计算机操作系统原理
实验指导书

孙英华

计算机科学技术学院

目　　录

【预备知识】　Linux 程序设计环境

一. Linux 编程语言

- 高级程序设计语言
  - C/C++, Java, Fortran…
  - ELF binary format

    Excutable and Linkable Format
- 脚本语言
  - Shell: sh/bash, csh, ksh
  - Perl, Python, tcl/tk, sed, awk…

二. Linux 开发工具

- 编译工具 GCC
  - GNU C Compiler -> GNU Compiler Collection
  - The gcc command: Front end
- 调试工具 GDB
  - GNU Debugger
  - The gdb command
  - xxdgb, ddd…
- 管理项目 GUN Make

三. Linux 集成开发环境

- IDE
  - Emacs/xemacs
  - Kdevelop
  - Eclipse
  - Kylix3
- Command line
  - Editor:　vi/vim/gvim, emacs/xemacs, pico
  - Source Reader:　source navigator; vi/emacs+ ctags/etags
  - Configure Tools: automake, autoconf, m4

实验一　Linux 常用命令和编辑环境 vi 的使用

［实验目的］

1. 熟悉了解 Linux 的登录和退出；
2. 了解 Linux 的命令及使用格式；
3. 掌握 Linux 的常用基本操作命令；
4. 练习并掌握 Linux 提供的 vi 编辑器；
5. 了解 Makefile 文件的编写和使用。

［实验内容］

1. 熟悉如何登录 Linux;
2. 了解 Linux 的命令及使用格式；
3. 掌握 Linux 常用基本命令如：man, ls, who, cp, rm, mv, cd...等命令；
4. 掌握 vi 编辑器的用法。

［实验指导］

一、 Linux 的登录与退出

1．登录

Linux 是一个多用户操作系统，可以有多个用户同时使用一台计算机，运行各自的应用程序。为了区分各个用户，每个用户都拥有自己独立的用户帐号。用户在使用 Linux 时以各自的用户名登录。登录提示为 login:

在 bash shell 下”#”为 root 用户的命令行提示符，”$”为一般用户的命令行提示符。

Login:　　(输入 username)

Password:　(输入密码)

2．退出

在 Linux 系统提示符下$下，输入 logout、exit 或 shutdown 或按 CTRL+ALT+DEL 退出系统。

例：$logout

3. 关闭系统

关机一般由 root 用户进行。关机的命令：

＃halt(或＃shutdown)

二、 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，如果命令出错或未完全完成，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。

三、 Linux 常用命令：

（一）帮助命令：

1．man 获取相关命令的帮助信息

eg：man dir 可以获取关于 dir 的使用信息。

2．info 获取相关命令的详细使用方法

eg：info info 可以获取如何使用 info 的详细信息。

注：按 q 键或者 ctrl+c 退出,在 linux 下可以使用 ctrl+c 终止当前程序运行

（二）文件操作：

1．cat 和 more： 显示文件内容和合并多个文件

两个命令所不同的是:cat 把文件内容一直打印出来,而 more 则分屏显示。

Eg：cat>1.c //就可以把代码粘帖到 1.c 文件里，按 ctrl+d 保存代码。

cat 1.c 或 more 1.c //都可以查看里面的内容,与 DOS 的 type 命令同

2．clear 清屏

clear:清屏,相当于 DOS 下的 cls

3．cp 拷贝文件

cp source target 将文件 source 复制为 target eg:cp 1.c netseek/2.c //将 1.c 拷到 netseek 目录下命名为 2.c

cp –av source_dir target_dir 将整个目录复制，两目录完全一样

cp –fr source_dir target_dir 将整个目录复制，并且是以非链接方式复制，当 source 目录带有符号链接时，两个目录不相同

4．diff 比较两个文本文件，列出行不同之处

diff file1 file2 比较文件 1 与文件 2 的内容是否相同,如果是文本格式的文件，则将不相同的内容显示，如果是二进制代码则只表示两个文件是不同的。

5．grep 文本内容搜索

eg: grep success 查找当前目录下面所有文件里面含有 success 字符的文件

6．find 文件或者目录名以及权限属主等匹配搜索

7．rm 删除文件

rm file 删除某个文件 eg: rm 1.c //将 1.c 这个文件删除

---

8. find 查找文件

find -name /path file 在/path 目录下查找看是否有文件 file

9．mv 文件或目录的移动或更名

mv source target 将文件或者目录 source 更名为 target

mv filename1 filename2 //将 filename1 改名为 filename2

mv qib.tgz ../qib.tgz //移到上一级目录

---

10. chmod 改变文件或目录的权限

chmod 一位 8 进制数 filename

eg: chmod u+x filenmame //只想给自己运行，别人只能读

//u 表示文件主人, g 表示文件文件所在组. o 表示其他人;r 表可读,w 表可写,x 表可以运行

（三）磁盘操作命令：

1. ls 查看目录或者文件的属,列举出任一目录下面的文件，相当于 DOS 中的 dir 命令。

ls 以默认方式显示当前目录文件列表

ls -a 显示所有文件包括隐藏文件

ls -l 显示文件属性，包括大小，日期，符号连接，是否可读(r)可写(w)及是否可执行(x)

---

2. cd 命令: 改变当前目录 pwd 查看当前所在目录完整路径

cd dir 切换到当前目录下的 dir 目录

cd .. 切换到上一级目录

cd ~ 退出当前目录，切换到用户目录, 比如是 root 用户，则切换到/root 下

3．mkdir 创建目录

eg: mkdir netseek //创建 netseek 这个目录

---

4. rmdir 删除目录 //使用该命令要确保目录下已无任何文件

rm -rf dir 删除当前目录下叫 dir 的整个目录

---

5. diff 比较目录

diff dir1 dir2 比较目录 1 与目录 2 的文件列表是否相同，但不比较文件的实际内容，不同则列出。

---

6. pwd 显示当前工作目录

7. fdisk /dev/hda 就像执行了 dos 的 fdisk 一样

---

8. mount/umount 加载文件系统/卸载文件系统

mount [参数] 要加载的设备载入点

mount -t ext2 /dev/hda1 /mnt 把/dev/hda1 装载到 /mnt 目录

cd /mnt/cdrom 进入光盘目录

umount /mnt 将/mnt 目录卸载，/mnt 目录必须处于空闲状态

（四）系统操作命令：

1. alias　设置别名
2. date　显示或设置系统时间与日期
3. passwd 可以设置口令

---

4. finger 查找并显示用户信息

eg: finger　//查看所用用户的使用资料

finger root　//查看 root 的资料

---

5. Who,Whoami,whereis,which,id

Who　列出正在使用系统的用户

whoami　确认自己身份.

Whereis　查询命令所在目录以及帮助文档所在目录.

Which　查询该命令所在目录(类似 whereis)

Id　打印出自己的 UID 以及 GID.(UID:用户身份唯一标识.GID:用户组身份唯一标识.每一个用户只能有一个唯一的 UID 和 GID)

eg: whoami　//显示你自己登陆的用户名

whereis bin　//显示 bin 所在的目录，将显示为：/usr/local/bin

which bin　//显示 bin 所在的目录，将显示为：/usr/local/bin

---

6. su　root　切换到超级用户
7. reboot　重启计算机

（五）压缩与备份:

1．tar 解压命令

Eg1:　tar xfzv file.tgz　将文件 file.tgz 解压

2．gzip/gunzip　.gz 文件的压缩/解压缩程序

gzip directory.tar　将覆盖原文件生成压缩的 directory.tar.gz

gunzip directory.tar.gz　覆盖原文件解压生成不压缩的 directory.tar。

四、文本编辑器 vi 的使用

进入 vi 的命令：

vi file　打开或新建编辑文件 file，并将光标置于第一行首

vi –r file　在上次正用 vi 编辑时发生系统崩溃，恢复 file

vi file1….file　打开多个文件，依次进行编辑

---

vi 分为编辑状态和命令状态。输入命令要先按 ESC，退出编辑状态，然后输入命令。

常用行方式命令：

:e file　打开文件 file 进行编辑

:w　保存当前文件

:w!　不询问方式写入文件

:x　　保存当前文件并退出
:x!　　退出不保存当前文件
:q　　退出 vi
:q!　　不保存文件并退出
:r file　读文件 file
:r! command　　将命令 command 的输出结果放到当前行
:!command　　执行 shell 命令 command
i 进入编辑插入状态
ESC 退出编辑状态

五、Makefile 的编写

假设我们有这样的一个程序,由以下五个文件组成，源代码如下:

```
/* main.c */
#include "mytool1.h"
#include "mytool2.h"
int main(int argc,char **argv)
{
mytool1_print("hello mytool1!");
mytool2_print("hello mytool2!");
return 0;
}

/* mytool1.h */
#ifndef _MYTOOL_1_H
#define _MYTOOL_1_H
void mytool1_print(char *print_str);
#endif

/* mytool1.c */
#include "mytool1.h"
#include <stdio.h>
void mytool1_print(char *print_str)
{
printf("This is mytool1 print %s\n",print_str);
return 0;
}

/* mytool2.h */
#ifndef _MYTOOL_2_H
#define _MYTOOL_2_H
void mytool2_print(char *print_str);
#endif

/* mytool2.c */
```

```
#include "mytool2.h"
#include <stdio.h>
void mytool2_print(char *print_str)
{
printf("This is mytool2 print %s\n",print_str);
return 0;
}
```

当然由于这个程序是很短的，我们可以这样来编译

```
gcc -c main.c
gcc -c mytool1.c
gcc -c mytool2.c
gcc -o main main.o mytool1.o mytool2.o
```

这样的话我们也可以产生 main 程序，而且也不是很麻烦。但是我们考虑一下，如果修改了其中的一个文件(比如说 mytool1.c)那么难道我们还要重新输入上面的命令？也许你会说,这个很容易解决啊，我写一个 SHELL 脚本，让它帮我去完成不就可以了。是的，对于这个程序来说，是可以起到作用的。但是当我们把事情想的更复杂一点，如果我们的程序有几百个源程序的时候，难道也要编译器重新一个一个的去编译？

为此，聪明的程序员们想出了一个很好的工具来做这件事情，这就是 make。我们只要执行 make，就可以把上面的问题解决掉。在我们执行 make 之前，我们要先编写一个非常重要的文件—Makefile。对于上面的那个程序来说，可能的一个 Makefile 的文件是:

```
# 这是上面那个程序的 Makefile 文件
main:main.o mytool1.o mytool2.o
    gcc -o main main.o mytool1.o mytool2.o
main.o:main.c mytool1.h mytool2.h
    gcc -c main.c
mytool1.o:mytool1.c mytool1.h
    gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h
    gcc -c mytool2.c
```

有了这个 Makefile 文件，不论我们什么时候修改了源程序当中的文件，我们只要执行 make 命令，我们的编译器都只会去编译和我们修改的文件有关的文件，其它的文件她连理都不想去理的。

下面我们学习 Makefile 是如何编写的.

