**Linux下一个功能完备 shell**

**汪宇 2706005012**

**指导老师 :**

这个程序实现了一个功能完备的linux下的shell,实现了 | 串联命令, 实现了以>,< ,>>标置的IO重定向,实现了&后台运行)

这个shell定义了2个头文件和5个源文件,分别是 wy.h , shell.h, gettoken.c

invoke.c command.c redirect.c main.c

gettoken函数实现了一个词法分析功能分析,输入的下一个token是什么,其被command函数调用

各文件及源代码及说明注释如下

gettoken函数是用来返回命令中的下一个符号是什么, 可以是一个单词(命令名,参数,文件名都归为单词),分隔符 | ,后台运行符&,分号;,以及IO重定向符<,>, 或定向输出且是添加到输出文件中的>>

#include"shell.h" //

typedef enum {T\_WORD , T\_BAR , T\_AMP , T\_SEMI , T\_GT , T\_GTGT , T\_LT , T\_NL , // T\_NL means a new line

T\_EOF , T\_ERROR} TOKEN ;

static bool store\_char(char \* word , int maxword , int c , int \* np);//只在gettoken.c文件中使用的函数

TOKEN gettoken(char \*word , int maxword)

{

enum { NEUTRAL , GTGT , INQUOTE , INWORD} state = NEUTRAL ;

int c;

int wordn=0;

while((c=getchar())!=EOF)

{

switch (state)

{

case NEUTRAL :

switch (c)

{

case ';':

return T\_SEMI;

case '&':

return T\_AMP;

case '|':

return T\_BAR;

case '<':

return T\_LT;

case '\n':

return T\_NL;

case ' ':

case '\t':

continue;

case '>':

state = GTGT;

continue;

case '"':

state = INQUOTE;

continue;

default :

state = INWORD;

store\_char(word , maxword , c , &wordn);

continue;

}

case GTGT :

if (c=='>')

return T\_GTGT;

ungetc(c,stdin);//extern int ungetc (int \_\_c, FILE \*\_\_stream); <stdio.h>中定义的函数

return T\_GT;

case INQUOTE :

switch (c)

{

case '\\' :

if((c=getchar()) == EOF)

c = '\\';

store\_char(word , maxword , c ,&wordn);

continue;

case '"' :

store\_char(word , maxword , '\0' , &wordn);

return T\_WORD;

default:

store\_char(word , maxword , c ,&wordn);

continue;

}

case INWORD:

switch (c)

{

case ';':

case '&':

case '|':

case '<':

case '>':

case '\n':

case ' ':

case '\t':

ungetc(c,stdin);

store\_char(word , maxword ,'\0', &wordn);

return T\_WORD;

default :

store\_char(word , maxword ,c , &wordn);

continue;

}

}

}

return T\_EOF;

}

// 这个函数把字符c(定义为int 型了)存储在字符数组word中,并通过(\*np)来计数字符数组中存了多少个字符

static bool store\_char(char \* word, int maxword , int c , int \* np)

{

word[(\*np)++] = c;

return true;

}

下面是command.c文件,实现command.c函数,command函数每调用一次,用来处理每一次回车键按下所要处理的命令,可以是一个simple command,也可以是由| 串起来的一个pipeline

#include"shell.h"

#define MAXARG 50 //max args in command

#define MAXFNAME 500 //max chars in file name

#define MAXWORD 500 //max chars in arg

extern pid\_t invoke(int argc , char \*argv[] , int srcfd , const char \* srcfile,

int dstfd, const char \* dstfile , bool append , bool bckgrnd , int closefd); //closefd ?

TOKEN command(pid\_t \* wpid , bool makepipe , int \* pipefdp)

{

TOKEN token , term; //定义了两个token,term是结束一次命令最后的那个token,并用作command函数的返回值

int argc , srcfd , dstfd , pid ,pfd[2] ={-1,-1};

char \*argv[MAXARG], srcfile[MAXFNAME]="", dstfile[MAXFNAME]="";

char word[MAXWORD];

bool append;

argc = 0; //用来计数参数值,初始化为0

srcfd = STDIN\_FILENO;

dstfd = STDOUT\_FILENO;

while (true)

{

switch (token = gettoken(word , sizeof(word)))

{

case T\_WORD:

if (argc >=MAXARG - 1)

{

fprintf(stderr , "too many args\n");

continue;

}

if((argv[argc]=(char \*)malloc(strlen(word)+1))==NULL)

{

fprintf(stderr,"out of arg memory\n");

continue;

}

strcpy(argv[argc],word);

argc++;

continue;

case T\_LT:

if (makepipe)

{

fprintf(stderr, "extra <\n");

break;

}

if (gettoken(srcfile ,sizeof(srcfile)) != T\_WORD)

{

fprintf(stderr,"illegal <\n");

break;

}

srcfd = -1;

continue;

case T\_GT:

case T\_GTGT:

if (dstfd != STDOUT\_FILENO)

{

fprintf(stderr , "extra > or >> \n");

break;

}

if (gettoken(dstfile, sizeof(dstfile))!=T\_WORD)

{

fprintf(stderr , "illegal > or >> \n");

break;

}

dstfd = -1;

append=true; // 已经得到了>>

continue;

case T\_BAR:

case T\_AMP:

case T\_SEMI:

case T\_NL:

argv[argc]=NULL;

if(token==T\_BAR) //遇到| 则command被递归调用

{

if(dstfd!=STDOUT\_FILENO)

{

fprintf(stderr, "> or >> conflicts with | \n"); // 每一次的command调用处理一个simple command, 每一个simple command都有其输入输出

break; // 这是退出怎么while loop

}

term = command(wpid , true , &dstfd);

if(term == T\_ERROR)

return T\_ERROR;

}

else

term = token;

if(makepipe)

