Shell-Dokumentation

Laurin Wassmann

April 2022

Inhaltsverzeichnis

1 Aufgabenstellung

Es soll eine Shell programmiert werden die folgende Programme beinhalten soll:

- **wc:** Zählt die Anzahl von Wörtern in einer Datei.
- **cat:** Gibt den Inhalt einer Datei aus.
- **pwd:** Gibt den aktuellen Pfad aus.
- **echo:** Gibt einen String am Bidlschirm aus.
- **touch:** Erstellt eine neue Datei.
- **rm:** Löscht eine Datei.
- **ls:** Gibt den Inhalt des aktuellen Ordners aus.
- **mkdir:** Erstellt einen neuen Ordner im aktuellen Verzeichnis.
- **cd:** Wechselt in ein anderes Verzeichnis.

2 Aufbau/Theorie

2.1 Prozesse

2.1.1 Was ist ein Prozess

Ein Prozess ist ein Programm während der Ausführung, bzw. eine Aktivität irgend einer Art, inklusive der Befehle und der Registerinhalte.

2.1.2 Prozess erstellen und entfernen

Ein Prozess wird vom INIT-Prozess erzeugt. Dieser INIT-Prozess wird beim Booten eines UNIX-Systems erzeugt. Dieser INIT-Prozess kann mithilfe des Systemaufrufs fork eine exakte Kopie des aufrufenden Prozess mit dem gleichen Speicherabbild, den gleichen Umgebungsvariablen und den gleichen geöffneten Dateien erstellen. Wenn der Rückgabewert 0 ist handelt es sich um das Kindprozess. Ist er positiv ist es der Elternprozess und der Rückgabewert ist die PID. Bei einem negativen Rückgabewert ist ein Fehler aufgetreten.

Der Kindprozess führt danach den Systemaufruf execvp aus, der das Speicherbild wechselt und ein neues Programm ausführt.

2.2 Aufbau der Shell

2.2.1 Main

Die Main besteht aus einer do-while Schleife, die die Funktionen Readlinie, Splitline und startprocess aufruft, solange die Funktion startprocess nicht NULL zurückgibt.

```
1
2           int main(int argc, char *argv[]){
3
4       char *line;
5       char **arguments;
6       int status;
7
8       do{
9           printf(" ");
10          line = ReadLine();
11          arguments = SplitLine(line);
12          status = startprocess(arguments);
13          free(line);
14          free(arguments);
15      } while(status);
16  }
```

2.2.2 Eingabe/Aufteilung in Argumente

Für die Eingabe habe ich die Funktion ReadLine erstellt. Diese liest solange Character in ein Char-Array ein, bis "öder getchar einen Fehler zurückgibt. In diesem Fall wird dann am Ende ein "ëingefügt. Falls das Char-Array voll ist, wird weiterer Platz zum Array hinzugefügt.

```
1   char *ReadLine(){
2
3       int i = 0, size_buffer = 1024, c;
4       char *buffer = ((char*) malloc(sizeof(char) * size_buffer
        ));
5
6       while(1){
7           c = getchar();  //if c -1 == endoffile
8
9           if(c == '\n'||c == -1){
10              buffer[i] = '\0';
11              return buffer;
12          }else{
13              buffer[i] = c;
14          }
15
16          i++;
17
18          if(i >= size_buffer){
19              size_buffer += size_buffer;
```

```
20              buffer = realloc(buffer, size_buffer);
21
22              if(!buffer){
       //if buffer == 0
23                  perror("Fehler!");
24                  exit(EXIT_FAILURE);
25              }
26          }
27      }
28  }
```

Als nächstes wird dieses Char-array dann in die Funktion SplitLine gegeben, wo mithilfe der Funktion strtok die eingegebene Zeile in mehrere Token aufgeteilt wird, indem nach Leerzeichen im Char-array gesucht wird. Die einzelnen Token werden dann in das Char-array Tokens gegeben. Diese werden dann zurückgegeben.

```
1
2   char **SplitLine(char *line){
3
4       int index = 0, size_buffer = 64;
5       char *token;
6       char **tokens = ((char**) malloc(sizeof(char) *
        size_buffer));
7       token = strtok(line, " ");
8
9       while(token != NULL){
10          tokens[index] = token;
11          index++;
12
13          if(index >= size_buffer){
14              size_buffer += size_buffer;
15              tokens = realloc(tokens, size_buffer);
16
17              if(!tokens){
18                  perror("The following error occured: ");
19                  exit(EXIT_FAILURE);
20              }
21
22          }
23          token = strtok(NULL, " \t\r\n\a");
24      }
25      tokens[index] = NULL;
26      return tokens;
27  }
```

