操作系统原理

实用实验教程

张鸿烈　编著

山东大学齐鲁软件学院
山东大学计算机科学与技术学院

前言

在现代计算机系统中，操作系统已经成为整个计算机系统的核心。所有计算机硬件和应用软件都必须依赖操作系统的支撑，接受操作系统的管理。一个性能优秀的操作系统会使整个计算机系统的资源发挥到极致；同时也为用户使用计算机提供了最大可能的方便。因此介绍计算机操作系统原理的课程也就成为了计算机科学中最主要的课程之一。

本实验教程主要是为了配合操作系统原理阶段的实验教学而编写的。实验内容和顺序主要考虑：

- 紧密围绕操作系统原理课的主要教学内容。重点选择操作系统科学领域中经典的、有启发性的实验课题。
- 实验课题的范围能基本覆盖整个操作系统原理课所讲授的内容。
- 实验的顺序能基本对应操作系统原理课讲授的进度。

因为大多数操作系统原理教材都是以 Unix/Linux 系统为主要蓝本讲授，因此本实验教程采用 Linux 系统作为实验环境。

本实验教程安排了两个部分的实验内容：

- 操作系统命令实验
- 操作系统算法实验。

其中操作系统命令实验的目的，一方面是为了让学生体验和熟悉一下经典操作系统最基本的操作界面，另一方面是为了能使不太熟悉 Linux 系统编程环境的学生尽快的了解 Linux 系统最基本的程序开发环境，以便迅速展开操作系统原理的算法实验。因此这一部分的内容并不是对于 Linux 系统应用的全面介绍，仅是围绕操作系统的算法实验可能用到的系统操作安排的实验。

操作系统算法实验是本实验教程重点安排的实验。这一部分共安排了十个在操作系统原理中最为重要的问题作为实验课题，这十个课题基本覆盖了大多数操作系统原理教材所讲授的内容，同时也揭示 Linux 内核最精彩的一些系统功能的用法。每个实验课题都给出了 5 个方面的指导：

- 实验目的　说明本实验所要达到的实验结果，以便学生能够明确做过这个实验后应有哪些收获。
- 实验说明　说明本实验用到的算法的基本原理和实现中用到的详细的 Linux 技术文档，以便学生能够看懂示例程序和利用这些技术文档独立的编程调试。
- 示例实验　给出一个能实现所要实验的算法的完整的程序源代码，包括程序的开发、调试、分析的详细步骤和过程。以便使学生能通过开发调试这个程序去亲身体验算法具体的实现过程和实现效果。
- 独立实验　给出一个与该算法有关但不同于示例实验的并且可以借助于示例实验实现的问题。以便训练学生独立编程调试的能力。
- 实验要求　给出对该实验的过程和结果应做哪些分析和总结的指导。

给出的示例实验程序的源代码在关键语句处均添加了详细的中文注释，以方便学生阅读和理解编程的思路。其中除有少数实验，例如管程，涉及到要封装对象，以及几个用对象更容易说明问题的实验采用了 C++语言编写了程序，其余示例程序均为 C 语言编写。这样的选择主要是考虑到目前操作系统原理教材中算法的讨论大部分采用 C 语言来描述，同时 C 语言也是 Linux 系统的宿主语言，可以直接的与操作系统内核接口；另外也不疏忽使用面向对象的程序设计方法来研究操作系统。

这些示例程序均在 RedHat、Fedora、Ubuntu、SUSE 等系列的 Linux 系统中进行了测试。示例程序的内容和开发不涉及特权用户权限和图形界面环境。以普通用户身份，文本界面进入系统就可以进行实验。因此在单机上没有 Linux 系统环境时，可以采用远程终端方式开展实验。这些示例程序的篇幅都不是很大，可以让学生在 2—3 个学时内调试和分析出来，从而对操作系统中抽象概念有个快速的感性认识。

独立实验的课题都是和示例实验相类似的一些问题，主要为了训练学生独立思考、编程和调试、分析程序的技能。只要学生在真正领会了示例程序的基础上，参考示例程序再通过自己的认真思考后，完成这些实验也是不难的。当然，这些独立实验可以根据教学要求的深度、学生的编程能力、以及实验时间的多少来选作。

由于时间和水平的限制，这本实验教材肯定存在很多缺陷和不足，其中的实验课题的安排和设计可能不是最优。但希望它能对我们的操作系统原理课程的教学有所裨益，对于学生实践操作系统原理，熟悉 Unix/Linux 系统核心编程环境有所帮助。

作者

2008 年 12 月

目录

前言 ..... **I**

第一部分、操作系统命令实验 **6**

实验一、系统的注册与注销 6
1.1 登录系统 6
1.2 获取帮助 7
1.3 虚拟终端 7
1.4 退出系统 7
1.5 关机 7
实验二、文件系统主要命令 8
2.1 文件目录基本结构 8
2.2 常用目录操作命令 8
2.3 常用文件操作命令 9
2.4 文件属性 11
实验三、进程管理主要命令 12
3.1 进程状态查询 12
3.2 进程控制 13
实验四、SHELL 命令控制符 13
4.1 通配符 13
4.2 输入输出重定向 14
4.3 命令控制符 14
4.4 Shell 变量 14
实验五、常用软件开发工具 16
5.1 gcc 和 g++ 语言编译器 16
5.2 make 项目管理器 17
5.3 gdb 程序调试器 18
实验要求 20

第二部分 操作系统算法实验 **21**

实验一、进程控制实验 21
1.1 实验目的 21
1.2 实验说明 21
1.3 示例实验 23
1.4 独立实验 27
1.5. 实验要求 27
实验二、线程和进程/线程管道通信实验 28
2.1 实验目的 28
2.2 实验说明 28
2.3 示例实验 30
2.4 独立实验 35
2.5 实验要求 35

实验三、进程调度算法实验 36
3.1 实验目的 36
3.2 实验说明 36
3.3 示例实验 37
3.4 独立实验 39
3.5 实验要求 39
实验四、进程同步实验 40
4.1 实验目的 40
4.2 实验说明 40
4.3 示例实验 44
4.4 独立实验 55
4.5 实验要求 60
实验五、进程互斥实验 61
5.1 实验目的 61
5.2 实验说明 61
5.3 示例实验 61
5.4 独立实验 73
5.5 实验要求 77
实验六、死锁问题实验 78
6.1 实验目的 78
6.2 实验说明 78
6.3 示例实验 78
6.4 独立实验 89
6.5 实验要求 94
实验七、内存页面置换算法实验 95
7.1 实验目的 95
7.2 实验说明 95
7.3 示例实验 95
7.4 独立实验 102
7.5 实验要求 102
实验八、磁盘移臂调度算法实验 103
8.1 实验目的 103
8.2 实验说明 103
8.3 示例实验 103
8.4 独立实验 108
8.5 实验要求 108
实验九、文件系统接口实验 109
9.1 实验目的 109
9.2 实验说明 109
9.3 示例实验 111
9.4 独立实验 116
9.5 实验要求 116
实验十、分布式系统实验 117
10.1 实验目的 117
10.2 实验说明 117

10.3 示例实验.....120
10.4 独立实验.....126
10.5 实验要求.....126
附录 A 操作系统原理实验建议.....127
附录 B 操作系统原理实验报告纲要.....129
附录 C 操作系统原理实验报告样例.....131
参考文献:.....136

第一部分、操作系统命令实验

本部分实验指导仅对常用的 shell 命令的使用以及 Liunx 操作系统环境作一些简明介绍，以便使学生能够快速适应 Linux 操作系统的工作环境，迅速展开在 Linux 操作系统环境下的操作系统算法实验。若要全面地去掌握 shell 的详细操作可将本节介绍的内容作为 shell 命令练习的提纲，再结合 shell 的专用教程和 Linux 系统联机帮助进一步的去学习。其中介绍的 shell 语言为 Bash-shell 语言。

实验一、系统的注册与注销

实验目的

练习进入和退出系统的操作；学习 linux 联机帮助命令的使用，学会怎样利用借助联机帮助命令随时查阅系统说明文档。

1.1 登录系统

linux 是一个多用户多任务的操作系统。多个用户可以同时使用一台机器，每个人又可以同时启动多个程序。为了区分不同的用户，他们多有自己独立的用户帐号。例如系统启动后显示：
login:
password:

第一次登录是时，必须用系统安装时确定的 root 用户及口令以系统管理员身份登录，然后由系统管理员建立其他用户的账号和口令，这样其他用户才可以以他们各自的账号和口令登录系统。登录成功后如果你使用的是文本界面系统在显示器上会显示 linux 命令解释程序 shell 准备好提示符。如果您使用的是图形界面系统会显示 linux 系统桌面，需要您打开 linux 控制台终端窗体才能使用 shell 命令。shell 命令提示符的符号会根据用户的身份和使用的 shell 命令的种类而不同。一般系统管理员为#，使用 Bash-shell 的用户为$。此时系统输入光标已经定位在 shell 提示符右边您可以在该命令行上输入操作命令了。Shell 命令取分大小写字符。命令的语法一般为：

$命令动词 [–选项符 1 –选项符 2 …] [命令参数 1 命令参数 2 …]

其中各项使用空格键分开，回车键结束。
例如，列出文件 hello 的详细属性信息的命令可以输入命令：
$ls –l hello
这里 ls 是命令动词（它是列表 list 的缩写）；-l 是选项符，表示要列出详细信息；hello 是 ls 的命令参数，它是一个您想要了解的文件名。这将列出 hello 文件详细的属性信息。

1.2 获取帮助

linux 带有联机手册，可以用 help 命令和 man 命令随时查阅系统的内部/外部命令及系统调用的语法。
例如要查阅内部命令 history 的用法可以输入命令：
$help history
help 命令会显示 history 命令的详细使用方法。
又例如，要查询外部命令 ls 的使用语法可以输入命令：
$man ls
man 命令将显示 ls 命令的详细使用方法。
$man –a sleep
man 命令将显示所有与 sleep 相关的系统文档。

1.3 虚拟终端

微机系统只有一个显示终端，但微机上的 linux 可提供 6 个虚拟字符终端和一个图形终端，模拟 7 个用户同时联机操作。用户可以通过 Alt+Ctrl+F1 切换到第一个虚拟字符终端，用第一个用户名登录系统；Alt+Ctrl+F2 切换到第二个虚拟字符终端，用第二个用户名登录系统；直道 Alt+F6 切换到第六个虚拟字符终端，用第六个用户名登录系统；Alt+Ctrl+F7 切换到图形界面终端，它也是 linux 系统启动时默认的显示终端。在图形界面中又可启动无数个 Shell 窗体终端和各种图形窗体终端。这些机制体现出了 Linux 系统的多用户多任务的高级特征。

1.4 退出系统

要注销当前账号，或换一账号重新登录系统有多种方法，可以使用 exit 或 logout，也可以同时键入 Ctrl+D。例如：
$exit
login:
系统将注销您的当前账号，再次等待您登录系统。

1.5 关机

关闭系统或重新启动系统，可以使用命令 halt、reboot 或 shutdown 命令，也可以同时使如 Ctrl+Alt+Del 键。

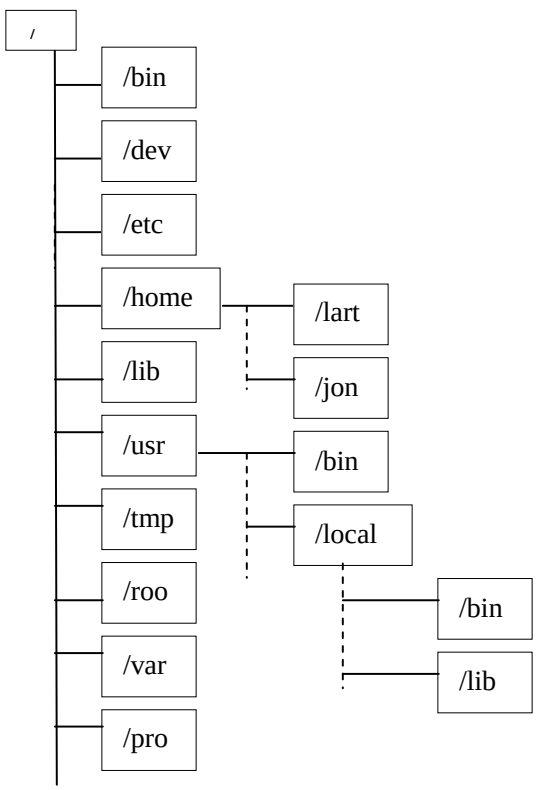
实验二、文件系统主要命令

实验目的

熟悉 linux 文件系统的基本结构和主要的使用方法。

2.1 文件目录基本结构

linux 文件系统是多级树形结构。典型 linux 文件系统大致的结构如下图：



Linux 系统的根目录表示符号为“/”。多级目录的各级目录也用“/”分割，组成一个完整的路径名。例如，当用户首次登录系统时当前工作目录为该用户的主目录，一般为/home/用户账号名目录。当前目录表示符号为“.”，上级目录表示符号为“..”。

2.2 常用目录操作命令

- 显示当前工作目录全名
  pwd
- 改变当前工作目录
  cd [路径名]

- 建立新目录
  mkdir　　路径名
- 删除一个空目录
  rmdir　　路径名
- 列出目录或文件属性信息
  ls　[选项] [路径名|文件名]

2.3 常用文件操作命令

- 串联显示文本文件内容
  cat　[文件名 1 文件名 2　…]
  例如，demo1 文件中仅有一行文字“Ok”，demo2 文件中仅有一行文字“hello”。显示 demo1 和 demo2 两文件的内容：
  $ *cat demo1 demo2*
  Ok
  hello
  可以利用该命令将多个文件合并。例如，将文件 demo1 和 demo2 合并为 demo
  $ *cat demo1 demo2　> demo*
- 拷贝文件
  cp [选项] 源文件名 目标文件名
  可以借助统配符递归的拷贝整个目录。例如,将 sub 子目录中文件全部拷贝到当前目录中：
  $　cp -rv　./sub/. ./
  “./sub/./demo1” -> “././demo1”
  “./sub/./demo” -> “././demo”
  “./sub/./demo2” -> “././demo2”
- 文件换名或文件移动
  mv [选项] 旧文件名 新文件名
- 删除文件
  rm [选项] 文件名
- 对文件内容进行排序
  sort　[选项] [文件名]
  例如，按字典升序排序列出文件 demo 的内容：
  $ *sort demo*
  hello
  Ok
- 在文件中查找给定的字符串或正则表达式
  grep [选项] 要查找的字符串或正则表达式 文件名
  例如，要在当前目录之下递归的查找哪些文件中哪一行上有“hello”一词，可输入命令：
  $ *grep -r -n "hello" ./*
  ./sub/demo:2:hello
  ./sub/demo1:1:hello
  ./hello.c:5:hello
  grep 查出在子目录 sub 的 demo 文件的第 2 行和当前目录的 hello.c 文件的第 5

行有”hello”一词

- 按类型查找文件

  find [路径名] [类型] 文件名

  例如，要查找当前目录下是否有叫“demo”的文件：

  $ *find ./ -name demo*

  ./sub/demo

  find 在当前目录的子目录中找到有一个叫“demo”的文件。

- 文件归档和恢复

  tar [选项] 归档文件名[源文件名]

  例如，将当前目录中的子目录 sub 压缩归档为文件 sub.tar：

  $ *tar -cvzf sub.tar sub*

  sub/

  sub/demo

  sub/demo1

  sub/demo2

  将文件 sub.tar 解压恢复到当前目录中：

  $ *tar -xvzf sub.tar sub*

  sub/demo

  sub/demo1

  sub/demo2

- 创建或编辑文本文件

  vi [文件名]

  vi 编辑器为全屏幕文本键盘操作编辑器，无鼠标编辑功能。具有命令态和编辑两种编辑状态。初始启动时为命令态，此时键入的所有字符都被解释为 vi 子命令，无字符图形显示，仅有命令效果。

  常用 vi 子命令：

  - i 向屏幕缓冲区当前光标前插入文本。此时 vi 状态自动切换为编辑状态其后键入的字母数字都可以显示在屏幕上，同时键盘右边编辑小键盘全部激活，可利用它们定位光标位置进行屏幕编辑。输入 Esc 键将结束插入编辑状态返回命令状态。
  - a 向屏幕缓冲区当前光标后插入文本。其他功能与 i 命令相同。
  - : 进入提示行子命令。此时光标进入屏幕最下行一冒号后，准备接受文件操作子命令。
  - 常用文件操作子命令：
    - w [文件名]

      如果 vi 启动时指定文件名，则可以不要[文件名]，编辑的文本将写入启动时指定文件名，否则写入 w 命令指定的文件。
    - r 文件名

      将 r 子命令指定的文件内容读入屏幕编辑缓冲区。
    - q 或 q!
    - 退出 vi 编辑命令。如果带有惊叹号则不保存当前编辑内容强行退出。

2.4 文件属性

文件的属性是对文件进行管理的一组控制信息。常见文件属性有：

- 文件类型

linux 中文件被划分为 3 种基本类型：普通文件(代表符号-)、目录文件(代表符号 d)、设备文件(块设备代表符号 b，字符设备代表符号 c)

- 文件权限

linux 中每个文件都具有 3 种用户权限和 3 种操作权限。
用户权限有：文件主权限(owner)、同组用户权限(group)、其他用户权限(other)
操作权限有：读文件权限 r、写文件权限 w、执行文件权限 x

- 文件的链接数

linux 中文件可以通过使用 ln 命令给文件起多个别名，来共享一个文件。共享次数称为链接计数，对该文件的删除仅当链接计数为 0 时才进行，否则仅对该链接计数值减 1。

- 所属文件主账号
- 所属用户组号
- 文件长度
- 文件创建或修改的日期和时间
- 文件名

可以从 ls –l 命令的输出中看到这些文件属性的信息。例如：

```
$ls -l
drwxr-xr-x    2    root root 1024      Aug 18 02:50 bin
-rw-r—r--     1    root roor 849       Aug 18 03:00 junk
```

以上列出的第一行文件信息表示该文件为目录文件，文件主权限为可读可写可执行，同组用户权限为可读可执行，其他用户权限为可读可执行，文件链接计数为 2，文件主是 root,,文件属于 root 用户组，文件长度为 1024 字节，文件建立日期为 8 月 18 日 2 点 50 分，文件名为 bin
第二行文件信息表示该文件为普通文件，文件主权限为可读可写不可执行，同组用户权限为只读不可写不可执行，其他用户权限为只读不可写不可执行，文件链接计数为 1，文件主是 root,,文件属于 root 用户组，文件长度为 849 字节，建立日期为 8 月 18 日 3 点 00 分，文件名为 junk。

文件的操作权限属性可以用 chmod 来修改。chmod 的第一种语法为：
chomd {a,u,g,o} [+,-,=] {r,w,x} 文件名
这里，a，u，g，o 分别代表所有用户，文件主，文件同组者，其他用户。
+，-，= 分别代表添加、删除、赋予权限。
r，w，x 分别代表可读、可写、可执行。
chmod 的第一种语法为：
chomd nnn 文件名
这里 nnn 代表 3 个八进制数字，每个分别对应文件主，文件同组者，其他用户。
每个八进制数的 3 位二进制数又分别代表读，写，执行权限。1 表示添加该权限，0 表示删除该权限。例如：

- 使所有用户都具有对文件 junk 的读权限
  $chmod a +r junk
- 使文件主对文件 junk 具有执行权限，其他用用户只有读权限
  $chmod 744 junk

实验三、进程管理主要命令

实验目的

观察和了解 linux 系统中进程的运行情况，熟悉 linux 系统中主要进程管理和进程控制命令的用法。

3.1 进程状态查询

linux 是一个多任务多用户系统，即这种系统可以使一个中央处理器以极短的时间轮流执行多个任务的进程，宏观上产生多个任务在并发执行的效果。这些并发的进程又可分为前台进程和后台两类进程。通常前台进程是一些需要人机交互的进程，如文本编辑、图像编辑等；而后台进程是一些无需人机交互的进程，如数值计算、文件打印、系统服务等等。shell 提供了一些控制并发进程的命令。可以使用这些命令监控和管理任务的执行和联机用户的工作情况。

常用的进程控制命令：

- 显示当前各个进程的工作信息命令 ps
  例如要显示系统当前所有进程的详细信息，可输入命令：

$ *ps aux*

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	1984	652	?	S	08:44	0:01	init [5]
root	2	0.0	0.0	0	0	?	SN	08:44	0:00	[ksoftirqd/0]
root	3	0.0	0.0	0	0	?	S	08:44	0:00	[watchdog/0]
root	4	0.0	0.0	0	0	?	S<	08:44	0:00	[events/0]
root	5	0.0	0.0	0	0	?	S<	08:44	0:00	[khelper]
root	6	0.0	0.0	0	0	?	S<	08:44	0:00	[kthread]
root	8	0.0	0.0	0	0	?	S<	08:44	0:00	[kblockd/0]
root	9	0.0	0.0	0	0	?	S<	08:44	0:00	[kacpid]
root	66	0.0	0.0	0	0	?	S<	08:44	0:00	[khubd]

.....

以上列表各列信息的含义为:

USER 代表进程拥有者
PID 进程号
%CPU 进程自上次切换已来占用CPU时间的百分比
%MEM 进程占内存总量的百分比
VSZ 虚存的占有量
RSS 实存的占有量
TTY 所在终端号
STAT 进程状态(R执行,S睡眠)
START 开始时间

TIME　　执行时间
COMMAND　命令名

- 动态跟踪显示当前各个进程的工作信息

$ *top*

```
top - 10:25:04 up  1:40,  3 users,  load average: 0.13, 0.20, 0.22
Tasks: 127 total,   1 running, 125 sleeping,   0 stopped,   1 zombie
Cpu(s): 21.9% us,  5.3% sy,  0.0% ni, 72.8% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    515852k total,   502360k used,    13492k free,    42828k buffers
Swap:  1048568k total,      132k used,  1048436k free,   196892k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
3723 root     15   0 81424  19m  11m S  3.6  3.9   0:10.60 gnome-terminal
3194 root     15   0 68416  11m 7564 S  2.3  2.4   2:07.14 Xorg
4638 root     16   0  2116 1024  792 R  0.7  0.2   0:00.28 top
2900 root     16   0  6244 1192  648 S  0.3  0.2   0:00.29 crond
3387 root     15   0 31432 7496 6356 S  0.3  1.5   0:05.65 gnome-settings-
   1 root     16   0  1984  652  564 S  0.0  0.1   0:01.30 init
.....
```

top命令不断报告当前系统的处理器、内存、对换空间和各进程的使用信息。其中各列信息的意义同ps命令。可用Ctrl+C键中断top命令的执行。

3.2　进程控制

- 终止指定进程的工作或向进程发送指定的信号
  kill [-s 信号]　进程号
  例如，终止进程号为 4676 的应用程序的执行：
  $ *kill 4676*
- Ctrl+C 打断前台进程的工作
- 将后台进程转换为前台进程
  fg 后台进程名
- Ctrl+Z 将前台进程暂停，并放入后台
- 启动暂停的后台进程重新运行
  bg 后台进程名

实验四、shell 命令控制符

实验目的

Shell 不仅仅是一些操作命令的集合，它还具一组系统控制符和内部变量以及流程控制子命令。可以配置系统环境和控制系统运行自动化。因此 Shell 可进行程序设计. 所以准确地讲，它应当称为 Shell 命令编程语言。本节不对 Shell 编程进行介绍，仅对一些常用的 Shell 控制符和内部变量进程简要介绍。以便快速了解怎样利用 Shell 控制符和内部变量配置和管理操作系统环境。

4.1 通配符

- 通配符“*”，代表文件名中任意长度的字符串。例如：

$ls a*
将列出当前目录中以 a 开头的所有文件名

- 通配符“?”，代表文件名中任一字符。例如：
  $ls a?.c
  将列出当前目录中以 a 开头的两个字符的文件名。
- 通配符“[]”括起的一组范围字符，代表文件名中属于该范围的任意一组字符串。例如：ls a[0-9]
  将列出当前目录中以 a 开头其后为数字的所有文件名

4.2 输入输出重定向

linux 系统中默认的输入为键盘，输出为显示器。称为标准输入（程序中缩写为 stdin）和标准输入（程序中缩写为 stdout）。在 shell 命令中可以通过输入输出重定向符改变输入和输出的来源和目标，简称 I/O 重定向。shell 还可以使用一种称为“管道”的符号连接一个命令的输出到另一个命令的输入。

- 输入重定向符“<”,表示输入来自其右边说明的文件或设备。例如：
  $read < mytext
  不是从键盘而是从 mytext 文件读入内容。
- 输出重定向符“>”, 表示输出将创建并发向其右边说明的目标，例如：
  $ls > filelist
  将创建文件 filelist，并将列出的当前目录的文件名存入其中。
- 输出重定向符“>>”, 表示将追加输出到其右边说明的目标，例如：
  $ls >> filelist
  将列出的当前目录文件名追加存入 filelist 中。
- 管道操作符“|”, 表示连接一个命令的输出到另一个命令的输入，例如：
  $ls | sort | lp
  将列出的当前目录的文件名排序后送入打印队列。
- I/O 重定向和管道可以组合使用。例如：
  $ls >filelist | sort
  ls 列出的当前目录的文件名存入文件 filelist 后，再通过 sort 命令排序显示。

4.3 命令控制符

shell 命令可以一次发出多个，也可以控制命令启动后在后台执行。

- 命令连接符“;”可以在一行上连发多个命令。例如：
  $ls ; ls –l ; ls –l –a
  先以短格式列出文件名，再以长格式列出文件名，再以长格式列出所有文件名。
- 命令后台启动符“&”，表示启动的命令在后台执行。例如：
  $grep “argv” *.c & ; vi
  首先在后台启动查询命令在当前目录的所有 C 文件中查找字符串“argv”，然后在前台启动 vi 文本编辑命令编写文件。

4.4 Shell 变量

shell 不仅是命令解释器，而且也是一种功能强大的命令程序设计语言，可以用它来编写控制多任务处理的程序，定制系统工作环境。因此，shell 允许定义和使用

变量。Shell 内部提供的一些全局性的变量定制了的系统工作环境，称为环境变量。用户可以通过改变这些环境变量定制自己的工作环境，以及与系统内核通讯。在此我们不对 shell 编程进行练习，主要熟悉和练习 shell 提供的一些常用的环境变量。

