一、Linux 简介

Linux 是 UNIX 的一个变种，它的许多思想都来自 UNIX。它是一个多用户、多任务的分时操作系统，这意味着可以有很多个用户同时使用一台计算机来运行各自不同的应用程序。下面我们对其作一简要介绍。

1、 登录和退出

在 Linux 系统中，为了标识系统中的用户，系统管理员给每个用户建立了一个帐户（account），它包括用户名、个人目录等。

登录就是注册到自己的帐户。具体讲就是在 login 提示符下，输入你的用户名，若你的帐户还有口令的话，还需要在 passwd 提示符下输入你的口令，这样你就登录到系统中来了。如果你登录成功，会出现系统提示符，通常是一个目录后跟一“#”（当然也可以是别的什么）。现在你可以输入 Linux 命令了。若登录失败，则会出现 login incorrect 提示。

为了安全起见，登录后也许你想改变或设置口令，此时可在系统提示符下，输入 passwd 命令，随后输入旧口令（若没有，回车即可），再输入新口令，并再次输入新口令确认。现在，你有了自己的新口令了。

如果你想退出系统，千万不能直接关机，作为一个多用户、多任务的操作系统，Linux 不允许这样作，因为直接关机后，你的帐户还在活跃着，并未真正从系统中退出。正确的做法是：输入 logout 或 exit 命令，再关机。

2 、文件和目录

文件是用文件名来管理信息的。文件可以是程序、论文、信件

等等。

与文件密切相关的概念是目录，目录是许多文件的集合，它可以看作包含许多文件的文件夹。目录是树形结构的，也就是说，目录可以包含其它的目录。

每个文件都有一个路径名，它是由文件的目录名加上文件名构成的，用来定位文件，其中的文件名和目录名，以及目录名之间用／隔开。

Linux 中，文件是分层的，其文件层次结构呈树状，它从根目录（以“／”表示）开始。

通常，每个用户都有一个个人目录，这个目录用来存放属于这个用户的个人文件。个人目录放在／home 下，由目录所有者的名字来命名。如用户 stu1 的个人目录是/home/stu1。

当前工作目录就是你当前定位的目录，当你注册时，你的当前工作目录就是你的个人目录。任何时刻，你所输入的命令都是工作在你的当前目录下。

3、与目录和文件有关的常用命令

1）、cd 命令

使用语法：cd <directory>。

功能：改变当前目录。

<directory>为想转换的目录名，父目录为“..”，目录本身为“.”。该命令不带<directory>参数时，表示返回到个人目录。

2）、ls 命令

使用语法：ls <directory>。

功能：查看目录内容，显示指定目录下的文件和字目录。

如果没有<directory>参数，则显示当前目录下的文件。如果想获得进一步的信息，譬如想知道显示出来的是文件还是目录，是什

么样的文件等，可带参数-F

ls –F

若显示出来的文件名后面有“/”，表示它是子目录；有“*”，表示该文件是一个可执行文件；什么都没有，那么它既不是目录，又不是可执行文件。

ls 命令还有许多参数，在使用时，都是从“-”开始，如 ls –l –F；也可以几个参数共享一个“-”，如 ls-lF 等价于前。其它 Linux 命令也大抵如此。

3）、mkdir 命令

使用语法：mkdir <directory>。

功能：在当前目录下创建一个子目录<directory>。

4）、rmdir 命令

使用语法：rmdir <directory>。

功能：删除子目录<directory>。

被删除的子目录必须是空的，且不能是当前工作目录。

5）、cp 命令

使用语法：cp 源文件 目的文件或目的目录

功能：复制文件。将一个文件复制成另一个文件，或者复制数个文件，将其存入指定目录下。

6）、mv 命令

使用语法：mv 源文件 目的文件或目的目录。

功能：搬移文件。将一个文件移为另一个文件（源文件就不存在了），或者将数个文件移到另一个目录下(源文件也不复存在了)。

7）、rm 命令

使用语法：mv　文件名。

功能：删除文件。

8）、more　命令

使用语法：more　文件名。

功能：查看文件内容。

查看时，一次显示一屏，按空格显示下一页，b 显示上一页，q 退出。

9）、cat　命令

使用语法：cat　文件名

或者　cat　文件 1　文件 2 …>文件名

功能：一次显示整个文件

或者　将几个文件连接成一个文件。

4、文件权限

在 Linux 中，为防止用户的私人文件被其它用户所侵犯，实行文件授权机制。即允许文件和目录由部分用户所拥有，而拒绝其它用户访问。

文件的权限主要分成三种：读、写、执行。在命令中，分别用 r、w、x 来表示。这些权限给三种类型的用户：文件所有者、文件所有者的同组用户（通常用户都至少属于一个组，给予这个组的权限，该组用户都享有）、其它用户（既不是文件所有者，也不和文件所有者同组）。

