```
then
    case $2 in
    +) let z=$1+$3;;
    -) let z=$1-$3;;
    /) let z=$1/$3;;
    x | X) let z=$1*$3;;
    *) echo "Warning-$2 invalied operator，only +、−、×、÷ "operator allowed.
       exit;;
    esac
        echo "Answer is $z"
    else
        echo "Usage-$0 value1 operator value2."
    fi
```

5.3 实 验 内 容

5.3.1 实验 1 编写一个简单的 Shell 程序——MyShell

5.3.1.1 实验说明

设计并实现一个简单的交互式 Shell——MyShell。MyShell 要具备如下功能：

- 支持程序后台运行。
- 支持重定向。
- 支持管道。
- 支持设置搜索路径。
- 支持内置命令：cd(切换目录)、exit(退出 Shell)和 path(设置搜索路径)。

5.3.1.2 解决方案

1. 总体结构

根据实验要求，MyShell 可以组织成图 5-3 所示的总体结构。

2. 交互界面显示

通常 Shell 会在命令提示符中根据用户的配置文件显示响应信息。在 MyShell 中，只需要固定显示当前用户关联的登录名、主机网络名、当前目录名即可。显示格式为：

```
[username@servername:pathname]#
```

要完成该功能，需要涉及的几个函数为：

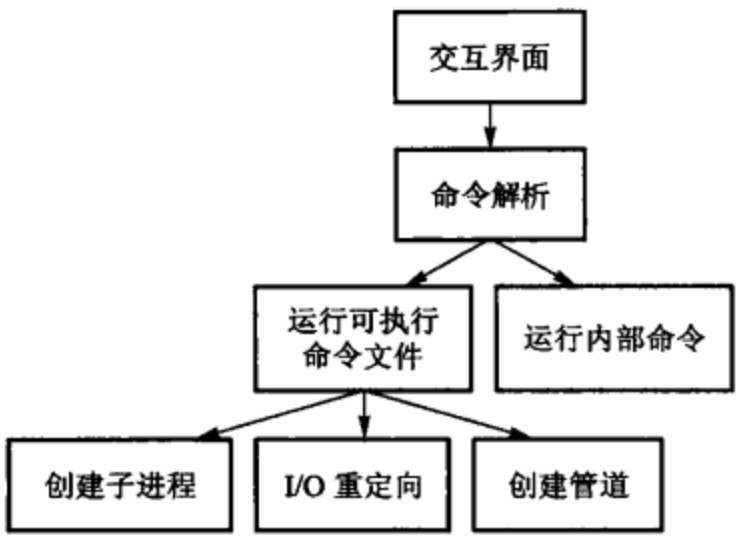

图 5-3 MyShell 总体结构

```
#include<unistd.h>
char *getlogin(void);
/*getlogin 函数返回的是与当前用户关联着的登录名*/
#include<unistd.h>
int gethostname(char *name, size_t namelen);
/*gethostname 函数把机器的网络名写到字符串 name 中。该字符串的长度至少要有 namelen 个字符。
gethostname 函数在成功时返回 0，否则返回-1*/
#include<unistd.h>
char *getcwd( char *buf, size_t size);
/*getcwd 函数的作用是把当前目录的名字写到给定的缓冲区 buf 里。如果子目录的名字超过参数 size
给出的缓冲区长度，返回 NULL。如果操作成功，它返回指针 buf*/
```

3. 命令解析

为了得到命令行，MyShell 执行一个阻塞型读操作，因此执行 MyShell 进程时将会被阻塞，直到用户根据提示符输入又一个命令行。MyShell 可以通过 gets( )获得用户输入的命令行，在获得用户的输入后，需要对输入进行解析，以获得命令和参数。在解析过程中，需要注意：

- 判断命令是否是内置命令。
- 是否包含“<”、“>”等字符，如包含，表明需要进行重定向。
- 是否包含“&”字符，如包含，表明命令要放入后台执行。
- 是否包含“|”字符，如包含，表明有多个命令，并要创建管道。