引用 Shell 变量的值时需要前导一个“$”符号，系统环境变量一般为大写字母。经常使用命令 echo 显示变量的值。例如，显示命令搜索路径：

echo $PATH

Shell 变量赋值时不用前导“$”,赋值号为“=”。Shell 变量值一律为字符串，所以 shell 变量无需说明数据类型。例如，重新设定命令搜索路径：

PATH = “/usr/local/bin:/home/mydir:”

可以使用 set 命令察看当前设定的 shell 变量

例如，常用的环境变量：

- 用户主目录：HOME
- 用户名：USER
- 计算机名：HOSTNAME
- 计算机类型：HOSTTYPE
- 终端类型：TERM
- 命令搜索路径：PATH
- 当前使用的 shell 的路径：SHELL
- shell 一级提示符：PS

实验五、常用软件开发工具

实验目的

Linux 环境中有众多的软件开发工具。其中的 C 语言编译器 gcc 可以编译和构造 Linux 操作系统内核，是与操作系统关系最直接的语言开发工具，也是我们进行操作系统算法实验首选的开发工具。因此本节实验的主要目的就是要熟悉 gcc 编译器及其相关的管理和调试工具。

5.1 gcc 和 g++ 语言编译器

gcc 能够支持多种 C 语言的变体，例如 K&R C 和 ANSI C；GCC 也是一个交叉平台编译器，能够开发不同 CPU 体系结构的软件；同时，GCC 也能够进行代码优化，提高执行程序的运行速度。g++是构建于 gcc 基础上的 C++语言编译器。

gcc 编译过程分为 4 个阶段：

- 预处理
- 编译
- 汇编
- 连接

最简单的 C 语言编译的例子：

用 vi 建立一个 hello.c 文件

```
$vi hello.c
```

输入字符 i,插入文本以下文本

```
/*
* hello.c
  */
#include <stdio.h>
int main(void)
{
      printf("Hello World!\n");
      return 0;
}
```

最后输入字符<Esc>:wq,返回命令行，键入以下编译命令：

```
$gcc hello.c
```

如果没有错误 gcc 将生成默认的可执行文件 a.out,执行 a.out:

```
$./a.out
Hello World!
$
```

gcc 带有多达数页的编译选项，我们仅列出最常用的几项：

-o 可执行文件名	指定输出的可执行文件名，而不是默认的 a.out
-c	只编译生成.o 的目标文件,不连接生成可执行文件
-s	只编译生成.s 的汇编文件,不连接生成可执行文件
-g	在可执行文件中加入标准调试信息
-Wall	允许 GCC 发出警告型错误信息

选项使用的例子:
对以上 hello.c 使用-o,-g 常用选项重新编译、执行:
$gcc -g hello.c -o hello
$./hello
Hello World!
$
GCC 默认的扩展文件名：

.c	C 语言源代码
.C .cc	C++语言源代码
.i	预处理后的 C 语言源代码
.ii	预处理后的 C++语言源代码
.S .s	汇编语言源代码
.o	编译后的目标代码
.a .so	编译后的库代码

5.2 make 项目管理器

make 项目管理器（GNU 中的名称为 gmake）可以根据项目开发者说明的项目开发文件 Makefile 自动的进行编译配置和重复编译,能实现复杂项目的编译自动化。
项目开发文件 Makefile 的编写使用以下规则：

```
目标体 1：    依赖体 1 [依赖体 2 [...]]
              命令行 1
              命令行 2
              [...]
目标体 2：    依赖体 1 [依赖体 2 [...]]
              命令行 1
              命令行 2
              [...]
              [...]
```

其中目标体是命令行要生成的输出文件，依赖体是命令行要输入的文件或选项，命令行序列是要创建目标体文件所需要的步骤，例如编译命令。无特别指定，make 总是使用当前目录中的 Makefile 进行自动编译。
例如我们在当前目录中有两个项目开发文件 hello.c 和 hello.h，则 Makefile 文件可以编写为：

```
hello:          hello.o
                gcc hello.o -o hello
hello.o:        hello.c hello.h
                gcc -s hello.c
                gcc -c hello.c
clean:
                rm hello *.o *.s
```

make 命令的使用：
$gmake
输入 make 或 make hello 将生成 Makefile 中所有的目标文件，即 hello,hello.o,hello.s。
$gmake hello.o

将仅生成目标文件 hello.o 和 hello.s
$gmake clean
是一条伪目标生成命令，该目标没有依赖体，它只执行对已生成目标文件的删除。当我们对以上依赖体中的任意一个进行了修改，重新 make 时仅会引发对应目标体的重新生成，从而提高了编译的效率并保证了项目开发的正确性。

5.3 gdb 程序调试器

如果您在 gcc 编译选项中用到了-g 调试选项，则编译出的可执行文件就会带有符号表。这样的程序就可以使用 gdb 跟踪调试，观察到它的高级语言源代码的执行过程和变量的中间结果，从而能快速的排除程序运行时发生的错误。
以下是一个带有运行时错误的C程序,注意程序想通过传地址方式在一个函数中为字符变量 C 赋一个字符，但它引用了一个空指针，这将引发运行时的段非法错误使得程序异常终止 。但我们可以通过 gdb 跟踪到它产生错误的位置，从而分析出产生错误的原因。

```
/*
* debugmy.c
*/
#include <stdio.h>
void myputc(char * cptr)
{
      *cptr = 'a';
      printf("myputc=%c\n",*cptr);
  }
int main(void)
{
    char c;
    char * cptr;
    c = 'A';
    myputc(cptr);
    return 0;
}
```

使用带-g 选项的 gcc 编译、执行：

```
$gcc -g debugmy.c -o debugmy
$ ./debugmy
段错误
$
```

使用 gdb 跟踪查错

```
$ gdb ./debugmy
GNU gdb Red Hat Linux (6.3.0.0-1.122rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".
```

(gdb)
现在进入了 gdb 调试状态，可以使用 gdb 的调试 子 命令跟踪程序的执行。Gdb 常用命令：
list [行号]　　列出指定行号的上下行（缺省为 10 行）
break [源程序文件名：] 行号　　建立一个断点
run　　启动被调试的程序
next　　从断点处向下执行一行
step　　从断点处向下执行一行，当前行为函数则跟踪进入函数
continue　　继续从断点处连续执行
print 变量名 打印变量当前值
quit　　退出 gdb
让我们现使用 list 命令查看一下要调试的程序是否已经装入，输入：
(gdb)list 10
```
5           void myputc(char * cptr)
6           {
7                *cptr = 'a';
8                printf("myputc=%c\n",*cptr);
9            }
10          int main(void)
11          {
12              char c;
13              char * cptr;
14              c = 'A';
```
我们将断点设在第 10 行上，输入：
(gdb) break 15
Breakpoint 1 at 0x80483c0: file debugmy.c, line 15.
开始跟踪执行，输入：
(gdb) run
Starting program: /root/ipc/debugmy
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000

Breakpoint 1, main () at debugmy.c:15
15　　myputc(cptr);
程序执行到第 15 行上停止，我们采用单步执行跟踪错误的发生，输入:
(gdb) step
myputc (cptr=0x9bbe40 "U\211\203WVS\203\215L$\215s") at debugmy.c:7
7　　*cptr = 'a';
程序执行一行，进入函数 myputc,再单步执行一行，再次输入：
(gdb) step
Program received signal SIGSEGV, Segmentation fault.
0x0804838d in myputc (cptr=0x9bbe40 "U\211\203WVS\203\215L$\215s") at debugmy.c:7
7　　*cptr = 'a';
此时 gdb 报告在执行改行时接受到一个段失败的信号，由此我们可以知道错误发生在该行上，进一步我们可以推断出该错误的发生是因为指针 cptr 未初始化，它指向

了一个非法的地址，所以在向它指向的单元赋值时发生了段错误。
改正它：

```
int main(void)
{
    char c;
    char * cptr;
    c = 'A';
    cptr = &c ;
    myputc(cptr);
    return 0;
}
```

再次编译运行这个程序，

```
$gcc -g debugmy.c -o debugmy
$ ./debugmy
myputc=a
$
```

我们终于得到了正确的结果。

实验要求

记录每步实验中出现的结果，分析输出结果，如果是错误的排除错误，如果是正确的说明您使用的命令完成那些操作系统功能。您将怎样利用这些功能。将您在实验中操作的命令信息、命令输出信息进行总结分析，比较 shell 命令工作方式和可视化系统图形命令操作的优劣。将分析结果写成实验报告

第二部分 操作系统算法实验

本部分的实验内容主要是为了加深理解和掌握操作系统原理中的经典算法而安排的。其中不涉及到要封装对象的实验全部用 C 语言编写。涉及到要封装对象的实验使用 C++语言编写。实验环境均为 Linux 操作系统，开发工具为 gcc 和 g++。

实验一、进程控制实验

1.1 实验目的

加深对于进程并发执行概念的理解。实践并发进程的创建和控制方法。观察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习 Linux 系统中进程创建与控制有关的系统调用的编程和调试技术。

1.2 实验说明

1）与进程创建、执行有关的系统调用说明

进程可以通过系统调用 fork()创建子进程并和其子进程并发执行.子进程初始的执行映像是父进程的一个复本.子进程可以通过 exec()系统调用族装入一个新的执行程序。父进程可以使用 wait()或 waitpid()系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

fork()系统调用语法:

```
#include <unistd.h>
pid_t fork(void);
```

fork 成功创建子进程后将返回子进程的进程号,不成功会返回-1.

exec 系统调用有一组 6 个函数,其中示例实验中引用了 execve 系统调用语法:

```
#include <unistd.h>
int execve(const char *path,   const char *argv[],   const char * envp[]);
```

path 要装入的新的执行文件的绝对路径名字符串.

argv[] 要传递给新执行程序的完整的命令参数列表(可以为空).

envp[] 要传递给新执行程序的完整的环境变量参数列表(可以为空).

Exec 执行成功后将用一个新的程序代替原进程，但进程号不变，它绝不会再返回到调用进程了。如果 exec 调用失败，它会返回-1。

wait() 系统调用语法:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid,int *status,int option);
```

status 用于保留子进程的退出状态

pid 可以为以下可能值：

-1 等待所有 PGID 等于 PID 的绝对值的子进程

1 等待所有子进程

0 等待所有 PGID 等于调用进程的子进程

>0 等待 PID 等于 pid 的子进程

option 规定了调用 waitpid 进程的行为：

WNOHANG 没有子进程时立即返回

WUNTRACED 没有报告状态的进程时返回

wait 和 waitpid 执行成功将返回终止的子进程的进程号，不成功返回-1。

getpid()系统调用语法:

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

getpid 返回当前进程的进程号，getppid 返回当前进程父进程的进程号

2) **与进程控制有关的系统调用说明**

可以通过信号向一个进程发送消息以控制进程的行为。信号是由中断或异常事件引发的，如：键盘中断、定时器中断、非法内存引用等。信号的名字都以 SIG 开头，例如 SIGTERM、SIGHUP。可以使用 kill -l 命令查看系统当前的信号集合。

信号可在任何时间发生，接收信号的进程可以对接收到的信号采取 3 种处理措施之一：

- 忽略这个信号
- 执行系统默认的处理
- 捕捉这个信号做自定义的处理

信号从产生到被处理所经过的过程：

产生(generate)-> 挂起(pending)-> 派送(deliver)-> 部署(disposition)或忽略(igore)

一个信号集合是一个 C 语言的 sigset_t 数据类型的对象，sigset_t 数据类型定义在<signal.h>中。被一个进程忽略的所有信号的集合称为一个信号掩码(mask)。

从程序中向一个进程发送信号有两种方法：调用 shell 的 kill 命令，调用 kill 系统调用函数。kill 能够发送除杀死一个进程(SIGKILL、SIGTERM、SIGQUIT)

之外的其他信号，例如键盘中断(Ctrl+C)信号 SIGINT,进程暂停(Ctrl+Z)信号 SIGTSTP 等等。

调用 Pause 函数会令调用进程的执行挂起直到一个任意信号到来后再继续运行。

调用 sleep 函数会令调用进程的执行挂起睡眠指定的秒数或一个它可以响应的信号到来后继续执行。

每个进程都能使用 signal 函数定义自己的信号处理函数，捕捉并自行处理接收的除 SIGSTOP 和 SIGKILL 之外的信号。以下是有关的系统调用的语法说明。

kill 系统调用语法：

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

int kill(pid_t pid, int sig);
pid 接收信号的进程号
signal 要发送的信号
kill 发送成功返回接收者的进程号，失败返回-1。

pause 系统调用语法：

#include <unistd.h>
int pause(void);
pause 挂起调用它的进程直到有任何信号到达。调用进程不自定义处理方法，则进行信号的默认处理。只有进程自定义了信号处理方法捕获并处理了一个信号后，pause 才会返回调进程。pause 总是返回-1,并设置系统变量 errno 为 EINTR。

sleep 系统调用语法：

#include <unistd.h>
unsigned int sleep(unsigned int seconds);
seconds 指定进程睡眠的秒数
如果指定的秒数到，sleep 返回 0。

signal 系统调用语法为:

#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
signum 要捕捉的信号
handler 进程中自定义的信号处理函数名
signal 调用成功会返回信号处理函数的返回值，不成功返回-1,并设置系统变量 errno 为 SIG_ERR。

1.3 示例实验

以下实验示例程序应实现一个类似子 shell 子命令的功能,它可以从执行程序中启动另一个新的子进程并执行一个新的命令和其并发执行.

1) **打开一终端命令行窗体，新建一个文件夹，在该文件夹中建立以下名为pctl.c的C语言程序：**

```
/*
   * Filename                  : pctl.c
   * copyright                 : (C) 2006 by 张鸿烈
   * Function                  : 父子进程的并发执行
*/
#include "pctl.h"
int main(int argc, char *argv[])
{
   int i;
   int pid;       //存放子进程号
   int status;    //存放子进程返回状态
```

```
  char *args[] = {"/bin/ls","-a",NULL};   //子进程要缺省执行的命令
  signal(SIGINT,(sighandler_t)sigcat); //注册一个本进程处理键盘中断的函数
  pid=fork() ; //建立子进程
  if(pid<0)   //  建立子进程失败?
  {
     printf("Create Process fail!\n");
     exit(EXIT_FAILURE);
  }
  if(pid == 0) //  子进程执行代码段
  {
     //报告父子进程进程号
       printf("I am Child process %d\nMy father is %d\n",getpid(),getppid());
       pause();     //暂停，等待键盘中断信号唤醒
       //子进程被键盘中断信号唤醒继续执行
       printf("%d child will Running: \n",getpid()); //
       if(argv[1] != NULL){
          //如果在命令行上输入了子进程要执行的命令
          //则执行输入的命令
              for(i=1; argv[i] != NULL; i++)    printf("%s ",argv[i]);
                printf("\n");
              //装入并执行新的程序
            status = execve(argv[1],&argv[1],NULL);
         }
  else{
          //如果在命令行上没输入子进程要执行的命令
          //则执行缺省的命令
            for(i=0; args[i] != NULL; i++)    printf("%s ",args[i]);
            printf("\n");
              //装入并执行新的程序
            status = execve(args[0],args,NULL);
         }
  }
 else //父进程执行代码段
 {
    printf("\nI am Parent process    %d\n",getpid()); //报告父进程进程号
    if(argv[1] != NULL){
     //如果在命令行上输入了子进程要执行的命令
     //则父进程等待子进程执行结束
         printf("%d    Waiting for child done.\n\n", pid);
           waitpid(pid,&status,0);     //等待子进程结束
         printf("\nMy child exit! status = %d\n\n",status);
     }
  else{
       sleep(1) ; //等待子进程建立
     //如果在命令行上没输入子进程要执行的命令
       //唤醒子进程，与子进程并发执行不等待子进程执行结束，
```

```
                if(kill(pid,SIGINT) >= 0)
                    printf("%d Wakeup %d child.\n",getpid(),pid) ;
            printf("%d don't Wait for child done.\n\n",getpid());
        }
    }
    return EXIT_SUCCESS;
}
```

2) 再建立以下名为 pctl.h 的 C 语言头文件：

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
//进程自定义的键盘中断信号处理函数
typedef void (*sighandler_t) (int);
void sigcat(){
    printf("%d Process continue\n",getpid());
}
```

3) **建立以下项目管理文件 Makefile**

```
head = pctl.h
srcs = pctl.c
objs = pctl.o
opts = -g -c
all:   pctl
pctl:       $(objs)
            gcc $(objs) -o    pctl
pctl.o:     $(srcs)    $(head)
            gcc $(opts) $(srcs)
clean:
            rm pctl    *.o
```

4) **输入 make 命令编译连接生成可执行的 pctl 程序**

```
$ gmake
gcc -g -c pctl.c
gcc pctl.o -o    pctl
```

5) **执行 pctl 程序(注意进程号是动态产生的,每次执行都不相同)**

```
$ ./pctl
I am Child process 4113
My father is 4112

I am Parent process    4112
Wakeup 4113 child.
4112 don't Wait for child done.
```

4113 Process continue
4113 child will Running: /bin/ls -a
. .. Makefile pctl pctl.c pctl.h pctl.o
$
以上程序的输出说明父进程 4112 创建了一个子进程 4113，子进程执行被暂停。父进程向子进程发出键盘中断信号唤醒子进程并与子进程并发执行。父进程并没有等待子进程的结束继续执行先行结束了（此时的子进程成为了孤儿进程，不会有父进程为它清理退出状态了）。而子进程继续执行，它变成了列出当前目录所有文件名的命令 ls -a。在完成了列出文件名命令之后，子进程的执行也结束了。此时子进程的退出状态将有初始化进程为它清理。

6) 再次执行带有子进程指定执行命令的 pctl 程序:

$ ./pctl /bin/ls -l
I am Child process 4223
My father is 4222

I am Parent process 4222
4222 Waiting for child done.
可以看到这一次子进程仍然被挂起，而父进程则在等待子进程的完成。为了检测父子进程是否都在并发执行，请输入 ctrl+z 将当前进程放入后台并输入 ps 命令查看当前系统进程信息，显示如下：
[1]+ Stopped ./pctl /bin/ls -l
$ *ps -l*

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	0	4085	4083	0	76	0	-	1413	wait	pts/1	00:00:00	bash
0	T	0	4222	4085	0	76	0	-	360	finish	pts/1	00:00:00	pctl
1	T	0	4223	4222	0	76	0	-	360	finish	pts/1	00:00:00	pctl
0	R	0	4231	4085	0	78	0	-	1302	-	pts/1	00:00:00	ps

可以看到当前系统中同时有两个叫 pctl 的进程，它们的进程号分别是 4222 和 4223。它们的状态都为“T”,说明当前都被挂起。4223 的父进程是 4222，而 4222 的父进程是 4085，也就是 bash-shell。为了让 pctl 父子进程继续执行，请输入 fg 命令让 pctl 再次返回前台，显示如下：
$ fg
./pctl /bin/ls -l
现在 pctl 父子进程从新返回前台。我们可以通过键盘发键盘中断信号来唤醒 pctl 父子进程继续执行，输入 ctrl+c，将会显示：
4222 Process continue
4223 Process continue
4223 child will Running: /bin/ls -l
total 1708
-rw-r--r-- 1 root root 176 May 8 11:11 Makefile
-rwxr-xr-x 1 root root 8095 May 8 14:08 pctl
-rw-r--r-- 1 root root 2171 May 8 14:08 pctl.c
-rw-r--r-- 1 root root 269 May 8 11:10 pctl.h
-rw-r--r-- 1 root root 4156 May 8 14:08 pctl.o

My child exit! status = 0
以上输出说明了子进程在捕捉到键盘中断信号后继续执行了指定的命令，按我们要求的长格式列出了当前目录中的文件名，父进程在接收到子进程执行结束的信号后将清理子进程的退出状态并继续执行，它报告了子进程的退出编码（0 表示子进程正常结束)最后父进程也结束执行。

1.4 独立实验

参考以上示例程序中建立并发进程的方法，编写一个多进程并发执行程序。父进程首先创建一个执行 ls 命令的子进程然后再创建一个执行 ps 命令的子进程，并控制 ps 命令总在 ls 命令之前执行。

1.5. 实验要求

根据实验中观察和记录的信息结合示例实验和独立实验程序，说明它们反映出操作系统教材中进程及处理机管理一节讲解的进程的哪些特征和功能？在真实的操作系统中它是怎样实现和反映出教材中讲解的进程的生命期、进程的实体和进程状态控制的。你对于进程概念和并发概念有哪些新的理解和认识？子进程是如何创建和执行新程序的？信号的机理是什么？怎样利用信号实现进程控制？根据实验程序、调试过程和结果分析写出实验报告。

实验二、线程和进程/线程管道通信实验

2.1 实验目的

通过 Linux 系统中线程和管道通信机制的实验,加深对于线程控制和管道通信概念的理解,观察和体验并发进/线程间的通信和协作的效果 ,练习利用无名管道进行进/线程间通信的编程和调试技术。

2.2 实验说明

2.2.1 与线程创建、执行有关的系统调用说明

线程是在共享内存中并发执行的多道执行路径,它们共享一个进程的资源,如进程程序段、文件描述符和信号等,但有各自的执行路径和堆栈。线程的创建无需像进程那样重新申请系统资源,线程在上下文切换时也无需像进程那样更换内存映像。多线程的并发执行即避免了多进程并发的上下文切换的开销又可以提高并发处理的效率。

Linux 利用了特有的内核函数__clone 实现了一个叫 phread 的线程库,__clone 是 fork 函数的替代函数,通过更多的控制父子进程共享哪些资源而实现了线程。Pthread 是一个标准化模型,用它可把一个程序分成一组能够并发执行的多个任务。phread 线程库是 POSIX 线程标准的实现,它提供了 C 函数的线程调用接口和数据结构。

线程可能的应用场合包括:

- 在返回前阻塞的 I/O 任务能够使用一个线程处理 I/O,同时继续执行其他处理。需要及时响应多个前台用户界面操作同时后台处理的多任务场合。
- 在一个或多个任务受不确定事件影响时能够处理异步事件同时继续进行正常处理。
- 如果某些程序功能比其他功能更重要,可以使用线程以保证所有功能都出现,但那些时间密集型的功能具有更高优先级。

下面介绍 pthread 库中最基本的调用。

pthread_create 系统调用语法:
#include <pthread.h>
Int pthread_create(pthread_t *thread,pthread_attr_t *attr,void *(*start_routine)(void *)
Void *arg);
pthread_create 函数创建一个新的线程。pthread_create 在 thread 中保存新线程的标识符。
Attr 决定了线程应用那种线程属性。使用默认可给定参数 NULL;
(*start_routine) 是一个指向新线程中要执行的函数的指针
arg 是新线程函数携带的参数。
Pthread_create 执行成功会返回 0 并在 thread 中保存线程标识符。执行失败则返回一个非 0 的出错代码。

pthread_exit 系统调用语法:
#include <pthread.h>
void pthread_exit(void *retval);
pthread_exit 函数使用函数 pthread_cleanup_push 调用任何用于该线程的清除处理函数,然后中止当前进程的执行,返回 retval。
Retval 可以由父线程或其他线程通过 pthread_join 来检索。一个线程也可以简单地通过从其初始化函数返回来终止。

pthread_join 系统调用语法:
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
int pthread_detach(pthread_t th);
函数 pthread_join 用于挂起当前线程,直到 th 指定的线程终止运行为止。另一个线程的返回值如果不为 NULL,则保存在 thread_return 指向的地址中。一个线程所使用的内存资源在对该线程应用 pthread_join 调用之前不会被重新分配。因而对于每个可切入的线程(默认的)必须调用一次 pthread_join 函数。线程必须是可切入的而不是被分离的状态,并且其他线程不能对同一线程再应用 pthread_join 调用。通过在 pthread_create 调用创建一个线程时使用 PTHREAD_CREATE_DETACHED 属性或者使用 pthread_detach 可以让线程处于被分离状态。
注意不像由 fork 创建的进程可以使用众多 wait 等待子进程退出,在 pthread 多线程中似乎没有等待某个线程退出的方法。

2.2.2 管道通信机制

管道 pipe 是进程间通信最基本的一种机制。在内存中建立的管道称为无名管道,在磁盘上建立的管道称为有名管道。无名管道随着进程的撤消而消失,有名管道则可以长久保存,shell 命令符| 建立的就是无名管道,而 shell 命令 mkfifo 建立的是有名管道。两个进程可以通过管道一个在管道一端向管道发送其输出,给另一进程可以在管道的另一端从管道得到其输入.管道以半双工方式工作,即它的数据流是单方向的.因此使用一个管道一般的规则是读管道数据的进程关闭管道写入端,而写管道进程关闭其读出端。管道既可以采用同步方式工作也可以采用异步方式工作。

pipe 系统调用的语法为:
#include <unistd.h>
int pipe(int pipe_id[2]);
pipe 建立一个无名管道,pipe_id[0]中和 pipe_id[1]将放入管道两端的描述符如果 pipe 执行成功返回 0。.出错返回-1.

