GCC 可以指定编译器处理步骤多少。-c 选项告诉 GCC 仅把源代码编译为目标代码而不进行汇编和连接步骤。这个选项使用得非常频繁。因为它编译多个 C 程序时速度更快且更易于管理。默认时 GCC 建立的目标代码文件有一个.o 的扩展名。例如：

gcc -c count.c

编译成功后，在当前目录下产生了一个 count.o 文件。

3、文件的执行

格式： ./可执行文件名

例：./a.out

./count

三、gdb 调试工具

LINUX 包含一个叫 gdb 的 GNU 调试程序。gdb 是一个用来调试 C 和 C++程序的强有力调试器。它使你能在程序运行时观察程序的内部结构和内存的使用情况。它具有以下一

些功能：

- 监视程序中变量的值；
- 设置断点以使程序在指定的代码行上停止执行；
- 一行行的执行代码。

以下是利用 gdb 进行调试的步骤：

1、编译时打开调试选项

为了使 gdb 正常工作，必须使你的程序在编译时包含调试信息。调试信息里包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb 利用这些信息使源代码和机器码相关联。

在编译时用 –g 选项打开调试选项编译。

2、gdb 基本命令

命　令	描　　述
file	装入欲调试的可执行文件
kill	终止正在调试的程序
list	列出产生执行文件的源代码部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码并进入函数内部
run	执行当前被调试的程序
quit	终止 gdb
watch	监视一个变量的值而不管它何时被改变
break	在代码里设置断点，使程序执行到这里时被挂起
make	不退出 gdb 就可以重新产生可执行文件
shell	不离开 gdb 就执行 UNIX shell 命令

3、应用举例

(1)编辑源程序 greet.c

(2)编译　gcc -g gdb –o greet greet.c

(3)gdb greet ，出现提示符(gdb)，此时可在提示符下输入 gdb 的命令了，如：

(gdb)run

(gdb)list

(4)退出调试状态，返回系统提示符下， (gdb)quit

四、参考程序

```
main( )
{
 printf("Hello,world!\n");
}
```

实验二　进程的创建

实验目的

1、掌握进程的概念，明确进程的含义

2、认识并了解并发执行的实质

实验内容

1、编写一段程序，使用系统调用 fork( )创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示'a'，子进程分别显示字符'b'和字符'c'。试观察记录屏幕上的显示结果，并分析原因。

2、修改上述程序，每一个进程循环显示一句话。子进程显示'daughter　…'及'son　……'，父进程显示 'parent　……'，观察结果，分析原因。

实验指导

一、 进程

LINUX 作为一个多用户操作系统，支持多道程序设计，支持分时处理和软中断处理，也带有某些微内核的特征。LINUX 进程控制块是 LINUX 最复杂的数据结构之一，内容包括进程调度、信号处理、进程队列指针、进程标识和用户标识、定时控制、信号量处理、上下文切换、文件系统管理、内存管理、进程状态和进程标志等信息，总计有 80 多类属性。LINUX 进程控制块常驻内存。LINUX 进程控制块由 struct task_struct 定义，长 1680 个字节。见附录二。

进程是一个动态的概念。LINUX 的进程状态有：

TASK__RUNNING　　正在运行或在就绪队列中准备运行的进程

TASK__INTERRUPTIBLE　处于等待队列中的进程，待资源有效时唤醒，也可由其它进程通过信号或定时中断唤醒

TASK__UNINTERRUPTIBLE 处于等待队列中的进程，待资源有效时唤醒，不可由其它进程通过信号或定时中断唤醒

TASK__ZOMBIE　　进程结束但尚未消亡释放 PCB，（僵死状态）

TASK__STOPPED　　进程被暂停，通过其它进程信号才能唤醒

各进程状态之间的转换关系见图 1。

二、所涉及的系统调用

1、fork( )

创建一个新进程。

系统调用格式：　　pid=fork( )

参数定义：　　int　fork( )

fork( )返回值意义如下：若子进程创建失败，则调用进程获得返回值-1。若创建成功，在子进程中，fork( )返回值为 0，表示当前进程是子进程；在父进程中，fork( )返回值为子进程的 id 值（>0）。

如果 fork( )调用成功，它向父进程返回子进程的 PID，并向子进程返回 0，即 fork( )调用一次，但在不同进程返回不同的值。此时 OS 在内存中建立一个新进程，所建的新进程是调用 fork( )父进程（parent process）的副本，称为子进程（child process）。子进程继承了父进程的许多特性，并具有与父进程完全相同的用户级上下文。父进程与子进程并发执行。

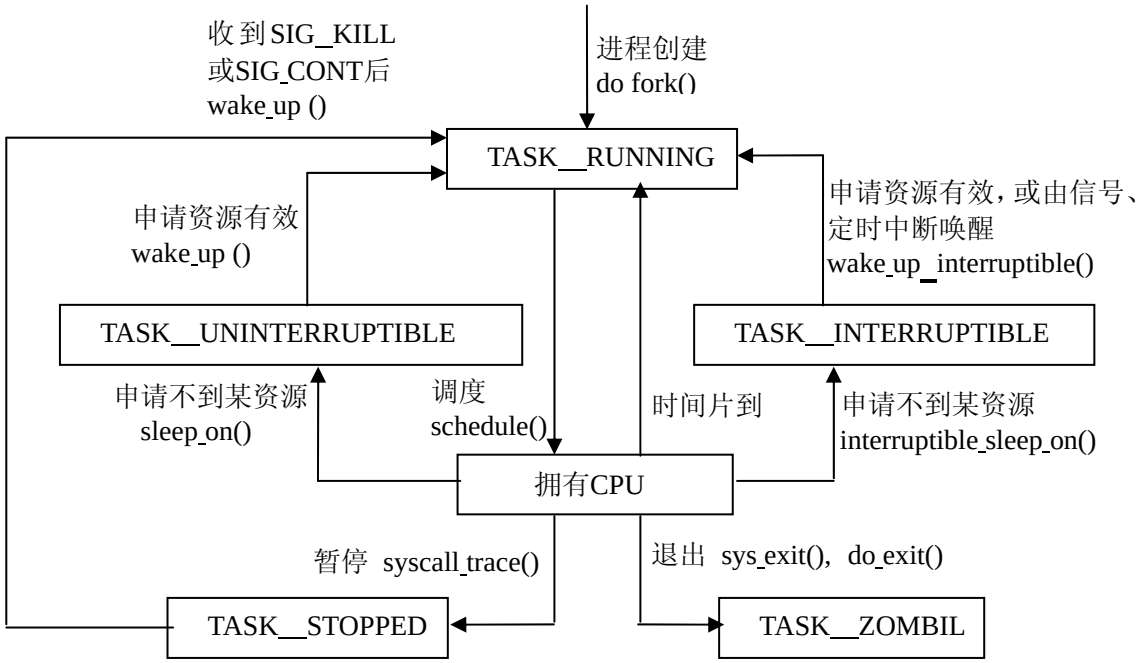


图 1　LINUX 进程状态之间的转换

核心为 fork( )完成以下操作：

（1）为新进程分配一进程表项和进程标识符

进入 fork( )后，核心检查系统是否有足够的资源来建立一个新进程。若资源不足，则 fork( )系统调用失败；否则，核心为新进程分配一进程表项和唯一的进程标识符。

（2）检查同时运行的进程数目

超过预先规定的最大数目时，fork( )系统调用失败。

（3）拷贝进程表项中的数据

将父进程的当前目录和所有已打开的数据拷贝到子进程表项中，并置进程的状态为“创建”状态。

（4）子进程继承父进程的所有文件

对父进程当前目录和所有已打开的文件表项中的引用计数加 1。

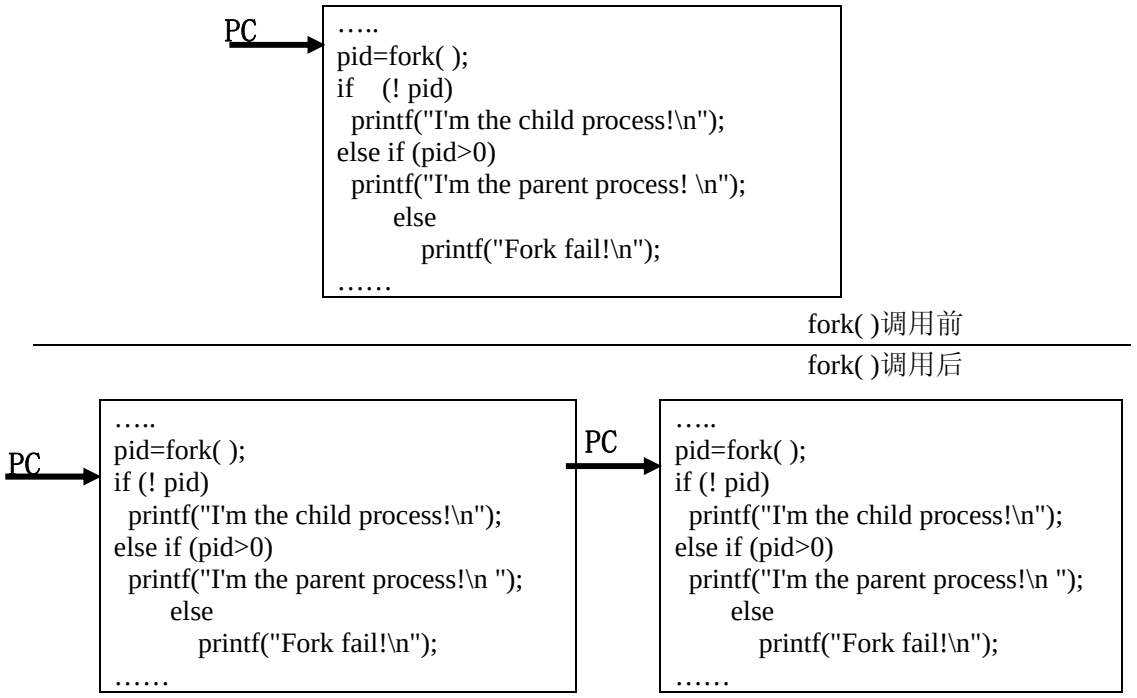
（5）为子进程创建进程上、下文

进程创建结束，设子进程状态为“内存中就绪”并返回子进程的标识符。

（6）子进程执行

虽然父进程与子进程程序完全相同，但每个进程都有自己的程序计数器 **PC(注意子进程的 PC 开始位置)**，然后根据 pid 变量保存的 fork( )返回值的不同，执行了不同的分支语句。

例：



2．getpid( )，getppid( )

getpid( )　　获得当前进程的进程 id

getppid( )　　获得当前父进程的进程 id

调用方式：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

四、参考程序

1、

```
#include <stdio.h>
main( )
{
    int p1,p2;
    while((p1=fork( ))= = -1);          /*创建子进程 p1*/
    if (p1= =0)   putchar('b');
    else
        {
            while((p2=fork( ))= = -1);    /*创建子进程 p2*/
            if(p2= =0)   putchar('c');
            else   putchar('a');
        }
}
```

2、

```
#include <stdio.h>
main( )
```

```
{
    int p1,p2,i;
    while((p1=fork( ))= = -1);                  /*创建子进程 p1*/
    if (p1= =0)
        for(i=0;i<10;i++)
            printf("daughter    %d\n",i);
    else
        {
            while((p2=fork( ))= = -1);      /*创建子进程 p2*/
            if(p2= =0)
                for(i=0;i<10;i++)
                    printf("son    %d\n",i);
            else
                for(i=0;i<10;i++)
                    printf("parent    %d\n",i);
        }
}
```

五、运行结果

1、bca，bac, abc ,……都有可能。

2、 parent…
son…
daughter..
daughter..

或 parent…
son…
parent…
daughter…等

六、分析原因

除 strace 外，也可用 ltrace -f -i -S ./executable-file-name 查看以上程序执行过程。

1、从进程并发执行来看，各种情况都有可能。上面的三个进程没有同步措施，所以父进程与子进程的输出内容会叠加在一起。输出次序带有随机性。Fork()创建进程所需的时间多于输入一个字符的时间，因此在主进程创建进程 2 的同时，进程 1 就输出了 b，而进程 2 和主程序的输出次序是随机性的。

2、由于函数 printf( )在输出字符串时不会被中断，因此，字符串内部字符顺序输出不变。但由于进程并发执行的调度顺序和父子进程抢占处理机问题，输出字符串的顺序和先后随着执行的不同而发生变化。这与打印单字符的结果相同。

补充：进程树

在 UNIX 系统中，只有 0 进程是在系统引导时被创建的，在系统初启时由 0 进程创建 1 进程，以后 0 进程变成对换进程，1 进程成为系统中的始祖进程。UNIX 利用 fork( )为每个终端创建一个子进程为用户服务，如等待用户登录、执行 SHELL 命令解释程序等，每个终端进程又可利用 fork( )来创建其子进程，从而形成一棵进程树。可以说，系统中除 0 进程外的所有进程都是用 fork( )创建的。

七、思考题

（1）系统是怎样创建进程的？

（2）当首次调用新创建进程时，其入口在哪里？

实验三　进程的控制

实验目的

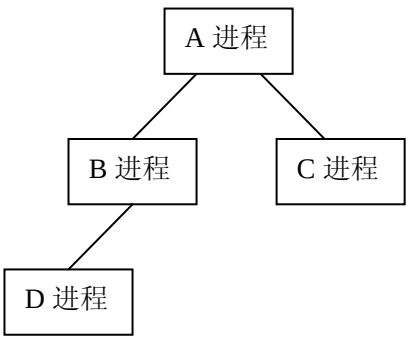
1、掌握进程另外的创建方法

2、熟悉进程的睡眠、同步、撤消等进程控制方法

实验内容

1、用 fork( )创建一个进程，再调用 exec( )用新的程序替换该子进程的内容

2、用 fork( )建立如下形式的进程树：



各个进程中都打印出本身 PID 和其父进程的 PID，并用 wait( )来控制进程执行顺序，打印出正确和期望的结果。

实验指导

一、所涉及的系统调用

在 UNIX/LINUX 中 fork( )是一个非常有用的系统调用，但在 UNIX/LINUX 中建立进程除了 fork( )之外，也可用与 fork( ) 配合使用的 exec( )。

1、exec( )系列系统调用

系统调用 exec( )系列，也可用于新程序的运行。fork( )只是将父进程的用户级上下文拷贝到新进程中，而 exec( )系列可以将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上，以更改新进程的用户级上下文。exec( )系列中的系统调用都完成相同的功能，它们把一个新程序装入内存，来改变调用进程的执行代码，从而形成新进程。如果 exec( )调用成功，调用进程将被**覆盖**，然后从新程序的入口开始执行，这样就产生了一个新进程，新进程的进程标识符 id 与调用进程相同。

exec( )没有建立一个与调用进程并发的子进程，而是用新进程取代了原来进程。所以 exec( )调用成功后，没有任何数据返回，这与 fork( )不同。exec( )系列系统调用在系统库 unistd.h 中，共有 execl、execlp、execle、execv、execvp 五个，其基本功能相同，只是以不同的方式来给出参数。

一种是直接给出参数的指针，如：

```
int    execl(path,arg0[,arg1,...argn],0);
char    *path,*arg0,*arg1,...,*argn;
```

另一种是给出指向参数表的指针，如：

```
int execv(path,argv);
char *path,*argv[ ];
```

具体使用可参考有关书。

2、exec( )和 fork( )联合使用

系统调用 exec 和 fork( )联合使用能为程序开发提供有力支持。用 fork( )建立子进程，然后在子进程中使用 exec( )，这样就实现了父进程与一个与它完全不同子进程的并发执行。

一般，wait、exec 联合使用的模型为：

```
int status;
   .............
if (fork( )= =0)
   {
        ...........;
        execl(...);
        ...........;
   }
wait(&status);
```

3、wait（）

等待子进程运行结束。如果子进程没有完成，父进程一直等待。wait( )将调用进程挂起，直至其子进程因暂停或终止而发来软中断信号为止。如果在 wait( )前已有子进程暂停或终止，则调用进程做适当处理后便返回。

系统调用格式：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t    wait(int *status)
```

其中，status 是用户空间的地址。它的低 8 位反映子进程退出状态，为 0 表示子进程正常结束，非 0 则表示出现了各种各样的问题；高 8 位则带回了 exit( )的返回值。exit( )返回值由系统给出。wait 的返回值为结束子进程的 PID。

核心对 wait( )作以下处理：

（1）首先查找调用进程是否有子进程，若无，则返回出错码；

（2）若找到一处于“僵死状态”的子进程，则将子进程的执行时间加到父进程的执行时间上，并释放子进程的进程表项；

（3）若未找到处于“僵死状态”的子进程，则调用进程便在可被中断的优先级上睡眠，等待其子进程发来软中断信号时被唤醒。

4、exit（）

终止进程的执行。

系统调用格式：

```
        void exit(int status)
```

其中，status 是返回给父进程的一个整数，以备查考。

为了及时回收进程所占用的资源并减少父进程的干预，UNIX/LINUX 利用 exit( )来实现进程的自我终止，通常父进程在创建子进程时，应在进程的末尾安排一条 exit( )，使子进程自我终止。exit(0)表示进程正常终止，exit(1)表示进程运行有错，异常终止。

如果调用进程在执行 exit( )时，其父进程正在等待它的终止，则父进程可立即得到其返回的整数。核心须为 exit( )完成以下操作：

（1）关闭软中断

（2）回收资源

（3）写记帐信息

（4）置进程为“僵死状态”

二、参考程序

```
#include<stdio.h>
#include<unistd.h>
main( )
```

```
{
    int pid;
    pid=fork( );              /*创建子进程*/
    switch(pid)
        {
            case  -1:                                   /*创建失败*/
                printf("fork fail!\n");
                exit(1);
            case  0:                                      /*子进程*/
                execl("/bin/ls","ls","-1","-color",NULL);
                printf("exec fail!\n");
                exit(1);
            default:                                      /*父进程*/
                wait(NULL);                             /*同步*/
                printf("ls completed !\n");
                exit(0);
        }
}
```

三、运行结果

执行命令 ls -l -color ，（按倒序）列出当前目录下所有文件和子目录；

ls completed!

四、分析原因

程序在调用 fork( )建立一个子进程后，马上调用 wait( )，使父进程在子进程结束之前，一直处于睡眠状态。子进程用 exec( )装入命令 ls ，exec( )后，子进程的代码被 ls 的代码取代，这时子进程的 PC 指向 ls 的第 1 条语句，开始执行 ls 的命令代码。

注意在这里 wait( )给我们提供了一种实现进程同步的简单方法。

五、思考

（1）可执行文件加载时进行了哪些处理？

（2）什么是进程同步？wait( )是如何实现进程同步的？

（3）为什么有时打印出父进程的 PID 不正确，是 1？

实验四　进程的互斥

实验目的

1、进一步认识并发执行的实质

2、分析进程竞争资源的现象，学习解决进程互斥的方法

实验内容

1、修改实验二中的程序 2，用 lockf( )来给每一个进程加锁，以实现进程之间的互斥

2、观察并分析出现的现象

实验指导

一、所涉及的系统调用

lockf(files,function,size)

用作锁定文件的某些段或者整个文件。

本函数的头文件为

#include "unistd.h"

参数定义：

int　lockf(files,function,size)
int　files,function;
long　size;

其中：files 是文件描述符；function 是锁定和解锁：1 表示锁定，0 表示解锁。size 是锁定或解锁的字节数，若为 0，表示从文件的当前位置到文件尾。

二、参考程序

```
#include <stdio.h>
#include <unistd.h>
main(　)
{　int p1,p2,i;
　while((p1=fork( ))= = -1);　　/*创建子进程 p1*/
　if (p1= =0)
　{　lockf(1,1,0);　/*加锁，第一个参数为 stdout（标准输出设备的描述符）*/
　　for(i=0;i<10;i++)　printf("daughter %d\n",i);
　　lockf(1,0,0);　　/*解锁*/
　}
　else
　{　while((p2=fork( ))= =-1);　/*创建子进程 p2*/
　　if (p2= =0)
　　{　lockf(1,1,0);　/*加锁*/
　　　for(i=0;i<10;i++)　printf("son %d\n",i);
　　　lockf(1,0,0);　/*解锁*/
　　}
　　else
　　{　lockf(1,1,0);　/*加锁*/
　　　for(i=0;i<10;i++)　printf(" parent %d\n",i);
　　　lockf(1,0,0);　/*解锁*/
　　}
　}
```

```
    }
```

三、运行结果

```
      parent0
      parent1
       ┇
      parent9
      son0
      son1
       ┇
      son9
      daughter0
      daughter1
       ┇
      daughter9
或   daughter0
      daughter1
       ┇
      daughter9
      parent0
      parent1
       ┇
      parent9
      son0
      son1
       ┇
      son9
```

随着执行时间不同，输出结果的顺序有所不同。但总是一个进程的结果输出完后再输出另一个进程的结果。

四、分析原因

上述程序执行时，不同进程之间存在共享临界资源，所以加锁与不加锁效果不同。

五、分析以下程序的输出结果：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main()
{
int p1,p2,i;
char outstr[20];
int fd;
fd = creat("to_be_locked.txt",0777);    /* 调用系统服务creat，建立共享文件
                                           to_be_locked.txt，返回文件描述符 fd */
if (fd==0)                              /* 建立失败，则终止退出 */
  { printf("Fail to create file!");
```

```
    exit(-1);
  }
while((p1=fork())==-1);                  /* 创建子进程p1 */
if (p1==0)
  { lockf(fd,1,0);                       /* 加锁 */
    for(i=0; i<5; i++) { sprintf(outstr,"daughter %d\n",i); /* 形成输出串 */
                         write(fd,outstr,11);              /* 写入共享文件 */
                       }
    lockf(fd,0,0);                       /* 解锁 */
  }
else
  {
    while((p2=fork())==-1);
    if (p2==0)
      { lockf(fd,1,0);                   /*加锁*/
        for(i=0; i<5; i++) {  sprintf(outstr,"son %d\n",i);
                              write(fd,outstr,6);
                           }
        lockf(fd,0,0);                   /*解锁*/
      }
    else
      { lockf(fd,1,0);                   /*加锁*/
        for(i=0; i<5; i++) { sprintf(outstr,"parent %d\n",i);
                             write(fd,outstr,9);
                           }
        lockf(fd,0,0);                   /*解锁*/
      }
  }
close(fd);
}
```

用命令 cat to_be_locked.txt 可查看输出结果。比较加锁与不加锁的效果。

注：系统调用 lockf 在 WIN 仿真系统中不起作用，本实验的两个程序必须在 LINUX 系统中运行才能得到正确的结果。

实验五　进程通信——信号机制和管道

UNIX/LINUX 系统的进程间通信机构（IPC）允许在任意进程间大批量地交换数据。本实验的目的是了解和熟悉 LINUX 支持的信号量机制、管道机制、消息通信机制及共享存储区机制。

(一) 信号机制实验

实验目的

1、了解什么是信号

2、熟悉 LINUX 系统中进程之间软中断通信的基本原理

实验内容

1、编写程序：用 fork( )创建两个子进程，再用系统调用 signal( )让父进程捕捉键盘上来的中断信号（即按^c 键）；捕捉到中断信号后，父进程用系统调用 kill( )向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child process1 is killed by parent!

