Eine Mini-Shell als Literate Program

Hans-Georg Eßer

16.10.2013

Inhaltsverzeichnis

1 Eine Mini-Shell

Dieses Programm implementiert eine kleine Shell, vergleichbar mit der Bash – nur mit deutlich weniger Features: Die Mini-Shell kennt zwei interne Kommandos `exit` (zum Verlassen der Shell) und `ver` (das die Versionsnummer anzeigt); für alle anderen Befehle versucht die Shell, ein externes Programm zu starten. Parameter werden dabei an das Programm beim Aufruf mit `exec` übergeben.

Das Programm muss diverse Header-Dateien einbinden, die wir jeweils angeben, wenn sie das erste Mal benötigt werden.

1 ⟨* 1⟩≡

```
⟨header file inclusions 2c⟩

int main () {
  ⟨variable declarations 2b⟩
  ⟨implementation 2a⟩
}
```

Defines:
`main`, never used.

Das Hauptprogramm besteht im Wesentlichen aus einer Endlosschleife, in der die Shell immer wieder einen Prompt ausgibt, eine Eingabe akzeptiert und diese dann interpretiert:

2a ⟨*implementation* 2a⟩≡ (1)
```
while (1) {
  printf ("spsh$ ");
  fgets (command, sizeof(command), stdin);
  ⟨interpret command 2d⟩
}
```
Uses command 2b.

In command legt die Funktion `fgets` die Eingabe ab; wir müssen die Variable noch deklarieren:

2b ⟨*variable declarations* 2b⟩≡ (1) 3a▷
```
char command[255];     // speichert aktuelles Kommando
```
Defines:
command, used in chunks 2 and 3c.

Für die Funktionen `printf` und `fgets` sowie die Konstante `stdin` (Standardeingabe) benötigen wir die Header-Datei `stdio.h`:

2c ⟨*header file inclusions* 2c⟩≡ (1) 2e▷
```
#include <stdio.h>      // stdin, fgets(), printf()
```

1.1 Einen Befehl parsen

Um ein Kommando zu interpretieren, schneiden wir von der Eingabe zunächst das letzte Byte (ein Zeilenumbruchzeichen, `\n`) ab:

2d ⟨*interpret* command 2d⟩≡ (2a) 3b▷
```
command[strlen(command)-1] = (char) 0;
```
Uses command 2b.

Damit wir `strlen()` verwenden können, binden wir die Header-Datei `string.h` ein:

2e ⟨*header file inclusions* 2c⟩+≡ (1) ◁2c 4b▷
```
#include <string.h>     // strlen()
```

Jetzt können wir den Befehl in den Befehlsnamen und weitere Argumente zerlegen; dazu verwenden wir die Funktion `strtok`, die ein interessantes Funktionsprinzip hat: Sie merkt sich – über mehrere Aufrufe hinweg –, an welchem String sie gerade arbeitet.

Beim ersten Aufruf muss man `strtok` im ersten Argument einen zu zerlegenden String übergeben und erhält einen Zeiger auf den ersten Teilstring zurück; bei allen weiteren Aufrufen übergibt man im ersten Argument `NULL` und sagt der Funktion damit, dass sie denselben String weiterbearbeiten soll. Gibt die Funktion selbst `NULL` zurück, ist die Zerlegung abgeschlossen.

Für das Zerlegen in mehrere Argumente muss `strtok` erfahren, welche Zeichen als Trennzeichen fungieren; in unserem Fall sind das Leerzeichen und Tabulatoren. Wir deklarieren an dieser Stelle außerdem die Variable `no_args`, welche die Argumente zählt und bei jedem Durchlauf zunächst auf 0 gesetzt werden muss. Den Rückgabewert von `strtok` speichern wir jeweils zunächst in `part` und übertragen ihn dann in das Array `args[]`, das wir auch hier deklarieren:

3a ⟨*variable declarations* 2b⟩+≡ (1) ◁2b 4g▷

```
char  seps[] = " \t";  // Separatoren (Blank und Tabulator)
short no_args;         // Anzahl der Argumente im aktuellen Kmd.
char *args[10];        // Array fuer bis zu 10 Argumente
char *part;            // Zeiger auf den naechsten Teil-String
```

Defines:
`args`, used in chunks 3–5.
`no_args`, used in chunks 3 and 4e.
`part`, used in chunk 3c.
`seps`, used in chunk 3c.

3b ⟨*interpret* `command` 2d⟩+≡ (2a) ◁2d 3c▷

```
no_args = 0;   // initialisieren
```

Uses `no_args` 3a.

Das Zerlegen in Teil-Strings (Befehl und Argumente) gelingt nun mit folgendem Code:

3c ⟨*interpret* `command` 2d⟩+≡ (2a) ◁3b 3d▷

```
part = strtok (command, seps);
while ( part != NULL ) {
  args[no_args] = part;
  no_args++;
  part = strtok (NULL, seps);
};
```

Uses `args` 3a, `command` 2b, `no_args` 3a, `part` 3a, and `seps` 3a.