在 Makefile 中由#开始的行都是注释行。Makefile 中最重要的是描述文件的依赖关系的说明。一般的格式是:

target: components

TAB rule

（注：上面 TAB 就是按键盘上的 TAB 键）

第一行表示的是依赖关系。第二行是规则。

比如说我们上面的那个 Makefile 文件的第二行

```
main:main.o mytool1.o mytool2.o
```

表示我们的目标(target)main 的依赖对象(components) 是 main.o mytool1.o mytool2.o 当依赖的对象在目标修改后修改的话，就要去执行规则一行所指定的命令，就象我们的上面那个 Makefile 第三行所说的一样要执行

```
gcc -o main main.o mytool1.o mytool2.o
```

Makefile 有三个非常有用的变量，分别是$@、$^、$<。代表的意义分别是:
$@--目标文件，
$^--所有的依赖文件，
$<--第一个依赖文件。
如果我们使用上面三个变量，那么我们可以简化我们的 Makefile 文件为:

```
# 这是简化后的 Makefile
main:main.o mytool1.o mytool2.o
    gcc -o $@ $^
main.o:main.c mytool1.h mytool2.h
    gcc -c $<
mytool1.o:mytool1.c mytool1.h
    gcc -c $<
mytool2.o:mytool2.c mytool2.h
    gcc -c $<
```

经过简化后我们的 Makefile 是简单了一点，不过人们有时候还想简单一点。这里我们学习一个 Makefile 的缺省规则

```
..c.o:
gcc -c $<
```

这个规则表示所有的.o 文件都是依赖与相应的.c 文件的。例如 mytool.o 依赖于 mytool.c，这样 Makefile 还可以变为:

```
# 这是再一次简化后的 Makefile
main:main.o mytool1.o mytool2.o
    gcc -o $@ $^
..c.o:
    gcc -c $<
```

如果想知道更多的关于 Makefile 规则可以查看相应的文档.

其它 Linux 命令参考“附录 2：Linux 常用命令”

实验二　编译器 gcc 和调试器 gdb 的使用

【实验目的】

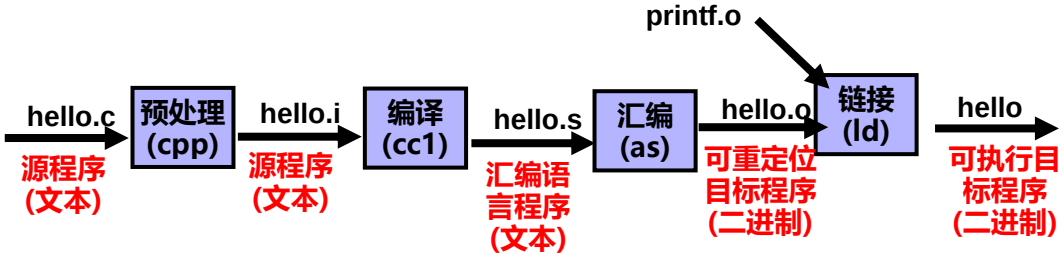
1. 掌握 Linux 操作系统下最常用的 C 语言编译器 gcc 的使用；
2. 掌握 Linux 操作系统下最常用的代码调试器 gdb 的使用；
3. 掌握调试代码的基本方法，如观察变量、设置断点等。

【实验预备知识】

(1) 阅读在线帮助命令 man gcc 的内容，了解 gcc 的基本使用
(2) 阅读在线帮助命令 man gdb 的内容，了解 gdb 的基本使用
(3) gcc 简介

GCC 是一套由 GNU 项目开发的编程语言编译器，可处理 C 语言、C++、Fortran、Pascal、Objective-C、Java 等等。GCC 通常是跨平台软件的编译器首选。gcc 是 GCC 套件中的编译驱动程序名。

下图是用 GCC 命令将一个 hello.c 源程序转换为可执行目标代码的过程。



gcc –E hello.c –o hello.i

gcc –S hello.i –o hello.s

gcc –c hello.s –o hello.o

gcc hello.o –o hello

Unix 上使用的 C 语言编译器 cc，在 Linux 上的派生就是 gcc。通常，在使用 vi 编写完源程序之后，返回到 shell 下面，使用 gcc 对源程序进行编译的命令是：

gcc 源程序

其中，“源程序”即为你编写的以.c 为扩展名的 C 语言源代码文件。

如果源代码没有语法错误，使用以上命令编译，会在当前目录下生成一个名为 a.out 的可执行文件。如果源代码有语法错误，则不会生成任何文件，gcc 编译器会在 shell 中提示你错误的地点和类型。

也可以使用以下方法编译源代码文件，生成自命名的可执行文件：

gcc 源文件 -o 自命名的文件名

执行当前目录下的编译生成的可执行文件，使用以下格式：

./可执行文件名

当使用 gcc 编译你写的程序源代码的时候，可能会因为源代码存在语法错误，编译无法进行下去，这时候，就可以使用调试器 gdb 来对程序进行调试。

(4) gdb 简介

Linux 包含了一个叫 gdb 的 GNU 调试程序。 gdb 是一个用来调试 C 和 C++ 程序的强力调试器。它使你能在程序运行时观察程序的内部结构和内存的使用情况。以下是 gdb 所提供的一些功能：

- 能监视你程序中变量的值。
- 能设置断点以使程序在指定的代码行上停止执行。

- 能一行行的执行你的代码。

在命令行上键入 gdb 并按回车键就可以运行 gdb 了，如果一切正常的话，gdb 将被启动并且你将在屏幕上看到类似的内容：

```
GDB is free software and you are welcome to 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.
GDB 4.14 (i486-slakware-linux), Copyright 1995 Free Software Foundation, Inc.
(gdb)
```

当启动 gdb 后，能在命令行上指定很多的选项。你也可以以下面的方式来运行 gdb：

gdb <fname>

当你用这种方式运行 gdb ，就能直接指定想要调试的程序。这将告诉 gdb 装入名为 fname 的可执行文件。你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件，或者与一个正在运行的程序相连。你可以参考 gdb 指南页或在命令行上键入 gdb -h 得到一个有关这些选项的说明的简单列表。

为调试编译代码(Compiling Code for Debugging)。为了使 gdb 正常工作，你必须使你的程序在编译时包含调试信息。调试信息包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb 利用这些信息使源代码和机器码相关联。在编译时用 -g 选项打开调试选项。

gdb 支持很多的命令使你能实现不同的功能。这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令，下表列出了你在用 gdb 调试时会用到的一些命令。想了解 gdb 的详细使用请参考 gdb 的指南页（man gdb）。

基本 gdb 命令的功能描述如下：

命　令	描　述
file	装入想要调试的可执行文件
kill	终止正在调试的程序
list	列出产生执行文件的源代码的一部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
quit	终止 gdb
watch	使你能监视一个变量的值而不管它何时被改变
break	在代码里设置断点，这将使程序执行到这里时被挂起
make	使你能不退出 gdb 就可以重新产生可执行文件
shell	使你能不离开 gdb 就执行 Linux shell 命令

gdb 支持很多与 Linux shell 程序一样的命令编辑特征。你能象在 bash 或 tcsh 里那样按 Tab 键让 gdb 帮你补齐一个唯一的命令，如果不唯一的话 gdb 会列出所有匹配的命令。你也能用光标键上下翻动历史命令。

【实验内容】

将下面的程序输入到一个文件名字为 test.c 的磁盘文件中，利用调试程序找出其中的错误，修改后存盘。该程序的功能是显示一个简单的问候语，然后用反序方式将它列出。

```
#include   <stdio.h>
main ()
{
   char my_string[] = "hello there";
   my_print (my_string);
   my_print2 (my_string);
}
void my_print (char *string)
{
   printf ("The string is %s\n", string);
}
void my_print2 (char *string)
{
   char *string2;
   int size, i;
   size = strlen (string);
   string2 = (char *) malloc (size + 1);
   for (i = 0; i < size; i++)
   string2[size - i] = string[i];
   string2[size+1] = `\0';
   printf ("The string printed backward is %s\n", string2);
}
```

【设计参考】

对于给出的程序代码，用下面的命令编译它：

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

执行编译得到的程序

```
./test
```

显示如下结果：

```
The string is helle there
The string printed backward is
```

输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的。我们所设想的输出应该是：

The string printed backward is ereht olleh

由于某些原因，my_print2 函数没有正常工作。让我们用 gdb 看看问题究竟出在哪儿，先键入如下命令：

```
gdb test
```

如果你在输入命令时忘了把要调试的程序作为参数传给 gdb ，你可以在 gdb 提示符下用 file 命令来载入它：

```
file test
```

这个命令将载入 test 可执行文件就象你在 gdb 命令行里装入它一样。

这时你能用 gdb 的 run 命令来运行 test 了：

```
run
```

当它在 gdb 里被运行后结果大约会象这样：

```
Starting program: /home/user1/test
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 shell 中运行的结果一样。问题是，为什么反序打印没有工作？为了找出症结所在，我们可以在 my_print2 函数的 for 语句后设一个断点，具体的做法是在 gdb 提示符下键入 list 命令三次，列出源代码：

```
list
list
list
```

第一次键入 list 命令的输出如下：

```
1             #include   <stdio.h>
2             main ()
3             {
4                 char my_string[] = "hello there";
5                 my_print (my_string);
6                 my_print2 (my_string);
7             }
8           my_print (char *string)
9           {
10        printf ("The string is %s\n", string);
```

如果按下回车，gdb 将再执行一次 list 命令，给出下列输出：

```
11          }
12          my_print2 (char *string)
13          {
14              char *string2;
15              int size, i;
16              size = strlen (string);
17              string2 = (char *) malloc (size + 1);
18              for (i = 0; i < size; i++)
19                  string2[size - i] = string[i];
20        string2[size+1] =  '\0' ;
```

再按一次回车将列出 test.c 程序的剩余部分：

```
21              printf ("The string printed backward is %s\n", string2);
22          }
```

根据列出的源程序，你能看到要设断点的地方在第 19 行，在 gdb 命令行提示符下键入如下命令设置断点：

```
break 19
```

gdb 将作出如下的响应：

```
Breakpoint 1 at 0x139: file test.c, line 19
```

现在再键入 run 命令，将产生如下的输出：

```
Starting program: /home/user1/test
The string is hello there
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at test.c :19
19  string2[size-i]=string[i];
```

你能通过设置一个观察 string2[size - i] 变量的值的观察点来看出错误是怎样产生的，做法是键入：

```
watch string2[size - i]
```

gdb 将作出如下回应：

```
Watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步的执行 for 循环了：

```
next
```

经过第一次循环后，gdb 告诉我们 string2[size - i] 的值是‘h’，gdb 用如下的显示来告诉你这个信息：

```
Watchpoint 2, string2[size - i]
Old value = 0  '\000'
New value = 104  'h'
my_print2(string = 0xbfffdc4 "hello there") at test.c:18
18 for (i=0; i<size; i++)
```

这个值正是期望的。后来的数次循环的结果都是正确的。当 i=10 时，表达式 string2[size - i] 的值等于‘e’，size - i 的值等于 1，最后一个字符已经拷到新串里了。

