《操作系统 A》
实验指导书

倪福川 编

华中农业大学 信息学院

2016 年 9 月 V2.0

目录

预备实验一　熟悉操作系统环境.....2#
预备实验二　shell 脚本.....10#
实验一　进程控制.....14#
实验二　进程间通信.....18#
（一）信号量机制实验.....18#
（二）进程的管道通信实验.....24#
（三）消息的发送与接收实验.....27#
（四） 共享存储区通信.....33#
选作实验　死锁避免的算法.....39#
实验三　进程调度.....41#
实验四　存储管理.....54#
(一)常用页面置换算法.....54#
（二）动态分区分配算法（选做）.....63#
实验四：设备驱动实现.....64#
实验五　文件操作.....73#

预备实验一　熟悉操作系统环境

【实验目的】

1．了解 Ubuntu 系统基本操作方法，学会独立使用该系统。
2．熟悉 Ubuntu 下如何编辑、编译和运行一个 C 语言程序。
3．学会利用 gcc、gdb 编译、调试 C 程序。

【实验内容】

一、登陆 Linux

开机，在 Ubuntu 登陆窗口，输入用户名： user ，输入密码： 123456 ，进入 Ubuntu 图形桌面环境。

二、熟悉 Ubuntu 图形桌面环境

桌面包含上方的菜单栏和下方的任务栏。菜单栏包含“应用程序菜单”、“位置”、“系统”。

通过主菜单可访问应用程序。

(1) “应用程序”菜单

“应用程序”菜单包含“办公”、“附件”、“互联网”、“图形”、“游戏”等。

“办公”包含了文字处理及电子表格等应用程序。

“附件”下包含了“搜索工具”、“计算器”、“文本编辑器”和“终端”等应用程序。

(2)“位置”菜单

“位置”菜单包含“主文件夹”、“桌面”等信息。

(3)“系统”菜单

“系统”菜单包含“首选项”和“系统管理”等信息。

(4)启动终端模拟器

GNOME 终端模拟器用一个窗口来模拟字符终端的行为。终端常常被称为命令行或者 shell,Linux 中绝大部分工作都可以用命令行完成。要启动一个终端，可以选择 应用程序 → 附件 → 终端。

三、练习常用的 Shell 命令。（重点）

当用户登录到字符界面系统或使用终端模拟窗口时，就是在和称为 shell 的命令解释程序进行通信。当用户在键盘上输入一条命令时，shell 程序将对命令进行解释并完成相应的动作。这种动作可能是执行用户的应用程序，或者是调用一个编辑器、GNU/Linux 实用程序或其他标准程序，或者是一条错误信息，告诉用户输入了错误的命令。

1．目录操作

mkdir　abc	创建一个目录 abc
cd　 abc	将工作目录改变到 abc
cd	改变当前目录到主目录
ls	列出当前目录的内容
ls　-l	输出当前目录内容的长列表，每个目录或文件占一行

pwd　　显示当前目录的全路径

2．文件显示实用程序

cat　mx.c　　显示 mx.c 文件内容
more　mx.c　　分屏显示 mx.c 内容
tail　mx.c　　显示文件后几行
cat　file1　file2　　连接 file1 和 file2
head　filename　　显示文件 filename 的开始 10 行
wc　filename　　统计文件 filename 中的行数、单词数和字符数
od　文件　　查看非文本文件

3．文件管理实用程序

cp　file1　file2　　将文件 1 复制到文件 2
mv　file1　file2　　将文件重命名为 file2
rm　filename　　删除文件 filename
rm　-i　filename　　请求用户确认删除

4．数据操作实用程序

tty　　显示当前终端的路径和文件名
who　　显示当前登录用户的列表
sort　filename　　显示文件 filename 中的行的排序结果
spell　filename　　检查文件 filename 中的拼写错误

5．其他实用程序

date　　输出系统日期和时间
cal　　显示本月的日历。cal 2002 显示 2002 年的日历
clear　　清除终端屏幕
history　　显示你以前执行过的命令的列表
man　　显示实用程序的有用信息，并提供该实用程序的基本用法
echo　　读取参数并把它写到输出

四、目录和文件系统

Linux 和 Unix 文件系统被组织成一个有层次的树形结构。文件系统的最上层是 /，或称为 根目录。在 Unix 和 Linux 的设计理念中，一切皆为文件——包括硬盘、分区和可插拔介质。这就意味着所有其它文件和目录（包括其它硬盘和分区）都位于根目录中。 例如：/home/jebediah/cheeses.odt 给出了正确的完整路径，它指向 cheeses.odt 文件，而该文件位于 jebediah 目录下，该目录又位于 home 目录，最後，home 目录又位于根(/) 目录下。 在根 (/) 目录下，有一组重要的系统目录，在大部分 Linux 发行版里都通用。直接位于根 (/) 目录下的常见目录列表如下：

- /bin - 重要的二进制 (binary) 应用程序
- /boot - 启动 (boot) 配置文件
- /dev - 设备 (device) 文件
- /etc - 配置文件、启动脚本等 (etc)
- /home - 本地用户主 (home) 目录
- /lib - 系统库 (libraries) 文件

- /lost+found - 在根 (/) 目录下提供一个遗失+查找(lost+found) 系统
- /media - 挂载可移动介质 (media)，诸如 CD、数码相机等
- /mnt - 挂载 (mounted) 文件系统
- /opt - 提供一个供可选的 (optional) 应用程序安装目录
- /proc - 特殊的动态目录，用以维护系统信息和状态，包括当前运行中进程 (processes) 信息。
- /root - root (root) 用户主文件夹，读作“slash-root”
- /sbin - 重要的系统二进制 (system binaries) 文件
- /sys - 系统 (system) 文件
- /tmp - 临时(temporary)文件
- /usr - 包含绝大部分所有用户(users)都能访问的应用程序和文件
- /var - 经常变化的(variable)文件，诸如日志或数据库等

五．打开 PROC 目录了解系统配置

把/proc 作为当前目录，就可使用 ls 命令列出它的内容。

/proc 文件系统是一种内核和内核模块用来向进程 (process) 发送信息的机制 。这个伪文件系统让你可以和内核内部数据结构进行交互，获取有关进程的有用信息，在运行中改变设置 (通过改变内核参数)。 **与其他文件系统不同，/proc 存在于内存之中而不是硬盘上。**

1．察看 /proc 的文件

/proc 的文件可以用于访问有关内核的状态、计算机的属性、正在运行的进程的状态等信息。大部分 /proc 中的文件和目录提供系统物理环境最新的信息。尽管 /proc 中的文件是虚拟的，但它们仍可以使用任何文件编辑器或像'more', 'less'或 'cat'这样的程序来查看。

2．得到有用的系统/内核信息

/proc 文件系统可以被用于收集有用的关于系统和运行中的内核的信息。下面是一些重要的文件：

- /proc/cpuinfo - CPU 的信息 (型号, 家族, 缓存大小等)
- /proc/meminfo - 物理内存、交换空间等的信息
- /proc/mounts - 已加载的文件系统的列表
- /proc/devices - 可用设备的列表
- /proc/filesystems - 被支持的文件系统
- /proc/modules - 已加载的模块
- /proc/version - 内核版本
- /proc/cmdline - 系统启动时输入的内核命令行参数

proc 中的文件远不止上面列出的这么多。想要进一步了解的读者可以对 /proc 的每一个文件都'more'一下 。

3．有关运行中的进程的信息

/proc 文件系统可以用于获取运行中的进程的信息。在 /proc 中有一些编号的子目录。每个编号的目录对应一个进程 id (PID)。这样，每一个运行中的进程 /proc 中都有一个用它的 PID 命名的目录。这些子目录中包含可以提供有关进程的状态和环境的重要细节信息的文件。

/proc 文件系统提供了一个基于文件的 Linux 内部接口。它可以用于确定系统的各种不同设备和进程的状态。对他们进行配置。因而，理解和应用有关这个文件系统的知识是理解你的 Linux 系统的关键。

六、熟悉 vim 编辑器

在编写文本或计算机程序时，需要创建文件、插入新行、重新排列行、修改内容等，计算机文本编辑器就是用来完成这些工作的。

Vim 编辑器的两种操作模式是命令模式和输入模式（如图 2 所示）。当 vim 处于命令模式时，可以输入 vim 命令。例如，可以删除文本并从 vim 中退出。在输入模式下，vim 将把用户所输入的任何内容都当作文本信息，并将它们显示在屏幕上。

vi 的工作模式见图 2 所示。

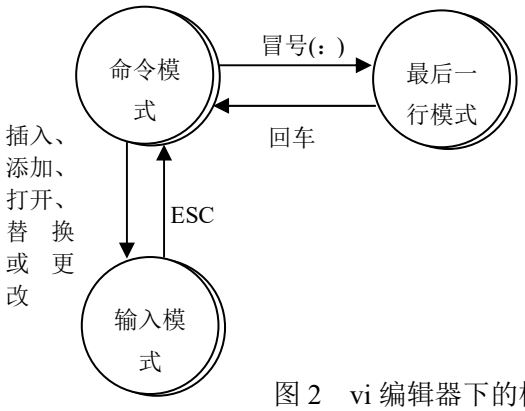


图 2　vi 编辑器下的模式

⑴命令模式

在输入模式下，按 ESC 可切换到命令模式。命令模式下，可选用下列指令离开 vi：

命令	作　用
：q!	离开 vi，并放弃刚在缓冲区内编辑的内容
：wq	将缓冲区内的资料写入当前文件中，并离开 vi
：ZZ	同 wq
：x	同 wq
：w	将缓冲区内的资料写入当前文件中，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

命令模式下光标的移动 ：

命　令	作　用
h 或左箭头	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

⑵输入模式

输入以下命令即可进入 vi 输入模式：

命　令	作　用
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
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

启动 vim 命令：

命令	作用
vim *filename*	从第一行开始编辑 *filename* 文件
vim +*filename*	从最后一行开始编辑 *filename* 文件
vim -r *filename*	在系统崩溃之后恢复 *filename* 文件
vim -R *filename*	以只读方式编辑 *filename* 文件

更多用法见 info vi。

vim 下程序录入过程：
①$ *vim aaa.c* ↙ 进入 vim 命令模式
② *i* ↙ 进入输入模式输入 C 源程序（或文本）
③ *ESC* ↙ 回到命令模式
④ *ZZ* ↙ 保存文件并退出 vim
⑤ *CAT aaa.c* ↙ 显示 aaa.c 文件内容

七、熟悉 gcc 编译器

GNU/Linux 中通常使用的 C 编译器是 GNU gcc。编译器把源程序编译生成目标代码的任务分为以下 4 步：
a. 预处理，把预处理命令扫描处理完毕；
b. 编译，把预处理后的结果编译成汇编或者目标模块；
c. 汇编，把编译出来的结果汇编成具体 CPU 上的目标代码模块；
d. 连接，把多个目标代码模块连接生成一个大的目标模块；

1．使用语法：
gcc [option | filename]...
其中 option 为 gcc 使用时的选项，而 filename 为 gcc 要处理的文件。

2．GCC 选项
GCC 的选项有很多类，这类选项控制着 GCC 程序的运行，以达到特定的编译目的。

⑴全局选项(OVERALL OPTIONS)
全局开关用来控制在“GCC 功能介绍”中的 GCC 的 4 个步骤的运行，在缺省的情况下，这 4 个步骤都是要执行的，但是当给定一些全局开关后，这些步骤就会在 某一步停止执行，这产生中间结果，例如可能你只是需要中间生成的预处理的结果或者是汇编文件(比如你的目的是为了看某个 CPU 上的汇编语言怎么写)。
① -x language
对于源文件是用什么语言编写的，可以通过文件名的后缀来标示，也可以用这开关。指定输入文件是什么语言编写的，language 可以是如下的内容
a. c
b. objective-c
c. c-header
d. c++
e.cpp-output
f.assembler
g.assembler-with-cpp
②-x none
把-x 开关都给关掉了。
③ -c
编译成把源文件目标代码，不做连接的动作。
④-S
把源文件编译成汇编代码，不做汇编和连接的动作。
⑤-E
只把源文件进行预处理之后的结果输出来。不做编译，汇编，连接的动作。

⑥ -o file（常用）

指明输出文件名是 file。

⑦-v

把整个编译过程的输出信息都给打印出来。

⑧-pipe

由于 gcc 的工作分为好几步才完成，所以需要在过程中生成临时文件，使用-pipe 就是用管道替换临时文件。

⑵ 语言相关选项(Language Options)

用来处理和语言相关的选项。

①-ansi

这个开关让 GCC 编译器把所有的 GNU 的编译器特性都给关掉，让你的程序可以和 ansi 标准兼容。

②-include file

在编译之前，把 file 包含进去，相当于在所有编译的源文件最前面加入了一个#include <file>语句，

③-C

同-E 参数配合使用。让预处理后的结果，把注释保留，让人能够比较好读它。

⑶连接开关(Linker Options)

用来控制连接过程的开关选项。

① -llibrary

连接库文件开关。例如-lugl，则是把程序同 libugl.a 文件进行连接。

② -lobjc

这个开关用在面向对象的 C 语言文件的库文件处理中。

③ -nostartfiles

在连接的时候不把系统相关的启动代码连接进来。

④ -nostdlib

在连接的时候不把系统相关的启动文件和系统相关的库连接进来。

⑤-static

在一些系统上支持动态连接，这个开关则不允许动态连接。

⑥shared

生成可共享的被其他程序连接的目标模块。

⑷目录相关开关(Directory Options)

用于定义与目录操作相关的开关。

-Ldir

搜寻库文件(*.a)的路径。

⑸调试开关(Debugging Options)