2.2.3 Prozess ausführen

Die Funktion startprocess überprüft erst, ob in der Eingabe ëxitëigegeben wurde und falls dies der Falls ist, wird dieser Befehl ausgeführt. Ansonsten wird mit fork ein Kind erstellt und dann überprüft ob "cdëingegeben wurde. Ist das der Fall, wird dieser Befehl ebenfalls ausgeführt. Ansonsten wird mit execvp der Pfad des auszuführenden Befehls und die Übergabeparameter (tokens aus der Funktion SplitLine) übergeben, der den Befehl ausführt. Der Vater wartet bis das Kind sich beendet hat und falls ein Fehler aufgetreten ist, wird dieser augegeben.

```
1
2
3  int startprocess(char **args){
4
5      pid_t pid;
6      char befehl[100];
7
8      //Ordner, worin sich die Befehle befinden.
9      strcpy(befehl,"/home/mp3bruh/Vorlagen/CLionProjects/
       Shell_1.2");
10     strcat(befehl,"/");
11     strcat(befehl,args[0]);
12
13
14     if(strcmp(args[0], "exit") == 0){
15         return NULL;
16     }
17
18     else{
19         pid = fork(); // Kind wird erstellt
20         if(pid == 0){
21
22             if(strcmp("cd",args[0]) == 0) {
23
24                 if(chdir(args[1]) == -1) {
25                     perror("Fehler!\n");
26                     return EXIT_FAILURE;
27                 }
28             }
29
30             else if(execvp(befehl,args) == -1){
31                 printf("\n'%s' nicht gefunden!\n",args[0]);
32             }
33         }
34
```

```
35          else if(pid < 0){
36              printf("%s konnte nicht ausgefhrt werden!\n",args
        [0]);
37              exit(EXIT_FAILURE);
38          }
39          else{
40              waitpid(-1, NULL, 0);
41          }
42          return 1;
43      }
44  }
```

3 Durchführung/Code

3.1 cd

Der Befehl cd wird in der Funktion startprocess ausgeführt , welche die Funktion chdir ausführt. Diese findet man in der unistd.h Bibliothek. Falls das Verzeichnis mit dem eingegebenen Namen nicht gefunden wird, wird ein Fehler zurückgegeben.

```
1  if(strcmp("cd",args[0]) == 0) {
2
3                  if (chdir(args[1]) == -1) {
4                      perror("Fehler!\n");
5                      return EXIT_FAILURE;
6                  }
7              }
```

3.2 ls

Der Befehl ls benutzt die Funktion scandir aus der Bibliothek dirent.h . Die Funktion bekommt "."für den aktuellen Ordner und NULL für die zu suchenden Dateien, also alle Typen und alphasort um nach Alphabetischer Reihenfolge zu suchen. Diese werden dann in namelist gespeichert und anschließend rückwärts ausgegeben.

```
1
2  #include <dirent.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6
7  int main(void)
```

```
{
    struct dirent **namelist;
    int n = scandir(".", &namelist, NULL, alphasort);

    if(n == -1){
        perror("scandir");
        exit(EXIT_FAILURE);
    }

    while(n--){
        printf("%s\n", namelist[n]->d_name);
        free(namelist[n]);
    }

    printf("\n");
    free(namelist);
    exit(EXIT_SUCCESS);
}
```

3.3 cat

Bei dem Befehl cat wird mit der Funktion fopen die Datei mit dem eingegeben Namen geöffnet ("r"für read). Dann werden mit getc() solange Wörter aus der Datei gelesen, bis das Ende der Datei erreicht wurde. Mit printf() wird das Wort ausgegeben. Falls der Rückgabeparameter von fopen() NULL ist, wird eine Fehlermeldung ausgegeben und der Prozess beendet.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    int c;
    FILE *file = NULL;
    file = fopen(*(argv+1),"r");

    if(file) {
        while ((c = getc(file))!= EOF){
            printf("%c",c);
        }
        fclose(file);
        printf("\n");
    }
    else{
        perror("Fehler");
```

```
19          exit(EXIT_FAILURE);
20      }
21  }
```

3.4 wc

Der Befehl wc ist vom Aufbau gleich wie cat, nur das anstatt Wörter eine Zählervariable iteriert und diese zum Schluss ausgegeben wird.