{

pipe(pfd);

\*pipefdp = pfd[1];

srcfd = pfd[0];//

}

pid = invoke(argc , argv , srcfd , srcfile , dstfd , dstfile , append , term == T\_AMP , pfd[1]);

//

if (token != T\_BAR)

\*wpid = pid; // wpid is used for what?

while(--argc>=0)//??empty argv for the next use

free(argv[argc]);

return term;

case T\_ERROR:

return T\_ERROR;

}

}

}

下面是invoke.c文件,其实现了invoke.c函数,invoke函数被command函数所调用,用来处理每一个单独的命令,其会为检查每一个单独命令的输 入输 出,将其设为正确的形式,重置 Io的任务放到了一个单独的函数 redirect中来完成的,其放在redirect.c 函数中,待会将会写出来

invoke函数会为每一个单独的命令fork()出 个进程来执行它

#include"shell.h"

pid\_t invoke(int argc, char \*argv[], int srcfd, const char \* srcfile,

int dstfd, const char \* dstfile, bool append, bool bckgrnd, int closefd)

{

pid\_t pid;

char \* cmdname, \*cmdpath;//must be written like this,learn why from compilation later

//if(argc==0 || builtin(argc,argv,srcfd,dstfd))

// return 0;

switch(pid = fork())

{

case -1:

fprintf(stderr, "Can't create new process\n");

return 0;

case 0:

//there are 4 sentences of error handling which I didn't write now

redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd);

//cmdname = strchr(argv[0],'/');

cmdname = argv[0];

cmdpath = argv[0];//the code in the book is not like this, but i don't understand

execvp(cmdpath,argv);

//?

exit(0);

}

return pid; //parent

}

下面是redirect.c文件,里面实现了 redirect函数,用来设置好每个命令的IO.

//invoke calls (redirect) to redirect I/O and to ensure that the source and destination are duped to be STDIN\_FILENO and STDOUT\_FILENO

#include"shell.h"

void redirect(int srcfd, const char \* srcfile, int dstfd, const char \* dstfile, bool append, bool bckgrnd)

{

int flags;

if (srcfd == STDIN\_FILENO && bckgrnd)

{

srcfile = "/dev/null";

srcfd = -1;

}

if (srcfile[0] != '\0')

srcfd = open(srcfile , O\_RDONLY, 0);

dup2(srcfd, STDIN\_FILENO);// STDIN\_FILENO is a constant

if (srcfd != STDIN\_FILENO)

close(srcfd);

if (dstfile[0] != '\0')

{

flags = O\_WRONLY | O\_CREAT;

if (append)

flags |= O\_APPEND;

else

flags |= O\_TRUNC;

dstfd = open(dstfile, flags, PERM\_FILE);

}

dup2(dstfd, STDOUT\_FILENO);

if (dstfd != STDOUT\_FILENO)

close(dstfd);

return;

}

特别注意,代码中一两行是这样的

if (srcfd == STDIN\_FILENO && bckgrnd)

{

srcfile = "/dev/null";

srcfd = -1;

}

这是当一个命令需要在后台运行的时候,且其默认输入又设为标准输入设备的时候,其不能读入任何数据,此时将其输入置为 /dev/null , 这是一个空文件,一旦去读,便会报错.

下面便是 main 函数了,很简单,一个while无限循环接受输入,一次循环处理一回车键所按下的命令

#include"shell.h"

#define PROMPT "wy :"

int main(int argc, char \*\* argv)

{

pid\_t pid;

TOKEN term = T\_NL;

while (true)

{

if( T\_NL == term)

printf(PROMPT);

term = command(&pid , false , NULL);

}

}

如果最后按下了回的话,打印PROMT 提示符,其定义为一个宏,这里定义为”wy”

另外注意到,我们为这个程序写了一个头文件 shell.h,里面包含了一系列的标准头文件,声明了各个函数,定义了一个枚举类型,用于我们的gettoken词法分析器,还定义了几个宏

//from here,learn how to write a head file for a project(for a software)

#ifndef \_SHELL\_H

#define \_SHELL\_H

#include"../wy.h"

typedef enum {T\_WORD, T\_BAR, T\_AMP, T\_SEMI, T\_GT, T\_GTGT, T\_LT, T\_NL, T\_EOF, T\_ERROR} TOKEN;

extern TOKEN gettoken (char \*word,int maxword);

extern void redirect(int srcfd, const char \* srcfile, int dstfd , const char \* dstfile, bool append, bool bckgrnd);

extern TOKEN command (pid\_t \*wpid , bool makepipe , int \*pipeline);

extern pid\_t invoke(int argc , char \*argv[] , int srcfd , const char \* srcfile,

int dstfd, const char \* dstfile , bool append , bool bckgrnd , int closefd); //closefd ?

#endif // \_SHELL\_H

其中wy.h又是另外一个头文件,里面包含了一系列的标准头文件

#ifndef \_WY\_H

#define \_WY\_H

#define SA struct sockaddr //attention, struct must be used like this

#define MAXLINE 100

#define PERM\_FILE (S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH)

#include<fcntl.h> // O\_APPEND .. ard defined here

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

#include<stddef.h>

#include<string.h>

#include<unistd.h>

#include<signal.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<sys/termios.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

#include<sys/un.h>

#include<bits/signum.h>

#define SERV\_PORT 9877

#define SERV\_PORT\_STR "9877"

#ifndef TIOCGWINSZ

#include<sys/ioctl.h>

#endif

#endif // \_WY\_H

编程程序的命令为

$ g++ -o shell main.c redirect.c invoke.c gettoken.c command.c

生成目标文件./shell

执行这个命令

$ ./shell

下面是一些运行结果截图



下面是另一些截图,试验了我们实现的 输出重定向 >功能

