後期 SAIL レポート
-簡易シェルの実装-

24266098
工学部知能情報システム工学科
見崎成

2025 年 2 月 11 日

1　目的

本レポートの目的は、シェルの内部の動作原理を理解することである。シェルとは、ユーザが入力したコマンドを解釈し、コマンドに応じて必要な処理を行うプログラムである。このシェルの機能を理解するために、代表的なシェルの機能であるリダイレクトやパイプを実装することで、シェルの動作原理に関して理解を深めることを目的とする。また、他人が作成した洗練されたコードを読むことで、プログラムの作成方法やコードの見やすさについて学ぶことも目的とする。

2　設計

今回のシェルはパイプ、リダイレクトの実装を主としている。そのため、プログラムの流れは以下のように設計した。

(1) ユーザが入力したコマンドを分割する
(2) 入力されたコマンドをすべて見る
(3) コマンドのみの時は、通常のコマンドを実行する
(4) リダイレクトがあるときは、リダイレクトを実行する
(5) パイプとリダイレクト両方があるときは、両方に対応するように処理をする。

3　実装

実装したプログラムは長いためすべては付録に記載する。ここでは、cd コマンドの処理とパイプとリダイレクトの処理を行っている関数について説明する。

3.1　cd コマンドの処理

次に示すコードが cd コマンドの処理を行っている関数である。

ソースコード 1　cd コマンドの処理

```
1  void cd(char **argv){
2    char path[BUFSIZE];
3    getcwd(path,BUFSIZE);
4    if(strstr(argv[1],"/")==NULL){
5      strcat(path,"/");
6      strcat(path,argv[1]);
7    }
8    if(chdir(path)==-1){
9      printf("no file\n");
10     exit(1);
11   }
12 }
```

引数として、標準入力で受け取った文字列を渡す。cd の後ろは相対パスが入力されると考え絶対パス表示にするため,getcwd を利用している。ユーザの入力の先頭に/がないと絶対パスの指定がうまくいかないため、/がない場合は/を追加して絶対パスを指定できるようにしている。

3.2 パイプリダイレクト処理

パイプリダイレクト処理は関数が長いため、付録で記載する。

(1) 入力された文字列をすべて確認し、リダイレクトパイプの場所を記録する
(2) 通常のコマンドの場合はそのままコマンドを実行する
(3) 逆のリダイレクトがあるときは、逆のリダイレクトを実行する
(4) リダイレクトがあるときは、リダイレクトを実行する
(5) パイプとリダイレクト両方があるときは、両方に対応するように処理をする。そうでない場合はパイプのみを実行する。

4 リファクタリング

自分が作成したプログラムは、拡張性がなく、可読性が低い。コマンドとしてほかにも考えられる入力が存在するが、あらゆる場合に対応するとなるとプログラムがより複雑になる。そのため、XV6 の OS のコードを読み、コードの見やすさや拡張性について理解を得た。参考プログラムから、リファクタリングを施した点は次の 3 つである。

(1) コマンドの内容を構造体で管理
(2) リスト構造でデータを柔軟に管理
(3) 逐次処理に変更

次に、リファクタリングを施した 3 つの点を具体的に説明する。

4.1 コマンドの内容を構造体で管理

構造体は、同じ種類のデータを管理することに長けている。リダイレクト、パイプなどのコマンドを実行するには、必要な情報が異なるが、リダイレクトの処理は同じ情報で処理できる。このように、同じタイプのデータをまとめて構造体で管理することで、処理の見通しが良くなると考える。次に実際に用いた構造体の定義を示す。

ソースコード 2　利用した構造体

```
1  struct cmd{ ···(1)
2    int type;
3  };
4
5  struct execcmd { ···(2)
6    int type;
7    char *argv[MAXARGS];
8  };
9
10 struct redircmd { ···(3)
11   int type;
12   struct cmd *cmd;
13   char *file;
14   int mode;
15   int fd;
16 };
17
18 struct pipecmd { ···(4)
19   int type;
20   struct cmd *left;
21   struct cmd *right;
22 };
```

(1) 基底構造体
基底構造体の役割は、クラスの抽象化と似ている。プログラム全体を通して同一の型を持つ構造体を用いることで、様々な処理を統一的に管理することができる。
(2) コマンド実行構造体
コマンド実行構造体は、ls や echo hello のようなコマンドを実行するための構造体である。argv の配列にコマンドの文字列を代入して、execvp 関数で実行する。
(3) リダイレクト構造体
リダイレクト構造体は、コマンドの出力先を標準に入力からファイルに変更するための情報を管理するための構造体である。ここで、struct cmd *cmd は、出力先をファイルに変更されたコマンドである。
(4) パイプ構造体
パイプ構造体は、パイプ処理を行う部分に関する構造体である。パイプ構造体自体はパイプの