Child process2 is killed by parent!

父进程等待两个子进程终止后，输出如下的信息后终止：

Parent process is killed!

2、分析利用软中断通信实现进程同步的机理

实验指导

一、信号

1、信号的基本概念

每个信号都对应一个正整数常量(称为 signal　number,即信号编号。定义在系统头文件<signal.h>中)，代表同一用户的诸进程之间传送事先约定的信息的类型，用于通知某进程发生了某异常事件。每个进程在运行时，都要通过信号机制来检查是否有信号到达。**若有，便中断正在执行的程序，转向与该信号相对应的处理程序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。**实质上，信号机制是对中断机制的一种模拟，故在早期的 UNIX 版本中又把它称为软中断。

信号与中断的相似点：

（1）采用了相同的异步通信方式；

（2）当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；

（3）都在处理完毕后返回到原来的断点；

（4）对信号或中断都可进行屏蔽。

信号与中断的区别：

（1）中断有优先级，而信号没有优先级，所有的信号都是平等的；

（2）信号处理程序是在用户态下运行的，而中断处理程序是在核心态下运行；

（3）中断响应是及时的，而信号响应通常都有较大的时间延迟。

信号机制具有以下三方面的功能：

（1）发送信号。发送信号的程序用系统调用 kill( )实现；

（2）预置对信号的处理方式。接收信号的程序用 signal( )来实现对处理方式的预置；

（3）收受信号的进程按事先的规定完成对相应事件的处理。

2、信号的发送

信号的发送，是指由发送进程把信号送到指定进程的信号域的某一位上。如果目标进程正在一个可被中断的优先级上睡眠，核心便将它唤醒，发送进程就此结束。一个进程可能在其信号域中有多个位被置位，代表有多种类型的信号到达，但对于一类信号，进程却只能记住其中的某一个。

进程用 kill( )向一个进程或一组进程发送一个信号。

3、对信号的处理

当一个进程要进入或退出一个低优先级睡眠状态时，或一个进程即将从核心态返回用户态时，核心都要检查该进程是否已收到软中断。当进程处于核心态时，即使收到软中断也不予理睬；只有当它返回到用户态后，才处理软中断信号。对软中断信号的处理分三种情况进行：

（1）如果进程收到的软中断是一个已决定要忽略的信号（function=1），进程不做任何处理便立即返回；

（2）进程收到软中断后便退出（function=0）；

（3）执行用户设置的软中断处理程序。

二、所涉及的中断调用

1、kill( )

系统调用格式： int kill(int pid,int sig)

其中，pid 是一个或一组进程的标识符，参数 sig 是要发送的软中断信号。

（1）pid>0 时，核心将信号发送给进程 pid。

（2）pid=0 时，核心将信号发送给与发送进程同组的所有进程。

（3）pid=-1 时，核心将信号发送给所有用户标识符真正等于发送进程的有效用户标识号的进程。

2、signal( )

预置对信号的处理方式，允许调用进程控制软中断信号。（预置是什么意思？）

系统调用格式： #include <signal.h>

int signal(int sig,void *function)

其中 sig 用于指定信号的类型，sig 为 0 则表示没有收到任何信号，余者如下表：

值	符号名字	说　　明
01	SIGHUP	挂起（hangup）
02	SIGINT	中断，当用户从键盘按^c 键或^break 键时
03	SIGQUIT	退出，当用户从键盘按 quit 键时
04	SIGILL	非法指令
05	SIGTRAP	跟踪陷阱（trace trap），启动进程，跟踪代码的执行
06	SIGIOT	IOT 指令
07	SIGEMT	EMT 指令
08	SIGFPE	浮点运算溢出
09	SIGKILL	杀死、终止进程
10	SIGBUS	总线错误
11	SIGSEGV	段违例（segmentation violation），进程试图去访问其虚地址空间以外的位置
12	SIGSYS	系统调用中参数错，如系统调用号非法
13	SIGPIPE	向某个非读管道中写入数据
14	SIGALRM	闹钟。当某进程希望在某时间后接收信号时发此信号
15	SIGTERM	软件终止（software termination）

16	SIGUSR1	用户自定义信号 1
17	SIGUSR2	用户自定义信号 2
18	SIGCLD	某个子进程死
19	SIGPWR	电源故障

function：在该进程中的一个函数入口地址，在核心返回用户态时，它以软中断信号的序号作为参数调用该函数，对除了信号 SIGKILL，SIGTRAP 和 SIGPWR 以外的信号，核心自动地重新设置软中断信号处理程序的值为 SIG_DFL，一个进程不能捕获 SIGKILL 信号。

function 的解释如下：

（1）function=1 时，进程对 sig 类信号不予理睬，亦即屏蔽了该类信号；

（2）function=0 时，缺省值，进程在收到 sig 信号后应终止自己；

（3）function 为非 0，非 1 类整数时，function 的值即作为信号处理程序的指针。

三、参考程序

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

void waiting( ),stop( );
int wait_mark;
main( )
  {   int p1,p2,stdout;
      while((p1=fork( ))= =-1);            /*创建子进程 p1*/
      if (p1>0)
      {   while((p2=fork( ))= =-1);        /*创建子进程 p2*/
          if(p2>0)
          {   wait_mark=1;
              signal(SIGINT,stop);       /*接收到^c 信号，转 stop*/
              waiting( );                  //如无按键将等待
              kill(p1,16);               /*向 p1 发软中断信号 16*/
              kill(p2,17);               /*向 p2 发软中断信号 17*/
              wait(0);                   /*同步*/等待子进程结束
              wait(0);
              printf("Parent process is killed!\n");
              exit(0);//退出进程
          }
          else
          {   wait_mark=1;
              signal(SIGINT, SIG_IGN); /*设置信号函数处理方式，当收到键盘发送键盘中断（如break键被按下）信号时将会忽略此信号。*/
              signal(17,stop);     /*接收到软中断信号 17，转 stop*/接受父进程发送信号
              waiting( );
              lockf(stdout,1,0);
              printf("Child process 2 is killed by parent!\n");
              lockf(stdout,0,0);
              exit(0);
```

```
            }
        }
        else
        {   wait_mark=1;
            signal(16,stop);            /*接收到软中断信号 16，转 stop*/
            waiting( );
            lockf(stdout,1,0);
            printf("Child process 1 is killed by parent!\n");
            lockf(stdout,0,0);
            exit(0);
        }
    }

void waiting( )
{
    while(wait_mark!=0);//通过循环使子进程停止
}

void stop( )
  {
    wait_mark=0;//使子进程继续运行
  }
```

四、运行结果

屏幕上无反应，按下^C 后，显示 Parent process is killed!

五、分析原因

上述程序中，signal( )都放在一段程序的前面部位，而不是在其他接收信号处。这是因为 signal( )的执行只是为进程指定信号值 16 或 17 的作用，以及分配相应的与 stop( )过程链接的指针。因而，signal( )函数必须在程序前面部分执行。

本方法通信效率低，当通信数据量较大时一般不用此法。

当你 Ctrl-c 的时候，系统会给父进程及其两个子进程都发送 SIGINT 信号(对 bash 来说，这三个进程都是前台进程，所以都发送)，对于父进程来说，收到这个信号自然是调用 stop 函数了，但是对于两个子进程来说，默认的对这个信号的处理就是退出(exit)，所以你看不到子进程的打印。

进程接收到信号以后，可以有如下 3 种选择进行处理：

- 接收默认处理：接收默认处理的进程通常会导致进程本身消亡。例如连接到终端的进程，用户按下 CTRL+c，将导致内核向进程发送一个 SIGINT 的信号，进程如果不对该信号做特殊的处理，系统将采用默认的方式处理该信号，即终止进程的执行；
- 忽略信号：进程可以通过代码，显示地忽略某个信号的处理，例如：signal(SIGINT,SIGDEF)；但是某些信号是不能被忽略的，
- 捕捉信号并处理：进程可以事先注册信号处理函数，当接收到信号时，由信号处理函数自动捕捉并且处理信号。

六、思考

1、该程序段前面部分用了两个 wait(0)，它们起什么作用？

2、该程序段中每个进程退出时都用了语句 exit(0)，为什么？

3、为何预期的结果并未显示出？

4、程序该如何修改才能得到正确结果？

5、不修改程序如何得到期望的输出？

在不修改程序的情况下要输出期望的结果，可以单独向父进程发送 SIGINT 信号，这样即可避免子进程由于收到 SIGINT 信号执行默认操作而自我终止，具体实现方法：

（该实验中程序名为：demo2
）
首先让程序在后台运行，命令：./demo2&
执行该命令后，会在后台生成 3 个进程，使用 ps 命令可以查看到它们的 PID （相对小的 PID 应该为父进程的
PID，原因是创建时间相对早）。
然后向后台的父进程发送 SIGINT 信号，命令：kill －SIGINT 28664 (其 中 28664 为父进程的
PID)

（二）进程的管道通信实验

实验目的

1、了解什么是管道
2、熟悉 UNIX/LINUX 支持的管道通信方式

实验内容

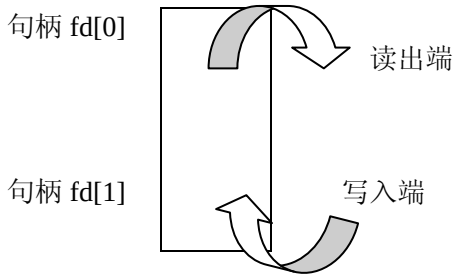
编写程序实现进程的管道通信。用系统调用 pipe( )建立一管道，二个子进程 P1 和 P2 分别向管道各写一句话：
Child 1 is sending a message!
Child 2 is sending a message!
父进程从管道中读出二个来自子进程的信息并显示（要求先接收 P1，后 P2）。

实验指导

一、什么是管道

UNIX 系统在 OS 的发展上，最重要的贡献之一便是该系统首创了管道（pipe）。这也是 UNIX 系统的一大特色。

所谓管道，是指能够连接一个写进程和一个读进程的、并允许它们以生产者—消费者方式进行通信的一个共享文件，又称为 pipe 文件。由写进程从管道的写入端（句柄 1）将数据写入管道，而读进程则从管道的读出端（句柄 0）读出数据。



二、管道的类型

1、有名管道

一个可以在文件系统中长期存在的、具有路径名的文件。用系统调用 mknod( )建立。它克服无名管道使用上的局限性，可让更多的进程也能利用管道进行通信。因而其它进程可以知道它的存在，并能利用路径名来访问该文件。对有名管道的访问方式与访问其他文件一样，需先用 open( )打开。

2、无名管道

一个临时文件。利用 pipe( )建立起来的无名文件（无路径名）。只用该系统调用所返

回的文件描述符来标识该文件，故只有调用 pipe( )的进程及其子孙进程才能识别此文件描述符，才能利用该文件（管道）进行通信。当这些进程不再使用此管道时，核心收回其索引结点。

二种管道的读写方式是相同的，本文只讲无名管道。

3、pipe 文件的建立

分配磁盘和内存索引结点、为读进程分配文件表项、为写进程分配文件表项、分配用户文件描述符

4、读/写进程互斥

内核为地址设置一个读指针和一个写指针，按先进先出顺序读、写。

为使读、写进程互斥地访问 pipe 文件，需使各进程互斥地访问 pipe 文件索引结点中的直接地址项。因此，每次进程在访问 pipe 文件前，都需检查该索引文件是否已被上锁。若是，进程便睡眠等待，否则，将其上锁，进行读/写。操作结束后解锁，并唤醒因该索引结点上锁而睡眠的进程。

三、所涉及的系统调用

1、pipe( )

建立一无名管道。

系统调用格式：　# include<unistd.h>
　　int pipe(int filedes[2])

其中，filedes[1]是写入端文件描述符，filedes[0]是读出端文件描述符。

2、read( )

系统调用格式：　#include<stdio.h>
　　int　read(int fd, char *buf, unsigned nbyte)

功能：从 fd 所指示的文件中读出 nbyte 个字节的数据，并将它们送至由指针 buf 所指示的缓冲区中。如该文件被加锁，等待，直到锁打开为止。

3、write( )

系统调用格式：　#include<stdio.h>
　　int　write(int fd, char *buf, unsigned nbyte)

功能：把 nbyte 个字节的数据，从 buf 所指向的缓冲区写到由 fd 所指向的文件中。如文件加锁，暂停写入，直至开锁。

四、参考程序

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
int pid1,pid2;
main( )
{    int fd[2];
     char outpipe[100],inpipe[100];
     pipe(fd);                              /*创建一个管道*/
     while ((pid1=fork( ))= =-1);
     if(pid1= =0)
          {    lockf(fd[1],1,0);
               sprintf(outpipe,"child 1 process is sending message!");
                                              /*把串放入数组 outpipe 中*/
               write(fd[1],outpipe,50);       /*向管道写长为 50 字节的串*/
               sleep(5);                      /*自我阻塞 5 秒*/
               lockf(fd[1],0,0);
```

```
                exit(0);
            }
        else
            {   while((pid2=fork( ))= =-1);
                if(pid2= =0)
                  {   lockf(fd[1],1,0);                    /*互斥*/
                      sprintf(outpipe,"child 2 process is sending message!");
                      write(fd[1],outpipe,50);
                      sleep(5);
                      lockf(fd[1],0,0);
                      exit(0);
                  }
                else
                  {   wait(0);                             /*同步*/
                      read(fd[0],inpipe,50);               /*从管道中读长为 50 字节的串*/
                      printf("%s\n",inpipe);
                      wait(0);
                      read(fd[0],inpipe,50);
                      printf("%s\n",inpipe);
                      exit(0);
                  }
            }
    }
```

五、运行结果

延迟 5 秒后显示

child 1 process is sending message!

再延迟 5 秒

child 2 process is sending message!

六、思考题

1、程序中的 sleep(5)起什么作用？

2、子进程 1 和 2 为什么也能对管道进行操作？

实验六　进程通信——消息机制

实验目的

1、了解什么是消息

2、熟悉消息传送的机理

实验内容

消息的创建、发送和接收。使用系统调用 msgget( ),msgsnd( ),msgrev( ),及 msgctl( )编制一长度为 1 k 的消息发送和接收的程序。

实验指导

一、什么是消息

消息（message）是一个格式化的可变长的信息单元。消息机制允许由一个进程给其它任意的进程发送一个消息。当一个进程收到多个消息时，可将它们排成一个消息队列。消息使用二种重要的数据结构：一是消息首部，其中记录了一些与消息有关的信息，如消息数据的字节数；二是消息队列头表，其每一表项是作为一个消息队列的消息头，记录了消息队列的有关信息。

1、消息机制的数据结构

（1）消息首部

记录一些与消息有关的信息，如消息的类型、大小、指向消息数据区的指针、消息队列的链接指针等。

（2）消息队列头表

其每一项作为一个消息队列的消息头，记录了消息队列的有关信息如指向消息队列中第一个消息和指向最后一个消息的指针、队列中消息的数目、队列中消息数据的总字节数、队列所允许消息数据的最大字节总数，还有最近一次执行发送操作的进程标识符和时间、最近一次执行接收操作的进程标识符和时间等。

2、消息队列描述符

UNIX 中，每一个消息队列都有一个称为关键字（key）的名字，是由用户指定的；消息队列有一消息队列描述符，其作用与用户文件描述符一样，也是为了方便用户和系统对消息队列的访问。

二、涉及的系统调用

使用与消息机制有关的系统调用，都需包含以下头文件：

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

1. msgget( )

创建一个消息队列，获得一个消息队列描述符。核心将搜索消息队列头表，确定是否有指定名字的消息队列。若无，核心将分配一新的消息队列头，并对它进行初始化，然后给用户返回一个消息队列描述符，否则它只是检查消息队列的许可权便返回。

系统调用格式：msgqid=msgget(key,flag)

参数定义：

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

其中：

key 是用户指定的消息队列的名字；

flag 是用户设置的标志（控制命令）和访问方式，如：

IPC_CREAT |0400　　该队列是否已被创建，无则创建，是则打开；

IPC_EXCL |0400　　该队列的创建是互斥的，若已创建则失败；

msgqid 是该系统调用返回的描述符，失败则返回-1。

2. msgsnd（）

发送消息。向指定的消息队列发送一个消息，并将该消息链接到该消息队列的尾部。

系统调用格式：　msgsnd(msgqid,msgp,size,flag)

参数定义：　int msgsnd(msgqid,msgp,size,flag)

int msgqid,size,flag;

struct msgbuf　* msgp;

其中 msgqid 是 **msgget** 返回的消息队列描述符；msgp 是指向用户消息缓冲区的一个结构体指针。结构体中包含消息类型和消息正文，即

```
struct msgbuf
{   long    mtype;                  /*消息类型*/
    char    mtext[ ];               /*消息的文本*/
}
```

size 指示由 msgp 指向的数据结构中字符数组的长度；即消息的长度。这个数组的最大值由 MSG-MAX( )系统可调用参数来确定。

flag 规定当核心用尽内部缓冲空间时应执行的动作:进程是等待，还是立即返回。若在标志 flag 中未设置 IPC_NOWAIT 位，则当该消息队列中的字节数超过最大值时，或系统范围的消息数超过某一最大值时，调用 msgsnd 进程睡眠。若是设置 IPC_NOWAIT，则在此情况下，msgsnd 立即返回。

对于 msgsnd( )，核心须完成以下工作：

（1）对消息队列的描述符和许可权及消息长度等进行检查。若合法才继续执行，否则返回；

（2）核心为消息分配消息数据区。将用户消息缓冲区中的消息正文，拷贝到消息数据区；

（3）分配消息首部，并将它链入消息队列的末尾。在消息首部中须填写消息类型、消息大小和指向消息数据区的指针等数据；

（4）修改消息队列头中的数据，如队列中的消息数、字节总数等。最后，唤醒等待消息的进程。

3. msgrcv( )

接受一消息。从指定的消息队列中接收指定类型的消息。

系统调用格式：　msgrcv(msgqid,msgp,size,type,flag)

参数定义：　int　msgrcv(msgqid,msgp,size,type,flag)

int　msgqid,size,flag;

struct　msgbuf　*msgp;

long　type;

其中，msgqid,msgp,size,flag 与 msgsnd 中的对应参数相似；type 是规定要读的消息类型。flag 规定倘若该队列无消息，核心应做的操作。如此时设置了 IPC_NOWAIT 标志，则立即返回，若在 flag 中设置了 MS_NOERROR，且所接收的消息大于 size，则核心截断所接收的消息。

对于 msgrcv 系统调用，核心须完成下述工作：

（1）对消息队列的描述符和许可权等进行检查。若合法，就往下执行；否则返回；

（2）根据 type 的不同分成三种情况处理：

type=0，接收该队列的第一个消息，并将它返回给调用者；

type 为正整数，接收类型 type 的第一个消息；

type 为负整数，接收小于等于 type 绝对值的最低类型的第一个消息。

（3）当所返回消息大小等于或小于用户的请求时，核心便将消息正文拷贝到用户区，并从消息队列中删除此消息，然后唤醒睡眠的发送进程。但如果消息长度比用户要求的大时，则做出错返回。

4. msgctl( )

消息队列的操纵。读取消息队列的状态信息并进行修改，如查询消息队列描述符、修改它的许可权及删除该队列等。

系统调用格式： msgctl(msgqid,cmd,buf);

参数定义： int msgctl(msgqid,cmd,buf);
int msgqid,cmd;
struct msgqid_ds *buf;

其中，函数调用成功时返回 0，不成功则返回-1。buf 是用户缓冲区地址，供用户存放控制参数和查询结果；cmd 是规定的命令。命令可分三类：

（1）IPC_STAT 查询有关消息队列情况的命令。如查询队列中的消息数目、队列中的最大字节数、最后一个发送消息的进程标识符、发送时间等，结果在 buf 中；

（2）IPC_SET 按 buf 指向的结构中的值，设置和改变有关消息队列属性的命令。如改变消息队列的用户标识符、消息队列的许可权等；

（3）IPC_RMID 从消息队列头表中删除该消息队列。

msgqid_ds 结构定义如下：

```
struct  msgqid_ds
    {   struct  ipc_perm    msg_perm;       /*许可权结构*/
        short   pad1[7];                    /*由系统使用*/
        ushort msg_qnum;                    /*队列上消息数*/
        ushort msg_qbytes;                  /*队列上最大字节数*/
        ushort msg_lspid;                   /*最后发送消息的 PID*/
        ushort msg_lrpid;                   /*最后接收消息的 PID*/
        time_t msg_stime;                   /*最后发送消息的时间*/
        time_t msg_rtime;                   /*最后接收消息的时间*/
        time_t msg_ctime;                   /*最后更改时间*/
       };
struct  ipc_perm
    {   ushort uid;                         /*当前用户*/
        ushort gid;                         /*当前进程组*/
        ushort cuid;                        /*创建用户*/
        ushort cgid;                        /*创建进程组*/
        ushort mode;                        /*存取许可权*/
        { short pid1; long pad2;}           /*由系统使用*/
     }
```