如果你再把循环执行下去，你会看到已经没有值分配给 string2[0] 了，而它是新串的第一个字符，因为 malloc 函数在分配内存时把它们初始化为空(null)字符，所以 string2 的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出了。

现在找出了问题出在哪里，修正这个错误是很容易的。你得把代码里写入 string2 的第一个字符的的偏移量改为 size - 1 而不是 size。 这是因为 string2 的大小为 12， 但起始偏移量是 0，串内的字符从偏移量 0 到 偏移量 10，偏移量 11 为空字符保留。

为了使代码正常工作有很多种修改办法。一种是另设一个比串的实际大小小 1 的变量。这是这种解决办法的代码：

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

```
    char my_string[] = "hello there";
    my_print (my_string);
    my_print2 (my_string);
}
my_print (char *string)
{
    printf ("The string is %s\n", string);
}
my_print2 (char *string)
{
    char *string2;
    int size, size2, i;
    size = strlen (string);
    size2 = size -1;
    string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
        string2[size2 - i] = string[i];
    string2[size] = `\0';
    printf ("The string printed backward is %s\n", string2);
}
```

实验三　多进程多线程的并发实践

【实验目的】

1. 辨析程序、进程、线程等概念，从实现角度明确程序、进程、线程的区别。
2. 加深理解进程并发执行的概念，编写多进程并发程序。
3. 学习线程库函数的使用，编写多线程并发程序。
4. 学习使用相关函数，实现进程和线程之间的同步和互斥。

【实验预备知识】

一．Linux 进程创建方法

- 在终端输入命令，由 shell 进程创建一个新进程
- 进程创建函数
  - pid_t fork(void);
  - pid_t vfork(void);
  - int clone(int (*fn)(void * arg), void *stack, int flags, void * arg)；
- 三函数都调用同一内核函数 do_fork( ) [/kernel/fork.c]

【注】内核拓展： do_fork()内核函数原型

- 函数调用形式
  - do_fork(unsigned long clone_flag, unsigned long stack_start, struct pt_regs *regs, unsigned long stack_size, int _user *parent_tidptr, int _user *child_tidptr)；
- 参数说明
  - clone_flag：子进程创建相关标志
  - stact_start：将用户态堆栈指针赋给子进程的 esp
  - regs：指向通用寄存器值的指针
  - stack_size：未使用（总设为 0）
  - parent_tidptr：父进程的用户态变量地址，若需父进程与新轻量级进程有相同 PID，则需设置 CLONE_PARENT_SETTID
  - child_tidptr：新轻量级进程的用户态变量地址，若需让新进程具有同类进程的 PID ，需设置 CLONE_CHILD_SETTID

子进程创建 CLONE 参数标志说明

1.fork() 函数说明

pid_t fork(void)

fork()会产生一个新的子进程。该函数包含于头文件 unistd.h 中。其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码、组代码、环境变量、已打开的文件代码、工作目录和资源限制等。Linux 使用 copy-on-write(COW)技术，只有当其中一进程试图修改欲复制的空间时才会做真正的复制动作，由于这些继承的信息是复制而来，并非指相同的内存空间，因此子进程对这些变量的修改和父进程并不会同步。此外，子进程不会继承父进程的文件锁定和未处理的信号。注意，Linux 不保证子进程会比父进程先执行或晚执行，因此编写程序时要留意死锁或竞争条件的发生。

返回值，如果 fork()调用成功则在父进程会返回新建立的子进程代码(PID)，而在新建立的子进程中则返回 0。如果 fork() 失败则直接返回-1，失败原因存于 errno 中。失败的原因有三个：

1) 系统内存不够；

2) 进程表满（容量一般为 200~400）；

3) 用户的子进程太多（一般不超过 25 个）。

错误代码：EAGAIN 内存不足；ENOMEM 内存不足，无法配置核心所需的数据结构空间。

2.getpid() 函数说明

#include<unistd.h>

pid_t getpid(void)

getpid（）用来取得目前进程的进程识别码。许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。返回值：目前进程的进程识别码。

3. getppid() 函数说明

#include<unistd.h>

pid_t getppid(void)

getppid()用来取得目前进程的父进程识别码。返回值：目前进程的父进程识别码。

4. exit() 函数说明

#include<stdlib.h>

void exit(int status)

exit()用来正常终结目前进程的执行。并通过参数 status 把进程的结束状态返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。如果不关心返回状态的话，status 参数也可以使用 0 。

5. vfork()说明

- vfork()创建的子进程与父进程共享地址空间
  - ✓ 子进程作为父进程的一个单独线程在其地址空间运行
  - ✓ 子进程从父进程继承控制终端、信号标志位、可访问的主存区、环境变量和其他资源分配
  - ✓ 子进程对虚拟空间任何数据的修改都可为父进程所见
  - ✓ 父进程将被阻塞，直到子进程调用 execve()或 exit()

❖ 与 fork()的关系

- 功能相同，但 vfork()但不拷贝父进程的页表项
- 子进程只执行 exec()时，vfork()为首选

6. clone 函数原型

- int clone(int (*fn)(void *), void *child_stack, int clone_flag, void *arg);

❖ 参数说明

- fn：待执行的程序
- child_stack：进程所使用的堆栈
- clone_flag：由用户指定，可以是多个标志的组合
- arg：执行 fn 所需的参数

❖ 功能

- 创建轻量级进程(LWP)的系统调用
- 通过 clone_flag 控制

clone()函数的常用 CLONE 标志：

❖ CLONE_VM：父子进程共享内存描述符及所有页表

❖ CLONE_FS：父子进程共享文件系统信息

❖ CLONE_FILES：父子进程共享文件描述符表

❖ CLONE_SIGHAND：父子进程共享信号描述符

❖ CLONE_PTRACE：若父进程被跟踪，子进程也被跟踪

❖ CLONE_PARENT：父进程的 real_parent 登记为子进程的 parent 和 real_parent

❖ CLONE_THREAD：子进程加入父进程的线程组

❖ CLONE_STOPPED：创建新进程但不运行之

7. exec()函数

❖ 大多情况下子进程从 fork 返回后都调用 exec()函数来执行新的程序

- 进程调用 exec()函数时，该进程完全由新程序替代，新程序从 main 开始执行
- exec()并不创建新进程，前后进程 ID 不变，但用另外一个程序替代当前进程的正文、数据、堆栈等
- 函数原型
  - ✓ int execl(const char *path, const char *arg, ...);
  - ✓ int execlp(const char *file, const char *arg, ...);
  - ✓ int execle(const char *path, const char *arg , ..., char* const envp[ ]);
  - ✓ int execv(const char *path, char *const argv[ ]);
  - ✓ int execve(const char *filename, char *const argv [ ], char *const envp[ ]);
  - ✓ int execvp(const char *file, char *const argv[ ]);

8. 函数原型

- **pid_t wait(int *status);**
- **pid_t waitpid(pid_t pid, int *status, int options);**

❖ 说明

- 均通过 wait4()系统调用实现
- 进程终止时，会向父进程发送 SIGCHLD 信号

❖ 调用 wait( )和 waitpid( )的进程的可能状态

- 阻塞
  - ✓ 如果子进程还在运行
- 正常返回
  - ✓ 返回子进程的终止状态（其中一个子进程终止)
- 出错返回
  - ✓ 没有子进程

二. Linux 的线程机制

❖ 其他多线程操作系统

- 通过两种数据结构来表示线程和进程：进程表项和线程表项
- 每个进程表项可以指向若干个线程表项，调度器在进程的时间片内再调度线程

❖ Linux 系统

- 从内核角度，没有线程概念，所有线程当进程来实现
- 线程被视为与其他进程共享某些资源的进程，都有自己的进程描述符 task_struct
- 内核没有针对线程的调度算法或数据结构

1. pthread_create 函数创建用户级线程

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr , void *(*start_rtn) (void*)，void * arg);

❖ 参数说明

- thread: 待创建线程的 id 指针
- attr: 创建线程时的线程属性
- v(*start_rtin)(void*): 返回值是 void*类型的指针函数

- arg: start_routine 的参数

❖ 返回值

- 成功返回 0
- 失败返回错误编号
  - ✓ EAGAIN：表示系统限制创建新的线程，如线程数目过多
  - ✓ EINVAL：代表线程属性值非法

2. pthread 介绍

❖ 基于 POSIX 标准的线程编程接口

- 包括一个 pthread.h 头文件和一个线程库
- 编译方法
  - ✓ gcc –g **.c -o *** –lpthread

❖ 功能

- 线程管理
  - ✓ 支持线程创建/删除、分离/联合，设置/查询线程属性
- 互斥
  - ✓ 处理互斥，称为“mutex”
    - ❖ 创建/销毁、加锁/解锁互斥量，设置/修改互斥量属性
- 条件变量
  - ✓ 支持基于共享互斥量的线程间通信
    - ❖ 建立/销毁、等待/触发特定条件变量，设置/查询条件变量属性

pthread 变量类型的命名规则

前缀形式	功能组
pthread_	线程自身及其他子例程
pthread_attr_	线程属性对象
pthread_mutex_	互斥变量
pthread_mutexattr_	互斥属性对象
pthread_cond_	条件变量
pthread_condattr_	条件属性对象
pthread_key_	特定线程数据键

3. 返回当前线程的 id 号

pthread_t pthread_self (void);

功能： 用于返回当前线程的 ID。

4. 线程终止

❖ 正常终止

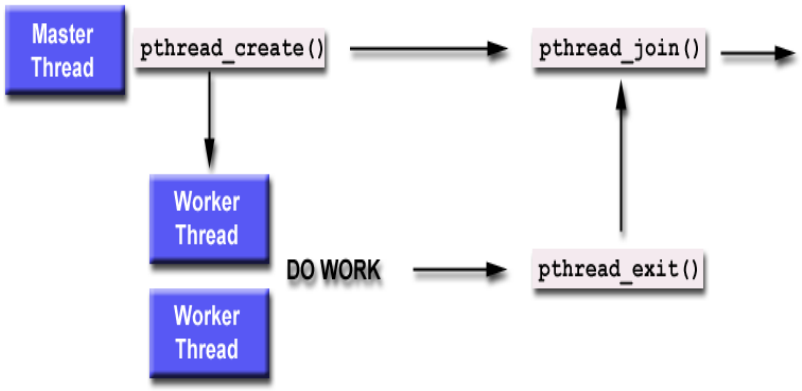
- 方法 1：线程自己调用 pthread_exit()
  - ✓ void pthread_exit(void *rval_ptr);
  - ✓ rval_ptr 线程退出返回的指针，进程中其他线程可调用 pthread_join

（）访问到该指针。

- 方法 2：线程函数执行 return。

5. 线程同步终止

- 函数原型
  - int pthread_join(pthread_t thread, void ** rval_ptr);
- 功能
  - 调用者将挂起并等待新线程终止
  - 当新线程调用 pthread_exit()退出或者 return 时，进程中的其他线程可通过 pthread_join()获得线程的退出状态
- 使用约束
  - 一个新线程仅仅允许一个线程使用该函数等待它终止
  - 被等待的线程应该处于可 join 状态，即非 DETACHED 状态
- 返回值
  - 成功结束返回值为 0,否则为错误编码
- 说明
  - 类似于 waitpid()



