// MyShell.

//Problems: Where to implement NONOHUP? Hardcode in main() or try to implement it like the other funtionalities.

// We create a void function and somehow call it in main()

#include "types.h"

#include "user.h"

#include "fcntl.h"

// Parsed command representation

#define EXEC 1

#define REDIR 2

#define PIPE 3

#define LIST 4

#define BACK 5

#define NONOHUP 6

#define MAXARGS 10

struct cmd {

int type;

};

struct execcmd {

int type;

char \*argv[MAXARGS];

char \*eargv[MAXARGS];

};

struct redircmd {

int type;

struct cmd \*cmd;

char \*file;

char \*efile;

int mode;

int fd;

};

struct pipecmd {

int type;

struct cmd \*left;

struct cmd \*right;

};

struct listcmd {

int type;

struct cmd \*left;

struct cmd \*right;

};

struct backcmd {

int type;

struct cmd \*cmd;

};

int fork1(void); // Fork but panics on failure.

void panic(char\*);

struct cmd \*parsecmd(char\*);

// Execute cmd. Never returns.

void

runcmd(struct cmd \*cmd)

{

int p[2];

struct backcmd \*bcmd;

struct execcmd \*ecmd;

struct listcmd \*lcmd;

struct pipecmd \*pcmd;

struct redircmd \*rcmd;

if(cmd == 0)

exit();

switch(cmd->type){

default:

panic("runcmd"); //if illegal type, then print runcmd

case EXEC:

ecmd = (struct execcmd\*)cmd;

if(ecmd->argv[0] == 0)

exit();

exec(ecmd->argv[0], ecmd->argv);

printf(2, "exec %s failed\n", ecmd->argv[0]);

break;

case REDIR:

rcmd = (struct redircmd\*)cmd;

close(rcmd->fd);

if(open(rcmd->file, rcmd->mode) < 0){

printf(2, "open %s failed\n", rcmd->file);

exit();

}

runcmd(rcmd->cmd);

break;

case LIST:

lcmd = (struct listcmd\*)cmd;

if(fork1() == 0)

runcmd(lcmd->left);

wait();

runcmd(lcmd->right);

break;

case PIPE:

pcmd = (struct pipecmd\*)cmd;

if(pipe(p) < 0)

panic("pipe");

if(fork1() == 0){

close(1);

dup(p[1]);

close(p[0]);

close(p[1]);

runcmd(pcmd->left);

}

if(fork1() == 0){

close(0);

dup(p[0]);

close(p[0]);

close(p[1]);

runcmd(pcmd->right);

}

close(p[0]);

close(p[1]);

wait();

wait();

break;

case BACK:

bcmd = (struct backcmd\*)cmd;

if(fork1() == 0)

runcmd(bcmd->cmd);

break;

}

exit();

}

int

getcmd(char \*buf, int nbuf)

{

printf(2, "$S20 "); //the shell prompt

memset(buf, 0, nbuf);

gets(buf, nbuf);

if(buf[0] == 0) // EOF

return -1;

return 0;

}

int

main(void)

{

static char buf[100];

int fd;

// Ensure that three file descriptors are open.

while((fd = open("console", O\_RDWR)) >= 0){

if(fd >= 3){

close(fd);

break;

}

}

// Read and run input commands.

while(getcmd(buf, sizeof(buf)) >= 0){

if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){

// Chdir must be called by the parent, not the child.

buf[strlen(buf)-1] = 0; // chop \n

if(chdir(buf+3) < 0) //if it can't cd

printf(2, "cannot cd %s\n", buf+3);

continue;

}

if(fork1() == 0) //otherwise run runcmd

runcmd(parsecmd(buf));

wait();

}

exit();

}

void

panic(char \*s)

{

printf(2, "%s\n", s);

exit();

}

int

fork1(void)

{

int pid;

pid = fork();

if(pid == -1)

panic("fork"); //if fork fails

return pid;

}

//PAGEBREAK!

// Constructors

struct cmd\*

execcmd(void)

{

struct execcmd \*cmd;

cmd = malloc(sizeof(\*cmd));

memset(cmd, 0, sizeof(\*cmd));

cmd->type = EXEC;

return (struct cmd\*)cmd;

}

struct cmd\*

redircmd(struct cmd \*subcmd, char \*file, char \*efile, int mode, int fd)

{

struct redircmd \*cmd;

cmd = malloc(sizeof(\*cmd));

memset(cmd, 0, sizeof(\*cmd));

cmd->type = REDIR;

cmd->cmd = subcmd;

cmd->file = file;

cmd->efile = efile;

cmd->mode = mode;

cmd->fd = fd;

return (struct cmd\*)cmd;

}

struct cmd\*

pipecmd(struct cmd \*left, struct cmd \*right)

{

struct pipecmd \*cmd;

cmd = malloc(sizeof(\*cmd));

memset(cmd, 0, sizeof(\*cmd));

cmd->type = PIPE;

cmd->left = left;

cmd->right = right;

return (struct cmd\*)cmd;

}

struct cmd\*

listcmd(struct cmd \*left, struct cmd \*right)

{

struct listcmd \*cmd;

cmd = malloc(sizeof(\*cmd));

memset(cmd, 0, sizeof(\*cmd));

cmd->type = LIST;

cmd->left = left;

cmd->right = right;

return (struct cmd\*)cmd;

}

struct cmd\*

backcmd(struct cmd \*subcmd)

{

struct backcmd \*cmd;

cmd = malloc(sizeof(\*cmd));

memset(cmd, 0, sizeof(\*cmd));

cmd->type = BACK;

cmd->cmd = subcmd;

return (struct cmd\*)cmd;

}

//PAGEBREAK!