ための処理を行わず、再帰処理を行うために利用される。

4.2 リスト構造でデータを柔軟に管理

リスト構造は、プログラム中にデータの個数が変化するものを扱うときに有効である。また、リスト構造を用いると再帰処理を記述しやすい。例えば、リダイレクトの処理はファイルの出力先を変更することと、コマンド実行する 2 つ行う必要がある。リストでそれぞれの手順を示すことができれば簡潔にコードを記述できる。次に示すコードはリダイレクト部分を記述した runcmd を一部抜粋したものである。

ソースコード 3　runcmd 一部抜粋

```
case REDIR:
  rcmd = (struct redircmd*)cmd;···(1)
  int fd=open(rcmd->file, rcmd->mode,
       S_IRUSR | S_IWUSR | S_IRGRP |
      S_IROTH);
  if (fd < 0) {
    perror("open failed");
    exit(-1);
  }
  dup2(fd,rcmd->fd);
  close(fd);
  runcmd(rcmd->cmd); ···(2)
  break;
```

(1) キャスト

基底構造体である cmd を、特定の構造体にキャストすることでリダイレクト構造体に変更することができる。リダイレクト構造体にキャストされたものは、リダイレクト情報をメンバに持つ。

(2) 再帰処理

この再帰処理は、リダイレクト構造体のメンバである cmd を引数としてもう一度 runcmd を行う。これにより、出力先を変更した後にコマンドを実行することができる

4.3 逐次処理に変更

自分が作成したプログラムは、コマンド一度すべて見てからどのようなコマンドが入力をされたか確認し、コマンドに応じた処理を行う。それに対して、修正したプログラムは先頭から文字列を見ていき、リダイレクトやパイプの場合はどうすればいいかを同一の手順で行うようにする。次に示すコードは parsecmd の一部を抜粋したコードである。

ソースコード 4　parsecmd 一部抜粋

```
exe = execcmd();
cmd = (struct execcmd*)exe;
while(argv[idx]!=NULL){
  if(strchr("<>",argv[idx][0])){
    if(strchr("<>|",argv[idx
        +1][0])){ ···(1)
      printf("syntax error\n");
      exit(-1);
    }
    switch(argv[idx][0]){ ···(2)
      case '<':
        exe=redircmd(exe,argv[idx+1],
            O_RDONLY, 0);
        break;
      case '>':
        if(argv[idx][1]=='>'){
          exe=redircmd(exe,argv[idx
              +1],O_WRONLY|O_CREAT,
              1);
        }else{
          exe=redircmd(exe,argv[idx
              +1],O_WRONLY|O_CREAT,
              1);
        }
        break;
    }

    idx+=2;
  }else if(strcmp(argv[idx],"|"
      )==0){
    idx++;
    exe=pipecmd(exe,parsepipe(argv));
        ···(3)
  } else { // コマンドのとき
    argc=0;
```

```
28      while(argv[idx]!=NULL){ ···(4)
29        if(strchr("<>",argv[idx][0])){
30          break;
31        }else if(strcmp(argv[idx],"|"
              )==0){
32          break;
33        }else{
34          cmd->argv[argc++]=argv[idx];
35          idx++;
36        }
37      }
38      cmd->argv[argc]=NULL;
39    }
40  }
41  idx=0;
42  return exe;
```

(1) エラー処理
リダイレクトの記号の次のコマンドブロックに、また同じリダイレクトまたはパイプ記号が存在するとき、コマンド書き方がおかしいとして、syntax error と出力するようにした。

(2) リダイレクトの種類を検出
リダイレクトの記号の種類を判別して、記号に応じて処理を変更する。

(3) パイプの再帰処理
パイプの左のポインタには、パイプのひとつ前のコマンドの構造体のアドレスを格納する。右には、パイプの後のコマンド内容を書き込む

(4) コマンド処理
コマンドを見つけた場合は、リダイレクトやパイプの記号になるまで cmd 構造体の argv にコマンド文字列を代入する。

5 実験

今回の実験は、リファクタリングしたコードは本来通りの動きを実現できていることとともに、コマンドが間違っている場合は正常にコマンド実行されず、syntax error と出力されるか確認する。次に、シェルプログラムを実行したときの実行結果である。ls > >と入力した場合はシェル上に syntax

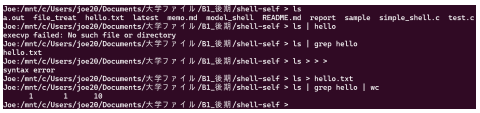

図 1　自作シェル実行結果

error と表示され、コマンドの書き方が間違っていると指摘している。通常のコマンド、パイプの実行、パイプが多段になった場合でも正常にプログラムが動作していることがわかる。また、リダイレクトの処理が正しく行われていることを確認するために、hello.txt の内容を次に示す。

```
1   README.md
2   a.out
3   file_treat
4   hello.txt
5   latest
6   memo.md
7   model_shell
8   report
9   sample
10  simple_shell.c
11  test.c
12  f
13  test.c
14
```

