Unix程序设计

实验（四） Shell程序设计和改进

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年11月4日
学　　院	信息学院
课程名称	Unix程序设计

实验（四） Shell程序设计和改进

目录

1 实验内容

简单Shell的设计和实现。要求：

1. 除了系统调用execve，不允许使用其他的exec函数。输入应当允许带多个参数(一行内可以表示)，不考虑通配符 (即“*”、“?”、“-”等等)的处理。
2. 输入错误命令能提示出错并进入下一轮接收命令状态。
3. 可以用Ctrl-C和Ctrl-\ 结束简单Shell的运行。
4. 程序运行正确，提示简洁明确。

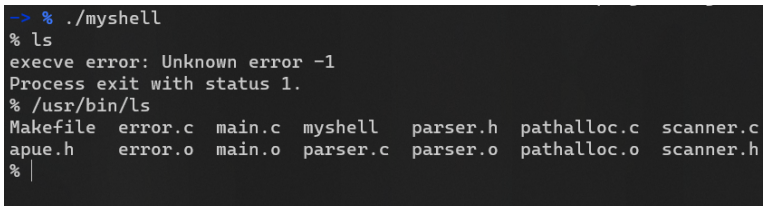
2 最简单的Shell实现

实现一个最简单的Shell只需要三步：首先从标准输入流中读入一整行数据并利用空格拆分，然后`fork`当前进程，最后用`execve`执行命令。最后使用`waitpid`等待子进程结束，然后再次读入命令。

2.1 运行效果

按这种思路实现的Shell运行结果如图 1，它可以通过用完整的绝对路径运行简单的Shell命令。 并且能够

图 1: 最简单的Shell



正确地传递命令行参数。

2.2 有什么问题

今天，当用户使用Shell的时候所期待的重要的功能并不是简单地运行相应的程序。而是看重Shell可以通过许多便利的高级功能实现对简单程序的组合，来方便一次完成批量、重复的操作。这是当今CLI相对GUI而言最主要的优点。显然，这些功能并不是简单地用空格拆分输入的命令就可以实现的。因此需要更进一步地设计处理输入的命令的流程。

例如合法的Shell表达式(1)，

$$grep\ ab\ <\ test.txt\ |\ grep\ abc \tag{1}$$

这条表达式先在文件test.txt中查找包含ab的行，然后再在这些行中查找包含abc的行。这个表达式中使用了管道符“|”和I/O重定向符。为了正确处理这条Shell表达式，直接用空格拆分输入的命令是不够的。因为管道符和I/O重定向符的存在，对一条命令的处理不只是简单的从左向右依次扫描，而需要对输入的命令进行更进一步分析，首先执行管道符左边的命令，执行时进行I/O重定向，然后将标准输出的结果为输入传递给管道符右边的命令。再比如，Shell表达式(2)，

$$wget\ <some\ url>\ |\ grep\ error > log.txt\ \& \tag{2}$$

这条表达式中使用了后台运行符“&”，这个符号的作用是要求Shell启动程序后立刻返回为下一条指令就绪，而将命令放到后台运行，而不是等待命令执行完毕后再执行下一条命令。这个符号的效果对整条表达式起效，而非最后一部分的`grep`。这样一来，空格拆分、从左往右的处理方式就不再适用，否则会给出错误的结果。

因此，当管道、*I/O*重定向、后台执行等功能嵌套混合使用时，*Shell*需要具有更强的对输入命令的分析能力，来处理不同表达式中存在的结合性和优先级。

3 改进的Shell设计

3.1 设计目标

为了改进Shell的设计，需要先定下合理的目标，因为现代Shell是一个极其复杂的系统，因此只能实现其中功能的一个子集。在这些功能中，最基础的是

- 支持管道的处理
- 支持I/O重定向的处理
- 支持用括号等符号改变运行顺序
- 支持Shell脚本语句结束的分号;的处理
- 支持后台运行符号&的处理

有了对这些功能的正确处理，就能实现绝大多数Shell所具有的、重要的基础功能。因此，这些功能是改进Shell设计的目标。而为了这些功能的实现，需要对输入的命令进行更进一步的分析，这个分析的过程需要进行词法分析、语法分析，然后形成一棵语法树，最后再对这棵语法树进行遍历，并运行相应的操作。为了符合Shell的一般规律，还需要实现在$PATH$环境变量中查找命令的功能。