1.2 Was tun mit dem Kommando?

Jetzt folgt die Verarbeitung des geparsten Befehls. Zunächst fangen wir drei Sonderfälle ab:

- Es wurde gar kein Befehl eingegeben; dann gilt `no_args == 0` und wir können die Verarbeitung abbrechen:

3d ⟨*interpret* `command` 2d⟩+≡ (2a) ◁3c 4a▷

```
if (no_args == 0) continue;
```

Uses `no_args` 3a.

- Es wurde das interne Kommando `exit` eingebeben, dann verlassen wir die Shell über einen Aufruf von `exit()`:

4a ⟨*interpret* `command` 2d⟩+≡ (2a) ◁3d 4c▷

```
if (!strcmp(args[0],"exit")) {
  printf ("Terminating\n");
  exit(0);
}
```

Uses `args` 3a.

Für die `exit`-Funktion benötigen wir die Header-Datei `stdlib.h`:

4b ⟨*header file inclusions* 2c⟩+≡ (1) ◁2e 5a▷

```
#include <stdlib.h>     // exit()
```

- Es wurde das interne Kommando `ver` eingegeben; dann geben wir die Versionsnummer aus springen mit `continue` an den Anfang der Schleife:

4c ⟨*interpret* `command` 2d⟩+≡ (2a) ◁4a 4d▷

```
else if (!strcmp(args[0],"ver")) {
  printf ("spsh 0.1\n");
  continue; // weiter in der Schleife
};
```

Uses `args` 3a.

1.2.1 Externer Programmaufruf

Jetzt bleibt nur noch der (Standard-) Fall, dass ein externes Kommando aufgerufen wird:

4d ⟨*interpret* `command` 2d⟩+≡ (2a) ◁4c

⟨*launch program* 4e⟩

Die Vorgehensweise, um das Programm zu starten, ist die folgende:

1. Argumentliste mit `NULL` terminieren (die `exec`-Funktion braucht das so)

4e ⟨*launch program* 4e⟩≡ (4d) 4f▷

```
args[no_args] = NULL;
```

Uses `args` 3a and `no_args` 3a.

2. mit `fork` einen Kindprozess erzeugen

4f ⟨*launch program* 4e⟩+≡ (4d) ◁4e 5b▷

```
pid = fork();
```

Uses `pid` 4g.

Die Variable `pid` müssen wir noch deklarieren:

4g ⟨*variable declarations* 2b⟩+≡ (1) ◁3a 5c▷

```
int pid;
```

Defines:
`pid`, used in chunks 4f and 5b.

Außerdem brauchen wir für `fork` die Header-Datei `unistd.h`:

5a ⟨*header file inclusions* 2c⟩+≡ (1) ◁4b 5d▷

```
#include <unistd.h>     // fork(), execvp()
```

3. im Kindprozess mit `exec` das externe Programm nachladen (dadurch wird im Kind der Shell-Code durch den Code dieses Programms ersetzt)

4. Im Vater-Prozess mit `waitpid` darauf warten, dass der Kindprozess fertig ist.

5b ⟨*launch program* 4e⟩+≡ (4d) ◁4f

```
if ( pid == 0 ) {     // Kindprozess
  execvp (args[0], args);
  // exec fehlgeschlagen?
  printf ("%s not found\n", args[0]);
  exit(0);
} else {              // Vaterprozess
  waitpid (pid, &status, 0);
};
```

Uses `args` 3a, `pid` 4g, and `status` 5c.

Die Funktion `waitpid` liest auch den Status der terminierten Kindprozesses aus und speichert ihn in der Variable `status`, die wir noch nicht deklariert haben:

5c ⟨*variable declarations* 2b⟩+≡ (1) ◁4g

```
int status;
```

Defines:
`status`, used in chunk 5b.

Um die `waitpid`-Funktion nutzen zu können, müssen wir gleich zwei Header-Dateien einbinden:

5d ⟨*header file inclusions* 2c⟩+≡ (1) ◁5a

```
#include <sys/types.h>  // waitpid()
#include <sys/wait.h>   // auch waitpid()
```

2 Zusammenfassung

Damit sind wir am Ende unseres kleines Literate Programs. Wie man hier gut sehen konnte, ist eine Shell mit den elementarsten Features schnell zusammen gebaut. Aus den Bibliotheken werden dabei nur wenige Funktionen benötigt. Der zentrale Mechanismus steckt im Zerlegen der Eingabe mit `strtok` und dem Programmaufruf mit `fork` und `exec`.

Chunk-Verzeichnis

Identifier-Verzeichnis