Praktikum Mishell

Die Mini–Shell "Mishell"

Frühlingssemester 2020

M. Thaler, J. Zeman

Inhaltsverzeichnis

1 Einführung

1.1 Ziel

In diesem Praktikum möchten wir Sie mit der Funktionsweise der kommandozeilenorientierten Anwenderschnittstelle unter Unix bzw. Linux vertraut machen, der so genannten Shell. *Die Shell ist nichts anderes als ein Programm, das die von Ihnen eingegebenen Befehle analysiert und entsprechende Programme startet*. Diese Programme kommunizieren mit dem Betriebssystem über das System Call Interface. Bei grafischen Benutzeroberflächen, z.B. KDE oder Windows, werden aufgrund von MausClicks auf grafischen Objekten ähnliche Befehle ans Betriebssystem weitergeleitet und auf die gleiche Art und Weise ausgeführt. Wir beschränken uns hier ausschliesslich auf die Kommandozeileneingabe.

In diesem Praktikum werden Sie einen Überblick zu folgen Punkten erhalten:

- wie funktioniert eine Shell
- wie analysiert (parst) die Shell eine Kommandozeile
- wieso und wie wird der Befehl in einem eigenen Prozess ausgeführt
- wie werden Parameter an ein Programm übergeben
- wie können Ein- und Ausgabekanäle auf Dateien umgeleitet werden

1.2 Durchführung und Leistungsnachweis

Es gelten grundsätzlich die Vorgaben Ihres Dozenten zur Durchführung der Praktika und zu den Leistungsnachweisen im Kurs BSy . Die Inhalte des Praktikums gehören zum Prüfungsstoff.

1.3 Aufbau der Praktikumsanleitung

Die Praktikumsanleitung besteht aus zwei Teilen: Abschnitt 2 mit den Grundlagen und der Theorie zur Shell, Abschnitt 3 mit den Aufgabenstellungen.
Studieren Sie bitte den Theorieteil bevor Sie mit der Lösung (Implementation) der Aufgaben beginnen und strukturieren Sie Ihr Programm zuerst auf Papier, z.B. mit einem Struktogramm, einer Flow-Chart, etc.. Zusätzliche Informationen zu den System-Funktionen, etc. finden Sie in den online-Manuals und der angegebenen Literatur ([1]) .

1.4 Praktikumsunterlagen

Die Beispielprogramme finden Sie auf dem WEB-Server zum Kurs Betriebssystem. Laden Sie die Datei MiShell.tar.gz in Ihr Arbeitsverzeichnis und packen Sie die Datei aus. In Ihrem Arbeits-verzeichnis wird der Ordner MiShell angelegt, in dem Sie alle Dateien zum Praktikum finden.

1.5 Literatur

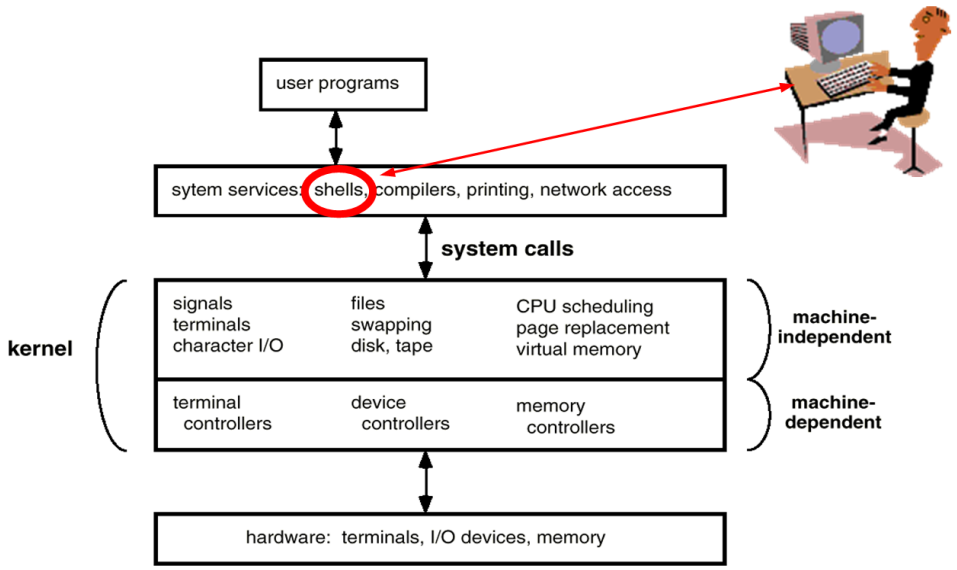
[1] H. Herold, Linux-Unix Systemprogrammierung, 4.Auflage 2004, Addison Wesley.