4 词法分析

4.1 词法分析的基本思想

Shell的用户输入是一个字符串。为了有效地处理，我们首先需要将其拆分成最小有意义的字符串单元，这些字符单元具有被赋予、能被识别的意义。这称之为一个Token，这个过程称之为词法分析 [1]。一个Token由名称和可选的Token值构成。在常见的编程语言中，Token包括标识符、运算符、关键字、字面量、分隔符等。在我们需要支持的Shell命令词法分析中，我们只需要支持标识符、分隔符（;、(、)）和运算符（|、>、<）。

例如表达式(1)经过词法分析，就会变成Token序列(3)。

$$[(identifier, grep), (identifier, ab), (operator, <), (identifier, test.txt), \\ (operator, |), (identifier, grep), (identifier, abc)] \quad (3)$$

为了消除歧义，这里采用最长匹配的消歧义原则。即，当遇到一个Token的前缀是另一个Token的前缀时，选择最长的那个Token。

4.2 词法分析的实现

词法分析的实现代码见附录：部分代码清单、文件结构和说明中的代码 4。其中第一个参数是所需要词法分析的字符串的开头指针，也用于返回获取一个Token以后的位置；第二个参数是其结尾指针；后两个参数用于返回Token的开头和结尾位置。返回值代表Token类型，其中返回值若是字符'a'就代表终结符IDENTIFIER。该函数的执行结果就是从命令中取出一个Token。

5 语法分析

得到了Token序列，需要正确地组织这些Token来形成符合Shell命令语义的抽象语法树（AST）。这个过程需要语法分析 [1]。要进行语法分析，首先需要形式地定义Shell命令的语法规则，这种定义需要上下文无关语法。

5.1 上下文无关语法

上下文无关文法（CFG）是一种形式化的语法定义方法，它定义了一种语言的语法规则。上下文无关文法由四元组(4)定义。

$$G = (\mathcal{V}, \Sigma, \mathcal{R}, \mathcal{S}) \tag{4}$$

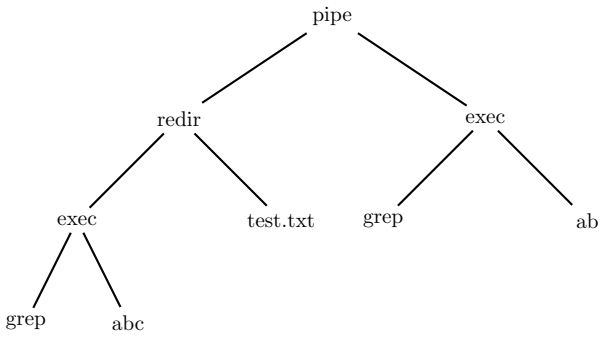
其中，$\mathcal{V}$是非终结符的集合，Σ是终结符的集合，$\mathcal{R}$是从$\mathcal{V}$到$(\mathcal{V} \cup \Sigma)^*$的关系，使得$\exists \omega \in (\mathcal{V} \cup \Sigma)^* : (S, \omega) \in \mathcal{R}$,$\mathcal{S}$是开始符号。

上下文无关文法具有足够强的表达力来表达大多数编程语言的语法规则。而巴克斯-诺尔范式（BNF)是上下文无关文法的一种常用表示。在Shell命令的语法分析中，我们需要通过CFG定义的是Shell命令的语法规则。根据我们需要支持的内容，我们需要支持括号表达式、重定向、管道等几种二元表达式、后台运行&和行结束;等几种一元表达式，因此可以定义(5)的CFG。

$$\begin{aligned}
command &\rightarrow (line)^* \\
line &\rightarrow pipe \,|\, back \,|\, list \\
back &\rightarrow line\text{'\&'} \\
list &\rightarrow list\text{';'} \\
pipe &\rightarrow exec \,|\, pipe \\
exec &\rightarrow paren \,|\, redirect \,|\, IDENTIFIER(\text{'-'}\, IDENTIFIER)^* \\
paren &\rightarrow \text{'('}\, redirect\, \text{')'} \\
redirect &\rightarrow exec(\text{'}>\text{'} \,|\, \text{'}>>\text{'})IDENTIFIER \,|\, exec\text{'}<\text{'}\, IDENTIFIER
\end{aligned} \tag{5}$$

根据(5)的定义对表达式(1)进行语法分析，得到的AST如图 2所示。

图 2: 命令(1)的AST



只要生成该AST，就可以对输入的命令进行解释执行了。因为Shell命令中我们支持的子集不需要语义分析。为了生成正确的AST，需要使用递归下降法实现正确的Parser进行语法分析。

5.2 递归下降法

递归下降法是一种自上而下的语法分析方法，它的基本思想是：从文法的开始符号开始，逐步推导出输入串的语法结构。递归下降法的实现需要使用递归函数，每个递归函数对应文法中的一个非终结符，每个递