三、参考程序

1、client.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define MSGKEY 75
struct  msgform
  {   long    mtype;
```

```
        char    mtext[1000];
    }msg;
int    msgqid;

void client()
{ int i;
    msgqid=msgget(MSGKEY,0777);      /*打开 75#消息队列*/
    for(i=10;i>=1;i--)
        {      msg.mtype=i;
               printf("(client)sent\n");
               msgsnd(msgqid,&msg,1024,0);          /*发送消息*/
        }
    exit(0);
}

main( )
{
    client( );
}
```

2、server.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define MSGKEY 75
struct    msgform
    {    long    mtype;
         char    mtext[1000];
    }msg;
int    msgqid;

void server( )
{
      msgqid=msgget(MSGKEY,0777|IPC_CREAT);    /*创建 75#消息队列*/
      do {        msgrcv(msgqid,&msg,1030,0,0);      /*接收消息*/
                  printf("(server)received\n");
          } while(msg.mtype!=1);
      msgctl(msgqid,IPC_RMID,0);    /*删除消息队列，归还资源*/
      exit(0);
}

main( )
    {
         server( );
    }
```

四、程序说明

1、为了便于操作和观察结果，编制二个程序 client.c 和 server.c，分别用于消息的发送与接收。

2、server 建立一个 key 为 75 的消息队列，等待其它进程发来的消息。当遇到类型为 1 的消息，则作为结束信号，取消该队列，并退出 server。server 每接收到一个消息后显示

一句“(server)received。”

3、client 使用 key 为 75 的消息队列，先后发送类型从 10 到 1 的消息，然后退出。最后一个消息，即是 server 端需要的结束信号。client 每发送一条消息后显示一句“(client)sent”。

4、注意： 二个程序分别编辑、编译，生成的执行程序为 client 与 server。执行：

./server&
ipcs -q
./client。

五、运行结果

从理想的结果来说，应当是每当 client 发送一个消息后，server 接收该消息，client 再发送下一条。也就是说“(client)sent”和 “(server)received”的字样应该在屏幕上交替出现。实际的结果大多是，先由 client 发送了两条消息，然后 server 接收一条消息。此后 client 、server 交替发送和接收消息。最后 server 一次接收两条消息。client 和 server 分别发送和接收了 10 条消息，与预期设想一致。

六、思考

message 的传送和控制并不保证完全同步，当一个程序不在激活状态的时候，它完全可能继续睡眠，造成了上面的现象，在多次 send message 后才 recieve message。这一点有助于理解消息传送的实现机理。

```
main( ) {
  int p,q;
 p=fork();
 if(!p)       client();
 else if(p>0)
{
    q=fork();
    if(!q)   server();
else
{wait(0); wait(0);
 }
}
 }
```

实验七　进程通信——共享存储区和信号量
(一) 共享存储区通信

实验目的

了解和熟悉共享存储机制

实验内容

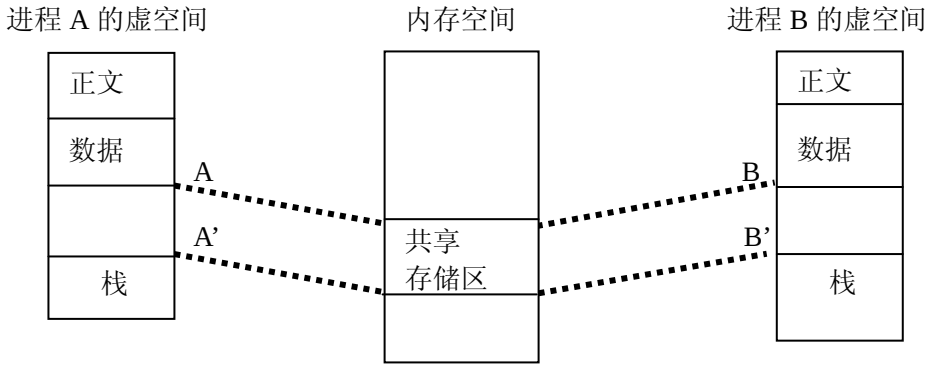
编制一长度为 1k 的共享存储区发送和接收的程序。

实验指导

一、共享存储区

1、共享存储区机制的概念

共享存储区（Share　Memory）是 UNIX 系统中通信速度最高的一种通信机制。该机制可使若干进程共享主存中的某一个区域，且使该区域出现（映射）在多个进程的虚地址空间中。另一方面，一个进程的虚地址空间中又可连接多个共享存储区，每个共享存储区都有自己的名字。当进程间欲利用共享存储区进行通信时，必须先在主存中建立一共享存储区，然后将它附接到自己的虚地址空间上。此后，进程对该区的访问操作，与对其虚地址空间的其它部分的操作完全相同。进程之间便可通过对共享存储区中数据的读、写来进行直接通信。图示列出二个进程通过共享一个共享存储区来进行通信的例子。其中，进程 A 将建立的共享存储区附接到自己的 AA'区域，进程 B 将它附接到自己的 BB'区域。



应当指出，共享存储区机制只为进程提供了用于实现通信的共享存储区和对共享存储区进行操作的手段，然而并未提供对该区进行互斥访问及进程同步的措施。因而当用户需要使用该机制时，必须自己设置同步和互斥措施才能保证实现正确的通信。

二、涉及的系统调用

使用与共享存储区机制有关的系统调用，都需包含以下头文件：

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

1、shmget( )

创建、获得一个共享存储区。

系统调用格式：　　shmid=shmget(key,size,flag)

参数定义：　　int　shmget(key,size,flag);
　　key_t　key;
　　int　size,flag;

其中，key 是共享存储区的名字；size 是其大小（以字节计）；flag 是用户设置的标志，如 IPC_CREAT。IPC_CREAT 表示若系统中尚无指名的共享存储区，则由核心建立一个共享存储区；若系统中已有共享存储区，便忽略 IPC_CREAT。

附：

操作允许权	八进制数
用户可读	00400
用户可写	00200
小组可读	00040
小组可写	00020
其它可读	00004
其它可写	00002

控制命令	值
IPC_CREAT	0001000
IPC_EXCL	0002000

例：shmid=shmget(key, size, IPC_CREAT|0640)

创建一个关键字为 key，长度为 size 的共享存储区，该共享存储区允许用户自己进行读写访问，只允许同组用户进行读访问。

2、shmat( )

共享存储区的附接。从逻辑上将一个共享存储区附接到进程的虚拟地址空间上。

系统调用格式：　virtaddr=shmat(shmid,addr,flag)

参数定义：　　char　*shmat(shmid,addr,flag);
　　int　shmid,flag;
　　char　* addr;

其中，shmid 是共享存储区的标识符；addr 是用户给定的，将共享存储区附接到进程的虚地址空间，当为 0 时，表示由操作系统分配一个地址；flag 其值为 SHM_RDONLY 时，表示只能读；其值为 0 时，表示可读、可写；其值为 SHM_RND（取整）时，表示操作系统在必要时舍去这个地址。该系统调用的返回值是共享存储区所附接到的进程虚地址 viraddr。

3、shmdt( )

把一个共享存储区从指定进程的虚地址空间断开。

系统调用格式：　　shmdt(addr)

参数定义：　　int　shmdt(addr);
　　char　addr;

其中，addr 是要断开连接的虚地址，亦即以前由连接的系统调用 shmat( )所返回的虚地址。调用成功时，返回 0 值，调用不成功，返回-1。

4、shmctl( )

共享存储区的控制，对其状态信息进行读取和修改。

系统调用格式：　　shmctl(shmid,cmd,buf)

参数定义：　　int　shmctl(shmid,cmd,buf);
　　int　shmid, cmd;
　　struct　shmid_ds　*buf;

其中，buf 是用户缓冲区地址，cmd 是操作命令。命令可分为多种类型：

（1）用于查询有关共享存储区的情况。如其长度、当前连接的进程数、共享区的创建者标识符等；

（2）用于设置或改变共享存储区的属性。如共享存储区的许可权、当前连接的进程计数等；

（3）对共享存储区的加锁和解锁命令；

（4）删除共享存储区标识符等。

上述的查询是将 shmid 所指示的数据结构中的有关成员，放入所指示的缓冲区中；而设置是用由 buf 所指示的缓冲区内容来设置由 shmid 所指示的数据结构中的相应成员。

三、参考程序

```
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#define   SHMKEY   75
int   shmid,i;     int   *addr;

void   client( )
{   int i;
    shmid=shmget(SHMKEY,1024,0777);          /*打开共享存储区*/
   addr=shmat(shmid,0,0);                    /*获得共享存储区首地址*/
   for (i=9;i>=0;i--)
      {   while (*addr!=-1);
          printf("(client) sent\n");
          *addr=i;
    }
   exit(0);
   }

void   server( )
{
   shmid=shmget(SHMKEY,1024,0777|IPC_CREAT); /*创建共享存储区*/
   addr=shmat(shmid,0,0);              /*获取首地址*/
   do
    {
          *addr=-1;
          while (*addr==-1);
          printf("(server) received\n");
     }while (*addr);
   shmctl(shmid,IPC_RMID,0);         /*撤消共享存储区，归还资源*/
   exit(0);
}

   main( )
   {
       while ((i=fork( ))= =-1);
       if (!i) server( );
       system(“ipcs   -m”);
       while ((i=fork( ))= =-1);
       if (!i) client( );
       wait(0);
```

```
        wait(0);
    }
```

四、程序说明

1、为了便于操作和观察结果，用一个程序作为“引子“，先后 fork()两个子进程，server 和 client，进行通信。

2、server 端建立一个 key 为 75 的共享区，并将第一个字节置为-1，作为数据空的标志。等待其他进程发来的消息。当该字节的值发生变化时，表示收到了信息，进行处理。然后再次把它的值设为-1，如果遇到的值为 0，则视为为结束信号，取消该队列，并退出 server。server 每接收到一次数据后显示“(server)received”。

3、client 端建立一个 key 为 75 的共享区，当共享取得第一个字节为-1 时，server 端空闲，可发送请求。client 随即填入 9 到 0。期间等待 server 端的再次空闲。进行完这些操作后，client 退出。client 每发送一次数据后显示“(client)sent”。

4、父进程在 server 和 client 均退出后结束。

五、运行结果

和预想的完全一样。但在运行过程中，发现每当 client 发送一次数据后，server 要等待大约 0.1 秒才有响应。同样，之后 client 又需要等待大约 0.1 秒才发送下一个数据。

六、程序分析

出现上述应答延迟的现象是程序设计的问题。当 client 端发送了数据后，并没有任何措施通知 server 端数据已经发出，需要由 client 的查询才能感知。此时，client 端并没有放弃系统的控制权，仍然占用 CPU 的时间片。只有当系统进行调度时，切换到了 server 进程，再进行应答。这个问题，也同样存在于 server 端到 client 的应答过程中。

七、思考题

1、比较两种通信机制中数据传输的时间和性能

由于两种机制实现的机理和用处都不一样，难以直接进行时间上的比较。如果比较其性能，应更加全面的分析。

（1）消息队列的建立比共享区的设立消耗的资源少。前者只是一个软件上设定的问题，后者需要对硬件的操作，实现内存的映像，当然控制起来比前者复杂。如果每次都重新进行队列或共享的建立，共享区的设立没有什么优势。

（2）当消息队列和共享区建立好后，共享区的数据传输，受到了系统硬件的支持，不耗费多余的资源；而消息传递，由软件进行控制和实现，需要消耗一定的 cpu 的资源。从这个意义上讲，共享区更适合频繁和大量的数据传输。

（3）消息的传递，自身就带有同步的控制。当等到消息的时候，进程进入睡眠状态，不再消耗 cpu 资源。而共享队列如果不借助其他机制进行同步，接收数据的一方必须进行不断的查询，白白浪费了大量的 cpu 资源。可见，消息方式的使用更加灵活。

(二) 信号量通信

实验目的

了解和熟悉信号量机制

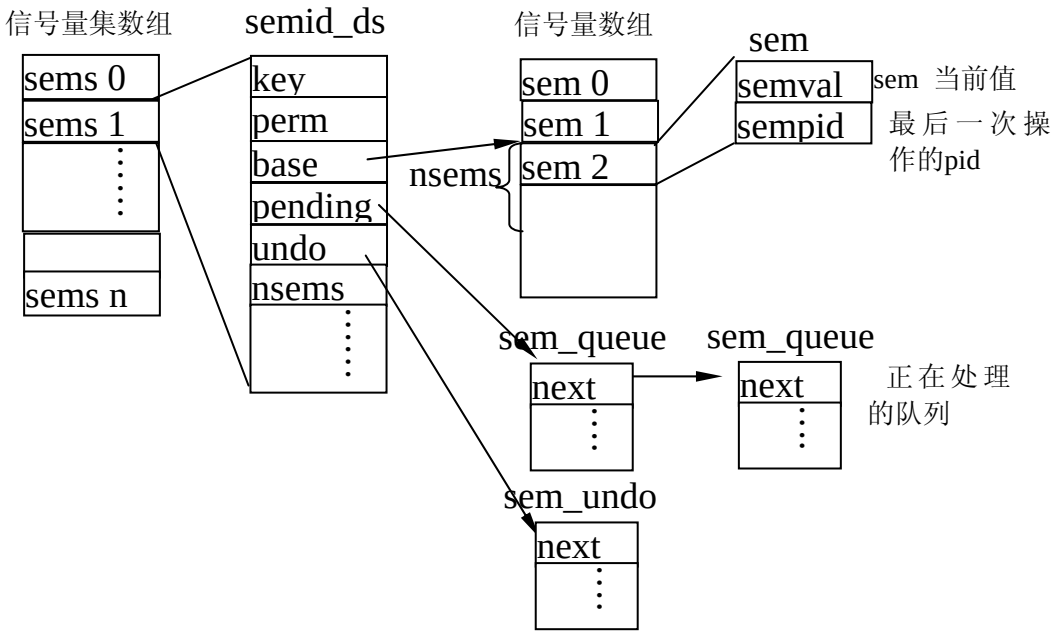
实验内容

编制程序，使用信号量通信机制实现生产者—消费者问题。

实验指导

一、信号量机制的概念

信号量（semaphore）与其它进程通信方式不同，它主要提供对进程间共享资源访问的控制机制。进程可以根据它判定是否能够访问某共享资源，同时，进程也可以修改它的值。除了用于访问控制外，还可用于进程同步。LINUX/UNIX 支持的是系统 V 信号量机制，是计数信号量集。相关数据结构在/usr/include/linux/sem.h 中有定义，图示如下：



二、相关的系统调用

使用与信号量机制有关的系统调用，都需包含以下头文件：

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

1、semget( )

创建、获得一个信号量集描述符。

系统调用格式：　semid=semget(key,nsems,flag)

参数定义：　int　semget(key,nsems,flag);

```
key_t　key;
int　nsems,flag;
```

其中，key 是信号量集的名字；nsems 是指该信号量集所包含信号量的个数；flag 是用户设置的标志和访问方式，如：

IPC_CREAT |0400　　该信号量集是否已被创建，无则创建，是则打开；

IPC_EXCL |0400　　该信号量集的创建是互斥的，若已创建则失败；

semid 是该系统调用返回的描述符，失败则返回-1。

2、semop( )

对信号量集中一个或几个信号量进行操作。

系统调用格式：　semop(semid, sops, nsops)

参数定义：　int semop(semid, sops, nsops)

int semid
struct sembuf *sops
unsigned nsops

其中，semid 是上述 semget 返回的描述符；sops 是结构数组指针；nsops 表示结构数组中成员个数。struct sembuf 定义如下：

struct sembuf { short sem_num;
short sem_op;
short sem_flg; }

其中，sem_num 表示所操作信号量在信号量集中位置，0 对应第一个信号量；sem_op 表示对该信号量的改变值，sem_op>0 表示进程要释放 sem_op 个共享资源，sem_op<0 表示进程要申请 sem_op 个资源；sem_flg 可取值 IPC_NOWAIT 和 SEM_UNDO，IPC_NOWAIT 表示申请资源不能满足时，进程并不等待，而是立即返回，告知申请失败，SEM_UNDO 在进程结束时，相应的操作将被取消。

系统调用的返回值：成功为 0，否则为-1。

3、semctl( )

对信号量集的控制操作

系统调用格式： semctl(semid, semnum, cmd, arg)

参数定义： int semctl(semid, semnum, cmd, arg)
int semid, semnum, cmd;
union semun { semid_ds *buf;
int *array;
int val;
} arg

其中，semid 是上述 semget 返回的描述符；semnum 指定对哪个信号量执行控制操作，只对一些 cmd 操作有意义；arg 用于设置或返回信息，其类型随 cmd 操作而定；cmd 指定执行的操作，可允许的值和意义如下：

IPC_STAT	获取信号量集信息，结果在 arg.buf 中
IPC_SET	设置信号量集信息，设置信息由 arg.buf 提供
IPC_RMID	删除该信号量集
SETALL	设置或更新所有信号量的值，设置值由 arg. array 提供
GETALL	获取所有信号量的值，结果保存在 arg. array 中
SETVAL	设置 semnum 所指定信号量的值为 arg. val
GETVAL	返回 semnum 所指定信号量的值
GETNCNT	返回等待 semnum 所指定信号量的值增加的进程数，即等待该共享资源的进程数
GETPID	返回最后一个对 semnum 所指定信号量执行 semop 操作的进程 ID
GETZCNT	返回等待 semnum 所指定信号量的值变为 0 的进程数

系统调用的返回值：失败为-1，成功与 cmd 相关。

三、参考程序

```
/*  信号量机制系统调用
    sem1.c                          */
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#define SHMKEY 70
#define SEMKEY 80
```

```
#define K           1024
typedef int arr[16][256];
int shmid;
int semid;

main()
{ int i, in=0;
   char *addr;
   arr    *arrp;
   struct sembuf ops1={0,-1,SEM_UNDO},
                 ops2={1, 1,SEM_UNDO};
   extern cleanup();

   for (i=0; i<31; i++)
        signal(i,cleanup);
   shmid = shmget(SHMKEY, 16*K, 0777|IPC_CREAT);
   addr = shmat(shmid, 0, 0);
   arrp = (arr *)addr;
   semid = semget(SEMKEY, 2, 0777|IPC_CREAT);
   semctl(semid,0,SETVAL,16);
   semctl(semid,1,SETVAL,0);
   for (i=0; i<256; i++)
   {
      semop(semid, &ops1, 1);
      (*arrp)[in][0] = i;
      in = (in+1) % 16;
      semop(semid, &ops2, 1);
   }
   sleep(1);
}

cleanup()
  { shmctl(shmid,IPC_RMID,0);
     semctl(semid,0,IPC_RMID,0);
     exit();
  }


/*       sem2.c          */
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#define SHMKEY 70
#define SEMKEY 80
#define K           1024
typedef int arr[16][256];
int shmid;
int semid;

main()
```

```
{ int i, in=0;
   char *addr;
   arr    *arrp;
   struct sembuf ops1={1,-1,SEM_UNDO},
                        ops2={0, 1,SEM_UNDO};

   shmid = shmget(SHMKEY, 16*K, 0777);
   addr = shmat(shmid, 0, 0);
   arrp = (arr *)addr;
   semid = semget(SEMKEY, 2, 0777);
   for (i=0; i<256; i++)
   {
      semop(semid, &ops1, 1);
      printf("sem2: %d = %d\n",i,(*arrp)[in][0]);
      in = (in+1) % 16;
      semop(semid, &ops2, 1);
   }
}
```

四、思考题

1. 对上述程序加上注释。
2. 上述程序中 sem1 是生产者，sem2 是消费者，如果改为 2 个生产者或 2 个消费者，情况会是如何？

实验八 处理机调度

实验目的

在多道程序设计或多任务系统中，同时处于就绪状态的进程有若干个。为使各进程能有条不紊地执行，操作系统必须按某种调度策略来选择一个进程运行。要求学生设计一个单处理机调度模拟算法程序，以巩固和加深处理机调度的概念。

实验内容

在下述三种调度算法中选择一题进行编程实验：

1. 设计一个按先来先服务调度的算法；
2. 设计一个按动态优先级调度的算法；
3. 设计一个按时间片轮转法调度的算法；

实验提示

一、 设计一个按先来先服务调度的算法

1. 假定系统内有 5 个进程，每个进程由一个进程控制块（PCB）来标识。进程控制块的内容如右图所示。

   进程名： 即进程标识。

   链接指针：按进程到达时间处于就绪状态的进程组成一个就绪队列。指针指出下一个到达进程的进程控制块。最后一个进程的链接指针为 NULL。

   到达时间：进程创建时的系统时间或由用户指定。调度时总是选择到达时间最早的进程。

   估计运行时间：设计者任意指定的一个时间值。

   进程状态：为简单起见，假定进程只有两种状态：就绪和完成。进程一创建就处于就绪状态（R），进程运行结束就置为完成状态。

进程名
链接指针
到达时间
估计运行时间
进程状态

2. 设置一个队首指针 head，用来指出最先进入系统的进程。
3. 处理机调度时，总是选择队首指针所指进程投入运行。由于本实验是模拟，所选进程并不实际运行，只是执行：估计运行时间减 1。
4. 程序中应有显示和打印语句，能显示进程的开始运行时间，剩余运行时间，结束时间等。最后显示每个进程的周转时间和平均周转时间。