2 Theorie zur Shell

2.1 Die Rolle der Shell im Betriebssystem

Die Shell ist die Schnittstelle zwischen Anwender und Betriebssystem. Sie erlaubt dem Benutzer interaktiv Befehle über die Tastatur einzugeben, um

- Anwendungen zu starten und zu überwachen,
- kleine Hilfsprogramme auszuführen, mit denen das Betriebssystem konfiguriert, überwacht und gewartet werden kann



Shell Befehle sind entweder Befehle, die von der Shell selbst ausgeführt werden (interne Bfehle), oder Linux Befehle, die durch eigenständige Programme (externe Befehle) realisiert werden und vom Betriebssystem als SSystem Servicesßur Verfügung gestellt werden.

2.2 Funktionsweise der Shell

Um einen Shell Befehl auszuführen sind 4 Schritte notwendig:

1. Befehlszeile einlesen
2. Befehlszeile in einzelne Komponenten (Worte bzw. Token [1]) aufteilen und ev. interpretieren
3. falls notwendig: Ein-/Ausgabekanäle umleiten
4. den Befehl ausführen

2.2.1 Aufbau einer Befehlszeile

Eine typische Kommandozeile unter Linux ist aus folgenden Komponenten (Worten) aufgebaut:

```
BEFEHL  [{-Optionen}]  [< InFile]  [> OutFile]'\n'
```

Felder in eckigen Klammern sind optional, die geschweiften Klammern bedeuten, dass das Feld beliebig oft wiederholt werden kann.

[1] Wort/Token: zusammenhängende Folge von Zeichen

Ein z.B. häufig verwendeter Befehl unter Unix/Linux ist das Auflisten der Dateien im aktuellen Verzeichnis mit Umleitung der Ausgabe in eine Datei:

```
ls -a -l > tmp.txt'\n'
```

Befehl:	`ls`	Dateien im aktuellen Verzeichnis auflisten
Optionen:	`-a`	alle Dateien (auch die, die mit einem "."beginnen)
	`-l`	langes Ausgabeformat
Ausgabe:	`> tmp.txt`	Umleitung der Bildschirmausgabe in die Datei `tmp.txt`
`'\n'`:		signalisiert Ende der Zeile

Wir können annehmen, dass die Worte mit Leerzeichen voneinander abgetrennt sind und die Zeile entweder mit einem `'\n'` (`EOL`: End Of Line) oder einem `EOF` (End Of File[2]) abgeschlossen ist. Befehlszeichen können in unserem Fall maximal 255 Zeichen lang sein.

2.2.2 Befehlszeile einlesen

Für das Einlesen einer Befehlszeile benötigen wir eine Funktion, die Zeichen von der Eingabe liest, bis entweder ein `EOL` oder `EOF` gefunden wurde, oder der Einlesebuffer voll ist. Die Funktion soll `getLine()` heissen und ist wie folgt definiert:

```
int getLine(const char *prompt, char *buf, int size)
```

Parameter:	`char *prompt`	Zeilenprompt
	`char *buf`	Buffer für die Zeichen der Befehlszeile
	`int size`	Länge des Buffers
Rückgabe:	`char *buf`	Buffer enthält Befehlszeile, abgeschlossen mit `'\0'`
	`int`	Anzahl gelesener Zeichen ohne `'\0'` (maximal `size`-1)

Für das Lesen von Zeichen steht die Funktion `getchar()` aus `stdio.h` zur Verfügung.