-g

把调试开关打开，让编译的目标文件有调试信息。

-V version

用来告诉编译器使用它的多少版本的功能，version 参数用来表示版本。

八、掌握 Ubuntu 下 C 程序编辑运行过程（重点）

Ubuntu 下编写 C 程序要经过以下几个步骤：

⑴启动常用的编辑器，键入 C 源程序代码。

例如，点击*应用程序/附件/文本编辑器*，进入编辑环境，输入 C 源程序，保存并命名为 hello.c

```
# include <stdio.h>
void main(void)
{
   Printf(“Hello world!\n”);
}
```

⑵编译源程序

点击*应用程序/附件/终端*，进入命令行。用 gcc 编译器对 C 源程序进行编译，以生成一个可执行文件。方法：

```
gcc  -o  hello.out  hello.c ↙
```

⑶运行可执行文件

```
•/hello.out ↙
```

注：命令行中 -o 选项表示要求编译器输出可执行文件名为 hello.out 文件，hello.c 是源程序文件。

【实验报告】

1. 举例列出常用的 shell 命令使用方法。
2. 通过实例总结上机调试 C 语言程序的过程及此次上机的感想。

预备实验二　shell 脚本

【实验目的】

1、了解和熟悉创建并使用脚本的步骤。
2、熟悉 bash 的控制结构。
3、学会简单的 shell 编程。

【实验内容】

1、 创建一个简单的列目录和日期的 shell 脚本并运行之。

步骤：
⑴输入下列命令，创建一个新文件：

```
cat >new_scrip
```

⑵输入下列行：

```
echo “Your files are”
ls
echo “today is”
date
```

按回车键将光标移到一个新行，按 Ctrl+D 键保存并退出。
⑶检查文件内容，确保它是正确的：

```
cat new_script
```

⑷运行脚本，输入它的文件名：

```
new_script
```

该脚本不运行。
⑸输入下列命令，显示文件的权限：

```
ls  -l new _script
```

权限表明该文件不是可执行。要通过简单调用文件名来运行脚本，必须有权限。
⑹输入下列命令，使 new_script 变成可执行文件。

```
chmod +x new_script
```

⑺要查看新的权限，输入：

```
ls  -l
```

现在拥有文件的读、写和执行权限。
⑻输入新脚本的名字以执行它：

```
new_script
```

所有输入到文件的命令都执行，并输出到屏幕上。
⑼如果接收到错误信息，比如：

```
command not found
```

输入下列命令：

```
./new_script
```

该命令行通知 shell 到哪里寻找 shell 脚本 new_script,即您的当前目录“.”。

2、 用 Shell 语言编制一 Shell 程序，该程序在用户输入年、月之后，自动打印数出该年该月的日历。

<参考程序>

```
echo “Please input the month:”
read month
echo “Please input the year:”
read year
cal $month $year
```

3、编程提示用户输入两个单词，并将其读入，然后比较这两个单词，如果两个单词相同则显示“Match”，并显示“End of program”，如果不同则显示“End of program”。

<参考程序>

```
 $ cat > if1
echo -n “word 1:”
read word1
echo -n “word 2:”
read word2

if test “$word1” = “$word2”
   then
        echo “Match”
fi
echo “End of program.”
```

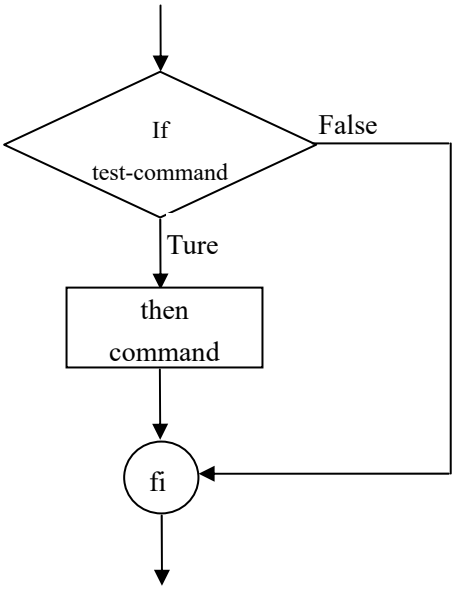


图 3.1　if...then 流程图

<程序说明>

①if…then 控制结构的语法是：

```
if test_command
   then
        commands
fi
```

②其中 test_command 为 test “$word1” = “$word2”，test 是一个内置命令，如果它的第一个参数和第三个参数存在第二个参数所指定的关系，那么 test 将返回 ture。Shell 将执行 then 和 fi 之间的命令。否则执行 fi 后面语句。

4、修改上述程序，编程提示用户输入两个单词，并将其读入，然后比较这两个单词，如果两个单词相同显示“Match”,不同则显示“Not match”，最后显示“End of program”。
<编程提示>请使用　if...then...else　控制结构。

5、编程使用 case 结构创建一个简单的菜单，屏幕显示菜单：

```
a. Current date and time
b. User currently logged in
c. Name of the working directory
d. Contents of the working directory
Enter a,b,c or d:
```

根据用户输入选项做相应操作。

<参考程序>

```
echo -e “\n   COMMAND MENU\n”
echo “ a. Current date and time”
```

```
echo “ b. User currently logged in”
echo “ c. Name of the working directory”
echo “ d. Contents of the working directory\n”
echo -n “Enter a,b,c or d:”
read answer
echo
case “$answer” in
    a)
        date
        ;;
    b)
        who
        ;;
    c)
        pwd
        ;;
    d)
        ls
        ;;
    *)
        Echo “There is no selection : $answer”
        ;;
esac
```

6、修改上题，使用户可以连续选择直到想退出时才退出。

7、编程使用 select 结构生成一个菜单如下：

```
1）apple        3)blueberry        5)orange          7)STOP
2) banana       4)kiwi             6)watermelon
Choose your favorite fruit from these possibilities:
```

用户输入所选项，如 1 显示：

```
You chose apple as you favorite.
That is choice number 1.
```

<参考程序>

```
#!/bin/bash
ps3=“Chose your favorite fruit from these possibilities:”
select FRUIT in apple banana blueberry kiwi orange watermelon STOP
do
if [ $FRUIT = STOP ]  then
echo “Thanks for playing!”
break
fi
echo “You chose $FRUIT as you favorite.”
echo “That is choice number $REPLY.”
echo
done
```

<程序说明>

①select 结构的语法如下：

```
select varname[in arg…]
do
    commands
done
```

②REPLY 是键盘变量。

【思考题】

1、什么选项通知 rm、cp 和 mv 在删除或覆盖文件前得到用户的确认？
2、如何确认自己在主目录中？然后再主目录中创建一个名为 Dannty 的目录，再进入到 Danny 目录，并确认你的位置？
3、命令 echo$PATH 的输出是什么？
4、下列命令的运行结果是什么？

```
who | grep $USER
grep \$HOME file1
```

【实验报告】

1．列出调试通过程序的清单，并加注释。
2．回答思考题。
3．总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

创建并使用脚本的步骤：
⑴创建 shell 命令文件。
⑵使用 chmod 命令使文件可执行。
⑶通过输入脚本文件名执行文件。

在执行脚本时，shell 读取脚本并按其指示执行。它逐行执行脚本，就像这些行是从键盘输入的一样。脚本中所有的实用程序都执行。

实验一　进程控制

【实验目的】

1、掌握进程的概念，明确进程和程序的区别。
2、认识和了解并发执行的实质。
3、分析进程争用资源的现象，学习解决进程互斥的方法。

【实验内容】

1、进程的创建（必做题）

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

<参考程序>

```
# include<stdio.h>
main()
{ int  p1, p2;
while((p1=fork())= = -1);
if(p1= =0)
putchar( ‘b’ );
else
  { while((p2=fork())= =-1);
    if(p2= =0)
    putchar( ‘c’ );
    else
    putchar(  ‘a’ );
   }
 }
```

2、 修改已编写的程序，将每个进程的输出由单个字符改为一句话，再观察程序执行时屏幕上出现的现象，并分析其原因。（必做题）

<参考程序>

```
# include<stdio.h>
main()
{ int  p1, p2, i;
  while((p1=fork())= = -1);
  if(p1= =0)
    for(i=0;i<500;i++)
    printf(“child%d\n”,i);
  else
   { while((p2=fork())= =-1);
     If(p2= =0)
```

```
            for(i=0;i<500;i++)
            printf(“son%d\n”,i);
        else
            for(i=0;i<500;i++)
            printf(“daughter%d\n”,i);
        }
    }
```

3、编写程序创建进程树如图 1 和图 2 所示，在每个进程中显示当前进程识别码和父进程识别码。(必做题)

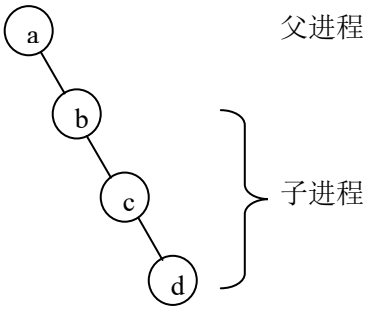


图 1　进程树

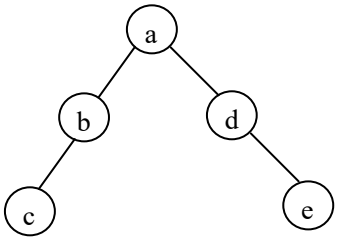


图 2　进程树

【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?
3、当前运行的程序（主进程）的父进程是什么?

【实验报告】

1、列出调试通过程序的清单，分析运行结果。
2、给出必要的程序设计思路和方法（或列出流程图)。
3、回答思考题。
4、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

一、进程概念

1．进程

UNIX 中，进程既是一个独立拥有资源的基本单位，又是一个独立调度的基本单位。一个进程实体由若干个区（段）组成，包括程序区、数据区、栈区、共享存储区等。每个区又分为若干页，每个进程配置有唯一的进程控制块 PCB，用于控制和管理进程。PCB 的数据结构如下：

⑴ 进程表项（Process　Table　Entry)。

包括一些最常用的核心数据,如: 进程标识符 PID、用户标识符 UID、进程状态、事件描述符、进程和 U 区在内存或外存的地址、软中断信号、计时域、进程的大小、偏置值 nice、指向就绪队列中下一个 PCB 的指针 P_Link、指向 U 区进程正文、数据及栈在内存区域的指针。

⑵ U 区（U Area)。

用于存放进程表项的一些扩充信息。每一个进程都有一个私用的 U 区，其中含有：进程表项指

针、真正用户标识符 u-ruid(read user ID)、有效用户标识符 u-euid(effective user ID)、用户文件描述符表、计时器、内部 I/O 参数、限制字段、差错字段、返回值、信号处理数组。

由于 UNIX 系统采用段页式存储管理，为了把段的起始虚地址变换为段在系统中的物理地址，便于实现区的共享，所以还有：

⑶ 系统区表项。

以存放各个段在物理存储器中的位置等信息。系统把一个进程的虚地址空间划分为若干个连续的逻辑区，有正文区、数据区、栈区等。这些区是可被共享和保护的独立实体，多个进程可共享一个区。为了对区进行管理，核心中设置一个系统区表，各表项中记录了以下有关描述活动区的信息：区的类型和大小、区的状态、区在物理存储器中的位置、引用计数、指向文件索引结点的指针。

⑷ 进程区表

系统为每个进程配置了一张进程区表。表中，每一项记录一个区的起始虚地址及指向系统区表中对应的区表项。核心通过查找进程区表和系统区表，便可将区的逻辑地址变换为物理地址。

2. 进程映像

UNIX 系统中，进程是进程映像的执行过程，也就是正在执行的进程实体。它由三部分组成：

⑴ 用户级上、下文。主要成分是用户程序；

⑵ 寄存器上、下文。由 CPU 中的一些寄存器的内容组成，如 PC，PSW，SP 及通用寄存器等；

⑶ 系统级上、下文。包括 OS 为管理进程所用的信息，有静态和动态之分。

3．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。

二、所涉及的中断调用

1、fork（ ）

创建一个新的子进程。其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码、组代码、环境变量、已打开的文件代码、工作目录和资源限制。

系统调用格式：

```
int fork()
```

如果 Fork 成功则在父进程会返回新建立的子进程代码（PID），而在新建立的子进程中则返回 0。如果 fork 失败则直接返回-1。

2、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：

```
int  wait(status)
int  *status;
```

其中，status 是用户空间的地址。它的低 8 位反应子进程状态，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。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。

3、exit（ ）

终止进程的执行。
系统调用格式：

```
void exit(status)
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”

实验二　进程间通信

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）信号量机制实验

【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、编写一段程序，使用系统调用 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 ctrl+c 键），当捕捉到中断信号后，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 process 1 is killed by parent!
Child process 2 is killed by parent!

父进程等待两个子进程终止后，输出以下信息后终止：

Parent process is killed!

<参考程序>

```
# include<stdio.h>
# include<signal.h>
# include<unistd.h>
main()
{ int  p1, p2;
signal(SIGINT,stop);
while((p1=fork())==-1);
 if(p1>0)
{ ①
  while((p2=fork())= =-1);
   If(p2>0)
    { ②
      wait_mark=1;
      waiting(0);
      kill(p1,10);
      kill(p2,12);
      wait();
      wait();
      printf(“parent process is killed!\n”);
      exit(0);
    }
    else
```