读权限允许一个用户读文件的内容，如果是一个目录的话，允许用 ls 命令列出目录的内容；写权限允许用户写或修改文件的内容，对目录来说，写权限允许用户创建新文件或删除该目录下的文件；执行权限允许用户运行这个文件，对目录来说，执行权限允许

用户用 cd 命令进入这个目录。

现在来看一个例子。Ls 命令带-l 参数可以显示文件的详细信息，其中就包括文件权限，假设当前目录下有一个文件 stuff，现在执行如下命令：

ls -l

执行结果显示如下：

-rw-r--r-- 1 larry users 505 Mar 13 19:09 stuff

显示内容共分 7 栏，意义如下：

第 1 栏 –rw-r--r--

其中第 1 列表示文件类型：

“-”表示是一个一般的文件

“b” 表示是一个块设备文件

“c”表示是一个字符设备文件

“d”表示是一个目录

“p”表示是一个命名管道

后面的 rw-r--r--分别表示文件所有者、文件所有者的同组用户、其它用户的权限。

第 2 栏表示文件的连接数，此处为 1。（关于连接，后面会提到）

第 3 栏表示文件所有者名字，此处为 larry。

第 4 栏表示文件所有者所属的组，此处为 users。

第 5 栏表示文件长度，此处为 505。

第 6 栏表示最近被更改的时间，此处为 Mar 13 19:09。

第 7 栏表示文件名称，此处为 stuff。

文件的权限也不是一成不变的，可以用 chmod 命令来重新设置，但只有文件所有者或授权的用户才有权改变。该命令使用语法如下：

chmod {a,u,g,o} {+-} {r,w,x} 文件名

其中 a,u,g,o 分别代表所有用户（all）、所有者用户(user)、同组

用户(group)、其它用户(other)。此处若缺省则默认为 a。

+ - 表示是增加权限还是减少权限。

r,w,x 表示增加或减少的权限。

如 chmod ug +rw stuff

表示给予文件所有者及其同组用户对文件 stuff 的读写权限。

最后要指出，文件权限不仅取决于文件本身权限，同时也依赖与文件所在目录。即使文件的权限设置成 rwxrwxrwx，如果你没有访问这个文件所在目录的权限，仍无法访问它。因此，如果你想限制对你文件的访问，只需把你的个人目录的权限设置成-rwx------即可。同样地，如果你要访问一个文件，除了必须拥有这个文件的读（或执行或写)权限，同时还必须拥有这个文件路径名中所有目录的执行权限（允许用 cd 命令进入）。

5、文件链

Linux 系统中，文件是由其 i 结点来标识的，它是这个文件读一无二的系统标识符。目录中每个文件名都与一个特定的 i 结点连接起来。用 ls -i 命令可以显示文件的结点号。

如 ls -i foo

结果为 22191 foo 表示文件 foo 的结点号为 22191。

我们可以通过 ln 命令给文件创建多个链，亦即让多个文件名连接到同一个 i 结点。

如 ln foo foo1

给文件 foo 建立一个连接 foo1，用 ls –i foo foo1 命令会看到它们具有相同的结点号。

6、复位向及管道命令

如果你不想把命令的结果显示出来，Linux 中，Shell 允许你用

“>” 把标准输出复位向到一个文件（Shell 就是一个读取并执行来自用户的命令的程序，可看作是 Linux 的外壳）。我们看下面的命令：

ls >filelist

该命令把 ls 的执行结果存放到文件 filelist，而不是显示在屏幕上。

复位向是破坏性的,它会重写所给文件的内容。如果想保留原来文件的内容，只是把输出附加在所给文件的后面，可用“>>”复位向。如 ls >>filelist 。

管道是把前一个命令的输出作为下一个命令的输入。管道符号为“|”。如命令 ls|sort，执行的结果是把 ls 的输出作为排序命令 sort 的输入。

7、作业控制

对 Shell 来说，作业就是一个正在运行的进程。作业既可以在前台也可以在后台运行，任一时刻，前台只能有一个作业。作业还可以挂起或中断（ctrl-c）。

通常在 Shell 提示符下输入的命令是前台执行的，即从键盘接受输入，输出送到屏幕，作业完成前，不显示系统提示符，不让你输入新的命令。在执行一个大程序时，你就得等很长时间才能做别的事情。如果把它放到后台，你就可以在等待的过程中，干点别的什么了。方法是在命令的尾部附加一个“&”字符。

