北京工业大学计算机学院

操作系统实验指导书

V 4.5, Sept 2020

目录

实验环境说明

操作系统实验需要在 Linux 操作系统中完成。具体发行版本不限，但要求有 C/C++编译器 GCC。

课表内的上机时间安排在计算中心机房，使用的台式机已经安装好 Linux 系统，有图形用户界面。

为了得到充分的锻炼，要求学生具备课后实验条件。学生可以从以下几种方式当中选择：

- 在自己的计算机上安装 Linux 虚拟机。建议使用 VirtualBox（开源）~~或 VMware Workstation（计算机学院软件系订阅了 VMware 学术计划 2016.08-2019.07，为同学创建账户以后可以使用正版软件）~~。Linux 操作系统的发行版不限，要求必须有 GCC 编译器。下面是推荐版本：
  - Ubuntu Desktop 18.04 或 20.04
  - Ubuntu Server 18.04 或 20.04（字符界面，资源占用少）
- 拷贝教师制作好的虚拟机文件。
- 登录专门为本课程准备的 Linux 服务器
  - 访问范围：校园网内，或者通过学校 VPN 登录进入校园网
  - 服务器 IP 地址：172.21.17.211
  - 登录用户名为学生本人的学号（例如 18072164），密码到本班课代表出查询
  - 远程登录、文件传输可以分别使用 PuTTY、WinSCP

推荐两款工具软件：

- PuTTY（http://www.chiark.greenend.org.uk/~sgtatham/putty/）：一款开源的终端仿真软件。用户在 Window 系统中运行 PuTTY，可以登录到 Lunix 主机（或虚拟机）。
  - 如果希望使用 PuTTY 登录到虚拟机，要求虚拟机的网络连接方式为 Bridge 或 Host Only；还需要知道该虚拟机的 IP 地址。获得虚拟机 IP 地址的方法：打开一个终端（Terminal）窗口，执行命令：ifconfig。
- WinSCP（http://winscp.net）：一款开源的文件传输软件，运行于 Windows 平台。该软件通过 SFTP、SCP、 FTPS 或 FTP 协议进行本机和远程计算机的文件传输。使用该软件在本机（Windows）和 Linux 机器之间传输文件，通过鼠标拖拽即可完成。

实验报告要求

实验报告应包括如下内容：

- 报告表头：实验报告题目、姓名、学号、完成时间
- 实验目的
- 实验内容
- 实验要求
- 实验设计（功能设计、数据结构、程序框图、参数说明）
- 实验测试结果及分析
- 运行结果

- 收获及体会
- 参考资料

以上是文字资料。通过教师上机测试后，要按教师要求上交源代码、目标代码、测试数据的文本（如光盘、电子邮件等方式）。

主要参考资料

- **W. R. Stevens, S. A. Rago, 戚正伟、张亚英、尤晋元译. UNIX 环境高级编程（第 3 版）. 人民邮电出版社. 2014.06.**
  （使用这本书的第 1、2 版或英文影印版也可以）
- 系统联机手册页面（Man Pages）。在 Linux 的命令行窗口中，可以通过使用 man 命令获得联机帮助。联机手册一般分为以下几个部分：

  1 User Commands
  2 System Calls
  3 C Library Functions
  4 Devices and Special Files
  5 File Formats and Conventions
  6 Games
  7 Miscellanea
  8 System Administration tools and Deamons

  例如，如果希望了解 fork 系统调用的详细信息，可以直接输入下面的命令查看：

  ```
  man fork
  ```

  可以 PgUp、PgDn 键上下翻页，按“q”退出联机手册。有的条目，如“open”，相同名称的条目出现在多个部分，这是可以在使用 man 命令时指定选择查看哪个部分，例如下面的命令是查看系统调用 open 的帮助信息：

  ```
  man 2 open
  ```

  可以通过使用“-a”参数，一次查看各个部分所有同名条目，例如：

  ```
  man -a open
  ```

  在下面的网页，可以查看各种典型 UNIX 类操作系统的 Man Pages：
  http://www.freebsd.org/cgi/man.cgi
  （我们可以选择操作系统为 Red Hat Linux/i386 9）

实验一 UNIX/Linux 入门

实验学时：1 学时
实验类型：验证型

一、实验目的

1、了解 UNIX/Linux 运行环境。
2、熟悉 UNIX/Linux 的常用基本命令。
3、熟悉和掌握 UNIX/Linux 下 C 语言程序的编写、编译、调试和运行方法。
4、掌握在 C 语言程序中使用**命令行参数**。

二、实验内容

【任务 1.1】了解 UNIX/Linux 文件系统的树形目录结构，熟悉 UNIX/Linux 的常用基本命令的使用方法（参见“五、Linux 常用命令”）：

下面表格中的命令，要求每名同学熟练掌握。

`ls`	`list files and directories` 列出文件和目录的清单
`ls -a`	`list all files and directories` 列出所有文件和目录的清单

`ls -l`	以详细方式列出文件和目录的清单。
	在结果中可以看到每个文件/目录的访问权限、文件所有者、长度、时间、文件名等信息。
	
	练习：看看下面 3 条命令，执行结果的异同。
	`ls -l`
	`ls -a -l`
	`ls -al`

`mkdir`	`make a directory`
	创建一个目录。
	
	例如，下面的命令在当前目录下创建一个目录 abc
	`mkdir abc`

`cd` *`directory`*	`change to named directory` 将当前目录改变为指定目录 *`directory`*
`cd`	`change to home-directory` 将当前目录改变为主目录
`cd ..`	`change to parent directory` 将当前目录改变为父目录

pwd	display the path of the current directory 显示当前目录的路径
cp *file1 file2*	copy file1 and call it file2 拷贝文件 file1，并将结果文件命名为 file2
mv *file1 file2*	move or rename file1 to file2 移动（或重命名）文件 file1 为 file2 练习：把当前目录下的 abc.txt 文件移动的父目录。
rm *file*	remove a file 删除文件
rmdir *directory*	remove a directory 删除目录
cat *file*	display a file 显示一个文件（文件名为 *file*） 练习：分别执行下列命令，观察结果。（假定当前目录下有 hello.c） cat hello.c cat –n hello.c
more *file*	display a file a page at a time 显示文件，每次一页（一屏）。按空格键翻到下一页，按回车键向下滚动一行。
head *file*	display the first few lines of a file 显示一个文件的最初若干行
tail *file*	display the last few lines of a file 显示一个文件的最后若干行
grep *'keyword' file*	search a file for keywords 在一个文件中查找关键字的出现
wc *file*	count number of lines/words/characters in file 计算一个文件中的行数、单词数、字符数
command > *file*	redirect standard output to a file 将 标准输出 重定向 到一个文件。 例如，如果仅执行下面的命令： ls 命令的结果----文件和目录清单 将被输出到 标准输出，即屏幕。 如果执行 ls > list.txt ls 命令的输出结果将不会显示在屏幕，而是被重定向到文件 list.txt，即文件和目录清单被保存到 list.txt 文件。