```
 {
  wait_mark=1;
  signal(12,stop);
  waiting();
  lockf(1,1,0);
  printf(“child process 2 is killed by parent!\n”);
  lockf(1,0,0);
  exit(0);
  }
}
else
{
   wait_mark=1;
   signal(10,stop);
   waiting();
   lockf(1,1,0);
   printf(“child process 1 is killed by parent!\n”);
   lockf(1,0,0);
   exit(0);
  }
}
void waiting()
{
   while(wait_mark!=0);
}
void stop()
{
   wait_mark=0;
}
```

实验要求：

⑴、运行程序并分析结果。
⑵、如果把 signal(SIGINT,stop)放在①号和②号位置，结果会怎样并分析原因。
⑶、该程序段前面部分用了两个 wait(0),为什么？
⑷、该程序段中每个进程退出时都用了语句 exit(0),为什么？

2、修改上面的程序，增加语句 signal(SIGINT,SIG_IGN)和语句 signal(SIGQUIT,SIG_IGN)，再观察程序执行时屏幕上出现的现象，并分析其原因。

<参考程序>

```
# include<stdio.h>
# include<signal.h>
# include<unistd.h>
main()
{ int  pid1, pid2;
  int  EndFlag=0;
  void IntDelete()
```

```
{
  kill(pid1,10);
  kill(pid2,12);
  EndFlag=1;
}
void Int1()
{
   printf(“child process 1 is killed by parent !\n”);
   exit(0);
}
void Int2()
{
   printf(“child process 2 is killed by parent !\n”);
   exit(0);
}
int exitcode;
signal(SIGINT,SIG_IGN);
signal(SIGQUIT,SIG_IGN);
while((pid1=fork())==-1);
if(pid==0)
{
  signal(SIGUSER1,Int1);
  signal(SIGQUIT,SIG_IGN);
  pause();
  exit(0);
 }
 else
 {
  while((pid2=fork())= =-1);
   if(pid2==0)
   {
    signal(SIGUSER1,Int1);
    signal(SIGQUIT,SIG_IGN);
    pause();
    exit(0);
    }
   else
    {
     signal(SIGINT,IntDelete);
     waitpid(-1,&exitcode,0);
     printf(“parent process is killed \n”);
     exit(0);
    }
 }
}
```

实验要求：运行程序并分析结果。

⑶司机售票员问题（选做题）

编程用 fork()创建一个子进程代表售票员，司机在父进程中，再用系统调用 signal()让父进程（司机）捕捉来自子进程（售票员）发出的中断信号，让子进程（售票员）捕捉来自（司机）发出的中断信号，以实现进程间的同步运行。

【实验报告】

1、列出调试通过程序的清单，分析运行结果。
2、给出必要的程序设计思路和方法（或列出流程图）。
3、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

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。

⑴ 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。

⑵ 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。

⑶ 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(pid,sig)
参数定义
int pid,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。
系统调用格式
signal(sig,function)
头文件为

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

参数定义
signal(sig,function)
int sig;
void (*func) ( )

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	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。

3. lockf()函数允许将文件区域用作信号量（监视锁），或用于控制对锁定进程的访问（强制模式记录锁定）。试图访问已锁定资源的其他进程将返回错误或进入休眠状态，直到资源解除锁定为止。当关闭文件时，将释放进程的所有锁定，即使进程仍然有打开的文件。当进程终止时，将释放进程保留的所有锁定。

```
#include <unistd.h>
int lockf(int fd, int cmd, off_t len);
```

（1）fd 是打开文件的文件描述符。

为通过此函数调用建立锁定，文件描述符必须使用只写权限（O_WRONLY）或读写权限（O_RDWR）打开。如果调用进程是具有 PRIV_LOCKRDONLY 权限的组的成员，它也可以使用 lockf()来锁定使用只读权限（O_RDONLY）打开的文件。

（2）cmd 是指定要采取的操作的控制值，允许的值在中定义。

如下所示：

```
# define F_ULOCK 0 //解锁
# define F_LOCK 1 //互斥锁定区域
# define F_TLOCK 2 //测试互斥锁定区域
# define F_TEST 3 //测试区域
```

F_ULOCK 请求可以完全或部分释放由进程控制的一个或多个锁定区域。如果区域未完全释放，剩余的区域仍将被进程锁定。如果该表已满，将会返回[EDEADLK]错误，并且不会释放请求的区域。

使用 F_LOCK 或 F_TLOCK 锁定的区域可以完全或部分包含同一个进程以前锁定的区域，或被同一个进程以前锁定的区域包含。此时，这些区域将会合并为一个区域。如果请求要求将新元素添加到活动锁定表中，但该表已满，则会返回一个错误，并且不会锁定新区域。

F_LOCK 和 F_TLOCK 请求仅在采取的操作上有所差异（如果资源不可用）。如果区域已被其他进程锁定，F_LOCK 将使调用进程进入休眠状态，直到该资源可用，而 F_TLOCK 则会返回[EACCES]错误。

F_TEST 用于检测在指定的区域中是否存在其他进程的锁定。如果该区域被锁定，lockf()将返回 0，否则返回-1；在这种情况下，errno 设置为[EACCES]。F_LOCK 和 F_TLOCK 都用于锁定文件的某个区域（如果该区域可用）。F_ULOCK 用于删除文件区域的锁定。

（3）len 是要锁定或解锁的连续字节数。

要锁定的资源从文件中当前偏移量开始，对于正 len 将向前扩展，对于负 len 则向后扩展（直到但不包括当前偏移量的前面的字节数）。如果 len 为零，则锁定从当前偏移量到文件结尾的区域（即从当前偏移量到现有或任何将来的文件结束标志）。要锁定一个区域，不需要将该区域分配到文件中，因为这样的锁定可以在文件结束标志之后存在。
使用 S_ENFMT 文件模式的常规文件（未设置组执行位）将启用强制策略。启用强制策略后，如果清除了 O_NDELAY，访问锁定区域的读取和写入将进入休眠状态，直到整个区域可用为止，但是如果设置了 O_NDELAY，将会返回-1 并设置 errno。由其他系统函数（如 exec()）访问的文件不受强制策略的影响。
返回值
此函数调用成功后，将返回值 0，否则返回-1，并且设置 errno 以表示该错误。 由于当文件的某部分被其他进程锁定后，变量 errno 将会设置为[EAGAIN]而不是[EACCES]，因此可移植应用程序应对这两个值进行预计和测试。

（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】

1、编制一段程序，实现进程的管道通信。使用 pipe()建立一条管道线。两个子进程 p1 和 p2 分别向管道各写一句话：

Child 1 is sending message!
Child 2 is sending message!

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。

<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);
while((pid1=fork())= = -1);
if(pid1= =0)
{
  lockf(fd[1],1,0);
  sprintf(OutPipe, ” child 1 process is sending message!”);
  write(fd[1],OutPipe,50);
  sleep(5);
  lockf(fd[1],0,0);
  exit(0);
```

```
}
  else
  {
while((pid2=fork())= = -1);
if(pid2= =0)
{
   lockf(fd[1],1,0);
   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);
  read(fd[0],InPipe,50);
  printf(“%s\n”,InPipe);
  wait(0);
  read(fd[0],InPipe,50);
  printf(“%s\n”,InPipe);
  exit(0);
}
}
}
```

实验要求：运行程序并分析结果。

2．在父进程中用 pipe()建立一条管道线，往管道里写一句话，两个子进程接收这句话。

【实验报告】

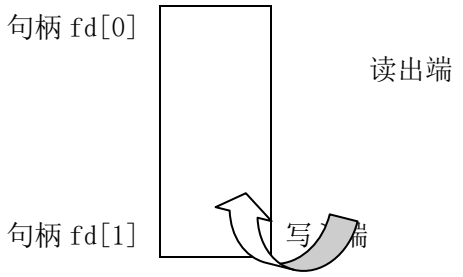
1、列出调试通过程序的清单，分析运行结果。
2、给出必要的程序设计思路和方法（或列出流程图）。
3、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

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

pipe(filedes)

参数定义

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

其中，filedes[1]是写入端，filedes[0]是读出端。
该函数使用头文件如下：

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

2、read( )

系统调用格式

read(fd,buf,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。

参数定义

```
int  read(fd,buf,nbyte);
int  fd;
char *buf;
unsigned  nbyte;
```

3、write( )

系统调用格式

read(fd,buf,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。

参数定义同 read( )。

（三）消息的发送与接收实验

【实验目的】

1、了解什么是消息。

2、熟悉消息传送机理。

3、编程实现消息的发送与接收。

【实验内容】

1、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。

<参考程序>

①client.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( );
}
```

②server.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( );
}
```

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。
②server 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。”
③client 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、编译为 client 与 server。执行：

2、 在父进程中创建一个消息队列，用 fork 创建一个子进程，在子进程中将一条消息传送至消息队列，父进程接受队列的消息，并将消息送屏幕显示。

【实验报告】

1、列出调试通过程序的清单，分析运行结果。
2、给出必要的程序设计思路和方法（或列出流程图）。
3、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

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、消息队列的描述符

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

1. msgget( )

创建一个消息，获得一个消息的描述符。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)
该函数使用头文件如下：
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
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 是用户设置的标志和访问方式。如 IPC_CREAT |0400 是否该队列已被创建。无则创建，是则打开；
IPC_EXCL |0400　　是否该队列的创建应是互斥的。
msgqid 是该系统调用返回的描述符，失败则返回-1。

2. msgsnd（）

发送一消息。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)
该函数使用头文件如下：
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
参数定义：
int msgsnd(msgqid,msgp,size,flag)
int msgqid,size,flag;
struct msgbuf * msgp;
其中 msgqid 是返回消息队列的描述符；msgp 是指向用户消息缓冲区的一个结构体指针。缓冲区中包括消息类型和消息正文，即
{
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)
本函数使用的头文件如下：
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
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);
```

本函数使用的头文件如下：

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

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；
（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。消除消息队列的标识符。

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*/
    }
```

（四） 共享存储区通信

【实验目的】

1、了解和熟悉共享存储机制。
2、学会用共享存储区方法进行通信。

【实验内容】

1、编制一长度为 1k 的共享存储区发送和接收的程序。

<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。
②server 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”。
③client 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。

<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( );
   while ((i=fork( ))= =-1);
   if (!i) client( );
   wait(0);
   wait(0);
}
```

实验要求：运行程序,并分析运行结果。

2、 编程在主进程中创建两个子进程，在子进程 shmw()中创建一个系统 V 共享内存区，并在其中写入格式化数据；在子进程 shmr()中访问同一个系统 V 共享内存区，读出其中的格式化数据。

【实验报告】

1、列出调试通过程序的清单，分析运行结果。
2、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

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 1 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。

Linux 的 2.2.x 内核支持多种共享内存方式，如 mmap()系统调用，Posix 共享内存，以及系统 V 共享内存。linux 发行版本如 Redhat 8.0 支持 mmap()系统调用及系统 V 共享内存，但还没实现 Posix

共享内存，这里主要介绍系统 V 共享内存 API 的原理及应用。

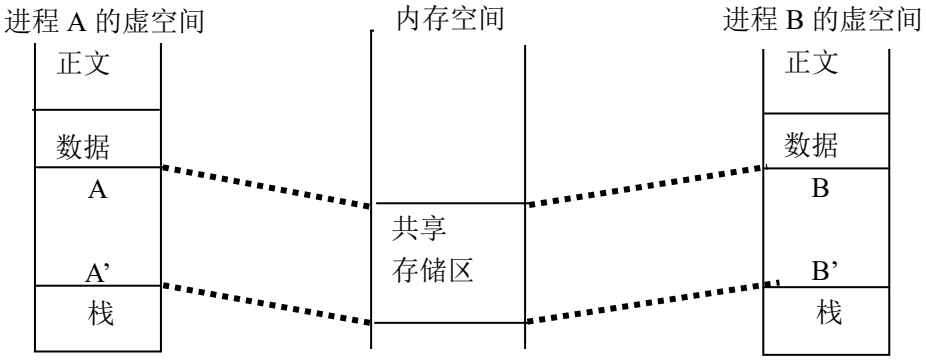


图 1　共享存储区

2、系统 V 共享内存原理

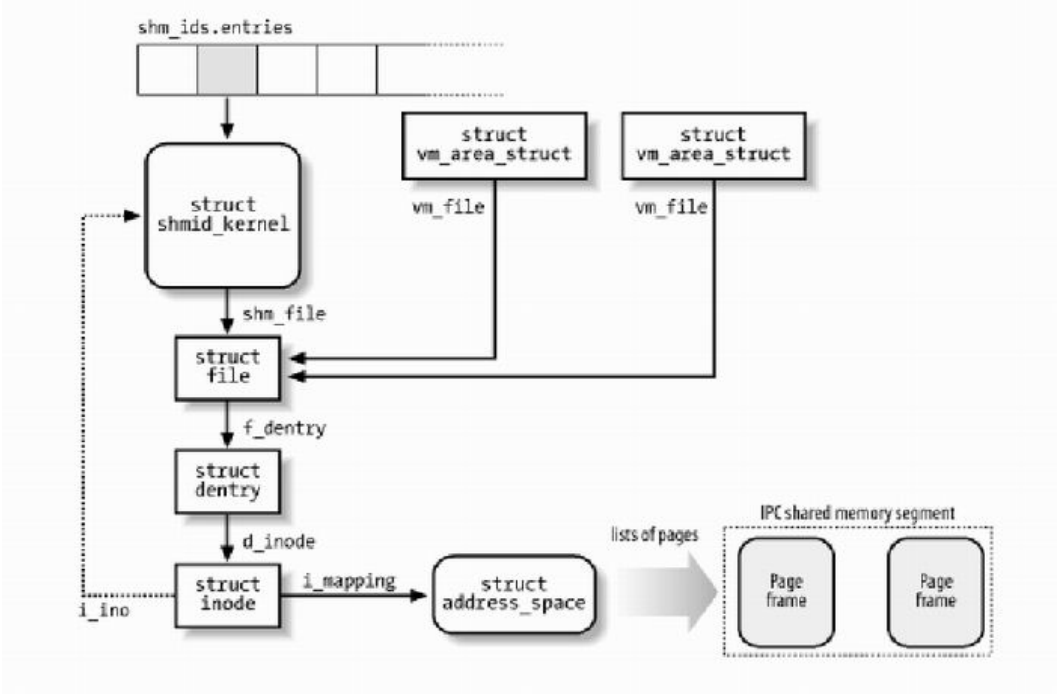
进程间需要共享的数据被放在一个叫做 IPC 共享内存区域的地方，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。系统 V 共享内存通过 shmget 获得或创建一个 IPC 共享内存区域，并返回相应的标识符。内核在保证 shmget 获得或创建一个共享内存区，初始化该共享内存区相应的 shmid_kernel 结构同时，还将在特殊文件系统 shm 中，创建并打开一个同名文件，并在内存中建立起该文件的相应 dentry 及 inode 结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享内存区）。所有这一切都是系统调用 shmget 完成的。

注：每一个共享内存区都有一个控制结构 struct shmid_kernel，shmid_kernel 是共享内存区域中非常重要的一个数据结构，它是存储管理和文件系统结合起来的桥梁，定义如下：