管道读写的系统调用语法为:
#include <unistd.h>
ssize_t read(int pipe_id,const void *buf,size_t count);
ssize_t write(int pipe_id,const void *buf,size_t count);

read 和 write 分别在管道的两端进行读和写。
pipe_id 是 pipe 系统调用返回的管道描述符。
Buf 是数据缓冲区首地址,

count 说明数据缓冲区以 size_t 为单位的长度。
read 和 write 的返回值为它们实际读写的数据单位。

注意管道的读写默认的通信方式为同步读写方式,即如果管道读端无数据则读者阻塞直到数据到达,反之如果管道写端有数据则写者阻塞直到数据被读走。

2.3 示例实验

1) 以下示例实验程序实现并发的两个线程合作将整数 X 的值从 1 加到 10 的功能。它们通过管道相互将计算结果发给对方。
(1) 在新建文件夹中建立以下名为 tpipe.c 的 C 语言程序

```
/*
 *  description           : tpipe.c
 *  copyright             : (C)  by 张鸿烈
 *  Function              : 利用管道实现在在线程间传递整数
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void task1(int *); //线程 1 执行函数原型
void task2(int *); //线程 2 执行函数原型

int pipe1[2],pipe2[2];//存放两个无名管道标号
pthread_t thrd1,thrd2;//存放两个线程标识

int main(int argc,char *arg[])
{
  int ret;
  int num1,num2;

    //使用pipe()系统调用建立两个无名管道。建立不成功程序退出，执行终止
   if(pipe(pipe1) < 0){
      perror("pipe not create");
    exit(EXIT_FAILURE);
    }
    if(pipe(pipe2) < 0){
      perror("pipe not create");
    exit(EXIT_FAILURE);
    }

  //使用pthread_create系统调用建立两个线程。建立不成功程序退出，执行终止
  num1 = 1 ;
```

```
  ret = pthread_create(&thrd1,NULL,(void *) task1,(void *) &num1);
  if(ret){
    perror("pthread_create: task1");
    exit(EXIT_FAILURE);
  }

  num2 = 2 ;
  ret = pthread_create(&thrd2,NULL,(void *) task2,(void *) &num2);
  if(ret){
    perror("pthread_create: task2");
    exit(EXIT_FAILURE);
   }

 //挂起当前线程切换到thrd2 线程
 pthread_join(thrd2,NULL);

 //挂起当前线程切换到thrd1 线程
 pthread_join(thrd1,NULL);

 exit(EXIT_SUCCESS);
}


//线程 1 执行函数，它首先向管道写，然后从管道读
void task1(int *num)
{
    int x=1;

        //每次循环向管道 1 的 1 端写入变量X的值,并从
        //管道 2 的 0 端读一整数写入X再对X加 1，直到X大于 10
        do{
              printf("thread%d read: %d\n",*num,x++);
              write(pipe1[1],&x,sizeof(int));
              read(pipe2[0],&x,sizeof(int));
         }while(x<=9);

    //读写完成后,关闭管道
    close(pipe1[1]);
    close(pipe2[0]);
}

//线程 1 执行函数，它首先从管道读，然后向管道写
void task2(int * num)
```

```
{
    int x;
       //每次循环从管道 1 的 0 端读一个整数放入变量X中,
       //并对X加 1 后写入管道 2 的 1 端，直到X大于 10
       do{
           read(pipe1[0],&x,sizeof(int));
           printf("thread2 read: %d\n",x++);
         write(pipe2[1],&x,sizeof(int));
       }while( x<=9 );

      //读写完成后,关闭管道
       close(pipe1[0]);
       close(pipe2[1]);
}
```

(2) 再建立程序的 Makefile 文件:

```
src = tpipe.c
obj = tpipe.o
opt = -g -c
all:        tpipe:
tpipe:     $(obj)
           gcc   $(obj)   -l   pthread   –o   tpipe
tpipe.o:   $(src)
           gcc   $(opt)   $(src)
clean:
           rm    tpipe *.o
```

(3) 使用 make 命令编译连接生成可执行文件 tpipe.c :

```
$ gmake
gcc -g -c tpipe.c
gcc tpipe.o -l pthread -o tpipe
```

(4) 编译成功后执行 tpipe:命令:

```
$ ./tpipe
thread1 read: 1
thread2 read: 2
thread1 read: 3
thread2 read: 4
thread1 read: 5
thread2 read: 6
thread1 read: 7
thread2 read: 8
thread1 read: 9
thread2 read: 10
```

可以看到以上程序的执行中线程 1 和线程 2 交替的将整数 X 的值从 1 加了 10。

2) 以下示例实验程序实现并发的父子进程合作将整数 X 的值从 1 加到 10 的功能。

(1) 在新建文件夹中建立以下名为 ppipe.c 的 C 语言程

```
/*
   * Filename                  : ppipe.c
   * copyright                 : (C) 2006 by 张鸿烈
   * Function                  : 利用管道实现在父子进程间的消息通信
  */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   int pid;                //进程号
   int pipe1[2];     //存放第一个无名管道标号
   int pipe2[2];    //存放第二个无名管道标号
   int x;              // 存放要传递的整数
      //使用 pipe()系统调用建立两个无名管道。建立不成功程序退出，执行终止
    if(pipe(pipe1) < 0){
        perror("pipe not create");
     exit(EXIT_FAILURE);
     }
     if(pipe(pipe2) < 0){
        perror("pipe not create");
     exit(EXIT_FAILURE);
     }
     //使用 fork()系统调用建立子进程,建立不成功程序退出，执行终止
    if((pid=fork()) <0){
          perror("process not create");
       exit(EXIT_FAILURE);
     }
    //子进程号等于 0 表示子进程在执行,
    else if(pid == 0){
     //子进程负责从管道 1 的 0 端读,管道 2 的 1 端写，
     //所以关掉管道 1 的 1 端和管道 2 的 0 端。
        close(pipe1[1]);
        close(pipe2[0]);
         //每次循环从管道 1 的 0 端读一个整数放入变量 X 中,
         //并对 X 加 1 后写入管道 2 的 1 端，直到 X 大于 10
         do{
              read(pipe1[0],&x,sizeof(int));
              printf("child %d read: %d\n",getpid(),x++);
```

```
            write(pipe2[1],&x,sizeof(int));
        }while( x<=9 );
       //读写完成后,关闭管道
        close(pipe1[0]);
        close(pipe2[1]);
        //子进程执行结束
         exit(EXIT_SUCCESS);
      }
    //子进程号大于 0 表示父进程在执行,
    else{
    //父进程负责从管道 2 的 0 端读,管道 1 的 1 端写,
    //所以关掉管道 1 的 0 端和管道 2 的 1 端。
         close(pipe1[0]);
         close(pipe2[1]);
         x=1;
        //每次循环向管道 1 的 1 端写入变量 X 的值,并从
        //管道 2 的 0 端读一整数写入 X 再对 X 加 1，直到 X 大于 10
         do{
              write(pipe1[1],&x,sizeof(int));
              read(pipe2[0],&x,sizeof(int));
              printf("parent %d read: %d\n",getpid(),x++);
         }while(x<=9);
    //读写完成后,关闭管道
    close(pipe1[1]);
    close(pipe2[0]);
     }
   //父进程执行结束
    return EXIT_SUCCESS;
}
```

2) **在当前目录中建立以下Makefile文件：**

```
srcs =    ppipe.c
objs =    ppipe.o
opts =    -g -c
all:        ppipe
ppipe:    $(objs)
          gcc $(objs)    -o ppipe
ppipe.o:  $(srcs)
          gcc    $(opts)    $(srcs)
clean:
          rm ppipe *.o
```

3) **使用make命令编译连接生成可执行文件ppipe:**

```
$ gmake
gcc    -g -c    ppipe.c
 gcc ppipe.o    -o ppipe
```

4) 编译成功后执行ppipe:命令:

```
$ ./ppipe
    child 8697 read: 1
    parent 8696 read: 2
    child 8697 read: 3
    parent 8696 read: 4
    child 8697 read: 5
    parent 8696 read: 6
    child 8697 read: 7
    parent 8696 read: 8
    child 8697 read: 9
    parent 8696 read: 10
```

可以看到以上程序的执行中父子进程合作将整数 X 的值从 1 加到了 10。

2.4 独立实验

设有二元函数 f(x,y) = f(x) + f(y)

其中：
f(x) = f(x-1) * x (x >1)
f(x)=1 (x=1)

f(y) = f(y-1) + f(y-2) (y> 2)
f(y)=1 (y=1,2)

请编程建立 3 个并发协作进程（或线程），它们分别完成 f(x,y)、f(x)、f(y)
其中由父进程（或主线程）完成：f(x,y) = f(x) + f(y)

由子进程 1（或线程 1）完成：f(x) = f(x-1) * x (x >1)
f(x)=1 (x=1)

由子进程 2（或线程 2）完成：f(y) = f(y-1) + f(y-2) (y> 2)
f(y)=1 (y=1,2)

2.5 实验要求

根据示例实验程序和独立实验程序观察和记录的调试和运行的信息，说明它们反映出操作系统教材中讲解的进程协作和进程通信概念的哪些特征和功能？在真实的操作系统中它是怎样实现和反映出教材中进程通信概念的。你对于进程协作和进程通信的概念和实现有哪些新的理解和认识？管道机制的机理是什么？怎样利用管道完成进程间的协作和通信？根据实验程序、调试过程和结果分析写出实验报告。

实验三、进程调度算法实验

3.1 实验目的

加深对进程调度概念的理解，体验进程调度机制的功能，了解 Linux 系统中进程调度策略的使用方法。 练习进程调度算法的编程和调试技术。

3.2 实验说明

在 linux 系统中调度策略（policy）可以是以下 3 种：

- SCHED_OTHER 默认的分时调度策略(值等于 0)
- SCHED_FIFO 先进先先出调度策略(值等于 1)
- SCHED_RR 时间片轮转调度策略(值等于 2)

后两种专用于对响应时间有特殊要求的进程，并且会抢先于 SCHED_OTHER 调度策略的进程而执行。一个具有 SCHED_FIFO 调度策略的进程只能被更高优先级的进程抢先，但具有 SCHED_RR 调度策略的进程必要时可以与同级进程共享时间片。

进程优先数(prio)由静态优先数和动态优先数两部分组成，值越小调度优先级越高 。具有 SCHED_OTHER 策略的进程静态优先数总是 0。动态优先数与进程的执行状态有关，但可以使用 nice 命令或系统调用加大进程优先数使其优先级降低，或用系统调用 setpriority 分别按进程或进程组或用户号设置介于-20 到+20 之间的动态优先数。

与进程调度策略有关的系统调用函数原型都声明在以下文件中：

```
#include <sched.h>
#include <sys/time.h>
#include <sys/resource.h>
```

设置进程调度策略的系统调用语法为：

```
int sched_setscheduler(pid_t pid,int policy,const struct sched_param *sp);
    pid         进程号
    policy      以上说明的 3 种调度策略之一
    sp          调度参数结构指针,调度参数结构主要存有调度优先数
                struct sched_param {
                  ...
                        int sched_priority;
                  ...
                };
    返回值: 执行成功后返回 0
```

获取进程调度策略的系统调用语法为:

```
int sched_getscheduler(pid_t pid);
    pid         进程号
  返回值: 进程当前的调度策略
```

获取进程动态优先数的系统调用语法为：

```
int getpriority(int which,int who);
```

which 设置的对象。可以是:
进程 PRIO_PROCESS
进程组 PRIO_PGRP
用户 PRIO_USER
who 对应设置对象的进程号或组号或用户号
返回值: 所有匹配进程中的最高优先数

设置进程动态优先数的系统调用语法为:
int setpriority(int which,int who,int prio);
which 设置的对象。可以是:
进程 PRIO_PROCESS
进程组 PRIO_PGRP
用户 PRIO_USER
who 对应设置对象的进程号或组号或用户号
prio 要设置的进程优先数
返回值: 所有匹配进程中的最高优先数

3.3 示例实验

以下示例实验程序要测试在 linux 系统中不同调度策略和不同优先数的调度效果。

1) **在新建文件夹中建立以下名为 psched.c 的 C 语言程序:**

```
/*
    *   Filename        : psched.c
    *   copyright       : (C) 2006 by 张鸿烈
    *   Function        : 父进程创建 3 个子进程为它们设置不同的优先数和调度策略
*/
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <sys/time.h>
#include <sys/resource.h>

int main(int argc, char *argv[])
{
  int i,j,status;
  int pid[3]; //存放进程号
  struct sched_param p[3]; //设置调度策略时使用的数据结构

   for(i=0; i<3;i++){
      //循环创建 3 个子进程
     if((pid[i]=fork()) >0){;
        //取进程优先数放在调度策略数据结构中
       p[i].sched_priority = (argv[i+1] != NULL) ? atoi(argv[i+1]):10;
        //父进程设置子进程的调度策略.如果命令行第 4,5,6 参数指定了 3 个策略
```

```
值则按指定的数设置,否则都为默认策略
          sched_setscheduler(pid[i],(argv[i+4] != NULL) ? atoi(argv[i+4]) :
SCHED_OTHER,&p[i]);
            //父进程设置子进程的优先数,如果命令行第 1,2,3 参数指定了 3 个优先数
则按指定的数设置,否则都为 10
          setpriority(PRIO_PROCESS,pid[i],(argv[i+1] != NULL) ? atoi(argv[i+1]):10);
         }
       //各子进程循环报告其优先数和调度策略
      else{
        sleep(1);
         //每隔 1 妙报告一次进程号和优先级
         for(i=0; i<10; i++){
            printf("Child PID = %d priority
= %d\n",getpid(),getpriority(PRIO_PROCESS,0));
           sleep(1);
         }
          exit( EXIT_SUCCESS);
       }
    }
    //父进程报告子进程调度策略后先行退出
       printf("My child %d policy is %d\n",pid[0],sched_getscheduler(pid[0]));
       printf("My child %d policy is %d\n",pid[1],sched_getscheduler(pid[1]));
       printf("My child %d policy is %d\n",pid[2],sched_getscheduler(pid[2]));
       return EXIT_SUCCESS;
}
```

2) **在当前目录中建立以下 Makefile 文件：**

```
srcs = psched.c
objs = psched.o
 opts = -g -c
 all:            psched
 psched:         $(objs)
                 gcc $(objs) -o psched
 psched.o:       $(srcs)
                 gcc    $(opts)    $(srcs)
 clean:
                 rm psched *.o
```

3) **使用make命令编译连接生成可执行文件psched:**

```
    $gmake
    gcc -c psched.c
    gcc psched.o -o psched
```

4) 改变到 root 用户

```
$ su    （或 sudo ./psched 10   5 -10   0 0 0 ）
口令：(输入 root 口令)
#
```

5) **运行 psched，指定 3 个子进程的优先数为 10, 5, -10；调度策略都是默认策略**
#./psched 10 5 -10 0 0 0

```
child 10771 policy is 0
child 10772 policy is 0
child 10773 policy is 0
Child PID = 10773 priority = -10
Child PID = 10772 priority = 5
Child PID = 10771 priority = 10
......
Child PID = 10773 priority = -10
Child PID = 10772 priority = 5
Child PID = 10771 priority = 10
```

可以看出在相同的调度策略下优先数小的进程先得到了执行。

6) **再次运行 psched，指定 3 个子进程的优先数为 10, 5, 18；调度策略分别是 0,0,1**
./psched 10 5 18 0 0 1

```
My child 11306 policy is 0
My child 11307 policy is 0
My child 11308 policy is 1
Child PID = 11308 priority = 18
Child PID = 11307 priority = 5
Child PID = 11306 priority = 10
Child PID = 11308 priority = 18
Child PID = 11307 priority = 5
Child PID = 11306 priority = 10
```

可以看出虽然 11308 进程其优先数最大，但由于其调度策略为先进先出，因此总是首先得到调度。

7) *# exit*
$

3.4 独立实验

设有两个并发执行的父子进程，不断循环输出各自进程号、优先数和调度策略。进程初始调度策略均为系统默认策略和默认优先级。当父进程收到由键盘 Ctrl+C 发出的 SIGINT 信号时会自动将其优先数加 1，子进程收到由键盘 Ctrl+Z 发出的 SIGTSTP 信号时会自动将其优先数减 1。请编程实现以上功能。

3.5 实验要求

根据以上示例程序和独立实验程序中观察和记录的信息，说明它们反映出操作系统教材中讲解的哪些进程调度调度策略和功能？在真实的操作系统中它是怎样实现教材中讲解的进程调度效果的。你对于进程调度的概念哪些新的理解和认识？根据实验程序、调试过程和结果分析写出实验报告。

实验四、进程同步实验

4.1 实验目的

加深对并发协作进程同步与互斥概念的理解，观察和体验并发进程同步与互斥操作的效果，分析与研究经典进程同步与互斥问题的实际解决方案。了解 Linux 系统中 IPC 进程同步工具的用法，练习并发协作进程的同步与互斥操作的编程与调试技术。

4.2 实验说明

在 linux 系统中可以利用进程间通信（interprocess communication ）IPC 中的 3 个对象：共享内存、信号灯数组、消息队列，来解决协作并发进程间的同步与互斥的问题。

1）共享内存是 OS 内核为并发进程间交换数据而提供的一块内存区（段）。如果段的权限设置恰当，每个要访问该段内存的进程都可以把它映射到自己私有的地址空间中。如果一进程更新了段中数据，那么其他进程立即会看到这一更新。进程创建的段也可由另一进程读写。

linux 中可用命令 ipcs -m 观察共享内存情况。

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 327682     student    600        393216     2          dest
0x00000000 360451     student    600        196608     2          dest
0x00000000 393220     student    600        196608     2          dest
```

key 共享内存关键值
shmid 共享内存标识
owner 共享内存所由者（本例为 student）
perm 共享内存使用权限(本例为 student 可读可写）
byte 共享内存字节数
nattch 共享内存使用计数
status 共享内存状态
上例说明系统当前已由 student 建立了一些共享内存,每个都有两个进程在共享。

2)信号灯数组是OS内核控制并发进程间共享资源的一种进程同步与互斥机制。linux 中可用命令 ipcs -s 观察信号灯数组的情况。

```
$ ipcs -s
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0000000 163844     apache     600        1
0x4d00f259 294920     beagleind 600        8
0x00000159 425995     student    644        1
```

semid　信号灯的标识号
nsems　信号灯的个数
其他字段意义同以上共享内存所述。
上例说明当前系统中已经建立多个信号灯。其中最后一个标号为 425996 是由 student 建立的,它的使用权限为 644，信号灯数组中信号灯个数为 1 个。

3)消息队列是 OS 内核控制并发进程间共享资源的另一种进程同步机制。linux 中可用命令 ipcs -q 观察消息队列的情况。

```
$ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x000001c8 0          root       644        8            1
```

msgmid　消息队列的标识号
used-bytes　消息的字节长度
messages　消息队列中的消息条数
其他字段意义与以上两种机制所述相同。
上例说明当前系统中有一条建立消息队列，标号为 0，为 root 所建立，使用权限为 644，每条消息 8 个字节，现有一条消息。

4) 在权限允许的情况下您可以使用 ipcrm 命令删除系统当前存在的 IPC 对象中的任一个对象。

ipcrm -m 21482　　删除标号为 21482 的共享内存。
ipcrm -s 32673　　删除标号为 32673 的信号灯数组。
ipcrm -q 18465　　删除标号为 18465 的消息队列。

5) **在 linux 的 proc 文件系统中有 3 个虚拟文件动态记录了由以上 ipcs 命令显示的当前 IPC 对象的信息，它们分别是：**

/proc/sysvipc/shm　共享内存
/proc/sysvipc/sem　信号量
/proc/sysvipc/msg　消息队列
我们可以利用它们在程序执行时获取有关 IPC 对象的当前信息。

6) IPC 对象有关的系统调用函数原型都声明在以下的头文件中 ：

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

创建一段共享内存系统调用语法 ：

```
#include <sys/shm.h>
int shmget(key_t key,int size,int flags);
```

key 共享内存的键值,可以为 IPC_PRIVATE,也可以用整数指定一个
size 共享内存字节长度
flags 共享内存权限位。

shmget 调用成功后，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的共享内存段标识符。 如果指定的 key 已存在则返回与 key 关联的标识符。 不成功返回-1

令一段共享内存附加到调用进程中的系统调用语法：

#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr,int flags)
shmid 由 shmget 创建的共享内存的标识符
shmaddr 总为 0,表示用调用者指定的指针指向共享段
flags 共享内存权限位
shmat 调用成功后返回附加的共享内存首地址

令一段共享内存从到调用进程中分离出去的系统调用语法：

#include <sys/shm.h>
int shmdt(char *shmadr);
shmadr 进程中指向附加共享内存的指针
shmdt 调用成功将递减附加计数，当计数为 0，将删除共享内存。调用不成功返回-1。

创建一个信号灯数组的系统调用有语法:

#include <sys/sem.h>
int semget(key_t key,int nsems, int flags);
key 信号灯数组的键值,可以为 IPC_PRIVATE,也可以用整数指定一个
nsems 信号灯数组中信号灯的个数
flags 信号灯数组权限位。如果 key 用整数指定，应设置 IPC_CREAT 位。
semget 调用成功，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的信号等数组标识符。 如果指定的整数 key 已存在则返回与 key 关联的标识符。 不成功返回-1

操作信号灯数组的系统调用语法:

#include <sys/sem.h>
int semop(int semid,struct sembuf *semop, unsigned nops);
semid 由 semget 创建的信号灯数组的标识符
semop 指向 sembuf 数据结构的指针
nops 信号灯数组元素的个数。
semop 调用成功返回 0，不成功返回-1。

控制信号灯数组的系统调用语法:

#include <sys/sem.h>
int semctl(int semid,int semnum,int cmd, union semun arg);
semid 由 semget 创建的信号灯数组的标识符
semnum 该信号灯数组中的第几个信号灯
cmd 对信号灯发出的控制命令。例如:
GETVAL 返回当前信号灯状态
SETVAL 设置信号灯状态
IPC_RMID 删除标号为 semid 的信号灯
arg 保存信号灯状态的联合体,信号灯的值是其中一个基本成员
union semun {

```
    int val;          /* value for SETVAL */
      ......
    };
```

semctl 执行不成功返回-1，否则返回指定的 cmd 的值。

创建消息队列的系统调用语法:

```
#include<sys/msg.h>
int msgget(key_t key,int flags)
```

key 消息队列的键值,可以为 IPC_PRIVATE,也可以用整数指定一个

flags 消息队列权限位。

msgget 调用成功，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的消息队列标识符。 如果指定的整数 key 已存在则返回与 key 关联的标识符。成功返回-1。

追加一条新消息到消息队列的系统调用语法：

```
#include <sys.msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

msqid 由消息队列的标识符

msgp 消息缓冲区指针。消息缓冲区结构为：

```
struct msgbuf {
    long mtype;       /* 消息类型，必须大于 0  */
    char mtext[1];    /* 消息数据，长度应于 msgsz 声明的一致*/
  }
```

msgsz 消息数据的长度

msgflg 为 0 表示阻塞方式，设置 IPC_NOWAIT 表示非阻塞方式

msgsnd 调用成功返回 0，不成功返回-1。

从到消息队列中读出一条新消息的系统调用语法：

```
#include <sys.msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,   long msgtype, int msgflg);
```

msqid 由消息队列的标识符

msgp 消息缓冲区指针。消息缓冲区结构为：

```
struct msgbuf {
    long mtype;       /* 消息类型，必须大于 0  */
    char mtext[1];    /* 消息数据，长度应于 msgsz 声明的一致*/
}
```

msgsz 消息数据的长度

msgtype 决定从队列中返回哪条消息：

=0 返回消息队列中第一条消息

>0 返回消息队列中等于 mtype 类型的第一条消息。

<0 返回 mtype<=type 绝对值最小值的第一条消息。

msgflg 为 0 表示阻塞方式，设置 IPC_NOWAIT 表示非阻塞方式

msgrcv 调用成功返回 0，不成功返回-1。

删除消息队列的系统调用语法：

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

msqid由消息队列的标识符

cmd　　控制命令。常用的有：

IPC_RMID　删除 msgid 标识的消息队列

IPC_STAT　为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲 buf

IPC_SET　改变队列的 UID，GID，访问模式和最大字节数。

msgctl 调用成功返回 0，不成功返回-1。

4.3 示例实验

以下示例实验程序应能模拟多个生产/消费者在有界缓冲上正确的操作。它利用 N 个字节的共享内存作为有界循环缓冲区，利用写一字符模拟放一个产品，利用读一字符模拟消费一个产品。当缓冲区空时消费者应阻塞睡眠，而当缓冲区满时生产者应当阻塞睡眠。一旦缓冲区中有空单元，生产者进程就向空单元中入写字符,并报告写的内容和位置。一旦缓冲区中有未读过的字符，消费者进程就从该单元中读出字符，并报告读取位置。生产者不能向同一单元中连续写两次以上相同的字符，消费者也不能从同一单元中连续读两次以上相同的字符。

1) 在当前新建文件夹中建立以下名为 ipc..h 的 C 程序的头文件，该文件中定义了生产者/消费者共用的 IPC 函数的原型和变量：

```
/*
 *  Filename          :  ipc.h
 *  copyright         : (C) 张鸿烈
 * Function           : 声明IPC机制的函数原型和全局变量
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <signal.h>

#define BUFSZ   256

//进程自定义的键盘中断信号处理函数原型
typedef void (*sighandler_t) (int);
void sigcat(void);
```

```
//建立或获取ipc的一组函数的原型说明

int get_ipc_id(char *proc_file,key_t key);

char *set_shm(key_t shm_key,int shm_num,int shm_flag);

int set_msq(key_t msq_key,int msq_flag);

int set_sem(key_t sem_key,int sem_val,int sem_flag);

int P(int sem_id);
int V(int sem_id);

void ipc_init(void);

/*信号灯控制用的共同体*/
typedef union semuns {
    int val;
} Sem_uns;

/* 消息结构体*/
typedef struct msgbuf {
    long mtype;
    char mtext[1];
} Msg_buf;

//生产消费者共享缓冲区即其有关的变量
 key_t buff_key;
 int buff_num;
 char *buff_ptr;

//生产者放产品位置的共享指针
 key_t pput_key;
 int pput_num;
 int *pput_ptr;

//消费者取产品位置的共享指针
 key_t cget_key;
 int cget_num;
 int *cget_ptr;

//生产者有关的信号量
 key_t prod_key;
```

```
 key_t pmtx_key;
 int prod_sem;
 int pmtx_sem;

//消费者有关的信号量
 key_t cons_key;
 key_t cmtx_key;
 int cons_sem;
 int cmtx_sem;

 int sem_val;
 int sem_flg;
 int shm_flg;
 int rate;
```