command >> *file*	append standard output to a file 把标准输出追加到一个文件。 请同学进行一个对比实验。 首先，执行下面两条命令： ls > list111.txt ls > list111.txt 然后，执行下面两条命令： ls > list222.txt ls >> list222.txt 最后，对比 list111.txt 与 list222.txt 有何不同。
command < *file*	redirect standard input from a file 把 标准输入 重定向 为一个文件。 执行一个命令时，默认的标准输入是键盘。输入重定向使得运行的程序从一个文件中读取输入，而不是等待键盘输入。我们在调试/测试一个程序时可以利用输入重定向，这样，就不必辛辛苦苦地一遍一遍敲输入数据了
command1 | *command2*	pipe the output of command1 to the input of command2 管道，将 *command1* 的输出 作为 *command2* 的输入 练习： ls | grep ".c"
cat *file1 file2* > *file0*	concatenate file1 and file2 to file0 将 *file1* 和 *file2* 连接，结果保存在 *file0*
sort	sort data 排序数据
who	list users currently logged in 列出 当前登录用户 的清单 （**注意，“登录”和“登陆”是两回事**）
ps	list current processes 列出当前各个进程。 练习：执行下面 4 个命令，观察输出结果。 ps ps u ps au ps aux

【任务 1.2】练习 UNIX/Linux 的文本编辑器（如 gedit 或 vi）的使用方法。也可以熟悉 Linux 中的 C/C++集成开发环境（IDE）的使用（例如 Code::Blocks）。

【任务 1.3】熟悉 UNIX/Linux 下 C 语言编译器 gcc 的使用方法。

（1）用 gedit 或其他编辑器编写一个简单的 C 语言程序 hello.c，功能是在标准输出设备上显示“Hello,World!”。

（2）在终端窗口中，用 gcc 编译 hello.c，得到名为 hello 的可运行程序：

```
gcc hello.c -o hello
```

注意，上面一行的编译命令中，“hello.c -o hello”是 gcc 命令的命令行参数。其中，

- “hello.c”是被编译的源程序文件名。
- “-o hello”表示编译的输出结果（在本例中即可执行程序）命名为 hello 的文件，

（3）运行可执行程序 hello，并观察运行结果：

```
./hello
```

【任务 1.4】掌握 C 语言程序命令行参数的使用方法。

（1）使用编辑器编辑下面的 C 语言源程序（假定源程序的文件名为 argtest.c），**请特别注意 main 函数的参数**：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char** argv) {
5      int i;
6
7      printf("argc = %d\n", argc);
8
9      for (i = 0; i <argc; i++) {
10         printf("Argument %d: %s\n", i, argv[i]);
11     }
12
13     return 0;
14 }
```

（2）编译：

```
gcc argtest.c -o argtest
```

（3）在终端窗口中，分别以下列 3 种方式运行，观察输出结果，**请解释程序的功能**：

```
./argtest

./argtest hello

./argtest hello world
```

【任务 1.5】编写一个 C 语言程序，完成文件拷贝功能。

要求：

（1）源文件名和目标文件名通过命令行参数指定。

（2）注意出错检查和错误提示。包含但不限于：命令行参数个数是否争取、文件打开是否成功、文件读/写是否成功等。

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、关于在 Linux 系统中编辑、编译 C 语言程序的几点说明

在本次实验中，建议初次使用 Linux 系统的同学在编辑 C 语言源程序时，使用 gedit 编辑器（类似于 Windows 中的“记事本”编辑器）。

如果有兴趣学习最基本的 vi 功能，可以阅读一份网上资料：“vi in 10 minutes" (http://www.unix-manuals.com/tutorials/vi/vi-in-10-1.html)

gcc 是 GNU C/C++编译程序（compiler，也称作编译器）。

在 UNIX 中 C 语言程序的 main 函数应该具有 int 返回值类型。通常，返回“0”表示成功，非零表示各种失败。

有些版本的 GCC C/C++编译器要求源程序最后一个字符后面至少有一个回车符。

五、Linux 常用命令

Linux 系统常用命令格式：

```
command [option] [argument1] [argument2] ...
```

其中 option 以“-”开始，多个 option 可用一个“-”连起来，如“ls -l -a” 与“ls -la”的效果是一样的。根据命令的不同，参数分为可选的或必须的；所有的命令从标准输入接受输入，输出结果显示在标准输出，而错误信息则显示在标准错误输出设备。可使用重定向功能对这些设备进行重定向。

命令在正常执行结果后返回一个 0 值，如果命令出错可未完全完成，则返回一个非零值(在 shell 中可用变量**$?**查看，在命令行提示符后面输入

```
echo $?
```

即可查看刚刚执行完的程序的返回值)。 在 shell script 中可用此返回值作为控制逻辑的一部分。

在 UNIX/Linux 系统中，命令名、文件名、目录名都是大小写敏感的（case sensitive）。

命令和各个参数（或选项）之间要以空格（或制表符）分割。比如，“`ls -a`”是指一个命令“`ls`”后面跟一个选项“`-a`”；“`ls-a`”则指一个名为“`ls-a`”的命令。

如果希望知道一个命令（比如 gcc）在文件系统的什么位置，可以使用“which”命令，例如，在终端窗口键入下面的命令：

```
which gcc
```

帮助命令：man（获取相关命令的帮助信息），info（获取相关命令的详细使用方法）。

例如，

```
man dir
```

可以获取关于 ls 的使用信息。

例如，

```
info info
```

可以获取如何使用 info 的详细信息。

为了同学们系统地熟悉 Linux 系统，建议阅读英国 Surrey 大学电机工程系提供的《UNIX 初学者教程》（http://www.ee.surrey.ac.uk/Teaching/Unix/）。。

请同学在按照该教程边阅读、边操作，注意理解 UNIX/Linux 文件系统的树形结构。

下面分类列出了 UNIX/Linux 命令。**注意，不要求同学全面掌握下列所有命令。**

压缩、备份命令：

```
bzip2/bunzip2  .bz2 文件的压缩/解压缩程序
cpio  备份文件
dump  备份文件系统
gzip/gunzip  .gz 文件的压缩/解压缩程序
gzexe  压缩可执行文件
restore 还原由倾倒(Dump)操作所备份下来的文件或整个文件系统(一个分区)
tar  将若干文件存档或读取存档文件
unarj  解压缩.arj 文件
zip/unzip  压缩/解压缩 zip 文件
```

磁盘操作：