4. 内置命令

MyShell 需要处理 3 个内置命令：cd、path 和 exit。

cd 命令用于切换当前目录。可以通过 chdir()函数更改当前目录：

```
#include<unistd.h>
```

```
int chdir(const char *pathname);
/*pathname 为新目录名。如果切换成功，chdir 返回 0；否则返回–1 */
```

path 命令用于设置 MyShell 的搜索路径。在 MyShell 中，需要用一个全局变量 gpath 记录用户设置的搜索路径。当用户设置新搜索路径时，MyShell 对 gpath 变量进行更新。

exit 命令用于退出 MyShell：

如果用户输入“exit”命令，MyShell 可以通过 exit()函数退出程序：

```
#include <stdlib.h>
void exit(int status);
/*status 为程序的退出码*/
```

在 Linux 中，通常退出码为 0 时，表示程序正常退出。在 MyShell 中，用户输入 exit 命令是一个正常的退出请求，可以将退出码置为 0。

5. 执行命令

如果用户输入的不是内置命令，MyShell 需要在用户设置的路径中搜索命令，并测试该命令是否可执行。可以通过 access( )函数进行测试：

```
#include<unistd.h>
int access(const char *pathname, int mode);
```

access( )按照用户 ID 和组 ID 进行存取许可的测试。pathname 为待测试文件的路径名；mode 为需要进行的测试，常用的 mode 常数有 R_OK(测试读许可权)、W_OK(测试写许可权)、X_OK(测试执行许可权)和 F_OK(测试文件是否存在)。如果测试成功，access( )函数返回 0。在搜索时需要注意，用户可能已提供绝对路径名作为命令名单词或者根据 path 命令设置的搜索路径进行限定的相对路径名。此“/”、“./”和“../”开头的名称是可以用于启动执行的绝对路径名。否则 MyShell 需要在用户定义的搜索路径中进行查找。

在 Linux 系统中，执行一个命令通常使用 fork( )和 exec( )。MyShell 可以通过调用 fork( )创建一个子进程，通过 exec( )改变子进程当前执行的程序。

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

由 fork( )函数创建的新进程被称为子进程。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值是新子进程的进程 ID。因为一个进程可以创建多个子进程，为了区分这些子进程，子进程需要将自己的进程 ID 返回给父进程。

新创建的进程是父进程的副本，子进程需要通过 exec( )类函数执行新功能。当进程调用一种 exec( )函数时，该进程完全由新进程替代，而新进程则从其 main( )函数开始执行。exec( )类函数并不创建新进程，而只是用另一个新进程替换当前进程的正文、数据和堆栈段。MyShell 通过调用 execve( )实现进程的替换：

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

pathname参数是将要被执行的新进程的文件的路径名，argv是一个参数字符串列表，而envp数组是一个环境变量字符串和值的列表，它们将在进程开始执行新进程时使用。execve( )如果成功不会返回，如果失败返回–1。

MyShell在执行命令时还需要考虑是否将该命令放入后台执行。如果将命令放入后台执行，父进程(MyShell)需要在fork( )成功之后，直接将命令提示符打印出来。如果需要将程序放在前台运行，则父进程需要等待子进程运行结束，可以通过执行waitpid( )函数等待子进程运行结束。

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

参数pid表示需要等待的子进程的进程ID。statloc为用户存储子进程退出时的退出状态，MyShell并不关心子进程的退出状态，因此可以将statloc设置为NULL。options参数一般设置为0。

执行命令的大体框架为：

```
if  ((pid=fork( ))==0 ){
    /*子进程*/
    execve(path, argv, envp);
}else{
    /*父进程*/
    if ( forground )                    /*前台运行*/
        waitpid( pid, &status, 0 );     /*等待子进程退出*/
}
```

6. I/O重定向

已经打开的文件描述符将会在fork( )和exec( )调用中保持下来，利用这一点可以实现文件的输入、输出重定向，为了说明如何实现这一点，需要了解内核用于I/O的数据结构。

