实验一　UNIX/LINUIX 入门

实验学时：2 学时
实验类型：验证型

一、　实验目的

了解 UNIX/LINUX 运行环境，熟悉 UNIX/LINUX 的常用基本命令，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。

二、　实验内容

- 熟悉 UNIX/LINUX 的常用基本命令如 ls、who、pwd、ps 等。
- 练习 UNIX/LINUX 的文本行编辑器 vi 的使用方法
- 熟悉 UNIX/LINUX 下 c 语言编译器 cc/gcc 的使用方法。用 vi 编写一个简单的显示“Hello,World!”c 语言程序，用 gcc 编译并观察编译后的结果，然后运行它。

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、　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 shell script 中可用此返回值作为控制逻辑的一部分。

帮助命令：

```
man   获取相关命令的帮助信息
      例如：man dir 可以获取关于 dir 的使用信息。

info  获取相关命令的详细使用方法
```

例如：info info 可以获取如何使用 info 的详细信息。

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/pwd 切换目录/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 process status 报告程序状况
reboot 重启计算机
rhwo 查看系统用户
rlogin 远程登入
rpm 管理 Linux 各项套件的程序
shutdown 关机
su switch user 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 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

实验二　进程管理

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。

二、实验内容

（1）进程创建

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

（2）进程控制

修改已编写的程序，将每一个进程输出一个字符改为每一个进程输出一句话，再观察程序执行时屏幕上出现的现象，并分析原因。

（3）进程的管道通信

编写程序实现进程的管道通信。使用系统调用 pipe()建立一个管道，二个子进程 P1 和 P2 分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收 P1，再接收 P2）。

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、实验报告要求

- 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）
- 实验结果及分析
- 运行结果
- 感想
- 参考资料

五、补充材料

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。

无名管道由 pipe（）函数创建：

```
#include <unistd.h>
int pipe(int filedis[2]);
```

参数 filedis 返回两个文件描述符：filedes[0]为读而打开，filedes[1]为写而打开。filedes[1]的输出是 filedes[0]的输入。下面的例子示范了如何在父进程和子进程间实现通信。

```
#define INPUT 0
#define OUTPUT 1
void main() {
int file_descriptors[2];
/*定义子进程号 */
pid_t pid;
char buf[256];
int returned_count;
/*创建无名管道*/
pipe(file_descriptors);
/*创建子进程*/
if((pid = fork()) == -1) {
printf("Error in fork\n");
exit(1);
}
/*执行子进程*/
if(pid == 0) {
printf("in the spawned (child) process...\n");
/*子进程向父进程写数据，关闭管道的读端*/
close(file_descriptors[INPUT]);
write(file_descriptors[OUTPUT], "test data", strlen("test data"));
exit(0);
} else {
/*执行父进程*/
printf("in the spawning (parent) process...\n");
/*父进程从管道读取子进程写的数据，关闭管道的写端*/
close(file_descriptors[OUTPUT]);
returned_count = read(file_descriptors[INPUT], buf, sizeof(buf));
printf("%d bytes of data received from spawned process: %s\n",
returned_count, buf);
}
}
```

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：设计

一、 实验目的

编写 Linux 环境下，fork()与 exec()的结合使用实现一个进程启动另一个程序的执行的基本方法，掌握 exec()的几种调用方法。

二、 实验内容

父进程从终端读取要执行的命令，并交给子进程执行。父进程等待子进程结束，并打印子进程的返回值。

提示：从终端读取要执行的命令可用 fgets()实现。

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、 实验报告要求

- 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）
- 实验结果及分析
- 运行结果
- 感想
- 参考资料

五、补充材料

一个进程如何来启动另一个程序的执行？

在 Linux 中要使用 exec()类的函数实现在一个进程来启动另一个程序。exec 类的函数不止一个，但大致相同，在 Linux 中，它们分别是：execl，execlp，execle，execv，execve 和 execvp，下面以 execlp 为例，其它函数究竟与 execlp 有何区别，请通过 manexec 命令来了解它们的具体情况。

一个进程一旦调用 exec 类函数，它本身就“死亡”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段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。