【必做实验内容】

一、多进程并发程序设计

编写程序，利用 fork()产生两个子进程，首先显示一下两个子进程及父进程的进程标识符；然后让父进程显示 1-26 个数字，子进程 1 显示 26 个大写字母，子进程 2 显示 26 个小写字母。让大小写字母及数字是夹杂交错输出的。修改程序，让两个子进程夹杂输出结束后，父进程输出开始，参见图 3-1。

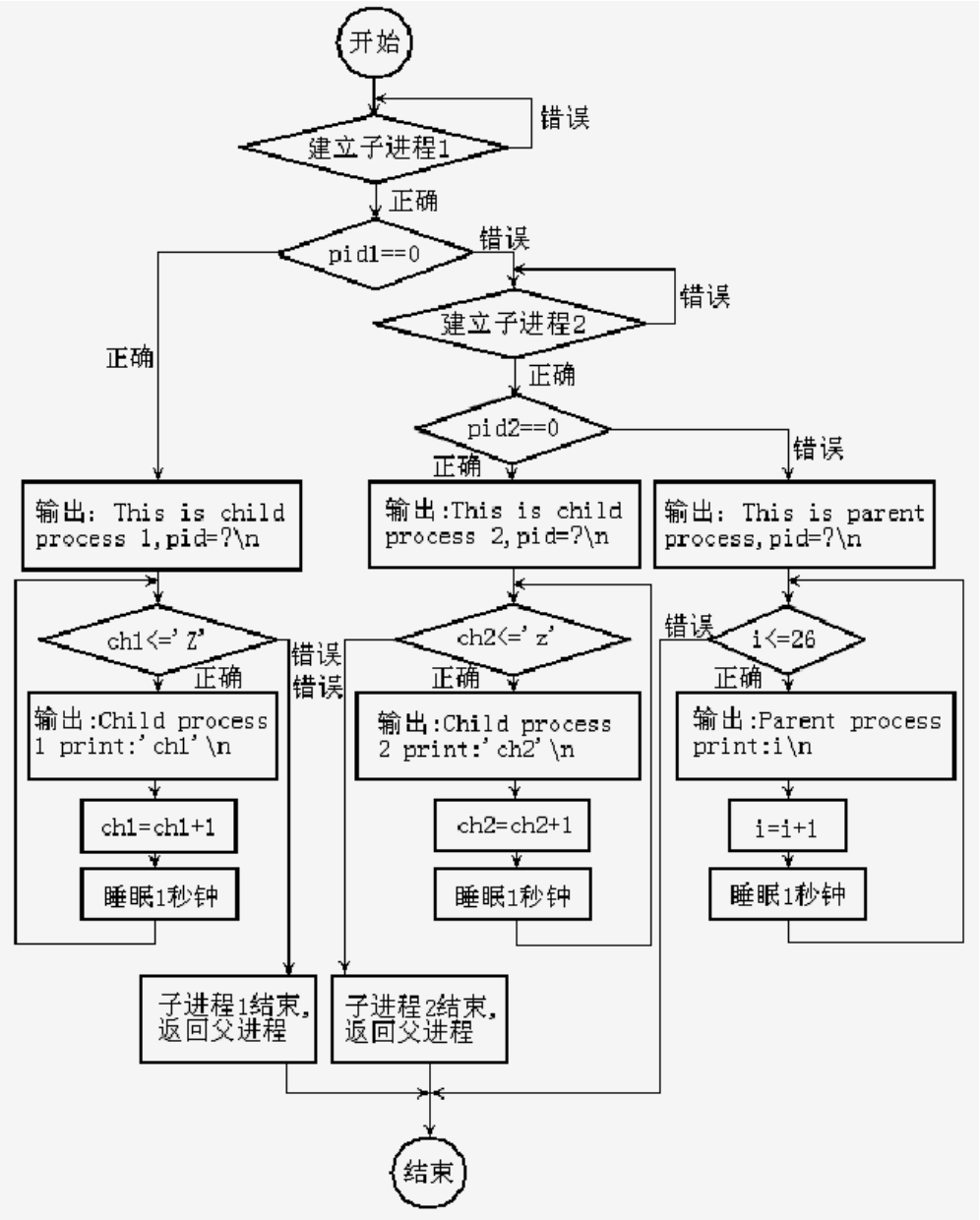


图 3-1　多进程并发处理流程

二、多线程程序并发设计

编写程序，利用 pthread_create ()创建三个线程，每个线程执行实现不同功能的子函数。线程 1 显示 1-26 个数字，线程 2 显示 26 个大写字母，线程 3 显示 26 个小写字母。三个线程并发执行完成后，父进程输出“三个线程结束”。

【实验三可参考程序实例】

注：下页参考程序供没有并发程序设计基础的同学参考。

1. fork()函数编程实例

拓展：多进程并发实例

三个进程并发运行的结果

```
yushengxian@yushengxian-PC:~/Code$ ./test
parent:1
child1:A
child2:a
child1:B
parent:2
child2:b
child1:C
parent:3
child2:c
parent:4
child2:d
child1:D
parent:5
child2:e
child1:E
parent:6
child2:f
child1:F
parent:7
child1:G
child2:g
```

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>

int main()
{
    int pid1,pid2;
    if(pid1 = fork()){
        if(pid2 = fork()){
            for(int i = 0; i < 26; i++){
                printf("parent:%d\n",i+1);
                sleep(1);
            }
        }
        else {
            for(int j = 0; j < 26; j++){
                printf("child1:%c\n",(char)(j+'A'));
                sleep(1);
            }
        }
    }
    else {
        for(int k = 0; k < 26; k++){
                printf("child2:%c\n",(char)(k+'a'));
                sleep(1);
            }
    }
}
```

2. vfork()和 exec 类函数编程实例

课外拓展：vfork()与exec类函数编程示例

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

int main(int argc,char *argv[])
{
        char *arg[] = {"ls","-a",NULL};
        int pid;
        if(pid = vfork()){
                printf("This is vfork\n");
                execl("/bin/ls","ls","-a",NULL);
                printf("This is parent\n");
        }
        else{
                printf("This is child\n");
        }
}
```

输出结果

```
yushengxian@yushengxian-PC:~/Code$ ./fork
This is child
This is vfork
.  ..  fork  fork.c  test  test.c
yushengxian@yushengxian-PC:~/Code$ ls
fork  fork.c  test  test.c
```

拓展：学习exec类函数，编写多进程并发程序

3. pthread_create()函数编程实例

注：在实际应用程序中，多线程并发通常伴随着线程之间的同步和互斥。下例中同时示例了采用互斥锁实现线程互斥的实现方法。


```c
/*Linux多线程编程之互斥锁的使用*/
/*双十一互斥锁实例*/
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
int kucun = 10;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

//pthread_mutex_lock(&mutex);  p操作
//pthread_mutex_unlock(&mutex);  v操作

void *func(void *arg)
{
        int threadno = *(int*)arg;
        int i = 0;
        while( kucun > 0)
        {
                pthread_mutex_lock(&mutex);
                printf("    同学 %d 抢到了倒数第 %d 件货 \n", threadno, kucun);
                kucun--;
                pthread_mutex_unlock(&mutex);
                sleep(1);
        }
        return NULL;
}
int main()
{
        pthread_t t1,t2;
            printf("    双11就要到了，两个同学抢ljq直播间的最后10件货\n");
        int i1 = 1,i2 = 2;
        pthread_create(&t1,NULL,func, &i1);
        pthread_create(&t2,NULL,func, &i2);
        pthread_join(t1,NULL);
        pthread_join(t2,NULL);
        printf("    库存还剩下 %d \n", kucun);
        return 0;
}
```

【拓展选做实验内容】

1. 编程实现一百篇英文小说的单词数统计，进而热词统计。
2. 自主设计实现两人互动的贪吃蛇小游戏。
3. 自主设计实现具有实际功能的应用程序，要求多进程、多线程并发实现。

实验四　管道通信和软中断信号通信

【实验目的】

1. 学习无名管道和有名管道通信的原理；
2. 编程实践进程间的管道通信；
3. 学习软中断通信的原理和实现原理；
4. 编程实践进程间的软中断通信。

【实验预备知识】

一、可能用到的函数：

1.pipe() 函数说明

#include<unistd.h>

int pipe(int filedes[2])

pipe() 会建立无名管道。并将文件描述词由 filedes 数组返回。 filedes[0] 为管道的读取端，filedes[1] 为管道的写入端。该函数若执行成功，则返回 0，否则返回-1，错误原因存于 errno 中。

2.read() 函数说明：

#include<unistd.h>

ssize_t read(int fd,void * buf ,size_t count)

read()会把参数 fd 所指的文件传送 count 个字节到 buf 指针所指的内存中。若参数 count 为 0，则 read()不会有作用并返回 0。返回值为实际读取到的字节数，如果返回 0，表示已到达文件尾或是无可读取的数据，此外文件读写位置会随读取到的字节移动。附加说明如果顺利 read()会返回实际读到的字节数，最好能将返回值与参数 count 作比较，若返回的字节数比要求读取的字节数少，则有可能读到了文件尾、从管道(pipe)或终端机读取，或者是 read()被信号中断了读取动作。当有错误发生时则返回-1，错误代码存入 errno 中，而文件读写位置则无法预期。

错误代码：EINTR 此调用被信号所中断；EAGAIN 当使用不可阻断 I/O 时（O_NONBLOCK），若无数据可读取则返回此值；EBADF 参数 fd 非有效的文件描述词，或该文件已关闭。

3.write() 函数说明：

#include<unistd.h>

ssize_t write (int fd,const void * buf,size_t count)

write()会把参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。同时，文件读写位置也会随之移动。返回值如果顺利 write()会返回实际写入的字节数。当有错误发生时则返回-1，错误代码存入 errno 中。

错误代码：EINTR 此调用被信号所中断；EAGAIN 当使用不可阻断 I/O 时（O_NONBLOCK），若无数据可读取则返回此值；EADF 参数 fd 非有效的文件描述词，或该文件已关闭。

4.lockf() 函数说明：

#include<sys/file.h>

int lockf(int fd,int cmd,off_t len)

lockf() 用于对一个已经打开的文件，施加、检测和移除 POSIX 标准的软锁。fd 是目标文件的文件描述符，用户进程必须对 fd 所对应的文件具有 O_WRONLY 或是 O_RDWR 的权限。cmd 是指 lockf()所采取的行动，其合法值定义于 unistd.h 内，其值为以下几种：

F_ULOCK 释放锁定,也可用整数 0 代替；

F_LOCK 将指定的范围上锁,也可用整数 1 代替；

F_TLOCK 先测试再上锁,也可用整数 2 代替；

F_TEST 测试指定的范围是否上锁。

len 是指上锁的解锁的范围，以字节为单位。lockf()函数计算上锁范围是以文件读写指针加上 len，因此 lockf()通常与 lseek()搭配使用。在该实验程序中，使用 lockf(fd[1],1,0) 来实现对管道写入端的加锁，以防止写冲突，使用 lockf(fd[1],0,0) 来实现对管道写入端的解锁。