図 2　hello.txt 中身

確かに、ls の出力内容が hello.txt に反映されていることがわかる。

6 まとめ

本レポートでは、シェルの動作原理とリファクタリングについて学ぶことを目的とした。空白で区切られたコマンドの形式が定まっていることで、コマンドを容易に処理することができるのだと理解した。また、リファクタリングを行ったことで、リスト構造をうまく利用して簡潔かつ拡張性に優れたコードを記述する技術を取得できた。

付録

付録 1　パイプリダイレクト関数

```
1  void PipeRedirect(char **argv){
2    int is_redirect=0;
3    int is_oppose_redirect=0;
4    int *pipe_locate=NULL;
5    pipe_locate=(int*)malloc(sizeof(
         int));
6    pipe_locate[0]=-1;
7    int pipe_count=0;
8
9    for(int i=0;argv[i]!=NULL;i++){ ··
         ·(1)
10     if(strcmp(argv[i],">")==0){
11       is_redirect=i;
12       argv[i]=NULL;
13     }else if(strcmp(argv[i],"<"
           )==0){
14       is_oppose_redirect=i;
15       argv[i]=NULL;
16     }else if(strcmp(argv[i],"|"
           )==0){
17       pipe_count++;
18       pipe_locate=realloc(pipe_locate
             ,(pipe_count+1) * sizeof(
             int));
19       pipe_locate[pipe_count]=i;
20       argv[i]=NULL;
21     }
22   }
23
24   int pipefd[pipe_count+1][2];
25
26   if(pipe_count==0 && is_redirect==0
         && is_oppose_redirect==0){ ···
         (2)
27     if(fork()==0){
28       if(execvp(argv[0],argv)){
29         perror("execvp");
30         exit(EXIT_FAILURE);
31       }
32     }
33   }else if(is_oppose_redirect!=0){ ··
         ·(3)
34     if(fork()==0){
35       if(is_FileOrDir(argv[is_redirect
             +1])==0){
36         exit(1);
37       }
38       if(is_FileOrDir(argv[
             is_oppose_redirect+1])==0){
39         exit(1);
40       }
41       int fd_in=open(argv[
             is_oppose_redirect+1],
             O_RDONLY);
42       dup2(fd_in,STDIN_FILENO);
43       close(fd_in);
44       if(is_redirect!=0){
45         int fd_out=open(argv[
               is_redirect+1], O_CREAT |
                O_WRONLY, S_IRUSR |
               S_IWUSR | S_IRGRP |
               S_IROTH);
46         dup2(fd_out,STDOUT_FILENO);
47         close(fd_out);
48       }
49       if(execvp(argv[0],argv)){
50         perror("execvp");
51         exit(EXIT_FAILURE);
52       }
53     }
54   }else if(is_redirect!=0 &&
         pipe_count==0){ ···(4)
55     if(fork()==0){
56       if(is_FileOrDir(argv[is_redirect
             +1])==0){
57         exit(1);
58       }
59       int fd=open(argv[is_redirect
             +1], O_CREAT | O_WRONLY,
             S_IRUSR | S_IWUSR | S_IRGRP
              | S_IROTH);
60       dup2(fd, 1);
61       close(fd);
62       if(execvp(argv[0],argv)){
63         perror("execvp");
64         exit(EXIT_FAILURE);
65       }
66     }
```

```
67  }else{ ···(5)
68    for(int i=0;i<pipe_count+1;i++){
69      if(i!=pipe_count){
70        pipe(pipefd[i]);
71      }
72      if(fork()==0){
73        if(i==0){
74          close(pipefd[i][0]);
75          dup2(pipefd[i][1],
                STDOUT_FILENO);
76          close(pipefd[i][1]);
77        }else if(i==pipe_count){
78          dup2(pipefd[i-1][0],
                STDIN_FILENO);
79          close(pipefd[i-1][0]);
80          if(is_redirect){
81            int fd=open(argv[
                  is_redirect+1],
                  O_CREAT | O_WRONLY,
                  S_IRUSR | S_IWUSR |
                  S_IRGRP | S_IROTH);
82            dup2(fd, 1);
83            close(fd);
84          }
85          close(pipefd[i-1][1]);
86        }else{
87          close(pipefd[i][0]);
88          close(pipefd[i-1][1]);
89          dup2(pipefd[i-1][0],
                STDIN_FILENO);
90          dup2(pipefd[i][1],
                STDOUT_FILENO);
91          close(pipefd[i][1]);
92          close(pipefd[i-1][0]);
93        }
94        if(execvp(argv[pipe_locate[i]
              + 1], argv + pipe_locate[
              i] + 1) < 0){
95          perror("execvp");
96          exit(EXIT_FAILURE);
97        }
98      }else{
99      }
100   }
101 }
102 for(int i=0;i<pipe_count-1;i++){
103   wait(NULL);
104 }
105 free(pipe_locate);
106 }
```