2.2.3 Aufspaltung der Befehlszeile in einzelne Worte

Die einzelnen Worte (Token) in der Eingabezeile können mit der Systemfunktion `strtok()` ausgelesen werden. Die System-Funktion `strtok()` ist wie folgt definiert (siehe auch online-Manual):

```
char *strtok(char *buf, const char *delims)
```

Parameter:	`char *buf`	Buffer mit der zu analysierenden Befehlszeile
	const char *delims	String mit den Zeichen, die einzelne Worte abgrenzen (z.B. " `$t`" für Leerschlag und Tabulator)
Rückgabe:	char *	Zeiger auf das Wort NULL, wenn das Ende des Strings bzw. Befehlszeile erreicht ist

Achtung: beim ersten Aufruf muss der Buffer als Zeiger angegeben werden, in allen folgenden Aufrufen muss ein`NULL`-Pointer übergeben werden.

Die Funktion `strtok()` gibt Zeiger auf die eingelesenen Worte zurück und schliesst die Worte mit einem `'\0'` ab (wird in `buf` eingefügt), d.h. die Worte bleiben in `buf` gespeichert.

2.2.4 Umleiten der Ein- / Ausgabekanäle

In Unix/Linux ist alles was gelesen oder beschrieben werden kann eine Datei. Das gilt auch für die Ein- und Ausgabe auf dem Terminal. Der Zugriff auf Dateien selbst wird unter Linux generell über so genannte Datei-Deskriptoren geregelt, d.h. jeder Datei wird beim Öffnen ein Datei-Deskriptor

[2] Die Konstante `EOF` ist in `stdio.h` definiert (i.d.R. `EOF = -1`)

zugewiesen (mehr dazu später in er Vorlesung). Schreib- und Lesebefehle können dann über diese Deskriptoren auf die Daten zugreifen.
Beim Starten eines neuen Prozesses werden automatisch die drei Dateien `stdin`, `stdout` und `stderr` geöffnet, dazu werden die Datei-Deskriptoren wie folgt vergeben:

Datei-Deskriptor	Ein-/Ausgabekanal	Kurzbezeichnung
0	Eingabe	`stdin`
1	Ausgabe	`stdout`
2	Fehlermeldungen	`stderr`

Wenn Sie Befehle zum Lesen von der Tastatur (scąnf(), `getchar()`, ...) oder Schreiben auf den Bildschirm (printf(), putchar(), ...) verwenden, dann lesen Sie eigentlich von der Datei mit Deskriptor 0 bzw. schreiben in die Datei mit Deskriptor 1.

Für das Erzeugen von Datei Deskriptoren gibt es den Systemaufruf `open()`. Das folgende Beispiel zeigt, wie die Datei `InData.txt` zum Lesen und die Datei `OutData.txt` zum Schreiben geöffnet werden kann, anschliessend kann mit `read()` und `write()` auf die Dateien zugegriffen werden:

```
int fdr, fdw;   // Deklaration der Datei-Deskriptoren
char buf[100]   // Zeichenbuffer
   ....
fdr = open("InData.txt",  O_RDONLY);
fdw = open("OutData.txt", O_CREAT | O_TRUNC | O_WRONLY, 0644);
read(fdr,  &buf, anz);  // aus der Datei lesen
write(fdw, &buf, anz);  // in die Datei schreiben
```

`O_CREAT`	erstellt die Datei neu, wenn sie nicht existiert
`O_TRUNC`	bewirkt, dass die Datei geleert wird falls sie bereits existiert
`modus`	definiert die Zugriffsrechte, Benutzer kann hier Lesen und Schreiben (6) (siehe auch online-Manuals: ”man 3 read” und ”man 3 write”).