在后台运行的程序是看不见的，那么怎样才能知道当前都有哪些进程呢？ps 命令可以列出当前附属于所用终端的所有进程。ps 命令执行结果的第一列是进程的 ID 号（进程标识符），最后一列是命令的名字。jobs 命令也可以列出正在运行的所有进程，只不过显示结果的第一列是作业号，而不是进程号。如果杀死一个进程，用 kill 命令。下面是这三个命令的使用语法：

ps

jobs

kill %作业号 或 kill ID 号

前台和后台的作业是可以切换的。fg 命令把后台的作业切换到前台，bg 命令把前台的作业切换到后台，若命令已在前台执行，可先挂起（ctrl-z），再切换。使用语法如下：

fg %作业号

bg %作业号

8、用户管理

Linux 中，每个使用系统的人都应有自己的独立帐户，帐户是标识系统中用户的唯一方法。系统保存着有关每个用户的一系列信息，描述如下：

用户名(username)：标识系统中每个用户的唯一标识符，可以是字母、数字、下划线、句点，一般不超过 8 个字符。

用户 ID(UID)：系统给每个用户的唯一数字，系统就是通过它来保存用户信息的，而不是通过用户名。

组 ID(GID)：用户所属组的唯一标识数字，每个用户被管理员定义属于一个或多个组。

口令(passwd)：用户自己设置的口令。

全名(full name)：用户的完整名字。

个人目录(home directory)：用户在注册时最初所处的目录。

注册 Shell(login shell)：用户在注册时启动的 Shell。

可以往系统中增加或删除用户，可以用如下命令：

adduser 或 useradd 增加用户

userdel 或 deluser 删除用户

9、Vi 编辑器

文本编辑器是编辑文本文件（如程序等）的程序，下面就简要

介绍一下 Linux 中普遍存在的编辑器—Vi。

使用语法：Vi　<被编辑的文件名>

Vi 有三种模式：命令模式、插入模式、最后行模式。

命令模式：

刚启动 Vi 后，就处于该模式。在此模式下，允许用 Vi 的子命令来编辑文件或转移到其它模式。如：

x 命令：删除光标上面的字符。

方向键：移动光标。

i 命令：进入插入模式，可在当前光标处插入字符。

a 命令：进入插入模式，可在当前光标后插入字符。

R 命令：从当前光标处开始替换文本。

r 命令：替换当前光标处字符。

~命令：对当前光标处字母进行大小写转换。

dd 命令：删除光标所在行。

dw 命令：删除光标所在处字。

o 命令：在当前行下面插入一行。

h 命令：光标左移。

l 命令：光标右移。

k 命令：光标上移。

j 命令：光标下移。

： 命令：进入最后一行模式。

插入模式：

此模式下允许输入文本，用回车键换行，Esc 进入命令模式。

最后一行模式：

w 命令：将文件存盘，但不退出。

wq 命令：将文件存盘，并退出。

q! 命令：不保存文件并退出。

r 命令：将另一个文件内容插入当前光标处。

10、C 语言程序的编译与执行

源程序在 Vi 编辑器中编好后，需要经过编译变成可执行文件，才能运行。Linux 提供了 cc 编译程序，用法如下：

cc 源程序

生成的可执行文件为 a.out。

若源程序有错误,编译器会在 shell 中指出,可修改后重新编译.

11 、在当前目录下执行文件:

./a.out

实验一 gcc 和 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。

【实验预备内容】

(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

Unix 上使用的 C 语言编译器 cc，在 Linux 上的派生就是 gcc。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);
}
```

【实验指导】

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);
}
```

实验二　进程管理

【实验目的】

(1) 对理论课中学习的进程，程序等的概念做进一步地理解，明确进程和程序的区别；

(2) 加深理解进程并发执行的概念，认识多进程并发执行的实质；

(3) 观察进程争夺用资源的现象，分析其过程和原因，学习解决进程互斥的方法；

(4) 了解 Linux 系统中多进程之间通过管道通信的基本原理和应用方法；

(5) 对典型的多用户、多任务的、优先级轮转调度系统 Linux 有一定的了解。

【实验预备内容】

一、可能用到的函数：

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.pipe() 函数说明

#include<unistd.h>

int pipe(int filedes[2])

pipe() 会建立管道。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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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)。

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。

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。

(2) 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。

(3) 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;