```
struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm    shm_perm;
    struct file *       shm_file;
    int         id;
    unsigned long       shm_nattch;   /*
    unsigned long       shm_segsz;   /*共享内存的大小（bytes）*/
    time_t          shm_atim;   /*最后一次 attach 此共享内存的时间*/
    time_t          shm_dtim;   /*最后一次 detach 此共享内存的时间*/
    time_t          shm_ctim;   /*最后一次更动此共享内存结构的时间*/
    pid_t           shm_cpid;   /*建立此共享内存的进程识别码*/
    pid_t           shm_lpid;   /*最后一个操作此共享内存的进程识别码*/
};
```

该结构中最重要的一个域应该是 shm_file，它存储了将被映射文件的地址。每个共享内存区对象都对应特殊文件系统 shm 中的一个文件，一般情况下，特殊文件系统 shm 中的文件是不能用 read()、write()等方法访问的，当采取共享内存的方式把其中的文件映射到进程地址空间后，可直接采用访问内存的方式对其访问。

这里我们采用[1]中的图表给出与系统 V 共享内存相关数据结构：



正如消息队列和信号灯一样，内核通过数据结构 struct ipc_ids shm_ids 维护系统中的所有共享内存区域。上图中的 shm_ids.entries 变量指向一个 ipc_id 结构数组，而每个 ipc_id 结构数组中有个指向 kern_ipc_perm 结构的指针。到这里读者应该很熟悉了，对于系统 V 共享内存区来说，kern_ipc_perm 的宿主是 shmid_kernel 结构，shmid_kernel 是用来描述一个共享内存区域的，这样内核就能够控制系统中所有的共享区域。同时，在 shmid_kernel 结构的 file 类型指针 shm_file 指向文件系统 shm 中相应的文件，这样，共享内存区域就与 shm 文件系统中的文件对应起来。

在创建了一个共享内存区域后，还要将它映射到进程地址空间，系统调用 shmat()完成此项功能。由于在调用 shmget()时，已经创建了文件系统 shm 中的一个同名文件与共享内存区域相对应，因此，调用 shmat()的过程相当于映射文件系统 shm 中的同名文件过程，原理与 mmap()大同小异。

3、系统 V 共享内存 API

对于系统 V 共享内存，主要有以下几个 API：shmget()、shmat()、shmdt()及 shmctl()。

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

shmget()用来获得共享内存区域的 ID，如果不存在指定的共享区域就创建相应的区域。shmat()把共享内存区域映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区域进行访问操作。shmdt()调用用来解除进程对共享内存区域的映射。shmctl 实现对共享内存区域的控制操作。

注：shmget 的内部实现包含了许多重要的系统 V 共享内存机制；shmat 在把共享内存区域映射到进程空间时，并不真正改变进程的页表。当进程第一次访问内存映射区域访问时，会因为没有物理页表的分配而导致一个缺页异常，然后内核会根据相应的存储管理机制为共享内存映射区域分配相应的页表。

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

1、shmget( )

创建、获得一个共享存储区。

系统调用格式：

shmid=shmget(key,size,flag)

该函数使用头文件如下：

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

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|0400))

创建一个关键字为 key，长度为 size 的共享存储区

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)

该函数使用头文件如下：

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

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；flag 规定共享存储区的读、写权限，以及系统是否应对用户规定的地址做舍入操作。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)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

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)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

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。

cmd 有下列几种数值：

IPC_STAT　　把共享内存的

IPC_SET　　将参数 buf 所指的 shmid_ds 结构中的 shm_perm.uid、shm_perm.gid 和 shm_perm.mode 复制到共享内存的 shmid_ds 结构内。

IPC_RMID　　删除共享内存和数据结构。

SHM_LOCK　　不让此共享内存置换到 swap。

SHM_UNLOCK　允许此 gon 共享内存置换到 swap。

SHM_LOCK 和 SHM_UNLOCK 为 LUNIX 特有，且唯有超级用户（root）允许使用

选作实验　死锁避免的算法

【实验目的】

1、 了解死锁避免的原理。
2、 研究银行家算法的实现方法。

【实验内容】

编程实现银行家算法。

【实验报告】

1、 列出调试通过程序的清单，并附上文档说明。
2、 总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

一、死锁概念

多个并发进程,每个进程占有部分资源,又都等待其它进程释放所占资源,造成均不能向前推进的现象。

二、死锁的避免

死锁避免原理就是使系统始终处于安全状态。

安全状态：所谓安全状态是指能够找到一个安全序列，系统能够按照这个安全序列依次为进程分配资源，使所有进程都能执行完毕，如果找不到这样的安全序列，系统就处于不安全状态。

三、银行家算法

银行家算法要求每个进程的最大资源需求，其基本思想是：始终保持系统处于安全状态，当进程提出资源请求时，系统先进行预分配，再判断系统分配后是否仍然处于安全状态。如果仍然处于安全状态，就进行实际分配；如果处于不安全状态，则拒绝该进程的资源请求。

四、银行家算法相关数据结构

1. 最大需求矩阵：

$$d(t)=\begin{pmatrix} d_{11}d_{12}...d_{1m} \\ d_{21}d_{22}...d_{2m} \\ .... \\ d_{n1}d_{n2}...d_{nm} \end{pmatrix}$$

其中，行表示进程，列表示资源，如：d_{ij}=5 表示第 i 个进程最多需要 j 类资源 5 个。

2. 资源分配矩阵：

$$a(t)=\begin{pmatrix} a_{11}a_{12}...a_{1m} \\ a_{21}a_{22}...a_{2m} \\ .... \\ a_{n1}a_{n2}...a_{nm} \end{pmatrix}$$

元素 a_{ij}=8 表示分配给第 i 进程 8 个 j 类资源。

3. 需求矩阵：

$$b(t)=\begin{pmatrix} b_{11}b_{12}...b_{1m} \\ b_{21}b_{22}...b_{2m} \\ .... \\ b_{n1}b_{n2}...b_{nm} \end{pmatrix}$$

元素 b_{ij}=3 表示第 i 类进程还需要 3 个 j 类资源。

最大需求矩阵=分配矩阵+需求矩阵，即 d(t)=a(t)+b(t)。

实验三　进程调度

【实验目的】

实验目的：（1）通过编写程序实现进程或作业先来先服务、高优先权、按时间片轮转调度算法，使学生进一步掌握进程调度的概念和算法，加深对处理机分配的理解。

（2）了解进程（线程）的调度机制。

（3）学习使用进程（线程）调度算法，掌握相应的与调度有关的 API 函数。

实验要求：（1）经调试后程序能够正常运行。

（2）采用多进程或多线程方式运行，体现了进程或作业先来先服务、高优先权、按时间片轮转调度的关系。

（3）程序界面美观。

【实验内容】（含原理图、流程图、关键代码，或实验过程中的记录、数据等）

进程调度的算法:

1)先来先服务算法：如果早就绪的进程排在就绪队列的前面，迟就绪的进程排在就绪队列的后面，那么先来先服务（FCFS：first come first service）总是把当前处于就绪队列之首的那个进程调度到运行状态。

2)轮转法就是按一定时间片(记为 q)轮番运行各个进程。如果 q 是一个定值，则轮转法是一种对各进程机会均等的调度方法。

3)优先级调度的基本思想是，把当前处于就绪队列中优先级最高的进程投入运行，而不管各进程的下一个 CPU 周期的长短和其他因素。

实验步骤：（1）需求分析：了解基本原理，确定程序的基本功能，查找相关资料，画出基本的数据流图；

（2）概要设计：确定程序的总体结构、模块关系和总体流程；

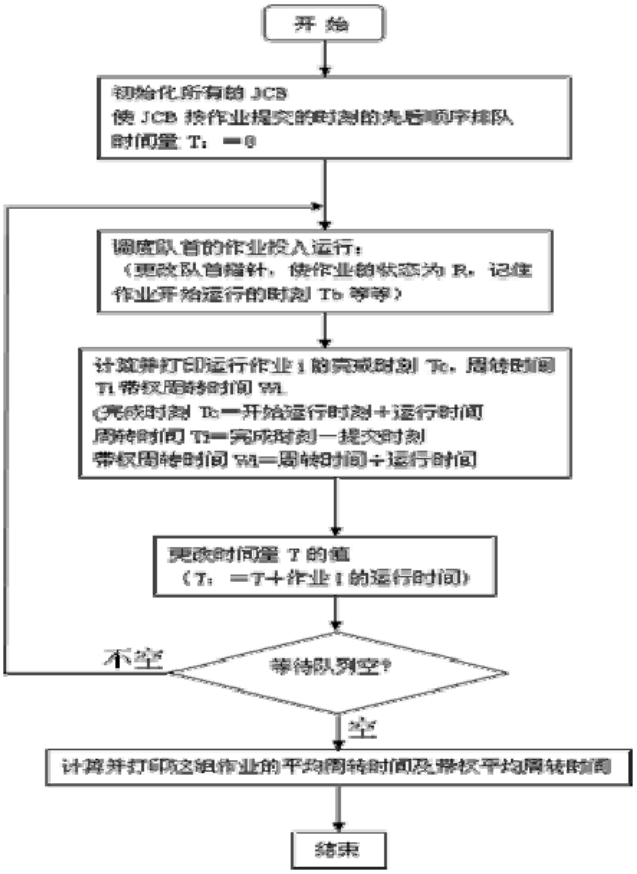
（3）详细设计：确定模块内部的流程和实现算法；

（4）上机编码和调试；

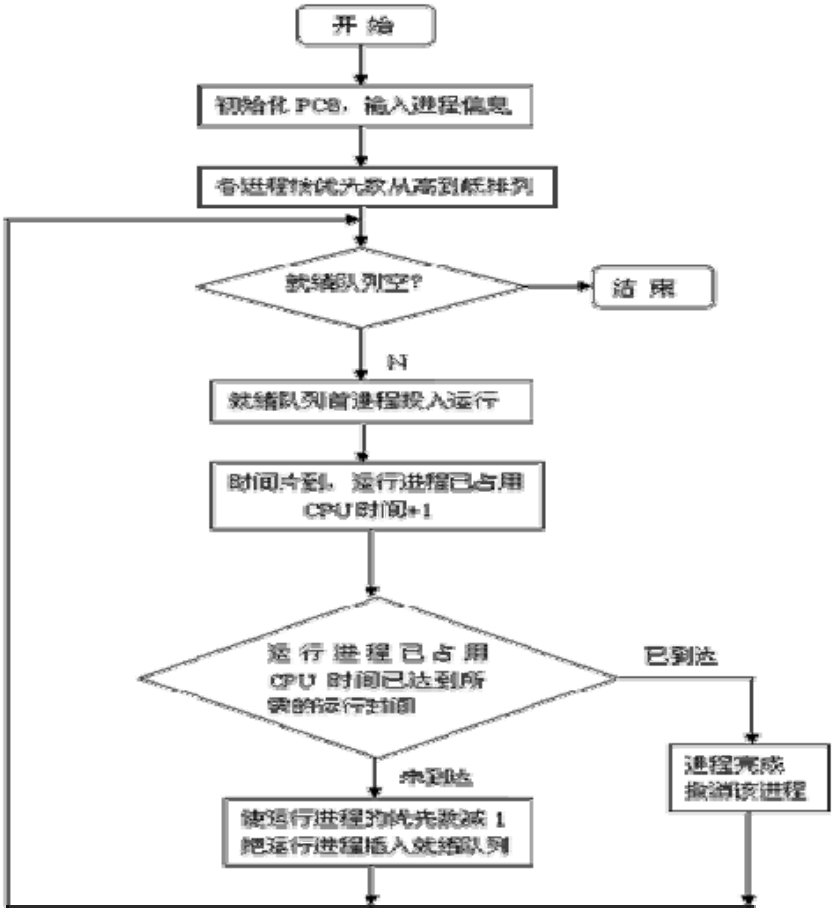
（5）运行测试；

（6）编写实验报告。

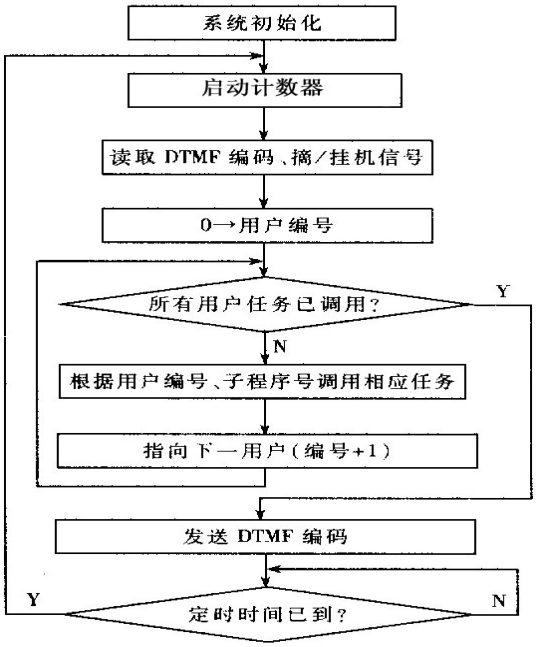
流程图：


开 始
初始化所有的 JCB
使 JCB 按作业提交的时刻的先后顺序排队
时间量 T：＝0
调度队首的作业投入运行：
（更改队首指针，使作业的状态为 R，记住
作业开始运行的时刻 Tb 等等）
计算并打印运行作业 i 的完成时刻 Tc，周转时间
Ti 带权周转时间 Wi
(完成时刻 Tc＝开始运行时刻＋运行时间
周转时间 Ti＝完成时刻－提交时刻
带权周转时间 Wi＝周转时间＋运行时间
更改时间量 T 的值
（T：＝T＋作业 i 的运行时间）
不空
等待队列空？
空
计算并打印这组作业的平均周转时间及带权平均周转时间
结束

（先来先服务流程图）


开 始
初始化 PCB，输入进程信息
各进程按优先数从高到低排列
就绪队列空？
结 束
N
就绪队列首进程投入运行
时间片到，运行进程已占用
CPU 时间+1
运行进程已占用
CPU 时间已达到所
需的运行时间
已到达
未到达
进程完成
撤销该进程
使运行进程的优先数减 1
把运行进程插入就绪队列

（高优先权流程图）



（按时间片轮转调度）

程序说明及实现:

1)先来先服务调度算法：

高响应比优先实现进程调度.(用 C 语言实现),

2) 优先级调度程序：

该程序由主程序、构造队列子程序、打印子程序、运行子程序构成。

3) 时间片轮转法程序：

在此程序中由于程序比较小，未进行分模块设计。直接采用单一模块。