5.perror() 函数说明

#include<stdio.h>

void perror(const char *s)

perror() 用来将上一个函数发生错误的原因输出的标准错误(stderr)。参数 s 所指的字符串会先打印出来，后面再加上错误的原因。上一函数的错误原因将依照全局变量 errno 的值来决定输出什么字符串。

6.isascii() 函数说明

#include<ctype.h>

int isascii(int c)

此函数检查参数 c 是否为 ASCII 码字符，也就是判断 c 的范围是否在 0 到 127 之间。若参数 c 是 ASCII 码字符，则返回 TRUE，否则返回 NULL(0)。另外，这实际上是在 ctype.h 文件中定义的一个宏，而非真正的函数。

7.isupper() 函数说明

#include<ctype.h>

int isupper(int c)

检查参数 c 是否为大写英文字母。返回值若参数 c 为大写英文字母，则返回 TRUE，否则返回 NULL(0)。附加说明此为宏定义，非真正函数。

8.islower() 函数说明

#include<ctype.h>

int islower(int c)

函数说明检查参数 c 是否为小写英文字母。返回值若参数 c 为小写英文字母，则返回 TRUE，否则返回 NULL(0)。附加说明此为宏定义，非真正函数。

9.toupper() 函数说明

#include<ctype.h>

int toupper(int c)

函数说明若参数 c 为小写字母则将该对映的大写字母返回。返回值返回转换后的大写字母，若不须转换则将参数 c 值返回。

10.tolower()函数说明

#include<stdlib.h>

int tolower(int c)

函数说明若参数 c 为大写字母则将该对应的小写字母返回。返回值返回转换后的小写字母，若不须转换则将参数 c 值返回。

11.signal() 函数说明

#include<signal.h>

void (*signal(int signum,void (*handler)(int)))(int)

signal() 会依照参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达的时候，就会跳转到参数 handler 指定的函数执行。如果参数 handler 不是函数指针，则必须是下列两个常数之一：

SIG_IGN 忽略参数 signum 指定的信号；

SIG_DFL 将参数 signum 指定的信号重设为核心预设的信号处理方式。

在使用 signal()函数设置信号处理方式之后，如果规定的信号到来，则在跳转到自定义的 handler 处理函数执行后，系统会自动将此信号的处理函数换回原来系统预先设定的处理方式。即：signal()的作用是宣告性的，仅仅是修改信号处理表格的某一项，而不是立即执行函数 handler。当程序执行过 signal()以后，表示自此参数 1 的信号（signum）将受到参数 2 的函数（handler）的管制。并非是执行到该行就立即会对信号做什么操作。信号被接受后，进程对该信号的处理是先重设信号的处理方式为默认状态，再执行所指定的信号处理函数。所以，如果每次都要用指定的函数来接受特定信号，就必须在函数里再设定一次接受信号的操作。该函数若执行成功,则返回先前的信号处理函数指针,否则返回-1(SIG_ERR)。

12.kill() 函数说明

#include<sys/types.h>

#include<signal.h>

int kill(pid_t pid,int sig)

kill() 可以用来送参数 sig 所指定的信号给参数 pid 所指定的进程，参 pid 有几种情况：

pid>0　将信号传送给进程识别码为 pid 的进程；

pid=0　将信号传送给和目前进程相同进程组的所有进程；

pid=-1　将信号像广播般传送给系统内所有的进程；

pid<0　将信号传送给进程组识别码为 pid 绝对值的所有进程。

该函数若执行成功。则返回 0，否则返回-1。

13.wait() 函数说明

#include<sys/types.h>

pid_t wait(int *status)

wait() 会暂停目前进程的执行，直到有信号来到或子进程结束。如果在调用 wait()时子进程已经结束，则 wait()会立即返回子进程的结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一块返回。如果不在意结束状态值，则参数 status 也可以设置成 NULL。该函数若执行成功，则返回子进程识别码(pid)，否则返回-1，失败原因存于 errno 中。

二、Linux 的软中断信号

阅读 Linux 系统中的/usr/include/sched.h 和/usr/src/linux-2.4/kernel/fork.c 源代码文件，对进程的创建和管理做了解和分析。

Linux 信号列表：

编号	名称	操作	简单说明
1	SIGHUP	A	当终端机察觉到终止连线操作时便会传送这个信号
2	SIGINT	A	当由键盘要求某个中断的方式时(CTRL+C)则会产生此信号
3	SIGQUIT	A	当由键盘要求停止执行时,如 CTRL+\,则会产生此信号
4	SIGILL	A	进程执行了一个不合法的 CPU 指令
5	SIGTRAP	CG	当子进程因被追踪而暂停时,便会产生此信号给父进程
6	SIGABRT	C	调用 abort() 时会产生此信号
7	SIGBUS	AG	Bus 发生错误时会产生此信号
8	SIGFPE	C	进行数值运算时发生了例外的状况,如除以 0
9	SIGKILL	AEF	终止进程的信号,此信号不能被拦截或忽略
10	SIGUSR1	A	供用户自定的信号
11	SIGSEGV	C	试图存取不被允许的内存地址
12	SIGUSR2	A	供用户自定的信号
13	SIGPIPE	A	错误的管道:欲写入无读取端的管道时,便会产生此信号
14	SIGALRM	A	利用 alarm() 所设置的时间到时就产生此信号
15	SIGTERM	A	终止进程
16	SIGSTKELT	AG	堆栈错误
17	SIGCHLD	B	子进程暂停或结束时便会传送此信号给父进程
17	SIGCLD		和 SIGCHLD 相同
18	SIGCONT		此信号会让暂停的进程继续执行
19	SIGSTOP	DEF	此信号用来让进程暂停执行,此信号不能被拦截或忽略
20	SIGTSTP	D	当由键盘(CTRL+Z)表示暂停时就产生此信号
21	SIGTTIN	D	背景进程欲从终端机读取数据时便产生此信号
22	SIGTTOU	D	背景进程欲写入数据至终端机时便产生此信号
23	SIGURG	BG	socket 的紧急状况发生时便产生此信号
24	SIGXCPU	AG	当进程超过可用的 CPU 时间时便产生此信号
25	SIGXFSZ	AG	当进程的文件大小超过系统限制知便产生此信号
26	SIGVTALRM	AG	利用 settitimer() 所设置的虚拟计时到达
27	SIGPROF	AG	利用 settitimer() 设置的间隔计时器到达
28	SIGWINCH	BG	当视窗大小改变时便产生此信号
29	SIGIO	AG	发生了一个非同步事件
30	SIGPWR	AG	主机电源不稳时则产生此信号
31	SIGUNUSED		未使用

其中操作符号所代表的意义：

A: 默认为终止进程； B: 默认为忽略此信号；

C: 默认为内存倾卸（core dump）；

D: 默认为暂停进程执行； E: 此信号不可拦截；

F:　此信号不可忽略；　　G:　此信号非 POSIX 标准。

【必做实验内容】

1. 进程间的管道通信

进程间可以通过管道相互传送消息，我们可以把管道看作是一块空间，进程可以经由两个不同的文件描述字（文件描述字是 Linux 用来识别文件的，所有已打开的文件，都有系统分配的唯一文件描述字）来分享这块空间。存取这块空间的文件描述字可经下面的 pipe()系统调用取得，由于返回的是两个文件描述字，因此用数组来表示。

程序设计要求一:编写一个关于进程管道通信的简单程序,子进程送一串消息给父进程，父进程收到消息后把它显示出来。

要求:两个子进程分别向管道写一句话：

Child process 1 is sending a message!

Child process 2 is sending a message!