```
1   #include <stdlib.h>
2
3   int main(int argc, char *argv[]){
4
5       int c,i = 0;
6       FILE *file = NULL;
7
8       file = fopen(*(argv+1),"r");
9       if(file != NULL) {
10          while ((c = getc(file))!= EOF){
11              i++;
12          }
13          fclose(file);
14          printf("%s -> %d Wrter\n",*(argv+1), i);
15      }else{
16          perror("Fehler");
17          fclose(file);
18          exit(EXIT_FAILURE);
19
20      }
21  }
```

3.5 echo

Der Befehl echo ist wahrscheinlich der Einfachste. Es werden alle Übergabeparameter nach dem Befehl ausgegeben bis keine mehr übrig sind.

```
1   #include <stdio.h>
2
3   int main(int argc, char *argv[]){
4
5       int i = 1;
6       while(*(argv+i)!= NULL){
7           printf("%s ", *(argv+i));
8           i++;
9       }
```

```
10      printf("\n");
11  }
```

3.6 touch

Bei touch wird erst versucht den eingegeben Namen der Datei zu öffnen. Falls der Rückgabewert nicht NULL ist, wird eine Fehlermeldung zurückgegeben. Dies bedeutet, dass die Datei bereits existiert. Ansonsten wird mit fopen() und (w für write) die Datei erstellt.

```
1   #include <stdio.h>
2   #include <sys/stat.h>
3   #include <fcntl.h>
4   #include <stdlib.h>
5
6   int main(int argc, char *argv[]) {
7
8       FILE* file;
9
10      if((file = fopen(*(argv+1),"r")) != NULL){
11          fclose(file);
12          perror("Fehler");
13          exit(EXIT_FAILURE);
14
15
16      }
17      file = fopen(*(argv+1),"w");
18      fclose(file);
19      printf("%s wurde erfolgreich erstell!\n", *(argv+1));
20
21  }
```

3.7 mkdir

Bei mkdir wird die Funktion mkdir aus der Bibliothek stdio.h benutzt. Hierbei wird der Name des zu erstellenden Verzeichnisses und die Zahl 0755 in die Funktion gegeben. 0755 steht dabei für die Rechte und heißt, dass der Eigentümer alle Rechte hat und der Rest nur lesen und ausführen darf.

```
1   #include <sys/stat.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   int main(int argc, char *argv[]) {
6
```

```
7       if(mkdir(*(argv+1),0755) == -1){
8           perror("Fehler");
9           exit(EXIT_FAILURE);
10      }else{
11          printf("%s wurde erfolgreich erstellt!\n", *(argv +
        1));
12      }
13
14  }
```

3.8 rm

Mit dem Befehl rm können sowohl Dateien als auch Verzeichnisse gelöscht werden. Ich benutze hierbei die Funktion remove aus der stdlib.h - Bibliothek. Remove benutzt dabei die Funktionen unlink für Dateien und rmdir für Verzeichnisse. Falls ein anderer Prozess gerade auf die Datei zugreift, wird die Datei erst dann gelöscht, sobald alle Datei-Pointer, die auf die Datei zeigen, geschlossen wurden. Falls remove einen Fehler zurückgibt, wird dieser ausgegeben.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(int argc, char *argv[]) {
5
6       if(remove(*(argv+1)) == -1){
7           perror("Fehler");
8           exit(EXIT_FAILURE);
9       }
10      printf("%s wurde erfolgreich gelscht!\n", *(argv+1));
11
12  }
```

3.9 pwd

Bei dem Befehl pwd wird mithilfe der Funktion getcwd aus der Bibliothek unistd.h der Name des aktuellen Verzeichnisses geholt und anschließend ausgegeben.

```
1   #include <unistd.h>
2   #include <stdio.h>
3
4   int main(int argc, char *argv[]) {
5
6       char pwd[256];
7
```

```
8       printf("%s\n",getcwd(pwd,sizeof(pwd)));
9   }
```

3.10 exit

Der Befehl exit befindet sich wie cd nicht in einer eigenen c-Datei sondern in der Funktion startprocess. Hierbei wird nur überprüft ob ëxitëingegeben wurde und falls dies der Fall ist NULL zurückgegeben. Nur in diesem Fall wird in der Main die do-while Schleife beendet.

```
1
2   if(strcmp(args[0], "exit") == 0){
3           return NULL;
4       }
```