```
1 先来先服务
#i nclude<stdio.h>
float t,d; /*定义两个全局变量*/
struct /*定义一个结构体数组，包括进程的信息*/
{
int id;
float ArriveTime;
float RequestTime;
float StartTime;
float EndTime;
float RunTime;
float DQRunTime;
int Status;
}arrayTask[4]; /*定义初始化的结构体数组*/
GetTask()/*给结构体数组赋值，输入到达，服务时间*/
{ int i;
```

```
float a;
for(i=0;i<4;i++)
{arrayTask[i].id=i+1;
printf("input the number");
printf("input the the ArriveTime of arrayTask[%d]:",i); /*用户输入进程的时间，初始为零 */
scanf("%f",&a);
arrayTask[i].ArriveTime=a;
printf("input the RequestTime of arrayTask[%d]:",i);
scanf("%f",&a);
arrayTask[i].RequestTime=a;
arrayTask[i].StartTime=0;
arrayTask[i].EndTime=0;
arrayTask[i].RunTime=0;
arrayTask[i].Status=0; /*开始默认的标志位零*/
}
}
int fcfs() /*定义 FCFS 中寻找未执行的进程的最先到达时间*/
{
int i,j,w=0; /*在结构体数组中找到一个未执行的进程*/
for(i=0;i<4;i++)
{
if(arrayTask[i].Status==0)
{
t=arrayTask[i].ArriveTime;
w=1;
}
if(w==1)
break;
}
for(i=0;i<4;i++) /*查找数组中到达时间最小未执行的进程*/
{
if(arrayTask[i].ArriveTime<t&&arrayTask[i].Status==0)
t=arrayTask[i].ArriveTime;
} /*返回最小到达时间的数组的下标*/
for(i=0;i<4;i++)
{
if (arrayTask[i].ArriveTime==t)
return i;
}
}
int sjf() /*定义 FCFS 中寻找未执行的进程的最先到达时间*/
{
int i,x=0,a=0,b=0; /*判断是不是第一个执行的进程*/
float g;
for(i=0;i<4;i++)
{
if(arrayTask[i].Status==1)
```

```
{g=arrayTask[i].EndTime;
x=1;
}
}
if(x==0) /*第一个执行的进程按 FCFS*/
{
t=arrayTask[0].ArriveTime;
for(i=0;i<4;i++)
{
if(arrayTask[i].ArriveTime<t)
{ t=arrayTask[i].ArriveTime;
a=i;
}
}
return a;}
else
{
for(i=0;i<4;i++)
{if(arrayTask[i].EndTime>g)
g=arrayTask[i].EndTime;
}
for(i=0;i<4;i++)
{if(arrayTask[i].Status==0&& arrayTask[i].ArriveTime<=g)
{t=arrayTask[i].RequestTime;
a=i;
b=1;} /*判断有没有进程在前个进程完成前到达*/
}
if(b!=0) /*有进程到达则按 SJF*/
{for(i=0;i<4;i++)
{
if(arrayTask[i].Status==0&&arrayTask[i].ArriveTime<=g&&arrayTask[i].RequestTime<t)
{t=arrayTask[i].RequestTime;
a=i;}
}
return a;}
else{ /*否则按 FCFS*/
for(i=0;i<4;i++)
{if(arrayTask[i].Status==0)
t=arrayTask[i].ArriveTime;
}
for(i=0;i<4;i++)
{
if(arrayTask[i].Status==0&&arrayTask[i].ArriveTime<t)
{t=arrayTask[i].ArriveTime;
a=i;
}
}
```

```
return a;}
}
}
new(int s) /*定义执行进程后相关数据的修改*/
{
int i,g=0;
for(i=0;i<4;i++)
{
if(arrayTask[i].Status==0)
continue;
else
{
g=1;
break;
}
}
if(g==0) /*当处理的是第一个未执行的进程时执行*/
{
arrayTask[s].StartTime=arrayTask[s].ArriveTime;
arrayTask[s].EndTime=arrayTask[s].RequestTime+arrayTask[s].ArriveTime;
arrayTask[s].RunTime=arrayTask[s].RequestTime;
arrayTask[s].Status=1;
g=2;
}
if(g==1) /*当处理的不是第一个未执行的进程时执行*/
{
arrayTask[s].Status=1;
for(i=0;i<4;i++)
{
if(arrayTask[i].Status==1)
d=arrayTask[i].EndTime;
}
for(i=0;i<4;i++) /*查找最后执行的进程的完成时间*/
{
if(arrayTask[i].EndTime>d&&arrayTask[i].Status==1)
d=arrayTask[i].EndTime;
}
if(arrayTask[s].ArriveTime<d) /*判断修改的进程的到达时间是否在前一个执行的进程的完成时间前面*/
arrayTask[s].StartTime=d;
else
arrayTask[s].StartTime=arrayTask[s].ArriveTime;
arrayTask[s].EndTime=arrayTask[s].StartTime+arrayTask[s].RequestTime;
arrayTask[s].RunTime=arrayTask[s].EndTime-arrayTask[s].ArriveTime;
}
arrayTask[s].DQRunTime=arrayTask[s].RunTime/arrayTask[s].RequestTime;
}
Printresult(int j) /*定义打印函数*/
```

```
{
printf("%d\t",arrayTask[j].id);
printf("%5.2f\t",arrayTask[j].ArriveTime);
printf("%5.2f\t",arrayTask[j].RequestTime);
printf("%5.2f\t",arrayTask[j].StartTime);
printf("%5.2f\t",arrayTask[j].EndTime);
printf("%5.2f\t",arrayTask[j].RunTime);
printf("%5.2f\n",arrayTask[j].DQRunTime);
}
main()
{ int i,b,k,a,c=0;
int d[4];
clrscr();
printf("\t F. FCFS \n");
printf("\t S. SFJ    \n");
printf("\t Q. EXIT \n");
for(i=0;;i++)
{if(c)
break;
printf("please input the number a:\n");
scanf("%d",&a);
switch(a)
{
case Q: c=1;
break;
case F:printf("please input the different-ArriveTime of arrayTasks\n");
GetTask();
printf("****************************the result of fcfs\n");
printf("Number\tArrive\tServer\tStart\tFinish\tTurnove\tTake power turnover time\n");
for(b=0;b<4;b++) /*调用两个函数改变结构体数的值*/
{
k=fcfs();
d[b]=k;
new(k);
}
for(b=0;b<4;b++)
Printresult(d[b]);/*调用打印函数打出结果*/
continue;
case S: printf("please input the different-RequestTime of arrayTasks\n");
GetTask();
printf("****************************the result of sjf\n");
printf("Number\tArrive\tRequest\tStart\tEnd\tRun\tDQRun time\n");
for(b=0;b<4;b++)
{
k=sjf();
d[b]=k;
new(k);
```

```
}
for(b=0;b<4;b++)
Printresult(d[b]);
continue;
default:printf("the number Error.please input another number!\n");
}
}
}
```

2 时间片轮转法：

```
#include"string.h"
#include    "stdio.h"
#include    "conio.h"
#include    "graphics.h"
#define    NULL    0
typedef    struct    quen          /*定义结构*/
{    char    pname[8];
      int       time1;
      int       time2;
      char    state;
      struct    quen    *next;
}    QUEN;
main()/*主程序*/
               {
                     QUEN    *q,*p,*head,*m;
                     char    str[8],f;
                     int    t,d,n;
                     clrscr();
textmode(C80);
textbackground(0);
textcolor(15);
printf("Enter the maxnumber of nodes(n):\n");/*输入进程数*/
scanf("%d",&n);
d=n;
if(d>0)
{    printf("enter thepname:");
   scanf("%s",str);
   printf("enter    the    need    time:");
   scanf("%d",&t);
   head=p=(QUEN    *)malloc(sizeof(QUEN));
   strcpy(p->pname,str);
   p->time1=t;
   p->time2=0;
   p->state='R';
   p->next=NULL;
   head=p;
   getchar();
   --d;}
```

```
while(d>0)    {/*构建队列表*/
   printf("enter    the    pname:");
   scanf("%s",str);
   printf("enter    need    time:");
   scanf("%d",&t);
   q=(QUEN    *)malloc(sizeof(QUEN));
   strcpy(q->pname,str);
   q->time1=t;
   q->time2=0;
   q->state='R';
   q->next=NULL;
   p->next=q;
   p=q;
--d;
p->next=head;
q=head;}
printf("process name need time runned static\n");
do{    printf(" %s%d %d %c\n",q->pname,q->time1,q->time2,q->state);
        q=q->next;
      }while(q!=head);
      printf("\n");
do{

if(head->time2<head->time1)
{head->time2++;
    if(head->time2==head->time1)
      {    head->state='E';
               q=head;
               textbackground(0);
               printf("The running process is %s\n",q->pname);
               printf("process name left time runned static\n");
         do{    textcolor(15);
/*输入队列表*/
         printf(" %s %d %d %c\n",q->pname,q->time1,q->time2,q->state);
q=q->next;}
while(q!=head);
         printf("\n");
         head=head->next;
         q=head;
         p->next=head;
           }
   else{
     printf("The running process is %s\n",q->pname);
      printf("process name left time runned static\n");
      do {
               printf( “%s%d%d %c\n",q->pname,q->time1,q->time2,q->state);
                  q=q->next;}while(q!=head);
```

```
                printf("\n");
                head=head->next;
                q=head;
                p=p->next;}

   printf("Is it  needing new process?(y or n)\n");/*是否加入新的进程*/
         getchar();
         scanf("%c",&f);
   if(f=='Y'||f=='y'){
         getchar();
         printf("Enter the new pname:");
         scanf("%s",str);
         printf("Enter the new neededtime:");
         scanf("%d",&t);
   m=(QUEN    *)malloc(sizeof(QUEN));
   strcpy(m->pname,str);
   m->time1=t;
   m->time2=0;
   m->state='R';
   m->next=NULL;
if(q->next->state=='E')
         {p=m;
            head=m;
            p->next=head;
q=head;}
else{p->next=m;
            m->next=head;
            p=m;}}
}}while(q->next->state!='E');
         printf("The processes are finished\n");
}
```

3 优先级调度方法：

```
#include       <stdio.h>
#include    "conio.h"
typedef    struct    pcb/*定义结构*/
            {char name[5];
               struct pcb *next;
               int    needtime;
               int    priority;
               char state[5];
            }NODE;

NODE   *create_process(int    n)/*创建队列*/
         {NODE       *head,*s,*t;
            int    time,i=0,j;
            char       pname[5];
            head=(NODE    *)malloc(sizeof(NODE));
```

```
            printf("please    input    process    name:");
            scanf("%s",pname);
            strcpy(head->name,pname);
            printf("please    input    need    time:");
            scanf("%d",&time);
            head->needtime=time;
            printf("please    input    priority:");
            scanf("%d",&j);
            head->priority=j;
            strcpy(head->state,"ready");
            head->next=NULL;
            t=head;
            for(i=1;i<n;i++)
                  {   s=(NODE *)malloc(sizeof(NODE));
printf("please input process name:");
getchar();
gets(pname);
strcpy(s->name,pname);
printf("please input need time:");
canf("%d",&time);
s->needtime=time;
printf("please input priority:");
scanf("%d",&j);
s->priority=j;
strcpy(s->state,"ready");
s->next=NULL;
  t->next=s;
t=s;
      }
  return   head;
      }
pri_process(NODE *p)/*输出进程队列*/
      {int    i;
         NODE   *q;
         q=p->next;
         printf("\n name\tneedtime\tpriority \t state\n");
         while(q!=NULL)
               {printf("%5s\t %2d \t %2d \t  %5s \n",
              q->name,q->needtime,q->priority,q->state);
                  q=q->next;
               }
      }
NODE    *order(NODE    head_sort)/*对进程的优先级进行排序*/
   {NODE    *p,*s,*q,*head,*r,*t;
      int       m,pr;
      char       name[5];
      head=head_sort;
```