归函数的功能是：从输入串中读取一个符号，判断该符号是否符合文法中对应的语法规则，如果符合，则继续读取下一个符号，直到读取完整个输入串。如果输入串中的符号不符合文法中对应的语法规则，则报错。

本Shell的实现采用了语法制导的翻译（Syntax-directed Translation）。即整个语句的编译过程完全由Parser部分驱动。在运行语法分析的过程中同时进行词法分析。而不是对不同阶段进行明确的区分。这样做的原因是所支持的Shell命令的语法规则比较简单，不需要进行模块间非常明确的隔离就可以完成。同时，同时进行语法分析和语义分析有助于减少内存足迹（Memory Footprint），提高Shell运行的效率。

在这一过程中，语法错误错误的命令输入会被发现，然后输出错误信息。而路径错误等语义上的错误会在运行时被发现。

本Shell的语法分析实现位于`parser.c`中，其中`parse_line`函数对应文法中的`line`非终结符，以此类推。具体见代码实现，或者附录：部分代码清单、文件结构和说明中的代码 5。

6 解释执行

得到了抽象语法树，就可以对输入的命令进行解释执行了。这个过程需要对抽象语法树进行遍历，然后对遍历到的节点进行相应的操作。这是对树状结构的深度优先遍历。具体的过程在`main.c`的`run_command`中，它判断`command_t`中的类型字段，然后转换成正确的结构体类型，最后对相应的表达式进行执行，执行时会对自身进行递归调用。

在这一阶段会通过`fork`创建子进程，在子进程中通过`execve`对相应的可执行文件进行执行。由于处理`fork`返回值的过程固定而重复，并且在Shell中的处理就是输出报错信息并停止执行。所以实现了`fork_panic`函数来简化处理，其代码如代码 1。

代码 1: fork_panic

```
1 int fork_panic()
2 {
3         int pid = fork();
4         if (pid < 0)
5         {
6                 err_exit(pid, "fork");
7         }
8         return pid;
9 }
```

6.1 执行I/O重定向

为了执行I/O重定向，就需要将标准输入输出的文件描述符重新打开到相应的文件中，如代码 2。

代码 2: I/O重定向

```
1 rcmd = (redir_command_t *)cmd;
2 fd = open(rcmd->file, rcmd->mode | O_SYNC | O_TRUNC);
3 if (fd < 0)
4 {
5         err_exit(fd, "open %s failed\n", rcmd->file);
6 }
7 dup2(fd, rcmd->fd);
8 close(fd);
9 run_command(rcmd->cmd);
```

6.2 执行管道

管道是一种重要的Shell特性，它可以将一个命令的输出作为另一个命令的输入。一个进程在管道中写入的内容会被另一个进程读取，实现了一种进程间通信机制。管道的实现需要使用pipe函数创建管道，然后使用dup2函数将管道的读写端重定向到两个进程的标准输入输出上。因此我们创建两个子进程，进行相应的重定向，然后在两个子进程中分别递归调用来执行 pipe AST节点的左右两个操作数。其实现片段如代码 3。在主进程中，我们也需要等待两个子进程的结束。

代码 3: fork_panic

```
pcmd = (pipe_command_t *)cmd;
if (pipe(p) < 0)
{
        err_quit("pipe");
}
if ((pid1 = fork_panic()) == 0)
{
        dup2(p[1], STDOUT_FILENO);
        close(p[0]);
        close(p[1]);
        run_command(pcmd->left);
}
if ((pid2 = fork_panic()) == 0)
{
        dup2(p[0], STDIN_FILENO);
        close(p[0]);
        close(p[1]);
        run_command(pcmd->right);
}
close(p[0]);
close(p[1]);
waitpid(pid1, NULL, 0);
waitpid(pid2, NULL, 0);
```

6.3 后台执行

支持后台执行符号&，即在命令后面加上&，可以使命令在后台执行，而不会阻塞Shell的执行。这一功能的实现相当简单。在执行之后不经过wait即返回，这样就不会阻塞Shell的执行。

其它解释执行阶段的具体实现见代码，或者附录：部分代码清单、文件结构和说明中的代码 6。

7 从环境变量$PATH$中查找程序

在Shell中，我们可以直接输入程序名，而不需要输入程序的绝对路径，Shell会自动从环境变量$PATH$中查找程序。这是Shell最重要的功能之一。为了实现这一功能，我们再进行execve前也必须进行查找。