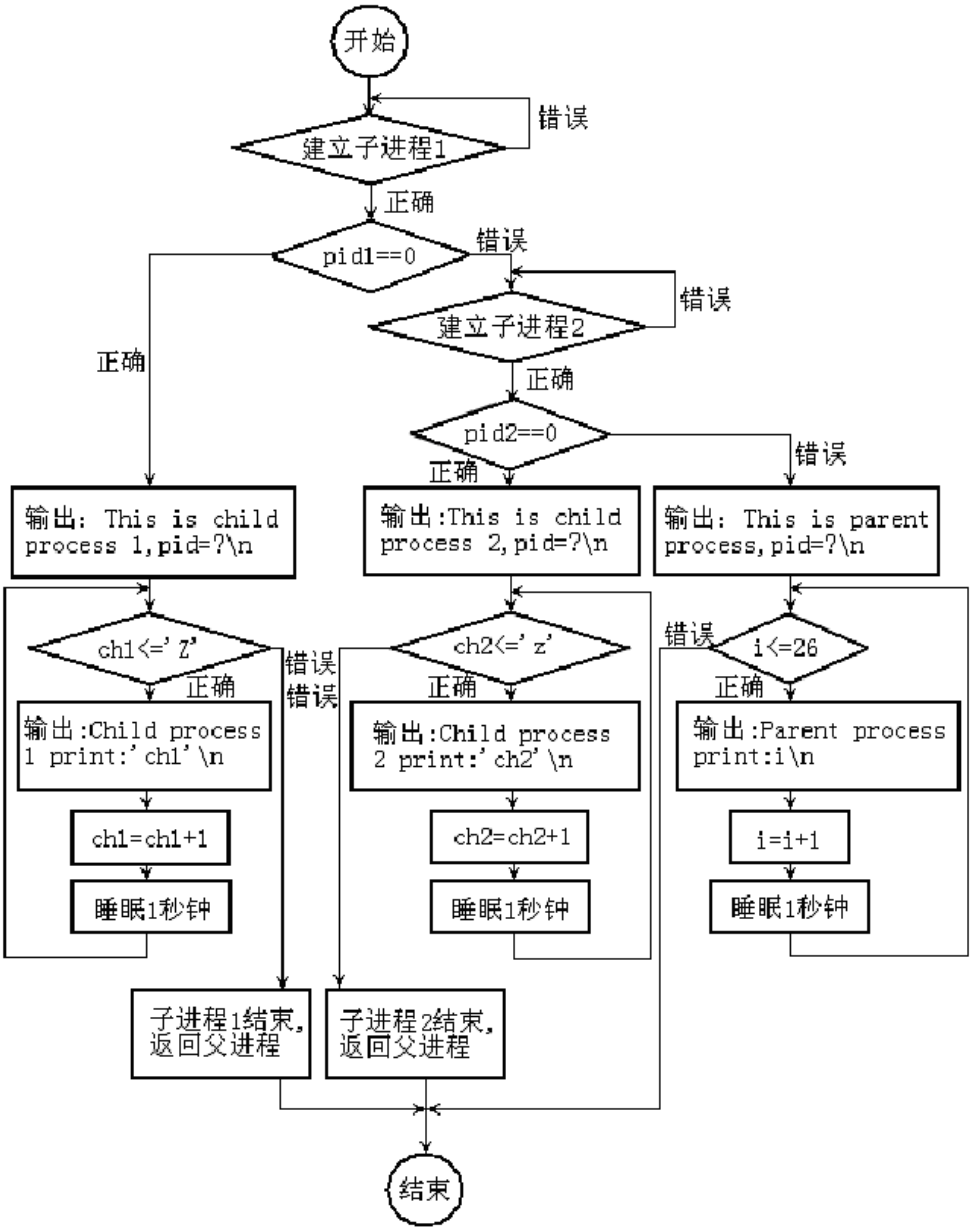
G: 此信号非 POSIX 标准。

【实验内容】

一 多进程并发执行

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。