2) 在当前新建文件夹中建立以下名为 ipc.c 的 C 程序，该程序中定义了生产者/消费者共用的 IPC 函数：

```
/*
  *  Filename          :  ipc.c
  *  copyright         : (C)张鸿烈
   * Function          : 一组建立IPC机制的函数
*/
#include "ipc.h"
/*
   * get_ipc_id() 从/proc/sysvipc/文件系统中获取IPC的id号
   * pfile: 对应/proc/sysvipc/目录中的IPC文件分别为
   *        msg-消息队列,sem-信号量,shm-共享内存
   * key:   对应要获取的IPC的id号的键值
*/
int get_ipc_id(char *proc_file,key_t key)
{
   FILE *pf;
   int i,j;
   char line[BUFSZ],colum[BUFSZ];

   if((pf = fopen(proc_file,"r")) == NULL){
    perror("Proc file not open");
      exit(EXIT_FAILURE);
    }
   fgets(line, BUFSZ,pf);
   while(!feof(pf)){
     i = j = 0;
```

```
      fgets(line, BUFSZ,pf);
      while(line[i] == ' ') i++;
      while(line[i] !=' ') colum[j++] = line[i++];
      colum[j] = '\0';
      if(atoi(colum) != key) continue;
      j=0;
      while(line[i] == ' ') i++;
      while(line[i] !=' ') colum[j++] = line[i++];
      colum[j] = '\0';
      i = atoi(colum);
      fclose(pf);
      return i;
     }
     fclose(pf);
     return -1;
}
/*
 * 信号灯上的PV操作
 * semid:信号灯数组标识符
 * semnum:信号灯数组下标
 * buf:操作信号灯的结构
*/
int P(int sem_id) // P操作
{
  struct sembuf buf;
  buf.sem_op = -1;
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  if((semop(sem_id,&buf,1)) <0) {
    perror("down error ");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}

int V(int sem_id) // V操作
{
  struct sembuf buf;
  buf.sem_op = 1;
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  if((semop(sem_id,&buf,1)) <0) {
    perror("up error ");
```

```
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}

/*
*   set_sem函数建立一个具有n个信号灯的信号量
*         如果建立成功，返回 一个信号灯数组的标识符sem_id
*         输入参数：
*   sem_key 信号灯数组的键值
*   sem_val 信号灯数组中信号灯的个数
*   sem_flag 信号等数组的存取权限
*/
int set_sem(key_t sem_key,int sem_val,int sem_flg)
{
 int sem_id;
 Sem_uns sem_arg;

    //测试由sem_key标识的信号灯数组是否已经建立
  if((sem_id = get_ipc_id("/proc/sysvipc/sem",sem_key)) < 0 )
    {
      //semget新建一个信号灯,其标号返回到sem_id
      if((sem_id = semget(sem_key,1,sem_flg)) < 0)
             {
         perror("semaphore create error");
         exit(EXIT_FAILURE);
             }

         //设置信号灯的初值
         sem_arg.val = sem_val;
         if(semctl(sem_id,0,SETVAL,sem_arg) <0)
              {
         perror("semaphore set error");
         exit(EXIT_FAILURE);
                }
      }

   return sem_id;
}

/*
*   set_shm函数建立一个具有n个字节 的共享内存区
*         如果建立成功，返回 一个指向该内存区首地址的指针shm_buf
```

```
*        输入参数:
*   shm_key 共享内存的键值
*   shm_val 共享内存字节的长度
*   shm_flag 共享内存的存取权限
*/
char * set_shm(key_t shm_key,int shm_num,int shm_flg)
{
   int i,shm_id;
   char * shm_buf;

        //测试由shm_key标识的共享内存区是否已经建立
   if((shm_id = get_ipc_id("/proc/sysvipc/shm",shm_key)) < 0 )
        {
      //shmget新建 一个长度为shm_num字节的共享内存,其标号返回到shm_id
      if((shm_id = shmget(shm_key,shm_num,shm_flg)) <0)
              {
       perror("shareMemory set error");
       exit(EXIT_FAILURE);
              }
      //shmat将由shm_id标识的共享内存附加给指针shm_buf
      if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0)
              {
       perror("get shareMemory error");
       exit(EXIT_FAILURE);
              }
      for(i=0; i<shm_num; i++) shm_buf[i] = 0; //初始为0
   }
   //shm_key标识的共享内存区已经建立,将由shm_id标识的共享内存附加给指针
shm_buf
   else if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0)
       {
       perror("get shareMemory error");
       exit(EXIT_FAILURE);
       }

   return shm_buf;
}

/*
*   set_msq函数建立一个//进程自定义的键盘中断信号处理函数
*   如果建立成功，返回 一个消息队列的标识符msq_id
*   输入参数:
*   msq_key 消息队列的键值
```

```
*   msq_flag 消息队列的存取权限
*/
int set_msq(key_t msq_key,int msq_flg)
{
   int msq_id;

    //测试由msq_key标识的消息队列是否已经建立
   if((msq_id = get_ipc_id("/proc/sysvipc/msg",msq_key)) < 0 )
    {
      //msgget新建一个消息队列,其标号返回到msq_id
      if((msq_id = msgget(msq_key,msq_flg)) < 0)
        {
         perror("messageQueue set error");
       exit(EXIT_FAILURE);
        }
    }
   return msq_id;
}

//收到中断信号后释放申请的IPC
void sigcat(){

    semctl(prod_sem,0,IPC_RMID);
    semctl(cons_sem,0,IPC_RMID);
    semctl(pmtx_sem,0,IPC_RMID);
    semctl(cmtx_sem,0,IPC_RMID);

    shmdt(cget_ptr);
    shmdt(pput_ptr);
    shmdt(buff_ptr);
    exit(0);
}

void ipc_init()
{
  //共享内存使用的变量
  buff_key = 101;//缓冲区任给的键值
  buff_num = 8;  //缓冲区任给的长度
  pput_key = 102;//生产者放产品指针的键值
  cget_key = 103;//消费者放产品指针的键值
  pput_num = 1;  //写位置记录单元数
  cget_num = 1;  //读位置记录单元数
  shm_flg = IPC_CREAT | 0644;//共享内存读写权限
```

```
  //获取缓冲区使用的共享内存，buff_ptr指向缓冲区首地址
  buff_ptr = (char *)set_shm(buff_key,buff_num,shm_flg);
  //获取生产者放产品位置指针pput_ptr
  pput_ptr = (int *)set_shm(pput_key,pput_num,shm_flg);
  //获取消费者取产品指针，cget_ptr指向索引地址
  cget_ptr = (int *)set_shm(cget_key,cget_num,shm_flg);

  //信号量使用的变量
  prod_key = 201;//生产者同步信号灯键值
  pmtx_key = 202;//生产者互斥信号灯键值
  cons_key = 301;//消费者同步信号灯键值
  cmtx_key = 302;//消费者互斥信号灯键值
  sem_flg = IPC_CREAT | 0644;

  //生产者同步信号灯初值设为缓冲区最大可用量
  sem_val = buff_num;
  //获取生产者同步信号灯，引用标识存prod_sem
  prod_sem = set_sem(prod_key,sem_val,sem_flg);

  //消费者初始无产品可取，同步信号灯初值设为0
  sem_val = 0;
  //获取消费者同步信号灯，引用标识存cons_sem
  cons_sem = set_sem(cons_key,sem_val,sem_flg);

  //生产者互斥信号灯初值为1
  sem_val = 1;
  //获取生产者互斥信号灯，引用标识存pmtx_sem
  pmtx_sem = set_sem(pmtx_key,sem_val,sem_flg);

  //消费者互斥信号灯初值为1
  sem_val = 1;
  //获取消费者互斥信号灯，引用标识存pmtx_sem
  cmtx_sem = set_sem(cmtx_key,sem_val,sem_flg);

}
```

3) 在当前新文件夹中建立生产者程序 producer.c

```
/*
  *  Filename      : producer.c
  *  copyright          : (C) 张鸿烈
  *  Function           : 建立并模拟生产者进程
*/
```

```
#include "ipc.h"

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

  //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
  if(argv[1] != NULL)  rate = atoi(argv[1]);
  else rate = 1;  //不指定为1秒

  signal(SIGINT,(sighandler_t)sigcat); //注册一个本进程处理键盘中断的函
数

  //初始化同步机制
  ipc_init();

  //循环执行模拟生产者不断放产品
  while(1){
      //如果缓冲区满则生产者阻塞
     P(prod_sem);
     //如果另一生产者正在放产品，本生产者阻塞
     P(pmtx_sem);

      //用写一字符的形式模拟生产者放产品，报告本进程号和放入的字符及存放
的位置
      buff_ptr[*pput_ptr] = 'A'+ *pput_ptr;
      printf("%d producer put: %c to
      Buffer[%d]\n",getpid(),buff_ptr[*pput_ptr],*pput_ptr);

       //模拟写延迟
       sleep(rate);

      //存放位置循环下移
      *pput_ptr = (*pput_ptr+1) % buff_num;

     //唤醒阻塞的生产者
     V(pmtx_sem);
     //唤醒阻塞的消费者
     V(cons_sem);
    }

   return EXIT_SUCCESS;
}
```

4) 在当前新文件夹中建立消费者程序 consumer.c

```
/*
    * Filename          : consumer.c
    * copyright         : (C)  张鸿烈
    * Function          : 建立并模拟消费者进程
*/

#include "ipc.h"

int main(int argc,char *argv[])
{
  //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
  if(argv[1] != NULL)  rate = atoi(argv[1]);
  else rate = 1;  //不指定为1秒

  signal(SIGINT,(sighandler_t)sigcat); //注册一个本进程处理键盘中断的函
数

  //初始化同步机制
  ipc_init();

  //循环执行模拟消费者不断取产品
  while(1){
     //如果无产品消费者阻塞
     P(cons_sem);
      //如果另一消费者正在取产品，本消费者阻塞
      P(cmtx_sem);

        //用读一字符的形式模拟消费者取产品，报告本进程号和获取的字符及读
取的位置
        printf("%d consumer get: %c from
        Buffer[%d]\n",getpid(),buff_ptr[*cget_ptr],*cget_ptr);

        //模拟读延迟
        sleep(rate);

        //读取位置循环下移
        *cget_ptr = (*cget_ptr+1) % buff_num;

      //唤醒阻塞的消费者
      V(cmtx_sem);
       //唤醒阻塞的生产者
```

```
      V(prod_sem);
    }

   return EXIT_SUCCESS;
}
```

5）在当前文件夹中建立 **Makefile** 项目管理文件

```
hdrs = ipc.h
opts = -g -c
c_src = consumer.c ipc.c
c_obj = consumer.o ipc.o
p_src = producer.c ipc.c
p_obj = producer.o ipc.o

all:        producer consumer

consumer:   $(c_obj)
            gcc $(c_obj) -o consumer
consumer.o: $(c_src) $(hdrs)
            gcc $(opts) $(c_src)

producer: $(p_obj)
            gcc $(p_obj) -o producer
producer.o: $(p_src) $(hdrs)
            gcc $(opts) $(p_src)
clean:
            rm consumer producer *.o
```

6）使用 **make** 命令编译连接生成可执行的生产者、消费者程序

```
$ gmake
gcc -g -c producer.c ipc.c
gcc producer.o ipc.o -o producer
gcc -g -c consumer.c ipc.c
gcc consumer.o ipc.o -o consumer
```

7）在当前终端窗体中启动执行速率为 **1** 秒的一个生产者进程

```
$./producer   1
12263 producer put: A to Buffer[0]
12263 producer put: B to Buffer[1]
12263 producer put: C to Buffer[2]
12263 producer put: D to Buffer[3]
12263 producer put: E to Buffer[4]
12263 producer put: F to Buffer[5]
12263 producer put: G to Buffer[6]
12263 producer put: H to Buffer[7]
```

可以看到 12263 号进程在向共享内存中连续写入了 8 个字符后因为缓冲区满而

阻塞。

8） 打开另一终端窗体，进入当前工作目录，从中再启动另一执行速率为 3 的生产者进程:

$ *./producer 3*

可以看到该生产者进程因为缓冲区已满而立即阻塞。

9）再打开另外两个终端窗体，进入当前工作目录，从中启动执行速率为 2 和 4 的两个消费者进程:

$ ./consumer 2

12528 consumer get: B from Buffer[1]
12528 consumer get: D from Buffer[3]
12528 consumer get: F from Buffer[5]
12528 consumer get: H from Buffer[7]
……

$./consumer 4

12529 consumer get: A from Buffer[0]
12529 consumer get: C from Buffer[2]
12529 consumer get: E from Buffer[4]
12529 consumer get: G from Buffer[6]
……

在第一个生产者窗体中生产者 1 被再此唤醒输出:

12263 producer put: B to Buffer[1]
12263 producer put: D to Buffer[3]
12263 producer put: F to Buffer[5]
12263 producer put: H to Buffer[7]
……

在第二个生产者窗体中生产者 2 也被再此被唤醒输出

12264 producer put: A to Buffer[0]
12264 producer put: C to Buffer[2]
12264 producer put: E to Buffer[4]
12264 producer put: G to Buffer[6]

可以看到由于消费者进程读出了写入缓冲区的字符，生产者从新被唤醒继续向读过的缓冲区单元中同步的写入字符。

请用 ctrl+C 将两生产者进程打断，观察两消费者进程是否在读空缓冲区后而阻塞。反之，请用 ctrl+C 将两消费者进程打断，观察两生产者进程是否在写满缓冲区后而阻塞。

4.4 独立实验

抽烟者问题。假设一个系统中有三个抽烟者进程，每个抽烟者不断地卷烟并抽烟。抽烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和胶水。一个抽烟者有烟草，一个有纸，另一个有胶水。系统中还有两个供应者进程，它们无限地供应所有三种材料，但每次仅轮流提供三种材料中的两种。得到缺失的两种材料的抽烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供另外的两种材料。这一过程重

复进行。 请用以上介绍的 IPC 同步机制编程，实现该问题要求的功能。

参考解法分析

假设 T,P,G 分别为吸烟必备材料（T=烟草，P=纸，G=胶水）。供应商每次随机供应（生产）卷烟材料中的两种:T&P，P&G，G&T。

每种材料各放在一个容器中。要求两种材料来自于一个供应者在一次供应时提供的。例如：具有烟草 T 的吸烟者要获得 P&G，P&G 是一个供应者在某一次供应时提供的，不允许从一个供应者提供的 T&P 和另一个供应者提供的 G&T 中各取 P 和 G 进行卷烟。即要求原料成对放入，再按放入时的组合成对的取出，每对材料视为一件产品。

容器对缓冲区利用的约束：当供应者随机放入三种产品时，从缓冲区申请位置，然后放入。由于放入的产品具有随机性，所以产品在缓冲区的位置也就具有随机性，因此对卷烟者而言，就算知道缓冲区有自己想要的产品，但不知道从缓冲区的哪个位置去获得产品。所以要对缓冲区三种产品的存放位置进行约束。

最简单的模拟解决方案：申请一个三个字节共享内存缓冲区，用每个字节放一个材料，缓冲区首单元为放烟草 T 的容器，第二字节 为放纸张 P 的容器，第三字节为放胶水 G 的容器。每次同时操作 3 个容器中的两对组合。

供应者（生产者）进程：用同一个程序产生两个供应商进程，它们使用同一段程序上下文，每次放产品到容器时，必须保证吸烟者拿走旧材料后再放新材料，因此这就需要 3 个放产品的同步信号量对其进行约束；并且两供应商必须互斥的向容器中放入材料。因此这就需要 3 个放产品的互斥信号量对其进行约束。

吸烟者（消费者）进程：用同一个程序产生 3 个吸烟者进程，分别有各自的程序上下文，各为拥有烟草 T 材料需要材料 P&G 的吸烟者，拥有纸张 P 材料需要材料 G&T 的吸烟者，拥有胶水 G 材料需要材料 T&P 的吸烟者。吸烟者欲卷烟首先要获得另外两种所需材料。当容器中没有自己恰好需要的材料时，吸烟者就要等待。并且 3 个吸烟者必须互斥的从容器中获取材料。因此这就需要 3 个取产品的同步信号量，3 个取产品的互斥信号量对其进行约束。

参考解法伪代码：

为实验该问题在系统进程间的并发关系,可采用了 systemV 的 IPC 的信号量和共享内存，这样可以观察到进程在系统级上的并发和调度的情况。

创建 3 个字节的共享内存用作放材料的容器，用以下指针引用:

Char *tobacco 指向共享内存第 1 字节-T 容器
Char *paper 指向共享内存第 2 字节-P 容器
Char *glue 指向共享内存第 3 字节-G 容器

建立信号量，设置初值：

对于供应者：

信号量：TG_Psyn_sem 控制 T&P 容器的同步使用，初值为 1
信号量：PG_Psyn_sem 控制 P&G 容器的同步使用，初值为 1
信号量：GT_Psyn_sem 控制 G&T 容器的同步使用，初值为 1
信号量：TG_Pmtx_sem 控制 T&P 容器的互斥使用，初值为 1

信号量：PG_Pmtx_sem 控制 P&G 容器的互斥使用，初值为 1
信号量：GT_Pmtx_sem 控制 G&T 容器是互斥使用，初值为 1
对于吸烟者：
信号量：TG_Csyn_sem 控制 T&P 容器的同步使用，初值为 0
信号量：PG_Csyn_sem 控制 P&G 容器的同步使用，初值为 0
信号量：GT_Csyn_sem 控制 G&T 容器的同步使用，初值为 0
信号量：TG_Cmtx_sem 控制 T&P 容器的互斥使用，初值为 1
信号量：PG_Cmtx_sem 控制 P&G 容器的互斥使用，初值为 1
信号量：GT_Cmtx_sem 控制 G&T 容器是互斥使用，初值为 1

供应商程序：

```
{
        建立和初始化以上设计的共享内存和信号量;
        循环 2 次{
            创建两个供应者子进程;
            每个子进程循环做{
                产生一个 0-2 之间随机数
                {

                情况 0：
                        TG_Psyn_sem 上的 P 操作;
                        TG_Pmtx_sem 上的 P 操作;
                        放烟草 T 到 T 容器;
                        放纸张 P 到 P 容器;
                        TG_Pmtx_sem 上的 V 操作;
                        TG_Csyn_sem 上的 V 操作;
                   情况 1：
                        PG_Psyn_sem 上的 P 操作;
                        PG_Pmtx_sem 上的 P 操作;
                        放胶水 G 到 G 容器;
                        放纸张 P 到 P 容器;
                        PG_Pmtx_sem 上的 V 操作;
                        PG_Csyn_sem 上的 V 操作;
                   情况 2：
                        GT_Psyn_sem 上的 P 操作;
                        GT_Pmtx_sem 上的 P 操作;
                        放烟草 T 到 T 容器;
                        放纸张 P 到 P 容器;
                        GT_Pmtx_sem 上的 V 操作;
                        GT_Csyn_sem 上的 V 操作;
                 }
            }
    }
    父进程等待
```

}

吸烟者程序：
{
建立和初始化以上设计的共享内存和信号量；
创建拥有烟草的吸烟者子进程；
拥有烟草的吸烟者进程循环做{
PG_Csyn_sem 上的 P 操作；
PG_Cmtx_sem 上的 P 操作；
从容器 P 取纸张 P；
从容器 G 取胶水 G；
PG_Cmtx_sem 上的 V 操作；
PG_Psyn_sem 上的 V 操作；
}
创建拥有纸张的吸烟者子进程；
拥有纸张的吸烟者进程循环做{
GT_Csyn_sem 上的 P 操作；
GT_Cmtx_sem 上的 P 操作
从容器 G 取胶水 G；
从容器 T 取烟草 T；
GT_Cmtx_sem 上的 V 操作；
GT_Psyn_sem 上的 V 操作；
}
创建拥有胶水的吸烟者子进程；
拥有胶水的吸烟者进程循环做{
PG_Csyn_sem 上的 P 操作；
PG_Cmtx_sem 上的 P 操作；
从容器 T 取烟草 T；
从容器 P 取纸张 P；
PG_Cmtx_sem 上的 V 操作；
PG_Psyn_sem 上的 V 操作；
}
父进程等待；

程序执行结果的测试和分析：

先启动供应商进程
```
$ ./provider
1806 供应商提供：烟草T,纸张P
1806 供应商提供：胶水G,烟草T
1806 供应商提供：纸张P,胶水G
```

由于没有吸烟者，供应商提供的产品放满缓冲区后，供应商睡眠。

再在另一终端窗体中启动消费者进程：
./smoker
1904 吸烟者有胶水G,取到烟草T,纸张 P.开始吸烟...
1905 吸烟者有烟草T,取到纸张P,胶水 G.开始吸烟...
1906 吸烟者有纸张P,取到胶水G,烟草 T.开始吸烟...
1904 吸烟者有胶水G,取到烟草T,纸张 P.开始吸烟...
1906 吸烟者有纸张 P,取到胶水 G,烟草 T.开始吸烟...

供应商被唤醒继续提供产品:

1807 供应商提供: 烟草T,纸张P
1806 供应商提供: 胶水G,烟草T
1807 供应商提供: 纸张P,胶水G
1806 供应商提供: 胶水G,烟草T
1807 供应商提供: 纸张P,胶水G
1806 供应商提供: 烟草T,纸张P
1806 供应商提供: 胶水 G,烟草 T

如果先启动吸烟者
$ ./smoker

吸烟者阻塞

再启动供应商:

$ ./provider
1912 供应商提供: 烟草T,纸张P
1912 供应商提供: 纸张P,胶水G
1913 供应商提供: 烟草T,纸张P
1912 供应商提供: 纸张P,胶水G

吸烟者得到产品开始吸烟:

1908 吸烟者有胶水G,取到烟草T,纸张 P.开始吸烟...
1909 吸烟者有烟草T,取到纸张P,胶水 G.开始吸烟...
1908 吸烟者有胶水G,取到烟草T,纸张 P.开始吸烟...
1909 吸烟者有烟草T,取到纸张P,胶水 G.开始吸烟...
1909 吸烟者有烟草T,取到纸张P,胶水 G.开始吸烟...
1910 吸烟者有纸张P,取到胶水G,烟草 T.开始吸烟...
1908 吸烟者有胶水G,取到烟草T,纸张 P.开始吸烟...

4.5 实验要求

根据示例实验程序和独立实验程序中观察和记录的信息，结合生产者/消费者问题和抽烟者问题的算法的原理，说明真实操作系统中提供的并发进程同步机制是怎样实现和解决同步问题的，它们是怎样应用操作系统教材中讲解的进程同步原理的？对应教材中信号灯的定义，说明信号灯机制是怎样完成进程的互斥和同步的？其中信号量的初值和其值的变化的物理意义是什么？使用多于 4 个的生产者和消费者，以各种不同的启动顺序、不同的执行速率检测以上示例程序和独立实验程序是否都能满足同步的要求。根据实验程序、调试过程和结果分析写出实验报告。

实验五、进程互斥实验

5.1 实验目的

进一步研究和实践操作系统中关于并发进程同步与互斥操作的一些经典问题的解法，加深对于非对称性互斥问题有关概念的理解。观察和体验非对称性互斥问题的并发控制方法。进一步了解 Linux 系统中 IPC 进程同步工具的用法，训练解决对该类问题的实际编程、调试和分析问题的能力。

5.2 实验说明

以下示例实验程序应能模拟一个读者/写者问题,它应能实现一下功能：

1. 任意多个读者可以同时读；
2. 任意时刻只能有一个写者写；
3. 如果写者正在写，那么读者就必须等待；
4. 如果读者正在读，那么写者也必须等待；
5. 允许写者优先；
6. 防止读者或写者发生饥饿。

为了能够体验 IPC 机制的消息队列的用法，本示例程序采用了 Theaker & Brookes 提出的消息传递算法。该算法中有一控制进程，带有 3 个不同类型的消息信箱，它们分别是：读请求信箱、写请求信箱和操作完成信箱。读者需要访问临界资源时首先要向控制进程发送读请求消息，写者需要访问临界资源时也要先向控制进程发送写请求消息，在得到控制进程的允许消息后方可进入临界区读或写。读或写者在完成对临界资源的访问后还要向控制进程发送操作完成消息。控制进程使用一个变量 count 控制读写者互斥的访问临界资源并允许写者优先。 count 的初值需要一个比最大读者数还要大的数，本例取值为 100。当 count 大于 0 时说明没有新的读写请求，控制进程接收读写者新的请求，如果收到读者完成消息，对 count 的值加 1，如果收到写者请求消息，count 的值减 100，如果收到读者请求消息，对 count 的值减 1。 当 count 等于 0 时说明写者正在写，控制进程等待写者完成后再次令 count 的值等于 100。 当 count 小于 0 时说明读者正在读，控制进程等待读者完成后对 count 的值加 1。

5.3 示例实验

我们可以利用上节实验中介绍的 IPC 机制中的消息队列来实验一下以上使用消息传递算法的读写者问题的解法，看其是否能够满足我们的要求。仍采用共享内存模拟要读写的对象，一写者向共享内存中写入一串字符后，多个读者可同时从共享内存中读出该串字符。

1）在新建的文件夹中建立以下 ipc.h 头文件

```
/*
   *   Filename              :   ipc.h
   *   copyright             : (C) 2006 by  张鸿烈
```

```
    * Function              : 声明 IPC 机制的函数原型和全局变量
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>

#define BUFSZ               256
#define MAXVAL              100
#define STRSIZ              8
#define WRITERQUEST         1    //写请求标识
#define READERQUEST         2    //读请求标识
#define FINISHED      3     //读写完成标识

/*信号灯控制用的共同体*/
typedef union semuns {
      int val;
} Sem_uns;

/*  消息结构体*/
typedef struct msgbuf {
      long mtype;
      int    mid;
} Msg_buf;

  key_t buff_key;
  int buff_num;
  char *buff_ptr;
  int shm_flg;

  int quest_flg;
  key_t quest_key;
  int     quest_id;

  int respond_flg;
  key_t respond_key;
  int     respond_id;

int get_ipc_id(char *proc_file,key_t key);

char *set_shm(key_t shm_key,int shm_num,int shm_flag);

int set_msq(key_t msq_key,int msq_flag);

int set_sem(key_t sem_key,int sem_val,int sem_flag);
```

```
int down(int sem_id);
int up(int sem_id);
```