二、设计一个按动态优先级调度的算法

1. 进程控制块内容和先来先服务调度的算法类似，如右图所示。其中进程优先数由用户指定或程序设定。且优先数越低，优先级越高。
2. 为了调度方便，设计一个指针 head 指向就绪队列，另一个指针指向当前正运行的进程。
3. 处理机调度时，总是选择队列中优先级最高（优先数最低）的进程运行。本实验是模拟，所选进程并不实际运行，只是执行：优先数加 1 和估计运行时间减 1。
4. 进程运行一次后；若剩余时间不为 0，且其优先级低于就绪队列中其它进程，则选择队列中高优先级进程枪占；若剩余时间为 0，则状态改为完成状态，并撤出就绪队列。
5. 若就绪队列不空，重复上述 3，4 步。
6. 程序中应有显示和打印语句，能显示进程的剩余运行时间，优先数，就绪队列中

进程名
链接指针
进程优先数
估计运行时间
进程状态

进排队情况等。

三、 设计一个按时间片轮转法调度的算法

1. 进程控制块内容和先来先服务调度的算法一样，如右图所示。系统设置一个队头和一个队尾指针，分别指向队列的第一个和最后一个进程。
2. 为每个进程确定一个要求运行时间和到达时间。
3. 按进程到达的先后次序排成一个循环队列。
4. 处理机调度时，开始选择队首第一个进程运行，另外再设一个当前运行进程指针，指向当前正运行的进程。
5. 本实验是模拟，所选进程并不实际运行，只是执行：估计运行时间减 1 和输出当前运行进程名。

进程名
链接指针
到达时间
估计运行时间
进程状态

6. 进程运行一次后，应将当前运行进程指针下移一个位置，指向下一个进程。同时还应判断该进程的剩余运行时间是否为 0。若不为 0，则等待下一轮调度；若为 0，则则状态改为完成状态，并撤出就绪队列。
7. 若就绪队列不空，重复上述 5，6 步。
8. 程序中应有显示和打印语句，显示每次选中进程名字和队列变化情况。

实验要求

1. 给出所选实验题目。
2. 给出源程序文件名和执行程序文件名。源程序中要有详细的注释。
3. 给出程序中使用的数据结构和符号说明。
4. 给出程序的流程图。
5. 给出运行结果。
6. 总结收获体会和对题解的改进意见及见解。

参考程序　　按时间片轮转法调度程序

```
#define N 20
#include <stdio.h>
#include <conio.h>
typedef struct pcb              /*  进程控制块定义  */
{ char pname[N];
   int runtime;
   int arrivetime;
   char state;
   struct pcb *next;
  } PCB;
PCB head_input;//就绪队列头指针
PCB head_run; //运行队列头指针
static char R='r',C='c';
unsigned long current; //记录系统当前时间
void inputprocess();        /*  建立进程函数  */
int readyprocess();         /*  建立就绪队列函数  */
int readydata();            /*  判断进程是否就绪函数  */
int runprocess();           /*  运行进程函数  */

int readyprocess()
{ while(1)
   { if(readydata()==0)
```

```
                return 1;
        else      runprocess();
    }
}
int readydata() //判断就绪队列是否为空
{ PCB *p1,*p2,*p3;
    if(head_input.next==NULL)
    { if(head_run.next==NULL)
                return 0;
        else    return 1;
    }
    p1 = head_run.next;
    p2 = &head_run;
    while (p1 != NULL)
    { p2 = p1;
        p1 = p2->next;
    }
    p1 = p2;
    p3 = head_input.next;
    p2 = &head_input;
    while (p3 != NULL)
    { if(( (unsigned long)p3->arrivetime <= current) && (p3->state == R))
        { printf("Time slice is %8ld( time %4ld); Process %s start.\n",
                        current, (current+500)/1000, p3->pname);
            p2->next = p3->next;
            p3->next = p1->next;
            p1->next = p3;
            p1 = p3;
            p3 = p2;
        }
        p2 = p3;
        p3 = p3->next;
    }
    return 1;
}

int runprocess()
{ PCB *p1,*p2;
    if(head_run.next == NULL)
    { current++;
        return 1;
    }
    else
    { p1 = head_run.next;
        p2 = &head_run;
        while(p1 != NULL)
        { p1->runtime--;
            current++;
            if(p1->runtime <= 0)
            { printf("Time slice is %8ld( time %4ld); Process %s end.\n",
                        current, (current+500)/1000, p1->pname);
```

```
            p1->state = C;
            p2->next = p1->next;
            free(p1);
            p1 = p2->next;
          }
          else
          { p2 = p1;
            p1 = p2->next;
          }
      }
      return 1;
   }
}

void inputprocess()
{ PCB *p1,*p2;
   int i,num;
   unsigned long max = 0;
   printf("How many processes do you want to run:");
   scanf("%d",&num);
   p2 = &head_input;

   for (i=0; i<num; i++)
   { p1 = (PCB *)malloc(sizeof(PCB));
      p2->next = p1;
      printf("No.%3d process    input pname:",i+1);
      scanf("%s",p1->pname);
      printf("                              runtime:");
      scanf("%d",&(p1->runtime));
      printf("                           arrivetime:");
      scanf("%d",&(p1->arrivetime));
      p1->runtime = (p1->runtime)*1000;
      p1->arrivetime = (p1->arrivetime)*1000;
      p1->state = R;
      if((unsigned long)(p1->arrivetime) > max)
          max = p1->arrivetime;
      p2 = p1;
   }
   p2->next = NULL;
}

void main()
{ printf("\ntime 1 = 1000 time slice\n");
   current = 0;
   inputprocess();
   readyprocess();
   getch();
}
```

实验九 死锁的避免
用银行家算法和随机算法实现资源分配

实验目的

为了了解系统的资源分配情况，假定系统的一种资源在任一时刻只能被一个进程使用，任何进程已经占有的资源只能由进程自己释放，而不能由其他进程抢占。当进程申请的资源不能满足时，必须等待。因此，只要资源分配算法能保证进程的资源请求，且不出现循环等待，则系统不会出现死锁。通过模拟系统的资源分配算法，了解死锁的产生和避免的办法。

实验内容

编写系统进行资源分配的模拟程序。一个是随机动态地进行资源分配，即只要系统剩余资源能满足进程的当前请求，就立即将资源分配给进程，以观察死锁产生情况；另一个采用银行家算法，可有效避免死锁的产生。

可设计 3~4 个并发进程，共享系统同类型 10 个不可抢占的资源。各进程动态地进行资源的申请和释放。

实验提示

1. 初始化每个进程的最大资源需求量和依次申请资源次序。假定进程控制块的格式如右图所示。其中，进程状态有就绪（R）、等待（W）和完成（C）。当系统不能满足进程的资源申请时，进程处于等待态。资源需求总量表示进程运行过程中对资源的最大需要量。已占资源数表示进程目前已经得到但还未归还的资源数。能执行完标志是银行家算法中在判断系统是否处于安全状态时使用。

进程名
进程状态
当前申请数
资源需求总量
已占资源数
能执行完标志

2. 银行家算法分配资源的原则是：
   （1）验证进程当前申请数是否合法，若当前申请数与已占资源数的和超过进程的资源需求总量，则当前申请数不合法，需重新输入申请数。
   （2）系统当前剩余资源量不能满足进程申请数，进程置为等待态。
   （3）系统当前剩余资源量若能满足进程申请数，则假定将所申请资源数分配给该进程，检查系统是否处于安全状态，即所有进程是否都能得到执行完标志。
   （4）若系统不处于安全状态，分配无效，置申请进程为等待态。否则分配有效，将当前申请数加入该进程的已占资源数。
   （5）检查该进程的已占资源数是否等于进程的资源需求总量。若成立，表示该进程可完成，置进程为完成态，释放所有已占有资源。同时检查是否有等待进程，现在的剩余资源数是否能分配给它们。
   （6）重复上述过程，直到所有进程都成功完成。
3. 随机算法进行资源分配的原则是：
   （1） 验证进程当前申请数是否合法，若当前申请数与已占资源数的和超过进程的资源需求总量，则当前申请数不合法，需重新输入申请数。
   （2） 系统当前剩余资源量不能满足进程申请数，进程置为等待态。
   （3） 系统当前剩余资源量若能满足进程申请数，则实施分配。
   （4） 检查该进程的已占资源数是否等于进程的资源需求总量。若成立，表示该进程可完成，置进程为完成态，释放所有已占有资源。同时检查是否有等待进程，

现在的剩余资源数是否能分配给它们。

（5） 重复上述过程，直到所有进程都成功完成或若干进程形成死锁，即都处于等待状态。

4．程序中应有显示和打印语句，显示每次分配资源的结果，进程状态的变化等信息。

实验要求

1．给出源程序文件名和执行程序名。
2．给出程序中使用的数据结构和符号说明。
3．给出程序的流程图。
4．给出运行结果。
5．总结收获体会和对题解的改进意见及见解。

参考程序 此为 Visual C++ 程序示例

```
#include <iostream>
#include <vector>
// #include <fstream>
using namespace std;
const int TASK_RUNNING = 0;
const int TASK_SUCCEED = 1;
const int TASK_WAITTING = 2;
const int RLength = 10;
int Rcs_left = RLength;
// ofstream ff("result.txt");
class pcb
{
public: int p_pid;
        int p_stat;
        int p_apply;
        int p_occupy;
        bool p_issuc;
        int p_require;
        pcb(int id, int require)
        { p_pid = id;
          p_require = require;
          p_stat = TASK_RUNNING;
          p_occupy = 0;
          p_issuc = false;
          p_apply = 0;
        }
        friend ostream & operator << (ostream &cout,const pcb & p)

{ cout<<p.p_pid<<'\t'<<p.p_stat<<'\t'<<p.p_require<<'\t'<<p.p_occupy<<endl;
          return cout;
        }
 };
void rand(/*vector<int>&resource,*/vector<pcb>&pgrp);
void banker(/*vector<int>&resource,*/vector<pcb>&pgrp);
```

```
int main()
{  vector<pcb>pgrp;
  cout<<"ENTER THE MAX NUMBER FOR THE REQESTED RESOURCE:"<<endl;
  cout<<"ID\tREQUESTED"<<endl;
  int qty;
  for(int i(1);i<=4;i++)
  { do { cout<<i<<'\t';
         cin>>qty;
       } while(qty>Rcs_left || qty<1);
    pgrp.insert(pgrp.begin(),pcb(i,qty));
  }

  cout<<"ALOGRITHM"<<endl
      <<"Random(R)"<<'\t'<<"Banker(B)"<<endl
      <<"ANY OTHER KEY TO QUIT"<<endl;
  char choice;
  cin>>choice;
  if(choice == 'R' || choice == 'r')
         rand(/*resource,*/pgrp);
  else if(choice == 'B' || choice == 'b')
         banker(/*resource,*/pgrp);
  else   return 0;
  return 1;
}

void rand(/*vector<int>&resource,*/vector<pcb>&pgrp)
{ vector<pcb>::iterator p,q;
  vector<pcb>::iterator current;
  int temp;
  cout<<"NOW--------RANDOM ALOGRITHM"<<endl;
  for(;;)
  { for(p=pgrp.begin();p!=pgrp.end();p++)
    { if(p->p_stat == TASK_RUNNING)
      { current = p;
        break;
      }
    }
    if(current->p_apply == 0)
    { cout<<"ENTER THE APPLY FOR THE PROCESS\n"<<current->p_pid<<'\t';
      cin>>temp;
      while(temp>p->p_require - p->p_occupy)
      { cout<<"beyond the real need!"<<endl;
        cout<<"ENTER THE APPLY FOR THE PROCESS\n"<<current->p_pid<<'\t';
        cin>>temp;
      }
      p->p_apply = temp;
    }
```

```
if(current->p_apply > Rcs_left)
{ current->p_stat = TASK_WAITTING;
  cout<<endl<<current->p_pid<<"is waitting\n";
  for(p=pgrp.begin();p!=pgrp.end();p++)
  { if(p->p_stat == TASK_RUNNING) break;
  }
  if(p==pgrp.end())
  { cout<<"LOCKED!!!"<<endl;
    exit(1);
  }
  continue;
}

cout<<temp<<"\tresource are accepted for"<<p->p_pid<<endl;
cout<<endl;
Rcs_left-=current->p_apply;
current->p_occupy+=current->p_apply;
current->p_apply = 0;
if(current->p_occupy < current->p_require)
{ pcb proc(*current);
  pgrp.erase(current);
  pgrp.insert(pgrp.end(),proc);
  continue;
}
cout<<endl<<"process\t"<<p->p_pid<<"\thas succeed!!"<<endl;
Rcs_left+=current->p_occupy;
current->p_stat = TASK_SUCCEED;
for(p=pgrp.begin();p!=pgrp.end();p++)
{ if(p->p_stat == TASK_WAITTING)
     break;
}
if(p==pgrp.end())
{ for(q=pgrp.begin();q!=pgrp.end();q++)
     if(q->p_stat == TASK_RUNNING) break;
  if(q==pgrp.end())
  { cout<<"SUCCEED!!"<<endl;
    exit(0);
  }
  else continue;
}
for(p=pgrp.begin();p!=pgrp.end();p++)
{ if(p->p_stat==TASK_WAITTING && Rcs_left>=p->p_apply)
       break;
}
if(p!=pgrp.end())
{ p->p_stat = TASK_RUNNING;
  pcb proc(*p);
  pgrp.erase(p);
```

```
      pgrp.insert(pgrp.end(),proc);
      continue;
    }
  }
}

void banker(/*vector<int>&resource,*/vector<pcb>&pgrp)
{ vector<pcb>::iterator p;
  vector<pcb>::iterator current,q;
  pcb proc(0,0);
  int length;
  cout<<"NOW--------BANKER ALOGRITHM"<<endl;
  for(;;)
  { for(p=pgrp.begin();p!=pgrp.end();p++)              /* 找到运行进程  */
    { if(p->p_stat == TASK_RUNNING)
      { current = p;
        break;
      }
    }
    if(current->p_apply == 0)                          /* 输入申请资源数 */
    { cout<<"ENTER THE APPLY FOR THE PROCESS\n"<<current->p_pid<<'\t';
      cin>>current->p_apply;
      while(current->p_apply > current->p_require - current->p_occupy)
      { cout<<"beyond the real need!"<<endl;
        cout<<"ENTER THE APPLY FOR THE PROCESS\n"<<current->p_pid<<'\t';
        cin>>current->p_apply;
      }
    }

    if(current->p_apply > Rcs_left)
    { current->p_stat = TASK_WAITTING;
      proc = *current;
      pgrp.erase(current);
      pgrp.insert(pgrp.end(),proc);
      cout<<endl<<p->p_pid<<"  is waitting!\n";
      continue;
    }

    pcb backup(*current);
    length = Rcs_left;
    current->p_occupy += current->p_apply;
    length -= current->p_apply;
    if(current->p_occupy == current->p_require)
    { length += current->p_require;
      current->p_issuc = true;
    }

   int flag = 1;
```

```
while(flag==1)
{ flag = 0;
for(p=pgrp.begin();p!=pgrp.end();p++)
{ if(p->p_stat == TASK_SUCCEED) continue;
  if(p->p_issuc==true) continue;
  if((p->p_require - p->p_occupy) > length) continue;
  else
  { p->p_issuc = true;
    length += p->p_occupy;
    flag = 1;
    cout<<p->p_pid<<"  "<<p->p_occupy<<"  "<<length<<endl;
    continue;
  }
}
}
for(p=pgrp.begin();p!=pgrp.end();p++)
{ if(p->p_issuc==false && p->p_stat != TASK_SUCCEED)
       break;
}
if(p!=pgrp.end())
{ current->p_occupy = backup.p_occupy;
  current->p_stat = TASK_WAITTING;
  cout<<endl<<current->p_pid<<"  is waitting."<<endl;
  proc = *current;
  pgrp.erase(current);
  pgrp.insert(pgrp.end(),proc);
  for(p=pgrp.begin();p!=pgrp.end();p++)
      p->p_issuc = false;
  continue;
}

Rcs_left-=current->p_apply;
cout<<endl<<current->p_pid<<" get "<<current->p_apply<<" resource(s)!"<<endl;
current->p_apply = 0;
for(p=pgrp.begin();p!=pgrp.end();p++)
   p->p_issuc = false;
if(current->p_occupy < current->p_require)
{ proc = *current;
  pgrp.erase(current);
  pgrp.insert(pgrp.end(),proc);
  continue;
}
current->p_stat = TASK_SUCCEED;
current->p_occupy = 0;
cout<<endl<<current->p_pid<<"  has finished!!!"<<endl;
Rcs_left+=current->p_require;

for(p=pgrp.begin();p!=pgrp.end();p++)
```

```
    { if(p->p_stat == TASK_WAITTING)
         break;
    }
    if(p==pgrp.end())
    { for(q=pgrp.begin();q!=pgrp.end();q++)
         if(q->p_stat == TASK_RUNNING) break;
      if(q==pgrp.end())
      { cout<<endl<<"SUCCEED!!"<<endl;
        exit(0);
      }
      else continue;
    }

    p->p_stat = TASK_RUNNING;
    continue;
  }
}
```

实验十 动态分区存储管理

实验目的

在多道程序系统中，由于内存资源有限，通常不可能把全部进程都装入内存。由操作系统对内存的使用进行合理和有效的管理，实现内存空间的分配和回收。通过模拟算法实现，加深理解动态分区存储管理的过程。

实验内容

动态分区存储管理是指在处理作业过程中建立分区，使分区大小正好适合作业的需要，而且分区的个数是可变的。当要装入一个作业时，根据作业需要的内存量，查看是否有足够的空闲空间。若有，则按需求量分割一部分给作业；若无，则作业等待。作业完成，所占有的存储空间需回收。

编写采用动态分区存储管理，使用首次（或最佳）适应算法实现内存分配和回收的模拟程序。

实验提示

1. 为了说明内存中哪些分区是空闲的，可以用来装入新的作业，必须要有一张空闲区表，如右图所示。其中，起始地址表示各空闲区的内存开始地址；长度表示空闲区的大小；状态未分配表示该栏目是有效空闲区，状态空条目表示该栏目没有登记信息或信息无效。
   由于分区个数不定，空闲区表中应有足够的空表目项。
   再设一个已分配区表，记录作业或进程的内存占有情况。

起始地址	长 度	状 态
45K	20K	未分配
110K	146K	未分配
30K	10K	空条目
		空条目
80K	15K	空条目
		空条目

2. 当有一个新作业要求装入内存时，查空闲区表，从中找出一个大小能满足需要的空闲区，分割出一部分给作业，余下的仍留在空闲区表中。为方便查找，要不断对表进行整理，把“空条目”表项总是留在表的后部。找不到能满足需要的空闲区，则等待。
3. 当一个作业执行完成时，作业所占有的内存分区应收回。收回时应考虑相邻空闲区的合并问题。分 4 种情况处理：回收区下邻（低地址）空闲区；回收区上邻（高地址）空闲区；回收区上下都有邻接空闲区；回收区不与空闲区邻接。
4. 开始应对空闲区表和已分配区表进行初始化。学生自己设计一个作业申请队列和作业完成后的释放次序。
5. 每次作业申请和完成后，显示和打印相应的处理结果及空闲区表的变化情况。
6. 有兴趣的同学可增加空闲区“紧凑”功能。即当一个新作业要求装入内存时，查空闲区表，找不到能满足需要的空闲区，但全部空闲区大小的总和能满足需求。这时，修改空闲区表和已分配区表，把分散的小空闲区归并成一个大空闲区，再进行分配。

实验要求

1. 给出源程序文件名和执行程序名。
2. 给出程序中使用的数据结构和符号说明。
3. 给出程序的流程图。

4. 给出运行结果。
5. 总结收获体会和对题解的改进意见及见解。

参考程序

```
#include"stdio.h"
#define N 5
struct freearea
{ int startaddr;
  int size;
  int state;
} freeblock[N] = {{20,20,1},{80,50,1},{150,100,1},{300,30,0},{600,100,1}};
/*  定义分配主存空间函数 alloc() */
int alloc(int applyarea)
{ int i,tag=0;
  for(i=0; i<N; i++)
  { if(freeblock[i].state==1 && freeblock[i].size>applyarea)
    { freeblock[i].startaddr += applyarea;
      freeblock[i].size -= applyarea;
      tag = 1;
      return freeblock[i].startaddr-applyarea;
    }
    else if(freeblock[i].state==1 && freeblock[i].size==applyarea)
    { freeblock[i].state = 0;
      tag = 1;
      return freeblock[i].startaddr;
    }
  }
  if(tag == 0) return -1;
}
/*  定义主存空间回收函数 setfree() */
void setfree()
{ int s,l,tag1=0,tag2=0,tag3=0,i,j;
  printf("\nInput free area startaddress:    ");
  scanf("%d",&s);//空闲区开始地址
  printf("\n              Input free area size:    ");
  scanf("%d",&l);//空闲区长度
  for(i=0; i<N; i++)
  { if(freeblock[i].startaddr==s+l && freeblock[i].state==1)
    { l += freeblock[i].size;
      tag1 = 1;
      for(j=0; j<N; j++)
      { if(freeblock[j].startaddr+freeblock[j].size==s && freeblock[j].state==1)
        { freeblock[i].state = 0;
          freeblock[j].size += l;
          tag2 = 1;
          break;
        }
      }
      if(tag2 == 0)
      { freeblock[i].startaddr = s;
```

```
            freeblock[i].size = l;
            break;
        }
      }
    }
    if(tag1 == 0)
    { for(i=0; i<N; i++)
      { if(freeblock[i].startaddr+freeblock[i].size==s && freeblock[i].state==1)
        { freeblock[i].size += l;
      tag3 = 1;
            break;
        }
      }
      if(tag3 == 0)
        for(j=0; j<N; j++)
          if(freeblock[j].state == 0)
          { freeblock[j].startaddr = s;
            freeblock[j].size = l;
            freeblock[j].state = 1;
            break;
          }
    }
}

void adjust()
{ int i,j;
    struct freearea middata;
    for(i=1; i<N; i++)//将空闲区按始地址顺序在表中排列
    for(j=0; j<N-i; j++)
      if(freeblock[j].startaddr > freeblock[j+1].startaddr)//排序
      { middata = freeblock[j];
        freeblock[j] = freeblock[j+1];
        freeblock[j+1] = middata;
      }
    for(i=1; i<N; i++)//将空表目放在表后面
    for(j=0; j<N-i; j++)
      if(freeblock[j].state == 0 && freeblock[j+1].state == 1)
      { middata = freeblock[j];
        freeblock[j] = freeblock[j+1];
        freeblock[j+1] = middata;
      }
}

void print()
{ int i;
    printf("        |----------------------------------|\n");
    printf("        |   startaddr      size          state      |\n");
    for(i=0; i<N; i++)
      printf("        |        %4d          %4d              %4d       |\n",
            freeblock[i].startaddr,freeblock[i].size,freeblock[i].state);
}
```

```
main()
{ int applyarea,start;
   long i;
   char end;
   printf("\nIs there any job request memory? (y or n): ");
   while( (end=getchar())=='y' )
   { printf("At first the free memory is this:\n");
      adjust();
      print();
      printf("Input request memory size: ");
      scanf("%d",&applyarea);
      start = alloc(applyarea);
      if(start==-1) printf("There is no fit memory.please wait!!\n");
      else { printf("Job's memory start address is: %d\n",start);
                printf("                              Job size is: %d\n",applyarea);
                printf("After allocation the free memory is this:\n");
                adjust();
                print();
           printf("Job is running.\n");
                for(i=0; i<100000; i++);
                printf("Job is terminated.\n");
              }
      setfree();
      adjust();
      print();
      printf("Is there any job that is waiting? (y or n): ");
      end = getchar();
   }
}→
```