每个进程描述符中包含一个files_struct结构，用来记录用户的文件打开情况，这个结构称为用户打开文件表。指向该结构的指针被保存在进程的task_struct结构的成员files中。用户打开的每一个文件都占用打开文件表中的一项，表中的每一项指向系统中的一个文件对象(file)。进程在进行I/O操作时，需要提供文件描述符。文件描述符是用户打开文件表的索引，内核通过文件描述符查找到相应的文件对象。例如，用户调用write(1,buf,count)时，内核会查找到文件描述符表中的第1项所对应的文件对象，并对相关文件进行写操作。

内核为系统中每个打开文件创建一个文件对象，每一个文件对象用一个file结构表示。每一个文件对象中包含文件偏移量，读写权限等信息。每当进程打开一个文件时，内核从files_struct结构中找一个空闲的文件描述符，使它指向文件对象file。当调用fork( )生成一个子进程时，子进程复制父进程的文件描述符，两者有相同的用户打开文件表，都有表项指向同一个file结构。当调用exec( )时，文件描述表也不会发生变化。因此子进程能够继承其父进程的文件描述符。图5-4显示父进程与子进程及打开文件表的关系。

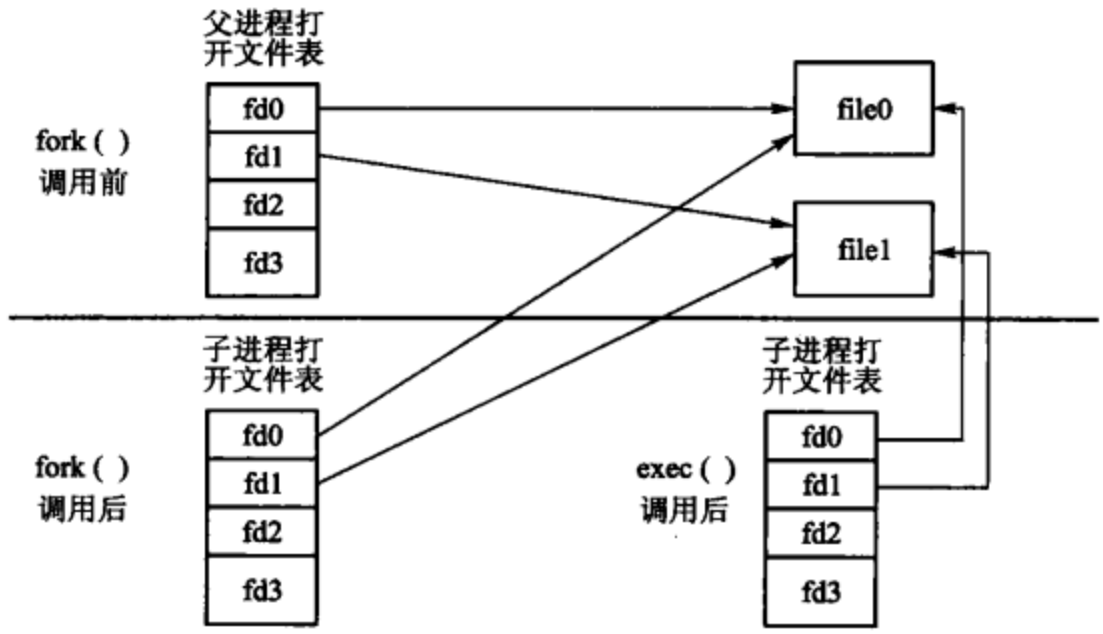

图 5-4 父进程、子进程与打开文件表关系

由于文件描述符是指向文件对象的指针，因此不同的文件描述符可以指向同一个文件对象，如图 5-5 所示。在没有进行 I/O 重定向时，文件描述符 0 指向标准输入，即指向键盘。在实现输入重定向时，程序需要先打开一个文件，内核会为该文件创建一个文件对象，然后，程序将文件描述符 0 指向已打开文件对应的文件对象。在这之后，程序通过标准输入(文件描述符 0)读取的数据便来自先前打开的文件。输出重定向的实现方法和输入重定向类似。