```
cd  切换目录
pwd 显示当前工作目录
df  显示磁盘的相关信息
du  显示目录或文件的大小
e2fsck  检查 ext2/ext3 文件系统的正确性
fdisk  对硬盘进行分区
fsck  检查文件系统并尝试修复错误
losetup  设置循环设备
ls  列出目录内容
mkdir  创建目录
mformat  对 MS-DOS 文件系统的磁盘进行格式化
mkbootdisk  建立目前系统的启动盘
mke2fs  建立 ext2 文件系统
mkisofs  制作 iso 光盘映像文件
mount/umount 加载文件系统/卸载文件系统
quota  显示磁盘已使用的空间与限制
sync  将内存缓冲区内的数据写入磁盘
tree  以树状图列出目录的内容
```

系统操作:

```
alias  设置指令的别名
chkconfig  检查，设置系统的各种服务
clock  调整 RTC 时间
date  显示或设置系统时间与日期
dmesg  显示开机信息
eval  重新运算求出参数的内容
exit  退出目前的 shell
```

export 设置或显示环境变量
finger 查找并显示用户信息
free 显示内存状态
hostid 显示主机标识
hostname 显示主机名
id 显示用户标识
kill 删除执行中的程序或工作
last 列出目前与过去登入系统的用户相关信息
logout 退出系统
lsmod 显示已载入系统的模块
modprobe 自动处理可载入模块
passwd 设置用户密码
ps 报告程序状况（process status）
reboot 重启计算机
rlogin 远程登入
rpm 管理 Linux 各项套件的程序
shutdown 关机
su 变更用户身份
top 显示，管理执行中的程序
uname 显示系统信息
useradd/userdel 添加用户 / 删除用户
userinfo 图形界面的修改工具
usermod 修改用户属性，包括用户的 shell 类型，用户组等，甚至还能改登录名
w 显示目前注册的用户及用户正运行的命令
whereis 确定一个命令的二进制执行码，源码及帮助所在的位置
who 列出正在使用系统的用户
whois 查找并显示用户信息

网络通信：

arp 网地址的显示及控制
ftp 文件传输
lftp 文件传输
mail 发送 / 接收电子邮件
mesg 允许或拒绝其他用户向自己所用的终端发送信息
mutt E-mail 管理程序
ncftp 文件传输
netstat 显示网络连接、路由表和网络接口信息
pine 收发电子邮件，浏览新闻组
ping 向网络上的主机发送 icmp echo request 包
ssh 安全模式下的远程登录
telnet 远程登录
talk 与另一用户对话
traceroute 显示到达某一主机所经由的路径及所使用的时间
wget 从网络上自动下载文件
write 向其他用户的终端写信息

文件操作：

```
cat  显示文件内容和合并多个文件
clear  清屏
chattr  改变文件属性
chgrp  改变文件组权
chmod  改变文件或目录的权限
chown  改变文件的属权
comm  比较两个已排过序的文件
cp  将文件拷贝至另一文件
dd  从指定文件读取数据写到指定文件
df  报告磁盘空间使用情况
diff  比较两个文本文件，列出行不同之处
du  统计目录 / 文件所占磁盘空间的大小
file  辨识文件类型
emacs  功能强大的编辑环境
find  搜索文件并执行指定操作(find2)
grep  按给定模式搜索文件内容
head  显示指定文件的前若干行
less  按页显示文件
ln  创建文件链接
locate  查找符合条件的文件
more  在终端屏幕按帧显示文本文件
mv  文件或目录的移动或更名
rm/rmdir  删除文件 / 目录
sed  利用 script 来处理文本文件
sort  对指定文件按行进行排序
tail  显示指定文件的最后部分
touch  创建文件
tr  转换字符
vi  全屏编辑器
wc  显示指定文件中的行数，词数或字符数
which  在环境变量 $PATH 设置的目录里查找符合条件的文件
```

实验二 进程创建与管道通信

实验学时：2 学时

实验类型：验证型、设计型

一、实验目的

加深对进程概念的理解，明确进程与程序的区别；进一步认识并发执行的实质。学习在 Linux 系统中创建子进程和进行管道通信的方法。

二、实验内容

【任务 2.1】运行创建子进程的示例程序，观察运行结果，做出正确解释。

说明：

- `getpid()` 用于获得本进程的 `PID`
- `getppid()` 用于获得父进程的 `PID`
- `wait(NULL)` 用于在父进程中等待子进程结束

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main() {
    pid_t pid; // process ID

    printf("Before fork, PID = %d, PPID = %d. ", getpid(), getppid());
    printf("\n");

    pid = fork(); // to create a child process
    if (pid == -1) {
        perror("Failed in calling fork");
        exit(1);
    } else if (pid == 0) {
        /* the child process */
        printf("In child process, PID = %d, PPID = %d. ", getpid(), getppid());
        printf("\n");

        // ----- A -----

        // ----- B -----
        exit(0);
    } else {
        /* the parent process */
        printf("In parent process, child's PID = %d. ", pid);
        printf("\n");

        printf("In parent process, PID = %d, PPID = %d. ", getpid(), getppid());
        printf("\n");

        // ----- C -----
        wait(NULL);
    }

    printf("Before return in main(), PID = %d, PPID = %d. ", getpid(), getppid());
    printf("\n");

    return (EXIT_SUCCESS);
}
```

程序 1 使用 fork()创建子进程

（1）编辑示例程序，如 程序 1 所示。在“Terminal”窗口，编译、执行这个程序。

解释输出的信息当中，哪些来自父进程、哪些来自子进程。

（2）在“Terminal”窗口，执行 ps 命令，可以显示出 Shell （可能是 bash）的进程 PID 以及 ps 命令执行所对应的进程的 ID。记住 Shell 进程的 ID，再次执行上面的程序，看看父进程的父进程 ID 与 Shell 进程 ID 的关系。

（3）在 程序 1 中“// ----- A -----”的下面，添加下面一条语句：

```
sleep(3);
```

这条语句的作用是“睡眠”3 秒钟后再执行。重新编译、执行程序，解释现象。

（4）在（3）的基础上，将“// ----- B -----”下面的一条语句删去。重新编译、执行程序，解释现象。

（5）在（4）的基础上，将“// ----- C -----”下面的一条语句删去。重新编译、执行程序，解释现象。

注意，在 UNIX/Linux 系统中，如果一个进程的父进程先于自己“结束”，系统将该进程的父进程设为 1 号进程。

【任务 2.2】运行示例程序，绘制进程家族树。

小明同学编写了一个程序，如 程序 2 所示。他希望该程序能够实现先后创建 2 个子进程的功能。

编译、运行该程序。回答下列问题：

（1）一共创建了几个进程？它们各自的 PID 是什么？

（2）根据输出，绘制一棵进程家族树，表示父进程和被创建的进程之间的关系。

（3）根据输出，说明每个进程都执行了哪些内容（注意时间顺序）。

（4）你认为这个程序是否实现了小明同学的目标？

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main() {

    pid_t result1;
    pid_t result2;