本实现中查找$PATH$环境变量内容的函数为find_in_path，其代码位于findcmd.c中，内容如 附录：部分代码清单、文件结构和说明中的代码 7。它的实现原理非常简单。首先，Linux下的$PATH$环境变量都是以冒号分隔的，所以我们可以通过strtok函数将其分割成多个字符串，然后利用上次实验中使用到的$path_alloc$函数分配路径的内存空间，最后将分割出来的字符串拼接到路径后面，然后用$fstat$判断该路径下是否存在该文件，如果存在则返回该路径。这样查找到的就是首个匹配的文件。

8 遇到的问题和解决方法

在实现以上功能时，所遇到的问题就是I/O重定向不正常工作。

8.1 I/O重定向不正常工作

这个问题的来源是因为误以为open打开的文件描述符是会选择最小的可用的文件描述符，但是实际上这种假设在一些内核版本中成立，而在一些内核中并不是这样的。因此，不能依赖先close相应的标准输入输出文件流，然后再open文件，这样会导致文件描述符错误，使得I/O重定向失败。

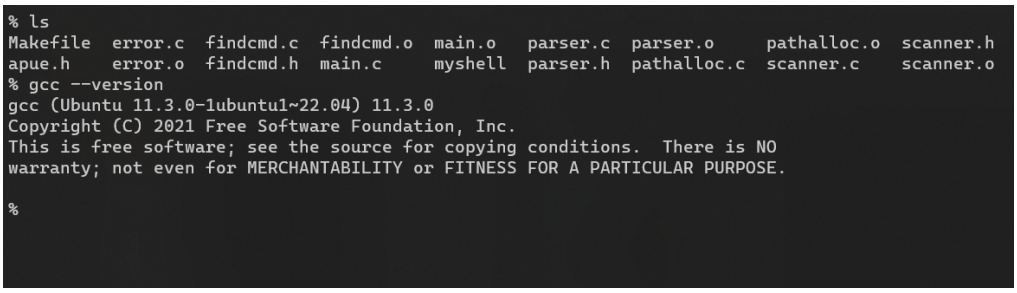
正确的方式是调用dup2函数，将文件描述符复制到标准输入输出文件流的文件描述符，然后关闭open返回的相应描述符。这样就可以保证文件描述符正确。

9 运行效果

9.1 简单的命令

运行简单的命令，如纯粹地运行某一程序、或者加入简单的参数的效果如图3所示。可见Shell能从$PATH$中

图 3: 简单的命令

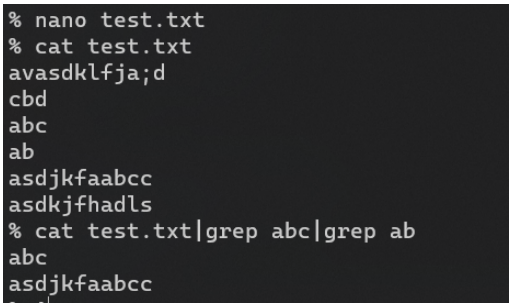


查找到相应的程序，然后使得程序能够正常运行。命令行参数，例如--version能够被正确解析然后传入gcc程序中。

9.2 管道

使用管道连接程序，从文本文件test.txt中读取内容，然后将先查找包含 abc的结果，再查找包含ab的结果，最后将结果输出到标准输出流中，显示在终端上。这一功能可以使用管道符连接一个cat命令和两个grep命令完成。实际的执行结果如图 4 可见，Shell能够正确地将管道符连接的命令的输入输出流连接起来，然后最