实验十一　虚拟存储管理
常用页面置换算法模拟实验

实验目的

通过模拟实现请求页式存储管理的几种基本页面置换算法，了解虚拟存储技术的特点，掌握虚拟存储请求页式存储管理中几种基本页面置换算法的基本思想和实现过程，并比较它们的效率。

实验内容

设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。

1、最佳淘汰算法（OPT）

2、先进先出的算法（FIFO）

3、最近最久未使用算法（LRU）

4、最不经常使用算法（LFU）

5、最近未使用算法（NUR）

命中率＝1 －页面失效次数 / 页地址流长度

实验预备内容

本实验的程序设计基本上按照实验内容进行。即首先用 srand( )和 rand( )函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。

（1）通过随机数产生一个指令序列，共 320 条指令。指令的地址按下述原则生成：

A：50%的指令是顺序执行的

B：25%的指令是均匀分布在前地址部分

C：25%的指令是均匀分布在后地址部分

具体的实施方法是：

A：在[0，319]的指令地址之间随机选取一起点 m

B：顺序执行一条指令，即执行地址为 m+1 的指令

C：在前地址[0,m+1]中随机选取一条指令并执行，该指令的地址为 m’

D：顺序执行一条指令，其地址为 m’+1

E：在后地址[m’+2，319]中随机选取一条指令并执行

F：重复步骤 A-E，直到 320 次指令

（2）将指令序列变换为页地址流

设：页面大小为 1K；

用户内存容量 4 页到 32 页；

用户虚存容量为 32K。

在用户虚存中，按每 K 存放 10 条指令排列虚存地址，即 320 条指令在虚存中的存放方式为：

第 0 条-第 9 条指令为第 0 页（对应虚存地址为[0，9]）

第 10 条-第 19 条指令为第 1 页（对应虚存地址为[10，19]）

……

第 310 条-第 319 条指令为第 31 页（对应虚存地址为[310，319]）

按以上方式，用户指令可组成 32 页。

实验指导

一、虚拟存储系统

UNIX 中，为了提高内存利用率，提供了内外存进程对换机制；内存空间的分配和回收均以页为单位进行；一个进程只需将其一部分（段或页）调入内存便可运行；还支持请求调页的存储管理方式。

当进程在运行中需要访问某部分程序和数据时，发现其所在页面不在内存，就立即提出请求（向 CPU 发出缺中断），由系统将其所需页面调入内存。这种页面调入方式叫请求调页。

为实现请求调页，核心配置了四种数据结构：页表、页框号、访问位、修改位、有效位、保护位等。

二、页面置换算法

当 CPU 接收到缺页中断信号，中断处理程序先保存现场，分析中断原因，转入缺页中断处理程序。该程序通过查找页表，得到该页所在外存的物理块号。如果此时内存未满，能容纳新页，则启动磁盘 I/O 将所缺之页调入内存，然后修改页表。如果内存已满，则须按某种置换算法从内存中选出一页准备换出，是否重新写盘由页表的修改位决定，然后将缺页调入，修改页表。利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。

常用的页面置换算法有

1、最佳置换算法（Optimal）

2、先进先出法（Fisrt In First Out）

3、最近最久未使用（Least Recently Used）

4、最不经常使用法（Least Frequently Used）

5、最近未使用法（No Used Recently）

三、参考程序：

```
// #include <time.h>                    /* 在 windows 系统中，增加此头文件   */
#define TRUE 1
#define FALSE 0
#define INVALID -1
#define NULL  0
#define RAND_MAX  32767/32767/2      /* 在windows系统中改为   32767   */

#define  total_instruction  320      /*指令流长*/
#define  total_vp  32                /*虚页长数*/
#define  clear_period  50            /*清0周期*/

typedef struct                       /*页面结构*/
{
    int pn,pfn,counter,time;//页号，页面框架号，计数器，时间
}pl_type;//页表项
pl_type pl[total_vp];                /*页面结构数组*/

struct pfc_struct{                   /*页面控制结构*/
    int pn,pfn;
    struct pfc_struct *next;
};//页表的链式存储方式

typedef struct pfc_struct pfc_type;
```

```
pfc_type pfc[total_vp],*freepf_head,*busypf_head,*busypf_tail;

int diseffect,  a[total_instruction];//页面失效次数，指令流数据组
int page[total_instruction],  offset[total_instruction];
//每条指令所属的页号；每页装入10条指令后取模运算页号偏移值
int  initialize(int);
int  FIFO(int);
int  LRU(int);
int  LFU(int);
int  NUR(int);
int  OPT(int);

int main( )
{
  int s,i,j;
  srand(10*getpid());                /* 由于每次运行时进程号不同，故可用来作为初始化
随机数队列的“种子”．在windows系统中改为 srand(time(NULL));  */

s=(float)319*rand( )/RAND_MAX +1;  //
for(i=0;i<total_instruction;i+=4) /*产生指令队列*/
{
     if(s<0||s>319)
     {
       printf("When i==%d,Error,s==%d\n",i,s);
       exit(0);
     }
     a[i]=s;                                  /*任选一指令访问点m*/
     a[i+1]=a[i]+1;                           /*顺序执行一条指令*/
     a[i+2]=(float)a[i]*rand( )/RAND_MAX; /*执行前地址指令m’ */
     a[i+3]=a[i+2]+1;                         /*顺序执行一条指令*/

     s=(float)(318-a[i+2])*rand( )/RAND_MAX +a[i+2]+2;
     if((a[i+2]>318)||(s>319))
       printf("a[%d+2],a number which is :%d and s==%d\n",i,a[i+2],s);

}
for (i=0;i<total_instruction;i++) /*将指令序列变换成页地址流*/
{
     page[i]=a[i]/10;
     offset[i]=a[i]%10;
}
for(i=4;i<=32;i++)   /*用户内存工作区从4个页面到32个页面*/
{
      printf("---%2d page frames---\n",i);
      FIFO(i);
      LRU(i);
      LFU(i);
```

```
      NUR(i);
      OPT(i);

}
   return 0;
}

int initialize(total_pf)              /*初始化相关数据结构*/
int total_pf;//分配给进程的物理块数                  /*用户进程的内存页面数*/
{int i;
diseffect=0;
for(i=0;i<total_vp;i++)//有32页，对每页进行循环，初始化页表
{
       pl[i].pn=i;//页号
       pl[i].pfn=INVALID; //物理块号       /*置页面控制结构中的页号，页面为空*/
       pl[i].counter=0;
       pl[i].time=-1;          /*页面控制结构中的访问次数为0，时间为-1*/
}
for(i=0;i<total_pf-1;i++)//total_pf,物理块数
{
       pfc[i].next=&pfc[i+1];
       pfc[i].pfn=i;
}   /*建立pfc[i-1]和pfc[i]之间的链接*///把可用的几个物理块前后链接起来
pfc[total_pf-1].next=NULL;//最后一个物理块要单独处理
pfc[total_pf-1].pfn=total_pf-1;
freepf_head=&pfc[0];          /*空页面队列的头指针为pfc[0]*/
//开始的时候所有的物理块都是空的
return 0;
}


int FIFO(total_pf)               /*先进先出算法*/
int total_pf;                     /*用户进程的内存页面数*/
{
     int i,j;
     pfc_type *p;
     initialize(total_pf);          /*初始化相关页面控制用数据结构*/
     busypf_head=busypf_tail=NULL; /*忙页面队列头，队列尾链接*/最初，所有物理块都
                                    //未被占用，因此忙队列为空
   for(i=0;i<total_instruction;i++)//对所有指令进行循环，模式顺序执行所有指令
   {
   if(pl[page[i]].pfn==INVALID)    /*页面失效*/,所有页表项还没有分配物理块号
     {//如果该指令对应的页还没有分配物理块的话，下面将调入内存
      diseffect+=1;                  /*失效次数*/
      if(freepf_head==NULL)          /*无空闲页面*///空闲页面指的是空闲物理块？
       {
         p=busypf_head->next;//忙内存块
         pl[busypf_head->pn].pfn=INVALID;//换出去，则以为着没有分配
```

```
          freepf_head=busypf_head;  /*释放忙页面队列的第一个页面*/因此空了一个块了
          freepf_head->next=NULL;//只释放一个页面，故只有一个元素
          busypf_head=p;//忙队列头指针向后移动一个位置
        }
         p=freepf_head->next;          /*按FIFO方式调新页面入内存页面*/
         freepf_head->next=NULL;
         freepf_head->pn=page[i];
         pl[page[i]].pfn=freepf_head->pfn;

    if(busypf_tail==NULL)//被占用一个了，所以忙队列不为空了
        busypf_head=busypf_tail=freepf_head;
     else
    {
      busypf_tail->next=freepf_head;  /*free页面减少一个*/
      busypf_tail=freepf_head;
     }
        freepf_head=p;
     }
}
printf("FIFO:%6.4f\n",1-(float)diseffect/320);

return 0;
}

int LRU (total_pf)        /*最近最久未使用算法*/
int total_pf;
{
 int min,minj,i,j,present_time;
 initialize(total_pf);
 present_time=0;

for(i=0;i<total_instruction;i++)
  {
      if(pl[page[i]].pfn==INVALID)              /*页面失效*/
    {
         diseffect++;
         if(freepf_head==NULL)               /*无空闲页面*/
      {
         min=32767;
         for(j=0;j<total_vp;j++)             /*找出time的最小值*/
           if(min>pl[j].time&&pl[j].pfn!=INVALID)
                    {
              min=pl[j].time;
              minj=j;
             }
          freepf_head=&pfc[pl[minj].pfn];   //腾出一个单元
          pl[minj].pfn=INVALID;
          pl[minj].time=-1;
```

```
            freepf_head->next=NULL;
         }
           pl[page[i]].pfn=freepf_head->pfn;   //有空闲页面,改为有效
           pl[page[i]].time=present_time;
       freepf_head=freepf_head->next;      //减少一个free 页面
        }
       else
          pl[page[i]].time=present_time;        //命中则增加该单元的访问次数

       present_time++;
      }
printf("LRU:%6.4f\n",1-(float)diseffect/320);
return 0;
}

int NUR(total_pf)                    /*最近未使用算法*/
int  total_pf;
{ int i,j,dp,cont_flag,old_dp;
pfc_type *t;
initialize(total_pf);
dp=0;
for(i=0;i<total_instruction;i++)
{ if (pl[page[i]].pfn==INVALID)          /*页面失效*/
     {diseffect++;
      if(freepf_head==NULL)                /*无空闲页面*/
         { cont_flag=TRUE;
           old_dp=dp;
           while(cont_flag)
             if(pl[dp].counter==0&&pl[dp].pfn!=INVALID)
                 cont_flag=FALSE;
             else
            {
         dp++;
             if(dp==total_vp)
            dp=0;
             if(dp==old_dp)
                for(j=0;j<total_vp;j++)
               pl[j].counter=0;
             }
           freepf_head=&pfc[pl[dp].pfn];
           pl[dp].pfn=INVALID;
           freepf_head->next=NULL;
         }
           pl[page[i]].pfn=freepf_head->pfn;
           freepf_head=freepf_head->next;
      }
else
     pl[page[i]].counter=1;
```

```
     if(i%clear_period==0)
        for(j=0;j<total_vp;j++)
             pl[j].counter=0;
}
printf("NUR:%6.4f\n",1-(float)diseffect/320);

return 0;
}

int OPT(total_pf)       /*最佳置换算法*/
int total_pf;
{int i,j, max,maxpage,d,dist[total_vp];
pfc_type *t;
initialize(total_pf);
for(i=0;i<total_instruction;i++)
{ //printf("In OPT for 1,i=%d\n",i);
//i=86;i=176;206;250;220,221;192,193,194;258;274,275,276,277,278;
  if(pl[page[i]].pfn==INVALID)      /*页面失效*/
   {
      diseffect++;
      if(freepf_head==NULL)         /*无空闲页面*/
         {for(j=0;j<total_vp;j++)
               if(pl[j].pfn!=INVALID) dist[j]=32767;  /* 最大"距离" */
           else dist[j]=0;
          d=1;
          for(j=i+1;j<total_instruction;j++)
            {
          if(pl[page[j]].pfn!=INVALID)
             dist[page[j]]=d;
             d++;
             }
          max=-1;
          for(j=0;j<total_vp;j++)
          if(max<dist[j])
            {
          max=dist[j];
          maxpage=j;
         }
           freepf_head=&pfc[pl[maxpage].pfn];
           freepf_head->next=NULL;
           pl[maxpage].pfn=INVALID;
          }
   pl[page[i]].pfn=freepf_head->pfn;
   freepf_head=freepf_head->next;
     }
}
printf("OPT:%6.4f\n",1-(float)diseffect/320);
```

```
return 0;
}

int  LFU(total_pf)        /*最不经常使用置换法*/
int total_pf;
{
 int i,j,min,minpage;
 pfc_type *t;
 initialize(total_pf);
 for(i=0;i<total_instruction;i++)
    {  if(pl[page[i]].pfn==INVALID)      /*页面失效*/
        { diseffect++;
          if(freepf_head==NULL)          /*无空闲页面*/
           { min=32767;
            for(j=0;j<total_vp;j++)
              {if(min>pl[j].counter&&pl[j].pfn!=INVALID)
                  {
            min=pl[j].counter;
                minpage=j;
           }
                pl[j].counter=0;
                }
            freepf_head=&pfc[pl[minpage].pfn];
            pl[minpage].pfn=INVALID;
            freepf_head->next=NULL;
            }
       pl[page[i]].pfn=freepf_head->pfn;   //有空闲页面,改为有效
           pl[page[i]].counter++;
       freepf_head=freepf_head->next;      //减少一个free 页面
        }
    else
           pl[page[i]].counter++;

      }
printf("LFU:%6.4f\n",1-(float)diseffect/320);

return 0;
}
```

四、运行结果

```
 4 page frams
 FIFO: 0.7312
 LRU: 0.7094
LFU: 0.5531
NUR: 0.7688
OPT: 0.9750
 5 page frams
.....
```

五、分析

1、从几种算法的命中率看，OPT 最高，其次为 NUR 相对较高，而 FIFO 与 LRU 相差无几，最低的是 LFU。但每个页面执行结果会有所不同。

2、OPT 算法在执行过程中可能会发生错误

六、思考

1、为什么 OPT 在执行时会有错误产生？

实验十二 SPOOLING 技术

实验目的

在系统配备的设备数量远小于系统中并发运行进程数时，为了使并发进程能更好地共享系统设备，把独占设备转化为共享设备使用，操作系统提供了 SPOOLING 假脱机技术。通过设计一个 SPOOLING 假脱机输出模拟程序，帮助学生更好地理解和掌握这种技术。

实验内容

设计一个 SPOOLING 假脱机输出模拟系统，包括 SPOOLING 输出进程，两个请求输出的用户进程，以及一个 SPOOLING 输出服务程序。当用户进程需要输出信息时，调用输出服务程序，由输出服务程序把输出信息送入输出井中；并申请产生一个输出请求块（用来记录请求输出的用户进程名，输出信息在输出井中的位置，要输出信息的长度等），通知和等待 SPOOLING 输出进程进行实际输出。SPOOLING 输出进程工作时，根据输出请求块记录的内容，把输出井中信息显示或打印出来。SPOOLING 进程和两个用户进程可并发执行。

实验提示

1. SPOOLING 进程和每个用户进程都有相应的进程控制块表示。进程的状态有以下几种： 可执行态 0 表示进程正在执行或等待调度；
   等待态 1 表示输出请求块满，请求输出的用户进程等待；
   等待态 2 表示输出井空，SPOOLING 输出进程等待；
   等待态 3 表示输出井满，请求输出的用户进程等待；
   完成态 表示执行完成。
2. 进程状态转换条件是：
   （1） 各进程初始都为可执行态；
   （2） 服务程序在将信息送输出井时，发现输出井已满，将调用进程置为等待态 1；
   （3） SPOOLING 进程执行输出时，若输出井为空，则置为等待态 2；
   （4） SPOOLING 进程完成一个输出请求块的信息输出后，应立即释放该请求块所占的输出井空间，并将在等待输出井（若有）的用户进程置为可执行态；
   （5） 服务程序将输出信息送入输出井并形成输出请求块后，发现 SPOOLING 进程处于等待态 2，应把它置为可执行态；
   （6） 用户进程申请输出请求块时，没有可用的请求块，置为等待态 3；
   （7） 用户进程完成全部输出任务后，置为完成态。
3. 进程调度采用随机算法。两个请求输出的用户进程的调度概率各为 45%，SPOOLING 输出进程为 10%，这由随机数发生器产生的随机数来模拟确定。
4. 输出进程的输出信息可假定为一串随机数，当遇到数为 0 时，表示该次输出结束。
5. 输出请求块可以定义为一个结构数组，输出井 BUFFER 可定义为一个整型数组。两者都是循环使用。为此，每种数据都有两个指针：一个是使用（放数）指针，一个是释放（取数）指针。
6. 程序中应显示和打印用户进程送入输出井的信息和 SPOOLING 进程输出的信息，检查两者是否一致。

实验要求

1. 给出源程序文件名和执行程序名。
2. 给出程序中使用的数据结构和符号说明。
3. 给出程序的流程图。
4. 给出运行结果。
5. 总结收获体会和对题解的改进意见及见解。

参考程序

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

struct pcb
{ int id;
   int status;
   int firstaddr;
   int length;
   int outbufword;
}*PCB[3];

FILE *f;
struct req//输出请求块
{ int reqname;//请求输出的用户进程名
   int length;//要输出信息的长度
   int addr;//输出信息在输出井中的位置
}reqblock[10];
int buffer[2][100];//为两个用户进程设置两个输出井，每个可存放 100 个信息
int c3=10;
int l1=1,l2=1;
int head=0,tail=0;
int t1,t2;
void request(int i)
{ int j,length=0,m;
   struct req *run;
   if( (tail-head) == 10 )
   { PCB[i-1]->status =1;
      return;
   }
   if(i==1)     t1--;
   else            t2--;
   run = &reqblock[tail%10];
   run->reqname = i;
   run->length = 0;
   if(tail == 0)      run->addr = 0;
   else { int index = (tail-1)%10;
             run->addr = reqblock[index].addr + reqblock[index].length;
          }
   for(m=0; m<100; m++)
   { if(buffer[i-1][m] == 0)
      { run->addr = m;
         break;
```

```
        }
    }
    printf("process %d: ",i);
    while(1)
    { j = rand()%10;
        if(j == 0)
        { run->length = length;
            break;
        }
        buffer[i-1][run->addr + length] = j;
        printf("%d ",j);
        length++;
    }
    printf("\n");
    PCB[i-1]->length += length;
    length = 0;
    if(PCB[2]->status == 2)      PCB[2]->status = 0;
    tail++;
}
void spooling()
{ int i,j;
    struct req *run;
    run = &reqblock[head%10];
    printf("PID %d: ",run->reqname);
    fprintf(f,"PID %d:",run->reqname);
    for(i=0; i<run->length; i++)
    { printf("%d ",buffer[run->reqname - 1][run->addr + i]);
        fprintf(f,"%d ",buffer[run->reqname - 1][run->addr + i]);
    }
    printf("\n");
    fprintf(f,"\n");
    head++;
    for(j=0; j<2; j++)
        if(PCB[j]->status == 1)    PCB[j]->status = 0;
}

int main()
{ int i,n;
    f = fopen("result.txt","w");
    for(i=0; i<2; i++)
        for(n=0; n<100; n++)
                buffer[i][n] = 0;
    for(i=0; i<3; i++)
    { struct pcb *temppcb;
        temppcb = (struct pcb *)malloc(sizeof(struct pcb));
        temppcb->id = i;
        temppcb->status = 0;
        temppcb->firstaddr = 0;
        temppcb->length = 0;
        temppcb->outbufword = 1;
        PCB[i] = temppcb;
```