    printf("<A> -- pid = %d, ppid = %d\n", getpid(), getppid());

    result1 = fork();
    if (result1 < 0) {
        perror("Failed to create process after <A>");
        exit(1);
    }

    printf("<B> -- pid = %d, ppid = %d, result1 = %d.\n", getpid(), getppid(), result1);

    result2 = fork();
    if (result2 < 0) {
        perror("Failed to create process after <B>");
        exit(1);
    }

    printf("<C> -- pid = %d, ppid = %d, result2 = %d.\n", getpid(), getppid(), result2);

    sleep(2);
    return (EXIT_SUCCESS);
}
```

程序 2 进程家族树实例代码

【任务 2.3】编写一个程序。该程序的功能：

- 输出一个字符串 “Original\n”。
- 创建一个子进程
  - 在子进程中，输出字符串 “Child 1\n”
  - 在父进程中，
    - 输出字符串 “Parent\n”
    - 再次创建一个子进程
      - 在第二个子进程中，输出字符串 “Child 2\n”。

【任务 2.4】编写程序，实现进程之间的管道通信。

具体要求：

- 首先使用系统调用 pipe( )建立一个管道；
- 然后分别创建 2 个子进程，要求 2 个子进程分别向管道各写一句话：“Child 1 is sending a message!” 和 “Child 2 is sending a message!”
- 最后，父进程从管道中读出二个来自子进程的信息并显示。

请参考“四、补充材料”。

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、补充材料

（一）注意采用**良好的程序书写风格**。比如，缩进格式、有明确意义的函数名、变量名等。

在执行系统调用、库函数调用后，应该立刻检查返回值，根据不同的返回值做相应处理包括**出错处理**。例如，在使用 fork 系统调用时，通常都采用下面的程序结构：

```
pid = fork(); // to create a child process
if (pid < 0) {
    perror("Failed in calling fork");
    exit(1);
}
else if (pid == 0) {
    /* the child process */
    ........
}
else {
    /* the parent process */
    ......
}
```

（二）关于进程创建，阅读课本 3.3.1 节。

实现本实验的要求，需要用到下面的系统调用：

1、创建子进程 fork。

2、创建管道 pipe。管道是进程间通信中最古老的方式，它包括无名管道和有名管道两种，前者用于父进程和子进程间的通信，后者用于运行于同一台机器上的任意两个进程间的通信。

3、读文件 read。

4、写文件 write。

5、关闭文件 close。

下面的例子示范了如何在父进程和子进程间实现管道通信。下面代码完成的功能是：首先创建了一个管道，然后创建了一个子进程；子进程向管道写了一句话；父进程从管道读取信息并输出。**特别注意，在创建管道、向管道写、从管道读之后，都应当检查是否有错误。**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  #include <string.h>
6  #include <errno.h>
7
8  #define INPUT 0
9  #define OUTPUT 1
10
11 int main() {
12     int file_descriptors[2];
13     pid_t pid; // process ID
```

```
14      char *msg = "I have a dream. "; // message to send
15      char buf[256]; // buffer to store the message
16      int read_count;
17      int result;
18
19      /* to create a pipe */
20      result = pipe(file_descriptors);
21      // file_descriptors[INPUT] for read end of the pipe
22      // file_descriptors[OUTPUT] for write end of the pipe
23      if (result == -1) {
24         // fail to create pipe
25         perror("Failed in calling pipe");
26         exit(1);
27      }
28
29      pid = fork(); // to create a child process
30      if (pid == -1) {
31         perror("Failed in calling fork");
32         exit(1);
33      } else if (pid == 0) {
34         /* the child process */
35         printf("In the child process...\n");
36         // to close the input end of the pipe and
37         // write a message to the output end
38         close(file_descriptors[INPUT]);
39
40         result = write(file_descriptors[OUTPUT], msg, strlen(msg));
41         if (result == -1) {
42            perror("In Child1, failed to write to pipe");
43            exit(1);
44         }
45
46         close(file_descriptors[OUTPUT]);
47         exit(0);
48      } else {
49         /* the parent process */
50         printf("In the parent process...\n");
51         // to close the output end and
52         // read from the input end
53         close(file_descriptors[OUTPUT]);
54
55         read_count = read(file_descriptors[INPUT], buf, sizeof(buf));
56         if (read_count == -1) {
57            perror("In parent, failed to read from");
```

```
58            exit(1);
59         } else if (read_count == 0) {
60            printf("In parent, 0 byte read from pipe.\n");
61         } else {
62            // read_count > 0
63            buf[read_count] = '\0'; // set the end of string
64            printf("%d bytes of data received from spawned process: %s\n",
65                  read_count, buf);
66         }
67
68         close(file_descriptors[INPUT]);
69      }
70      return (EXIT_SUCCESS);
71   }
```

实验三 一个进程启动另一个程序的执行

实验学时：2 学时

实验类型：验证型、设计型

一、实验目的

在 Linux 环境系统中，**execve 系统调用**用于执行一个程序（可执行二进制文件或脚本）。**exec 函数家族，包括 execl、execlp、execle、execv、execvp**，是 execve 系统调用的前端。本实验要求学生学习在一个进程中启动另一个程序执行的基本方法，了解 execve 系统调用和 exec 函数家族的使用方法。

二、实验内容

【任务 3.1】运行示例程序，初步认识“在一个进程中启动另一个程序的执行”。

1、编辑一个源程序 dummy.c，并编译为可执行程序 dummy。

```
// dummy.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char** argv) {
   int result;

   printf("\nYou are now in a running program \"%s\". \n", argv[0]);
   printf("My PID is %d. My parent's PID is %d.\n", getpid(), getppid());
   printf("Please input an integer (0-255), which will be returned to my parent
process:\n");
   scanf("%d", &result);
   printf("Goodbye.\n\n");

   return (result & 0377);
}
```

2、在 Terminal 窗口，多次运行 ./dummy ，每次输入不同值（0-255 之间）并且立即执行下面的命令并注意观察结果：

```
echo $?
```

3、再编辑一个源程序 task-3.1.c，并编译为可执行程序 task-3.1。

```
// task-3.1.c
#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <sys/types.h>
```

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

int main(int argc, char** argv) {
   int result;

   result = fork();
   if (result < 0) {
      perror("Failed to create child");
      exit(1);
   } else if (result == 0) {
      // Child 1
      char *cmd = "./dummy";

      printf("Child process's PID is %d. My parent's PID is %d.\n", getpid(), getppid());
      printf("Child process is about to execute \"%s\"\n\n", cmd);

      result = execlp(cmd, cmd, NULL);
      if (result == -1) {
         perror("In child process, failed to exec a program");
      }

      exit(1);
   } else {
      // parent
      int status;

      printf("Parent process's PID is %d.\n", getpid());
      printf("Parent process is waiting ... \n");
      wait(&status);
      printf("In parent process, status = 0x%x, WEXITSTATUS(status) = %d (i.e. 0x%x)\n", \
            status, WEXITSTATUS(status), WEXITSTATUS(status));
   }

   return (EXIT_SUCCESS);
}
```

4、执行程序 task-3.1，观察、分析执行结果。（注意，两个可执行程序都在当前目录下）

【任务 3.2】编写程序，实现在一个进程中启用运行另外一个程序。

具体要求：

- 获取命令行参数（参加【任务 1.4】）。例如，假定该程序的可执行文件为 myprog，用户在 Terminal 中输入的命令为