如果你的程序想启动另一程序的执行但自己仍想继续运行的话，怎么办呢？那就是结合 fork 与 exec()的使用。下面一段代码显示如何启动运行其它程序：

```
#include <stdio.h>

char command[256];
int main()
{
    int rtn; /*子进程的返回数值*/
    int errorno;
    while(1) {
        /* 从终端读取要执行的命令 */
        printf( ">" );
        fgets( command, 256, stdin );
        command[strlen(command)-1] = 0;
        if ( fork() == 0 ) {
            /* 子进程执行此命令 */
            errorno=execlp(command, command, NULL, NULL);
            /* 如果 exec 函数返回，表明没有正常执行命令，打印错误信息*/
            perror( command );
            exit(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( " child process return %d\n", rtn );
        }
    }
    return 0;
}
```

表 1 exec()族调用

execl（执行文件）

相关函数	fork，execle，execlp，execv，execve，execvp
表头文件	#include<unistd.h>
定义函数	int execl(const char * path,const char * arg,....);
函数说明	execl()用来执行参数 path 字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的 argv(0)、argv[1]……，最后一个参数必须用空指针(NULL)作结束。
返回值	如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

范例	#include<unistd.h>
	main()
	{
	execl(“/bin/ls”,”ls”,”-al”,”/etc/passwd”,(char *)0);

}

执行 /*执行/bin/ls -al /etc/passwd */
-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execlp（从 PATH 环境变量中查找文件并执行）

相关函数 fork，execl，execle，execv，execve，execvp

表头文件 #include<unistd.h>

定义函数 int execlp(const char * file,const char * arg,……);

函数说明 execlp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的 argv[0]、argv[1]……，最后一个参数必须用空指针(NULL)作结束。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

错误代码 参考 execve()。

范例
```
/* 执行 ls -al /etc/passwd execlp()会依 PATH 变量中的/bin 找到/bin/ls */
#include<unistd.h>
main()
{execlp("ls","ls","-al","/etc/passwd",(char *)0);}
```

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execv（执行文件）

相关函数 fork，execl，execle，execlp，execve，execvp

表头文件 #include<unistd.h>

定义函数 int execv (const char * path, char * const argv[ ]);

函数说明 execv()用来执行参数 path 字符串所代表的文件路径，与 execl()不同的地方在于 execve()只需两个参数，第二个参数利用数组指针来传递给执行文件。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

错误代码 请参考 execve（）。

范例
```
/* 执行/bin/ls -al /etc/passwd */
#include<unistd.h>
main()
{char * argv[ ]={"ls","-al","/etc/passwd",(char*) }};
execv("/bin/ls",argv);}
```

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execve（执行文件）

相关函数 fork，execl，execle，execlp，execv，execvp

表头文件 #include<unistd.h>

定义函数 int execve(const char * filename,char * const argv[ ],char * const envp[ ]);

函数说明	execve()用来执行参数 filename 字符串所代表的文件路径，第二个参数系利用数组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组。
返回值	如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

范例	#include<unistd.h>
	main()
	{
	char * argv[ ]={“ls”,”-al”,”/etc/passwd”,(char *)0};
	char * envp[ ]={“PATH=/bin”,0}
	execve(“/bin/ls”,argv,envp);
	}

执行	-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd
	execvp（执行文件）
相关函数	fork，execl，execle，execlp，execv，execve
表头文件	#include<unistd.h>
定义函数	int execvp(const char *file ,char * const argv []);
函数说明	execvp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个参数 argv 传给该欲执行的文件。
返回值	如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。
错误代码	请参考 execve（）。

范例	/*请与 execlp（）范例对照*
	#include<unistd.h>
	main()
	{ char * argv[ ] ={ “ls”,”-al”,”/etc/passwd”,0}; execvp(“ls”,argv);
	}

执行	-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

实验四 基于消息队列和共享内存的进程间通信

一、实验目的

Linux 系统的进程通信机构（IPC）允许在任意进程间大批量地交换数据。本实验的目的是了解和熟悉：

1. Linux 支持的消息通信机制及其使用方法
2. Linux 系统的共享存储区的原理及使用方法。

二、实验内容

1．消息的创建、发送和接收

使用消息调用 msgget（）、msgsnd（）、msggrev（）、msgctrl（）编制长度为 1K 的消息的发送和接收程序。

2．共享存储取得创建、附接和断接

使用系统调用 shmget（）、shmat（）、shmctl（）、shmctl（），编制一个与上述功能相同的程序。

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、实验报告要求

- 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）
- 实验结果及分析
- 运行结果
- 感想
- 参考资料

五、设计提示

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

server 端建立一个 key 为 75 的消息队列，等待其他进程发来的消息。当遇到类型为 1 的消息时，则作为结束信号，取消该队列，并退出 Server。Server 每收到一个消息后显示一句“（Server）received”；