二 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。

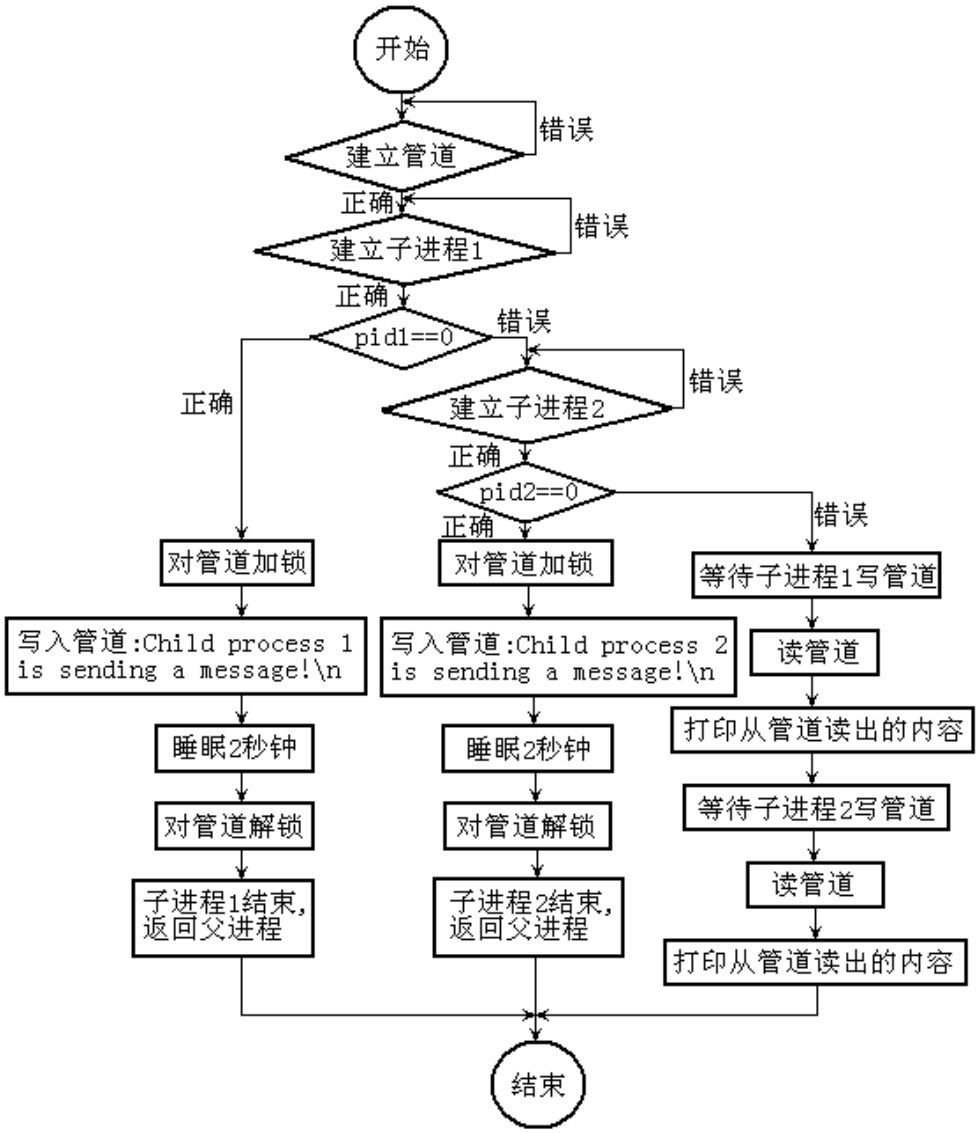
程序一:编写一个关于进程管道通信的简单程序,子进程送一串消息给父进程，父进程收到消息后把它显示出来。