// Parsing

char whitespace[] = " \t\r\n\v";

char symbols[] = "<|>&;()";

int

gettoken(char \*\*ps, char \*es, char \*\*q, char \*\*eq)

{

char \*s;

int ret;

s = \*ps;

while(s < es && strchr(whitespace, \*s))

s++;

if(q)

\*q = s;

ret = \*s;

switch(\*s){

case 0:

break;

case '|':

case '(':

case ')':

case ';':

case '&':

case '<':

s++;

break;

case '>':

s++;

if(\*s == '>'){

ret = '+';

s++;

}

break;

default:

ret = 'a';

while(s < es && !strchr(whitespace, \*s) && !strchr(symbols, \*s))

s++;

break;

}

if(eq)

\*eq = s;

while(s < es && strchr(whitespace, \*s))

s++;

\*ps = s;

return ret;

}

int peek(char \*\*ps, char \*es, char \*toks) //(vector of strings, string, string)

{

char \*s;

s = \*ps;

while(s < es && strchr(whitespace, \*s))

s++;

\*ps = s;

return \*s && strchr(toks, \*s);

}

struct cmd \*parseline(char\*\*, char\*);

struct cmd \*parsepipe(char\*\*, char\*);

struct cmd \*parseexec(char\*\*, char\*);

struct cmd \*nulterminate(struct cmd\*);

struct cmd\*

parsecmd(char \*s)

{

char \*es;

struct cmd \*cmd;

es = s + strlen(s);

cmd = parseline(&s, es);

peek(&s, es, "");

if(s != es){

printf(2, "leftovers: %s\n", s);

panic("syntax");

}

nulterminate(cmd);

return cmd;

}

struct cmd\*

parseline(char \*\*ps, char \*es)

{

struct cmd \*cmd;

cmd = parsepipe(ps, es);

while(peek(ps, es, "&")){

gettoken(ps, es, 0, 0);

cmd = backcmd(cmd);

}

if(peek(ps, es, ";")){

gettoken(ps, es, 0, 0);

cmd = listcmd(cmd, parseline(ps, es));

}

return cmd;

}

struct cmd\*

parsepipe(char \*\*ps, char \*es)

{

struct cmd \*cmd;

cmd = parseexec(ps, es);

if(peek(ps, es, "|")){

gettoken(ps, es, 0, 0);

cmd = pipecmd(cmd, parsepipe(ps, es));

}

return cmd;

}

struct cmd\*

parseredirs(struct cmd \*cmd, char \*\*ps, char \*es)

{

int tok;

char \*q, \*eq;

while(peek(ps, es, "<>")){

tok = gettoken(ps, es, 0, 0);

if(gettoken(ps, es, &q, &eq) != 'a')

panic("missing file for redirection");

switch(tok){

case '<':

cmd = redircmd(cmd, q, eq, O\_RDONLY, 0);

break;

case '>':

cmd = redircmd(cmd, q, eq, O\_WRONLY|O\_CREATE, 1);

break;

case '+': // >>

cmd = redircmd(cmd, q, eq, O\_WRONLY|O\_CREATE, 1);

break;

}

}

return cmd;

}

struct cmd\*

parseblock(char \*\*ps, char \*es)

{

struct cmd \*cmd;

if(!peek(ps, es, "("))

panic("parseblock");

gettoken(ps, es, 0, 0);

cmd = parseline(ps, es);

if(!peek(ps, es, ")"))

panic("syntax - missing )");

gettoken(ps, es, 0, 0);

cmd = parseredirs(cmd, ps, es);

return cmd;

}

struct cmd\*

parseexec(char \*\*ps, char \*es)

{

char \*q, \*eq;

int tok, argc;

struct execcmd \*cmd;

struct cmd \*ret;

if(peek(ps, es, "("))

return parseblock(ps, es);

ret = execcmd();

cmd = (struct execcmd\*)ret;

argc = 0;

ret = parseredirs(ret, ps, es);

while(!peek(ps, es, "|)&;")){

if((tok=gettoken(ps, es, &q, &eq)) == 0)

break;

if(tok != 'a')

panic("syntax");

cmd->argv[argc] = q;

cmd->eargv[argc] = eq;

argc++;

if(argc >= MAXARGS)

panic("too many args");

ret = parseredirs(ret, ps, es);

}

cmd->argv[argc] = 0;

cmd->eargv[argc] = 0;

return ret;

}

// NUL-terminate all the counted strings.

struct cmd\*

nulterminate(struct cmd \*cmd)

{

int i;

struct backcmd \*bcmd;

struct execcmd \*ecmd;

struct listcmd \*lcmd;

struct pipecmd \*pcmd;

struct redircmd \*rcmd;

if(cmd == 0)

return 0;

switch(cmd->type){

case EXEC:

ecmd = (struct execcmd\*)cmd;

for(i=0; ecmd->argv[i]; i++)

\*ecmd->eargv[i] = 0;

break;

case REDIR:

rcmd = (struct redircmd\*)cmd;

nulterminate(rcmd->cmd);

\*rcmd->efile = 0;

break;

case PIPE:

pcmd = (struct pipecmd\*)cmd;

nulterminate(pcmd->left);

nulterminate(pcmd->right);

break;

case LIST:

lcmd = (struct listcmd\*)cmd;

nulterminate(lcmd->left);

nulterminate(lcmd->right);

break;

case BACK:

bcmd = (struct backcmd\*)cmd;

nulterminate(bcmd->cmd);

break;

}

return cmd;

}