SHELL作业报告

软工2班 3017218092 侯雨茜

一、准备工作

下载6.828 shell并查看。6.828 shell包含两个主要部分：解析shell命令并实现它们。解析器仅识别简单的Shell命令，例如：

```
ls > y
cat < y | sort | uniq | wc > y1
cat y1
rm y1
ls |  sort | uniq | wc
rm y
```

将上面的命令保存在 `t.sh` 文件中，以便之后使用。

使用下面的命令进行编译：

```
gcc sh.c
```

生成一个 `a.out` 文件，可以运行下面命令进行测试，此命令将打印错误消息：

```
./a.out <t.sh
```

因为功能尚未实现，目前执行结果如下：

```
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ gcc sh.c
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ ./a.out <t.sh
redir not implemented
exec not implemented
pipe not implemented
exec not implemented
exec not implemented
pipe not implemented
exec not implemented
```

二、分析sh.c

`main()`函数：

```
int
main(void)
{
  static char buf[100];
  int fd, r;

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    // 如果输入cd filename，就切换目录
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Clumsy but will have to do for now.
      // Chdir has no effect on the parent if run in the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        fprintf(stderr, "cannot cd %s\n", buf+3);
      continue;
    }
    // fork出子进程，来执行输入的命令
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    // wait()函数用于使父进程(也就是调用wait()的进程)阻塞，直到一个子进程结束或该进程接收到一个指定的信号为止。如果该父进程没有子进程或它的子进程已经结束，则wait()就会立即返回。
    wait(&r);
  }
  exit(0);
}
```

当终端有输入后，执行函数`getcmd()`：

```
int
getcmd(char *buf, int nbuf)
{
  // 判断是否为终端输入
  if (isatty(fileno(stdin)))
    fprintf(stdout, "6.828$ ");
  memset(buf, 0, nbuf);
  if(fgets(buf, nbuf, stdin) == 0)
    return -1; // EOF
  return 0;
}
```

之后进入while循环：

1. 判断如果输入为 `cd` 就直接执行目录切换操作
2. 否则，函数 `fork1()` 会 `fork()` 一个子进程，并返回父/子进程的 `pid`
3. `main()` 函数内通过判断返回的 `pid` 来判断当前执行的是哪个进程，从而在子进程中继续执行相应的命令
4. 父进程中使用 `wait(&r)` 进行阻塞，等待子进程返回后再继续执行

parsecmd()函数：

```
struct cmd*
parsecmd(char *s)
{
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);
  cmd = parseline(&s, es);
  peek(&s, es, "");
  if(s != es){
    fprintf(stderr, "leftovers: %s\n", s);
    exit(-1);
  }
  return cmd;
}
```

结构体cmd定义：

```
// All commands have at least a type. Have looked at the type, the code
// typically casts the *cmd to some specific cmd type.
struct cmd {
  int type;          //  ' ' (exec), | (pipe), '<' or '>' for
redirection
};
```

函数parsecmd()用于解析输入的命令，主要是判断输入的命令种类。

runcmd()函数：

```
// Execute cmd.  Never returns.
void
runcmd(struct cmd *cmd)
{
  int p[2], r;
  struct execcmd *ecmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    _exit(0);

  switch(cmd->type){
  default:
    fprintf(stderr, "unknown runcmd\n");
    _exit(-1);

  case ' ':
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      _exit(0);
    fprintf(stderr, "exec not implemented\n");
    // Your code here ...
    break;

  case '>':
  case '<':
    rcmd = (struct redircmd*)cmd;
    fprintf(stderr, "redir not implemented\n");
    // Your code here ...
    runcmd(rcmd->cmd);
    break;

  case '|':
    pcmd = (struct pipecmd*)cmd;
    fprintf(stderr, "pipe not implemented\n");
    // Your code here ...
    break;
  }
  _exit(0);
}
```

`runcmd()` 函数接受一个参数：结构体 `cmd` ，通过这个结构体中的 `type` 值进行进一步的处理。从 `switch case` 语句的判断条件可以看出，将命令的类型分成三类，分别是： `case ''` 可执行命令、`case '<'` `case '>'` 重定向命令和 `case '|'` 管道命令。需求即为补全不同类型命令里具体执行命令的代码。

三、实现Executing simple commands

解析器已经构建了一个 `execcmd`，因此唯一需要编写的代码就是 `runcmd中` 的 `''` 情况。

首先使用 `access()` 函数检查要执行的命令文件是否存在，如果存在就直接执行，否则，在系统的 `/bin/` 目录和 `/usr/bin/` 目录下查找相应的命令，如果有就执行，否则抛出错误。代码如下：

```
case ' ':
  ecmd = (struct execcmd *)cmd;
  if (ecmd->argv[0] == 0)
      _exit(0);
  // fprintf(stderr, "exec not implemented\n");
  // Your code here ...
  if (access(ecmd->argv[0], F_OK) == 0)
  {
      execv(ecmd->argv[0], ecmd->argv);
  }
  else
  {
      const char *bin_path[] = {
          "/bin/",
          "/usr/bin/"};
      char *abs_path;
      int bin_count = sizeof(bin_path) / sizeof(bin_path[0]);
      int found = 0;
      for (int i = 0; i < bin_count && found == 0; i++)
      {
          int pathLen = strlen(bin_path[i]) + strlen(ecmd->argv[0]);
          abs_path = (char *)malloc((pathLen + 1) * sizeof(char));
          strcpy(abs_path, bin_path[i]);
          strcat(abs_path, ecmd->argv[0]);
          if (access(abs_path, F_OK) == 0)
          {
              execv(abs_path, ecmd->argv);
              found = 1;
          }
          free(abs_path);
      }
      if (found == 0)
```

```
        {
            fprintf(stderr, "%s: Command not found\n", ecmd->argv[0]);
        }
    }
    break;
```