付録 2　修正したコード

```
1  /*
2  * Copyright (c) 2017 Hiroshi
       Yamada <hiroshiy@cc.tuat.ac.jp
       >
3  *
4  * a simple shell
5  */
6
7  #include <stdio.h>
8  #include <string.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11
12 #include <sys/types.h>
13 #include <sys/wait.h>
14 #include <fcntl.h>
15 #include <linux/stat.h>
16 #include <sys/stat.h>
17
18 #define BUFSIZE 1024
19 #define ARGVSIZE 100
20
21 const char whitespace[] = " \t\r\n\
       v";
22
23 int idx=0;//配列の番号をプログラムを通
       して判断する
24
25 // Parsed command representation
26 #define EXEC 1
27 #define REDIR 2
28 #define PIPE 3
29
30 #define MAXARGS 10
31
32 struct cmd{
33   int type;
34 };
35
36 struct execcmd {
```

```
37   int type;
38   char *argv[MAXARGS];
39 };
40
41 struct redircmd {
42   int type;
43   struct cmd *cmd;
44   char *file;
45   int mode;
46   int fd;
47 };
48
49 struct pipecmd {
50   int type;
51   struct cmd *left;
52   struct cmd *right;
53 };
54
55 struct cmd*
56 execcmd(void)
57 {
58   struct execcmd *cmd;
59
60   cmd = malloc(sizeof(*cmd));
61   memset(cmd, 0, sizeof(*cmd));
62   cmd->type = EXEC;
63   return (struct cmd*)cmd;
64 }
65
66 struct cmd*
67 redircmd(struct cmd *subcmd, char *
       file, int mode, int fd)
68 {
69   struct redircmd *cmd;
70
71   cmd = malloc(sizeof(*cmd));
72   memset(cmd, 0, sizeof(*cmd));
73   cmd->type = REDIR;
74   cmd->cmd = subcmd;
75   cmd->file = file;
76   cmd->mode = mode;
77   cmd->fd = fd;
78   return (struct cmd*)cmd;
79 }
80
81 struct cmd*
82 pipecmd(struct cmd *left, struct cmd
       *right)
83 {
84   struct pipecmd *cmd;
85
86   cmd = malloc(sizeof(*cmd));
87   memset(cmd, 0, sizeof(*cmd));
88   cmd->type = PIPE;
89   cmd->left = left;
90   cmd->right = right;
91   return (struct cmd*)cmd;
92 }
93
94 struct cmd*
95 parsepipe(char **argv){
96   struct execcmd *cmd;
97   struct cmd *exe;
98
99   int argc;
100
101   exe = execcmd();
102   cmd = (struct execcmd*)exe;
103   while(argv[idx]!=NULL){
104     if(strchr("<>",argv[idx][0])){
105       if(strchr("<>|",argv[idx
            +1][0])){
106         printf("syntax error\n");
107         exit(-1);
108       }
109       switch(argv[idx][0]){
110         case '<':
111           exe=redircmd(exe,argv[idx
                +1],O_RDONLY, 0);
112           break;
113         case '>':
114           if(argv[idx][1]=='>'){
115             exe=redircmd(exe,argv[idx
                  +1],O_WRONLY|O_CREAT,
                  1);
116           }else{
117             exe=redircmd(exe,argv[idx
                  +1],O_WRONLY|O_CREAT,
                  1);
118           }
119           break;
120       }
```

```
121
122       idx+=2;
123     }else if(strcmp(argv[idx],"|"
            )==0){
124       idx++;
125       exe=pipecmd(exe,parsepipe(argv
              ));
126     } else { // コマンドのとき
127       argc=0;
128       while(argv[idx]!=NULL){
129         if(strchr("<>",argv[idx
                ][0])){
130           break;
131         }else if(strcmp(argv[idx],"|"
                )==0){
132           break;
133         }else{
134           cmd->argv[argc++]=argv[idx];
135           idx++;
136         }
137       }
138       cmd->argv[argc]=NULL;
139     }
140   }
141   return exe;
142 }
143
144 struct cmd* parsecmd(char **argv,
        char *buf, char *ebuf)
145 {
146   char *s;
147   int i = 0;
148
149   s = buf;
150
151   while (s < ebuf) {
152     while (s < ebuf && strchr(
            whitespace, *s)) s++;
153     if (ebuf <= s) break;
154
155     argv[i++] = s;
156     while (s < ebuf && !strchr(
            whitespace, *s)) s++;
157     *s = '\0';
158     s++;
159   }
160   argv[i++]=NULL;
161
162   struct execcmd *cmd;