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。



程序二：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)。

三 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。

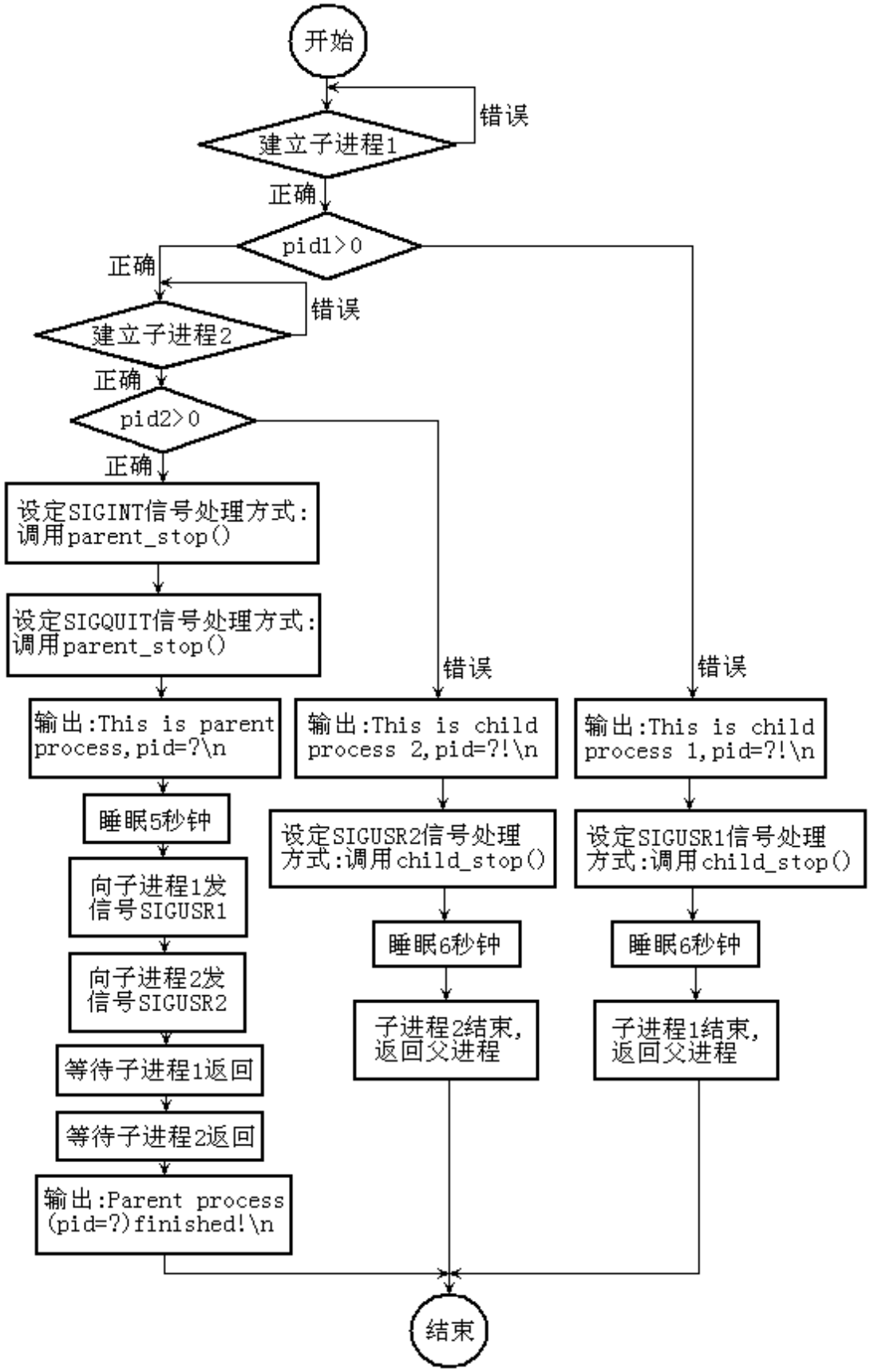
编写程序，使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号(即按 ctrl+c 或是 ctrl+\键)，5 秒钟内若父进程未接收到这两个软中断的某一个，则父进程用系统调用 kill()向两个子进程分别发送软中断信号 SIGUSR1 和 SIGUSR2，子进程获得对应的软中断信号，然后分别输出下列信息后终止：

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!



实验三　进程通信

【实验目的】

(1) 对理论课中学习的进程通信机制(IPC)：消息队列，共享内存，信号量的概念做进一步地理解，明确进程通信的原理；

(2) 加深理解进程通信所使用的各种方法的概念，认识进程通信机制的优点；

(3) 观察进程通信的过程，分析过程和原因，学习进程通信的方法。

【实验预备内容】

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 所关联的消息队列识别代号。如果参数 key 为 IPC_PRIVATE 则会建立新的消息队列，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 的消息队列。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，则表示消息队列已存在。此外，参数 msgflg 也用来决定消息队列的存取权限。

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

EACCESS　参数 key 所指的消息队列存在，但无存取权限；

EEXIST　欲建立 key 所指新的消息队列，但该队列已经存在；

EIDRM　参数 key 所指的消息队已经删除；

ENOENT　参数 key 所指的消息队列不存在，而参数 msgflg 也未设 IPC_CREAT 位；

ENOMEM　核心内存不足；

ENOSPC　超过可建立消息队列的最大数目。

(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 指定的消息送至参数 msqid 的消息队列内，参数 msgp 为 msgbuf 结构，起定义如下：

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

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 为消息数据的长度，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　无权限写入该消息队列；

EFAULT　参数 msgp 指向无效的内存地址；

EIDRM　参数 msqid 所指的消息队列已删除；

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 指定的消息读取信息出来，然后存于参数 msgp 所指定的结构内，参数 msgsz 为消息数据的长度，即 mtext 参数的长度，参数 msgtyp 是用来指定所要读取的消息种类：msgtyp=0 返回队列内第一项消息.msgtyp>0 返回队列内第一项 msgtyp 与 mtype 相同的消息。msgtyp<0 返回队列内第一项 mtype 小于或等于 msgtyp 绝对值的消息。参数 msgflg 可以设成 IPC_NOWAIT，指定若队列内没有消息可读，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消息数据长度，否则返回-1，错误原因存于 errno 中：