图 4: Pipe

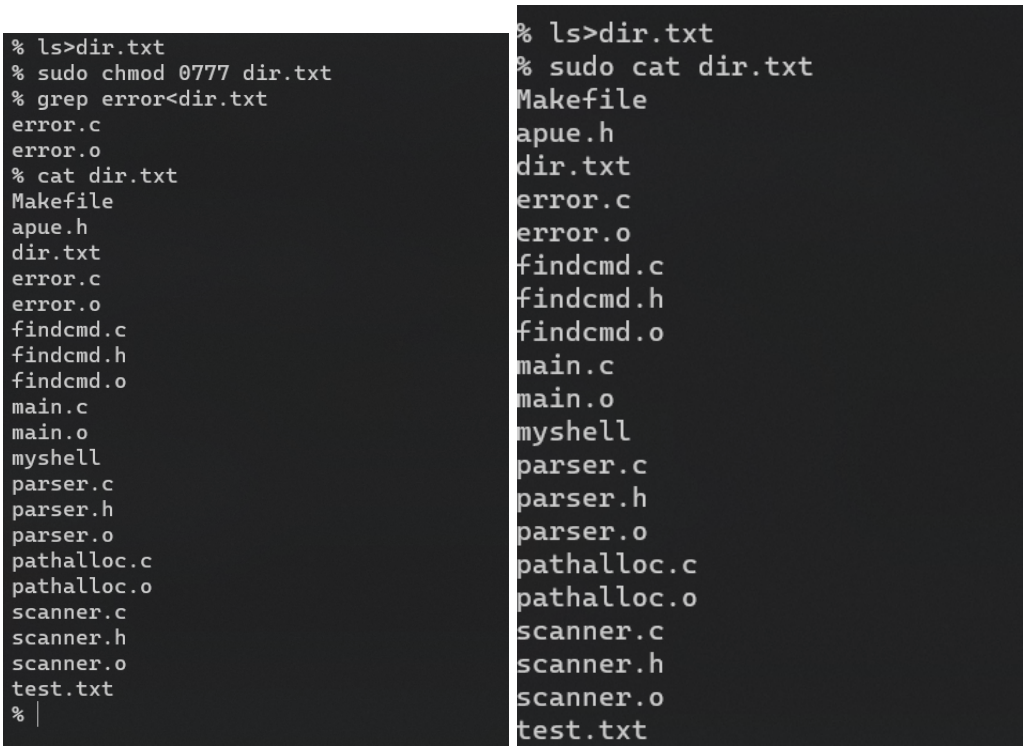


右侧的grep输出了正确的结果。

9.3 I/O重定向

为了测试I/O重定向的功能，就需要测试对标准输入的重定向和对标准输出的重定向两个部分。我们首先将ls的结果重定向到文件dir.txt中，然后再将dir.txt的内容定向到grep的输入中，在其中查找串error。这样就可以测试I/O重定向的功能。实际的执行结果如图 ??。 可见，Shell能够正确地将标准输入重定向到文

图 5: I/O重定向结果



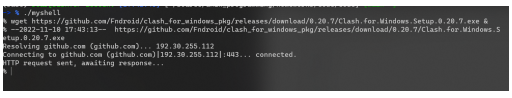
(a) 重定向标准输入 (b) 重定向标准输出

件中，然后将文件的内容作为标准输入传入grep程序中。这表明I/O重定向的两个方向都被Shell正确地实现了。

9.4 后台执行

为了测试后台运行，我们使用wget命令对文件进行下载来模拟长时操作。然后使用&符号将其放到后台执行。运行结果如图 6所示。 可以发现，执行wget之后，Shell会立即返回，而不会等待wget的执行结果。这

图 6: 后台执行



是出现了Shell提示符”%”。这表示后台执行的功能得到了正确的实现。

10 软件工程上的处理

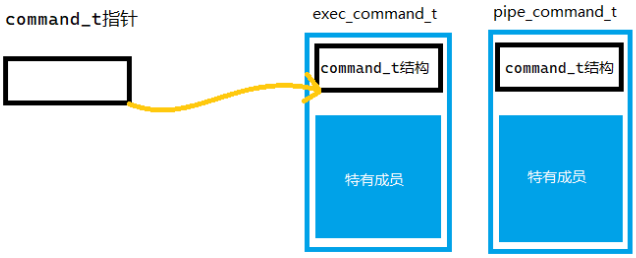
为了对命令进行更好的解析处理，该Shell实现的代码规模相对较大。为了有效地进行管理，保证可扩展性和可维护性，采用了一些措施。

10.1 利用内存布局实现伪多态

为了适应抽象语法树的节点有多个不同类型的特点，在设计抽象语法树的节点的数据结构时，可以通过指针类型转换的规则模拟“类型擦除”来实现一定的多态能力。这一点在主要使用C语言的大型项目，例如Linux内核中得到了大量的应用。

这种方式主要的原理如图 7所示。 每一个AST节点中都具有一个command_t类型的“头”结构，当我们传

图 7: AST节点结构



递一个AST节点的指针时，直接传递 command_t类型的指针即可。在需要使用具体的AST节点类型时，可以通过其中的字段进行判断，然后利用指针类型转换的规则 [2]将command_t转换成对应的具体类型。

这种方法在逻辑上实现了一种“is-a”的继承关系。这解决了缺乏面向对象程序设计能力的C语言当代码库的规模扩大时面对大量类型不容易管理的问题。

10.2 恰当分割文件

为了保证代码的可读性，每个文件不宜太长 [3]。因此在这个Shell实现中，将代码分割成了多个文件，每个文件大致负责一个功能的实现，较好地控制了文件的长度。

为了有效地组织编译，使用了Makefile对编译进行处理。

11 实验总结

这次实验中，我实现了可以处理复杂Shell命令语法的Shell。进一步了解了Shell这一类UNIX操作系统中不可或缺、极其重要的组件是如何实现的。在实现过程中，我使用语法分析-语义分析这一依赖于编译原理的方法对Shell命令进行解析，然后在抽象语法树中进行遍历和执行的方法，实现了除简单运行程序之外的重要的 Shell基础设施。对Shell的实现有了更加深入的理解。

参考文献

References

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[2] WG14 ISO and Y N1124. Iso/iec 9899: 1999. *IProgramming languages±C", ISO*, 1999.