而父进程则从管道中读出来自两个子进程的信息，显示在屏幕上，且父进程要先接收子进程 1 发来的消息，然后再接收子进程 2 发来的消息，参见图 4-1。

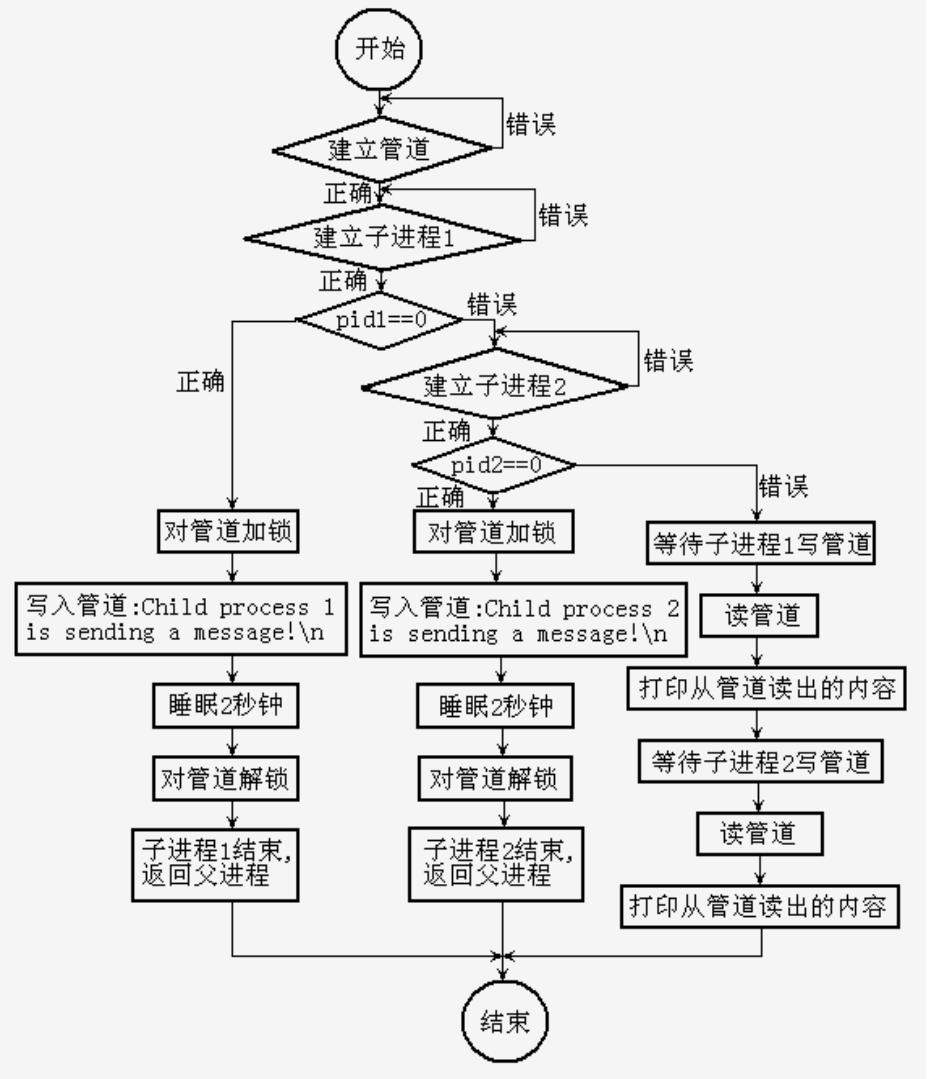


图 4-1　进程间管道通信

程序设计要求二：父进程等待用户从控制台(键盘)输入字符串，通过管道传给子进程；子进程收到后，对字符进行大小写转换后输出到标准输出(显示器)。

2. 进程间的软中断通信

信号是传送给进程的一种事件通知，Linux 系统中所有信号均定义在头文件<signal.h>中。

信号发生时，Linux 内核可以采取下面 3 种动作之一：

① 忽略信号　大部分信号可以被忽略，除 SIGSTOP 信号和 SIGKILL 例外；

② 捕获信号　指定动作；

③ 信号默认动作起作用。

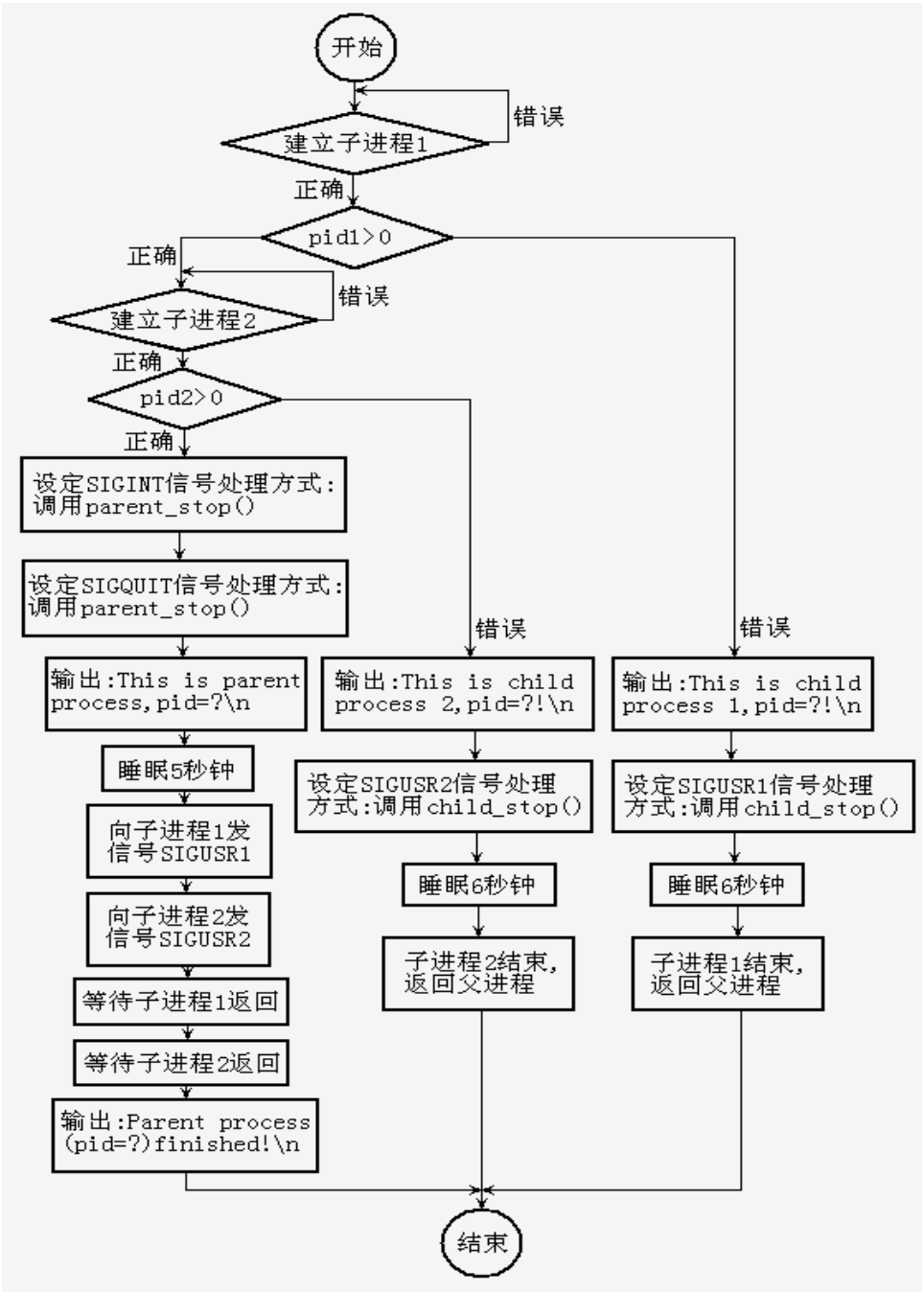


图 4-2 进程间软中断通信

编写程序，使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号(即按 ctrl+c 或是 ctrl+\键)，5 秒钟内若父进程未接收到这两个软中断的某一个，则父进程用系统调用 kill()向两个子进程分别发送软中断信号 SIGUSR1 和 SIGUSR2，子进程获得对应的软中断信号，然后分别输出下列信息后终止：

Child process (pid=?) be killed!

Child process (pid=?) be killed!

父进程调用 wait()函数等待两个子进程终止后，输出以下信息，结束进程执行：

Parent process (pid=?) finished!

进程利用软中断进行通信的流程设计可参见图 4-2。

【拓展选做实验内容】

1.写一个程序，父子进程之间使用有名管道实现通信。

2.将进程通信功能应用到实验三拓展内容中你正在开发的多进程多线程并发程序当中。

【实验四可初步学习和参考的例程】

管道通信例程：

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <ctype.h>

    //父进程从控制台读数据, 并向管道中写数据
void user_handler(int input_pipe[], int output_pipe[])
{
    int c;
    int rc;
    //关闭不必要的文件描述字
    close(input_pipe[1]);
    close(output_pipe[0]);
    while ((c = getchar()) > 0) {   //读字符
            //向管道中写字符
        rc = write(output_pipe[1], &c, 1);
        if (rc == -1) {
            perror("user_handler: write");
            close(input_pipe[0]);
            close(output_pipe[1]);
            exit(1);
        }
        //从管道中读字符
        rc = read(input_pipe[0], &c, 1);
        if (rc <= 0) {
```

```
                perror("user_handler: read");
                close(input_pipe[0]);
                close(output_pipe[1]);
                exit(1);
            }
            //回显字符
            putchar(c);
        }
            //退出时关闭管道
        close(input_pipe[0]);
        close(output_pipe[1]);
        exit(0);
    }

void translator(int input_pipe[], int output_pipe[])
    {
        int c,rc;  char ch;
                //关闭不必要的文件描述字
        close(input_pipe[1]);
        close(output_pipe[0]);
            //从管道中读字符
        while (read(input_pipe[0], &c, 1) > 0) {
            //大小写转换
            ch = (char) c;
            if(isascii(ch) && isupper(ch)) c = tolower(ch);
            else if(isascii(ch) && islower(ch)) c = toupper(ch);
            //向管道中写字符
            rc = write(output_pipe[1], &c, 1);
            if (rc == -1) {
                perror("translator: write");
                close(input_pipe[0]);
                close(output_pipe[1]);
                exit(1);
            }
        }
        //退出时关闭管道
        close(input_pipe[0]);
        close(output_pipe[1]);
        exit(0);
    }
    int main(int argc, char* argv[])
    {
            //两个管道
        int user_to_translator[2];
```

```
    int translator_to_user[2];
    int pid;
    int rc;
        //建立两个管道
    rc = pipe(user_to_translator);
    if (rc == -1) {
        perror("main: pipe user_to_translator");
        exit(1);
    }
    rc = pipe(translator_to_user);
    if (rc == -1) {
        perror("main: pipe translator_to_user");
        exit(1);
    }
        //建立子进程
    pid = fork();
        switch (pid) {
        case -1:    //错误
            perror("main: fork");
            exit(1);
        case 0:     //子进程
            translator(user_to_translator,
                translator_to_user);
        default:
            user_handler(translator_to_user,
                user_to_translator);
    }
    return 0;
}
```

2、进程间的软中断信号通信

```
#include<signal.h>
#include<unistd.h>
#include<sys/types.h>
void parent_stop();
void child_stop();
main(){
    int pid1,pid2;
    while((pid1=fork())==-1);
    if(pid1>0){
        while((pid2=fork())==-1);
        if(pid2>0){
            printf(“This is parent process ,pid=%d\n”,getpid());
```

```
            signal(SIGINT,parent_stop);
            //按”ctrl+c”退出
            signal(SIGQUIT,parent_stop);
            //按”ctrl+\”退出
            sleep(5);
            kill(pid1,SIGUSR1);
            kill(pid2,SIGUSR2);
            wait(0);
            wait(0);
            printf(“Parent process (pid=%d) finished!\n”,getpid());
            exit(0);
        }else{
            printf(“This is child process 2 pid=%d\n”,getpid());
            signal(SIGUSR2,child_stop);
            sleep(6);
        }
    }else{
        printf(“This is child process 1 pid=%d\n”,getpid());
        signal(SIGUSR1,child_stop);
        sleep(6);
    }
}
void parent_stop(){
    printf(“Parent process will be killed.The children will be killed
first!\n”);
    exit(0);
}
void child_stop(){
    printf(“Child process (pid=%d) be killed!\n”,getpid());
    exit(0);
}
```

实验五　共享内存通信和消息队列通信

【实验目的】

1. 熟悉进程通信机制(IPC)：消息队列，共享内存，明确进程通信的原理。
2. 观察进程通信的过程，认识进程通信机制的优点。
4. 编程模拟进程间共享内存通信的实现方式。
5. 编程模拟进程间通过消息队列通信的实现方式。

【实验预备知识】

一、本试验可能用到以下 8 个函数：

(1) shmget()

该函数定义在 sys/ipc.h 文件和 sys/shm.h 文件中，函数原形为 int shmget(key_t key,int size,int shmflg) 。

shmget()用来取得参数 key 所关联的共享内存识别代号。如果参数 key 为 IPC_PRIVATE 则会建立新的共享内存，其大小由参数 size 决定。Shmflg 参数在实验程序中，其值可以为 0 。另外，key 和 shmflg 参数的具体使用，可以使用 man 命令查询。该函数若成功，则返回共享内存识别代号，否则返回-1，错误原因存于 errno 中。

(2) shmat()

该函数定义在 sys/types.h 文件和 sys/shm.h 文件中，函数原形为 void* shmat(int shmid,const void* shmaddr,int shmflg)。

shmat()函数用来将参数 shmid 所指的共享内存和目前进程连接(attach)。参数 shmid 为欲连接的共享内存识别代码，而参数 shmaddr 有下列几种情况：

1) shmaddr 为 0，核心自动选择一个地址；

2) shmaddr 不为 0，参数 shmflg 也没有指定 SHM_RND 旗标，则参数 shmaddr 为连接地址；

3) shmaddr 不为 0，但参数 shmflg 设置了 SHM_RND 旗标，则参数 shmaddr 会自动调整为 SHMLAB 的整数倍。

参数 shmflg 还可以有 SHM_RDONLY 旗标，代表此连接只是用来读取该共享内存。另外，各参数的具体使用，可以使用 man 命令查询。该函数若成功，则返回共享内存识别代号，否则返回-1，错误原因存于 errno 中：

附加说明：在经过 fork()后，子进程将继承已连接的共享内存地址；在经过 exec()后，已连接的共享内存地址将会自动脱离（detach）；在结束进程后，已连接的共享内存地址将会自动脱离（detach）。

(3) shmdt()

该函数定义在 sys/types.h 文件和 sys/shm.h 文件中，函数原形为 int shmdt(const void* shmaddr) 。

shndt()用来将先前用 shmat()连接(attach)好的共享内存脱离(detach)目前的进程。参数 shmaddr 为先前 shmat()返回的共享内存地址。该函数若成功，则返回 0，否则返回 -1，错误原因存于 errno 中：

(4) shmctl()

该函数定义在 sys/ipc.h 文件和 sys/shm.h 文件中，函数原形为 int shmctl(int shmid,int cmd,struct shmid_ds *buf) 。

shmctl()提供了几种方式来控制共享内存的操作。参数 shmid 为欲处理的共享内存识别代码，参数 cmd 为欲控制的操作,有以下几种数值：

1) IPC_STAT　　把共享内存的 shmid_ds 数据结构复制到引数 buf；

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

3) IPC_RMID　　删除共享内存和其数据结构；

4) SHM_LOCK　　不让此共享内存置换到 swap；

5) SHM_UNLOCK　允许此共享内存置换到 swap；(SHM_LOCK 和 SHM_UNLOCK 为 Linux 特有，且只有超级用(root)户允许使用) 。

shmid_ds 的结构定义如下：