2）在新建的文件夹中建立以下 ipc.c 文件。

```
#include "ipc.h"
/*
   * get_ipc_id() 从/proc/sysvipc/文件系统中获取IPC的id号
   * pfile: 对应/proc/sysvipc/目录中的IPC文件分别为
   *        msg-消息队列,sem-信号量,shm-共享内存
   * key:   对应要获取的IPC的id号的键值
*/
int get_ipc_id(char *proc_file,key_t key)
{
   FILE *pf;
   int i,j;
   char line[BUFSZ],colum[BUFSZ];

   if((pf = fopen(proc_file,"r")) == NULL){
    perror("Proc file not open");
      exit(EXIT_FAILURE);
    }
   fgets(line, BUFSZ,pf);
   while(!feof(pf)){
     i = j = 0;
     fgets(line, BUFSZ,pf);
     while(line[i] == ' ') i++;
     while(line[i] !=' ') colum[j++] = line[i++];
     colum[j] = '\0';
     if(atoi(colum) != key) continue;
     j=0;
     while(line[i] == ' ') i++;
     while(line[i] !=' ') colum[j++] = line[i++];
     colum[j] = '\0';
     i = atoi(colum);
     close(pf);
     return i;
    }
    close(pf);
    return -1;
}

/*
 * 信号灯上的down/up操作
 * semid:信号灯数组标识符
 * semnum:信号灯数组下标
```

```
 * buf:操作信号灯的结构
*/
int down(int sem_id)
{
  struct sembuf buf;
  buf.sem_op = -1;
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  if((semop(sem_id,&buf,1)) <0) {
    perror("down error ");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}

int up(int sem_id)
{
  struct sembuf buf;
  buf.sem_op = 1;
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  if((semop(sem_id,&buf,1)) <0) {
    perror("up error ");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}

int set_sem(key_t sem_key,int sem_val,int sem_flg)
{
 int sem_id;
 Sem_uns sem_arg;

    //信号量是否建立
   if((sem_id = get_ipc_id("/proc/sysvipc/sem",sem_key)) < 0 )
    {
      //semget建立一个信号灯,其标号返回到sem_id
      if((sem_id = semget(sem_key,1,sem_flg)) < 0)
       {
         perror("semaphore create error");
         exit(EXIT_FAILURE);
        }

      sem_arg.val = sem_val; //设置信号量的初值
      if(semctl(sem_id,0,SETVAL,sem_arg) <0)
```

```
        {
          perror("semaphore set error");
          exit(EXIT_FAILURE);
         }
    }

    return sem_id;
}


char * set_shm(key_t shm_key,int shm_num,int shm_flg)
{
    int shm_id;
    char * shm_buf;

    if((shm_id = get_ipc_id("/proc/sysvipc/shm",shm_key)) < 0 )
     {
      //shmget建立一个长度为shm_num字节的共享内存,其标号返回到shm_id
      if((shm_id = shmget(shm_key,shm_num,shm_flg)) <0)
       {
         perror("shareMemory set error");
         exit(EXIT_FAILURE);
       }
      if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0)
       {
         perror("get shareMemory error");
         exit(EXIT_FAILURE);
       }
      *shm_buf = 0;
     }

    if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0)
     {
        perror("get shareMemory error");
        exit(EXIT_FAILURE);
      }

    return shm_buf;
}


int set_msq(key_t msq_key,int msq_flg)
{

    int msg_id;
```

```
    //测消息队列是否建立
   if((msg_id = get_ipc_id("/proc/sysvipc/msg",msq_key)) < 0 )
    {
      //msgget建立一个消息队列,其标号返回到msg_id
      if((msg_id = msgget(msq_key,msq_flg)) < 0)
        {
         perror("messageQueue set error");
       exit(EXIT_FAILURE);
        }
    }
   return msg_id;
}
```

3)在当前目录中建立如下的控制者程序 control.c

```
/*
* Filename
* copyright
* Function
: control.c
: (C) 2006 by 张鸿烈
: 建立并模拟控制者进程
*/
#include "ipc.h"
int count = MAXVAL;

//进程自定义的键盘中断信号处理函数原型
typedef void (*sighandler_t) (int);
void sigcat(void);

//收到中断信号后释放申请的IPC
void sigcat(){
  struct msqid_ds msg_ds;
  msgctl(quest_id,IPC_RMID,&msg_ds);
  msgctl(respond_id,IPC_RMID,&msg_ds);
  shmdt(buff_ptr);
  exit(0);
}

int main(int argc,char *argv[])
{
int i;
int rate;
Msg_buf msg_arg;
struct msqid_ds msg_inf;
```

```
//注册一个本进程处理键盘中断的函数
signal(SIGINT,(sighandler_t)sigcat);
//建立一个共享内存先写入一串 A 字符模拟要读写的内容
buff_key = 101;
buff_num =STRSIZ+1;
shm_flg = IPC_CREAT | 0644;
buff_ptr = (char *)set_shm(buff_key,buff_num,shm_flg);
for(i=0; i<STRSIZ; i++) buff_ptr[i] = 'A';
buff_ptr[i] = '\0';
//建立一条请求消息队列
quest_flg = IPC_CREAT| 0644;
quest_key = 201;
quest_id = set_msq(quest_key,quest_flg);
//建立一条响应消息队列
respond_flg = IPC_CREAT|0644;
respond_key = 202;
respond_id = set_msq(respond_key,respond_flg);
//控制进程准备接收和响应读写者的消息
printf("Wait quest \n");
while(1){
//当 count 大于 0 时说明没有新的读写请求,查询是否有任何新请求
if(count > 0){
quest_flg = IPC_NOWAIT; //以非阻塞方式接收请求消息
if(msgrcv(quest_id,&msg_arg,sizeof(msg_arg),
WRITERQUEST,quest_flg) >= 0){
//有写者请求,允许写者写
count -= MAXVAL;
msg_arg.mtype = msg_arg.mid;
msgsnd(respond_id,&msg_arg,sizeof(msg_arg),0);

printf("%d quest write \n",msg_arg.mid);
}
else if(msgrcv(quest_id,&msg_arg,sizeof(msg_arg),
FINISHED,quest_flg) >= 0){
//有读者完成
count++;
printf("%d reader finished\n",msg_arg.mid);
}
else if(msgrcv(quest_id,&msg_arg,sizeof(msg_arg),
READERQUEST,quest_flg) >=0){
//有读者请求,允许读者读
count --;
msg_arg.mtype = msg_arg.mid;
msgsnd(respond_id,&msg_arg,sizeof(msg_arg),0);
```

```
printf("%d quest read\n",msg_arg.mid);
}
}
//当 count 等于 0 时说明写者正在写,等待写完成
if(count == 0){
//以阻塞方式接收消息
msgrcv(quest_id,&msg_arg,sizeof(msg_arg),FINISHED,0);
count = MAXVAL;
printf("%d write finished\n",msg_arg.mid);
if(msgrcv(quest_id,&msg_arg,sizeof(msg_arg),
READERQUEST,quest_flg) >=0){
//有读者请求,允许读者读
count --;
msg_arg.mtype = msg_arg.mid;
msgsnd(respond_id,&msg_arg,sizeof(msg_arg),0);
printf("%d quest read\n",msg_arg.mid);
}
}
//当 count 小于 0 时说明有多个读者正在读,等待它们读完
while(count < 0){
//以阻塞方式接收消息
msgrcv(quest_id,&msg_arg,sizeof(msg_arg),FINISHED,0);
count ++;
printf("%d reader finish\n",msg_arg.mid);
}
}
return EXIT_SUCCESS;
}
```

4）在当前目录中建立如下的读者程序 reader.c

```
/*
   *    Filename              :   reader.c
   *    copyright             : (C) 2006 by 张鸿烈
    * Function                : 建立并模拟读者进程
*/
#include "ipc.h"

int main(int argc,char *argv[])
{
   int i;
   int rate;
   Msg_buf    msg_arg;

   //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
   if(argv[1] != NULL)    rate = atoi(argv[1]);
   else rate = 3;
```

```
   //附加一个要读内容的共享内存
   buff_key = 101;
   buff_num = STRSIZ+1;
   shm_flg = IPC_CREAT | 0644;
   buff_ptr = (char *)set_shm(buff_key,buff_num,shm_flg);
  //联系一个请求消息队列
   quest_flg = IPC_CREAT| 0644;
   quest_key = 201;
   quest_id =    set_msq(quest_key,quest_flg);
   //联系一个响应消息队列
   respond_flg = IPC_CREAT| 0644;
   respond_key = 202;
   respond_id =    set_msq(respond_key,respond_flg);
   //循环请求读
   msg_arg.mid = getpid();
   while(1){
         //发读请求消息
         msg_arg.mtype = READERQUEST;
         msgsnd(quest_id,&msg_arg,sizeof(msg_arg),0);
         printf("%d reader quest\n",msg_arg.mid);
      //等待允许读消息
         msgrcv(respond_id,&msg_arg,sizeof(msg_arg),msg_arg.mid,0);
         printf("%d reading: %s\n",msg_arg.mid,buff_ptr);
         sleep(rate);
      //发读完成消息
         msg_arg.mtype = FINISHED;
         msgsnd(quest_id,&msg_arg,sizeof(msg_arg),quest_flg);
    }
    return EXIT_SUCCESS;
}
```

5）在当前目录中建立如下的写者程序 writer.c

```
/*
   *    Filename              :    writer.c
   *    copyright             : (C) 2006 by 张鸿烈
    * Function                :  建立并模拟写者进程
*/
#include "ipc.h"

int main(int argc,char *argv[])
{
   int i,j=0;
   int rate;
   Msg_buf    msg_arg;

   //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
   if(argv[1] != NULL)    rate = atoi(argv[1]);
```

```
    else rate = 3;
    //附加一个要读内容的共享内存
    buff_key = 101;
    buff_num = STRSIZ+1;
    shm_flg = IPC_CREAT | 0644;
    buff_ptr = (char *)set_shm(buff_key,buff_num,shm_flg);
    //联系一个请求消息队列
    quest_flg = IPC_CREAT| 0644;
    quest_key = 201;
    quest_id =    set_msq(quest_key,quest_flg);
    //联系一个响应消息队列
    respond_flg = IPC_CREAT| 0644;
    respond_key = 202;
    respond_id =    set_msq(respond_key,respond_flg);
    //循环请求写
    msg_arg.mid = getpid();
    while(1){
          //发写请求消息
          msg_arg.mtype = WRITERQUEST;
          msgsnd(quest_id,&msg_arg,sizeof(msg_arg),0);
          printf("%d writer quest\n",msg_arg.mid);
       //等待允许写消息
          msgrcv(respond_id,&msg_arg,sizeof(msg_arg),msg_arg.mid,0);
          //写入 STRSIZ 个相同的字符
          for(i=0; i<STRSIZ; i++) buff_ptr[i] = 'A'+j;
          j = (j+1) % STRSIZ ; //按  STRSIZ  循环变换字符
          printf("%d writing: %s\n",msg_arg.mid,buff_ptr);
          sleep(rate);
       //发写完成消息
          msg_arg.mtype = FINISHED;
          msgsnd(quest_id,&msg_arg,sizeof(msg_arg),0);
     }
     return EXIT_SUCCESS;
}
```

6) 在当前目录中建立如下 Makefile 文件

```
hdrs = ipc.h
c_src = control.c ipc.c
c_obj = control.o ipc.o
r_src = reader.c ipc.c
r_obj = reader.o ipc.o
w_src = writer.c ipc.c
w_obj = writer.o ipc.o
opts    = -g -c

all:          control reader writer
```

```
control:   $(c_obj)
           gcc $(c_obj) -o control
control.o: $(c_src) $(hdrs)
           gcc $(opts) $(c_src)

reader:    $(r_obj)
           gcc $(r_obj) -o reader
reader.o:  $(r_src) $(hdrs)
           gcc $(opts) $(r_src)

writer:    $(w_obj)
           gcc $(w_obj) -o writer
writer.o:$(w_src) $(hdrs)
           gcc $(opts) $(w_src)

clean:
           rm control reader writer *.o
```

7）在当前目录中执行 make 命令编译连接，生成读写者，控制者程序：

```
$gmake
gcc -g -c control.c ipc.c
gcc control.o ipc.o -o control
gcc -g -c reader.c ipc.c
gcc reader.o ipc.o -o reader
gcc -g -c writer.c ipc.c
gcc writer.o ipc.o -o writer
$
```

8）可打开四个以上的终端模命令窗体，都将进入当前工作目录。先在一窗体中启动./control 程序：

```
$ ./control
Wait quest
```

现在控制进程已经在等待读写者的请求。

9）再在另两不同的窗体中启动两个读者，一个让它以 1 秒的延迟快一些读，一个让它以 10 秒的延迟慢一些读：

```
$ . ./reader 10
3903 reader quest
3903 reading: AAAAAAAA
......
$./reader 1
3904 reader quest
3904 reading: AAAAAAAA
3904 reader quest
3904 reading: AAAAAAAA
3904 reader quest
3904 reading: AAAAAAAA
3904 reader quest
```

.....
现在可以看到控制进程开始响应读者请求，让多个读者同时进入临界区读：
Wait quest
3903 quest read
3904 quest read
3904 quest read
3904 reader finished
3904 reader finished
3904 quest read
3904 reader finished
3903 reader finished
.....
10）再在另一终端窗体中启动一个延迟时间为 8 秒的写者进程：
$ ./writer 8
3906 writer quest
3906 writing: AAAAAAAA
3906 writer quest
3906 writing: BBBBBBBB
3906 writer quest
3906 writing: CCCCCCCC
.....
此时可以看到控制进程在最后一个读者读完后首先响应写者请求：
3906 quest write
3904 reader finish
3903 reader finish
3906 write finished
.....
在写者写完后两个读者也同时读到了新写入的内容：
3903 reader quest
3903 reading: BBBBBBBB
3903 reader quest
3903 reading: CCCCCCCC
.....
3904 reading: BBBBBBBB
3904 reader quest
3904 reading: CCCCCCCC
.....
请仔细观察各读者和写者的执行顺序：可以看出在写者写时不会有读者进入，在有读者读时不会有写者进入，但一旦读者全部退出写者会首先进入。分析以上输出可以看出该算法实现了我们要求的读写者问题的功能。

10）请按与以上不同的启动顺序、不同的延迟时间，启动更多的读写者。观察和分析是否仍能满足我们要求的读写者问题的功能。

11) 请修改以上程序，制造一个读者或写者的饥饿现象。观察什么是饥饿现象，说明为什么会发生这种现象。

5.4 独立实验

理发店问题：假设理发店的理发室中有 3 个理发椅子和 3 个理发师，有一个可容纳 4 个顾客坐等理发的沙发。此外还有一间等候室，可容纳 13 位顾客等候进入理发室。顾客如果发现理发店中顾客已满（超过 20 人），就不进入理发店。

在理发店内，理发师一旦有空就为坐在沙发上等待时间最长的顾客理发，同时空出的沙发让在等候室中等待时间最长的的顾客就坐。顾客理完发后，可向任何一位理发师付款。但理发店只有一本现金登记册，在任一时刻只能记录一个顾客的付款。理发师在没有顾客的时候就坐在理发椅子上睡眠。理发师的时间就用在理发、收款、睡眠上。

请利用 linux 系统提供的 IPC 进程通信机制实验并实现理发店问题的一个解法。

参考解法分析：

该解法利用消息队列的每条消息代表每个顾客，将进入等候室的顾客组织到一个队列，将坐入沙发的顾客组织到另一个队列。理发师从沙发队列请出顾客，空出的沙发位置再从等候室请入顾客进入沙发队列。三个理发师进程使用相同的程序段上下文，所有顾客使用同一个程序段上下文。这样可避免产生太多进程，以便节省系统资源。

参考解法伪代码：

```
理发师程序（Barber）
 {
    建立一个互斥帐本信号量：  s_account,初值=1;
    建立一个同步顾客信号量：  s_customer,初值=0;
        建立沙发消息队列：  q_sofa;
        建立等候室消息队列：q_wait;
        建立 3 个理发师进程：  b1_pid, b2_pid, b3_pid;
    每个理发师进程作：
        while(1)
          {
            以阻塞方式从沙发队列接收一条消息，
            如果有消息，则消息出沙发队列(模拟一顾客理发);
                唤醒顾客进程(让下一顾客坐入沙发)。
                用进程休眠一个随机时间模拟理发过程。
                 理完发,使用帐本信号量记账。
                 互斥的获取账本
                     记账
                  唤醒用账本理发师者

            否则没有消息(沙发上无顾客)
                则理发师进程在沙发队列上睡眠;

           当沙发队列有消息时被唤醒(有顾客坐入沙发)。
```

```
        }
}

顾客程序(customer)
{
    while(1)
    {
        取沙发队列消息数(查沙发上顾客数) ；
        如果消息数小于 4(沙发没座满）
            以非阻塞方式从等候室队列接收一条消息(查等候室有顾客否)，
            如果有消息将接收到的消息发送到沙发队列(等候室顾客坐入沙发)；
            否则发送一条消息到沙发队列(新来的顾客直接坐入沙发)；
        否则(沙发坐满)
            取等候室队列消息数(查等候室顾客数)  ；
            如果消息数小于 13
                发送一条消息到等候室队列(等候室没满,新顾客进等候室)；
            否则
                在顾客同步信号量上睡眠(等候室满暂不接待新顾客)；

        用进程休眠一个随机时间模拟顾客到达的时间间隔。
        }
}
```

程序执行结果的测试和分析：

假设先运行理发师程序 ：
$ ./barber
8377 号理发师睡眠
8379 号理发师睡眠
8378 号理发师睡眠
此时理发师进程由于无顾客而阻塞。

再在另一窗体运行顾客程序：
$ ./customer
1 号新顾客坐入沙发
2 号新顾客坐入沙发
3 号新顾客坐入沙发
4 号新顾客坐入沙发
5 号新顾客坐入沙发
6 号新顾客坐入沙发
7 号新顾客坐入沙发
8 号新顾客坐入沙发
9 号新顾客坐入沙发

10 号新顾客坐入沙发
11 号新顾客坐入沙发
12 号新顾客坐入沙发
沙发坐满　13 号顾客在等候室等候
13 号顾客从等候室坐入沙发
沙发坐满　14 号顾客在等候室等候
14 号顾客从等候室坐入沙发
沙发坐满　15 号顾客在等候室等候
15 号顾客从等候室坐入沙发
沙发坐满　16 号顾客在等候室等候
16 号顾客从等候室坐入沙发
17 号新顾客坐入沙发
沙发坐满　18 号顾客在等候室等候
18 号顾客从等候室坐入沙发
沙发坐满　19 号顾客在等候室等候
19 号顾客从等候室坐入沙发
沙发坐满　20 号顾客在等候室等候
20 号顾客从等候室坐入沙发
沙发坐满　21 号顾客在等候室等候
21 号顾客从等候室坐入沙发
…

在理发师窗体理发师进程被唤醒：
8603 号理发师为 1 号顾客理发
8603 号理发师收取 1 号顾客交费
8604 号理发师为 2 号顾客理发
8605 号理发师为 3 号顾客理发
8603 号理发师睡眠
8604 号理发师收取 2 号顾客交费
8603 号理发师为 4 号顾客理发
8604 号理发师睡眠
8604 号理发师为 5 号顾客理发
8605 号理发师收取 3 号顾客交费
8605 号理发师睡眠
8603 号理发师收取 4 号顾客交费
8605 号理发师为 6 号顾客理发
8603 号理发师睡眠
8603 号理发师为 7 号顾客理发
8604 号理发师收取 5 号顾客交费
8604 号理发师睡眠
8605 号理发师收取 6 号顾客交费
8604 号理发师为 8 号顾客理发
…

反之，如果先运行顾客程序：
$ ./customer
1 号新顾客坐入沙发
2 号新顾客坐入沙发
3 号新顾客坐入沙发
4 号新顾客坐入沙发
沙发坐满　5 号顾客在等候室等候
沙发坐满　6 号顾客在等候室等候
沙发坐满　7 号顾客在等候室等候
沙发坐满　8 号顾客在等候室等候
沙发坐满　9 号顾客在等候室等候
沙发坐满　10 号顾客在等候室等候
沙发坐满　11 号顾客在等候室等候
沙发坐满　12 号顾客在等候室等候
沙发坐满　13 号顾客在等候室等候
沙发坐满　14 号顾客在等候室等候
沙发坐满　15 号顾客在等候室等候
沙发坐满　16 号顾客在等候室等候
沙发坐满　17 号顾客在等候室等候
等候室满　18 号顾客没有进入理发店
当 18 号顾客到达时理发店 20 个位置已满，顾客进程阻塞(假设理发师进程没运行表示三个理发师正坐在 3 个理发椅上睡觉) 。

再运行理发师程序：
$ ./barber
8557 号理发师睡眠
8557 号理发师为 1 号顾客理发
8558 号理发师睡眠
8558 号理发师为 2 号顾客理发
8559 号理发师睡眠
8559 号理发师为 3 号顾客理发
8557 号理发师收取 1 号顾客交费
8557 号理发师睡眠
8557 号理发师为 4 号顾客理发
8558 号理发师收取 2 号顾客交费
8558 号理发师睡眠
8559 号理发师收取 3 号顾客交费
…

在顾客窗体顾客进程被唤醒重新开始运行：
5 号顾客从等候室坐入沙发
6 号顾客从等候室坐入沙发
7 号顾客从等候室坐入沙发

8 号顾客从等候室坐入沙发
9 号顾客从等候室坐入沙发
沙发坐满　18 号顾客在等候室等候
10 号顾客从等候室坐入沙发
11 号顾客从等候室坐入沙发
沙发坐满　19 号顾客在等候室等候
12 号顾客从等候室坐入沙发
13 号顾客从等候室坐入沙发
14 号顾客从等候室坐入沙发
沙发坐满　20 号顾客在等候室等候
…

5.5 实验要求

总结和分析示例实验和独立实验中观察到的调试和运行信息，说明您对与解决非对称性互斥操作的算法有哪些新的理解和认识？ 为什么会出现进程饥饿现象？本实验的饥饿现象是怎样表现的？怎样解决并发进程间发生的饥饿现象？您对于并发进程间使用消息传递解决进程通信问题有哪些新的理解和认识？根据实验程序、调试过程和结果分析写出实验报告。

实验六、死锁问题实验

6.1 实验目的

通过本实验观察死锁产生的现象，考虑解决死锁问题的方法。从而进一步加深对于死锁问题的理解。掌握解决死锁问题的几种算法的编程和调试技术。练习怎样构造管程和条件变量，利用管程机制来避免死锁和饥饿问题的发生。

6.2 实验说明

以下示例实验程序采用经典的管程概念模拟和实现了哲学家就餐问题。其中仍使用以上介绍的 IPC 机制实现进程的同步与互斥操作。为了利用管程解决死锁问题，本示例程序利用了 C++语言的类机制构造了哲学家管程，管程中的条件变量的构造利用了 linux 的 IPC 的信号量机制，利用了共享内存表示每个哲学家的当前状态。

6.3 示例实验

1）在新建的文件夹中建立以下 dp.h 头文件

```
/*
  *  Filename          : dp.h
  *  copyright         : (C) 2006 by 张鸿烈
   * Function          : 声明 IPC 机制的函数原型和哲学家管程类
*/

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <sys/wait.h>

/*信号灯控制用的共同体*/
typedef union semuns {
    int val;
} Sem_uns;

//哲学家的 3 个状态（思考、饥饿、就餐）
enum State {thinking,hungry,eating};
```

```
//哲学家管程中使用的信号量
class Sema{
public:
    Sema(int id);
    ~Sema();
    int down(); //信号量加 1
    int up();   //信号量减 1

private:
    int sem_id; //信号量标识符
};

//哲学家管程中使用的锁
class Lock{
public:
    Lock(Sema *lock);
    ~Lock();
   void close_lock();
   void open_lock();
private:
   Sema *sema; //锁使用的信号量
};

//哲学家管程中使用的条件变量
class Condition{
public:
    Condition(char *st[], Sema *sm);
    ~Condition();

   void Wait(Lock *lock,int i);  //条件变量阻塞操作
   void Signal(int i); //条件变量唤醒操作
 private:
   Sema *sema; //哲学家信号量
   char **state;//哲学家当前的状态
};

//哲学家管程的定义
class dp{
public:
      dp(int rate);          //管程构造函数
      ~dp();
      void pickup(int i);  //获取筷子
      void putdown(int i); //放下筷子
```

```
        //建立或获取 ipc 信号量的一组函数的原型说明
        int get_ipc_id(char *proc_file,key_t key);
        int set_sem(key_t sem_key,int sem_val,int sem_flag);
        //创建共享内存，放哲学家状态
        char *set_shm(key_t shm_key,int shm_num,int shm_flag);
private:
      int rate ;      //控制执行速度
      Lock *lock;     //控制互斥进入管程的锁
      char *state[5];//5 个哲学家当前的状态
      Condition *self[5]; //控制 5 个哲学家状态的条件变量
};
```

