Praktikum MiShell

Die MiniShell "MiShell"

Frühlingssemester 2016
M. Thaler, J. Zeman

Inhalt

1. Einführung

1.1 Ziel

In diesem Praktikum möchten wir Sie mit der Funktionsweise der kommandozeilenorientierten Anwenderschnittstelle unter Unix resp. 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 Maus-Clicks 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 Arbeitsverzeichnis wird der Ordner `MiShell` angelegt, in dem Sie alle Dateien zum Praktikum finden.

1.5 Literatur

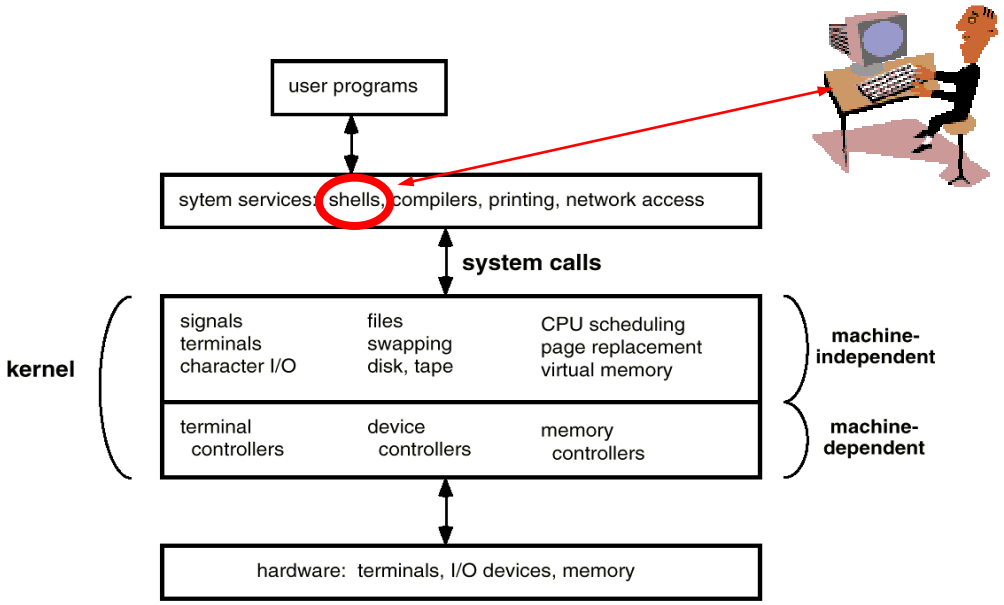
[1] H. Herold, *Linux-Unix Systemprogrammierung*, 4.Auflage 2004, Addison Wesley.

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, oder Linux Befehle, die durch eigenständige Programme realisiert werden und vom Betriebssystem als "System Services" zur 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[1]) aufteilen und ev. zu 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: eine 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[2] (End Of File) abgeschlossen ist. Befehlszeilen können gemäss POSIX Standard aus maximal 255 Zeichen bestehen (Grösse des Eingabepuffers).

2.2.2 Befehlszeile einlesen

Als erstes wollen wir eine ganze Befehlszeile einlesen. Dazu benötigen wir eine Funktion, die solange Zeichen von der Eingabe liest, bis entweder ein EOL oder EOF gefunden wurde, oder der Einlesebuffer voll ist. Die Funktion soll `readline()` heissen und ist wie folgt definiert:

```
int readline(char *buf, int size)
```

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

Für das Lesen der einzelnen Zeichen von der Eingabezeile steht die Funktion `getchar()` zur Verfügung, die in `stdio.h` definiert ist.

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 das 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 die einzelnen Worte abgrenzen (z.B. `" \t"` für Leerschlag und Tabulator)
Rückgabe:	`char *`	Zeiger auf das Wort `NULL`, wenn das Ende des Strings resp. 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] Die Konstante `EOF` ist in `stdio.h` als -1 definiert.

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 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 (`scanf()`, `getchar()`, ...) oder Schreiben auf den Bildschirm (`printf()`, `putchar()`, ...) verwenden, dann lesen Sie eigentlich von der Datei mit Deskriptor 0 resp. 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` zu 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_WRONLY, modus);
```

Unter Linux erhält eine Datei immer den ersten freien Deskriptor mit der kleinsten "Zahl".

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[3]. 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.

[3] Programme werden auf Betriebssystemen immer im Kontext von Prozessen ausgeführt

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 eigentlichen 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`":