163   struct cmd *exe;
164
165   int argc;
166
167   exe = execcmd();
168   cmd = (struct execcmd*)exe;
169   while(argv[idx]!=NULL){
170     if(strchr("<>",argv[idx][0])){
171       if(strchr("<>|",argv[idx
              +1][0])){
172         printf("syntax error\n");
173         exit(-1);
174       }
175       switch(argv[idx][0]){
176         case '<':
177           exe=redircmd(exe,argv[idx
                  +1],O_RDONLY, 0);
178           break;
179         case '>':
180           if(argv[idx][1]=='>'){
181             exe=redircmd(exe,argv[idx
                    +1],O_WRONLY|O_CREAT,
                    1);
182           }else{
183             exe=redircmd(exe,argv[idx
                    +1],O_WRONLY|O_CREAT,
                    1);
184           }
185           break;
186       }
187
188       idx+=2;
189     }else if(strcmp(argv[idx],"|"
            )==0){
190       idx++;
191       exe=pipecmd(exe,parsepipe(argv
              ));
192     } else { // コマンドのとき
193       argc=0;
194       while(argv[idx]!=NULL){
195         if(strchr("<>",argv[idx
                ][0])){
196           break;
```

```
197         }else if(strcmp(argv[idx],"|"
                )==0){
198           break;
199         }else{
200           cmd->argv[argc++]=argv[idx];
201           idx++;
202         }
203       }
204       cmd->argv[argc]=NULL;
205     }
206   }
207   idx=0;
208   return exe;
209 }
210
211 void runcmd(struct cmd* cmd)
212 {
213   int p[2];
214
215   struct execcmd *ecmd;
216   struct redircmd *rcmd;
217   struct pipecmd *pcmd;
218
219   if(cmd==NULL)
220     exit(-1);
221
222   switch(cmd->type){
223     default:
224       exit(-1);
225     case EXEC:
226       ecmd=(struct execcmd*)cmd;
227       execvp(ecmd->argv[0], ecmd->
              argv);
228       perror("execvp failed");
229       break;
230     case REDIR:
231       rcmd = (struct redircmd*)cmd;
232       int fd=open(rcmd->file, rcmd->
              mode);
233       if (fd < 0) {
234         perror("open failed");
235         exit(-1);
236       }
237       dup2(fd,rcmd->fd);
238       close(fd);
239       runcmd(rcmd->cmd);
240       break;
241
242     case PIPE:
243       pcmd = (struct pipecmd*)cmd;
244       if(pipe(p) < 0)
245         perror("pipe");
246       if(fork() == 0){
247         close(1);
248         dup(p[1]);
249         close(p[0]);
250         close(p[1]);
251         runcmd(pcmd->left);
252       }
253       if(fork() == 0){
254         close(0);
255         dup(p[0]);
256         close(p[0]);
257         close(p[1]);
258         runcmd(pcmd->right);
259       }
260       close(p[0]);
261       close(p[1]);
262       wait(NULL);
263       wait(NULL);
264       break;
265   }
266
267   exit(0);
268 }
269
270 int getcmd(char *buf, int len)
271 {
272   char path_name[BUFSIZE];
273   getcwd(path_name, BUFSIZE);
274   printf("Joe:%s ",path_name);
275   printf("> ");
276   fflush(stdin);
277   memset(buf, 0, len);
278   fgets(buf, len, stdin);
279
280   if (buf[0] == 0)
281     return -1;
282   return 0;
283 }
284
285 int main(int argc, char**argv)
```

```
286 {
287   static char buf[BUFSIZE];
288
289   while(getcmd(buf, BUFSIZE) >= 0) {
290     if(buf[0] == 'c' && buf[1] == '
          d' && buf[2] == ' '){
291       // Chdir must be called by
            the parent, not the child
            .
292       buf[strlen(buf)-1] = 0; //
            chop \n
293       if(chdir(buf+3) < 0)
294         printf("cannot cd %s\n", buf
              +3);
295       continue;
296     }
297     if (fork() == 0)
298       runcmd(parsecmd(argv, buf, &buf[
            strlen(buf)]));
299     wait(NULL);
300   }
301
302   exit(0);
303 }
```