```
struct shmid_ds
{
struct ipc_perm shm_perm;          //所使用的 ipc_perm 结构
int shm_segsz;                     //共享内存的大小(bytes)
time_t shm_atime;                  //最后一次 attach 此共享内存的时间
time_t shm_dtime;                  //最后一次 detach 此共享内存的时间
time_t shm_ctime;                  //最后一次更动此共享内存结构的时间
unsigned short shm_cpid;           //建立此共享内存的进程识别码
    unsigned short shm_lpid;       //最后一个操作此共享内存的进程识别码
    short shm_nattch;
    unsigned short shm_npages;
    unsigned long *shm_pages;
    struct shm_desc *attaches;
}
```

shm_perm 所使用的 ipc_perm 结构定义如下：

```
struct ipc_perm
{
    key_t key;                    //此信息的 IPC key
    unsigned short int uid;       //此信息队列所属的用户识别码
    unsigned short int gid;       //此信息队列所属的组识别码
    unsigned short int cuid;      //建立信息队列的用户识别码
    unsigned short int cgid;      //建立信息队列的组识别码
    unsigned short int mode;      //此信息队列的读写权限
    unsigned short int seq;       //序号
}
```

(5) msgget()

该函数定义在 sys/types.h 文件 sys/ipc.h 文件 sys/msg.h 文件中，函数原形为 int msgget(key_t key,int msgflg) 。

Msgget()用来取得参数key所关联的信息队列识别代号。如果参数key为IPC_PRIVATE则会建立新的信息队列，如果 key 不为 IPC_PRIVATE，也不是已建立的 IPC key，则系统会视参数 msgflg 是否有 IPC_CREAT 位(msgflg&IPC_CREAT 为真)来决定建立 IPC key 为key的信息队列。如果参数 msgflg 包含了 IPC_CREAT 和 IPC_EXCL 位，而无法依参数 key 来建立信息队列，则表示信息队列已存在。此外，参数 msgflg 也用来决定信息队列的存取权限。

该函数若调用成功则返回信息队列识别代号，否则返回-1,错误原因存于 errno 中：

EACCESS　参数 key 所指的信息队列存在，但无存取权限；
EEXIST　欲建立 key 所指新的信息队列，但该队列已经存在；
EIDRM　参数 key 所指的信息队已经删除；
ENOENT　参数 key 所指的信息队列不存在，而参数 msgflg 也未设 IPC_CREAT 位；
ENOMEM　核心内存不足；
ENOSPC　超过可建立信息队列的最大数目。

(6) msgsnd()

该函数定义在 sys/types.h 文件 sys/ipc.h 文件 sys/msg.h 文件中，函数原形为 ing msgsnd(int msqid,struct msgbuf *msgp,int msgsz,int msgflg) 。

msgsnd()用来将参数 msgp 指定的信息送至参数 msqid 的信息队列内，参数 msgp 为 msgbuf 结构，起定义如下：

```
struct msgbuf
{
    long mtype;          //信息的种类,必须大于 0
    char mtext[1];       //信息数据
}
```

在以上结构体中，mtext 是消息正文，该域可以由程序员重新定义为任意数据结构，但消息及消息队列的长度是有限的，这些值的设置在系统配置时可以改变，缺省值定义在 /include/lixun/msg.h 中：

```
#define MSGMAX 4056     //一条消息的最大字节数
#define MSGMNI 128      //消息队列个数的最大值
#define MSGMNB 16384    //一个消息队列的最大字节数
```

参数 msgsz 为信息数据的长度，即 mtext 参数的长度，参数 msgflg 可以设成 IPC_NOWAIT，意为如果队列已满或是有其他情况无法马上送入信息，则立即返回 EAGAIN。

该函数若调用成功则返回 0，否则返回-1，错误代码存于 errno 中：

EAGAIN　参数 msgflg 设 IPC_NOWART，对立已满；
EACCESS　无权限写入该信息队列；
EFAULT　参数 msgp 指向无效的内存地址；
EIDRM　参数 msqid 所指的信息队列已删除；
EINTR　对立已满而处于等待情况下被信号中断；
EINVAL　无效的参数 msqid,msgsz 或参数 mtype 小于 0。

(7) msgrcv()

该函数定义在 sys/types.h 文件 sys/ipc.h 文件 sys/msg.h 文件中，函数原形为 int msgrcv(int msqid,struct msgbuf *msgp,int msgsz,long msgtyp,int msgflg)。

msgrcv()函数用来从参数 msqid 指定的信息读取信息出来，然后存于参数 msgp 所指定的结构内，参数 msgsz 为信息数据的长度，即 mtext 参数的长度，参数 msgtyp 是用来指定所要读取的信息种类：msgtyp=0 返回队列内第一项信息.msgtyp>0 返回队列内第一项 msgtyp 与 mtype 相同的信息。msgtyp<0 返回队列内第一项 mtype 小于或等于 msgtyp 绝对值的信息。参数 msgflg 可以设成 IPC_NOWAIT，指定若队列内没有信息可读，则不用等待，立即返回 ENOMSG.如果 msgflg 设成 MSG_NOERROR,则信息大小超过参数 msgsz 时会被截断。

该函数调用成功则返回世界读取到的信息数据长度,否则返回-1,错误原因存于 errno 中：

E2BIG　　信息数据长度大于参数 msgsz，却没设置 MSG_NOERROR；
EACCESS　无权限读取该信息队列；
EFAULT　　参数 msgp 指向无效的内存地址；
EIDRM　　参数 msqid 所指的信息队列已经删除；
EINTR　　等待读取队列内的信息时被信号中断；
ENOMSG　　参数 msgflg 设成 IPC_NOWAIT，而队列内没有信息可读。

(8) msgctl()

该函数定义在 sys/types.h 文件 sys/ipc.h 文件 sys/msg.h 文件中，函数原形为 int msgctl(int msqid,int cmd,struct msqid_ds *buf)。

msgctl()提供了几种方式来控制消息队列的运作。参数 msgid 为欲处理的消息队列的识别代码，参数 cmd 为欲控制的操作，有下列几种数值：

IPC_STAT　把消息队列的 msqid_ds 结构数据复制到参数 buf 中；
IPC_SET　将参数 buf 所指的 msqid_ds 结构中的 msg_perm.uid、msg_perm.gid、msg_perm.mode 和 msg_qbytes 参数复制到消息队列的 msqid_ds 结构内;
IPC_RMID　删除消息队列及其数据结构。

其中，msqid_ds 结构体定义如下：