Wie können wir nun die Ein- und Ausgabe umleiten? Unter Unix/Linux müssen wir dazu einfach die entsprechende Datei schliessen (`close()`) und anschliessend die gewünschte Datei öffnen. Die neu geöffnete Datei erhält dabei den Deskritptors der soeben geschlossenen Datei. Beim folgenden Beispiel wird die Standard-Ausgabe auf das File `OutDat.txt` umgeleitet:

```
close(1);
fd = open("OutDat.txt", O_CREAT | O_TRUNC | O_RDWR, 0770);
```

Unter Linux wird einer Datei immer der erste freie Deskriptor mit der kleinsten ”Zahl” zugewiesen.

2.2.5 Ausführen von Shell Befehlen

Wenn die Shell einen externen Befehl ausführen will, muss Sie einen neuen Prozess mit dem entsprechenden Programm starten . Unter Linux und Unix geschieht dies immer in zwei Schritten:

1. Mit dem Systemaufruf `fork()` wird ein neuer Prozess, der Kindprozess, erzeugt, der den Programmcode des aufrufenden Prozesses (Elternprozess) erbt, in unserem Fall den Code der Shell.

2. Der vererbte Programmcode wird anschliessend mit dem Systemaufruf `execv()` durch das gewünschte Programm ersetzt. Mit dem `execv`-Systemaufruf können dem Programm ebenfalls Parameter übergeben werden. Die Parameter werden als Zeiger auf die Strings im Array `argv[]` übergeben, der letzte Eintrag von `argv[]` muss ein `NULL`-Zeiger sein.

Beispielprogramm zur Prozesserzeugung und zum Starten des Befehls (Programmes) "`ls -a`":

```
char *cmd = "/bin/ls";              // command
char *opt = "-l";                   // option
char  *argv[8];                     // pointer to command line arguments
pid_t PID;                          // process identifier

PID = fork();                       // make child process
if (PID == 0) {
   argv[0] = cmd;                   // first  argument: command
   argv[1] = opt;                   // second argument: option
   argv[2] = NULL;                  // last argument:   NULL pointer
   execv(argv[0], &argv[0]);        // execute command
   printf("!!! panic !!!");         // ... should not come here
   exit(-1);                        // terminate child
}
else if (PID < 0) {                 // if for failed
   printf("fork failed\n");
   exit(-1);                        //     fatal -> exit
}
else {
   wait(0);                         // wait for child to terminate
}
```

Das Programm im Kindprozess wird mit `execv()` gestartet. `Execv()` erwartet als Parameter den Pfad des Befehls[3], im Fall von `ls` ist dies "`/bin/ls`" , dann die Argumente , wobei das erste Argument der Befehlsnamen sein muss, hier `ls`, das letzte Argument muss ein `Null`-Zeiger sein.

Alternativ kann das Programm auch mit `execvp()` gestartet. `Execvp()` erwartet als ersten Parameter nur den Namen des Befehls, also `ls`, dann die Argumente wie bei `execv()`. Das Programm `ls` wird von `execvp()` in den Verzeichniseinträgen aus der Pfadvariablen `PATH` gesucht (die Pfade in `PATH` kann man wie folgt anzeigen: `echo $PATH`).

Kann das neue Programm nicht gestartet werden, kehrt `execvp()` ins aufrufende Programm zurück.

Das Umleiten der Ein-/Ausgabekanäle muss in der Shell im Kindprozess nach `fork()` aber, vor dem Start des Programms mit `execv()` ausgeführt werden. Wieso muss das so gemacht werden?

3 Aufgaben

3.1 Aufgabe 1: `getLine()`

Im Verzeichnis `SiShell` haben wir eine einfache Shell vorbereitet (`siShell.c`), die einen Befehl ohne Parameter einlesen kann. Die Funktion `getLine()` zum Einlesen der Befehle haben wir nur definiert aber nicht implementiert. Implementieren und testen sie `getLine()` nach den Spezifikationen aus Abschnitt 2.2.2. zusammen mit `siShell.c`.

3.2 Aufgabe 2: Die SiShell

Erweitern sie nun `siShell.c` so, dass sie Befehlszeilen mit externen Befehlen und bis zu 15 Argumenten eingeben und verarbeiten können (total 16 Tokens): z.B. `ls`, `ls -al`, `ls -a -l`.