```
./myprog ls -a -l /home
```

该程序的命令行参数中，第一个为将来希望执行的程序（ls），后面为执行这个程序时需要的命令行参数（ -a 、 -l 和 /home ）

- 创建子进程
  - 在子进程中，启动执行命令行参数中指定的新程序，并传递给它相应的参数。例如，执行 ls ，参数为 -a 、 -l 和 /home 。（**提示：使用 execvp**）
  - 在父进程中，等待子进程运行结束，并打印子进程的退出状态。

【任务 3.3】实现一个简单的命令解释外壳（Shell）。

1、基本功能：

（1）从标准输入读取一行字符串，其中包含欲执行的命令和它的命令行参数（如果有的话）。提示：需要将输入的一行字符串进行拆分，以**空格**、**制表符**（\t）作为分隔符，分解为命令、命令行参数（零个或多个）。如果用户输入的命令是“quit”，则退出执行。

（2）创建一个子进程。

（3）在子进程中，执行在（1）读入的命令，如果有命令行参数，也要传递。（**提示：使用 execvp**）

（4）在父进程中，等待子进程结束，然后打印子进程的返回值。

（5）在父进程中，控制转移至（1）。

2、扩展功能（选作）：实现子进程的输入/输出重定向。

3、提示：

（1）从终端读取要执行的命令可用 fgets()实现。

（2）关于拆分命令行参数。

首先，考虑自己编程实现这个功能。假设用户输入的一整行信息（字符串）是：

l	s		\t	-	a		-	l		\0

拆分以后，让一个字符指针的数组（char *arg[MAX_ARGC]）的各个元素分别执行命令和各个参数：

l	s	\0	\t	-	a	\0	-	l	\0	\0

arg[0]
arg[1]
arg[2]
arg[3] → NULL

利用 arg 数组，可以调用 execvp 实现执行一个程序。

除了上述自己动手实现的方法，还可以调用字符串处理函数 strtok_r。**特别说明：下学期《编译原理》词法分析实验中不允许使用这个函数！**理由到那时自己就应当明白。

（3）关于输入/输出重定向。当用户在命令行窗口执行一个程序时，操作系统在创建进程的同时，会为该进程打开三个文件：标准输入 stdin、标准输出 stdout 和标准错误输出 stderr，默认情况下，它们分别指向键盘、屏幕、屏幕。用户在运行一个程序时也可以指定

其他文件作为标准输入、标准输出。例如，执行下面的命令，当前目录的清单将保存在文件 abc.txt 中，而不是显示在屏幕上，这是一个输出重定向的例子：

```
ls > abc.txt
```

如果在命令的后面使用“<”和一个文件名，则进行输入重定向，即命令运行将从指定的文件（而不是键盘）读取输入内容。

为了在这个简单的 Shell 中实现输入输出重定向，首先需要分析使用 fgets 获得的用户键入的命令和参数，分析其中是否有输入输出重定向的指示。如果有，在创建的子进程中，必须先打开或创建文件，令 stdin/stdout 指向文件，然后启动执行期望的程序/命令。

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、补充材料

在 Linux 中要使用 execve 系统调用和 exec 函数家族实现在一个进程来启动另一个程序。在 Linux 中 exec 函数家族包括：execl、execlp、execle、execv 和 execvp。下面以 execlp 为例，其它函数究竟与 execlp 有何区别，请通过 man exec 命令来了解它们的具体情况。

一个进程一旦调用 execve 系统调用/exec 函数，它原本所执行的程序就“中止”了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆、栈，并为新程序分配新的数据段与堆、栈，唯一保留的是进程号。换言之，对系统而言，还是同一个进程，不过该进程已经执行另一个程序了。

如果希望在一个进程中启动另一个程序的执行而该进程仍然能够继续运行，可以使用 fork 和 execve 系统调用/exec 函数。试分析下面这个程序的功能。注意，这个例子只能处理一次只输入一条命令（例如“ls”），不允许包含其他命令行参数（例如“ls -a -l”）。

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    int pid;
    int rtn; /*子进程的返回数值*/
    int exec_errorno;
    char command[256];
    char *p;

    while (1) {
        /* 从终端读取要执行的命令 */
        printf(">");
```

```
        command[0] = '\0';
        p = fgets(command, 256, stdin);
        if (p == NULL) {
            perror("Error in fgets()");
            exit(-1);
        }

        // Delete the last char (new line) in the string returned by fgets()
        command[strlen(command) - 1] = '\0';

        // Quit if user inputs "quit"
        if (!strcmp(command, "quit")) {
            break;
        }

        // Create a child process to execute command
        pid = fork();
        if (pid < 0) {
            perror("Failed while calling fork");
            exit(-1);
        }
        else if (pid == 0) {
            /* 子进程执行此命令 */
            exec_errorno = execlp(command, command, NULL);
            /* 如果 exec 函数返回，表明没有正常执行命令 */
            /* 只有在这种情况下，才会执行下面的打印错误信息*/
            perror(command);
            exit(exec_errorno);
        }
        else {
            /* 父进程， 等待子进程结束，并打印子进程的返回值 */
            wait(&rtn);
            printf("\nValue returned from child process, rtn = %d\n", rtn);
            printf("WEXITSTATUS(rtn) = %d\n", WEXITSTATUS(rtn));
        }
    }

    return 0;
}
```

实验四 线程管理

实验学时：2 学时

实验类型：设计

一、实验目的

编写 Linux 环境下的 POSIX 多线程程序，了解多线程的程序设计方法，掌握最常用的三个函数 pthread_create、pthread_join 和 pthread_exit 的用法。

二、实验内容

【任务 4.1】在主程序中创建两个线程，myThread1 和 myThread2。每个线程打印一句话。

提示：

（1）阅读“四、补充材料”中的 2 个示例程序，编译、执行并观察运行效果。

（2）使用 pthread_create(&id, NULL, (void *) thread, NULL)完成。先定义每个线程的执行体，然后在 main( )中创建两个线程，最后主线程等待子线程结束后再退出。