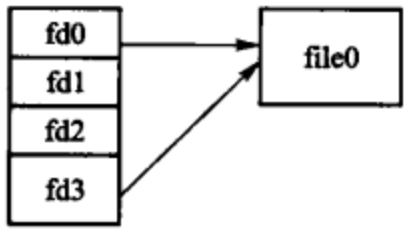

图 5-5 不同文件描述符指向同一文件对象

完成 I/O 重定向可以使用的函数有：

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

dup( )函数将指定的文件描述符 filedes 复制到当前第 1 个可用的文件描述符中。如果需要使用 dup( )函数完成 I/O 重定向，需要先将标准输入或输出关闭，例如：

```
fid = open( foo, O_WRONLY|O_CREAT);
close(1);
dup(fid);
```

在该段代码中，进程先打开准备用于输出重定向的文件，然后关闭标准输出。这时使用 dup( )

函数将把 fid 复制到标准输出的文件描述符中，其文件描述符表的变化如图 5-6 所示。

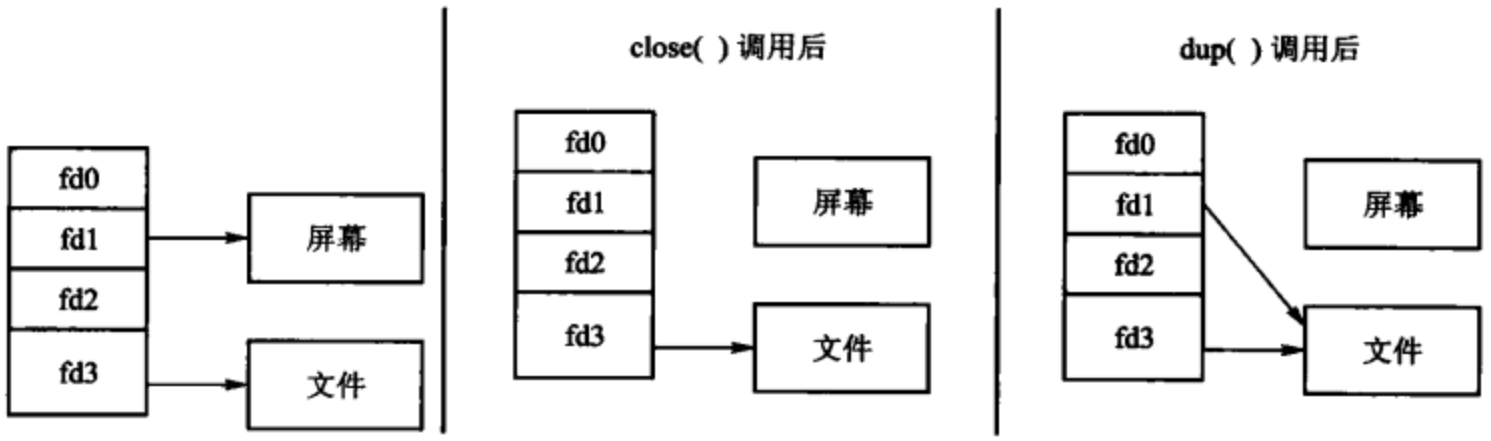

图 5-6 文件描述符变化过程

使用 close( )和 dup( )函数完成重定向是不安全的。因为在 close( )调用之后、dup( )调用之前，可能产生信号，在信号处理过程中可能修改文件描述符，这将导致 dup( )函数中的文件描述符复制错误。为了解决这个问题，需要使用 dup2( )函数，dup2( )函数将 filedes 文件描述符复制到 filedes2 文件描述符中。如果 filedes2 没有关闭，dup2( )函数将先把 filedes2 关闭。dup2( )函数是一个原子操作，因此在该函数的执行过程中不会被信号打断，能够避免使用 close( )和 dup( )函数可能产生的问题。

MyShell 实现 I/O 重定向的主要流程为：

```
if  ((pid=fork( ))==0 ){
        /*子进程*/
        /*实现输出重定向*/
        fid = open( foo, O_WRONLY|O_CREAT);
        close(1);
        dup(fid);
        execve(path, argv, envp);
}else{
        /*父进程*/
        if ( forground )                    /*前台运行*/
              waitpid( pid, &status, 0 );   /*等待子进程退出*/
}
```