```
      p=head->next;
      r=p;
      t=p;
      q=p->next;
      while(r!=NULL)
         {    while(q!=NULL)
               {if(p->priority<q->priority)
{m=p->priority;
   p->priority=q->priority;
   q->priority=m;
   strcmp(name,p->name);
   strcmp(p->name,q->name);
   strcmp(q->name,name);
   pr=p->needtime;
   p->needtime=q->needtime;
   q->needtime=pr;
}
                  p=q;
                  q=q->next;
               }
               r=r->next;
               p=t;
               q=p->next;
         }
      return(head_sort);
}
main()/*主程序*/
{       NODE    *p=NULL,*head=NULL,*m=NULL,*z=NULL,*n=NULL;
         int    j,time,x=0;
         char    c,pname[5];
         clrscr();
         printf("please input process number!");
         scanf("%d",&x);
         p=create_process(x);
         head->next=p;
         pri_process(head);
         getchar();
         while(x>0)
               { order(head);
            m=head->next;
strcpy(m->state,"run");
if(m->priority>=2)
         m->priority--;
         m->needtime--;
if(head->next!=NULL)
         pri_process(head);
if(m->needtime==0)
```

```
    {    head->next=m->next;
            printf("%s    has    finished\n",m->name);
            free(m);
         x--;
    }
            getchar();       }
            textmode(C80);
            textbackground(0);
            textcolor(4);
            printf("over!");
            getchar();
}
```

实验四　存储管理

(一)常用页面置换算法

【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】

1、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

2、 用系统提供的 malloc 函数和 free 函数模拟生成一些空闲区，用 FF、BF 算法组织形成空闲区队列，随机生成一个作业序列模拟其分配过程。（选做）

【实验准备】

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。

<参考程序>

```
#define TRUE 1
#define FALSE 0
#define INVALID -1
#define NULL  0

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

typedef struct                       /*页面结构*/
{
    int pn,pfn,counter,time;
}pl_type;
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;
};

typedef struct pfc_struct pfc_type;

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

int diseffect,  a[total_instruction];
int page[total_instruction],  offset[total_instruction];

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”*/
```

```
s=(float)319*rand( )/32767/32767/2+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( )/32767/32767/2; /*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( )/32767/32767/2+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;                          /*用户进程的内存页面数*/
{int i;
diseffect=0;
for(i=0;i<total_vp;i++)
{
       pl[i].pn=i;
       pl[i].pfn=INVALID;        /*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++)
{
       pfc[i].next=&pfc[i+1];
       pfc[i].pfn=i;
}   /*建立 pfc[i-1]和 pfc[i]之间的链接*/
pfc[total_pf-1].next=NULL;
pfc[total_pf-1].pfn=total_pf-1;
freepf_head=&pfc[0];          /*空页面队列的头指针为 pfc[0]*/

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*/
     for(i=0;i<total_instruction;i++)
   {
   if(pl[page[i]].pfn==INVALID)   /*页面失效*/
     {
      diseffect+=1;                   /*失效次数*/
      if(freepf_head==NULL)          /*无空闲页面*/
       {
         p=busypf_head->next;
         pl[busypf_head->pn].pfn=INVALID;
         freepf_head=busypf_head;  /*释放忙页面队列的第一个页面*/
         freepf_head->next=NULL;
         busypf_head=p;
       }
        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)
        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;
}
```

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
   ……
```

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

【实验报告】

1、列出调试通过程序的清单，并附文档说明。
2、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

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。

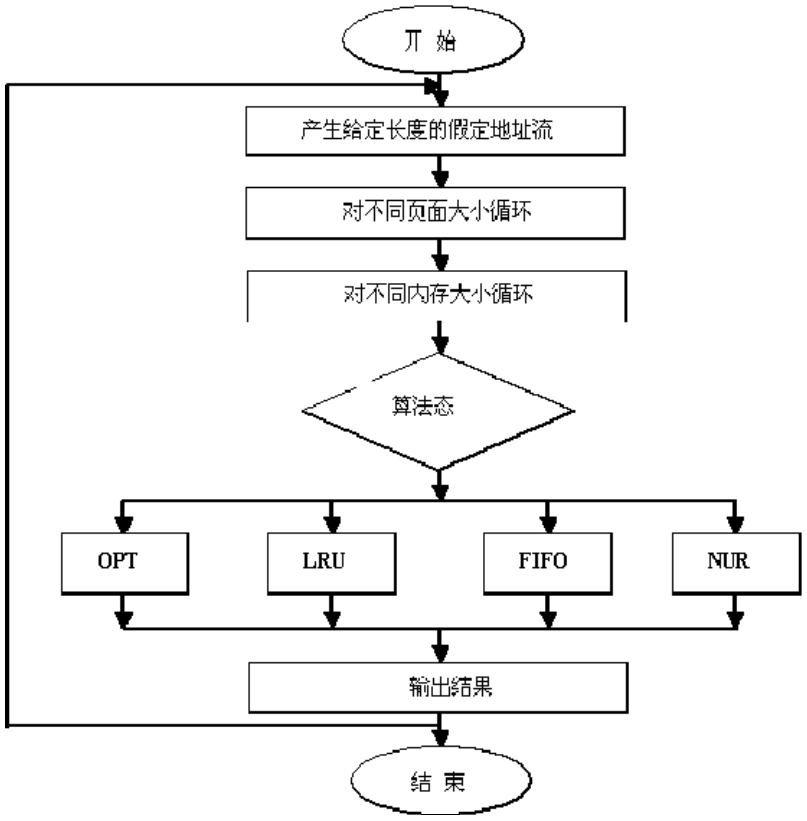
为实现请求调页，核心配置了四种数据结构：页表、页框号、访问位、修改位、有效位、保护位等。

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）



（二）动态分区分配算法（选做）

【实验目的】

1、 熟悉内存自由空闲队列的分配策略
2、 熟悉内存分区的回收原则及实现过程
3、 通过实验深刻理解主存管理的内容

【实验内容】

模拟内存的动态分配和回收，并编程实现。

【实验报告】

1、 列出调试通过程序的清单，并附上文档说明。
2、 总结上机调试过程中所遇到的问题和解决方法及感想。

【实验说明】

实验原理

内存的分配可采用操作系统中的首次适应算法、最佳适应算法或最坏适应算法。
内存的回收回收区与空闲区相邻的各种情况分别处理。

实验五 设备驱动实现

【实验目的】

1、 熟悉 linux 下驱动程序设计

2、 了解 linux 下字符设备驱动设计

实验要求：在 linux 系统下编译相应的内核（linux-2.6.32.69），然后编写设备驱动的程序。

【实验内容】

虚拟内存设备 globalmem 驱动实现，编写程序，然后将生成的驱动模块插入到驱动之中，接着编写测试程序，对设备 globalmem 进行测试。

【实验环境】（含主要设计设备、器材、软件等）

Pc 电脑一台 ，linux （Ubuntu 14.04）

【实验步骤、过程】（含原理图、流程图、关键代码，或实验过程中的记录、数据等）

实验步骤：

1. 建立内核源码树：

1.1 下载相应的内核源代码

下载的方式有两种，一种是直接在终端使用 adt-get install 命令下载。另一种是直接到 www.kernel.org 官网中下载所需的版本我下载的是 inux-2.6.32.69.tar.gz 内核 ，并且将其下载在桌面。然后执行命令 #mv /home/lzb/桌面/linux-2.6.32.69.tar.xz /home

```
root@lzb-desktop:/home# tar -xvf /home/linux-2.6.32.69.tar.xz
```

接着将其解压，

```
linux-2.6.32.69/tools/perf/util/include/linux/module.h
linux-2.6.32.69/tools/perf/util/include/linux/poison.h
linux-2.6.32.69/tools/perf/util/include/linux/prefetch.h
linux-2.6.32.69/tools/perf/util/include/linux/rbtree.h
linux-2.6.32.69/tools/perf/util/levenshtein.c
linux-2.6.32.69/tools/perf/util/levenshtein.h
linux-2.6.32.69/tools/perf/util/map.c
linux-2.6.32.69/tools/perf/util/module.c
linux-2.6.32.69/tools/perf/util/module.h
```

1.2 安装编译内核时需要的工具

总共安装两个软件，都采用 apt-get install 命令安装

build-essential(基本的编程库（gcc, make 等） libncurses5-dev (make menuconfig 要调用的）

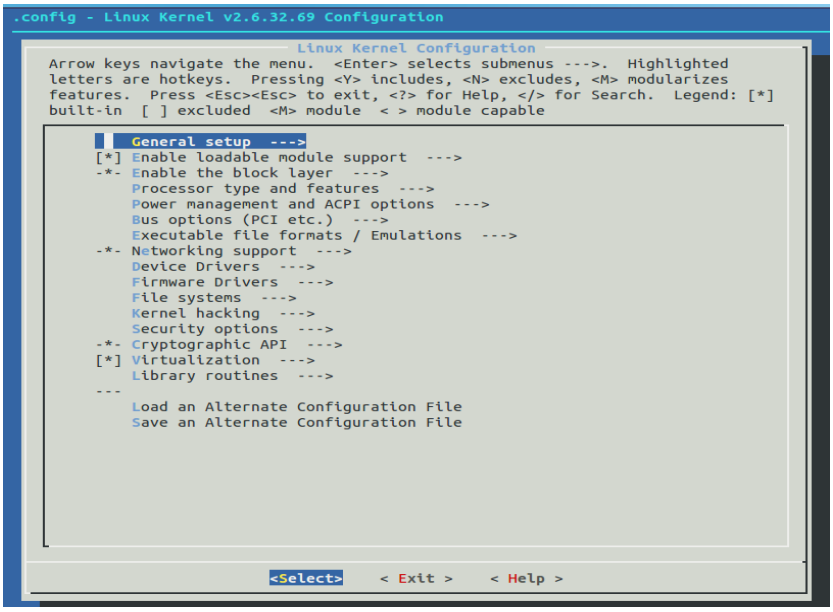
```
root@lzb-desktop:/home# apt-get install libncurses5-dev
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
libncurses5-dev 已经是最新的版本了。
下列软件包是自动安装的并且现在不需要了：
  diffstat hardening-includes libapt-pkg-perl libarchive-zip-perl
  libauthen-sasl-perl libautodie-perl libclass-accessor-perl libclone-perl
  libdigest-hmac-perl libemail-valid-perl libio-pty-perl
  libio-socket-inet6-perl libio-socket-ssl-perl libio-string-perl
  libipc-run-perl libipc-system-simple-perl liblist-moreutils-perl
  libmailtools-perl libnet-dns-perl libnet-domain-tld-perl libnet-ip-perl
  libnet-libidn-perl libnet-smtp-ssl-perl libnet-ssleay-perl
  libparse-debianchangelog-perl libperlio-gzip-perl libsocket6-perl
  libsub-identify-perl libsub-name-perl libtext-levenshtein-perl liburi-perl
  patchutils t1utils
Use 'apt-get autoremove' to remove them.
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 248 个软件包未被升级。
```

1.3 编译相应的内核。

先使用 cd /home/linux-2.6.32.69,进入到内核源码之中。然后执行 sudo make mrproper (清除以前曾经编译过的旧文件),接着将把正在使用中的内核配置文件/boot/config-3.19.0-25-generic 拷贝到/home/linux-2.6.32.69，

```
root@lzb-desktop:/home# cd /home/linux-2.6.32.69
root@lzb-desktop:/home/linux-2.6.32.69# cp /boot/config-3.19.0-25-generic /home/linux-2.6.32.69
```

执行 make menuconfig ，会弹出一个配置界面。这个配置界面就是关于内核的一些详细配置。

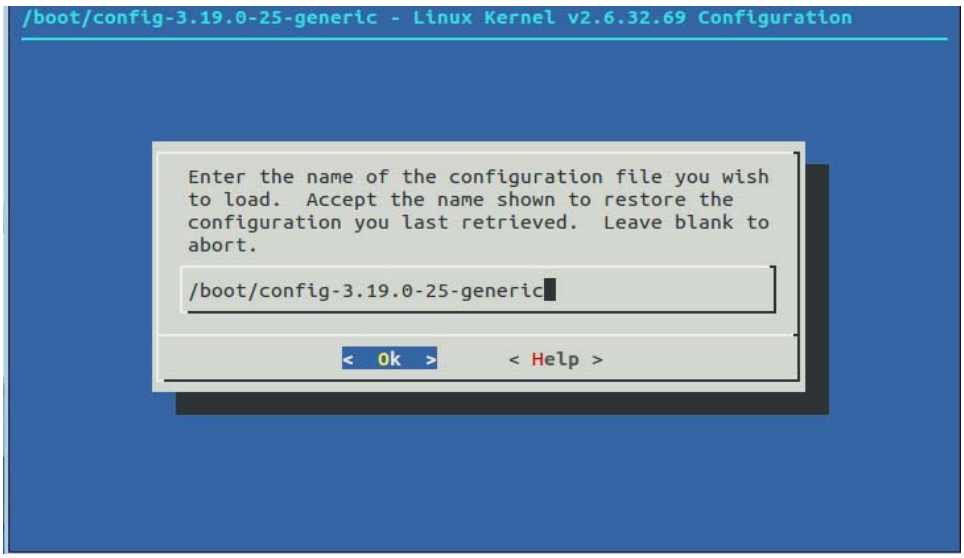


在这里可以看见一些条目，只关注两条：

load a kernel configuration（利用当前的内核配置详单来设置将要编译的内核）

save a kernel configuration

我们直接用刚刚中 boot 中考过来的以及配好的文件进行配置，点击 load a kernel configuration，然后输入/boot/config-3.19.0-25-generic