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

(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消息
    struct msg *msg_last;           //指向最后一个存于队列的消息
    time_t msg_stime;               //最后一次用 msgsnd()送入消息的时间
    time_t msg_rtime;               //最后一次用 msgrcv()读取消息的时间
    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()送入消息的进程识别码
    ipc_pid_t msg_lrpid;            //最后一个用 msgrcv()读取消息的进程识别码
}
```

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()

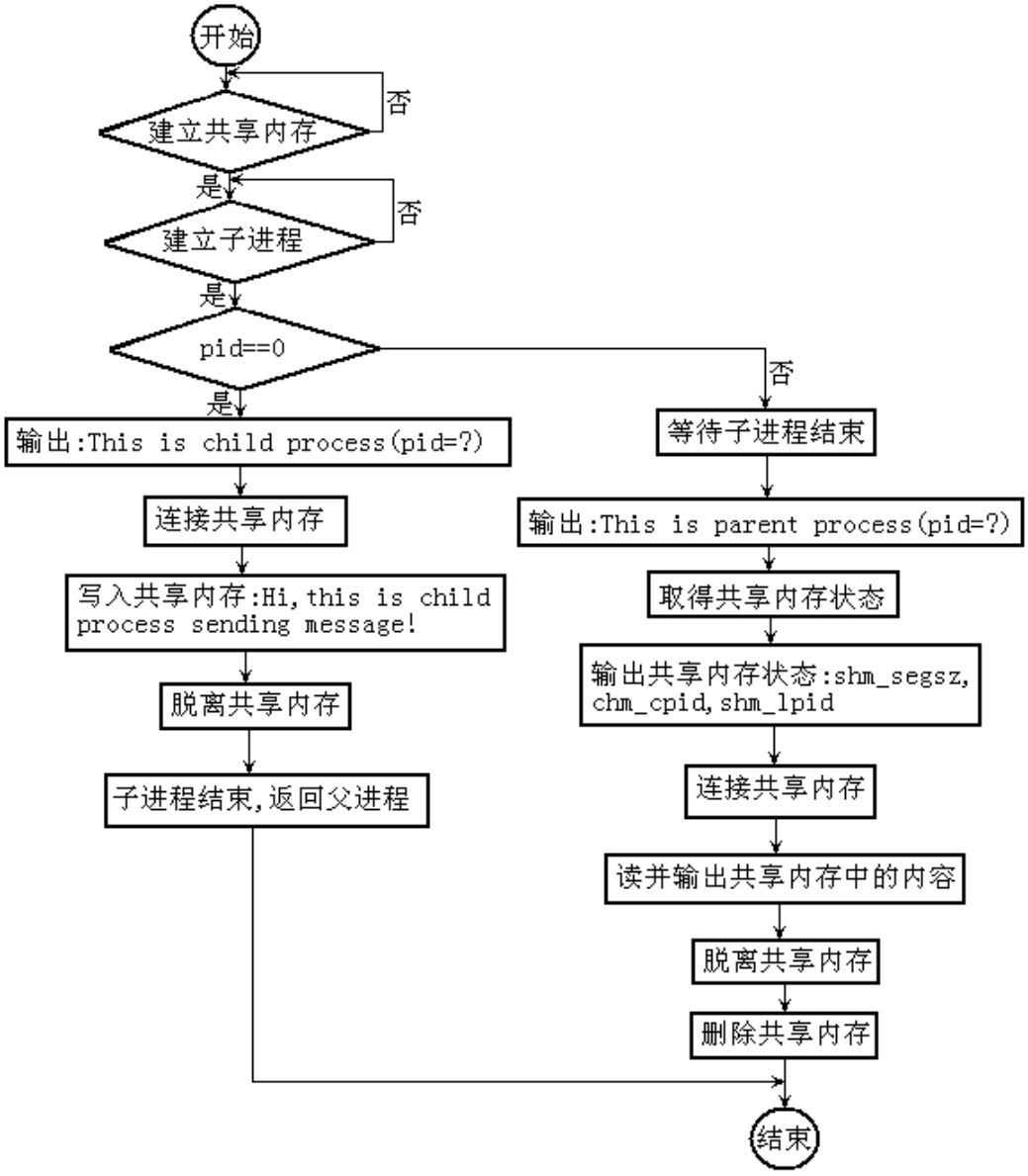
其中，共享内存可以使得两个或多个进程都能使用的地址空间中使用同一块内存.这就使得如果一个进程已经把信息写如到这一内存块中，马上就可以供其它共享这一块内存的进程使用.这样就能够克服诸如使用管道(PIPE)时候的几个问题:传递数据的大小限制(匿名管道大小为 4K，而共享内存大小可自定) ，传递数据的反复复制(可跟管道实验中的程序做比较) ，环境的切换(例如:试图读空管道的进程会被阻塞) 。