2）在当前目录中建立如下的哲学家就餐程序 dp.cc

```
/*
  *  Filename          :  dp.cc
  *  copyright         : (C) 2006 by 张鸿烈
  * Function           : 哲学家就餐问题的模拟程序
*/

#include "dp.h"

Sema::Sema(int id)
{
   sem_id = id;
}

Sema::~Sema(){ }

/*
 * 信号灯上的 down/up 操作
 * semid:信号灯数组标识符
 * semnum:信号灯数组下标
 * buf:操作信号灯的结构
*/
int Sema::down()
{
  struct sembuf buf;
  buf.sem_op = -1;
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  if((semop(sem_id,&buf,1)) <0) {
    perror("down error ");
    exit(EXIT_FAILURE);
```

```
  }
 return EXIT_SUCCESS;
}

int Sema::up()
{
  Sem_uns arg;
  struct sembuf buf;
  buf.sem_op = 1;
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  if((semop(sem_id,&buf,1)) <0) {
    perror("up error ");
    exit(EXIT_FAILURE);
  }
 return EXIT_SUCCESS;
}

/*
* 用于哲学家管程的互斥执行
*/
Lock::Lock(Sema * s)
{
    sema = s;
}

Lock::~Lock(){  }
//上锁
void Lock::close_lock()
{
   sema->down();
}
//开锁
void Lock::open_lock()
{
   sema->up();
}

//用于哲学家就餐问题的条件变量
Condition::Condition(char *st[] ,Sema *sm){
   state = st;
   sema = sm;
}
```

```
/*
  *  左右邻居不在就餐，条件成立，状态变为就餐
  *  否则睡眠，等待条件成立
*/
void Condition::Wait(Lock *lock,int i)
{
   if((*state[(i+4)%5] != eating) &&
     (*state[i] == hungry) &&
     (*state[(i+1)%5] != eating))
        *state[i] = eating;//拿到筷子，进就餐态
  else{
      cout << "p" << i+1 << ":" << getpid() << " hungry\n";
      lock->open_lock();//开锁
      sema->down();//没拿到，以饥饿态等待
      lock->close_lock();//上锁
      }
}

/*
  *左右邻居不在就餐，则置其状态为就餐，
  *将其从饥俄中唤醒。否则什么也不作。
*/
void Condition::Signal(int i)
{
  if((*state[(i+4)%5] != eating) &&
     (*state[i] == hungry) &&
     (*state[(i+1)%5] != eating))
       {  //可拿到筷子，从饥饿态唤醒进就餐态
       sema->up();
       *state[i] = eating;

       }
}

/*
   * get_ipc_id() 从/proc/sysvipc/文件系统中获取 IPC 的 id 号
   * pfile: 对应/proc/sysvipc/目录中的 IPC 文件分别为
   *        msg-消息队列,sem-信号量,shm-共享内存
   * key:   对应要获取的 IPC 的 id 号的键值
*/
int dp::get_ipc_id(char *proc_file,key_t key)
{
```

```
   #define BUFSZ 256
   FILE *pf;
   int i,j;
   char line[BUFSZ],colum[BUFSZ];

   if((pf = fopen(proc_file,"r")) == NULL){
    perror("Proc file not open");
      exit(EXIT_FAILURE);
    }
   fgets(line, BUFSZ,pf);
   while(!feof(pf)){
     i = j = 0;
     fgets(line, BUFSZ,pf);
     while(line[i] == ' ') i++;
     while(line[i] !=' ') colum[j++] = line[i++];
     colum[j] = '\0';
     if(atoi(colum) != key) continue;
     j=0;
     while(line[i] == ' ') i++;
     while(line[i] !=' ') colum[j++] = line[i++];
     colum[j] = '\0';
     i = atoi(colum);
     fclose(pf);
     return i;
    }
    fclose(pf);
    return -1;
}

/*
*   set_sem 函数建立一个具有 n 个信号灯的信号量
*        如果建立成功，返回 一个信号量的标识符 sem_id
*        输入参数：
*   sem_key 信号量的键值
*   sem_val 信号量中信号灯的个数
*   sem_flag 信号量的存取权限
*/
int dp::set_sem(key_t sem_key,int sem_val,int sem_flg)
{
 int sem_id;
 Sem_uns sem_arg;

  //测试由 sem_key 标识的信号量是否已经建立
```

```
  if((sem_id=get_ipc_id("/proc/sysvipc/sem",sem_key)) < 0 ){
      //semget 新建一个信号灯,其标号返回到 sem_id
      if((sem_id = semget(sem_key,1,sem_flg)) < 0){
         perror("semaphore create error");
         exit(EXIT_FAILURE);
       }
   }
    //设置信号量的初值
    sem_arg.val = sem_val;
    if(semctl(sem_id,0,SETVAL,sem_arg) < 0){
      perror("semaphore set error");
      exit(EXIT_FAILURE);
   }
   return sem_id;
}

/*
*   set_shm 函数建立一个具有 n 个字节 的共享内存区
*        如果建立成功，返回 一个指向该内存区首地址的指针 shm_buf
*        输入参数：
*   shm_key 共享内存的键值
*   shm_val 共享内存字节的长度
*   shm_flag 共享内存的存取权限
*/
char * dp::set_shm(key_t shm_key,int shm_num,int shm_flg)
{
   int i,shm_id;
   char * shm_buf;

   //测试由 shm_key 标识的共享内存区是否已经建立
   if((shm_id=get_ipc_id("/proc/sysvipc/shm",shm_key))<0){
   //shmget 新建 一个长度为 shm_num 字节的共享内存
   if((shm_id= shmget(shm_key,shm_num,shm_flg)) <0){
        perror("shareMemory set error");
        exit(EXIT_FAILURE);
    }
    //shmat 将由 shm_id 标识的共享内存附加给指针 shm_buf
    if((shm_buf=(char *)shmat(shm_id,0,0)) < (char *)0){
        perror("get shareMemory error");
        exit(EXIT_FAILURE);
              }
      for(i=0; i<shm_num; i++) shm_buf[i] = 0; //初始为 0
 }
```

```
//共享内存区已经建立,将由 shm_id 标识的共享内存附加给指针 shm_buf
   if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0) {
       perror("get shareMemory error");
       exit(EXIT_FAILURE);
       }
   return shm_buf;
}

//哲学家就餐问题管程构造函数
dp::dp(int r)
{
  int ipc_flg = IPC_CREAT | 0644;
  int shm_key = 220;
  int shm_num = 1;
  int sem_key = 120;
  int sem_val = 0;
  int sem_id;
  Sema *sema;
  rate = r;

//5 个中同时可以有 2 个在就餐，建立一个初值为 2 的用于锁的信号灯
if((sem_id = set_sem(sem_key++,2,ipc_flg)) < 0){
     perror("Semaphor create error");
     exit(EXIT_FAILURE);
}
sema = new Sema(sem_id);
lock = new Lock(sema);

for(int i=0; i<5; i++){
//为每个哲学家建立一个条件变量和可共享的状态
//初始状态都为思考
if((state[i]=(char *)set_shm(shm_key++,shm_num,ipc_flg)) == NULL){
        perror("Share memory create error");
        exit(EXIT_FAILURE);
 }
 *state[i]=thinking;
//为每个哲学家建立初值为 0 的用于条件变量的信号灯
    if((sem_id = set_sem(sem_key++,sem_val,ipc_flg)) < 0){
        perror("Semaphor create error");
        exit(EXIT_FAILURE);
          }
    sema = new Sema(sem_id);
    self[i] = new Condition(state,sema);
```

```
      }

}

//获取筷子的操作
//如果左右邻居都在就餐，则以饥俄状态阻塞
//否则可以进入就餐状态
void dp::pickup(int i){
   lock->close_lock();//进入管程，上锁
   *state[i]=hungry; //进饥饿态
   self[i]->Wait(lock,i); //测试是否能拿到两只筷子
   cout << "p" << i+1 << ":" << getpid() << " eating\n";
   sleep(rate);//拿到，吃 rate 秒
   lock->open_lock();//离开管程，开锁
}

//放下筷子的操作
//状态改变为思考，如左右邻居有阻塞者则唤醒它
void dp::putdown(int i){
   int j;
   lock->close_lock();//进入管程，上锁
   *state[i]=thinking; //进思考态
   j = (i+4)%5;
   self[j]->Signal(j);//唤醒左邻居
   j = (i+1)%5;
   self[j]->Signal(j);//唤醒右邻居
   lock->open_lock();//离开管程，开锁
   cout << "p" << i+1 << ":" << getpid() << " thinking\n";
   sleep(rate);//思考 rate 秒
}

dp::~dp(){ }

// 哲学家就餐问题并发执行的入口

int main(int argc,char *argv[])
{
  dp *tdp;       //哲学家就餐管程对象的指针
  int pid[5];    //5 个哲学家进程的进程号
  int rate ;

  rate = (argc > 1) ? atoi(argv[1]) : 3 ;
```

```
tdp = new dp(rate); //建立一个哲学家就餐的管程对象

pid[0]=fork(); //建立第一个哲学家进程
if(pid[0]<0){ perror("p1 create error"); exit(EXIT_FAILURE);}
else if(pid[0]==0){
//利用管程模拟第一个哲学家就餐的过程
  while(1){
     tdp->pickup(0);//拿起筷子
     tdp->putdown(0);//放下筷子
  }
}
pid[1]=fork();    //建立第二个哲学家进程
if(pid[1]<0){ perror("p2 create error"); exit(EXIT_FAILURE);}
else if(pid[1]==0){
  //利用管程模拟第二个哲学家就餐的过程
  while(1){
     tdp->pickup(1);//拿起筷子
     tdp->putdown(1);//放下筷子
  }
}
pid[2]=fork(); //建立第三个哲学家进程
if(pid[2]<0) { perror("p3 create error"); exit(EXIT_FAILURE);}
else if(pid[2]==0){
      //利用管程模拟第三个哲学家就餐的过程
    while(1){
       tdp->pickup(2);//拿起筷子
       tdp->putdown(2);//放下筷子
     }
 }
 pid[3]=fork(); //建立第四个哲学家进程
 if(pid[3]<0) { perror("p4 create error"); exit(EXIT_FAILURE);}
 else if(pid[3]==0){
      //利用管程模拟第四个哲学家就餐的过程
    while(1){
       tdp->pickup(3);//拿起筷子
       tdp->putdown(3);//放下筷子
    }
 }
 pid[4]=fork(); //建立第五个哲学家进程
 if(pid[4]<0) { perror("p5 create error"); exit(EXIT_FAILURE);}
 else if(pid[4]==0){
   //利用管程模拟第五个哲学家就餐的过程
    while(1){
```

```
            tdp->pickup(4);//拿起筷子
            tdp->putdown(4);//放下筷子
        }
    }
    return 0;
}
```

3）在新建文件夹中建立以下 Makefile 文件

```
head = dp.h
srcs = dp.cc
objs = dp.o
opts = -w -g -c
all:   dp
dp:     $(objs)
        g++ $(objs) -o dp
dp.o:   $(srcs) $(head)
        g++ $(opts)  $(srcs)
clean:
         rm dp *.o
```

4）在新建文件夹中执行 make 命令编译连接生成可执行的哲学家就餐程序

```
$make
   g++   -w -g -c   dp.cc
   g++ dp.o   -o dp
```

5) 执行的哲学家就餐程序 dp

```
$./dp 1
p1:4524 eating
p2:4525 hungry
p3:4526 eating
p4:4527 hungry
p5:4528 hungry
p1:4524 thinking
p5:4528 eating
p3:4526 thinking
p2:4525 eating
p1:4524 hungry
p5:4528 thinking
p4:4527 eating
p3:4526 hungry
p1:4524 eating
p2:4525 thinking
p5:4528 hungry
……
```

可以看到 5 个哲学家进程在 3 中状态中不断的轮流变换，且连续的 5 个输出中不应有多于 2 个的状态为 eating，同一进程号不应有两个连续的输出。您可用不同的执行速率长时间的让它们执行，观察是否会发生死锁或饥饿现象。如果始终没有产生死锁和饥饿现象，可用 kill 按其各自的进程号终止它们的执行。

6）请修改以上 dp.cc 程序，制造出几种不同的的死锁现象和饥饿现象，记录并分析各种死锁、饥饿现象和产生死锁、饥饿现象的原因。

6.4 独立实验

在两个城市南北方向之间存在一条铁路，多列火车可以分别从两个城市的车站排队等待进入车道向对方城市行驶，该铁路在同一时间，只能允许在同一方向上行车，如果同时有相向的火车行驶将会撞车。请模拟实现两个方向行车，而不会出现撞车或长时间等待的情况。您能构造一个管程来解决这个问题吗？

参考解法分析：

管程-Monitor

管程是一种高级抽象数据类型，它支持在它的函数中隐含互斥操作。结合条件变量和其他一些低级通信原语，管程可以解决许多仅用低级原语不能解决的同步问题。利用管程可以提供一个不会发生死锁或饥饿现象的对象;哲学家就餐问题和 Java 语言中的 synchronized 对象都是很好的管程的例子.

管程封装了并发进程或线程要互斥执行的函数。为了让这些并发进程或线程在管程内互斥的执行，进入管程的进/线程必须获取到管程锁或二值信号量

条件变量 Condition Variables

条件变量提供了一种对管程内并发协作进程的同步机制。如果没有条件变量，管程就不会有很有用。多数同步问题要求在管程中说明条件变量。条件变量代表了管程中一些并发进程或线程可能要等待的条件。一个条件变量管理着管程内的一个等待队列。如果管程内某个进程或线程发现其执行条件为假，则该进程或线程就会被条件变量挂入管程内等待该条件的队列。如果管程内另外的进程或线程满足了这个条件，则它会通过条件变量再次唤醒等待该条件的进程或线程，从而避免了死锁的产生。所以，一个条件变量 C 应具有两种操作 C.wait()和 C.signal()。

当管程内同时出现唤醒者和被唤醒者时，由于要求管程内的进程或线程必须互斥执行，因此就出现了两种样式的条件变量：

Mesa Style(signal-and-continue)： 唤醒者进程或线程继续执行，被唤醒者进程或线程等到唤醒者进程或线程阻塞或离开管程后再执行。

Hoare Style(signal-and-wait)： 被唤醒者进程或线程立即执行，唤醒者进程或线程阻塞，直道被唤醒者阻塞或离开管程后再执行。

实验 6 单行道(过桥)问题可以通过管程很好的解决。可以把单行道/桥封装为一个管程类，桥上通过的车辆是进入管程的进/线程，可以通过创建多个车辆进/线程并随机产生它们的行进方向，并指定桥上可同时行驶的车辆的个数来模拟该问题的各种现场随机情况。一个正确的实验结果应能实现在各种随机现场情况下车辆进程不会逆向上桥(死锁)，也不会使车少方向上的车辆无机会上桥(饥饿).

参考解法伪代码：

以下是一个单行道管程类及其方法和属性的大致描述：
定义一个单行道管程类：

```
class OneWay {
      public:
        OneWay();
        ~OneWay();
        void Arrive(int direc);     // 车辆准备上单行道，direc 为行车方向
        void Cross(int direc);       // 车辆正在单行道上
    void Quit(int direc);        // 车辆通过了单行道
  private:
         int rate;              //车速
         int *maxCars;        //最大同向车数
         int *numCars;             //当前正在通过的车辆数
         int *currentDirec;        //当前通过的车辆的方向
         Condition *OneWayFull;   //通过单行道的条件变量
         Lock *lock;                    //单行道管程锁
    };
```

定义一个进入管程的信号量 Sema 类和锁 Lock 类(可参考实验六的示例程序).
定义一个单行道管程的条件变量类：

```
class Condition {
      public:
        Condition(Sema *sema1 , Sema *sema2);
        ~Condition();
    void Wait(Lock *conditionLock,int direc);   //过路条件不足时阻塞
    void Signal( int direc);                    //唤醒相反方向阻塞车辆
      private:
         Sema* queue1;  // 一个方向阻塞队列
         Sema* queue2;  // 另一方向阻塞队列
         Lock* lock;    // 进入管程时获取的锁
};
```

Mesa 型条件变量的 Wait 和 Signal 方法：(也可设计成 Haoro 样式)

```
    Condition::Wait( Lock *conditionLock,int direct)
    {
       保存当前条件锁;
        释放锁;
        进入所在方向等待队列睡眠;
        被唤醒后重新获取锁;
    }
    Condition::Signal( Lock *conditionLock)
```

```
{
    唤醒相反方向队列等待者
}
```

单行道管程的 Arrive 和 Quit 方法：

```
OneWay::Arrive(int direc)
{
    获取管程锁；

    如果当前方向不是我的方向或者单行道上有车且车辆数大于等于上限数
        在条件变量上等待；
    单行道上车辆数加 1；
    当前方向为我的方向；

    释放管程锁；
}

OneWay::Quit(int direc)
{
    获取管程锁；
    单行道上车辆数减 1；
    车辆数为 0
    唤醒相反方向队列中的等待者
    释放管程锁
}
```

主程序

```
main()
 {
    建立单行道管程；
    建立多个行车子进程(最好不少于 5 个)，每个随机设定一个行车方向；
   每个子进程作：
    while(1) {
        单行道管程->Arrive(direc);
        单行道管程->Cross(direc);
        单行道管程->Exit(direc);
    }
}
```

程序执行结果的测试和分析：

1、单行道最多允许一辆车行驶，不同方向同时到达五辆车。

```
$ ./oneway 5 1

3429 号车向东进入单行道
3429 号车向东通过单行道,道上车数:1
3430 号车向东等待单行道
3431 号车向西等待单行道
3428 号车向西等待单行道
3427 号车向西等待单行道
3429 号车向东离开单行道
3431 号车向西进入单行道
3431 号车向西通过单行道,道上车数:1
3428 号车向西等待单行道
3427 号车向西等待单行道
3431 号车向西离开单行道
3430 号车向东进入单行道
3430 号车向东通过单行道,道上车数:1
3428 号车向西等待单行道
3430 号车向东离开单行道
3427 号车向西进入单行道
3427 号车向西通过单行道,道上车数:1
3428 号车向西等待单行道
3427 号车向西离开单行道
3428 号车向西进入单行道
3428 号车向西通过单行道,道上车数:1
3428 号车向西离开单行道
```

两辆向东三辆向西，正常通过。

2、单行道最多允许两辆车同向行驶，不同方向同时到达七辆车。

```
./oneway 7 2

3520 号车向东进入单行道
3520 号车向东通过单行道,道上车数:1
3522 号车向东进入单行道
3522 号车向东通过单行道,道上车数:2
3523 号车向东等待单行道
3521 号车向东等待单行道
3524 号车向西等待单行道
3525 号车向西等待单行道
3526 号车向东等待单行道
```

```
3520 号车向东离开单行道
3522 号车向东离开单行道
3524 号车向西进入单行道
3524 号车向西通过单行道,道上车数:1
3525 号车向西进入单行道
3525 号车向西通过单行道,道上车数:2
3524 号车向西离开单行道
3525 号车向西离开单行道
3523 号车向东进入单行道
3523 号车向东通过单行道,道上车数:1
3521 号车向东进入单行道
3521 号车向东通过单行道,道上车数:2
3526 号车向东等待单行道
3523 号车向东离开单行道
3521 号车向东离开单行道
3526 号车向东进入单行道
3526 号车向东通过单行道,道上车数:1
3526 号车向东离开单行道
```

五辆向东两辆向西，正常通过。

3、单行道最多允许四辆车同向行驶，不同方向同时到达十辆车。

```
$ ./oneway 10 4
```

```
3704 号车向西进入单行道
3704 号车向西通过单行道,道上车数:1
3706 号车向东等待单行道
3708 号车向东等待单行道
3705 号车向西进入单行道
3705 号车向西通过单行道,道上车数:2
3707 号车向西进入单行道
3707 号车向西通过单行道,道上车数:3
3709 号车向西进入单行道
3709 号车向西通过单行道,道上车数:4
3710 号车向东等待单行道
3711 号车向东等待单行道
3712 号车向东等待单行道
3713 号车向东等待单行道
3704 号车向西离开单行道
3705 号车向西离开单行道
3707 号车向西离开单行道
3709 号车向西离开单行道
```

```
3706 号车向东进入单行道
3706 号车向东通过单行道,道上车数:1
3708 号车向东进入单行道
3708 号车向东通过单行道,道上车数:2
3710 号车向东进入单行道
3710 号车向东通过单行道,道上车数:3
3711 号车向东进入单行道
3711 号车向东通过单行道,道上车数:4
3712 号车向东等待单行道
3713 号车向东等待单行道
3706 号车向东离开单行道
3708 号车向东离开单行道
3710 号车向东离开单行道
3711 号车向东离开单行道
3712 号车向东进入单行道
3712 号车向东通过单行道,道上车数:1
3713 号车向东进入单行道
3713 号车向东通过单行道,道上车数:2
3712 号车向东离开单行道
3713 号车向东离开单行道
$
```

六辆向东四辆向西，正常通过。

4、.....

6.5 实验要求

总结和分析示例实验和独立实验中观察到的调试和运行信息。分析示例实验是否真正模拟了哲学家就餐问题？为什么示例程序不会产生死锁？为什么会出现进程死锁和饥饿现象？怎样利用实验造成和表现死锁和饥饿现象？管程能避免死锁和饥饿的机理是什么？您对于管程概念有哪些新的理解和认识？条件变量和信号量有何不同？为什么在管程中要使用条件变量而不直接使用信号量来达到进程同步的目的？示例实验中构造的管程中的条件变量是一种什么样式的？其中的锁起了什么样的作用？你的独立实验程序是怎样解决单行道问题的？您是怎样构造管程对象的？根据实验程序、调试过程和结果分析写出实验报告。

实验七、内存页面置换算法实验

7.1 实验目的

加深对于存储管理的了解，掌握虚拟存储器的实现原理；观察和了解重要的页面置换算法和置换过程。练习模拟算法的编程技巧，锻炼分析试验数据的能力。

7.2 实验说明

1. 示例实验程序中模拟两种置换算法：LRU 算法和 FIFO 算法
2. 能对两种算法给定任意序列不同的页面引用串和任意帧实内存块数的组合测试，显示页置换的过程。
3. 能统计和报告不同置换算法情况下依次淘汰的页号、缺页次数（页错误数）和缺页率。比较两种置换算法在给定条件下的优劣。
4. 为了能方便的扩充页面置换算法，更好的描述置换过程，示例实验程序采用了 C++语言用 Replace 类描述了置换算法及其属性。

7.3 示例实验

1）在新建文件夹中建立以下 vmrp.h 文件：

```
/*
   *   Filename              :   vmrp.h
   *   copyright             : (C) 2006 by  张鸿烈
    * Function               :  声明虚拟内存页置换类
*/
#include <iostream>
#include <iomanip.h>
#include <malloc.h>

class Replace{
public:
      Replace();
      ~Replace();
     void InitSpace(char * MethodName); //初始化页号记录
      void Report(void); //  报告算法执行情况
      void Fifo(void); //先进先出算法
      void Lru(void);   //最近最旧未用算法
      void Clock(void); //时钟(二次机会）置换算法
      void Eclock(void); //增强二次机会置换算法
     void Lfu(void); //最不经常使用置换算法
     void Mfu(void);   //最经常使用置换算法
private:
      int * ReferencePage ; //存放要访问到的页号
```

```
    int * EliminatePage ; //存放淘汰页号
    int * PageFrames ;      //存放当前正在实存中的页号
    int PageNumber;          //访问页数
    int FrameNumber;         //实存帧数
    int FaultNumber;         //失败页数
};
```

2）在新建文件夹中建立以下 vmrp.cc 文件：

```
/*
   *   Filename              :   vmrp.cc
   *   copyright             : (C) 2006 by  张鸿烈
    * Function               :  模拟虚拟内存页置换算法的程序
*/
#include "vmrp.h"

Replace::Replace(){
 int i;
  //设定总得访问页数,并分配相应的引用页号和淘汰页号记录数组空间
  cout << "Please input page numbers :" ;
  cin >> PageNumber;
  ReferencePage = new int[sizeof(int) * PageNumber];
  EliminatePage = new int[sizeof(int) * PageNumber];

   //输入引用页号序列(页面走向),初始化引用页数组
  cout << "Please input reference page string :";
  for (i = 0; i < PageNumber; i++)
 cin >> ReferencePage[i];//引用页暂存引用数组

  //设定内存实页数(帧数),并分配相应的实页号记录数组空间(页号栈)
  cout << "Please input page frames :";
  cin >> FrameNumber;
  PageFrames = new int[sizeof(int) * FrameNumber];
}

Replace::~Replace(){

}

void Replace::InitSpace(char * MethodName)
{
  int i;
  cout << endl << MethodName << endl;
  FaultNumber=0;
  //引用还未开始,-1 表示无引用页
  for (i = 0; i < PageNumber; i++)
      EliminatePage[i] = -1;
  for(i = 0; i < FrameNumber; i++)
```

```
        PageFrames[i] = -1;
}

//分析统计选择的算法对于当前输入的页面走向的性能
void Replace::Report(void){
  //报告淘汰页顺序
  cout << endl << "Eliminate page:";
  for(int i=0; EliminatePage[i]!=-1; i++)    cout << EliminatePage[i] << " ";
  //报告缺页数和缺页率
  cout << endl << "Number of page faults = " << FaultNumber << endl;
  cout << setw(6) << setprecision(3) ;
  cout << "Rate of page faults = " << 100*(float)FaultNumber/(float)PageNumber <<
"%" <<endl;
}

//最近最旧未用置换算法
void Replace::Lru(void)
{
  int i,j,k,l,next;

  InitSpace("LRU");
  //循环装入引用页
  for(k=0,l=0; k < PageNumber; k++){
       next=ReferencePage[k];
       //检测引用页当前是否已在实存
       for (i=0; i<FrameNumber; i++){
          if(next == PageFrames[i]){
           //引用页已在实存将其调整到页记录栈顶
              next= PageFrames[i];
              for(j=i;j>0;j--)    PageFrames[j] =    PageFrames[j-1];
              PageFrames[0]=next;
              break;
            }
       }

       if(PageFrames[0] == next){
          //如果引用页已放栈顶，则为不缺页，报告当前内存页号
          for(j=0; j<FrameNumber; j++)
            if(PageFrames[j]>=0) cout << PageFrames[j] << " ";
          cout << endl;
          continue;   //继续装入下一页
       }
       else
           // 如果引用页还未放栈顶，则为缺页，缺页数加 1
           FaultNumber++;
       //栈底页号记入淘汰页数组中
       EliminatePage[l] = PageFrames[FrameNumber-1];
```

```
        //向下压栈
        for(j=FrameNumber-1;j>0;j--) PageFrames[j]= PageFrames[j-1];
        PageFrames[0]=next; //引用页放栈顶
        //报告当前实存中页号
        for(j=0; j<FrameNumber; j++)
            if(PageFrames[j]>=0) cout << PageFrames[j] << " ";
        //报告当前淘汰的页号
        if(EliminatePage[l]>=0)
            cout << "->" << EliminatePage[l++] << endl;
        else
         cout << endl;

   }
   //分析统计选择的算法对于当前引用的页面走向的性能
   Report();
}

//先进先出置换算法
void Replace::Fifo(void){
    int i,j,k,l,next;

    InitSpace("FIFO");
   //循环装入引用页
    for(k=0,j=l=0; k < PageNumber; k++){
        next=ReferencePage[k];
        //如果引用页已在实存中，报告实存页号
       for (i=0; i<FrameNumber; i++) if(next==PageFrames[i]) break;
        if (i<FrameNumber){
             for(i=0; i<FrameNumber; i++) cout << PageFrames[i] << " ";
          cout << endl;
          continue;    //  继续引用下一页
        }
        //引用页不在实存中，缺页数加 1
        FaultNumber++;
        EliminatePage[l]= PageFrames[j]; //最先入页号记入淘汰页数组
        PageFrames[j]=next;    //引用页号放最先入页号处
        j = (j+1)%FrameNumber; //最先入页号循环下移
        //报告当前实存页号和淘汰页号
        for(i=0; i<FrameNumber; i++)
             if(PageFrames[i]>=0) cout << PageFrames[i] << " ";
        if(EliminatePage[l]>=0)
             cout << "->" << EliminatePage[l++] << endl;
        else
          cout << endl;
   }
   //分析统计选择的算法对于当前引用的页面走向的性能
   Report();
```

```
}

//未实现的其他页置换算法入口
void Replace::Clock(void)
{

}

void Replace::Eclock (void)
{

}


void Replace::Lfu(void)
{

}

void Replace::Mfu(void)
{

}


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

  Replace * vmpr = new Replace();

        vmpr->Lru();
          vmpr->Fifo();

  return 0;
}
```