【任务 4.2】创建两个线程，分别向线程传递如下两种类型的参数并在线程中打印：

- 传递整型值
- 传递字符

【任务 4.3】实现向被创建进程传递的参数为**结构体**、从线程返回**结构体**。（例如，完成 2 个复数的加/减/乘/除）

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、补充材料

在 Linux 操作系统中可以使用 POSIX 线程接口（称为 pthread）进行多线程编程。编写 Linux 下的 POSIX 多线程程序，需要包含头文件 pthread.h，连接时需要使用 pthread 库。

POSIX 线程常用函数：

（1）创建线程 pthread_create。

（2）等待另一线程结束 pthread_join。

（3）线程终止 pthread_exit。

注意：编译时要使用如下命令（设 example.c 是源程序名字）。因为 pthread 的库不是 Linux 系统的库，所以在**进行编译的时候要加上-pthread**，否则编译不过。具体命令如下：

```
gcc -o example example.c -pthread
```

提示：如果使用 Netbeans C/C++ IDE，并且在该环境中编译源程序，也需要指明 lpthread。你能想到应该在哪里进行设置么？

下面展示一个简单的多线程程序 example.c [1]。

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid) {
   long tid;
   tid = (long) threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for (t = 0; t < NUM_THREADS; t++) {
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) t);
      if (rc) {
         perror("Failed in calling pthread_create");
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

程序 3　多线程示例一

在上面的示例中，使用 pthread_create( )创建了 5 个线程。编译后运行若干次，看看运行结果与预期是否一致。

下面的程序[1]展示了如何向被创建进程传递参数、等待一个线程结束以及获得结束线程的返回值。由于下面的程序中使用了 sin 等数学库中的函数，因此在编译时需要加入数学库，编译命令如下：

```
gcc example.c -o example -pthread -lm
```

程序清单：

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t) {
   int i;
```

```
   long tid;
   double result = 0.0;
   tid = (long) t;
   printf("Thread %ld starting...\n", tid);
   for (i = 0; i < 1000000; i++) {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n", tid, result);
   pthread_exit((void*) t);
}

int main(int argc, char *argv[]) {
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for (t = 0; t < NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *) t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is % d\n", rc);
         exit(-1);
      }
   }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for (t = 0; t < NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is % d\n", rc);
         exit(-1);
      }
      printf("Main: completed join with thread %ld having a status of %
ld\n",
            t, (long) status);
   }
```

```
    printf("Main: program completed. Exiting.\n");
    pthread_exit(NULL);
}
```

五、参考文献

[1] Blaise Barney. POSIX Threads Programming. Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/pthreads/

[2] 课本第 4.3.1 节。

六、进一步要求（选作）

1、了解 Linux 线程，并和 POSIX 线程进行对比。尝试创建 Linux 线程（参见课本 4.5.2 节）。

2、设计、实现课本第 4 章后面的编程项目——矩阵乘法。

实验五 线程间的互斥与同步

实验学时：2 学时

实验类型：验证、设计型

一、实验目的

理解 POSIX 线程（Pthread）互斥锁和 POSIX 信号量机制，学习它们的使用方法；编写程序，实现多个 POSIX 线程的同步控制。

二、实验内容

【任务 5.1】创建 4 个 POSIX 线程。

2 个线程(A 和 B)分别从 2 个数据文件(data1.txt 和 data2.txt)各读取 10 个整数。线程 A 和 B 每次读入一个整数，立即把该数据放入一个缓冲池；如此重复 10 次。

缓冲池由 n 个缓冲区构成（n=5，并可以方便地调整为其他值），每个缓冲区可以存放一个整数。

另外 2 个线程，C 和 D，每次从缓冲池读取 1 个数据；每读取 2 个数据后，每读出 2 个数据，分别求出两个数的和（线程 C）或乘积（线程 D），并打印输出。如此重复 5 次。

提示：

（1）在创建 4 个线程当中，A 和 B 是生产者，负责从文件读取数据到公共的缓冲区，C 和 D 是消费者，从缓冲区读取数据然后作不同的计算（加和乘运算）。使用互斥锁和信号量控制这些线程的同步。不限制线程 C 和 D 从缓冲区得到的数据来自哪个文件。

（2）在生产者线程中，确保从**文件读出数据**以后，再去“**生产**”。

在开始设计和实现之前，务必认真阅读下列内容：

- **课本 6.8.4 节；**
- **讲义（课堂 PPT）中关于“生产者-消费者问题”的部分；**
- **课本第 6 章后面的编程项目——生产者-消费者问题。**

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、补充材料

1、Synchronizing Threads with POSIX Semaphores.

http://www.csc.villanova.edu/~mdamian/threads/posixsem.html

2、关于 POSIX 信号量的 Man Pages：

（1）Pthread 互斥锁 （通过 `man` 查看 `pthread_mutex_lock` 帮助信息）

（2）POSIX 信号量 （通过 `man` 查看 `sem_init`、`sem_wait`、`sem_post` 帮助）

3、在本实验中，可能用到的与文件操作有关的库函数：fopen, fclose, feof, fscanf。请在终端窗口中使用 **man** 命令查看这些函数的详细联机帮助信息。

实验六 基于消息队列和共享内存的进程间通信

一、实验目的

“系统 V 进程间通信”（System V IPC，System V interprocess communication）机制包括：消息队列、信号量集和共享存储区。本实验的目的是了解和熟悉**消息通信机制**、**共享存储区**的原理，以及它们的使用方法。

二、实验内容

【任务 6.1】进程间通过消息队列通信。

要求：

- 编写 2 个程序，my_msg1.c 和 my_msg2.c。
- 第一个程序 my_msg1.c 首先创建一个消息队列，然后消息队列中发送 10 个消息（每一条消息的内容自定，可以是一行字符串，建议从文件中读取）。
- 第二个程序 my_msg2.c 从上述的消息队列中接受 10 条消息并在屏幕上显示，最后删除这个消息队列。
- 分别在 2 个终端窗口编译上面 2 个源程序得到可执行程序 my_msg1 和 my_msg2。先运行 my_msg1，等待 my_msg1 执行结束后，再运行 my_msg2。

提示：

（1） 使用 `msgget()`、`msgsnd()`、`msgrcv()`、`msgctl()`等系统调用完成上面的任务。

（2） 注意每一次系统调用后的错误处理。

尝试与思考：

（1） 在执行 my_msg1 之前，在终端窗口执行命令 ipcs；在执行 my_msg2 之前，执行命令 ipcs；my_msg2 执行结束后，再次执行 ipcs。试解释 ipcs 命令的输出结果。

（2） 如果将 2 个程序的执行顺序颠倒，会出现什么情况？

【任务 6.2】进程间通过共享内存通信。

要求：

- 编写 2 个程序，my_shm1.c 和 my_ shm2.c。
- 第一个程序 my_shm1.c 首先创建一个共享内存段（不小于 1KB），并将这段共享内存映射到自己的地址空间，然后向这段共享内存写入一组字符（不少于 1024 个，建议从文件中读取），最后解除共享内存的映射。。
- 第二个程序 my_shm2.c 首先获取上述共享内存段，并映射到自己的地址空间；从共享内存段读出数据并显示；解除共享内存短的映射；最后删除的消息队列中接受 10 条消息并在屏幕上显示，最后删除这个消息队列。
- 分别在 2 个终端窗口编译上面 2 个源程序得到可执行程序 my_shm1 和 my_shm2。先运行 my_shm1，等待 my_shm1 执行结束后，再运行 my_shm2。

提示：

（3） 使用 shmget()、shmat()、shmdt()、shmctl()等系统调用完成上面的任务。

（4） 注意每一次系统调用后的错误处理。

尝试与思考：

（1） 在执行 my_shm1 之前，在终端窗口执行命令 ipcs；在执行 my_shm2 之前，执行命令 ipcs；my_shm2 执行结束后，再次执行 ipcs。试解释 ipcs 命令的输出结果。

（2） 如果将 2 个程序的执行顺序颠倒，会出现什么情况？

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、补充材料

1、查看“System V interprocess communication mechanisms” Man Pages。命令：