Client 端使用 key 为 75 的消息队列，先后发送类型为 10 到 1 的消息，然后退出。最后一个消息即 server 端需要的结束信号。Client 每发送一条消息，显示一句“（Client）sent”；

父进程在 server 和 client 均退出后结束。

六、相关系统调用

1、共享内存

(1)共享存储区的建立

shmid=shmget (key ,size ,flag)：

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。

(2)共享存储区的控制

shmctl(id,cmd,buf)

对共享存储区的状态信息进行查询，如其长度、所连接的进程数、创建者标识符等；也可设置或修改其属性，如共享存储区的许可权、当前连接的进程计数等；还可用来对共享存储区加锁或解锁，以及修改共享存储区标识符等。

(3) 共享存储区的附接

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。

(4)共享存储区的断开

当进程不再需要该共享存储区时，再利用系统调用 shmdt(addr)把该区与进程断开。

2、消息队列

(1) msgqid　msgget(key,flag)

功能：获得一个消息的描述符，该描述符指定一个消息队列以便用于其他系统调用。

该函数使用头文件如下：

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

(2)msgsnd(id,msgp,size,flag)

功能：发送一条消息。

其中：id 是返回消息队列的描述符；

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。

(3)msgrcv(id,msgp,size,type,flag)

功能：接受一条消息。

其中，id 是消息描述符，

msgp 是用来存放欲接受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 ：接收该队列的第一个消息；
- 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 规定倘若该队列无消息，核心应当做什么事，如果此时设置了 IPC_NOWAIT 标志，则立即返回，若在 flag 中设置了 MSG_NOERROR，且所接收的消息大小大于 size ，

核心截断所接收的消息。

- count 是返回消息正文的字节数。

(4)msgctl(id,cmd,buf)

功能：查询一个消息描述符的状态,设置它的状态及删除一个消息描述符。

int id,cmd;

struct msqid_ds * buf;

其中：函数调用成功是返回 0，调用不成功时返回-1 。

- id 用来识别该消息的描述符； cmd 规定命令的类型。
- IPC_STAT 将与 id 相关联的消息队列首标读入 buf
- IPC_SET 为这个消息序列设置有效的用户和小组标识及操作允许权和字节的数量。
- IPC_RMID 删除 id 的消息队列

实验五 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 类操作系统信号机制，编写 Linux 环境下利用信号实现进程间通信的方法，掌握注册信号处理程序及 signal()调用方法。

二、实验内容

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、实验报告要求

- 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）
- 实验结果及分析
- 运行结果
- 感想
- 参考资料

五、补充材料（摘自[1]、[2]、[3]）

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 1 所述。

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
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

六、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)

[3] UNIX/Linux 操作系统的联机帮助（使用 man 命令）：signal(2)

实验六 线程的创建

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

1、主程序创建两个线程 myThread1 和 myThread2，每个线程打印一句话。使用 pthread_create(&id,NULL,(void *) thread,NULL)完成。

提示：

先定义每个线程的执行体，然后在 main 中()创建几个线程，最后主线程等待子线程结束后再退出。

2、创建两个线程，分别向线程传递如下两种类型的参数

- 传递整型值
- 传递字符

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、实验报告要求

- 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）
- 实验结果及分析
- 运行结果
- 感想
- 参考资料

五、补充材料：Linux 下的多线程编程简介

(1)pthread 库

Linux 下的多线程遵循 POSIX 线程接口，称为 pthread。编写 Linux 下的多线程程序，需要使用头文件 pthread.h，连接时需要使用库 pthread。Linux 下 pthread 的实现是通过系统调用 clone（）来实现的。clone（）是 Linux 所特有的系统调用，它的使用方式类似 fork，关于 clone（）的详细情况，可有关文档说明。

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-lpthread，否则编译不过。具体命令如下：

gcc example.c -lpthread -o example

下面展示一个最简单的多线程程序 example.c。

```
/* example.c*/
#include <stdio.h>
#include <pthread.h>
void thread(void)
{
   int i;
   for(i=0;i<3;i++)
      printf("This is a pthread.\n");
}
int main(void)
{
    pthread_t id;
    int i,ret;
    ret=pthread_create(&id,NULL,(void *) thread,NULL);
    if(ret!=0){
    printf ("Create pthread error!\n");
    exit (1);
    }
for(i=0;i<3;i++)
   printf("This is the main process.\n");
   pthread_join(id,NULL);
   return (0);
}
```