3）在新建文件夹中建立以下 Makefile 文件

```
head = vmrp.h
srcs = vmrp.cc
objs = vmrp.o
opts = -w -g -c
all:  vmrp
vmrp:     $(objs)
          g++   $(objs) -o vmrp
vmrp.o:  $(srcs) $(head)
          g++ $(opts) $(srcs)
clean:
          rm vmrp *.o
```

4) 执行 make 命令编译连接，生成可执行文件 vmpr

$ g*make*

```
g++ -g -c vmrp.cc vmrp.h
g++    vmrp.o -o vmrp
```

5) 执行 vmpr 命令，输入引用页数为 12，引用串为 Belady 串，内存页帧数为 3

$ ./vmpr

```
Please input reference page numbers :12
Please input reference page string :1 2 3 4 1 2 5 1 2 3 4 5
Please input page frames :3

FIFO
1
1 2
1 2 3
4 2 3 ->1
4 1 3 ->2
4 1 2 ->3
5 1 2 ->4
5 1 2
5 1 2
5 3 2 ->1
5 3 4 ->2
5 3 4

Eliminate page:1 2 3 4 1 2
Number of page faults = 9
Rate of page faults = 75%

LRU
1
2 1
3 2 1
4 3 2 ->1
1 4 3 ->2
2 1 4 ->3
5 2 1 ->4
1 5 2
2 1 5
3 2 1 ->5
4 3 2 ->1
5 4 3 ->2

Eliminate page:1 2 3 4 5 1 2
Number of page faults = 10
Rate of page faults = 83.3%
```

以上输出报告了 FIFO 和 LRU 两种算法的页置换情况。其中每一行数字为当前实存中的页号，->右边的数字表示当前被淘汰的页号。每种算法最后 3 行输出为：依次淘汰页号，缺页数，页出错率。

6) 再次执行 vmpr 命令，仍然输入 Belady 串，仅将页帧数改为 4

```
 $ ./vmpr
Please input reference page numbers :12
Please input reference page string :1 2 3 4 1 2 5 1 2 3 4 5
Please input page frames :4

FIFO
1
1 2
1 2 3
1 2 3 4
1 2 3 4
1 2 3 4
5 2 3 4 ->1
5 1 3 4 ->2
5 1 2 4 ->3
5 1 2 3 ->4
4 1 2 3 ->5
4 5 2 3 ->1

Eliminate page:1 2 3 4 5 1
Number of page faults = 10
Rate of page faults = 83.3%

LRU
1
2 1
3 2 1
4 3 2 1
1 4 3 2
2 1 4 3
5 2 1 4 ->3
1 5 2 4
2 1 5 4
3 2 1 5 ->4
4 3 2 1 ->5
5 4 3 2 ->1

Eliminate page:3 4 5 1
Number of page faults = 8
Rate of page faults = 66.7%
```

从以上输出中可以看出 FIFO 置换算法的 Belady 异常现象，即当在相同的引用串下内存页帧数从 3 帧增加到 4 帧，页出错率反而从 75%增加到了 83.3%。而在相同的

情况下 LUR 置换算法无此异常现象。

7.4 独立实验

请在以上示例实验程序中补充“增强二次机会”等置换算法的模拟程序。输入不同的内存页面引用串和实存帧数，观察并分析其页面置换效果和性能，并将其与 LRU 和 FIFO 算法进行比较。改进以上示例实验程序，使之能够随机的产生内存页面引用串，以便能动态的观测各种置换算法的性能。

7.5 实验要求

1. 说明您做了哪些不同的引用串在不同实存帧中的测试，发现了哪些现象？
2. 选择一些典型的现象作出不同算法中帧数与缺页数的曲线图。
3. 说明您的程序是怎样模拟“增强二次机会”等置换算法的？
4. 综合分析实验结果中各种页面置换算法各适应于怎样的页面引用串和和内存帧数。
5. 根据实验程序、调试过程和结果分析写出实验报告。

实验八、磁盘移臂调度算法实验

8.1 实验目的

加深对于操作系统设备管理技术的了解，体验磁盘移臂调度算法的重要性；掌握几种重要的磁盘移臂调度算法，练习模拟算法的编程技巧，锻炼研究分析试验数据的能力。

8.2 实验说明

1. 示例实验程序中模拟两种磁盘移臂调度算法：SSTF 算法和 SCAN 算法
2. 能对两种算法给定任意序列不同的磁盘请求序列，显示响应磁盘请求的过程。
3. 能统计和报告不同算法情况下响应请求的顺序、移臂的总量。比较两种算法在给定条件下的优劣。
4. 为了能方便的扩充磁盘移臂调度算法，更好的描述磁盘移臂调度过程，示例实验程序采用了 C++语言用 DiskArm 类描述了磁盘移臂调度算法及其属性。

8.3 示例实验

1）在新建文件夹中建立以下 dask.h 文件：

```
/*
   *    Filename              :    dask.h
   *    copyright             : (C) 2006 by  张鸿烈
    * Function                :  声明磁盘移臂调度类
*/
#include <iostream>
#include <iomanip.h>
#include <malloc.h>

class DiskArm{
public:
      DiskArm();
      ~DiskArm();
     void InitSpace(char * MethodName); //初始化寻道记录
      void Report(void); //  报告算法执行情况
      void Fcfs(void); //先来先服务算法
      void Sstf(void);    //最短寻道时间优先算法
      void Scan(void); //电梯调度算法
      void CScan(void); //均匀电梯调度算法
     void Look(void); //LOOK 调度算法
private:
      int *Request ;         //磁盘请求道号
```

```
    int *Cylinder;        //工作柱面道号号
    int RequestNumber;          //磁盘请求数
    int CurrentCylinder;      //当前道号
   int SeekDirection;         //磁头方向
    int SeekNumber;          //移臂总数
   int SeekChang;             //磁头调头数
};
```

2) 在新建文件夹中建立以下 dask.cc 文件：

```
/*
   *    Filename              :    dask.cc
   *    copyright             : (C) 2006 by  张鸿烈
    * Function               :  磁盘移臂调度算法
*/

#include "dask.h"
DiskArm::DiskArm(){
 int i;
   //输入当前道号
   cout << "Please input Current cylinder :" ;
   cin >> CurrentCylinder;
   //磁头方向，输入 0 表示向小道号移动，1 表示向大道号移动
   cout << "Please input Current Direction (0/1) :" ;
   cin >> SeekDirection;
   //输入磁盘请求数，请求道号
   cout << "Please input Request Numbers :" ;
   cin >> RequestNumber;
   cout << "Please input Request cylinder string :";
   Request = new int[sizeof(int) * RequestNumber];
   Cylinder = new int[sizeof(int) * RequestNumber];
   for (i = 0; i < RequestNumber; i++)
       cin >> Request[i];
 }

DiskArm::~DiskArm(){

}
//初始化道号，寻道记录
void DiskArm::InitSpace(char * MethodName)
{
   int i;
   cout << endl <<    MethodName << endl;
   SeekNumber = 0;
   SeekChang = 0;
   for (i = 0; i < RequestNumber; i++)
          Cylinder[i] = Request[i];
   }
```

```
// 统计报告算法执行情况
void DiskArm::Report(void){
    cout << endl;
    cout <<   "Seek Number: " << SeekNumber << endl;
    cout <<   "Chang Direction: " << SeekChang << endl << endl;
}

//先来先服务算法
void DiskArm::Fcfs(void)
{
      int   Current = CurrentCylinder;
      int   Direction = SeekDirection;
      InitSpace("FCFS");

      cout << Current;
      for(int i=0;   i<RequestNumber; i++){
          if(((Cylinder[i] >= Current) && !Direction)
            ||((Cylinder[i] <   Current) &&   Direction)){
              //需要调头
                SeekChang++;   //调头数加 1
              Direction = !Direction ; //改变方向标志
              //报告当前响应的道号
              cout   <<   endl << Current << " -> " << Cylinder[i];
          }
          else //不需调头，报告当前响应的道号
              cout << " -> " << Cylinder[i] ;
          //累计寻道数，响应过的道号变为当前道号
          SeekNumber += abs(Current -Cylinder[i]);
          Current = Cylinder[i];
    }
     //报告磁盘移臂调度的情况
    Report();
}

//最短寻道时间优先算法
void DiskArm::Sstf(void)
{
      int   Shortest;
      int   Distance = 999999 ;
      int   Direction = SeekDirection;
      int   Current = CurrentCylinder;
      InitSpace("SSTF");
      cout << Current;
      for(int i=0; i<RequestNumber; i++){
          //查找当前最近道号
          for(int j=0; j<RequestNumber; j++){
```

```
            if(Cylinder[j] == -1) continue;   //-1 表示已经响应过了
            if(Distance    > abs(Current-Cylinder[j])){
                //到下一道号比当前距离近，下一道号为当前距离
                  Distance    = abs(Current-Cylinder[j]);
                  Shortest = j;
              }
           }
        if((( Cylinder[Shortest] >= Current) && !Direction)
           ||((    Cylinder[Shortest] < CurrentCylinder) && Direction)){
             //需要调头
                SeekChang++;   //调头数加 1
              Direction = !Direction ;   //改变方向标志
              //报告当前响应的道号
              cout    <<    endl <<    Current << " -> " <<    Cylinder[Shortest];
         }
         else //不需调头，报告当前响应的道号
             cout << " -> " << Cylinder[Shortest] ;

        //累计寻道数，响应过的道号变为当前道号
         SeekNumber += abs(Current -Cylinder[Shortest]);
        Current = Cylinder[Shortest];
        //恢复最近距离，销去响应过的道号
        Distance = 999999;
        Cylinder[Shortest] = -1;
       }

     Report();

}

//电梯调度算法
void DiskArm::Scan(void){

}

//均匀电梯调度算法
void DiskArm::CScan(void){

}

//LOOK 调度算法
void DiskArm::Look(void)
{

}
//程序启动入口
int main(int argc,char *argv[]){
```

```
   //建立磁盘移臂调度类
   DiskArm *dask = new DiskArm();
   //比较和分析 FCFS 和 SSTF 两种调度算法的性能
   dask->Fcfs();
   dask->Sstf();
}
```

3）在新建文件夹中建立以下 Makefile 文件

```
head = dask.h
srcs = dask.cc
objs = dask.o
opts = -w -g -c
all:        dask
dask:       $(objs)
            g++    $(objs) -o dask
dask.o:     $(srcs) $(head)
            g++ $(opts) $(srcs)
clean:
            rm dask *.o
```

4) 执行 make 命令编译连接，生成可执行文件 dask

```
   $ gmake
   g++    -w -g -c dask.cc
   g++    dask.o -o dask
```

5) **执行 dask 命令，输入当前道号，当前寻道方向，当前请求寻道数，当前请求寻道的道号串：**

```
   ./dask
   Please input Current cylinder : 53
   Please input Current Direction (0/1) : 0
   Please input Request Numbers : 8
   Please input Request cylinder string : 98 183 37 122 14 124 65 67

   FCFS
   53
   53 -> 98 -> 183
   183 -> 37
   37 -> 122
   122 -> 14
   14 -> 124
   124 -> 65
   65 -> 67
   Seek Number: 640
   Chang Direction: 7
   SSTF
   53
   53 -> 65 -> 67
   67 -> 37 -> 14
   14 -> 98 -> 122 -> 124 -> 183
```

Seek Number: 236
Chang Direction: 3
$

可以看到以上程序的执行演示了 FCFS 和 SSTF 两种磁盘移臂调度算法响应磁盘请求的次序(其中每换一行表示磁头发生调头)。统计出了这两种算法的调度性能，从中看出在以上磁盘柱面请求序列下 SSTF 调度算法所产生的磁头移动为 236 柱面，约为 FCFS 调度算法所产生的磁头移动数量 640 柱面的三分之一稍多一点。磁头调头数 3 次也比 FCFS 调度算法的 7 次少了 4 次。因此对于以上磁盘柱面请求序列,SSTF 调度算法将比 FCFS 调度算法大大提高了磁盘的响应速度。

8.4 独立实验

请在以上示例实验程序中补充 SCAN，C-SCAN，LOOK 磁盘移臂调度算法的模拟程序。输入不同的磁盘柱面请求序列，观察和分析其调度效果和性能，并将其与 FCFS 和 SSTF 算法进行比较。改进以上示例实验程序，使之能够随机的产生磁盘柱面请求序列，以便能动态的观测各种调度算法的性能。

8.5 实验要求

1. 说明您做了哪些磁盘请求序列的测试，发现了哪些现象？
2. 选择一些典型磁盘请求序列的响应结果，画出不同算法中的寻道曲线图。
3. 说明您的程序是怎样模拟 SCAN 等算法的？
4. 综合分析实验结果中各种算法各适应于怎样的磁盘柱面寻道请求情况。
5. 根据实验程序、调试过程和结果分析写出实验报告

实验九、文件系统接口实验

9.1 实验目的

通过文件系统调用的编程和调试，加深对于文件系统的更深入的了解，体验文件系统接口机制的功能，掌握基于文件描述符的主要 I/O 操作技术。通过本试验的调试可更深入的理解教材中有关文件的分类方法，文件的逻辑结构、文件的物理结构、文件的存取控制、文件的保护方式等文件管理功能，以及 Unix/Linux 系统中一切皆文件的理念。

9.2 实验说明

1）打开文件与文件描述符

文件描述符是一个打开文件记录表的索引值，它对应一个正整数。它是许多低级 I/O 操作和网路操作的基本编程接口。

例如，每个进程启动时都会自动打开 3 个文件：标准输入、标准输出、标准错误输出。这 3 个文件对应的文件描述符的默认值为 0，1，2。为记忆方便同时也赋予它们 3 个标识名 stdin、 stdout、 stderr。有了文件描述符对于打开文件的所有 I/O 操作都可以方便的使用文件描述符来完成。

除了标准输入、标准输出、标准错误输出 3 个文件的描述符由系统自动给出之外，进程要打开的其他文件必须由进程通过打开文件系统调用 open 获得它的文件描述符后才能进行读、写、定位等操作。操作完成后应通过关闭文件操作 close 将文件描述符对应的缓冲区刷新并清理掉打开文件记录。

open 系统调用的语法：

```
#include <sys/type.h>
#include <sys.stat.h>
#include <fcntl.h>
int open(const char *pathname,int flags)
```

pathname	要打开或要创建的文件名串指针	
flags	要打开或要创建的文件的访问方式，有以下几种标志：	
	O_RDONLY	只读方式打开
	O_WRONLY	只写方式打开
	O_RDWR	读写方式打开
	O_CREAT	文件不存在则建立
	O_EXCL	与 O_CREAT 连用，如果文件已存在，则使打开失败
	O_NOCTTY	如果打开文件为终端，不让其成为打开进程的终端
	O_TRUNC	如果文件存在，则将其长度截为 0
	O_APPEND	追加方式打开（文件写指针置文件尾）
	O_NONBLOCK	以非阻塞方式打开。如读操作阻塞将只会读出 0 字节
	O_NODELY	同 O_NONBLOCK
	O_SYNC	在数据物理的写入设备后才返回

mode　　文件访问权限的位掩码和对应的宏，

S_IRWXU	00700	文件主可读可写可执行
S_IRUSR	00400	文件主可读
S_IWUSR	00200	文件主可写
S_IXUSR	00100	文件主可执行
S_IRWXG	00070	同组者可读可写可执行
S_IRGRP	00040	同组者可读
S_IWGRP	00020	同组者可写
S_IXGRP	00010	同组者主可执行
S_IRWXO	00007	其他用户可读可写可执行
S_IROTH	00004	其他用户可读
S_IWOTH	00002	其他用户可写
S_IXOTH	00001	其他用户可执行

open 调用成功后返回一个文件描述符。不成功返回-1，出错号放系统变量 errno 中。

close 系统调用的语法为：

```
#include <unistd.h>
int close(int fd)
```

fd　　由 open 返回的文件描述符

2）使用文件描述符读 read、写 write 文件操作的系统调用语法为：

```
#include <unistd.h>
ssize_t read(int fd,const void *buf,size_t count);
ssize_t write(int fd,const void *buf,size_t count);
```

fd　　由 open 返回的文件描述符

buf　　读写数据缓冲区指针

count　　指定要读写的字节数

read/write 调用成功返回实际读写的字节数 ,不成功返回-1，出错号放 errno。

3）使用文件描述符定位文件读写指针操作的 lseek 系统调用语法为：

```
#include <sys/type.h>
#include <unistd.h>
off_t lseek(int fd,off_t offset,int whence);
```

fd　　由 open 返回的文件描述符

offset　　文件读写指针要移动的相对字节偏量，可以为负值。

Whence　　文件读写指针要移动的相对位置，可以为：

SEEK_SET	从文件头移动 offset 字节
SEEK_CUR	从当前读写位置移动 offset 字节
SEEK_END	从文件尾移动 offset 字节

lseek 调用成功返回新文件读写指针位置 ,不成功返回-1，出错号放 errno。

4) **使用文件描述符获得文件控制信息操作的 fstat 系统调用语法为：**

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

#include <unistd.h>
int fstat(int fd,struct stat *buf);
fd　　由 open 返回的文件描述符
buf　　stat 结构指针
stat 结构定义了一个打开文件的控制信息结构。该结构的字段说明如下
struct stat {

```
dev_t     st_dev;     // 设备文件名
ino_t     st_ino;     // 文件的 I 节点编号
mode_t    st_mode;    // 文件的保护字段
nlink_t   st_nlink;   // 文件链接数
uid_t     st_uid;     // 文件所有者编号
gid_t     st_gid;     // 文件所有者组号
dev_t     st_rdev;    // 设备类型
off_t     st_size;    // 文件的字节总长度
blksize_t st_blksize; // 文件系统 I/O 块长
blkcnt_t  st_blocks;  // 文件已分配的块数
time_t    st_atime;   // 文件最后访问的时间
time_t    st_mtime;   // 文件最后修改的时间
time_t    st_ctime;   // 文件最后改变的时间
};
```

fstat 调用成功会把该文件的控制信息保存到 buf 所指向的数据结构中，并返回 0 值；调用失败返回-1，出错号放 errno。

9.3 示例实验

示例实验程序通过打开用户指定的输入文件，按输入参数定位读写位置并从指定位置读出指定长度的内容复制到指定的输出文件中（文件内容部分拷贝），之后又显示了输出文件的文件状态信息和写入的内容。它展示了基于文件描述符的流式文件的主要 I/O 操作技术，以及如何灵活的控制文件的读写位置和读写记录长度。同时也揭示了 Unix 系统中文件的各种管理信息以及对于各类文件的识别和控制方法。

1) 在新文件夹中中建立以下名为 filexm.h 的 C 语言头文件：

1) 在新文件夹中中建立以下名为 filexm.c 的 C 语言程序：

```
/*
   *  description   :  filexm.c
   *  copyright     : (C) by 张鸿烈
   * Function       : 本程序说明了怎样通过 linux 文件系统调用取得
   *                  完整的文件管理信息,以及基本的文件读写操作。
*/

#include "filexm.h"
char data[BSZ];   //文件读写缓冲区
```

```
int main(int argc, char *argv[])
{
  int   fd1,fd2;         //打开文件的文件描述字
  int   offset;        //文件读字节偏量
  int   i,size;        //文件读写字节数

  //检查命令是否带有约定的运行参数
  if(argc < 5){
    perror("USAGE: ./filexm read_filename write_filename offset size");
    exit(EXIT_FAILURE);
  }

  // 检查命令指定的读文件名是否存在,如果存在打开该文件
  if((fd1 = open(argv[1],O_RDONLY)) < 0){
    perror(argv[1]);
    exit(EXIT_FAILURE);
  }
  else //显示读文件名
    printf("Open read file: %s\n",argv[1]);

  // 建立或打开命令指定的写文件并准备向其追加内容
  if((fd2 = open(argv[2],O_CREAT | O_RDWR | O_APPEND ,0644)) < 0){
    perror(argv[2]);
    exit(EXIT_FAILURE);
  }
  else//显示写文件名
    printf("Create write File: %s\n",argv[2]);

  //显示文件读位移量
  offset = atoi(argv[3]);
  printf("File Read offset: %d\n",offset);

  //显示读写文件字节数
  size = atoi(argv[4]);
  printf("Copy total size: %d\n",size);

  //读写文件的内容
  lseek(fd1,offset,SEEK_SET); //读位置移到距离读文件头 offset 字节处
  for(i=0; i < size/BSZ; i++) {
      read (fd1,data,BSZ);    //读 BSZ 字节
      write(fd2,data,BSZ);    //写 BSZ 字节
  }
  read (fd1,data,size%BSZ);    //读 BSZ 字节
  write(fd2,data,size%BSZ);    //写 BSZ 字节

  //关闭文件
```

```
  if(close(fd2) <0){
    perror(argv[2]);
    exit(EXIT_FAILURE);
  }
  if(close(fd1) <0){
    perror(argv[1]);
    exit(EXIT_FAILURE);
  }


  //显示写后文件状态
  File_State(argv[2]) ;

  //显示写后文件内容
  if((fd2 = open(argv[2],O_RDONLY)) < 0){
    perror(argv[2]);
    exit(EXIT_FAILURE);
  }
  printf("========%s content========\n",argv[2]);
  while( read(fd2,data,1) > 0 )
    write(1,data,1);
  printf("\n");

  return EXIT_SUCCESS;
}
```

2) 在新文件夹中中建立以下名为 filexm.h 的 C 语言头文件：

```
/*
    * Filename          : filexm.h
    * copyright         : (C) by  张鸿烈
     * Function         :  获取的文件管理信息,检测文件读写操作。
*/

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

#define BSZ 16

//判断文件类型
void File_Type(mode_t mode){
    //显示文件类型
    printf("File Type:\t");
    //链接文件
```

```
    if(S_ISLNK(mode)) printf("Symbolic Linke\n");
    //普通文件
    else if(S_ISREG(mode)) printf("Regular\n");
    //目录文件
    else if(S_ISDIR(mode)) printf("Directoy\n");
    //字符设备
    else if(S_ISCHR(mode)) printf("Character Device\n");
    //块设备
    else if(S_ISBLK(mode)) printf("Block Device\n");
    //管道文件
    else if(S_ISFIFO(mode)) printf("FIFO\n");
    //套接口
    else if(S_ISSOCK(mode)) printf("Socket\n");
    //不可识别的设备
    else printf("Unkown type\n");
}

//显示文件控制信息
void File_State(char * fn)
{
   struct stat buf;    //保存打开文件信息的缓冲区
   int fd ;
   //打开文件
   if((fd = open(fn,O_RDONLY)) < 0){
      perror(fn);
      exit(EXIT_FAILURE);
   }

   //是否能读出该文件的控制信息
   if((fstat(fd,&buf)) < 0){
      perror(fn);
      exit(EXIT_FAILURE);
   }

   //读出写后的文件的控制信息,显示文件控制信息
   //显示节点号
   printf("========%s state========\n",fn);
   printf("File INODE:\t%ld\n",buf.st_ino);
   //显示设备号
   printf("File DEVICE:\t%d,%d\n",major(buf.st_dev),minor(buf.st_dev));
   //显示保护方式
   printf("File MODE:\t%#o\n",buf.st_mode & ~(S_IFMT));
   //显示连接数
   printf("File LINKS:\t%d\n",buf.st_nlink);
   //显示用户 ID
   printf("File UID:\t%d\n",buf.st_uid);
```

```
    //显示组 ID
    printf("File GID:\t%d\n",buf.st_gid);
    //分析并显示文件的类型
    File_Type(buf.st_mode);
    //显示文件长度
    printf("File SIZE:\t%ld\n",buf.st_size);
    //显示文件系统 I/O 块长度
    printf("File BLK SIZE:\t%ld\n",buf.st_blksize);
    //显示已分配 I/O 块长度
    printf("File BLOCK:\t%ld\n",buf.st_blocks);
    //显示最后访问文件的时间
    printf("File ACCESSED:\t%s",ctime(&buf.st_atime));
    //显示最后修改文件的时间
    printf("File MODIFIED:\t%s",ctime(&buf.st_mtime));
    //显示最后改变文件的时间
    printf("File CHANGED:\t%s",ctime(&buf.st_ctime));
}
```

3）在新建文件夹中建立以下 Makefile 文件

```
srcs = filexm.c
objs = filexm.o
all:        filexm
filexm:     $(objs)
            gcc $(objs) -o filexm
filexm.o:  $(srcs)
            gcc -c $(srcs)
clean:
            rm filexm *.o
```

4）在当前目录中输入make编译命令:

```
$ gmake
gcc -c filexm.c
gcc filexm.o -o filexm
```

2) 建立一个测试文件：

```
$ vi f1
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890!@#$%^&*()-+=[];
```