```
man 7 svipc
```

2、消息队列

消息是一个用户可定义的通用结构，例如：

```
struct msgbuf {
   long mtype; /* 消息类型，必须 > 0 */
   char mtext[1]; /* 消息文本 */
};
```

(1) 创建/获取消息队列 msgget

int msgget(key_t key, int msgflg);

功能：创建或者访问一个消息队列。

该函数使用头文件如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

参数：

key：消息队列关联的键。

msgflg：消息队列的建立标志和存取权限。如果 flag 为 IPC_CREAT，表示如果内核中没有此队列，则创建它。 Flag 为 IPC_EXCL 当和 IPC_CREAT 一起使用时，表示如果队列已经存在，则失败。 IPC_EXCL 单独使用是没有用处的。

下面是一个创建和打开消息队列的例子。注意，第 3 行的 0666 是一个八进制整数（如果以二进制表示，是 110 110 110），它表示了所创建的消息队列的**访问权限**，分别针对所有者（owner）、所有者的同组（group）和其他（other）对该消息队列“可读”、“可写”。在创建 System V IPC 中的消息队列、信号量集、共享存储区时，都需要指定访问权限。

```
int open_queue(int keyval) {
   int qid;

   qid = msgget(keyval, IPC_CREAT | 0666);
```

```
    if (qid == -1) {
        perror("Failed in calling msgget");
        return (-1);
    }
    return (qid);
}
```

(2) 消息操作

发送消息 msgsnd

```
int msgsnd(msgid, msgp, size, flag)
```

功能：发送一条消息。

其中：

msgid 消息队列的描述符；

msgp 是指向用户存储区的一个构造体指针，

size 指示由 msgp 指向的数据结构中字符数组的长度； 即消息的长度。

flag 规定当核心用尽内部缓冲空间适应执行的动作；若在标志 flag 中未设置 IPC_NOWAIT 位，则当该消息队列中的字节数超过一最大值时，或系统范围的消息数超过某一最大值时，调用 msgsnd() 进程睡眠。若是设置 IPC_NOWAIT，则在此情况下，msgsnd() 立即返回。

接收消息 msgrcv

```
ssize_t msgrcv(msgid, msgp, size, type, flag)
```

功能：接收一条消息。

其中：msgid 是消息队列的识别码。

msgp 是用来存放欲接收消息的用户数据结构的地址。

size 是 msgp 中数据数组的大小；

type 是用户要读的消息类型：

type 为 0 ：接收该队列中的全部消息；

type 为正： 接收类型 type 的第一个消息；

type 为负： 接收小于或等于 type 绝对值的最低类型的第一个消息/

flag：用来指明核心程序在队列没有数据的情况下所应采取的行动。如果 msgflg 和常数 IPC_NOWAIT 合用，则在 msgsnd()执行时若是消息队列已满，则 msgsnd()将不会阻塞，而会立即返回-1，如果执行的是 msgrcv()，则在消息队列呈空时，不做等待马上返回-1，并设定错误码为 ENOMSG。当 msgflg 为 0 时，msgsnd()及 msgrcv()在队列呈满或呈空的情形时，采取阻塞等待的处理模式。

count 是返回消息正文的字节数。

(3) 消息控制操作 msgctl

功能描述：在指定的消息队列上执行某种控制操作。

用法：

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

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

参数：

msqid：消息队列标识码。

cmd：操作命令，可能值如下面：

IPC_STAT //将 msqid 所指定的消息队列的信息拷贝一份到 buf 指针所指向的地址。调用者必须对消息队列有读权限。

IPC_SET //将由 buf 所指向的 msqid_ds 结构的一些成员写入到与这个消息队列关联的内核结构。同时更新的字段有 msg_ctime。

IPC_RMID //删除指定的消息队列，唤醒所有等待中的读者和写者进程。

3、共享存储区

(1)共享存储区的建立 shmget

功能描述：

- 新建一个的共享储存器区段，
- 获取一个共享储存器区段的访问权。

建立（获得）一块共享存储区，返回该共享存储区的描述符 shmid；若尚未建立，便为进程建立一个指定大小的共享存储区。如果是一个新的共享存储段，shmget()将初始共享存储段中所有单元为 0。

用法：

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

int shmget(key_t key, size_t size, int shmflg);
```

参数：

key：共享储存器区段的键。

size：共享储存器区段的大小。

shmflg：建立标志和储存权限，可能值有下面那些，可通过 or 组合在一起：

IPC_CREAT　// 建立新的共享区段。

IPC_EXCL　// 和 IPC_CREAT 标志一起使用，如果共享区段已存在失败返回。

SHM_HUGETLB　// 使用"huge pages"来分配共享区段。

SHM_NORESERVE　// 不要为共享区段保留交换空间。

(2)共享存储区的控制 shmctl

shmctl 对共享存储区执行多种控制操作，对其状态信息进行读取和修改。其函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

说明:

cmd 参数指定下列 5 种命令中一种，使其在 shmid 指定的共享存储区上执行：

IPC_STAT　对此段取 shmid_ds 结构，并存放在由 buf 指向的结构中。

IPC_SET 按 buf 指向的结构中的值设置与此段相关结构中的下列三个字段：shm_perm.uid、shm_perm.gid 以及 shm_perm.mode。

IPC_RMID　从系统中删除该共享存储段。因为每个共享存储段有一个连接计数（shm_nattch 在 shmid_ds 结构中）,所以除非使用该段的最后一个进程终止或与该段脱接，否则不会实际上删除该存储段。不管此段是否仍在使用，该段标识符立即被删除,所以不能再用 shmat 与该段连接。

