2. Mini-Shell

In dieser Aufgabe geht es darum, eine kleine **Shell** zu entwickeln, also ein Programm, das Kommandos entgegennimmt, interpretiert und ausführt.

Sie bearbeiten diese Aufgabe in mehreren Einzelschritten.

a) Geben Sie in einem Editor das folgende Miniprogramm ein und speichern Sie es als `shell01.c`:

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

int main () {
  char command[255];

  while (1) {
    printf ("Eingabe: ");
    fgets (command, sizeof(command), stdin);
    printf ("Eingabe: '%s', Laenge: %d \n", command, strlen(command));
  }
}
```

Wenn Sie das Programm kompilieren (`gcc -o shell01 shell01.c`) und dann ausführen (`./shell01`), sehen Sie, dass es nicht wie gewünscht arbeitet. (Um das laufende Programm abzubrechen, drücken Sie [Strg-C].)

b) Warum erscheint in der Ausgabe ein (unerwünschter) Zeilenumbruch?

c) Modifizieren Sie das Programm so, dass es nach dem `fgets()`-Aufruf den String verändert – das folgende `printf()`-Kommando soll den Zeilenumbruch nicht mehr ausgeben. Hier hilft Ihnen auch die Funktion `strlen()`. Die neue Version speichern Sie als `shell02.c` und übersetzen sie mit `gcc -o shell02 shell02.c`.

d) Was würde passieren, wenn Sie einen sehr langen Text (z. B. mit 300 Zeichen) eingeben? Um diese Frage (ohne Testen) beantworten zu können, lesen Sie mit `man fgets` die Manpage zu `fgets()`. Bis zu wie vielen Zeichen funktioniert das Programm korrekt?

e) Ihr Programm soll nun die Eingabe in einzelne „Worte“ unterteilen, Worttrenner sind so genannte Whitespace-Zeichen (Leerzeichen und Tabulator `\t`). Lesen Sie zunächst die Manpage der Funktion `strtok()` (mit `man strtok`). Die Funktion muss mehrfach aufgerufen werden, um alle Teile der Eingabe zu verarbeiten. Beim ersten Aufruf übergeben Sie ihr (im ersten Argument) den zu bearbeitenden String, bei allen folgenden Aufrufen einen Nullzeiger (`NULL`). Außerdem müssen Sie die Worttrenner (als zweites Argument) angeben.

Kopieren Sie das Programm nach `shell03.c` und fügen Sie in dieser Version eine Variable `seps` (Separatoren) sowie einen Char-Pointer `part` hinzu:

```
char seps[] = " \t";
char *part;
```

Testen Sie zunächst die Funktion `strtok()`, indem Sie diese zweimal aufrufen (`part = strtok (...);`) und nach jedem Aufruf den String `part` ausgeben. Der eingegebene String sollte dabei mindestens zwei durch Leerzeichen getrennte Worte enthalten.

Ersetzen Sie nun die zwei Aufrufe durch eine While-Schleife. Wenn es keine Wortbestandteile mehr gibt, gibt `strtok()` einen Nullzeiger zurück. Ihr Programm soll jetzt alle Teile der Eingabe separat ausgeben.

f) Im nächsten Schritt streichen Sie die Ausgaben und speichern stattdessen die einzelnen Teile in einem Array von Strings (`char *args[10]`). Damit schaffen Sie Platz für zehn Pointer. Erzeugen Sie dazu zunächst eine Kopie `shell04.c`, in der Sie dies implementieren.

Sie müssen für die einzelnen Bestandteile (im Folgenden „Argumente“ genannt) keinen Speicherplatz reservieren, weil `strtok()` die vorhandene Eingabe verändert und jeweils eines der Trennzeichen durch die String-Terminierung (`\0`) ersetzt. Platz brauchen Sie also nur für die Zeiger auf die einzelnen Argumente. Programmieren Sie die Schleife so, dass maximal neun Argumente akzeptiert werden. (Den 10. Pointer brauchen Sie für Teilaufgabe **g**.) Überzählige Argumente soll Ihr Programm verwerfen (ignorieren).

Wenn Sie die Zerlegung beendet haben, geben Sie in einer Schleife die Elemente von `args` aus, das soll so aussehen:

```
int i;
printf ("Zerlegung: \n");
for ( i=0; i<no_args; i++ ) {
  printf ("Argument %d: %s \n", i, args[i]);
}
```

g) Jetzt interpretieren wir die Eingabe als Kommando. In `args[0]` steht also der Name eines Programms, und in `args[1]`, `args[2]`, `args[3]` usw. stehen die Argumente. Sie sollen in dieser Teilaufgabe, für die Sie zunächst eine Kopie des Programms (`shell05.c`) erzeugen, die Funktion `execvp()` verwenden. Wenn Sie einen Blick in die Manpage werfen, finden Sie die folgende Beschreibung:

```
int
execvp(const char *file, char *const argv[]);
```

Die Funktion erwartet also zwei Argumente: einen String (`char*`) und ein Array von Strings (`char*[]`). Das erste Argument ist `args[0]`, das zweite einfach `args` – in den vorangehenden Teilaufgaben haben Sie darauf hingearbeitet, dass Ihre Datenstruktur genau zur Syntax von `execvp` passt. Allerdings muss die Liste der Pointer noch mit einem Nullzeiger abgeschlossen werden: Den müssen Sie ergänzen (und darum durften Sie in Teilaufgabe **f** nur neun der zehn Pointer für Argumente verwenden).

Probieren Sie es aus und testen Sie, was passiert, wenn Sie bei der Programmausführung `ls -l` eingeben. Testen Sie auch, was passiert, wenn Sie ein Kommando eingeben, das nicht gefunden werden kann.

h) Obwohl Sie in einer Schleife immer wieder eine neue Eingabe lesen, führt der erste (erfolgreiche!) Aufruf von `execvp()` dazu, dass Ihr Programm beendet wird. Das liegt daran, dass `execvp()` Ihr Programm durch das zu startende Programm ersetzt: Wenn letzteres fertig ist, endet der ganze Prozess. Das können Sie verhindern, indem Sie vorher mit der Funktion `fork()` einen neuen Prozess erzeugen.

Erstellen Sie eine Kopie (`shell06.c`) und ersetzen Sie darin den `execvp()`-Aufruf durch folgenden Block:

```
if ( fork() == 0 )
  // Kindprozess
  execvp ( /* ... Argumente aus Aufgabe g) ... */);
else
  // Vaterprozess
  wait (NULL);
```

Testen Sie das Programm: Jetzt sollten Sie eine funktionierende Mini-Shell haben. Lesen Sie in den Manpages zu `fork()` (`man fork`) und `wait()` (`man waitpid`) nach, was die Kommandos bewirken.