```
    }
    printf("How many work do process 1 want to do? ");
    fprintf(f,"How many work do process 1 want to do? ");
    scanf("%d",&t1);
    fprintf(f,"%d\n",t1);
    printf("How many work do process 2 want to do? ");
    fprintf(f,"How many work do process 2 want to do? ");
    scanf("%d",&t2);
    fprintf(f,"%d\n",t2);
    srand((unsigned)time(NULL));
    while(1)
    { i = rand()%100;
      if(i<=45)
      { if( (PCB[0]->status == 0) && (t1>0) )
            request(1);
      }
      else if(i<=90)
      { if( (PCB[1]->status == 0) && (t2>0) )
            request(2);
      }
      else if(head==tail) PCB[2]->status = 2;
      else    spooling();

      if( (t1==0) && (t2==0) && (head==tail) )    break;
    }
    for(i=0; i<3; i++)
    { free(PCB[i]);
      PCB[i] = NULL;
    }
    fclose(f);
    return 0;
}→
```

实验十三　文件系统

实验目的

通过一个简单多用户文件系统的设计,加深理解文件系统的内部功能及其内部实现

实验内容

为 LINUX 系统设计一个简单的二级文件系统,要求

（1）文件实现下列几条命令

login 用户登录
dir 列文件目录
create 创建文件
delete 删除文件
open 打开文件
close 关闭文件
read 读文件
write 写文件

（2）列目录时要列出文件名、物理地址、保护码和文件长度；

（3）源文件可以进行读写保护

实验指导

（1）首先应当确定文件系统的数据结构：主目录、子目录以及活动文件等。主目录和子目录都以文件的形式存放于磁盘，这样便于查找和修改。

（2）用户创建文件，可以编号存储于磁盘上。如 file0，file1，file2…并以编号作为物理地址，在目录中登记。

（3）参考程序见附录

附录一

UNIX/LINUX 简介

一、了解 UNIX

微型处理机的问世，给信息产业及整个人类社会带来了一场革命。随着基于 Intel80X86 处理器的 IBM PC 机及其兼容机以及接口设备性能指标的不断提高，人们所期望的真正的 PC 机多用户、多任务、分时 OS 应运而生。充满活力的 UNIX 就是其中重要一个。

UNIX 是一个操作系统，它于 1969 年由美国 Bell 实验室的 Ken.Thompson 和 Denuis.Ritchie 在 DEC 小型机上实现，用汇编语言编写的。1973 年用由 Denuis.Ritchie 设计的 C 语言改写了其内核代码的大部分内容。1983 年 UNIX 的设计师 Ken.Thompson 和 Denuis.Ritchie 荣获了图灵奖，充分肯定了 UNIX 在计算机世界中的地位。

UNIX 从一个非常简单的 OS 发展成为性能先进、功能强大、使用广泛的 OS，并成为事实上的多用户、多任务 OS 的标准。因此，在国外特别是在美国，几乎所有的 OS 教科书中，都是以 UNIX 作为实例，对它做了较深入的阐述。

二、UNIX 系统的特性

UNIX 系统能取得如此巨大成功的原因，可归结于它具有以下的一系列特征：

1、开放性

开放性是指系统遵循世界标准规范，特别是遵循了开放系统互连 OSI 国际标准。凡遵循国际标准所开放的硬件和软件，能彼此兼容，可方便地实现互连。UNIX 是目前开放性最好的 OS，它能广泛地配置在从微型机到大、中型机等各种机器上，而且还能方便地将已配置了 UNIX OS 的机器，互连成计算机网络。

2、多用户、多任务环境

它既可以同时支持数十个乃至数百个用户，通过各自的联机终端同时使用一台计算机，而且还允许每个用户同时执行多个任务。例如：在进行字符图形处理时，用户可建立多个任务，分别处理字符的输入、图形的制作和编辑等任务。

3、功能强大、实现高效

UNIX 系统提供了精选的、丰富的系统功能，它使用户能方便地、快速地完成许多其它 OS 所难于实现的功能。UNIX 已成为世界上功能最强大的操作系统之一，而且它在许多功能的实现上还有其独到之处，且是高效的。例如，UNIX 的目录结构、磁盘空间的管理方式、I/O 重定向和管道功能等。这些功能及其实现技术已被其它 OS 所借鉴。

4、提供了丰富的网络功能

各种版本 UNIX 普遍支持 TCP/IP 协议，并已成为 UNIX 系统与其它 OS 之间联网的最基本的选择。在 UNIX 中包括了网络文件系统 NFS 软件，客户/服务器协议软件 Lan Manager Client/Server、IPX/SPX 软件等。通过这些产品可以实现在 UNIX 系统之间、UNIX 与 Novell 的 Netware、MS-Windows NT、IBM LAN Server 等网络之间的互联和互操作。

5、安全性

UNIX 至少提供二道安全防线。一是在系统登录时要求提供合法的注册名和口令字，二是 UNIX 根据用户的注册名控制用户对文件及对系统服务的存取权限控制。

三、UNIX 系统核心的结构

整个 UNIX 系统分三个层次：

第三层：SHELL，用户接口 (命令接口、程序接口、图形用户接口)

第二层：内核，对对象控制和管理的软件集合 (处理机管理软件、存储器管理软件、设备管理软件、文件管理软件)
第一层：硬件，操作系统对象 (处理机、存储器、设备、文件和作业)

最低层是硬件，作为整个系统的基础。次低层是 OS 核心，包括教材所介绍的四大资源管理功能。最高层是 OS 与用户的接口 Shell 以及编译程序等。

1、内核。作为 OS 的核心，它应具有二个方面的接口：一方面是核心与硬件的接口，它通常是由一组驱动程序和一些基本的例行程序组成；二是核心与 Shell 的接口，由二组系统调用以及命令解释程序等组成。核心本身又可分成二大部分：一部分是进程控制子系统；另一部分则是文件子系统。二组系统调用分别与这二大子系统交互。

2、Shell 是用户与系统交互作用的界面。UNIX 中，Shell 作为解释程序出现：接收用户打入的命令，进行分析，创建子进程，由子进程实现命令所规定功能，等子进程终止工作后，发出提示符。这也是 Shell 最常见的使用方式。

Shell 除了作为命令解释程序以外，还是一种高级程序设计语言，它有变量、关键字、各种控制语句，如 if、case、while、for 等，有自己的语法结构。利用 Shell 程序设计语言可以编写出功能强大、代码简单的程序，特别是它把相关的 LINUX 命令有机地组合在一起，可大大提高编程的效率。

Shell 具有如下突出特点：

（1）把已有命令进行适当组合，构成新的命令，且组合方式简单；

（2）可以进行交互式处理，用户和 LINUX 系统之间通过 Shell 进行交互式会话，实现通信；

（3）灵活地利用位置参数传递参数值；

（4）结构化的程序模块，提供了顺序流程控制、条件控制、循环控制等；

（5）提供通配符、I/O 重定向、管道线等机制，方便了模式匹配、I/O 处理和数据传输；

（6）便于用户开发新的命令。利用 Shell 过程可把用户编写的可执行程序与 LINUX 命令结合在一起，当作新命令使用；

（7）提供后台处理方式，不打断前台工作。

UNIX 通常提供三种不同的 Shell，即 Bourne Shell （简称 bash），C-Shell （简称 csh）和 Korn Shell （简称 ksh）。Bourne Shell 是 AT&T Bell 实验室人员为 UNIX 开发的，它是其他 Shell 开发的基础，也是各种 UNIX/LINUX 系统上最常用、最基本的 Shell。C-Shell 是加州伯利克大学的 Bill Joy 为 BSD UNIX 开发的，它与 bash 不同，主要模拟 C 语言。ksh Shell 是 AT&T Bell 实验室开发的，与 bsh 兼容，但功能更强大。

3、Shell 程序示例

使用 Shell 最简单的方法是从键盘上直接打入命令行。例如：

```
ls  -l  /usr/example
```

Shell 命令解释程序对打入的命令进行分析，并创建子进程，完成该命令所对应的功能。Shell 程序也可存放在文件上，下面是二个 Shell 程序示例。

例如：由三条简单命令组成的 Shell 程序（文件名为 ex1）

```
date
pwd
cd..
```

执行这个 Shell 程序时，依次执行其中各条命令：显示日期、当前工作目录，返回上级目录。

四、什么是 LINUX

由于 UNIX 庞大的支持基础和发行系统，它成为世界范围内最有影响的 OS 之一。但另一方面，由于商业版 UNIX 非常昂贵，且源代码有专利，所以很难在计算机爱好者中广泛使用。于是，出现了这样一群人，他们是一支由编程高手、业余计算机玩家、黑客们组成的奇怪队伍，完全独立地开发出在功能上毫不逊色于商业版 UNIX OS 的一个全新的免费 UNIX OS——LINUX。

LINUX 是芬兰赫尔辛基大学的 Linus Torvalds 于 1991 年开始开发的。LINUX 是一个遵循 POSIX 标准的免费 OS，具有 BSD 和 SYSV 的扩展特性（其外表性能上同 UNIX 非常相象，但所有系统核心代码全部重新写过了）。

LINUX 可以运行在 X86 PC、SUN Sparc、Digital Alpha、PowerPC、MIPS 等平台上，可以说是目前运行硬件平台最多的 OS。

LINUX 上可运行大多数 UNIX 程序：X-Windows 系统、GNU C/C++编译器……。如今越来越多的商业公司采用 LINUX 作为 OS，例如，科学工作者使用 LINUX 进行分布式计算，ISP 使用 LINUX 配置 Internet 服务器、电话拨号服务器来提供网络服务；美国 1998 年 1 月最卖座的影片《泰坦尼克》中的计算机动画设计工作就是在 LINUX 平台上进行的。

LINUX 与 UNIX 有着密不可分的关系。实际上 LINUX 就是 UNIX 的克隆，只不过 LINUX 一般用于 PC 机，而大多数商业 UNIX 则主要用于工作站或大型机。

五、LINUX 的特点

1、支持多种硬件平台。它支持几乎所有的兼容芯片；

2、支持多种文件系统。

如 FAT、FAT32、EXT2、NFS 等，LINUX 可以将这些文件系统直接装载为系统的一个目录。LINUX 自己的文件系统 EXT2 非常先进，最多可支持到 2TB 的空间，文件名长度可达到 255 个字符。LINUX 可直接读写 DOS/WINDOWS9X 的 FAT 及 FAT32 文件系统，新的内核还支持直接读写 Windows NT 的 NTFS 文件系统。同时在 DOS 和 Windows95/NT 下也都有工具来直接读取 LINUX 文件系统上的文件；

3、多任务、多用户；

4、使用分页技术的虚拟内存。在 LINUX 下，系统核心并不把整个进程交换到硬盘上，而是按照内存页面来交换。虚拟内存的载体不仅可以是一个单独的分区，也可以是一个文件（如果用户在同时使用 Windows，LINUX 还可以同它们共享同一个交换文件，这是对硬盘紧张的用户的一个非正式的解决办法）。LINUX 还可以在系统运行时临时增减交换内存；

5、具有优秀的磁盘缓冲调度功能；

LINUX 最突出的一个优点就是它的磁盘 I/O 速度，因为它将系统没有用到的剩余物理内存全部用来作硬盘的高速缓冲，当有对内存要求比较大的应用程序运行时，它将会自动地将这部分内存释放出来给应用程序使用；

6、动态链接共享库。

同 Windows 的 DDL 一样，LINUX 也使用动态链接共享库（同时也提供静态链接库）。这个特性可以大大减小 LINUX 应用程序的大小。例如，一个普通的应用程序如果使用动态库，其程序大小只有 50KB 左右，但一旦在编译时改成静态链接，则该应用程序的大小将急增到 2MB。动态链接共享库是在程序运行时才动态链接的，并且被很多程序同时调用的一段代码只被加载一次，由众多程序共享；

7、丰富的软件；

8、软件移植性好（与其他 UNIX 系统的兼容性好）；

9、强大的网络功能。

LINUX 本身就是在 Internet 上成长起来的，所以它提供了全面的网络支持，如基本的

TCP/IP 网络、HTTP、FTP、NFS、E-Mail、UUCP 等；

10、提供全部源代码。

LINUX 最后也是最大的优点就是它的全部源代码都是公开的，这包括整个系统核心、所有的驱动程序、开发工具以及所有的应用程序。任何人只要有兴趣都可以将整个 LINUX 重新编译一遍。用户可以在 LINUX 的源代码中观察系统核心的运转，查看 Telnet、FTP 是如何实现的。整个 LINUX 对于用户就象是一个透明的发动机。

附录二　　文件系统参考程序

1 头文件 systemdata.h 定义了程序所需的结构，变量，函数

```
// 磁盘文件的参数
#define DISKSIZE        1024*1024        // 磁盘文件大小
#define BLOCKSIZE       512              // 磁盘块大小
#define FILENAME        "disk.txt"
// 超级块参数
#define SUPBLOCK        5                // 超级块所需盘块
#define bSTACKSIZE      50               // 空闲盘块栈大小
#define iSTACKSIZE      500              // 空闲 i 接点栈大小
#define SUPSTART        BLOCKSIZE        // 超级块起始的物理地址
#define SUPSIZE         sizeof(struct superblock)                    // 超级块大小
// i 结点参数
#define INODESIZE       sizeof(struct dinode)                        // 一个 i 结点大小
#define INODENUM_B      5                // 每个盘块 i 结点个数
#define INODEBLOCK      100              // i 结点所需盘块
#define   INODENUM      500              // i 结点数目
#define ADDRNUM         13               // i 结点可连接的盘块数
#define INODESTART      6*BLOCKSIZE      // i 界点起始的物理地址
// 用户密码区参数
#define USERSIZE        20               // 用户名长度
#define PWSIZE          30               // 密码长度
#define GROUPSIZE       20               // 组名长度
#define USERNUM         68               // 最多用户个数(其实比最大数多几个)
#define USERBLOCK       14               // 用户密码去所需盘块
#define PSWORDSTART     111*BLOCKSIZE    // 用户密码区起始地址
#define PSWORDSIZE      sizeof(struct psword)                        // 密码记录项长度
// 主用户目录区参数（特殊的文件目录，初始化就确定）
#define ROOTSTART       106*BLOCKSIZE    // /目录块地址
#define DIRBLOCK        4                // 主用户目录所需磁盘块数
#define DIRSTART        107*BLOCKSIZE    // 主用户目录起始地址
// 文件目录一般参数
#define DIRECTSIZE      sizeof(struct direct)                        // 目录项长度
#define DIRECTNUM       16               // 每块最多可分配目录项的数目
#define DIRECTSIZE_A    30               // 实际分配目录项长度
#define DIRSIZE         20               // 目录文件名长度
#define DIRNUM          (10*DIRECTNUM) // 最大子目录数(i 结点可连接 10 个盘块，每块可
放 17 项)
#define DATASTART       131              // 数据区开始地址
#define DATANUM         1917             // 数据区块数
#define HASHNUM         10               // HASH 表长度
#define DEFAULTMODE     00770            //默认权限
struct dinode{
    unsigned int      di_uid;            // 拥有该文件的用户
    unsigned char     di_gid;            // 拥有该文件的组 0：ROOT 1：USER
    unsigned int      di_mode;           // 存取权限
```

```
        unsigned int        di_addr[ADDRNUM];   // 文件物理地址
        unsigned long       di_size;            // 文件大小
        unsigned short      di_number;          // 文件连接数
        unsigned int        di_creattime;       // 文件创建时间
        unsigned int        di_visittime;       // 最近访问时间
};
struct inode{
        unsigned int        i_ino;              // 磁盘 i 结点标号
        unsigned short      i_uid;              // 拥有该文件的用户
        unsigned char       i_gid;              // 拥有该文件的组
        unsigned int        i_mode;             // 存取权限
        unsigned int        i_addr[ADDRNUM];    // 文件物理地址
        unsigned long       i_size;             // 文件大小
        unsigned short      i_number;           // 文件连接数
        unsigned int        i_creattime;        // 文件创建时间
        unsigned int        i_visittime;        // 最近访问时间
        unsigned char       i_flag;             // 上锁标记
        unsigned int        i_count;            // 引用计数
        struct      inode   *i_forw;            // 前向指针
        struct      inode   *i_back;            // 后向指针
};
struct superblock{
        unsigned short      s_isize;            // i 结点数目
        unsigned long       s_fsize;            // 数据块数目
        unsigned int        s_nfree;            // 空闲盘块数
        unsigned short      s_pfree;            // 空闲块指针
        unsigned int        s_free[bSTACKSIZE];  // 空闲块堆栈
        unsigned int        s_ninode;           // 空闲 i 结点数
        unsigned short      s_pinode;           // 空闲 i 结点指针
        unsigned int        s_inode[iSTACKSIZE];  // 空闲 i 结点堆栈
        unsigned int        s_rinode;           // 铭记 i 结点
        unsigned char       s_fmod;             // 修改标记(0：未修改，1 修改)
};
struct direct{
        char                name[DIRSIZE];      // 目录名
        unsigned int        d_ino;              // 目录对应的 i 结点号
        unsigned char       dir_flag;           // 1：目录；2：文件
};
struct dir{
        struct direct       direct[DIRNUM];     // 目录结构项
        unsigned int        size;               // 项的数目
};
struct psword{
        unsigned int        userid;             // 用户 id
        char                username[USERSIZE];     // 用户名
        char                password[PWSIZE];       // 用户密码
        char                group[GROUPSIZE];       // 用户所属组
};
struct psw{
```

```
    struct psword       psword[USERNUM];     // 用户记录表
    unsigned int        count;               // 用户记录个数
};
struct hinode{
    struct inode        *i_forw;             // hash 表头指针
};
// 全局变量定义如下：
extern struct superblock        superblock;          // 内存中超级块数据
extern FILE                     *fp;                 // 磁盘文件指针
extern struct hinode            hinode[HASHNUM];     // HASH 表
extern struct psw               thepsw;              // 用户表
extern struct direct            cur_direct;          // 当前的目录项
extern struct dir               cur_dir;             // 当前目录的子目录组
extern struct psword            cur_psword;          // 当前用户名密码
extern struct dir               users_dir;           // 用户文件表
// 函数定义
// 底层函数，主要操作磁盘文件
extern void             format();                    // 格式化文件系统
extern void             install();                   // 装载文件系统
extern struct dinode iget(int);                      // 根据 i 结点得到磁盘 i 结点
extern void             iput(struct dinode,int);     // 将磁盘 i 结点放入指定的 i 结点块中
extern int              ialloc();                    // 分配一个 i 结点，返回 i 结点号
extern void             ifree(int);                  // 将指定的 i 结点释放
extern int              balloc();                    // 分配一块磁盘块，返回盘块号
extern void             bfree(int);                  // 将指定的磁盘块释放
extern struct psw       psw_get();                   // 读 psw 信息
extern void             psw_put(struct psw);         // 写 psw 信息
extern void             psw_read();                  // 将 psw 中的用户信息读入内存 thepsw 中
extern void             psw_writeback();             // 将 thepsw 写回磁盘
extern struct dir       sub_dir_get(int);            // 根据 i 结点号，返回目录表
extern void             sub_dir_put(struct dir,int);
//extern void           sub_dir_read(int);           // 根据 i 结点号，将目录表读入 cur_dir
//extern void           sub_dir_writeback(struct dir,int);
                                                     // 将目录表放回指定的 i 结点子目录区
extern void             users_dir_writeback();       // 是上面函数的一个特例，放回的目录表是
                                                     // 用户目录表
//-------------------------------------------------------------
// 命令功能函数
extern bool             login(void);                 // 登录到文件系统
extern void             logout(void);                // 注销用户
extern bool             dir(void);                   // 列出目录
extern bool             cd(char *);                  // 改变目录
extern bool             mkdir(char *);               // 创建目录
extern bool             del(char *);                 // 删除目录或文件
extern bool             adduser(char *,char *);      // 增加用户
extern void             deluser(char *);             // 删除用户
extern bool             shell(void);                 // 命令解析
extern bool             chmod(char *);               // 改变文件权限
```

```
extern bool                pw(void);                          // 修改用户密码
extern bool                creatfile(char *);                 // 创建文件
extern int                 edit(char *);                      // 编辑文件
//--------------------------------
// 文件操作函数
// 函数 finished by murphydai
//extern bool               delfile(char *);
extern int                 fileopen(char *);                                  // 打开文件
extern bool                fileclose(unsigned int);                           // 关闭文件
extern int                 fileread(unsigned int,char *,int);                 // 读文件
extern int                 filewrite(unsigned int ,char*,int);                // 写文件
extern bool                access(unsigned int ,struct dinode );              // 文件访问控制
extern void inithash();
extern void addinode(int);
int    delinode(unsigned int );
struct inode* inodesearch(unsigned int );
```

2．主程序 main.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "systemdata.h"
  FILE                      *fp;                    // 磁盘文件指针
  struct superblock         superblock;             // 内存中超级块数据
  struct hinode             hinode[HASHNUM];    // HASH 表
  struct psw                thepsw;                 // 用户表
  struct direct             cur_direct;             // 当前的目录项
  struct dir                cur_dir;                // 当前目录的子目录组
  struct psword             cur_psword;             // 当前用户名密码
  struct dir                users_dir;              // 用户文件表
void main()
{
       format();
       if(!(fp=fopen(FILENAME,"r+b")))
       {
            cout<<"磁盘文件打开出错!!!";
            exit(0);
       }
     install();
     shell();
     fclose(fp);
}
```

3．Install.cpp 装载文件系统

```
#include <stdio.h>
#include "systemdata.h"
#include "hash.h"
struct superblock sup_get()
{
       struct superblock supb;
       fseek(fp,SUPSTART,SEEK_SET);
       fread(&supb,SUPSIZE,1,fp);
```

```
	return (supb);
}
void sup_put(struct superblock supb)
{
	fseek(fp,SUPSTART,SEEK_SET);
	fwrite(&supb,SUPSIZE,1,fp);
}
void install()
{
	struct dir dir;
	superblock=sup_get();
	thepsw=psw_get();
	dir=sub_dir_get(1);
	cur_direct=dir.direct[2];
	cur_dir=sub_dir_get(cur_direct.d_ino);
	users_dir=sub_dir_get(2);
	inithash();

}
```