```
pid_t   PID;                        // Deklaration Prozessidentifikator
char    args[8][80];                // 8 Strings zu 80 Zeichen für Parameter
char    *argv[8];                   // Zeiger auf die 8 Strings mit den Parametern
char    path[80];                   // String fuer den Pfadnamen
....
PID = fork();                       // Prozess erzeugen
if (PID == 0) {                     // wir sind im Kindprozess, wenn PID==0 ist
       strcpy(args[0]), "ls");      // Argument 0 immer = Name des Programms
       strcpy(args[1]), "-a");      // Argument 1
       argv[0] = args[0];           // Zeiger von args[0] speichern
       argv[1] = args[1];           // Zeiger von args[1] speichern
       argv[2] = NULL;              // letztes Argument = NULL-Zeiger
       strcpy(path, "/bin/ls");     // Pfad fuer Befehl setzen
       execv(path, &argv[0]);       // neues Programm starten
       printf("!!! panic !!!\n");// exec hat nicht geklappt
       exit(-1);                    // Prozess terminieren
}
else if (PID < 0) {                 // falls Fehler bei fork() -> terminieren
       printf("fork failed\n");     // Fehlermeldung
       exit(-1);                    // (Eltern)Prozess terminieren
}
else                                // hier befinden wir uns im Elternprozess
       wait(NULL);                  // der Elternprozess wartet, bis der
                                    // Kindprozess terminiert.
....                                // ... und arbeitet weiter
exit(0);                            // (Eltern)Prozess terminieren
```

`fork()` liefert drei verschieden Rückgabewerte (`pid_t` ist ein integer-Typ):

`0`	wenn wir uns im neu erzeugten Prozess (Kindprozess) befinden
`> 0`	wenn wir uns im Elternprozess befinden
`< 0`	wenn bei der Prozesserzeugung ein Fehler aufgetreten ist

`fork()` erstellt zuerst eine Kopie des Elternprozesses, dann fährt sowohl der Eltern- als auch der Kindprozess mit der nächsten Instruktion nach `fork()` weiter (**`if (PID == 0)`**).

Das Programm im Kindprozess wird mit mit `execv()` gestartet. `Execv()` erwartet als Parameter den **Pfad** des Befehls, im Fall von `ls` ist dies "`/bin/ls`"[4], dann die Argumente[5], wobei das erste Argument der Befehlsnamen sein muss, hier `ls`, das letzte Argument muss ein Null-Zeiger sein. Beachten Sie in obigem Beispiel, wie die konstanten Strings mit `strcpy()` in die entsprechenden Variablen kopiert werden.

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 `exec()`. 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 und fährt bei der nächsten Instruktion weiter, hier bei `printf()`.

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?

[4] der Pfad eines Befehls lässt sich mit "`which Befehl`" bestimmen (z.B. `which ls`)
[5] die Argumente werden als Zeiger auf die entsprechenden Strings übergeben

3. Aufgaben

3.1 Aufgabe 1: readline()

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

3.2 Aufgabe 2: Die SiShell

Erweitern Sie die Shell im Verzeichnis `SiShell` so, dass die Befehlszeile aus bis zu 16 Argumente (Worte) betstehen kann, z.B.: `ls`, `ls -al, ls -a -l`. Hinweis: das File `siShell.c` ist identisch mit `simpleShell.c`.

Geben Sie in Ihrer Shell den Befehl `find` ein. Was geschieht? Lösen Sie das Problem mit dem Vorschlag aus Abschnitt 2.2.5 (unten).

3.3 Aufgabe 3: Die MiShell

Erweitern Sie die SiShell so, dass sie

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

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 müssen, wobei **`logout`** und **`exit`** die Shell beenden. Der Wechsel des Arbeitsverzeichnisses mit **`cd`** lässt sich mit der Systemfunktion **`chdir(path)`** in C realisieren (um die Implementation nicht unnötig zu komplizieren, soll zusammen mit **`cd`** immer ein Pfadname angegeben werden).

Ob einer dieser Befehle eingegeben wurde, können Sie mit der Funktion **`strcmp()`** feststellen, bei übereinstimmenden Strings wird eine 0 zurückgegeben. Wenn zum Beispiel **`args[0]`** den Befehl **`logout`** enthält:

```
if (strcmp(args[0], "logout") == 0) .. terminate .. ;
```

Für die Stringverarbeitung verwenden wir die Funktionen **`strcpy()`**, **`strcat()`** und **`strcmp()`**, eine detaillierte Beschreibung finden Sie im online-Manual.

Hier zur Hilfestellung Pseudocode für die Struktur der MiShell:

```
while(1) {
    Prompt ausgeben
    Befehlszeile einlesen
    Worte der Befehlszeile speichern, Pfade für die Redirection in Hilfsvariablen speichern
    falls 1. Wort "logout" oder "exit" -> MiShell beenden
    falls 1. Wort "cd" -> chdir() ausführen
    Sonst neuen Prozess starten (fork())
        Kindprozess
            falls notwendig standard input und output umleiten
            Befehl mit execvp() ausführen
        Elternprozess
            auf Terminierung des Kindprozsses warten
}
```