付録 3　XV6 のコード

```
1    // Shell.
2
3  #include "types.h"
4  #include "user.h"
5  #include "fcntl.h"
6
7  // Parsed command representation
8  #define EXEC 1
9  #define REDIR 2
10 #define PIPE 3
11 #define LIST 4
12 #define BACK 5
13
14 #define MAXARGS 10
15
16 struct cmd {
17   int type;
18 };
19
20 struct execcmd {
21   int type;
22   char *argv[MAXARGS];
23   char *eargv[MAXARGS];
24 };
25
26 struct redircmd {
27   int type;
28   struct cmd *cmd;
29   char *file;
30   char *efile;//ファイルの名前を管理す
       るため
31   int mode;
32   int fd;
33 };
34
35 struct pipecmd {
36   int type;
37   struct cmd *left;
38   struct cmd *right;
39 };
40
41 struct listcmd {
42   int type;
43   struct cmd *left;
44   struct cmd *right;
45 };
46
47 struct backcmd {
48   int type;
49   struct cmd *cmd;
50 };
51
52 int fork1(void); // Fork but panics
       on failure.
53 void panic(char*);
54 struct cmd *parsecmd(char*);
55
56 // Execute cmd. Never returns.
57 void
58 runcmd(struct cmd *cmd)
59 {
60   int p[2];
61   struct backcmd *bcmd;
62   struct execcmd *ecmd;
63   struct listcmd *lcmd;
64   struct pipecmd *pcmd;
```

```
65  struct redircmd *rcmd;
66
67  if(cmd == 0)
68    exit();
69
70  switch(cmd->type){
71  default:
72    panic("runcmd");
73
74  case EXEC:
75    ecmd = (struct execcmd*)cmd;
76    if(ecmd->argv[0] == 0)
77      exit();
78    exec(ecmd->argv[0], ecmd->argv);
79    printf(2, "exec %s failed\n",
          ecmd->argv[0]);
80    break;
81
82  case REDIR:
83    rcmd = (struct redircmd*)cmd;
84    close(rcmd->fd);
85    if(open(rcmd->file, rcmd->mode) <
         0){
86      printf(2, "open %s failed\n",
           rcmd->file);
87      exit();
88    }
89    runcmd(rcmd->cmd);//再帰で呼び出し
        た場合は、その処理はどうなるの
        forkすると
90    break;
91
92  case LIST:
93    lcmd = (struct listcmd*)cmd;
94    if(fork1() == 0)
95      runcmd(lcmd->left);
96    wait();
97    runcmd(lcmd->right);
98    break;
99
100 case PIPE:
101   pcmd = (struct pipecmd*)cmd;
102   if(pipe(p) < 0)
103     panic("pipe");
104   if(fork1() == 0){
105     close(1);
106     dup(p[1]);
107     close(p[0]);
108     close(p[1]);
109     runcmd(pcmd->left);
110   }
111   if(fork1() == 0){
112     close(0);
113     dup(p[0]);
114     close(p[0]);
115     close(p[1]);
116     runcmd(pcmd->right);
117   }
118   close(p[0]);
119   close(p[1]);
120   wait();
121   wait();
122   break;
123
124 case BACK:
125   bcmd = (struct backcmd*)cmd;
126   if(fork1() == 0)
127     runcmd(bcmd->cmd);
128   break;
129 }
130 exit();
131 }
132
133 int
134 getcmd(char *buf, int nbuf)
135 {
136   printf(2, "$ ");
137   memset(buf, 0, nbuf);
138   gets(buf, nbuf);
139   if(buf[0] == 0) // EOF
140     return -1;
141   return 0;
142 }
143
144 int
145 main(void)
146 {
147   static char buf[100];
148   int fd;
149
150   // Ensure that three file
        descriptors are open.
```

```
151   while((fd = open("console", O_RDWR
          )) >= 0){
152     if(fd >= 3){
153       close(fd);
154       break;
155     }
156   }
157
158   // Read and run input commands.
159   while(getcmd(buf, sizeof(buf)) >=
          0){
160     if(buf[0] == 'c' && buf[1] == '
            d' && buf[2] == ' '){
161       // Chdir must be called by
              the parent, not the child
              .
162       buf[strlen(buf)-1] = 0; //
              chop \n
163       if(chdir(buf+3) < 0)
164         printf(2, "cannot cd %s\n",
                buf+3);
165       continue;
166     }
167     if(fork1() == 0)
168       runcmd(parsecmd(buf));
169     wait();
170   }
171   exit();
172 }
173
174 void
175 panic(char *s)
176 {
177   printf(2, "%s\n", s);
178   exit();