```
struct maqid_ds
{
    struct ipc_perm msg_perm;
    struct msg *mag_first;          //指向第一个存于队列的信息
    struct msg *msg_last;           //指向最后一个存于队列的信息
    time_t msg_stime;               //最后一次用 msgsnd()送入信息的时间
    time_t msg_rtime;               //最后一次用 msgrcv()读取信息的时间
    time_t msg_ctime                //最后一次更动此消息队列结构的时间
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    unsigned short int msg_cbytes; //目前消息队列中存放的字符数
    unsigned short int msg_qnum;   //消息队列中的信息个数
    unsigned short int msg_qbytes; //消息队列所能存放的最大字符数
    ipc_pid_t msg_lspid;        //最后一个用 msgsnd()送入信息的进程识别码
    ipc_pid_t msg_lrpid;        //最后一个用 msgrcv()读取信息的进程识别码
}
```

其中，msg_perm 所使用到的结构体 ipc_perm 的定义见本节函数 shmctl()的说明。

该函数若调用成功则返回 0，否则返回-1，错误原因存于 errno 中：

EACCESS　参数 cmd 为 IPC_STAT,却无权限读取该消息队列；

EFAULT　参数 buf 指向的内存地址无效；

EIDRM　参数 key 所指的消息队列已经删除；

EINVAL　无效的参数 cmd 或 msqid；

EPERM　参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行。

二、System V 进程通信机制(IPC)

作为一个成熟的操作系统，Linux 支持典型的三种进程间通信机制:共享内存(shared memory)，消息(message)，信号量(semaphores) 。

这三种通信方法虽然操作在不同的数据结构上，但在概念和用法上与共享内存很相似。它们每一种通信，都要先使用一个调用来创建用于通信的资源;同样，它们也要使用调用来进行控制操作和存取访问。

	共享内存	信号量	消息
创建	shmget()	semget()	msgget()
控制	shmctl()	semctl()	msgctl()
访问	shmat()	semop()	msgsnd()
	shmdt()		msgrcv()

其中，共享内存可以使得两个或多个进程都能使用的地址空间中使用同一块内存.这就使得如果一个进程已经把信息写如到这一内存块中，马上就可以供其它共享这一块内存的进程使用.这样就能够克服诸如使用管道(PIPE)时候的几个问题:传递数据的大小限制(匿名管道大小为 4K，而共享内存大小可自定) ，传递数据的反复复制(可跟管道实验中的程序做比较) ，环境的切换(例如:试图读空管道的进程会被阻塞) 。

而消息队列则是一个由消息缓冲区构成的链表，它允许一个或多个进程写信息，将消息缓冲区挂在一个队列上;一个或多个进程读去消息，从队列上摘取消息缓冲区。消息队列作为 System V 的 IPC 通信机制的一种，它的模型与共享内存有不少相似，通过一个消息队列标识符来唯一标识和进程访问权限检查。

【必做实验内容】

1. 进程共享内存通信

编写一个程序 sharedmem.c，在其中建立一个子进程，让父子进程通过共享内存的方法实现通信。其中，父进程创建一个共享内存段，然后由子进程将该共享内存附加到自己的地址空间中，并写入该共享内存下列信息：Hi,this is child process sending message!。在等待子进程对共享内存的操作完成后，父进程将该共享内存附加到自己的地址空间中，并读出该共享内存中的信息，与该共享内存段的基本信息(大小，建立该共享内存的进程 ID，最后操作该共享内存段的进程 ID)一并显示出来，参见图 4-1。

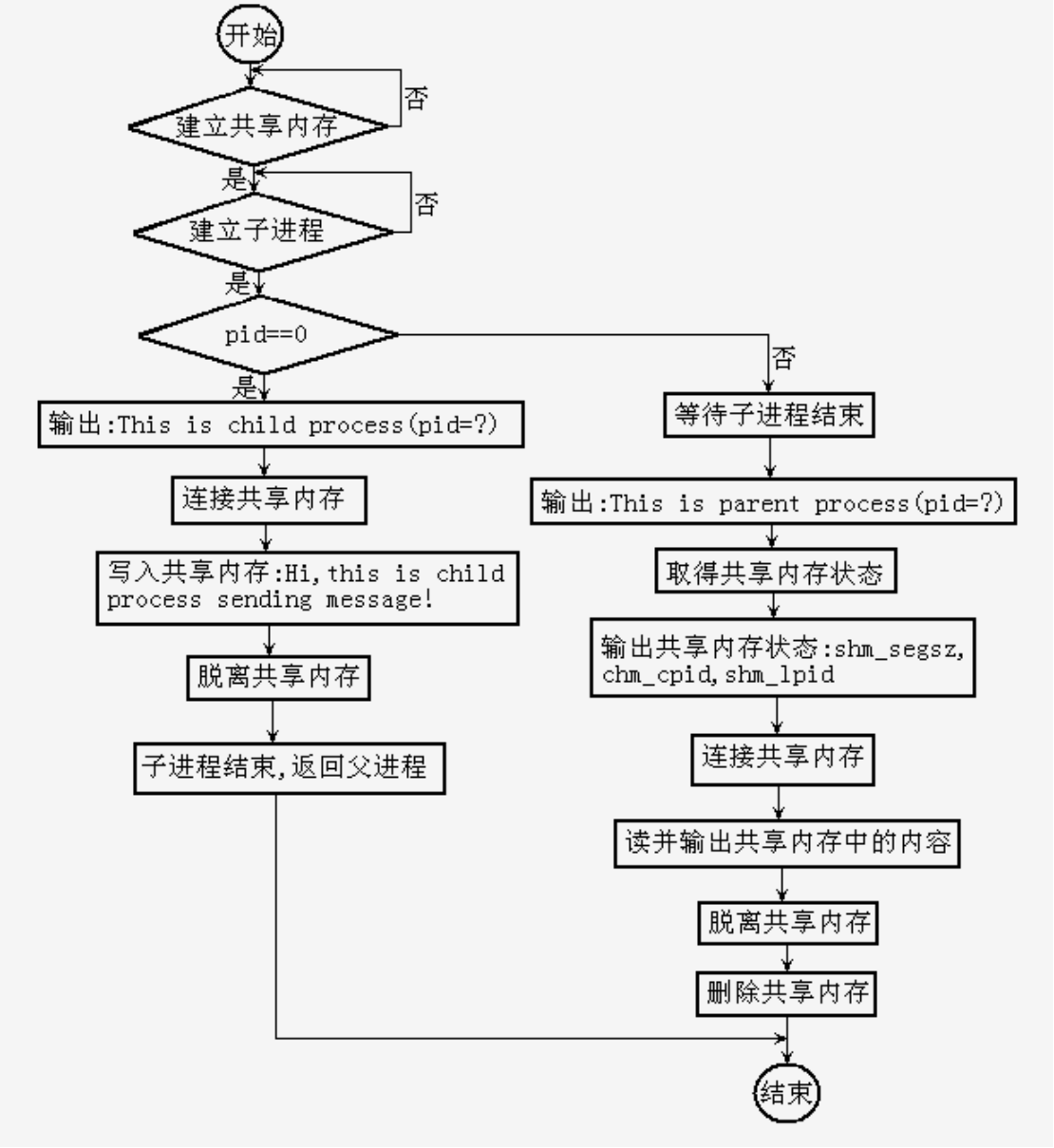


图 4-1　进程间共享内存通信

2. 进程消息队列通信

（1）编写一个程序 message.c，在其中由父进程建议一个信息队列，然后由新建立的子进程在表明身份后，想消息队列中写如下列信息：Hi,this is child process sending message!。父进程在等待子进程对消息队列写操作完成后，从中读取信息，并显示出来,。在删除消息队列后结束程序，参见图 4-2。

（2）编写程序 msg2.c 和 msg3.c。在 msg2.c 中建立一个消息队列，表明身份后，向消息队列中写下 2 条信息：“Hi,message1 is sending!”和“Hi,message2 is sending!”。在 msg3.c 中，从消息队列中读取信息 2，并显示出来。

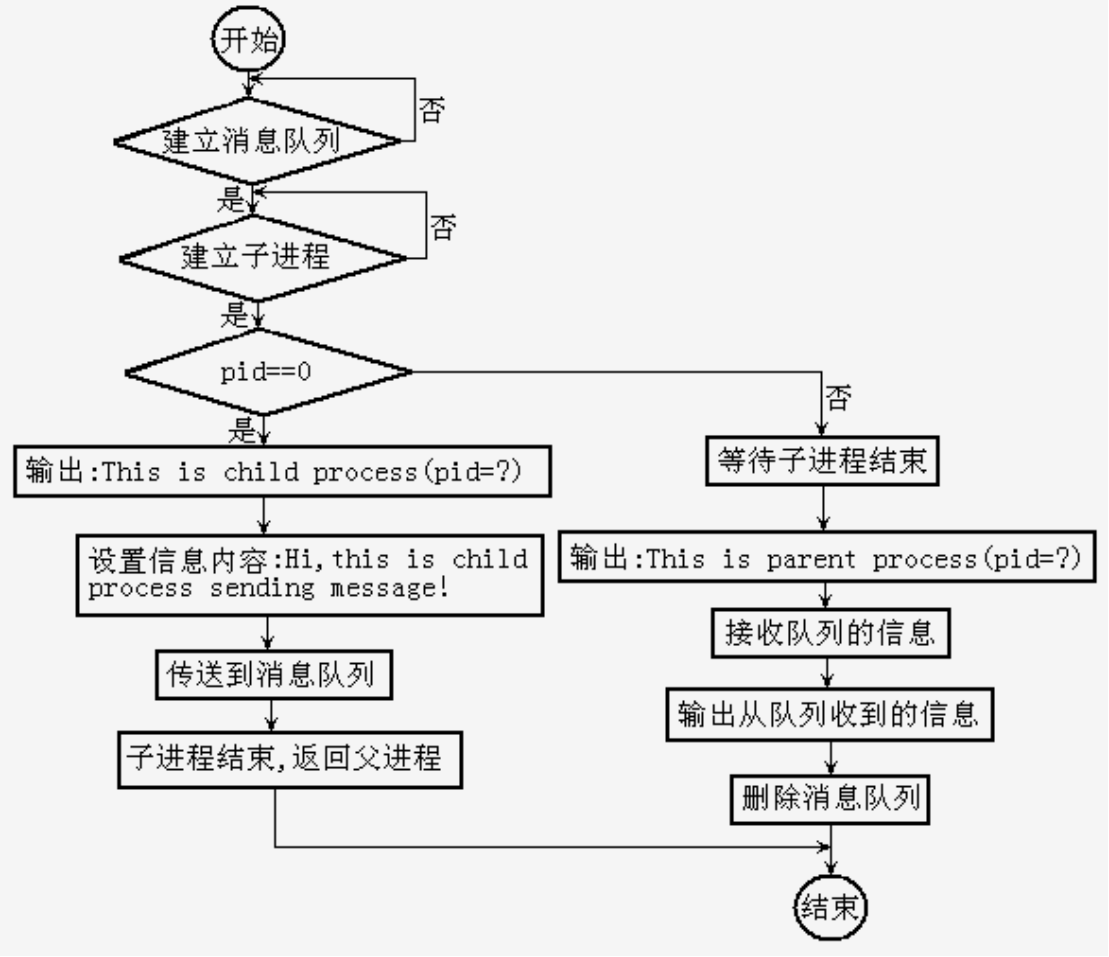


图 4-2　进程间消息队列通信

【拓展选做实验内容】

可将进程通信功能应用到实验三拓展内容 中你正在开发的多进程多线程并发程序中。

【实验五可初步学习和参考的例程】

1. 进程共享内存通信

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

#define SIZE 2048 //所要建立的共享内存段的大小
main()
{
    int shm_id;
```

```
    char *shm_addr;
    struct shmid_ds buf;
    int pid;
    while((shm_id=shmget(IPC_PRIVATE,SIZE,0))==-1);
    while((pid=fork())==-1);
    if(pid==0){
        printf(“This is child process (pid=%d)\n”,getpid());
        shm_addr=(char*)shmat(shm_id,NULL,0);
        strcpy(shm_addr,”Hi,this is child process sending message!\n”);
        shmdt(shm_addr);
    }else{
        wait(0);
        sleep(1);
        printf(“This is parent process (pid=%d)!\n”,getpid());
        shmctl(shm_id,IPC_STAT,&buf);
        printf(“The shm_segsz=%d bytes\n”,buf.shm_segsz);
        printf(“The shm_cpid=%d\n”,buf.shm_cpid);
        printf(“The shm_lpid=%d\n”,buf.shm_lpid);
        shm_addr=(char*)shmat(shm_id,NULL,0);
        printf(“The shared_memory contain:%s\n”,shm_addr);
        shmdt(shm_addr);
        shmctl(shm_id,IPC_RMID,NULL);
    }
}
```

2. 进程消息队列通信

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 1234
#define TEXT_SIZE 50

struct msgbuffer
{
    long mtype;
    char mtext[TEXT_SIZE];
}msgp;

main()
{
    int msgqid;
    int pid;
    msgqid=msgget(KEY,IPC_CREAT|0600);
```

```
    while((pid=fork())==-1);
    if(pid==0){
        printf(“This is child process,pid=%d\n”,getpid());
        msgp.mtype=1;
        ctrcpy(msgp.mtext,” Hi,this is child process sending message!”);
        msgsnd(msqid,&msgp,TEXT_SIZE,0);
        exit(0);
    }else{
        wait(0);
        printf(“This is parent process,pid=%d”,getpid());
        sleep(1);
        msgrcv(msqid,&msgp,TEXT_SIZE,1,0);
        printf(“parent process recrive mtext:%s”,msgp.mtext);
        msgctl(msqid,IPC_RMID,NULL);
    }
}
```