在上面的示例中，使用到了两个函数，pthread_create 和 pthread_join，并声明了一个 pthread_t 型的变量，它是一个线程的标识符。

（2）函数 pthread_create()的用法

函数 pthread_create()用来创建一个线程，它的原型为：

extern int pthread_create __P ((pthread_t *__thread, __const pthread_attr_t *__attr，void *(*__start_routine) (void *)， void *__arg));

第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里，如果所创建的函数 thread 不需要参数，则最后一个参数设为空指针。第二个参数一般也设为空指针，这样将生成默认属性的线程。

当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败，常见的错误返回代码为 EAGAIN 和 EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表

示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

声明和创建线程名为 myThread1 的代码片段：

```
…….
 pthread_t id1,id2;
 ret = pthread_create(&id2, NULL, (void*)myThread1, NULL);
if (ret)
    {
        printf("Create pthread error!\n");
        ….    }
```

向所创建的线程创建整数参数的代码片段：

主线程 main()中：

```
…..
pthread_t tidp;
 int error;
 int test=4;
 int *attr=&test;
 ret=pthread_create(&tidp,NULL,myThread,(void *)attr);
  ……..
```

线程 Mythread：

```
void *MyThread(void *arg)
{
    int *num;
    num=(int *)arg;
    printf("create parameter is %d \n",*num);
    return (void *)0;
}
```

(3)函数 pthread_join()的用法

函数 pthread_join()用来等待一个线程的结束。函数原型为：

```
extern int pthread_join __P ((pthread_t __th, void **__thread_return));
```

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。一个线程的结束有两种途径，一种是像上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数 pthread_exit 来实现。它的函数原型为：

```
extern void pthread_exit __P ((void *__retval)) __attribute__ ((__noreturn__));
```

唯一的参数是函数的返回代码，只要 pthread_join 中的第二个参数 thread_return 不是 NULL，这个值将被传递给 thread_return。最后要说明的是，一个线程不能被多个线程等待，

否则第一个接收到信号的线程成功返回，其余调用 pthread_join 的线程则返回错误代码 ESRCH。

在这一节里，我们编写了一个最简单的线程，并掌握了最常用的三个函数 pthread_create，pthread_join 和 pthread_exit。下面，我们来了解线程的一些常用属性以及如何设置这些属性。

实验七 利用信号量实现进程控制

实验学时：2 学时
实验类型：验证、设计型

一、实验目的

学习 UNIX 类操作系统信号量机制，编写 Linux 环境下利用信号量实现进程控制的方法，掌握相关系统调用的使用方法。

二、实验内容

创建 4 个线程，其中两个线程负责从文件读取数据到公共的缓冲区，另两个线程从缓冲区读取数据作不同的处理（加和乘运算）。使用信号量控制这些线程的执行。

提示：参见“五、补充材料”中的相关系统调用的基本用法。

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、实验报告要求

- 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）
- 实验结果及分析
- 运行结果
- 感想
- 参考资料

五、补充材料

信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。当公共资源增加时，调用函数 sem_post（）增加信号量。只有当信号量值大于 0 时，才能使用公共资源，使用后，函数 sem_wait（）减少信号量。函数 sem_trywait（）和函数 pthread_ mutex_trylock（）起同样的作用，它是函数 sem_wait（）的非阻塞版本。下面我们逐个介绍和信号量有关的一些函数，它们都在头文件 /usr/include/semaphore.h 中定义。

信号量的数据类型为结构 sem_t，它本质上是一个长整型的数。函数 sem_init（）用来初始化一个信号量。它的原型为：

```
extern int sem_init __P ((sem_t *__sem, int __pshared, unsigned int __value));
```

sem 为指向信号量结构的一个指针；pshared 不为 0 时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value 给出了信号量的初始值。

函数 sem_post(sem_t *sem)用来增加信号量的值。当有线程阻塞在这个信号量上时，

调用这个函数会使其中的一个线程不在阻塞，选择机制同样是由线程的调度策略决定的。

函数 sem_wait(sem_t *sem)被用来阻塞当前线程直到信号量 sem 的值大于 0，解除阻塞后将 sem 的值减一，表明公共资源经使用后减少。函数 sem_trywait (sem_t *sem)是函数 sem_wait（）的非阻塞版本，它直接将信号量 sem 的值减一。

函数 sem_destroy(sem_t *sem)用来释放信号量 sem。