179 }
180
181 int
182 fork1(void)
183 {
184   int pid;
185
186   pid = fork();
187   if(pid == -1)
188     panic("fork");
189   return pid;
190 }
191
192 //PAGEBREAK!
193 // Constructors
194
195 struct cmd*
196 execcmd(void)
197 {
198   struct execcmd *cmd;
199
200   cmd = malloc(sizeof(*cmd));
201   memset(cmd, 0, sizeof(*cmd));
202   cmd->type = EXEC;
203   return (struct cmd*)cmd;
204 }
205
206 /**
207  * 新しいポインタを宣言して、元々exec
        を示していたポインタをsubcmdに代
        入している
208  * redirectの部分を見るとそんな感じが
        する
209  * キャストは何でしているの？
210  */
211
212 struct cmd*
213 redircmd(struct cmd *subcmd, char *
        file, char *efile, int mode, int
         fd)
214 {
215   struct redircmd *cmd;
216
217   cmd = malloc(sizeof(*cmd));
218   memset(cmd, 0, sizeof(*cmd));
219   cmd->type = REDIR;
220   cmd->cmd = subcmd;
221   cmd->file = file;
222   cmd->efile = efile;
223   cmd->mode = mode;
224   cmd->fd = fd;
225   return (struct cmd*)cmd;
226 }
227
228 struct cmd*
229 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)//先頭と終端
{
  char *s;//これは文字列を操作してどんな文字であるかを見るために利用する
  int ret;

  s = *ps;
  while(s < es && strchr(whitespace, *s))//空白を飛ばして、ファイルの先頭に移動
    s++;
  if(q)//qはリダイレクトやパイプのファイル名を示している、アドレスが渡されているから、条件は真になる
    *q = s;
  ret = *s;//リダイレクト系の記号が入る、先頭が入る
  switch(*s){//はじめ
  case 0:
    break;
  case '|'://リダイレクトの符号があるときは、
  case '('':
  case ')':
  case ';':
  case '&':
  case '<':
    s++;
    break;
  case '>':
    s++;
    if(*s == '>'){
      ret = '+';
      s++;
    }
    break;
  default://文字コマンドを指している場合は、文字列を飛ばす
    ret = 'a';
    while(s < es && !strchr(
        whitespace, *s) && !strchr(
        symbols, *s))//文字列をすべて飛ばす
      s++;
    break;
  }
  if(eq)//コマンドの終端を表している、
```

```
        defaultの時は、アドレスが渡されているから条件は真になる
308     *eq = s;
309
310   while(s < es && strchr(whitespace, *s))//リダイレクトや何かの記号まで進める、sは結局何か意味ある文字の先頭を表す
311     s++;
312   *ps = s;
313   return ret;//retが+の時はダブルリダイレクト、コマンドの時はaでしゅ
314 }
315
316
317 /**
318  * @brief 文字列を受け取って、最後の文字まで進める
319  * @note 特定の文字の場合スキップする関数,whitespaceに含まれる文字
320  * 空白文字の場合に使用するはず
321  * 初め
322  */
323 int
324 peek(char **ps, char *es, char *toks)
325 {
326   char *s;
327
328   s = *ps;
329   while(s < es && strchr(whitespace, *s))//空白を飛ばす
330     s++;
331   *ps = s;
332   return *s && strchr(toks, *s);//真偽を返す
333 }
334
335 struct cmd *parseline(char**, char*);
336 struct cmd *parsepipe(char**, char*);
337 struct cmd *parseexec(char**, char*);
338 struct cmd *nulterminate(struct cmd*);
339
340 struct cmd*
341 parsecmd(char *s)
342 {
343   char *es;
344   struct cmd *cmd;
345
346   es = s + strlen(s);//esは終端文字
347   cmd = parseline(&s, es);
348   peek(&s, es, "");
349   if(s != es){//コマンドの最後まで行ったよ
350     printf(2, "leftovers: %s\n", s);
351     panic("syntax");
352   }
353   nulterminate(cmd);
354   return cmd;
355 }
356
357 struct cmd*
358 parseline(char **ps, char *es)// esは終端文字 sは先頭文字
359 {
360   struct cmd *cmd;
361
362   cmd = parsepipe(ps, es);
363   while(peek(ps, es, "&")){
364     gettoken(ps, es, 0, 0);
365     cmd = backcmd(cmd);
366   }
367   if(peek(ps, es, ";")){
368     gettoken(ps, es, 0, 0);
369     cmd = listcmd(cmd, parseline(ps, es));
370   }
371   return cmd;
372 }
373
374 struct cmd*
375 parsepipe(char **ps, char *es)// 先頭と終端
376 {
377   struct cmd *cmd;
378
379   cmd = parseexec(ps, es);
380   if(peek(ps, es, "|")){
```

```
381     gettoken(ps, es, 0, 0);
382     cmd = pipecmd(cmd, parsepipe(ps,
            es));
383   }
384   return cmd;
385 }
386
387 struct cmd*
388 parseredirs(struct cmd *cmd, char **
        ps, char *es)
389 {
390   int tok;
391   char *q, *eq;//リダイレクト先のfile