7. 管道

管道在使用时有如下两种限制：

- 管道是半双工的，数据只能在一个方向上流动。
- 管道只能在具有公共祖先的进程之间使用。

通常，一个管道由一个进程创建，然后该进程调用 fork( )，此后父、子进程之间就可以使用管道，管道是调用 pipe( )函数创建的：

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

经由参数 fd 返回两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。fd[1]的输出是 fd[0]的输入。pipe( )函数调用成功后，函数返回 0，否则返回–1。

单个进程中的管道几乎没有任何用处。通常，调用 pipe( )的进程接着调用 fork( )，这样就创建了从父进程到子进程或反之的管道。图 5-7 显示了这种情况。

使用 fork( )函数之后做什么取决于所需要的数据流的方向。对于从父进程到子进程的管道，父进程关闭管道的读端(fd[0])，子进程则关闭写端(fd[1])，图 5-8 显示描述符的最后安排。

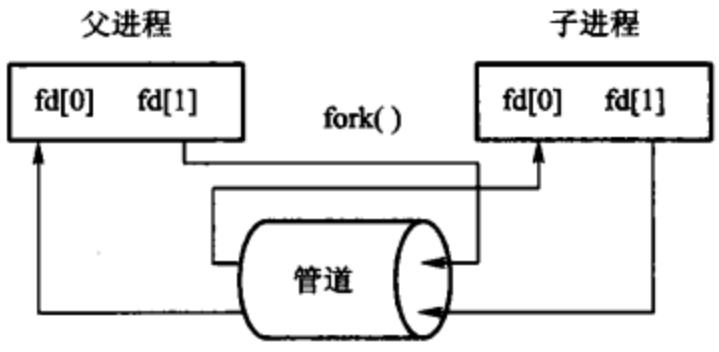

图 5-7 使用 fork( )函数之后的半双工管道

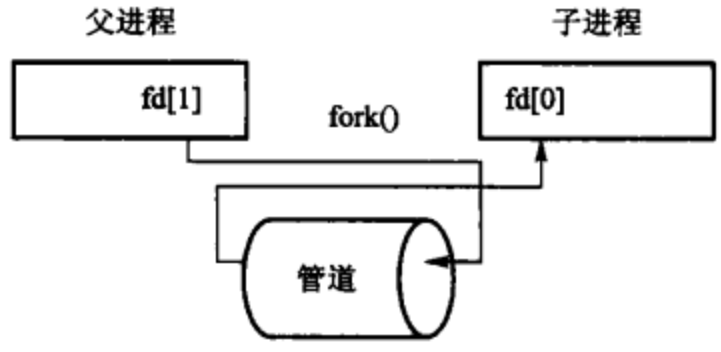

图 5-8 从父进程到子进程的管道

MyShell 需要实现的管道是将进程 1 的标准输出变成进程 2 的标准输入。实现该功能需要的步骤为：

① 创建进程 1。

② 在进程 1 中调用 pipe( )创建管道，并得到管道描述符 fd[0],fd[1]。

③ 在进程 1 中创建进程 2，使进程 2 成为进程 1 的子进程。

④ 进程 1 关闭描述符 fd[0]。

⑤ 进程 1 调用 dup2(fd[1],1)，将管道描述符复制到标准输出。

⑥ 进程 1 调用 exec( )执行新程序。

⑦ 进程 2 关闭描述符 fd[1]。

⑧ 进程 2 调用 dup2(fd[0],0)，将管道描述符复制到标准输入。

⑨ 进程 2 执行 exec( )执行新程序。

完成上述步骤后，进程 1 的标准输出将自动通过管道传递到进程 2 的标准输入中。代码的框架如下：

```
{   …
    int fd[2];
    pipe(fd);
    if ( fork( )>0 ) {
        /*父进程*/
        close(fd[0]);
```

```
            dup2(fd[1],1);
            exec(…);
        }else{
            close(fd[1]);
            dup2(fd[0],0);
            exec(…);
        }
        …
}
```