编译结果：

```
(base) houyuqiandeMacBook-Pro:~ yuqianhou$ cd /Users/yuqianhou/Documents/Softwar
e_Engineering/操作系统/作业/进程管理/shell
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ gcc sh.c
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ ./a.out
6.828$ ls
Shell作业报告.md        sh.c                    截图
a.out                   t.sh
6.828$
```

四、实现I/O redirection

实现I / O重定向命令，以便您可以运行：

```
echo "6.828 is cool" > x.txt
cat < x.txt
```

解析器已经识别出“>”和“<”，并构建了一个 `redircmd` ，只需在 `runcmd中` 为这些符号填写缺少的代码。

请注意， `redircmd` 中的 `mode` 字段包含访问模式（例如 `O_RDONLY` ），您应该将其传递给 `flags` 参数以 `open` 。

如果您使用的系统调用之一失败，请确保打印错误消息。

结构体 `redircmd` 定义：

```
struct redircmd {
  int type;          // < or >
  struct cmd *cmd;   // the command to be run (e.g., an execcmd)
  char *file;        // the input/output file
  int flags;         // flags for open() indicating read or write
  int fd;            // the file descriptor number to use for the file
};
```

先关闭当前的标准输入 / 输出，打开指定文件作为新的标准输入 / 输出，开始执行命令。代码如下：

```
  case '>':
  case '<':
    rcmd = (struct redircmd*)cmd;
    // fprintf(stderr, "redir not implemented\n");
    // Your code here ...
    close(rcmd->fd);
    if (open(rcmd->file, rcmd->flags, 0644) < 0)
    {
      fprintf(stderr, "Unable to open file: %s\n", rcmd->file);
      exit(0);
    }
    runcmd(rcmd->cmd);
    break;
```

将 `ls` 列出的文件名存入文件 `ls.tmp` 并使用 `cat` 命令读取并显示文件 `ls.tmp` 中的内容。编译结果：

```
(base) houyuqiandeMacBook-Pro:~ yuqianhou$ cd /Users/yuqianhou/Documents/Softwar
e_Engineering/操作系统/作业/进程管理/shell
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ gcc sh.c
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ ./a.out
6.828$ ls > ls.tmp
6.828$ cat < ls.tmp
Shell作业报告.md
a.out
ls.tmp
sh.c
t.sh
截图
6.828$
```

五、实现pipes

实现pipes，以便可以运行命令管道，例如：

```
$ ls | 排序| uniq | 厕所
```

解析器已经识别出“|”，并构建了一个 `pipecmd`，因此您必须编写的唯一代码就是在runcmd中的 `case '|'`。

结构体 `pipecmd` 的定义：

```
struct pipecmd {
  int type;          // |
  struct cmd *left;  // left side of pipe
  struct cmd *right; // right side of pipe
};
```

管道命令的标志是符号 `|` ， `|` 的左面和右面分别是不同的命令，我们需要逐步的执行这些命令。代码如下：

```
  case '|':
    pcmd = (struct pipecmd*)cmd;
    // fprintf(stderr, "pipe not implemented\n");
    // Your code here ...
    // 建立pipe
    if (pipe(p) < 0)
    {
      fprintf(stderr, "pipe failed\n");
    }
    if (fork1() == 0)
    {
      // 关闭标准输出
      close(1);
      // dup会把标准输出定向到 p[1] 所指文件，即管道写入端
      dup(p[1]);
      // 去掉管道对端口的引用
      close(p[0]);
      close(p[1]);
      // left的标准输入不变，标准输出流入管道
      runcmd(pcmd->left);
    }
    // fork一个子进程处理右边的命令，讲标准输入定向到管道的输出，即读取了来自左边命令返回的结果
    if (fork1() == 0)
    {
      // 关闭标准输出
      close(0);
      // dup会把标准输出定向到 p[1] 所指文件，即管道写入端
      dup(p[0]);
      // 去掉管道对端口的引用
      close(p[0]);
      close(p[1]);
      // left的标准输入不变，标准输出流入管道
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait(&r);
```

```
38      wait(&r);
39      break;
```

编译结果：

```
6.828$ (base) houyuqiandeMacBook-Pro:shell yuqianhou$ gcc sh.c
(base) houyuqiandeMacBook-Pro:shell yuqianhou$ ./a.out
6.828$ (base) houyuqiandeMacBook-Pro:shell yuqianhou$ ./a.out < t.sh
       7       7       53
       7       7       53
(base) houyuqiandeMacBook-Pro:shell yuqianhou$
```

至此，题目中的基本功能已成功实现。