[3] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

附录：部分代码清单、文件结构和说明

.1 文件结构和说明

为了保证每个文件不太长、便于管理和调试，将代码分割成了多个文件。每个文件大致负责一个功能的实现。文件和对应实现的功能如表 1

表格 1: 文件结构和说明

文件名	功能
apue.h, error.c, pathalloc.c	书附带用于处理错误、分配路径内存的函数。
main.c	主函数，调用语法分析、词法分析并解释执行。
findcmd.c	在$PATH$中查找命令的路径。
parser.c, parser.h	语法分析、抽象语法树的构造。
scanner.c, scanner.h	词法分析。

.2 部分代码清单

代码 4: 词法分析

```
1  static const char whitespace[] = " \t\r\n\v";
2  static const char symbols[] = "<|>&;()";
3
4  int get_token(char **ps, char *es, char **q, char **eq)
5  {
6          char *s = *ps;
7
8          while (s < es && strchr(whitespace, *s))
9                  s++;
10
11
12         if (q)
13         {
14                 *q = s;
15         }
16
17         int ret = *s;
18         switch (*s)
19         {
20         case 0:
21                 break;
22         case '|':
23         case '(':
24         case ')':
25         case ';':
26         case '&':
27         case '<':
28                 s++;
29                 break;
30         case '>':
31                 s++;
32                 if (*s == '>')
33                 {
34                         ret = '+';
```

```
35                          s++;
36                  }
37                  break;
38          default:
39                  ret = 'a';
40                  while (s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
41                          s++;
42                  break;
43          }
44          if (eq)
45          {
46                  *eq = s;
47          }
48
49          while (s < es && strchr(whitespace, *s))
50                  s++;
51
52          *ps = s;
53          return ret;
54  }
55
56  int peek(char **ps, char *es, char *toks)
57  {
58          char *s;
59
60          s = *ps;
61          while (s < es && strchr(whitespace, *s))
62                  s++;
63          *ps = s;
64          return *s && strchr(toks, *s);
65  }
```