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。

【实验内容】

一.进程共享内存通信

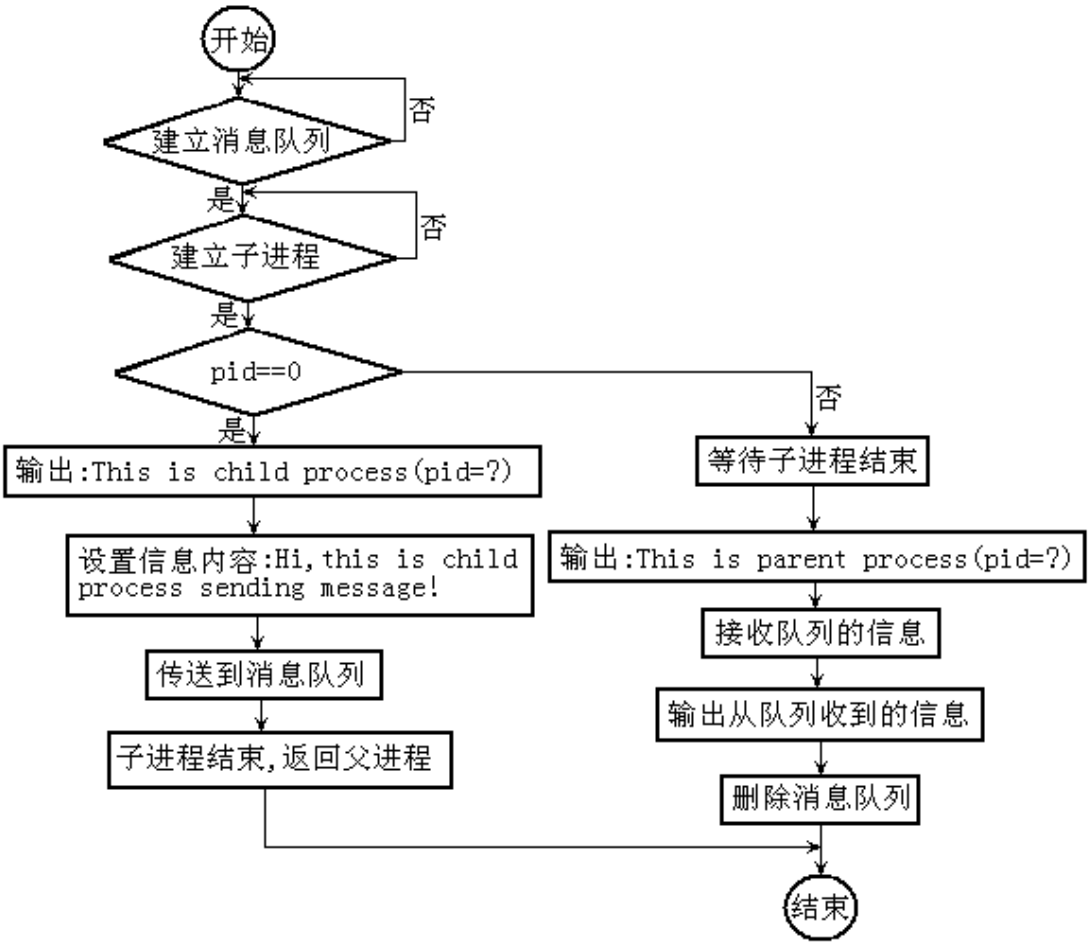
编写一个程序 sharedmem.c，在其中建立一个子进程，让父子进程通过共享内存的方法实现通信。其中，父进程创建一个共享内存段，然后由子进程将该共享内存附加到自己的地址空间中，并写入该共享内存下列信息：Hi,this is child process sending message!。在等待子进程对共享内存的操作完成后，父进程将该共享内存附加到自己的地址空间中，并读出该共享内存中的信息，与该共享内存段的基本信息(大小，建立该共享内存的进程 ID，最后操作该共享内存段的进程 ID)一并显示出来。



二.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。

(2). 编写程序 msg2.c 和 msg3.c。在 msg2.c 中建立一个信息队列，表明身份后，向消息队列中写如下列 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。



在前一个消息队列实验中，子进程通过建立的时候继承父进程的 msgqid 变量名和值，直接使用该变量通过消息队列与父进程通信。

在后一个实验中，在不同的 shell 中先后执行编译好的 msg2 和 msg3 程序，可见，没有父子继承关系的两个进程使用同一个消息队列实现了通信。