然点击 ok，退出这个界面进入终端，然后输入 make 指令进行内核编译。

```
root@lzb-desktop:/home/linux-2.6.32.69# make
  CHK     include/linux/version.h
  CHK     include/linux/utsrelease.h
  SYMLINK include/asm -> include/asm-x86
  CALL    scripts/checksyscalls.sh
  CHK     include/linux/compile.h
  VDSOSYM arch/x86/vdso/vdso-syms.lds
  VDSOSYM arch/x86/vdso/vdso32-int80-syms.lds
  VDSOSYM arch/x86/vdso/vdso32-syscall-syms.lds
  VDSOSYM arch/x86/vdso/vdso32-sysenter-syms.lds
  VDSOSYM arch/x86/vdso/vdso32-syms.lds
  LD      arch/x86/vdso/built-in.o
  LD      arch/x86/built-in.o
```

2 编写 globalmem 设备程序

2.1 虚拟内存设备 globalmem 驱动：

在 Linux2.6 内核以前注册字符设备的函数接口是 register_chrdev()，在 2.6 中其可继续使用。

register_chrdev 大致作用：向内核注册 cdev 结构体，当在用户空间打开设备文件时内核可以根据设备号快速定位此设备文件的 cdev->file_operations 结构体，从而调用驱动底层的 open、close、read、write、ioctl 等函数，当我们在用户空间 open 字符设备文件时，首先调用 def_chr_fops->chrdev_open() 函数（所有字符设备都调用此函数），chrdev_open 会调用 kobj_lookup 函数找到设备的 cdev->kobject，从而得到设备的 cdev，进而获得 file_operations。要想通过 kobj_lookup 找到相应的 cdev，必需调用 register_chrdev()函数。向内核注册。

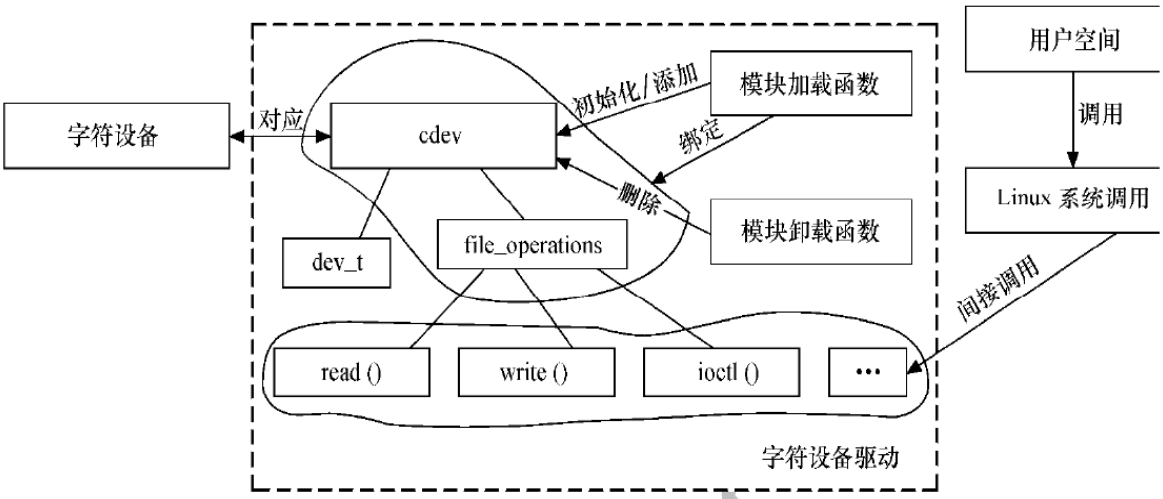


图 6.1　字符设备驱动的结构

globalmem 设备是我们构造的一个虚拟设备，globalmem 意味着“全局内存”，在 globalmem 字符设备驱动中会分配一片大小为 GLOBALMEM_SIZE（4KB）的内存空间，并在驱动中提供针对该片内存的读写、控制和定位函数，以供用户空间的进程能通过 Linux 系统调用访问这片内存。

2.2 编写globalmem驱动程序

1、打开终端输入命令 mkdir /home/user/globalmem　　创建 globalmem 文件夹

2、然后再输入命令 cd /home/user/globalmem　　进入 globalmem 文件夹，

3、接着输入命令　gedit　globalmem.c　　编写globalmem.c程序

在/home/user/globalmem新建文件globalmem.c如下：

```
#include <linux/module.h>
```

```
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <asm/io.h>
#include <asm/system.h>
#include <asm/uaccess.h>

#define GLOBALMEM_SIZE      0x1000    /*全局内存最大 4K 字节*/
#define MEM_CLEAR 0x1    /*清 0 全局内存*/
#define GLOBALMEM_MAJOR 150      /*预设的 globalmem 的主设备号*/

static int globalmem_major = GLOBALMEM_MAJOR;
/*globalmem 设备结构体*/
struct globalmem_dev
{
   struct cdev cdev; /*cdev 结构体*/
   unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
};

struct globalmem_dev *globalmem_devp; /*设备结构体指针*/

int globalmem_open(struct inode *inode, struct file *filp)
{
   filp->private_data = globalmem_devp;
   return 0;
}
int globalmem_release(struct inode *inode, struct file *filp)
{
   return 0;
}
static int globalmem_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd, unsigned long arg)
{
   struct globalmem_dev *dev = filp->private_data;/*获得设备结构体指针*/
   switch (cmd)
   {
      case MEM_CLEAR:
         memset(dev->mem, 0, GLOBALMEM_SIZE);
         printk(KERN_INFO "globalmem is set to zero\n");
         break;

      default:
         return    - EINVAL;
   }
   return 0;
}

static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size, loff_t *ppos)
{
```

```
    unsigned long p =    *ppos;
    unsigned int count = size;
    int ret = 0;
    struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/

    /*分析和获取有效的写长度*/
    if (p >= GLOBALMEM_SIZE)
        return count ?    - ENXIO: 0;
    if (count > GLOBALMEM_SIZE - p)
        count = GLOBALMEM_SIZE - p;

    /*内核空间->用户空间*/
    if (copy_to_user(buf, (void*)(dev->mem + p), count))
    {
        ret =    - EFAULT;
    }
    else
    {
        *ppos += count;
        ret = count;

        printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
    }

    return ret;
}

static ssize_t globalmem_write(struct file *filp, const char __user *buf, size_t size, loff_t *ppos)
{
    unsigned long p =    *ppos;
    unsigned int count = size;
    int ret = 0;
    struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/

    /*分析和获取有效的写长度*/
    if (p >= GLOBALMEM_SIZE)
        return count ?    - ENXIO: 0;
    if (count > GLOBALMEM_SIZE - p)
        count = GLOBALMEM_SIZE - p;

    /*用户空间->内核空间*/
    if (copy_from_user(dev->mem + p, buf, count))
        ret = - EFAULT;
    else
    {
        *ppos += count;
        ret = count;
        printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
    }

    return ret;
}
```

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
{
   loff_t ret = 0;
   switch (orig)
   {
      case 0:     /*相对文件开始位置偏移*/
         if (offset < 0)
         {
            ret =    - EINVAL;
            break;
         }
         if ((unsigned int)offset > GLOBALMEM_SIZE)
         {
            ret =    - EINVAL;
            break;
         }
         filp->f_pos = (unsigned int)offset;
         ret = filp->f_pos;
         break;
      case 1:     /*相对文件当前位置偏移*/
         if ((filp->f_pos + offset) > GLOBALMEM_SIZE)
         {
            ret =    - EINVAL;
            break;
         }
         if ((filp->f_pos + offset) < 0)
         {
            ret =    - EINVAL;
            break;
         }
         filp->f_pos += offset;
         ret = filp->f_pos;
         break;
      default:
         ret =    - EINVAL;
         break;
   }
   return ret;
}

static const struct file_operations globalmem_fops =
{
   .owner = THIS_MODULE,
   .llseek = globalmem_llseek,
   .read = globalmem_read,
   .write = globalmem_write,
   .ioctl = globalmem_ioctl,
   .open = globalmem_open,
   .release = globalmem_release,
};
static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
```

```
{
   int err, devno = MKDEV(globalmem_major, index);

   cdev_init(&dev->cdev, &globalmem_fops);
   dev->cdev.owner = THIS_MODULE;
   dev->cdev.ops = &globalmem_fops;
   err = cdev_add(&dev->cdev, devno, 1);
   if (err)
      printk(KERN_NOTICE "Error %d adding LED%d", err, index);
}
int globalmem_init(void)
{
   int result;
   dev_t devno = MKDEV(globalmem_major, 0);

   /* 申请设备号*/
   if (globalmem_major)
      result = register_chrdev_region(devno, 1, "globalmem");
   else   /* 动态申请设备号 */
   {
      result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
      globalmem_major = MAJOR(devno);
   }
   if (result < 0)
      return result;

   /* 动态申请设备结构体的内存*/
   globalmem_devp = kmalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
   if (!globalmem_devp)       /*申请失败*/
   {
      result =    - ENOMEM;
      goto fail_malloc;
   }
   memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
   globalmem_setup_cdev(globalmem_devp, 0);
   return 0;
fail_malloc: unregister_chrdev_region(devno, 1);
   return result;
}

void globalmem_exit(void)
{
   cdev_del(&globalmem_devp->cdev);     /*注销 cdev*/
   kfree(globalmem_devp);         /*释放设备结构体内存*/
   unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*释放设备号*/
}
MODULE_AUTHOR("Song Baohua");
MODULE_LICENSE("Dual BSD/GPL");
module_param(globalmem_major, int, S_IRUGO);
module_init(globalmem_init);
module_exit(globalmem_exit);
```

在/home/user/globalmem 新建 Makefile 文件如下：

```
ifneq ($(KERNELRELEASE),)
obj-m := globalmem.o
else
    KERNELDIR = /usr/src/linux-headers-2.6.31-14-generic/
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c
endif
```

编译globalmem.c，得到myglobalmem.ko文件。

运行insmod globalmem.ko命令加载模块，通过lsmod命令，发现globalmem模块已被加载。再通过cat /proc/devices命令查看，发现多出了主设备号为150的myglobalmem字符设备驱动，如下所示：

```
[root@EmbedSky /]# sudo insmod globalmem.ko
[root@EmbedSky /]# lsmod
    Not tainted
myglobalmem 3396 0 - Live 0xbf067000
ov9650 11088 0 - Live 0xbf05c000
……
[root@EmbedSky /]# cat /proc/devices
Character devices:
……
150 globalmem
……
[root@EmbedSky /]# sudo mknod /dev/globalmem c 150 0
```

编写应用程序test.c用于测试：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
main()
{
    char str[1024], str2[1024];
    int fd, stop;
    int ret;
    fd = open("/dev/globalmem", O_RDWR, S_IRUSR | S_IWUSR);
    if (fd != -1 )
    {
        printf("Please input the string written to globalmem\n");
        scanf("%s", str);
        printf("The str is %s\n", str);
        ioctl(fd, 1, NULL);
        ret = write(fd, str, strlen(str));
        printf("The ret is %d\n", ret);
        lseek(fd, 0, SEEK_SET);
          scanf("%d", &stop);
        read(fd, str2, 100);
        printf("The globalmem is %s\n", str2);
        close(fd);
```

```
    }
    else
    {
        printf("Device open failure\n");
    }
}
```

编译上述文件：

```
gcc -o test test.c
```

运行

```
$ sudo ./test
Please input the string written to globalmem
qwertyuiop
The str is qwertyuiop
The ret is 10
0
The globalmem is qwertyuiop
```

可以发现"globalmem"设备可以正确的读写。

实验六 文件操作

【实验目的】

1、熟悉 LINUX 文件系统。
2、掌握 LINUX 下文件操作。

【实验内容】

1、编程显示文件自身。
2、编程实现文件加密。
3、编程实现文件的合并。
4、为 LINUX 设计一个简单的二级文件系统。（选做）

【实验报告】

1、列出调试通过程序的清单，给出响应文档说明。
2、总结上机调试过程中所遇到的问题和解决方法及感想。

【实验相关资料】

一、文件系统概念

1. Linux 文件系统布局

Ext2 是 Linux 文件系统类型，它很好地继承了 Unix 文件系统的主要特色，如普通文件的三级索引结构，目录文件的树型结构和把设备作为特别文件等。Linux 文件系统是一个逻辑的自包含的实体，它含有 I 节点，目录和数据块。Linux 将整个磁盘划分成若干分区，每个分区被当做独立的设备对待；一般需要一个主分区 Native 和一个交换分区 swap。主分区用于存放文件系统，交换分区用于虚拟内存。主分区内的空间又分成若干个组，每个组内都包含有一个超级块的拷贝，以及 I 节点和数据块等信息，如下图所示。Ext2 文件卷的逻辑块大小可到 1 K，2 K 和 4 K 三种，当块大小为 1 K 时，每组内包含 1024*8=8192 个逻辑块。

文件卷逻辑块的磁盘存储空间安排如下：

引导块	超级块	I 节点位图	组描述符			…							…		
I 节点表				数据区											

当文件卷为可引导的系统时，引导块的内容为引导信息，否则内容为空。文件系统都以引导块（boot block）开始，引导块中包含有可执行代码。启动计算机时，硬件从引导设备将引导块读入内存，转而执行其代码。引导块代码开始操作系统本身的加载过程。一旦系统启动之后，引导块不再使用。