代码 5: 语法分析

```
1   //
2   // Created by bear on 11/4/2022.
3   //
4   #include "apue.h"
5
6   #include <stdlib.h>
7   #include <fcntl.h>
8
9   #include "parser.h"
10  #include "scanner.h"
11
12  static inline exec_command_t *make_exec_command(void)
13  {
14          exec_command_t *cmd = malloc(sizeof(exec_command_t));
15          cmd->base.type = CMD_EXEC;
16          return cmd;
17  }
18
19  static inline redir_command_t *make_redir_command(command_t *subcmd, char *file, char *efile, int
        mode, int fd)
20  {
```

```
21          redir_command_t *cmd = malloc(sizeof(redir_command_t));
22          cmd->base.type = CMD_REDIR;
23          cmd->cmd = subcmd;
24          cmd->file = file;
25          cmd->efile = efile;
26          cmd->mode = mode;
27          cmd->fd = fd;
28          return cmd;
29  }
30
31  static inline pipe_command_t *make_pipe_command(command_t *left, command_t *right)
32  {
33          pipe_command_t *cmd = malloc(sizeof(pipe_command_t));
34          cmd->base.type = CMD_PIPE;
35          cmd->left = left;
36          cmd->right = right;
37          return cmd;
38  }
39
40  static inline list_command_t *make_list_command(command_t *left, command_t *right)
41  {
42          list_command_t *cmd = malloc(sizeof(list_command_t));
43          cmd->base.type = CMD_LIST;
44          cmd->left = left;
45          cmd->right = right;
46          return cmd;
47  }
48
49  static inline back_command_t *make_back_command(command_t *subcmd)
50  {
51          back_command_t *cmd = malloc(sizeof(back_command_t));
52          cmd->base.type = CMD_BACK;
53          cmd->cmd = subcmd;
54          return cmd;
55  }
56
57  command_t *parse_line(char **ps, char *es);
58  command_t *parse_pipe(char **ps, char *es);
59  command_t *parse_exec(char **ps, char *es);
60  command_t *parse_redirs(command_t *cmd, char **ps, char *es);
61  command_t *ensure_nulterm(command_t *cmd);
62
63  command_t *
64  parse_line(char **ps, char *es)
65  {
66          command_t *cmd;
67
68          cmd = parse_pipe(ps, es);
69          while (peek(ps, es, "&"))
70          {
71                  get_token(ps, es, 0, 0);
72                  cmd = (command_t *)make_back_command(cmd);
73          }
74          if (peek(ps, es, ";"))
75          {
```

```
76                  get_token(ps, es, 0, 0);
77                  cmd = (command_t *)make_list_command(cmd, parse_line(ps, es));
78          }
79          return cmd;
80  }
81
82  command_t *
83  parse_pipe(char **ps, char *es)
84  {
85          command_t *cmd;
86
87          cmd = parse_exec(ps, es);
88          if (peek(ps, es, "|"))
89          {
90                  get_token(ps, es, 0, 0);
91                  cmd = (command_t *)make_pipe_command(cmd, parse_pipe(ps, es));
92          }
93          return cmd;
94  }
95
96  command_t *
97  parse_redirs(command_t *cmd, char **ps, char *es)
98  {
99          int tok;
100         char *q, *eq;
101
102         while (peek(ps, es, "<>"))
103         {
104                 tok = get_token(ps, es, 0, 0);
105                 if (get_token(ps, es, &q, &eq) != 'a')
106                 {
107                         err_quit("missing file for redirection");
108                 }
109                 switch (tok)
110                 {
111                 case '<':
112                         cmd = (command_t *)make_redir_command(cmd, q, eq, O_RDONLY, STDIN_FILENO);
113                         break;
114                 case '>':
115                         cmd = (command_t *)make_redir_command(cmd, q, eq, O_WRONLY | O_CREAT,
                                STDOUT_FILENO);
116                         break;
117                 case '+':  // >>
118                         cmd = (command_t *)make_redir_command(cmd, q, eq, O_WRONLY | O_CREAT,
                                STDOUT_FILENO);
119                         break;
120                 }
121         }
122         return cmd;
123 }
124
125 command_t *
126 parse_paren(char **ps, char *es)
127 {
128         command_t *cmd;
```

```
129
130         if (!peek(ps, es, "("))
131         {
132                 err_quit("parse_paren");
133         }
134         get_token(ps, es, 0, 0);
135         cmd = parse_line(ps, es);
136         if (!peek(ps, es, ")"))
137         {
138                 err_quit("syntax - missing )");
139         }
140         get_token(ps, es, 0, 0);
141         cmd = parse_redirs(cmd, ps, es);
142         return cmd;
143 }
144
145 command_t *
146 parse_exec(char **ps, char *es)
147 {
148         char *q, *eq;
149         int tok, argc;
150         exec_command_t *cmd;
151         command_t *ret;
152
153         if (peek(ps, es, "("))
154         {
155                 return parse_paren(ps, es);
156         }
157
158         ret = (command_t *)make_exec_command();
159         cmd = (exec_command_t *)ret;
160
161         argc = 0;
162         ret = parse_redirs(ret, ps, es);
163         while (!peek(ps, es, "|)&;"))
164         {
165                 if ((tok = get_token(ps, es, &q, &eq)) == 0)
166                 {
167                         break;
168                 }
169                 if (tok != 'a')
170                 {
171                         err_quit("syntax");
172                 }
173                 cmd->argv[argc] = q;
174                 cmd->eargv[argc] = eq;
175                 argc++;
176                 if (argc >= MAXARGS)
177                 {
178                         err_quit("too many args");
179                 }
180                 ret = parse_redirs(ret, ps, es);
181         }
182         cmd->argv[argc] = 0;
183         cmd->eargv[argc] = 0;
```

```
184         return ret;
185 }
186 
187 // NUL-terminate all the counted strings.
188 command_t *
189 ensure_nulterm(command_t *cmd)
190 {
191         int i;
192         back_command_t *bcmd;
193         exec_command_t *ecmd;
194         list_command_t *lcmd;
195         pipe_command_t *pcmd;
196         redir_command_t *rcmd;
197 
198         if (cmd == 0)
199         {
200                 return 0;
201         }
202 
203         switch (cmd->type)
204         {
205         case CMD_EXEC:
206                 ecmd = (exec_command_t *)cmd;
207                 for (i = 0; ecmd->argv[i]; i++)
208                         *ecmd->eargv[i] = 0;
209                 break;
210 
211         case CMD_REDIR:
212                 rcmd = (redir_command_t *)cmd;
213                 ensure_nulterm(rcmd->cmd);
214                 *rcmd->efile = 0;
215                 break;
216 
217         case CMD_PIPE:
218                 pcmd = (pipe_command_t *)cmd;
219                 ensure_nulterm(pcmd->left);
220                 ensure_nulterm(pcmd->right);
221                 break;
222 
223         case CMD_LIST:
224                 lcmd = (list_command_t *)cmd;
225                 ensure_nulterm(lcmd->left);
226                 ensure_nulterm(lcmd->right);
227                 break;
228 
229         case CMD_BACK:
230                 bcmd = (back_command_t *)cmd;
231                 ensure_nulterm(bcmd->cmd);
232                 break;
233         }
234         return cmd;
235 }
236 
237 command_t *parse_command(char *cmdline)
238 {
```

```
239         char *cbeg = cmdline, *cend = cmdline;
240         while (*cend)
241                 cend++;
242
243         command_t *cmd = parse_line(&cbeg, cend);
244         peek(&cbeg, cend, "");
245         if (cbeg != cend)
246         {
247                 err_quit("syntax error with leftovers: %s\n", cbeg);
248         }
249         ensure_nulterm(cmd);
250         return cmd;
251 }
```