SHM_LOCK　锁住共享存储段。此命令只能由超级用户执行。

SHM_UNLOCK　解锁共享存储段。此命令只能由超级用户执行。

(3) 共享存储操作

共享存储区的附接 shmat

在进程已经建立了共享存储区或已获得了其描述符后，还须利用系统调用 shmat(id, addr, flag)将该共享存储区附接到用户给定的某个进程的虚地址上，并指定该存储区的访问属性，即指明该区是只读，还是可读可写。此共享存储区便成为该进程虚地址空间的一部分。

系统调用格式：

```
addr = shmat(shmid,addr,flag)
```

该函数使用头文件如下：

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

参数定义:

shmid 是共享存储区的标识符；addr 是用户给定的，将共享存储区附接到进程的虚地址；flag 规定共享存储区的读、写权限，以及系统是否应对用户规定的地址做舍入操作。其值为 SHM_RDONLY 时，表示只能读；其值为 0 时，表示可读、可写；其值为 SHM_RND（取整）时，表示操作系统在必要时舍去这个地址。该系统调用的返回值是共享存储区所附接到的进程虚地址 addr。

共享存储区的断开 shmdt

当进程不再需要一个共享存储段时，可以调用 shmdt()将它从进程的地址空间分离。

其函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(void *addr);
```

说明：

addr 参数是以前调用 shmat()时的返回值。注意：这个函数仅是是一个共享存储段不再与调用进程的地址空间相连，它并不实际删除共享存储段本身，删除共享存储段是 shmctl()函数 IPC_RMID 命令的功能。

实验七 利用信号实现进程间通信

实验学时：2 学时
实验类型：设计

一、实验目的

学习 UNIX 类操作系统的信号（signal）机制，掌握注册信号处理程序的方法，编写程序，利用信号实现进程间通信。

二、实验内容

【任务 7.1】编写一个程序，完成下列功能：实现一个 SIGINT 信号的处理程序，注册该信号处理程序，创建一个子进程，父子进程都进入等待。SIGINT 信号的处理程序完成的任务包括打印接受到的信号的编号和进程 PID。编译并运行该程序，然后在键盘上敲 Ctrl + C，观察出现的现象，并解释。

提示：参见“五、补充材料”中的 signal()的基本用法。

三、实验要求

按照要求编写程序，放在相应的目录中，编译成功后执行，并按照要求分析执行结果，并写出实验报告。

四、补充材料（摘自[1]、[2]）

信号（signal）是 UNIX 提供的进程间通信与同步机制之一。信号用于通知进程发生了某个异步事件。信号与硬件中断相似，但不使用优先级。即，认为所有信号是平等的；同一时刻发生的多个信号，每次向进程提供一个，不会进行特别排序。

进程可以相互发送信号，内核也可以发出信号。信号的发送通过更新信号接收进程的进程表的特定域而实现。由于每个信号作为一个二进制位代表，同一类型的信号不能排队等待处理。

接收进程何时处理信号？仅在进程被唤醒运行，或者它将要从一个系统调用返回的时候，进程才会处理信号。进程可以对信号作出哪些反应？进程可以执行某些默认动作（如终止运行），或者执行一个信号处理函数，或者忽略那个信号。UNIX 信号及其描述如表 2 所述（信号类型依赖于机器和操作系统，还有其它类型的）。

表 2 UNIX 信号及其描述

信号编号 **Value**	**信号名称** **Name**	**描述** **Description**
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work 连接断开
02	SIGINT	Interrupt 当用户在终端上敲中断键（通常为DELTE或Cntl + C）时，由终端驱动程序发出。该信号发送给前台进程组中的所有进程。
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core

		dump 退出；当用户在终端上敲退出键（通常为Cntl + \）时，由终端驱动程序发出，终止前台进程组中的所有进程并生成**core**
04	SIGILL	Illegal instruction 执行了无效的硬件指令
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing 跟踪陷阱；触发执行用于跟踪的代码
06	SIGIOT	IOT instruction 由具体实现定义的硬件错误
07	SIGEMT	EMT instruction 由具体实现定义的硬件错误
08	SIGFPE	Floating-point exception 算数异常
09	SIGKILL	Kill; terminate process 终止进程
10	SIGBUS	Bus error 由具体实现定义的硬件错误。通常为某种类型的存储器错误
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space 无效的内存引用；进程试图访问其虚拟地址空间以外的位置
12	SIGSYS	Bad argument to system call 无效的系统调用
13	SIGPIPE	Write on a pipe that has no readers attached to it 如果向一个管道写，但是其读者已经终止，则产生此信号
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time 时钟信号
15	SIGTERM	Software termination 终止信号，默认情况下由kill(1)命令发出
16	SIGUSR1	User-defined signal 1 用户定义信号1
17	SIGUSR2	User-defined signal 2 用户定义信号2
18	SIGCHLD	Death of a child 子进程消亡
19	SIGPWR	Power failure 电源故障

注意：不同版本的 UNIX 类操作系统支持的信号种类和编号可能有区别。上面表格所列出的是基本一致的信号。更详细信息参见[2]。

如何注册信号处理程序？需要使用 signal 系统调用：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

其中 signum 是信号的编号，即表中的 Value 列的某个值，handler 为下列 3 中情况之一：

- 常量 SIG_IGN，表示告诉系统忽略这个信号
- 常量 SIG_DFL，表示信号发生是采取默认动作
- 一个函数地址（函数指针），当信号发生时调用该函数。

signal()函数的返回值就是信号处理函数的地址，但如果出错，则返回 SIG_ERR。

信号处理函数的原型为有一个 int 型参数、返回值类型为 void。

signal()调用举例如下。

观察与思考。在编辑器中编辑下列源程序[2]：

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

static void sig_usr(int);   /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    {
        printf("can't catch SIGUSR1\n");
        exit(1);
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    {
        printf("can't catch SIGUSR2\n");
        exit(1);
    }
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
    {
        printf("received signal %d\n", signo);
        exit(1);
    }
}
```

编译上述源程序，假设得到的可执行文件为 a.out。让这个程序在后台执行：

./a.out &

执行上述命令后，终端窗口中首先显示该进程的进程 ID（PID），随后显示命令提示符。假设其 PID 为 7216。依次执行下列命令，试解释命令的含义和执行结果：

```
kill -USR1 7216
kill -USR2 7216
kill 7216
```

有关的系统调用、命令的系统联机手册页面：

- 信号处理系统调用 signal（使用 `man 2 signal`）
- 结束进程的命令 kill（使用 `man 1 kill`）
- 向进程发送消息的系统调用 kill（使用 `man 2 kill`）

五、参考文献

[1] W. Stallings. Operating Sysem: Internals and Design Principles (5th Edition). Pearson Prentice Hall, 2005. (国内有中译本)

[2] W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX(R) Environment (2nd Edition). Addison-Wesley, 2005.(人民邮电出版社影印版)