4．Format.cpp 格式化

```
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "systemdata.h"
void format()
{
	char * buf;
	buf=(char *)malloc(DISKSIZE);
	if(!(fp=fopen(FILENAME,"w+b"))) exit(0);
	//开辟 1M 大小的文件做磁盘
	fwrite(buf,DISKSIZE,1,fp);
	fclose(fp);
	if(!(fp=fopen(FILENAME,"r+b"))) exit(0);
	//将 1--3 号 i 结点初始化
	struct dinode addinode;
	addinode.di_addr[0]=106;
	addinode.di_addr[1]=0;
	addinode.di_gid='r';
	addinode.di_uid=0;
	addinode.di_size=0;
	addinode.di_mode=DEFAULTMODE;
	iput(addinode,1);
	addinode.di_addr[0]=107;
	addinode.di_addr[1]=108;
	addinode.di_addr[2]=109;
	addinode.di_addr[3]=110;
	addinode.di_addr[4]=0;
	addinode.di_gid='r';
	addinode.di_uid=0;
	addinode.di_size=0;
	addinode.di_mode=DEFAULTMODE;
```

```
 iput(addinode,2);
 addinode.di_addr[0]=131;
 addinode.di_addr[1]=0;
 addinode.di_gid='r';
 addinode.di_uid=1;
 addinode.di_size=0;
 addinode.di_mode=DEFAULTMODE;
iput(addinode,3);
//根目录初始化
 struct dir dir;
 dir.size=3;
 dir.direct[0].d_ino=1;
 dir.direct[0].dir_flag='1';
 strcpy(dir.direct[0].name,".");
 dir.direct[1].d_ino=1;
 dir.direct[1].dir_flag='1';
 strcpy(dir.direct[1].name,"..");
 dir.direct[2].d_ino=2;
 dir.direct[2].dir_flag='1';
 strcpy(dir.direct[2].name,"/");
 sub_dir_put(dir,1);
 //主用户文件目录初始化
 dir.size=3;
 dir.direct[0].d_ino=2;
 dir.direct[0].dir_flag='1';
 strcpy(dir.direct[0].name,".");
 dir.direct[1].d_ino=1;
 dir.direct[1].dir_flag='1';
 strcpy(dir.direct[1].name,"..");
 dir.direct[2].d_ino=3;
 dir.direct[2].dir_flag='1';
 strcpy(dir.direct[2].name,"root");
 sub_dir_put(dir,2);
//root 用户目录初始化
 dir.size=2;
 dir.direct[0].d_ino=3;
 dir.direct[0].dir_flag='1';
 strcpy(dir.direct[0].name,".");
 dir.direct[1].d_ino=2;
 dir.direct[1].dir_flag='1';
 strcpy(dir.direct[1].name,"..");
 sub_dir_put(dir,3);
  //root 用户名密码初始化
  struct psword addpsword;
   struct psw        addpsw;
  addpsword.userid=1;
   strcpy(addpsword.username,"root");
   strcpy(addpsword.password,"root");
   strcpy(addpsword.group,"root");
   addpsw.count=1;
   addpsw.psword[0] =addpsword;
   psw_put(addpsw);
   //超级块初始化
```

```
    struct superblock initsupblock;
    initsupblock.s_isize=500;
    initsupblock.s_fsize=DATANUM;
    int i,j,begin,end,data[bSTACKSIZE],instack,freenum;
    instack=(DATANUM-1) % bSTACKSIZE;                    // 在堆栈中的块数
    freenum=(DATANUM-1) / bSTACKSIZE;                    // 存在数据区的组数
    begin=end=DATASTART+1;
    for(i=0;i<instack;i++)
    {
        initsupblock.s_free[i]=end;
        end++;
    }
    for(j=0;j<freenum;j++)
    {
//      cout<<begin<<endl;
        for(i=0;i<bSTACKSIZE;i++)
        {
            data[i]=end;
           // cout<<end<<" ";
            end++;
        }
        fseek(fp,begin*BLOCKSIZE,SEEK_SET);
        fwrite(data,sizeof(data),1,fp);
        begin=data[0];
    }
    initsupblock.s_nfree=DATANUM-1;
    initsupblock.s_pfree=instack-1;
    initsupblock.s_ninode=INODENUM-4;
    for(i=0;i<INODENUM-4;i++)
        initsupblock.s_inode[i]=i+4;
    initsupblock.s_pinode=INODENUM-5;
    initsupblock.s_rinode=0;
    initsupblock.s_fmod=0;
    fseek(fp,SUPSTART,SEEK_SET);
    fwrite(&initsupblock,SUPSIZE,1,fp);
    fclose(fp);
}
```

5 用户管理 USER.CPP 添加用户，删除用户

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include "systemdata.h"
unsigned int generateid()
{
    unsigned int i,id,temp=0;
    for(i=0;i<thepsw.count;i++)
      if(temp<thepsw.psword[i].userid)     temp=thepsw.psword[i].userid;
    id=temp+1;
    return id;
}
```

```
bool adduser(char name[],char group[])
{
	unsigned int i;
	int j;
	if(strcmp(cur_psword.username,"root")) //当前目录项假如是根，则添加用户
	{
		printf("您不能添加新用户，只有 root 才有添加用户的权限!\n");
		return false;
	}
	if(!strcmp(name,"root"))
	{
		printf("不能添加 root 用户!\n");
		return false;
	}
	for(i=0;i<thepsw.count;i++)
	if(!strcmp(thepsw.psword[i].username,name))
	{
		printf("该用户已存在!\n");
		return false;
	}
	char pwd[20],passwordok[20];
	printf("密码: ");
	for(int z=0;(pwd[z]=getch())!='\r';z++) putch('*');
	pwd[z]='\0';
	putch('\n');
	printf("重新输入一次:");
	for(z=0;(passwordok[z]=getch())!='\r';z++) putch('*');
	passwordok[z]='\0';
	putch('\n');
	if(strcmp(pwd,passwordok))
	{
		printf("两次输入密码不一样，添加用户失败!!!\n");
		return false;
	}
	 unsigned    int userid;
	 userid=generateid();//产生用户 id
	 j=thepsw.count;
	 thepsw.count+=1;
	 thepsw.psword[j].userid=userid;
	 strcpy(thepsw.psword[j].username,name);
	 //puts(thepsw.psword[j].username);
	 strcpy(thepsw.psword[j].password,pwd);
	 strcpy(thepsw.psword[j].group,group);//在 thepsw 中添加新的用户内容
	 int ni,nb;
	 ni=ialloc();
	 nb=balloc();//分配一个 i 节点和一个磁盘块
	 struct dinode di_new;
	 di_new=iget(ni);
	 di_new.di_uid=userid;
	 di_new.di_gid='0';
	 di_new.di_mode=DEFAULTMODE;
	 di_new.di_addr[0]=nb;
```

```
        di_new.di_addr[1]=0;
        di_new.di_size=0;
        di_new.di_number=0;
        di_new.di_creattime=0;
        di_new.di_visittime=0;//初始化磁盘 i 节点
                    j=users_dir.size;
                     strcpy(users_dir.direct[j].name,name);
                     users_dir.direct[j].dir_flag='1';
                     users_dir.direct[j].d_ino=ni;
                     users_dir.size++;
                     if(!strcmp(cur_direct.name,"/"))
                     cur_dir=users_dir;
                    struct dir buf_dir;
                    buf_dir.size=2;
                     strcpy(buf_dir.direct[0].name,".");
                     buf_dir.direct[0].d_ino=ni;
                     buf_dir.direct[0].dir_flag='1';
                     strcpy(buf_dir.direct[1].name,"..");
                     buf_dir.direct[1].d_ino=2;
                     buf_dir.direct[1].dir_flag='1';
                     psw_writeback();
                     sub_dir_put(users_dir,2);
                    iput(di_new,ni);
                    sub_dir_put(buf_dir,ni); //  将目录表放回指定的 i 结点子目录
                     printf("成功添加了新用户!\n");
                    return true;
}
void deluser(char name[])
{
      if(strcmp(cur_psword.username,"root"))       printf("您没有删除用户的权限，只有 root 才能删除用户!\n");
      else
      {     if(!strcmp(name,"root"))     printf("不能删除 root 用户!\n");
            else
             {
                  char yes_or_no;
                  cout<<"的确要删除此用户吗?(Y/N):";
                  cin>>yes_or_no;
                  if(yes_or_no=='Y' || yes_or_no=='y')
                  {
                      unsigned int i;
                      int flag=0;
                      for(i=0;i<thepsw.count;i++)
                       if(!strcmp(thepsw.psword[i].username,name))
                       {
                             flag=1;
                             break;
                       }
                       if(!flag)
                         printf("该用户不存在!\n");
                       else
                       {       unsigned int j,ni;
                               struct dir new_dir;
```

```
                    for(j=0;j<cur_dir.size;j++)
                        if(!strcmp(users_dir.direct[j].name,name))
                        {
                          ni=users_dir.direct[j].d_ino;
                          new_dir=sub_dir_get(ni);
                          break;
                        }
                     if(new_dir.size>2)
                      printf("用户目录非空，无法删除!\n");
                    else
                     {
                        struct dinode node;
                        node=iget(ni);
                        int k=0;
                       while(node.di_addr[k]>0 && k<10)
                        {
                            bfree(node.di_addr[k]);
                            k++;
                        }
                       ifree(ni);
                        users_dir.size--;
                        users_dir.direct[j]=users_dir.direct[users_dir.size];
                        sub_dir_put(users_dir,2);
                        if(!strcmp(cur_direct.name,"/"))
                                cur_dir=users_dir;
                        thepsw.count--;
                        thepsw.psword[i]=thepsw.psword[thepsw.count];
                        psw_put(thepsw);
                       printf("成功删除了一个用户\n");
                }
}}}}}
```

6 修改用户密码 PW.CPP

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "systemdata.h"
bool pw()
{
    unsigned int i,flag=0;
    char password[PWSIZE],passwordok[PWSIZE];
    printf("新密码: ");
    for(int z=0;(password[z]=getch())!='\r';z++) putch('*');
    password[z]='\0';
    putch('\n');
    printf("重新输入一次:");
    for(z=0;(passwordok[z]=getch())!='\r';z++) putch('*');
    passwordok[z]='\0';
    putch('\n');
    if(strcmp(password,passwordok))
    {
        printf("两次输入密码不一样，修改失败!!!\n");
        return false;
    }
```

```
    for(i=0;i<thepsw.count;i++)
        if(!strcmp(thepsw.psword[i].username,cur_psword.username))
        {
            flag=1;
            break;
        }
    if(!flag)
      {   printf("用户没有登录!!!\n");   return false;   }
    strcpy(thepsw.psword[i].password,password);
    psw_writeback();
    printf("密码修改成功!!!\n");
    return true;
}
```

7 命令解释 shell.cpp

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <conio.h>
#include "systemdata.h"
#define EXIT 10
#define LOGOUT 9
#define CMDNUM 13
int get_cmd_id(char *cmd)
{
    char *optable[]={"dir","ls","cd","mkdir","del","adduser",\
                     "deluser","creat","edit","chmod","pw","logout","exit"};
    int i,flag=0;
    for(i=0;i<CMDNUM;i++)
    {
        if(!strcmp(cmd,optable[i])){ flag=1; break; }
    }
    if(flag) return i;
    else{
        cout<<"没有这个命令!!!"<<endl;
        return -1;
    }
}
int docmd(char *cmd)
{
    int opp=0,parp=0,cmd_id;
    char op[20],par[20];
    while(*cmd==' ')   cmd++;
    while(*cmd!=' ' && *cmd!='\0')
        {
            op[opp]=*cmd;
            cmd++;
            opp++;
        }
        op[opp]='\0';
    while(*cmd==' ') cmd++;
    while(*cmd!='\0')
        {
            par[parp]=*cmd;
```

```
                // cout<<*cmd;
                  cmd++;
                  parp++;
              }
              par[parp]='\0';
             // cout<<par;
              cmd_id=get_cmd_id(op);
              switch(cmd_id)
              {
                    case 0:
                    case 1: dir();                                          break;
                    case 2: cd(par);                                        break;
                    case 3: mkdir(par);                                     break;
                    case 4: del(par);                                       break;
                    case 5: adduser(par,"user");                            break;
                    case 6: deluser(par);                                   break;
                    case 7: creatfile(par);                                 break;
                    case 8: edit(par);                                      break;
                    case 9: chmod(par);                                     break;
                    case 10: pw();                                          break;
                    case 11: printf("用户已注销!!!\n");          return LOGOUT;
                    case 12: cout<<"已退出文件系统!!!\n"; return EXIT;
                    default: return 0;
              }
        return 1;
}
void printinfor()
{
    printf("\t|-------------------------------------------------------|\n");
    printf("\t|                                                       |\n");
    printf("\t|                  文件系统(VFS)实例                    |\n");
    printf("\t|                                                       |\n");
    printf("\t|                  2002 - 7 - 9                       |\n");
    printf("\t|-------------------------------------------------------|\n\n");
}
bool shell()
{
    char cmd[20];
    int result;
start:
    printinfor();
    while(!login());
    do {
         printf("[%s @ localhost %s] ",cur_psword.username,cur_direct.name);
         gets(cmd);
         result=docmd(cmd);
        }while (result!=LOGOUT && result!=EXIT);
    if(result==LOGOUT) goto start;
    return true;
}
```

8 读出和写回用户信息表 psw_putget.cpp

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

```
struct psw psw_get()
{
       int i,cout;
       struct psw newpsw;
       fseek(fp,PSWORDSTART,SEEK_SET);
       fread(&cout,sizeof(cout),1,fp);
       newpsw.count=cout;
       for(i=0;i<cout;i++)
       {
              fseek(fp,PSWORDSTART+(i+1)*PSWORDSIZE,SEEK_SET);
              fread(&newpsw.psword[i],PSWORDSIZE,1,fp);
       }
    return (newpsw);
}
void psw_put(struct psw addpsw)
{
       int i,cout;
       cout=addpsw.count;
       fseek(fp,PSWORDSTART,SEEK_SET);
       fwrite(&cout,sizeof(cout),1,fp);
       for(i=0;i<cout;i++)
       {
              fseek(fp,PSWORDSTART+(i+1)*PSWORDSIZE,SEEK_SET);
              fwrite(&addpsw.psword[i],PSWORDSIZE,1,fp);
       }
}
void psw_writeback()
{
    psw_put(thepsw);
}
void psw_read()
{
       thepsw=psw_get();
}
```

9 建立和删除目录 mkdir_del.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "systemdata.h"
bool mkdir(char *dirname)
{
       int flag=0,newp,newdinode;
       unsigned int i;
       struct dinode node;
       struct dir dir;
       // /目录下不能建目录
       if(!strcmp(cur_direct.name,"/"))
       {
             printf("你不能随便在/目录下建目录!!!\n");
              return false;
       }
       //  检查是否在自己的目录下，不是则不能建立目录
```

```
    struct dinode dd_i;
    dd_i=iget(cur_direct.d_ino);
    if(dd_i.di_uid!=cur_psword.userid && strcmp(cur_psword.group,"root"))
    {
        printf("这不是你的家目录，你没有权限建立目录!!!\n");
        return false;
    }   // 检查重名
    for(i=0;i<cur_dir.size;i++)
    {
        if(!strcmp(dirname,cur_dir.direct[i].name)
            && cur_dir.direct[i].dir_flag=='1')
        {
            flag=1;
            break;
        }
    }
    if(flag)
    {
        puts("存在重名目录，请换一个名字!!!");
        return false;
    }
    // 分配 i 结点建立目录
    newdinode=ialloc();
    if(!newdinode)
    {
        puts("i 结点分配失败!!!");
        return false;
    }
    newp=cur_dir.size;
    cur_dir.size++;
    strcpy(cur_dir.direct[newp].name,dirname);
    cur_dir.direct[newp].dir_flag='1';
    cur_dir.direct[newp].d_ino=newdinode;
    node=iget(newdinode);
    node.di_gid='1';
    node.di_mode=DEFAULTMODE;
    node.di_size=0;
    node.di_uid=cur_psword.userid;
    iput(node,newdinode);
    dir.size=2;
    strcpy(dir.direct[0].name,".");
    dir.direct[0].dir_flag='1';
    dir.direct[0].d_ino=newdinode;
    strcpy(dir.direct[1].name,"..");
    dir.direct[1].dir_flag='1';
    dir.direct[1].d_ino=cur_direct.d_ino;
    sub_dir_put(dir,newdinode);                    // 将子目录表写回
    sub_dir_put(cur_dir,cur_direct.d_ino);         // 将当前目录表写回
    return true;
}
bool del(char *dirname)
{
    unsigned i;
```

```
int flag=0;
if(!strcmp(cur_direct.name,"/"))
{
      printf("用户目录不能随便删除!!!\n");
       return false;
}
// 检查是否在自己的目录下，不是则不能删除目录
struct dinode dd_i;
dd_i=iget(cur_direct.d_ino);
if(dd_i.di_uid!=cur_psword.userid && strcmp(cur_psword.group,"root"))
{
       printf("这不是你的家目录，你没有权限删除文件或目录!!!\n");
       return false;
}     // 检查是否存在该文件或目录
for(i=0;i<cur_dir.size;i++)
{
       if(!strcmp(dirname,cur_dir.direct[i].name))
       {
              flag=1;
              break;
       }
}
if(!flag)
{
       printf("没有你想要删除的文件!!!\n");
       return false;
}       // .和..目录不能删除
if(!strcmp(dirname,".") || !strcmp(dirname,".."))
{
       printf("不能删除此目录!!!\n");
       return false;
}
struct dinode node;
unsigned id,j=0;
char yes_or_no;
if(cur_dir.direct[i].dir_flag=='2')                // 如果是文件则作相应处理
{
       cout<<"的确要删除此文件吗?(Y/N):";
       cin>>yes_or_no;
       if(yes_or_no=='Y' || yes_or_no=='y')
       {
           id=cur_dir.direct[i].d_ino;
           node=iget(id);
           while(node.di_addr[j]>0 && j<10)
           {
                  bfree(node.di_addr[j]);
                  j++;
           }
           ifree(id);
           cur_dir.size--;
           cur_dir.direct[i]=cur_dir.direct[cur_dir.size];
           sub_dir_put(cur_dir,cur_direct.d_ino);
           return true;
```

```
            }
        }
        else if(cur_dir.direct[i].dir_flag=='1')                    // 是目录
        {
             id=cur_dir.direct[i].d_ino;
             struct dir dir;
             dir=sub_dir_get(id);
             if(dir.size>2)            // 有子目录,因为已经存在.和..目录，所以要大于 2
             {
                  puts("文件有子目录不能删除，请先删除子目录!!!");
                  return false;
             }
            cout<<"的确要删除此文件吗?(Y/N):";
            cin>>yes_or_no;
            if(yes_or_no=='Y' || yes_or_no=='y')
            {
                  node=iget(id);
                  bfree(node.di_addr[0]);
                 cur_dir.size--;
                  cur_dir.direct[i]=cur_dir.direct[cur_dir.size];
                  sub_dir_put(cur_dir,cur_direct.d_ino);         // 将新的文件目录写回
                  return true;
             }
        }
        return false;
}
```

10 用户登录 login_out.cpp

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include "systemdata.h"
bool login()
{
       unsigned int no,i,k;
       int flag=0;
       char username[USERSIZE],password[PWSIZE];
       printf("用户名: ");
       gets(username);
       for(i=0;i<thepsw.count;i++)
            if(!strcmp(thepsw.psword[i].username,username))
            {
                 flag=1;
                 no=i;
                 break;
            }
            if(!flag)
            {     printf("该用户不存在!\n");    return false;    }
            else
            {
                 printf("密码: ");
                 //gets(password);
                 for(int z=0;(password[z]=getch())!='\r';z++) putch('*');
```

```
                password[z]='\0';
                putch('\n');
                if(strcmp(thepsw.psword[no].password,password))
                {
                    printf("用户名和密码不匹配!\n");
                    return false;
                }
                else
                {
                   strcpy(cur_psword.group,thepsw.psword[no].group);
                  strcpy(cur_psword.username,thepsw.psword[no].username);
                   strcpy(cur_psword.password,thepsw.psword[no].password);
                   cur_psword.userid=thepsw.psword[no].userid;
                   for(k=0;k<cur_dir.size;k++)
                   {
                       if(!strcmp(cur_dir.direct[k].name,username))
                       {
                           cur_direct=cur_dir.direct[k];
                          cur_dir=sub_dir_get(cur_dir.direct[k].d_ino);
                           break;
                       }
                   }
                   puts("<恭喜，你已经成功登录>");
                   return true;
                }
            }
}
```

11 分配和回收 i 结点 i_alloc_free.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include "systemdata.h"
int ialloc()
{
    int ninode;
    struct superblock getblock;
    fseek(fp,SUPSTART,SEEK_SET);
    fread(&getblock,SUPSIZE,1,fp);
    if(getblock.s_ninode<=0)
    {
        cout<<"没有空闲 i 结点，请修改程序!!! ";
        return 0;
    }
    getblock.s_ninode--;
    ninode=getblock.s_inode[getblock.s_pinode--];
    fseek(fp,SUPSTART,SEEK_SET);
    fwrite(&getblock,SUPSIZE,1,fp);
    struct dinode node;
    for(int i=0;i<ADDRNUM;i++)
        node.di_addr[i]=0;
    iput(node,ninode);                    //将新分配的 i 节点 addr[]初始化为 0
    return (ninode);
}
```

```
void ifree(int i)
{
	struct superblock getblock;
	fseek(fp,SUPSTART,SEEK_SET);
	fread(&getblock,SUPSIZE,1,fp);
	if(getblock.s_ninode>=500)
	{
	    cout<<"系统错误：i 结点回收出错，请修改程序!!! ";
	    exit(0);
	}
	getblock.s_ninode++;
	getblock.s_inode[++getblock.s_pinode]=i;
	fseek(fp,SUPSTART,SEEK_SET);
	fwrite(&getblock,SUPSIZE,1,fp);
}
```