代码 6: 解释执行

```
1  void run_command(command_t *cmd)
2  {
3          int p[2];
4          int err;
5          int fd;
6          int pid, pid1, pid2;
7
8          back_command_t *bcmd;
9          exec_command_t *ecmd;
10         list_command_t *lcmd;
11         pipe_command_t *pcmd;
12         redir_command_t *rcmd;
13
14         if (cmd == 0)
15         {
16                 exit(0);
17         }
18
19         switch (cmd->type)
20         {
21         default:
22                 err_quit("run_command");
23
24         case CMD_EXEC:
25                 ecmd = (exec_command_t *)cmd;
26                 if (ecmd->argv[0] == 0)
27                 {
28                         exit(0);
29                 }
30                 char *name = find_in_path(ecmd->argv[0], getenv("PATH"));
31                 err = execve(name, ecmd->argv, environ);
32                 err_exit(err, "exec %s failed\n", ecmd->argv[0]);
33                 break;
34
35         case CMD_REDIR:
36                 rcmd = (redir_command_t *)cmd;
37                 fd = open(rcmd->file, rcmd->mode | O_SYNC );
38                 if (fd < 0)
39                 {
```

```
40                      err_exit(fd, "open %s failed\n", rcmd->file);
41              }
42              dup2(fd, rcmd->fd);
43              close(fd);
44              run_command(rcmd->cmd);
45              break;
46
47      case CMD_LIST:
48              lcmd = (list_command_t *)cmd;
49              pid = fork_panic();
50              if (pid == 0)
51              {
52                      run_command(lcmd->left);
53              }
54              waitpid(pid, NULL, 0);
55              run_command(lcmd->right);
56              break;
57
58      case CMD_PIPE:
59              pcmd = (pipe_command_t *)cmd;
60              if (pipe(p) < 0)
61              {
62                      err_quit("pipe");
63              }
64              if ((pid1 = fork_panic()) == 0)
65              {
66                      dup2(p[1], STDOUT_FILENO);
67                      close(p[0]);
68                      close(p[1]);
69                      run_command(pcmd->left);
70              }
71              if ((pid2 = fork_panic()) == 0)
72              {
73                      dup2(p[0], STDIN_FILENO);
74                      close(p[0]);
75                      close(p[1]);
76                      run_command(pcmd->right);
77              }
78              close(p[0]);
79              close(p[1]);
80              waitpid(pid1, NULL, 0);
81              waitpid(pid2, NULL, 0);
82              break;
83
84      case CMD_BACK:
85              bcmd = (back_command_t *)cmd;
86              if (fork_panic() == 0)
87              {
88                      run_command(bcmd->cmd);
89              }
90              break;
91      }
92      exit(0);
93  }
```

代码 7: find_in_path

```
char *find_in_path(char *exe, char *pathlist)
{
        char *p, *q, *path;
        struct stat st;

        if (strchr(exe, '/'))
        {
                if (stat(exe, &st) < 0)
                {
                        return 0;
                }
                return exe;
        }

        for (p = pathlist; *p; p = q + 1)
        {
                if ((q = strchr(p, ':')) == 0)
                {
                        q = p + strlen(p);
                }
                if (q == p)
                {
                        path = ".";
                }
                else
                {
                        path = p;
                }
                size_t buf_len = 0;
                char *buf = path_alloc(&buf_len);
                memset(buf, 0, buf_len);

                if ((q - p) + 1 + strlen(exe) + 1 > buf_len)
                {
                        free(buf);
                        continue;
                }
                snprintf(buf, buf_len, "%.*s/%s", (int)(q - p), path, exe);

                int err = stat(buf, &st);
                if (err == 0)
                {
                        return buf;
                }
        }
        return 0;
}
```