超级块（Super-block）中含有文件系统的布局信息，其主要功能是给出文件系统不同部分的大小。如给定块大小， I 节点总数，每组内 I 节点数，空闲块和 I 节点数等。在 Linux 启动时，根设备中的超级块被读入内存中，存放在 Struct ext2_super_block 结构中。

```
Struct ext2_super_block

 {
unsigned  long  s_inodes_count;            /*文件卷 I 节点数*/
```

```
unsigned  long  s_blocks_count;          /*文件卷逻辑块总数*/
unsigned  long  s_r_blocks_count;        /*为超级用户保留的块数*/
unsigned  long  s_free_blocks_count;     /*空闲块数*/
unsigned  long  s_free_inodes_count;     /*空闲 I 节点数*/
unsigned  long  s_first_data_block;      /*第一个数据块数*/
…
}
```

在软盘用作 Linux 文件系统之前，实用程序 mkfs 可用来创建文件系统。

```
#mkfs        /dev/fd0         1440
```

该命令在软驱中的软盘上创建 1400 个块的空文件系统，该命令还在超级块中写入魔数（s_magic），表明该文件系统是一个有效的 Linux 文件系统。魔数能表明文件系统的版本。Mount 系统调用检查超级块中的魔数和其它信息，以决定是否安装其它文件系统，如 windows9x 文件系统。

2、位图

Linux 文件系统用位图来管理磁盘块和 I 节点，位图分为块位图和 I 节点位图。块位图占用一个磁盘块，当某位为“ 1 ”时，表示磁盘块空闲，为“ 0 ”时表示磁盘块被占用。 I 节点位图也占用一个磁盘块，当它为“ 0 ”时，表示组内某个对应的 I 节点空闲，为“ 1 ”时表示已被占用。位图使系统能够快速地分配 I 节点和数据块，保证同一文件的数据块能在磁盘上连续存放，从而大大地提高了系统的实时性能。

在创建文件时，文件系统必须在块位图中查找第一个空闲 I 节点，把它分配给这个新创建的文件。在该空闲 I 节点分配使用后，就需要修改指针，使它指向下一个空闲 I 节点。同样地， I 节点被释放后，则需要修改指向第一个空闲 I 节点的指针。

3. I 节点表

I 节点表占用若干个磁盘块，它几乎与标准 Unix 的 I 节点表相同。每个 I 节点占 128 个字节，读入缓冲区后存放在 Struct ext2_inode 中。

```
Struct ext2_node
 {
u16   i_mode;                        /*文件模式* /
u16   i_uid;                         /*文件主用户标识符*/
u_32  i_size;                        /*文件大小*/
u_32  i_atime                        /*最佳访问文件时间*/
u_32  i_stime;                       /*文件创建时间*/
u_32  i_mtime;                       /*文件最近修改时间*/
u_32  i_dtime;                       /*文件删除时间*/
u16   i_links_count;                 /*文件连接计数*/
u16   i_gid;                         /*文件主的用户组标识符*/
u32   i_blocks;                      /*文件的总块数*/
u32   i_flag;                        /*文件标志*/
u32   i_blocd [EXT2_N_BLOCKS] ;      /*文件地址块索引表*/
u32   i_file_acl;                    /*文件访问控制表*/
u32   i_dir_acl;                     /*目录访问控制表*/
…
}
```

I 节点表中的模式信息，给出了文件的类型，如：普通文件，目录文件，块设备文件，字符设备文件，管道文件等。 I 节点的链接数域记录了有多少个目录指向这个 I 节点。这样文件系统就知

道什么时候该释放文件的存储区。

4．目录结构

ext2 采用动态方式管理它的目录，用一个单项链表示它的目录项，每个目录项的数据结构为：

```
Struct ext2_dir_entry
 {
_u32 inode;                    /*对应的 I 节点*/
_u16 inode;                    /*目录项长度*/
_u16 inode                     /*名字长度*/
charname [EXT2_NAME_LEN] ;     /*文件名*/
}
```

当要删除一个目录时，要将目录项中的 I 节点置为“ 0 ”,并把目录项从链表中删除,目录项所占用的空间被合并到前一个目录项空间中。

5．Ext2 程序库

Ext2 程序库提供了大量的例程，能通过直接控制物理设备来操作文件系统。Ext2 程序库使用软件抽象技术以达到最大限度的代码重用。例如,程序库提供了许多不同的可重复调用例程(iterator).程序可以简单将函数传递给 ext2fs_block_iterate(),它能在每个 inode 中被调用。另一种 iterator 函数为同一个目录中的每个文件调用一个用户定义的函数。许多 Ext2 例程（mke2fs、s2fsck、tune2fs、dumpe2fs 、debugfs）使用该 Ext2 程序库。这大大简化了这些例程的维护，因为 Ext2 升级后的新特性只需改变 ext2 库就可以反映出来。因为 Ext2 库可编译成共享库映像文件，所以这种代码重用减小了二进文件的长度。

因为 ext2 库的接口十分抽象和通用，无需考虑物理细节，所以编写需要直接存取 ext2 文件系统的程序很容易。例如，将 BSD 转储和备份恢复的特性转移到 linux 平台时，只需做少量的修改：一些依赖于文件系统的函数需要转移到 Ext2 库。

Ext2 库提供许多种操作。第一类操作是与文件系统相关的操作。程序可以用这些操作打开关闭文件、读写位图、在磁盘上创建新的文件系统，也可管理坏盘块列表。第二类操作用来控制目录，它们能建立和展开目录，增加和移走目录项，能析构文件名，找到 inode 号，也能由 inode 号确定文件名。最后一类操作是与 inode 相关的，它能扫描 inode 表，读写 inode，扫描一个 inode 中所有的盘块。分配和回收例程可以帮助用户程序分配和释放盘块和 inode.

由于 Ext2 核心代码包括多项性能优化，而且 Ext2 也能实现分配优化。所以它们能大大改善 I/O 速度，提高 I/O 组织的灵活性及编程效率。因此，Ext2 核心代码及 Ext2 程序库为开发嵌入式系统及实时应用系统提供了广泛的基础和手段。

二、文件操作

文件的创建和读写

当我们需要打开一个文件进行读写操作的时候，我们可以使用系统调用函数 open。使用完 成以后我们调用另外一个 close 函数进行关闭操作。

该函数使用的头文件如下：

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

定义函数：

```
int open(const char *pathname,int flags);
int open(const char *pathname,int flags,mode_t mode);
int close(int fd);
```

函数说明：

open 函数有两个形式。其中 pathname 是我们要打开的文件名(包含路径名称，缺省是认为在当前路径下面)。flags 可以取下面的一个值或者是几个值的组合。
O_RDONLY：以只读的方式打开文件。
O_WRONLY：以只写的方式打开文件。
O_RDWR：以读写的方式打开文件。
O_APPEND：以追加的方式打开文件。
O_CREAT：创建一个文件。
O_EXEC：如果使用了 O_CREAT 而且文件已经存在,就会发生一个错误。
O_NOBLOCK：以非阻塞的方式打开一个文件。
O_TRUNC：如果文件已经存在，则删除文件的内容。
前面三个标志只能使用任意的一个。如果使用了 O_CREATE 标志，那么我们要使用 open 的第二种形式。还要指定 mode 标志，用来表示文件的访问权限。mode 可以是以下情况的组合。

```
-------------------------------------------------------------
S_IRUSR 用户可以读 S_IWUSR 用户可以写
S_IXUSR 用户可以执行 S_IRWXU 用户可以读写执行
-------------------------------------------------------------
S_IRGRP 组可以读 S_IWGRP 组可以写
S_IXGRP 组可以执行 S_IRWXG 组可以读写执行
-------------------------------------------------------------
S_IROTH 其他人可以读 S_IWOTH 其他人可以写
S_IXOTH 其他人可以执行 S_IRWXO 其他人可以读写执行
-------------------------------------------------------------
S_ISUID 设置用户执行 ID S_ISGID 设置组的执行 ID
-------------------------------------------------------------
```

我们也可以用数字来代表各个位的标志。Linux 总共用 5 个数字来表示文件的各种权限。00000 第一位表示设置用户 ID，第二位表示设置组 ID，第三位表示用户自己的权限位，第四位表示组的权限，最后一位表示其他人的权限。 每个数字可以取 1(执行权限)，2(写权限)，4(读权限)，0(什么也没有)或者是这几个值的和。比如我们要创建一个用户读写执行，组没有权限，其他人读执行的文件。

设置用户 ID 位可以使用的模式是：

```
1. (设置用户 ID)0(组没有设置)7(1+2+4)0(没有权限,使用缺省) 5(1+4) 即 10705
open("temp",O_CREAT,10705);
```

如果我们打开文件成功，open 会返回一个文件描述符。对文件的所有操作就可以对这个文件描述符进行操作。

当我们操作完成以后，我们要关闭文件了，只要调用 close 就可以了，其中 fd 是我们要关闭的文件描述符。

文件打开了以后，就可以对文件进行读写了。可以调用函数 read 和 write 进行文件的读写。

该函数使用的头文件：

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

函数定义：

```
ssize_t read(int fd, void *buffer,size_t count);
ssize_t write(int fd, const void *buffer,size_t count);
```

函数说明：

fd 是我们要进行读写操作的文件描述符，buffer 是我们要写入文件内容或读出文件内容的内存地址，count 是我们要读写的字节数。对于普通的文件 read 从指定的文件(fd)中读取 count 字节到 buffer 缓冲区中(记住我们必须提供一个足够大的缓冲区)，同时返回 count，如果 read 读到了文件的结尾或者被一个信号所中断，返回值会小于 count。如果是由信号中断引起返回，而且没有返回数据，read 会返回-1，且设置 errno 为 EINTR。当程序读到了文件结尾的时候，read 会返回 0。

write 从 buffer 中写 count 字节到文件 fd 中，成功时返回实际所写的字节数。

2. 文件的各个属性

文件具有各种各样的属性，除了我们上面所知道的文件权限以外，文件还有创建时间、大小等等属性，有时侯我们要判断文件是否可以进行某种操作(读,写等等)。这个时候我们可以使用 access 函数。

```
#include <unistd.h>
int access(const char *pathname,int mode);
```

pathname：是文件名称，mode 是我们要判断的属性。可以取以下值或者是他们的组合。

R_OK 文件可以读
W_OK 文件可以写
X_OK 文件可以执行
F_OK 文件存在

当我们测试成功时，函数返回 0，否则如果有一个条件不符时，返回-1。如果我们要获得文件的其他属性，我们可以使用函数 stat 或者 fstat。

```
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name,struct stat *buf);
int fstat(int filedes,struct stat *buf);
struct stat {
dev_t st_dev;                    /* 设备 */
ino_t st_ino;                   /* 节点 */
mode_t st_mode;                 /* 模式 */
nlink_t st_nlink;                    /* 硬连接 */
uid_t st_uid;                   /* 用户 ID */
gid_t st_gid;                    /* 组 ID */
dev_t st_rdev;                   /* 设备类型 */
off_t st_off;                       /* 文件字节数 */
unsigned long st_blksize;       /* 块大小 */
unsigned long st_blocks;         /* 块数 */
time_t st_atime;              /* 最后一次访问时间 */
time_t st_mtime;              /* 最后一次修改时间 */
time_t st_ctime;                 /* 最后一次改变时间(指属性) */
};
```

Stat 用来判断没有打开的文件，而 fstat 用来判断打开的文件。我们使用最多的属性是 st_ mode。通过着属性我们可以判断给定的文件是一个普通文件还是一个目录，连接等等。可以使用下面几个宏来判断：

S_ISLNK(st_mode)是否是一个连接，S_ISREG 是否是一个常规文件，S_ISDIR 是否是一个目录，S_ISCHR是否是一个字符设备,S_ISBLK是否是一个块设备,S_ISFIFO是否是一个FIFO文件,S_ISSOCK是否是一个 SOCKET 文件。

目录文件的操作：

在我们编写程序的时候，有时候会要得到我们当前的工作路径。C 库函数提供了 getcwd 来解决这个问题。

```
#include <unistd.h>
char *getcwd(char *buffer,size_t size);
```

我们提供一个 size 大小的 buffer,getcwd 会把我们当前的路径考到 buffer 中。如果 buffer 太小，函数会返回-1 和一个错误号。

Linux 提供了大量的目录操作函数，我们学习几个比较简单和常用的函数。

```
#include <dirent.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path,mode_t mode);
DIR *opendir(const char *path);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir,off_t off);
int closedir(DIR *dir);
struct dirent {
long d_ino;
off_t d_off;
unsigned short d_reclen;
char d_name[NAME_MAX+1]; /* 文件名称 */
```

mkdir 创建一个目录，opendir 打开一个目录为以后读做准备，readdir 读一个打开的目录，rewinddir 是用来重读目录，closedir 是关闭一个目录，telldir 和 seekdir 类似与 ftee 和 fseek 函数。

附件　　华中农业大学　学生实验报告

专业：　　姓名：　　学号：

日期：　__年__月__日　　成绩_____

课程名称	计算机操作系统	实验名称		实验类型	验证 设计 综合 创新

【实验目的】

实验目的：

实验要求：

【实验内容】

【实验环境】（含主要设计设备、器材、软件等）

Pc 电脑一台 linux

【实验步骤、过程】（含原理图、流程图、关键代码，或实验过程中的记录、数据等）

内容:

实验步骤：

程序说明及实现:

【实验结果或总结】（对实验结果进行相应分析，或总结实验的心得体会，并提出实验的改进意见）