          のはじめと最後
392
393   while(peek(ps, es, "<>")){//リダイ
          レクトの記号があるかを確認
394     tok = gettoken(ps, es, 0, 0);//リ
          ダイレクトの符号を取得
395     if(gettoken(ps, es, &q, &eq) !=
          'a')//リダイレクト先のfileを
          取得
396       panic("missing file for
              redirection");
397     switch(tok){
398     case '<':
399       cmd = redircmd(cmd, q, eq,
              O_RDONLY, 0);
400       break;
401     case '>':
402       cmd = redircmd(cmd, q, eq,
              O_WRONLY|O_CREATE, 1);
403       break;
404     case '+': // >>
405       cmd = redircmd(cmd, q, eq,
              O_WRONLY|O_CREATE, 1);
406       break;
407     }
408   }
409   return cmd;
410 }
411
412 struct cmd*
413 parseblock(char **ps, char *es)
414 {
415   struct cmd *cmd;
416
417   if(!peek(ps, es, "("))
418     panic("parseblock");
419   gettoken(ps, es, 0, 0);
420   cmd = parseline(ps, es);
421   if(!peek(ps, es, ")"))
422     panic("syntax - missing )");
423   gettoken(ps, es, 0, 0);
424   cmd = parseredirs(cmd, ps, es);
425   return cmd;
426 }
427
428 struct cmd*
429 parseexec(char **ps, char *es)//先頭
        と終端,実行準備
430 {
431   char *q, *eq;
432   int tok, argc;
433   struct execcmd *cmd;
434   struct cmd *ret;
435
436   if(peek(ps, es, "("))//空白を飛ばし
          て、()があるかを確認
437     return parseblock(ps, es);
438
439   ret = execcmd();
440   cmd = (struct execcmd*)ret;
441
442   argc = 0;
443   ret = parseredirs(ret, ps, es);//
          retはtypeにEXECが入っている
444   while(!peek(ps, es, "|)&;")){//空
          白を飛ばして、記号がないとき
445     if((tok=gettoken(ps, es, &q, &eq
            )) == 0)//0は文字列の終端を表
            す
446       break;
447     if(tok != 'a')//コマンドの時
448       panic("syntax");
449     cmd->argv[argc] = q;//lsなら1
450     cmd->eargv[argc] = eq;//lsなら空
          白が入る
451     argc++;//コマンドの個数
452     if(argc >= MAXARGS)
453       panic("too many args");
454     ret = parseredirs(ret, ps, es);
```

```
455   }
456   cmd->argv[argc] = 0;//この処理は保
          険のためかな、何もないとここには0を
          代入している
457   cmd->eargv[argc] = 0;
458   return ret;
459 }
460
461 // NUL-terminate all the counted
        strings.
462 struct cmd*
463 nulterminate(struct cmd *cmd)
464 {
465   int i;
466   struct backcmd *bcmd;
467   struct execcmd *ecmd;
468   struct listcmd *lcmd;
469   struct pipecmd *pcmd;
470   struct redircmd *rcmd;
471
472   if(cmd == 0)
473     return 0;
474
475   switch(cmd->type){//終端にNULLを入
          れいている良そう
476   case EXEC:
477     ecmd = (struct execcmd*)cmd;
478     for(i=0; ecmd->argv[i]; i++)
479       *ecmd->eargv[i] = 0;//eargvです
            よ終端を表しているものですよ
480     break;
481
482   case REDIR:
483     rcmd = (struct redircmd*)cmd;
484     nulterminate(rcmd->cmd);
485     *rcmd->efile = 0;//ファイルの終端
            を表している
486     break;
487
488   case PIPE:
489     pcmd = (struct pipecmd*)cmd;
490     nulterminate(pcmd->left);
491     nulterminate(pcmd->right);
492     break;
493
494   case LIST:
495     lcmd = (struct listcmd*)cmd;
496     nulterminate(lcmd->left);
497     nulterminate(lcmd->right);
498     break;
499
500   case BACK:
501     bcmd = (struct backcmd*)cmd;
502     nulterminate(bcmd->cmd);
503     break;
504   }
505   return cmd;
506 }
```

参考文献

[1] open 関数の使い方、正しく理解していますか?, 2024/2/10, https://progzakki.sanachan.com/program-lang/c/how-to-use-open/

[2] シェルの多段パイプを自作してみる, 2024/2/10, https://keiorogiken.wordpress.com/2017/12/15/%E3%82%B7%E3%82%A7%E3%83%AB%E3%81%AE%E5%A4%9A%E6%AE%B5%E3%83%91%E3%82%A4%E3%83%97%E3%82%92%E8%87%AA%E4%BD%9C%E3%81%97%E3%81%A6%E3%81%BF%E3%82%8B/

[3] C 言語で理解するリダイレクトの全て！初心者もマスターできる 7 つのステップ, 2024/2/10, https://jp-seemore.com/iot/11246/