12 读出和写回指定 i 结点内容 i_getput.cpp

```
#include <stdio.h>
#include "systemdata.h"
struct dinode iget(int id)
{
	struct dinode newdinode;
	fseek(fp,long(INODESTART+id*INODESIZE),SEEK_SET);
	fread(&newdinode,INODESIZE,1,fp);
	return (newdinode);
}
void iput(struct dinode node,int id)
{
	fseek(fp,long(INODESTART+id*INODESIZE),SEEK_SET);
	fwrite(&node,INODESIZE,1,fp);
}
```

13 hash 表的相关操作 hash.cpp

```
#include "stdio.h"
#include "systemdata.h"
#include "malloc.h"
void inithash()
{
	int i;
	struct inode i_node;
	struct inode *pi_node;
	i_node.i_ino =0;
	i_node.i_forw =NULL;
	i_node.i_back =NULL;
	for(i=0;i<HASHNUM;i++)
	{
	    pi_node=(struct inode*)malloc(sizeof(struct inode));
	    *pi_node=i_node;
	    hinode[i].i_forw=pi_node;
	}                    // 10 linklist head
}
void addinode(int n)
{
	int i=n%10;
	int k=0,nbuf;
```

```
        struct inode *i_node=(inode *)malloc(sizeof(struct inode));
        struct inode *pi_node=hinode[i].i_forw;
        struct dinode di_node;
        di_node=iget(n);
        while((nbuf=di_node.di_addr [k])!=0)
        {
                i_node->i_addr [k]=nbuf;
                k++;
        }
        i_node->i_ino =n;
        i_node->i_addr [k]=0;
        i_node->i_back =NULL;
        i_node->i_count=1;
        i_node->i_creattime =di_node.di_creattime ;
        i_node->i_flag =0;
        i_node->i_forw =NULL;
        i_node->i_gid =di_node.di_gid;
        i_node->i_mode =di_node.di_mode ;
        i_node->i_number =di_node.di_number;
        i_node->i_size =di_node.di_size ;
        i_node->i_uid =di_node.di_uid ;
        i_node->i_visittime =di_node.di_visittime ;    // init inode
        pi_node=hinode[i].i_forw ;
        while(pi_node->i_forw !=NULL)
        {
                pi_node=pi_node->i_forw;
        }                                                       //search end
        pi_node->i_forw =i_node;
        i_node->i_back =pi_node;                                //insert a inode to hash list
}
int    delinode(unsigned int n)
{
        int i;
        i=n%10;
        struct inode* pinode;
        struct inode* ppinode;
        pinode=hinode[i].i_forw ;
        ppinode=pinode;
        while(pinode!=NULL&&pinode->i_ino!=n)
        {
                ppinode=pinode;
                pinode=pinode->i_forw;
        }
        if(pinode==NULL)
        {
                printf("该文件已经关闭！\n");
                return -1;
        }
        ppinode->i_forw =pinode->i_forw ;
        free(pinode);
        return n;
}
struct inode* inodesearch(unsigned int n)
{
```

```
    int i;
    i=n%10;
    struct inode* pinode;
    pinode=hinode[i].i_forw ;
    while(pinode!=NULL&&pinode->i_ino!=n)
    {
        pinode=pinode->i_forw;
    }
    if(pinode==NULL)
    {
        return NULL;
    }
    else return pinode;
}
```

14 文件操作 file.cpp

```
#include "stdio.h"
#include "systemdata.h"
#include "string.h"
#include "hash.h"
bool creatfile(char filename[])
{

    int filenum=cur_dir.size;
    int i,ni;
    struct dinode d_i;
    struct dinode dd_i;
    dd_i=iget(cur_direct.d_ino);
    if(strcmp(cur_direct.name,"/")==0)
    {
        printf("can not creat file here\n");
        return false;
    }
    for(i=0;i<filenum;i++)
    {
        if(cur_dir.direct [i].dir_flag =='2'&&strcmp(cur_dir.direct [i].name,filename)==0 )
        {
            printf("file already exist\n");
            return false;
        }
    }
    if(dd_i.di_uid !=cur_psword.userid &&strcmp(cur_psword.group ,"root")!=0)
    {
        printf("你无权在其他用户中创建文件\n");
        return false;
    }
    ni=ialloc();
    cur_dir.direct[filenum].d_ino =ni;
    cur_dir.direct[filenum].dir_flag ='2';
    strcpy(cur_dir.direct [filenum].name ,filename);
    cur_dir.size ++;                    //add item in cur_dir
    d_i=iget(ni);       //
    d_i.di_addr [0]=0;
    d_i.di_creattime =0;
```

```
        d_i.di_gid =(strcmp(cur_psword.group,"root")==0) ?'0':'1';
        d_i.di_mode =DEFAULTMODE;
        d_i.di_number =0;
        d_i.di_size =0;
        d_i.di_uid =cur_psword.userid ;
        d_i.di_visittime =0;                    //add and initiate dinode
        iput(d_i,ni);     //
       sub_dir_put(cur_dir,cur_direct.d_ino );   //当前目录回写
        return true ;
}
bool access(unsigned int m,struct dinode di_node)
{
              int mode=di_node.di_mode ;
              int j,one;
               int    buf[9];
               for(j=0;j<9;j++)
               {
                     one=mode % 2;
                     mode=mode / 2;
                     if(one) buf[j]=1;
                     else buf[j]=0;
               }
                if(buf[3+m])
                       return true;
                 else
                       return false;
}
int fileopen(char filename[])
{
       int i,filenum;
        unsigned int n_di;
        struct dinode di_node;
        filenum=cur_dir.size ;
        for(i=0;i<filenum;i++)
        {
              if(cur_dir.direct [i].dir_flag =='2'&&strcmp(cur_dir.direct [i].name,filename)==0 )
              break;
        }
        if(i==filenum)
        {
              printf("没有这个文件\n");
              return -1;
        }
        n_di=cur_dir.direct[i].d_ino ;      //find dinode number
        di_node=iget(n_di);
        if(strcmp(cur_psword.group,"root")!=0&&cur_psword.userid  !=di_node.di_uid  ) //not  root  or
owner
        {
              if(access(0,di_node)==false)
              {
                    printf("你没有权限打开\n");
                    return -1;                          //没有权限；
              }
```

```
        }
        if(inodesearch(n_di)==NULL)         //hash  表中不存在.
        {
                addinode(n_di);
                return n_di;
        }
        printf("文件已经打开\n");
        return -1;
}
bool fileclose(unsigned int dinode)
{
        int flag;
        flag=delinode(dinode);
        if(flag==-1)
                return false;
        return true;
}
int filewrite(unsigned int dinode,char *buf,int size)
{
        int i,j;
        struct inode* i_node;
        struct dinode di_node;
        di_node=iget(dinode);
        if(strcmp(cur_psword.group,"root")!=0&&cur_psword.userid !=di_node.di_uid ) //not root or
owner
        {
                if(access(2,di_node)==false)
                {
                        printf("没有权限写\n");
                        return -1;                          //没有权限；
                }
        }
        i_node=inodesearch(dinode);
        if(i_node==NULL)         //hash  表中不存在.
        {
                printf("文件没打开\n");
                return -1;              //文件没打开.
        }
        else
        {
                for(i=0;i<size/512&&i<ADDRNUM;i++)
                {
                        j=balloc();
                        fseek(fp,(long)(fp+512*j),SEEK_SET);
                        fwrite((buf+i*512),1,512,fp);
                        i_node->i_addr[i]=j;
                        di_node.di_addr[i]=j;
                }                   //整块       j 为块号
                if(i==ADDRNUM)
                        return 512*ADDRNUM;             //文件太大,剪断
                j=balloc();
                fseek(fp,(long)(512*j),SEEK_SET);
                fwrite((buf+i*512),1,size%512,fp);
```

```
            i_node->i_addr[i]=j;
            di_node.di_addr[i]=j;
           i++;
            i_node->i_addr[i]=0;
            di_node.di_addr[i]=0;                    //零头
        }                                          //分配磁盘块,给 inode->addr,dinode.addr 赋值
        i_node->i_size=size;
       i_node->i_count=1;
        i_node->i_flag =0;
        i_node->i_number =1;
        i_node->i_visittime =0;
        di_node.di_number =1;
        di_node.di_size =size;
        di_node.di_visittime =0;
       iput(di_node,dinode);
        return size;
}
int fileread(unsigned int dinode,char *buf,int size)
{
        int i,j;
        struct inode* i_node;
        struct dinode di_node;
        di_node=iget(dinode);
        if(strcmp(cur_psword.group,"root")!=0&&cur_psword.userid !=di_node.di_uid ) //not root or
owner
        {
            if(access(1,di_node)==false)
            {
                printf("没有权限读\n");
                return -1;                          //没有权限；
            }
        }
        i_node=inodesearch(dinode);
        if(i_node==NULL)        //hash 表中不存在.
        {
            printf("文件没打开\n");
            return -1;              //文件没打开.
        }
        else
        {
            if(i_node->i_size ==0)
            {
                printf("该文件为空\n");
                return -1;
            }
            for(i=0;i<size/512;i++)
            {
                j=i_node->i_addr[i];
               fseek(fp,(long)(512*j),SEEK_SET);
                fread((buf+i*512),1,512,fp);
            }                                          //整块   j 为块号
            j=i_node->i_addr [i];
            fseek(fp,(long)(512*j),SEEK_SET);
```

```
            fread((buf+i*512),1,size%512,fp);      //零头
        }
        di_node.di_visittime =0;
        iput(di_node,dinode);
        i_node->i_visittime=0;
        return size;
}
```

15 一个简单的编辑器,实现输入读出功能 edit.cpp

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include "systemdata.h"
int size(char *ch)
{
        int i=0;
        while(ch[i]!='\0')
        {
                i++;
        }
        return ++i;
}
char *input()
{
        int k=0,j=0,i=1;
        char *buf=(char*)malloc(1024*i);
        char ch[512];
        while(ch[j]!='$')
        {
        gets(ch);
                j=0;
                while(ch[j]!='\0'&&ch[j]!='$')
                {
                        buf[k++]=ch[j++];
                }
                buf[k++]='\n';
                buf[k]='\0';
                if(size(buf)>=1024*i-512)
                {
                        i++;
                        buf=(char *)realloc(buf,512*i);
                }
        };
        buf[k]='\0';
        return buf;
}
int edit(char name[])
{
        char filename[DIRSIZE];
        char *readbuf,*writebuf;
        unsigned int n_di;
        int nn;
        int flag=1;
        int closeflag=1;
```

```
        char yn;
        struct dinode di_node;
        strcpy(filename,name);
        while(flag)
        {
            if((n_di=fileopen(filename))==-1)
            {
                printf("你想打开另一个文件吗？<Y>\n");
                scanf("%c",&yn);
                getchar();
                if(yn=='y'||yn=='Y')
                {
                    printf("输入文件名\n");
                    gets(filename);
                    flag=1;
                }
                else
                    return 0;
            }
            else flag=0;
        }
        closeflag=0;
        di_node=iget(n_di);
        do{
        printf("1: 读文件\n");
        printf("2: 写文件（这将丢失文件原数据）\n");
        printf("3: 关闭文件\n");
        printf("4: 退出编辑\n");
        printf("你想做什么？\n");
        scanf("%d",&nn);
        getchar();
        switch (nn)
        {
        case 1:
            di_node=iget(n_di);
            readbuf=(char *)malloc(di_node.di_size);
            if(fileread(n_di,readbuf,di_node.di_size )!=-1)
            {
               printf("%s",readbuf);
            }
            else printf("读入失败！\n");
            free(readbuf);
            break;
        case 2:
            if(strcmp(cur_psword.group,"root")!=0&&cur_psword.userid !=di_node.di_uid ) //not root
or owner
                {
                    if(access(2,di_node)==false)
                    {
                        printf("没有权限写\n");
                        break;                  //没有权限；
                    }
```

```
                }
                  printf("输入内容，以'$'结尾\n");
                  writebuf=input();
                if( filewrite(n_di,writebuf,size(writebuf))==-1)
                      printf("写人失败！\n");
                free(writebuf);
            break;
        case 3:
            if(fileclose(n_di)==true)
                closeflag=1;
            else
            {
                printf("无法关闭，可能不存在该文件，或该文件已经关闭！\n");
                closeflag=1;
            }

            break;
        case 4:
            if(closeflag==1)
            return 0;
            else
                printf("请先关闭文件,再退出！\n");
            break;
        default :
            printf("没有这个选项！\n");
            break;
        }                   //switch
        }                  //do
        while(1);
}//edit
```

16 读出，写入目录项 dir_putget.cpp

```
#include <stdio.h>
#include <iostream.h>
#include "systemdata.h"
struct dir sub_dir_get(int id)
{
    int i,j,count;
    struct dinode node;
    struct dir dir;
    dir.size=0;
    node=iget(id);
    for(i=0;i<10 && node.di_addr[i]>0;i++)
    {
            // 读出这一块存放的目录项数
            fseek(fp,node.di_addr[i]*BLOCKSIZE,SEEK_SET);
            fread(&count,sizeof(count),1,fp);

            // 读出目录项
            for(j=1;j<=count;j++)
            {
                fseek(fp,node.di_addr[i]*BLOCKSIZE+j*DIRECTSIZE_A,SEEK_SET);
                fread(&dir.direct[dir.size],DIRECTSIZE,1,fp);
```

```
                dir.size++;             // 读出一项则计数加一
            }
        }
        return (dir);
}
void sub_dir_put(struct dir dir,int id)
{
        int i,j,p=0,arraynum,left,directnum;
        struct dinode node;
        node=iget(id);
        directnum=DIRECTNUM;                    // 每块磁盘块能存放的目录项数目
        arraynum=dir.size / DIRECTNUM;          // dir 中目录项需要几块整块的磁盘块
        left=dir.size % DIRECTNUM;              // 零头项的数目
        // 完整块的存放（这些块都是放满的）
        for(i=0;i<arraynum;i++)
        {
            if(node.di_addr[i]==0) node.di_addr[i]=balloc();    // 没有盘块则分配一块
            // 将这一块的目录项数写入
            fseek(fp,node.di_addr[i]*BLOCKSIZE,SEEK_SET);
            fwrite(&directnum,sizeof(directnum),1,fp);
            // 写入目录项
            for(j=1;j<=directnum;j++,p++)
            {
                fseek(fp,node.di_addr[i]*BLOCKSIZE+j*DIRECTSIZE_A,SEEK_SET);
                fwrite(&dir.direct[p],DIRECTSIZE,1,fp);
            }
        }
        // 以下保存零头项
        if(node.di_addr[i]==0) node.di_addr[i]=balloc();      // 没有盘块则分配
        // 写入零头项数目
        fseek(fp,node.di_addr[i]*BLOCKSIZE,SEEK_SET);
        fwrite(&left,sizeof(left),1,fp);
        // 保存零头项
        for(j=1;j<=left;j++,p++)
        {
            fseek(fp,node.di_addr[i]*BLOCKSIZE+j*DIRECTSIZE_A,SEEK_SET);
            fwrite(&dir.direct[p],DIRECTSIZE,1,fp);
        }
        // 以下将没有用到的磁盘块释放
        i++;
        while(i<10)
        {
            if(node.di_addr[i])
            {
                bfree(node.di_addr[i]);
                node.di_addr[i]=0;
            }
            i++;
        }
        iput(node,id);
}
void sub_dir_writeback(struct dir dir,int id)
```

```
{
	sub_dir_put(dir,id);
}
```

17 改变路径,进入目录 dir_cd.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "systemdata.h"
bool dir(void)
{
	unsigned int i,j,one,di_mode;
	struct dinode node;
	if(cur_direct.dir_flag!='1')
	{
		cout<<"此文件不是目录文件!!!"<<endl;
		return false;
	}
	for(i=0;i<cur_dir.size;i++)
	{
		// 列出文件名或目录名
		if(cur_dir.direct[i].dir_flag=='1')
			cout<<"["<<cur_dir.direct[i].name<<"]\t\t";
		else cout<<cur_dir.direct[i].name<<"\t\t";
		node=iget(cur_dir.direct[i].d_ino);		// 得到相应的 i 结点，以便得到详细信息
		// 显示存取权限
		di_mode=node.di_mode;
		char buf[9];
		for(j=0;j<9;j++)
		{
			one=di_mode % 2;
			di_mode=di_mode / 2;
			if(one) buf[j]='x';
			else buf[j]='-';
		}
		for(int p=8;p>=0;p--) cout<<buf[p];
		cout<<"\t";
		// 如果是文件则列出大小
		if(cur_dir.direct[i].dir_flag=='1')
			cout<<"<direct>\t\t";
		else{
			cout<<node.di_size<<"\t\t";
		}
		// 列出文件，目录的创建者名字
		for(unsigned int c=0;c<thepsw.count;c++)
		{
			if(thepsw.psword[c].userid==node.di_uid) cout<<thepsw.psword[c].username<<"\t\t";
		}
		cout<<endl;
	}
	return true;
}
bool cd(char * cmdname)
```

```
{
    struct dinode node;
    struct dir dir;
    int flag=0;
    unsigned int i,j,id;
    for(i=0;i<cur_dir.size;i++)
    {
        if(!strcmp(cmdname,cur_dir.direct[i].name))
        {
           flag=1;
           break;
        }
    }
    if(flag){
        if(cur_dir.direct[i].dir_flag=='2')
        {
            cout<<"这不是目录!!!"<<endl;
            return false;
        }
        // 如果 cur_dir 是主用户目录文件，则作特殊处理
        // 全局变量都保持不变
        if(cur_dir.direct[i].d_ino==1)return true;
        // cur-dir 不是用户目录，则根据目录关系建立新的状态
        node=iget(cur_dir.direct[i].d_ino);
        id=cur_dir.direct[i].d_ino;
        cur_dir=sub_dir_get(id);
        dir=sub_dir_get(cur_dir.direct[1].d_ino);
        for(j=0;j<dir.size;j++)
        {
            if(dir.direct[j].d_ino==id) break;
        }
        cur_direct=dir.direct[j];
        return true;
    }
    else {
        cout<<"此文件不是目录文件或不存在此目录!!!"<<endl;
        return false;
    }

}
```

18 修改文件权限 chmod.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include "systemdata.h"
bool chmod(char * filename)
{
    unsigned int i,flag=0;
    // 检查是否在自己的目录下，不是则不能改变文件权限
    struct dinode dd_i;
    dd_i=iget(cur_direct.d_ino);
    if(dd_i.di_uid!=cur_psword.userid && strcmp(cur_psword.group,"root"))
    {
```

```
            printf("这不是你的家目录，你不能改变文件权限!!!\n");
            return false;
        }
        // 查看该目录下是否有该文件
        for(i=0;i<cur_dir.size;i++)
        {
            if(!strcmp(filename,cur_dir.direct[i].name))
            {
                flag=1;
                break;
            }
        }
        if(!flag)
        {
            cout<<"该目录下找不到该文件!!!"<<endl;
            return false;
        }
        if(cur_dir.direct[i].dir_flag=='1')
        {
            cout<<"目录不能改变权限!!!"<<endl;
            return false;
        }
        int newmod;
        printf("请输入新权限:");
        scanf("%o",&newmod);
        getchar();
        struct dinode sub_i;
        sub_i=iget(cur_dir.direct[i].d_ino);
        sub_i.di_mode=newmod;
        iput(sub_i,cur_dir.direct[i].d_ino);
        return true;
}
```

19 分配和回收磁盘块 b_alloc_free.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include "systemdata.h"
int balloc()
{
        struct superblock block;
        int addr,bindex[bSTACKSIZE];
        fseek(fp,SUPSTART,SEEK_SET);
        fread(&block,SUPSIZE,1,fp);                                    // 读出超级块内容
        if(block.s_nfree<=0)
        {
            cout<<"已没有空闲盘块可分配!!! ";
            return 0;
        }
        if(block.s_pfree==0 && block.s_nfree>bSTACKSIZE)     // 判断堆栈中空闲块是否分配完
        {
            addr=block.s_free[0];                                      // 将堆栈底块号读出，为被分配块
号
```

```
            fseek(fp,addr*BLOCKSIZE,SEEK_SET);
            fread(bindex,sizeof(bindex),1,fp);
            for(int i=0;i<bSTACKSIZE;i++)
                block.s_free[i]=bindex[i];                    // 将 addr 块中的数据读出，并存入
堆栈
            block.s_pfree=bSTACKSIZE-1;                       // 堆栈指针指向满的位置
        }
        else{                                                 // 堆栈中可以直接分配
                addr=block.s_free[block.s_pfree];
                block.s_nfree--;
                block.s_pfree--;

        }
        fseek(fp,SUPSTART,SEEK_SET);
        fwrite(&block,SUPSIZE,1,fp);                          // 将超级块写回磁盘文件
        return addr;
}

void bfree(int id)
{
        struct superblock block;
        int bindex[bSTACKSIZE];
        fseek(fp,SUPSTART,SEEK_SET);
        fread(&block,SUPSIZE,1,fp);
        if(block.s_pfree>=bSTACKSIZE-1)                       // 堆栈已经放满
        {
            for(int i=0;i<bSTACKSIZE-1;i++)
            {
                bindex[i]=block.s_free[i];
            }
            fseek(fp,DATASTART+id*BLOCKSIZE,SEEK_SET);
            fwrite(&bindex,sizeof(bindex),1,fp);
            block.s_nfree++;
            block.s_pfree=0;                                  // 堆栈指针指向堆栈底
            block.s_free[0]=id;
        }
        else{                                                 // 直接释放
            block.s_nfree++;
            block.s_pfree++;
            block.s_free[block.s_pfree]=id;
        }
        fseek(fp,SUPSTART,SEEK_SET);
        fwrite(&block,SUPSIZE,1,fp);                          // 将超级块写回磁盘文件
}
```