5.3.1.3 程序框架

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

void init( );
void setpath(char* newpath);                                    /*设置搜索路径*/
int read_command( );                                            /*获得用户输入*/
int is_internal_cmd(char*cmd, int cmdlen);                      /*解析内部命令*/
int is_pipe(char*cmd, int cmdlen);                              /*解析管道*/
int is_io_redirect(char*cmd, int cmdlen);                       /*解析重定向*/
int normal_cmd(char*cmd, int cmdlen, int infd, int outfd, int fork);   /*执行普通命令*/

void init( )
{
   /*设置命令提示符*/
   /*设置默认的搜索路径*/
   setpath("/bin:/usr/bin");
}

int read_command( )
{
   /*读取用户命令，如果用户输入 exit，程序退出*/
}

void setpath(char* newpath)
{
```

```
  /*解析路径，并设置搜索路径*/
}

int is_internal_cmd(char* cmd, int cmdlen)
{
  /*判断是否是内部命令(cd，path)*/
}

int is_pipe(char* cmd, int cmdlen)
{
  /*为命令创建管道*/
}

int is_io_redirect(char* cmd, int cmdlen)
{
   /*进行 IO 重定向*/
}

/*搜索命令*/
char* find_exe(const char* exepath)
{
   /*在已设置的目录中搜索命令*/
}
/*执行命令*/
void fork_and_exec( char* path, char* argv[], int is_background, int infd, int outfd)
{
    …
}
/*解析命令参数*/
char** parse_argv(char* path)
{
     /*解析命令参数*/
}
/*执行一个普通命令*/
int normal_cmd(char* cmd, int cmdlen, int infd, int outfd, int fork)
{
  …
}
int main( int argc, char *argv[] )
{
```

```
    init( );
    while ( read_command( ) ){
      if ( is_internal_cmd(cmdbuf, cmdlength) )
        continue;
      if ( is_pipe(cmdbuf, cmdlength) )
        continue;
      if ( is_io_redirect(cmdbuf, cmdlength) )
        continue;
      normal_cmd(cmdbuf, cmdlength, –1, –1, 1);
    }
    return 0;
}
```

5.3.2 实验 2 基于 Shell 的网络管理

5.3.2.1 实验说明

该实验要求实现两个基本功能：

- 通过 Shell 编程访问某个 URL(如 www.nju.edu.cn)*N* 次。
- 在网关节点捕获指定计算机的 MAC 地址(如 00:11:5B:A3:65:F8)，并限制其访问。

5.3.2.2 解决方案

对 URL 的访问可通过方法 get 实现，而访问次数可通过 while 语句实现。对 arp 地址的获取可通过 arp 实现，对指定 MAC 地址的访问则可通过 grep 得到。而对 IP 地址的提取可通过 awk 和 sed 工具实现。

5.3.2.3 程序框架

```
/*****基于 Shell 的网络管理程序框架*****/
url="http://cs.nju.edu.cn /"
looptime=0
loopcount=N
/*变量置初值*/
echo "$(date | cut –c 10-18)"
/*打印开始时间*/
while [ $looptime –lt $loopcount ]
do
        content=$(GET    $url )
        let "looptime = $looptime + 1"
done
```

```
/*通过一个 while 循环，访问 url 变量对应的网页 loopcount 次*/
echo "$(date | cut –c 10-18)"
echo "done"
/*打印结束时间*/
arp –a | grep "00:11:5B:A3:65:F8" > ctx
/*第 1 句是得到对应 MAC 地址为“00:11:5B:A3:65:F8”的包含其 IP 地址的字符串*/
awk '{printf $2}' ctx   > ipaddress
sed –n 's/(// w temp' ipaddress
sed –n 's/)// w pureip' temp
/*使用 awk 和 sed 工具，提取出 IP 对应的字符串*/
RES_IP= 'cat pureip'
echo $RES_IP
/*将 IP 地址赋值给 RES_IP 变量，并打印在屏幕上*/
iptables –A FORWARD –s $RES_IP –d $RES_IP   –j REJECT
/*使用 iptables 工具，将 FORWARD 链置位 reject(包括到该 IP 的和从该 IP 出来的包)*/
rm temp
rm ipaddress
/*删除两个临时变量*/
```