4) **运行程序 filexm，**使用 f1 做为输入文件， f2 做为输出文件，从 f1 文件的第 27 个字节处，读 26 个字符写入文件 f2(即 f1 文件的第二行)，最后显示 f2 文件的文件信息和文件内容：

```
$ ./filexm f1 f2 27 26
Open read file: f1
Create write File: f2
File Read offset: 27
```

```
Copy total size: 26
======== f2 state ========
File INODE:   1468641
File DEVICE: 8,5
File MODE:    0644
File LINKS:    1
File UID: 1000
File GID: 1000
File Type:       Regular
File SIZE:       26
File BLK SIZE:      4096
File BLOCK:  8
File ACCESSED:   Thu Aug 16 20:49:12 2006
File MODIFIED:    Thu Aug 16 20:49:12 2006
File CHANGED:    Thu Aug 16 20:49:12 2006
======== f2 content ========
ABCDEFGHIJKLMNOPQRSTUVWXYZ
$
```

可以看到该程序报告了f2 文件在写入了f1 文件的部分内容后所具有的控制信息和复制的内容。

9.4 独立实验

利用流式文件系统调用的基本操作，构造一个能管理文本文件的学生成绩表的简单数据库管理系统。设文本文件的学生成绩数据表文件每条记录有 3 个字段构成：学号 20 个字节，姓名 20 个字节，成绩 10 个字节，字段间用空格分割对齐。
简单数据库管理系统具有基本的功能有：

追加一条记录，（仅允许文件主）
按学号读出一条记录，
按学号升序列出所有记录 ，.
按学号删除（逻辑删除）一条记录，（仅允许文件主）
压缩掉被删除的记录。（仅允许文件主）

（提示：可建立一个学生成绩表文件和一个以学号为主键的索引表文件）

9.5 实验要求

根据示例实验和独立实验中观察和记录的信息，说明它们反映出教材中讲解的文件系统的哪些原理和概念？如何使用基于文件描述符的流式文件的 I/O 技术，以及如何灵活的控制文件的读写位置和读写记录长度？Unix 系统中有哪些文件管理信息？系统是如何识别和控制各类文件的？通过这些信息说明操作系统是怎样将不同的物理设备映射为文件系统的统一的文件操作的？一切皆文件的理念为我们带来了什么好处。根据实验程序、调试过程和结果分析写出实验报告。

实验十、分布式系统实验

10.1 实验目的

加深对于分布式系统中的主要计算模型：客户 / 服务器模式的了解。熟悉客户 / 服务器模式的网络体系结构。掌握 TCP 套接口的基本编程技术。

10.2 实验说明

1）套接口（Socket）

在 TCP 协议中一个套接口是进程间通信的一个端点。每个套接口都有唯一的名字，所以其他进程都可以唯一的识别、连接并访问到它。一对连接的套接口构成进程间相互交换数据的一条信道。服务器程序创建的套接口起到了多个客户机信息汇集点的作用。套接口和字符设备的文件描述符有许多共同的特性。

2）通信域（Domain）

通信域用来说明套接口通信协议的语义。每个通信域都规定了一套协议、控制和解释的规则，以及套接口的地址格式。最广泛使用的通信域就是 Internet 通信域。

对于 Internet 通信域，套接口的地址格式就是一个 IP 地址和一个端口号。

3）套接口编程

套接口编程需要 4 个步骤：

（1）分配和初始化套接口。

类似与使用 open 打开文件分配一个文件描述符，连接一个网络端口要使用 socket 分配一个套接口描述符。

socket 分配一个套接口的系统调用的语法为：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

domain 说明本套接口的通信域。以下是一些常用的通信域：

AF_UNIX	UNIX 内部协议
AF_INET	IPv4 Internet 协议
AF_INET6	IPv6 Internet 协议
AF_IPX	IPX - Novell 协议
AF_APPLETALK	Appletalk 协议

type 说明本套接口类型，主要有以下类型：

SOCK_STREAM	面向连接的，可靠的顺序的双向连接
SOCK_DGRAM	非面向连接的，不可靠连接
SOCK_RAW	用于内部网络协议（root 用户专用）

protocol 说明一个附加的协议，无附加协议时总为 0。

Socket 系统调用很类似于 open 系统调用，成功后返回一个类似文件描述符的套接口描述符。不成功返回-1，出错号放系统变量 errno 中。

分配一个套接口的操作对于服务器和客户机来说都是一样的。 对于服务器来说在成功分配到一个套接口后，下一步还要将一个进程与该套接口绑定起来，准备

接收客户机的连接。客户机不必做这项工作。

bind 绑定一个套接口的系统调用语法为：

```
#include <sys/types.h>
#include <sys/socket.h>
int     bind(int     sockfd,  const  struct  sockaddr  *my_addr,  socklen_t
        addrlen);
```

sockfd　　由 socket 操作分配的套接口描述符

my_addr　　sockaddr 结构指针

sockaddr 结构为：

```
struct sockaddr {
        sa_family_t sa_family;
        char        sa_data[14];
}
```

sa_family　　通信域

sa_data　　14 个字节的协议地址

这个域地址在编程时由实际采用的通信域地址来替换。例如您工作在 Internet 通信域，则说明的实际地址结构应为以下 Internet 通信域地址：

```
struct sockaddr_in {
    sa_family_t     sin_family; /* AF_INET 通信协议*/
    u_int16_t       sin_port;   /* 网络端口号 */
    struct in_addr  sin_addr;   /* IP 地址结构 */
};
 /* IP 地址结构   */
 struct in_addr {
     u_int32_t      s_addr;      /* IP 地址 */
  };
```

其中的网络端口号可以用以下函数得到：

```
#include <arpa/inet.h>
uint16_t htons(uint16_t hostshort); 将一个短整数转换为一个网络端口号
uint32_t htonl(uint32_t hostlong); 将一个整数转换为一个网络端口号
```

其中的 IP 地址 s_addr 可以用常数 INADDR_ANY (0.0.0.0)任意地址填充；

或用常数 INADDR_LOOPBACK (127.0.0.1) 本机地址填充;

也可以从以下 gethostbyname 函数返回的 hostent 结构中得到，gethostbyname 将一个由主机域名或点分十进制表示的 IP 地址转换为一个 IP 协议格式定义的 hostent 结构。

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);
struct hostent {
        char    *h_name;        /* 公开的主机名 */
        char    **h_aliases;    /* 主机别名表 */
        int     h_addrtype;     /* 协议类型，总为 AF_INET 或 AF_INET6*/
        int     h_length;       /* 地址长度 */
        char    **h_addr_list; /* 地址表 */
```

}
#define h_addr h_addr_list[0] /* 首条地址 */
gethostbyname 成功返回一个指向 hostent 结构的指针，不成功返回 NULL，出错号放 h_errno 中。

（2）完成连接

对于服务器来说在成功获取并绑定一个套接口后，还要调用侦听客户机请求的系统调用 list，在接收队列上侦听客户的接入信号。

list 系统调用的语法：

#include <sys/socket.h>
int listen(int sockfd, int backlog);
sockfd 由 socket 操作分配的套接口描述符
backlog 接入信号缓冲队列的大小
成功返回 0，不成功返回-1，出错号放系统变量 errno 中。

当一个接入信号抵达服务器套接口后，首先挂入缓冲队列等待处理。在侦听到有信号接入后，accecp 系统调用负责检索和接收一个挂入的接入信号。

accept 系统调用的语法：

#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
sockfd 由 socket 操作分配的套接口描述符
addr sockaddr 结构指针，它是发送方通信域的一个地址
addrlen addr 能够容纳的最大字节数
成功返回非负值，不成功返回-1，出错号放系统变量 errno 中。

对于客户机上的进程，在成功分配到一个套接口后需要通过 connect 系统调用连接上远程服务器上对应的服务进程。

connect 系统调用的语法：

#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
sockfd 由 socket 操作分配的套接口描述符
serv_addr sockaddr 结构指针，
addrlen 设置 serv_addr 能够容纳的最大字节数
connect 成功返回 0，不成功返回-1，出错号放系统变量 errno 中。

(3) **传送数据**

一旦建立好连接，就可以开始在服务器/客户机之间传输数据了。对于面向连接的传输，recv 系统调用用于接收套接口传来的数据，send 系统调用用于向套接口发送的数据。

recv 和 send 系统调用的语法：

#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int socket, void *buf, size_t len, int flags);

ssize_t send(int socket, const void *buf, size_t len, int flags);

socket　　已建立连接的套接口

buf　　接收或发送数据的缓冲区指针

len　　接收或发送数据的缓冲区的长度

flags　　操作标志，可以为：

对于 recv 操作

MSG_OOB 处理带外数据。不设置，处理一般非带外数据

MSG_PEEK　查看接入消息但不处理

MSG_WAITALL 等待缓冲区数据填满后再返回

对于 send 操作

MSG_OOB 处理带外数据。不设置，处理一般非带外数据

MSG_DONTROUTE 不使用路由

recv 执行成功返回接收到的字节数，不成功返回-1，出错号放系统变量 errno 中。

send 执行成功返回发送出的字节数，不成功返回-1，出错号放系统变量 errno 中。

(4) **关闭连接**

当完成数据通信后需要使用 close 系统调用关闭套接口，以便断开网络的连接。

Close 系统调用的语法：

#include <unistd.h>

int close(int socket);

10.3 示例实验

示例实验程序要实现一个两个远程客户端程序通过远程服务器完成类似单机中两进程双向管道通信的操作。两个客户端程序同时向服务器发送数据，服务器在收到两个客户机发来的数据后，经简单的处理后再将数据交叉发回两客户机，从而实现两客户程序的信息交换。示例程序中服务器的 IP 地址采用本机回送地址 127.0.0.1，这样我们仅在同一台机器上就可以完成我们的实验。

1）在新建文件夹中建立以下 server.c 文件

```
/*
    * Filename          : server.c
    * copyright         : (C) by  张鸿烈
    * Function          ： 信息交换服务器。
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <ctype.h>

#define BUFSIZE   16384
char buf[2][BUFSIZE];              //收发数据缓冲区
```

```
int       server_socket;          //服务器套接口
int       port = 8088;            //服务端口号
struct sockaddr_in    server_in;   //服务器网络地址

int       client_socket[2];       //客户端套接口
socklen_t client_size;            //客户端网络地址长度
struct sockaddr_in    client_in;   //客户端网络地址

int main(int argc,char *argv[])
{
    int i,len[2];

  //申请 TCP/IP 协议的套接口
  if((server_socket = socket(AF_INET,SOCK_STREAM,0)) == -1 ){
    perror("call to socket");
    exit(1);
  }

  //初始化 IP 地址
  bzero(&server_in,sizeof(server_in));//清 0
  server_in.sin_family = AF_INET;
  server_in.sin_addr.s_addr = INADDR_ANY;
  server_in.sin_port = htons(port);

  //套接口捆定 IP 地址
  if(bind(server_socket,(struct sockaddr *)&server_in,sizeof(server_in)) == -1){
    perror("call to bind");
    exit(1);
  }

  //监听网络请求
  if(listen(server_socket,200) == -1){
    perror("call to listen");
    exit(1);
  }

  //循环等待接收信息和处理信息
  printf("Accepting connections ...\n");

  while(1){
    //接收第一客户请求
    if((client_socket[0] = accept(server_socket,(struct sockaddr
*)&client_in,&client_size)) == -1){
      perror("call to accept 1 ");
      exit(1);
    }
    //接收第二客户请求
```

```
    if((client_socket[1] = accept(server_socket,(struct sockaddr
*)&client_in,&client_size)) == -1 ){
        perror("call to accept 2 ");
        exit(1);
     }

     //获取第一客户发送的数据
     if(recv(client_socket[0],buf[0],BUFSIZE,00) == -1){
        perror("call to recv 1");
        exit(1);
     }
     printf("received from client1: %s\n",buf[0]);

     //获取第二客户发送的数据
     if(recv(client_socket[1],buf[1],BUFSIZE,0) == -1){
        perror("call to recvv 2");
        exit(1);
     }
     printf("received from client2: %s\n",buf[1]);

     //都转换为大写
     len[0] = strlen(buf[0]);
     for(i=0;i<len[0];i++) buf[0][i] = toupper(buf[0][i]);
     len[1] = strlen(buf[1]);
     for(i=0;i<len[1];i++) buf[1][i] = toupper(buf[1][i]);

     //把第二客户发来的数据发给第一客户
     if(send(client_socket[0],buf[1],len[1],0) == -1){
       perror("call to send 1");
       exit(1);
     }
     printf("send to client1: %s\n",buf[1]);

     //把第一客户发来的数据发给第二客户
     if(send(client_socket[1],buf[0],len[0],0) == -1){
       perror("call to send 1");
       exit(1);
     }
     printf("send to client2: %s\n",buf[0]);

    //断开与客户的连接
    close(client_socket[0]);
    close(client_socket[1]);
   }
}
```

3) **在新建文件夹中建立以下 client.c 文件**

```
/*
```

```
    * Filename              : client.c
    * copyright             : (C) by  张鸿烈
    * Function              ： 请求信息交换的客户程序
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <ctype.h>

#define BUFSIZE    16384
char buf[BUFSIZE]; //收发数据缓冲区
//  要连接的服务器的 IP 地址为本机回送地址
char * host_name = "127.0.0.1";
int port = 8088;    //服务端口号
int server_socket;
struct sockaddr_in server_in;
struct hostent *server_name;

int main(int argc, char *argv[])
{
  int rate;
  char str[64]="ok" ;

  //可以在命令行第二参数指定客户发送的字符穿，第三参数指定一个延迟秒数
  if(argc == 3){
      rate = atoi(argv[2]);
      strcpy(str,argv[1]);
  }
  else if(argc == 2){
      rate = 1;
      strcpy(str,argv[1]);
  }
  else
      rate = 1;

  //转换服务器域名或 IP 地址为 IP 结构体
  if((server_name = gethostbyname(host_name)) == 0){
            perror("Error resolving local host\n");
            exit(1);
  }

  //初始化 IP 地址结构
  bzero(&server_in,sizeof(server_in));
  server_in.sin_family = AF_INET;
```

```
    server_in.sin_addr.s_addr = htonl(INADDR_ANY);
    server_in.sin_addr.s_addr = ((struct in_addr*)(server_name->h_addr))->s_addr;
    server_in.sin_port = htons(port);

    while(1){
        //获取远程服务器套接口描述符
        if((server_socket = socket(AF_INET,SOCK_STREAM,0)) == -1){
          perror("call to socket");
          exit(1);
                }

        //发送连接请求到到服务器
       if(connect(server_socket,(void *)&server_in,sizeof(server_in)) == -1){
            perror("call to connect");
            exit(1);
               }

        //发送数据到服务器
       if(send(server_socket,str,strlen(str),0) == -1){
          perror("error in send \n");
          exit(1);
               }
       printf("Sending message: %s \n",str);

       //从服务器接收数据
       if(recv(server_socket,buf,BUFSIZE,0) == -1){
          perror("error in recv");
          exit(1);
               }
       printf("Respone from Server: %s \n",buf);

        sleep(rate);

        //断开连接
       close(server_socket);
    }
}
```

3) 在新建文件夹中建立以下 Makefile 文件

```
s_src = server.c
s_obj = server.o
c_src = client.c
c_obj = client.o
opts    = -g -c
all:          server client

server:     $(s_obj)
              gcc $(s_obj) -o server
```

```
server.o:  $(s_src)
           gcc $(opts) $(s_src)

client:    $(c_obj)
           gcc $(c_obj) -o client
client.o:  $(c_src)
           gcc $(opts) $(c_src)

clean:
           rm server client *.o
```

5) **使用 make 命令编译生成可执行的 sever 和 client 程序**

$ make

```
gcc -g -c server.c
gcc server.o    -o server
gcc -g -c client.c
gcc client.o -o client
```

6) **在当前目录中启动服务程序**

$ *./server*

Accepting connections ...

以上输出信息表示此时服务器已开始工作，准备接收客户机的请求。

6）**打开一个新的终端窗体进入当前目录启动第一个客户端程序**

```
$ ./client first
Sending message: first
```

可以看到由于现在只有一个客户所以它在等待第二个客户通过服务器发回的数据。

7）**现在再打开一个新的终端窗体进入当前目录启动第二个客户端程序**

```
$ ./client second
Sending message: second
Respone from Server: FIRST
Sending message: second
Respone from Server: FIRST
    ......
```

现在因为服务器同时收到了两个客户的数据，它可以将第一个客户发来的字母数据转换为大写转发给第二个客户，将第二个客户发来的字母数据转换为大写转发给第一个客户。两个客户程序在收到服务器发回的数据后也继续向服务器发送数据。

在第一客户终端看到：

```
Sending message: first
Respone from Server: SECOND
Sending message: first
Respone from Server: SECOND
```

Sending message: first
Respone from Server: SECOND
.....

在服务器窗体中此时可以看到它接收和发送的字符串：

received from client1: first
received from client2: second
send to client1: SECOND
send to client2: FIRST
received from client1: first
received from client2: second
send to client1: SECOND
send to client2: FIRST
.....

10.4 独立实验

请利用服务器/客户机网络计算模式，完成实验 2 独立实验中提出的计算任务。例如令服务器负责分派计算任务给 3 个客户机，3 个客户机一个负责计算 n 的阶乘，一个负责计算 fibonacc 序列，一个负责将另外两个客户机计算的结果加起来返回给用户，从而实现一个网上的分布式计算实验。

10.5 实验要求

根据示例实验和独立实验，分析服务器/客户机网络计算模式的特点。说明服务器/客户机基本工作步骤有那些？它们是怎样建立连接和交换数据的？它们反应出教材中介绍的分布式系统哪些原理和概念？您是怎样利用服务器/客户机网络计算模式实现一个网上的分布式计算实验的。根据实验程序、调试过程和结果分析写出实验报告。

附录 A 操作系统原理实验建议

A.1 认真阅读操作系统原理教程

在开始我们每项算法实验之前应认真温习操作系统教材相关章节讲授的算法原理以及认真阅读实验说明和示例程序。在理解了示例程序的基本思路后再结合算法原理展开我们的实验和分析。

A.2 实验时间安排

可以考虑以下的顺序和时间开展实验，其中不包括独立实验的设计时间。

其中难度系数设 4 级为最难。

次序	学时	实验内容	难度系数
1	2	操作系统命令实验	1
2	3	进程控制实验	3
3	3	进程通信实验	3
4	3	进程调度实验	3
5	4	进程同步实验	4
6	4	进程互斥实验	4
7	4	进程死锁实验	4
8	3	内存页置换实验	2
9	3	磁盘臂调度实验	2
10	3	文件操作实验	2
11	4	分布系统实验	4

A.3 实验过程的组织

- 可以组成 2-4 人的实验小组，以便于交流讨论。
- 控制实验进度，确保按时完成实验。

- 及时与指导教师交流。
- 按时提交实验报告

A.4 实验的检查

- 脱机检查：通过学生提交到网上的实验报告和实验结果检查
- 联机检查: 通过上机检查：包括编辑过程、编译过程、调试过程、结果测试过程。

A.5 实验成绩的评定

实验成绩的评定应从多方面考察。

- 实验题目的难度系数
- 实验报告的质量
- 实验结果的性能
- 实验完成的时间
- 实验的创新

附录 B 操作系统原理实验报告纲要

在完成了实验后，写好实验报告是很重要的，它可以帮助我们：

- 总结实验成果，启发创新思维。
- 锻炼科研精神，训练写作能力。
- 及时反馈实验效果，便于交流讨论
- 作为成绩评定的主要参考

指导教师应事先给出实验报告的标准格式，规定好报告的提交日期和提交方式。对于实验报告的检查应注意：

- 是否具有独到的创新设计
- 是否达到了基本要求
- 是否按规定的格式编写的报告
- 结构是否严谨，清晰
- 内容是否完整，全面
- 语言文字是否流畅，表达是否准确

建议的实验报告基本内容：

操作系统原理实验报告

1、 基本信息
   - 实验题目
   - 小组编号
   - 完成人
   - 报告日期
2、 实验内容简要描述
   - 实验目标
   - 实验要求
   - 实验的软硬件环境
3、 报告的主要内容
   - 实验的思路
   - 实验模型的描述
   - 主要数据结构的分析说明
   - 主要算法代码的分析说明
   - 项目管理文件的说明
4、 实验过程和结果
   - 实验投入的实际学时数
   - 调试排错过程的记录
   - 多种方式测试结果的记录
   - 实验结果的分析综合

5、 实验的总结

- 实验中遇到的问题和解决的方法
- 实验结果达到设计目标的程度
- 还可以进行哪些改进
- 实验得到哪些收获和启发

附录 1：参考文献

附录 2：程序源代码

附录 C 操作系统原理实验报告样例

山东大学XXX学院操作系统实验报告

（报告样例，仅供参考。张鸿烈）

学号：XXXXXXXXXX

专业：XX

班级：XX 班

姓名：XXX

Email：

日期：XXXX 年 XX 月 XX 日

实验题目： 吸烟者问题

假设一个系统中有三个吸烟者进程，每个吸烟者不断地卷烟并吸烟。吸烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和胶水。一个吸烟者有烟草，一个有纸，另一个有胶水。系统中还有两个供应者进程，它们无限地供应所有三种材料，但每次仅轮流提供三种材料中的两种。得到缺失的两种材料的吸烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供另外的两种材料。这一过程重复进行。请用IPC同步机制编程，实现该问题要求的功能。

实验目的：

加深对并发协作进程同步与互斥概念的理解,观察和体验并发进程同步与互斥操作的效果,分析与研究经典进程同步与互斥问题的实际解决方案。了解Linux系统中IPC进程同步工具的用法,练习并发协作进程的同步与互斥操作的编程与调试技术。

硬件环境：

CPU： P4/1.8MHz 内存：256MB 硬盘： 10GB

软件环境：

Ubuntu08.4－Linux 操作系统

Gnome 桌面 2.18.3

BASH_VERSION='3.2.33(1)-release

gcc version 4.1.2

vi 3.1.2

gedit 2.18.2

OpenOffice 2.3

实验步骤：

1、问题分析

假设 T,P,G 分别为吸烟必备材料（T=烟草，P=纸，G=胶水）。供应商每次随机

供应（生产）卷烟材料中的两种:T&P，P&G，G&T。

……..

2、算法设计说明如下：

为实验该问题在系统进程间的并发关系,本实验采用了systemV的 IPC 的信号量和共享内存，这样可以观察到进程在系统级上的并发和调度的情况。创建 3 个字节的共享内存用作放材料的容器，用以下指针引用:

……

供应商程序：

```
{
    ……
}
```

吸烟者程序：

```
{
    ……
}
```

3、开发调试过程：

在 shell 命令终端中建立一个 smoker 子目录：

```
$mkdir smoker
$cd   smoker
```

在该目录中编写供应商程序文件provider.c，吸烟者程序文件smoker.c,

编写一个项目 Makefile 文件

$gedit Makefile provider.c smoker.c ipc.c ipc.h &

使用 make 命令编译：

$make

```
 gcc -g -c provider.c ipc.c
 provider.c: 在函数 'main' 中：
 provider.c:70: 错误：expected expression before '==' token
 provider.c:70: 错误：expected statement before ')' token
  make: *** [provider.o] 错误 1
$
```

改正错误，重新编译 make 成功：

$make

…….

$ ls

生成了可执行文件 provider 和 smoker，在 不同的命令终端中进入该目录，分别运行:

$./provider

28780 供应商提供: 胶水 G,烟草 T
28781 供应商提供: 烟草 T,纸张 P

$./smoker

28756 吸烟者有胶水 G,取到烟草 T,纸张 P.开始吸烟...
28758 吸烟者有纸张 P,取到胶水 G,烟草 T.开始吸烟...

之后供应商/吸烟者进程好像都阻塞了，这几个并发进程似乎出现了死锁。

打断本次执行，再次运行：

$./provider
28783 供应商提供: 烟草 T,纸张 P

$./smoker
28786 吸烟者有胶水 G,取到烟草 T,纸张 P.开始吸烟...

之后供应商/吸烟者进程好像又都阻塞了，这几个并发进程似乎又出现了死锁。

分析调试：

这种现象说明供应商第一次放产品及唤醒吸烟者是对的，问题可能发生在吸烟者没有再次唤醒供应商或供应商之间互斥时没有相互唤醒。而且总在供应商提供:烟草 T,纸张 P 之后死锁。所以要重点检查 T&P 的同步互斥操作是否正确。

经检查供应商的 T&P 的互斥操作的 V 操作信号量使用了吸烟者的 T&P 信号量，致使提供 T&P 的供应商互斥后没有被唤醒，而吸烟者的 T&P 被唤醒了两次，从而导致在进行了一次存取后这几个进程出现了死锁。

改正后再次运行：

```
$./provider
28867 供应商提供: 烟草 T,纸张 P
28867 供应商提供: 纸张 P,胶水 G
……

$./smoker
28870 吸烟者有胶水 G,取到烟草 T,纸张 P.开始吸烟...
28871 吸烟者有烟草 T,取到纸张 P,胶水 G.开始吸烟...
……
```

可以看到 2 个供应者进程和 3 个吸烟者进程无论供应的产品怎样随机产生总能按

照我们期望的操作次序连续不但的工作。

结论分析与体会：

通过吸烟者问题的实验体会到现代操作系统中解决并发进程同步和互斥的操作的重要性。当协作进程并发执行时只有在一些关键点使用同步机制才能使多个并发进程正确的工作，…….。

通过本实验还练习了我们调试和测试并发程序的能力，学会了如何分析和排除并发环境中的程序错误。例如: ……

附录 A： 本实验全部程序源代码及注释

smoker.c

```
/*
```

```
   *    description              smoker.c
   *    copyright                 : (C) by  张鸿烈
   *    Function                  ：建立吸烟者进程
*/

#include "ipc.h"

int main(int argc,char *argv[])
{
……

}
```

smoker.h

……

provider.c

……

provider.h

……

Makefile

……

附录 B：　　参考教材/文献

……

参考文献:

1. Abraham Silberschatz, Peter Galvin, and Greg Gagne, Applied Operating system concept, John Wiley & Sons, Inc, Sixth Edition ,2002

2．Gary Nutt. Operating Systems: A Modern Perspective. Addison Wesley, 2002

3. Andrew S. Tanenbaum, Modern Operating Systems, Prentice Hall, Second Edition, 2001