Geben sie in der erweiterten Shell den Befehl `find` ein. Was geschieht? Falls es nicht funktioniert lösen sie das Problem mit dem Vorschlag aus Abschnitt 2.2.5.

[3] der Pfad eines Befehls lässt sich mit "`which Befehl`" anzeigen

3.3 Aufgabe 3: Die MiShell

Kopieren sie alle Files aus dem Verzeichnis `SiShell` ins Verzeichnis `MiShell` und arbeiten sie nun mit den kopierten Files.

Erweitern Sie `siShell.c` so, dass sie

1. Befehle mit "redirection" ausführen kann: beide Richtungen, zum Beispiel: `ls -al > dlist.txt`, bzw. `grep < getLine.c getchar > tmp.txt`
2. interne Befehle wie `logout`, `exit` und `cd` (change directory) ausführen kann

Für die Umleitung der Ein-/Ausgabeumleitung müssen sie die Funktionen `tokenizeCommand()`, `executeCommand()` und `externalCommand()` um den Parameter `char *redir[]` erweitern (ein `char *ptr` Array mit zwei Einträgen), in dem Referenzen auf die Namen der Umleitung-Files übergeben werden, falls keine Umleitung benötigt wird, muss ein `NULL`-Pointer übergeben werden.

Die Shell-Befehle `logout`,`exit` und `cd` sind keine Linux/Unix-Programme wie z.B. `ls`, sondern müssen direkt als interne Befehle ihrer Shell implementiert werden, wobei `logout` und `exit` die Shell beenden. Der Wechsel des Arbeitsverzeichnisses mit `cd` lässt sich mit der Systemfunktion `chdir(path)` in C realisieren. Falls kein Pfad angegeben wird, wechselt `cd` ins Heimverzeichnis: der Pfad des Heimverzeichnisses kann mit `getenv("HOME")` abgefragt werden.

Dazu müssen sie die Funktion `int internalCommand()` implementieren, die die internen Befehle erkennt und dann ausführt: Rückgabewert '0', falls kein interner Befehl vorliegt, andernfalls wird eine '1' zurückgegeben. Input/Output-Umleitung muss für interne Befehle nicht implementiert werden, sie können das aber optional tun.

Strings in C können sie mit der Funktion `strcmp()` vergleichen: falls gleich wird eine '0' zurückgegeben:

```
if (strcmp(argv[0], "logout") == 0)     // if command == logout
     exit(0);                           //     terminate shell
```

Eine detaillierte Beschreibung zu `strcmp()` finden Sie im online-Manual.

3.4 Aufgabe 4: Die "readline"-Bibliothek

Wie Sie festgestellt haben ist die Funktionalität der Kommandozeile in der Mishell etwas spartanisch. Dies lässt sich sehr einfach verbessern, indem die GNU readline-Bibliothek verwendet wird. Gehen Sie dazu wie folgt vor:

- installieren Sie die Bibliothek **`libreadline-dev`**
- kopieren Sie den Inhalt des Verzeichnisses `MiShell` ins Verzeichnis `MiShellMaBelle`
- ergänzen Sie im `makefile` den Eintrag `LIB = -lreadline`
- ersetzen Sie den Code in `getLine()` indem Sie `readline()` verwenden

Mit **`man readline`** finden Sie Informationen zur Funktion `readline()` und welche Header-Files eingebunden werden müssen.

Hinweis: achten Sie darauf, dass Sie kein Memory Leak erzeugen.

Wenn Sie zudem in `getLine()` die Funktion **`add_history(buf)`** aufrufen, lassen sich "alte" Befehle erneut abrufen: **`buf`** enthält dabei die soeben eingelesene Zeile. Mit folgender Code-Zeile können Sie zudem die Anzahl Einträge in der History auf z.B. 100 beschränken:

```
if (!history_is_stifled())  stifle_history(100);
```

Mit **`man history`** finden Sie Informationen zu den History-Funktionen.