Prof. Dr. M. Jäger
FB MNI

12. November 2018

Hausübung - Betriebssysteme - WS2018/19

1 Aufgabenstellung

Programmieren Sie in C eine Shell, die einfache und komplexe Kommandos verschiedener Typen interpretiert.

Die Hausübung ist in drei Teilaufgaben aufgeteilt:

1. *basicsh*

   Programmieren Sie in C eine Shell (*basicsh*), die einfache und komplexe Kommandos verschiedener Typen interpretiert, z.B. bedingte Ausführung oder Pipelines.

2. *advancedsh*

   Erweitern Sie *basicsh* um eine Prozessliste. Die erweiterte Shell (*advancedsh*) soll in der Liste verschiedene Informationen zu ihren Kindprozessen speichern und mit dem *status*-Kommando anzeigen können. Bei Subprozessterminierung soll ein Signalhandler für SIGCHLD aktiviert werden, der in der Prozessliste den Terminierungsstatus einträgt.

3. *jobcrtlsh*

   Erweitern Sie *advancedsh* um Job-Control-Funktionen. Dazu gehören z.B. Abbrechen von Jobs, Anhalten von Jobs, Fortsetzen im Vodergrund (fg) oder Hintergrund (bg).

1.1 Zeitplan

Die Teilaufgaben *basicsh* und *advancedsh* werden nicht separat abgegeben. Die nachfolgenden Zeiten dienen der Orientierung:

Teilaufgabe	**Fertigstellung**
basicsh	29. November 2018
advancedsh	13. Dezember 2018
jobcrtlsh	17. Januar 2018

Hinweise zur Abgabe der Hausübung finden sie in Abschnitt 2.1, Seite 5.

1.2 Shell mit Basisfunktionen: *basicsh*

1.2.1 Kommandotypen und Syntax

1. **Programmaufruf im Vordergrund**, Syntax: *Programmpfad Parameter ...*

   Aufruf eines Programms als Subprozess der Shell. Die Shell wartet auf die Terminierung.

2. **Programmaufruf im Hintergrund**, Syntax: *Programmpfad Parameter ...* &

   Aufruf eines Programms als Subprozess der Shell. Die Shell wartet *nicht* auf die Terminierung. Die Shell schreibt die PID und PGID des neuen Prozesses auf die Standardfehlerausgabe.

3. **Terminieren der Shell**

   Syntax: `exit [` *Exit-Wert* `]`

   Wenn kein Exit-Wert angegeben ist, soll die Shell mit *exit(0)* terminieren.

4. **Verzeichnis wechseln**

   Syntax: `cd [` *Dateipfad* `]`

   Die Shell ändert ihr aktuelles Verzeichnis. Bei fehlendem Pfad wird das Login-Verzeichnis des Benutzers verwendet. (Systemaufruf *chdir*)

5. **Umlenkungen der Ein- und/oder Ausgabe**

   - Ausgabeumlenkung, Syntax: *Programmaufruf* $>$ *Dateipfad*
     Die Standardausgabe des Programms wird in die angegebene Datei umgelenkt. Diese wird bei Bedarf erzeugt. Falls sie schon vorhanden ist, wird der alte Inhalt gelöscht.
   - Ausgabeumlenkung mit Anfügen, Syntax: *Programmaufruf* $>>$ *Dateipfad*
     Falls die Datei schon vorhanden ist, wird der neue Inhalt hinter den alten geschrieben.
   - Eingabeumlenkung, Syntax: *Programmaufruf* $<$ *Dateipfad*
     Die Datei wird zur Standardeingabe des Programms.

6. **Sequenz**, Syntax: *Programmaufruf* `;` *Programmaufruf* `;` ... `;` *Programmaufruf*

   Die Programme werden in Shell-Subprozessen nacheinander ausgeführt.

7. **Ausführung bei Erfolg**,

   Syntax: *Programmaufruf* && *Programmaufruf* && ...&& *Programmaufruf*

   Wie bei Sequenz, aber Abbruch, falls ein Programm nicht mit `exit(0)` terminiert.

8. **Ausführung bei Misserfolg**

   Syntax: *Programmaufruf* || *Programmaufruf* || ...|| *Programmaufruf*

   Wie bei Sequenz, aber Abbruch, falls ein Programm mit `exit(0)` terminiert.

9. **Pipeline**

   Syntax: *Programmaufruf* | *Programmaufruf* | ...| *Programmaufruf*

   Alle Programme werden in Shell-Subprozessen nebenläufig ausgeführt. Die Standardausgabe eines Pipeline-Teilnehmers wird zur zur Standardeingabe des nächsten.

10. **Bedingte Ausführung**

    Syntax:

    `if` *Programmaufruf* `;` `then` *Programmaufruf* `;` `else` *Programmaufruf* `;` `fi`
    `if` *Programmaufruf* `;` `then` *Programmaufruf* `;` `fi`

    (Wenn `then`, `else` oder `fi` auf einer neue Zeile stehen, kann das Semikolon davor entfallen.)

1.3 *Shell mit Prozessliste: advancedsh*

1.3.1 Prozessliste und Prozesszustände

Die Shell verwaltet eine Liste der Subprozesse. Das Kommando *status* gibt zu jedem Subprozess aus: PID, PGID, Zustand und Programmname. Als Zustand wird für noch nicht terminierte Prozesse „running“, für terminierte Prozesse entweder „exit(*n*)“ oder „signal(*s*)“ ausgegeben, je nachdem, ob der Prozess durch Signal oder exit-Aufruf terminiert wurde. Dabei ist *n* der Exit-Wert und *s* die Signalnummer. In der *jobcrtlsh* kommt für angehaltene Prozesse der Zustand *stopped* hinzu.

Terminierte Prozesse werden aus der Liste nach der Status-Ausgabe gelöscht.

Beispiel für die Verwendung von *status:*

```
>> pwd
/home/jaeger
>> /opt/firefox/bin/firefox &
PID=1454 PGID=1454
>> xterm &
PID=1455 PGID=1455
>> kill -15 1455
>> ls -l /xyz
Datei nicht gefunden
>> status
PID      PGID   STATUS       PROG
1412     1412   exit(0)      pwd
1454     1454   running      /opt/firefox/bin/firefox
1455     1455   signal(15)   xterm
1461     1461   exit(0)      kill
1464     1464   exit(1)      ls
>> status
PID      STATUS       PROG
1454     1454   running      /opt/firefox/bin/firefox
>>
```

1.3.2 Zombies, SIGCHLD

Damit das Betriebssystem nicht unnötig viele Zombies aufbewahrt, muss die Shell einen Signalhandler für das SIGCHLD-Signal haben, der den Status terminierter Prozesse in der Prozessliste aktualisiert.

1.3.3 Implementierungshinweis

Sie können die Prozessliste durch ein Array oder eine verkettete Liste implementieren. Beim *status*-Kommando muss diese Liste aktualisiert werden. Dazu kann man zu jedem Prozess in der Prozessliste einen nicht blockierenden waitpid-Aufruf verwenden. Es bietet sich an, den Status so abzuspeichern, wie er vom Systemkern geliefert wird und bei der Ausgabe der Liste dann daraus eine lesbare Statusinformation zu generieren.

1.4 Shell mit Job-Control-Funktionen: *jobcrtlsh*

1.4.1 Prozessgruppen (Jobs), Vordergrund- und Hintergrundausführung

Ein *Job* entspricht einem Kommando. Im einfachsten Fall ist ein Job ein Prozess, oft sind aber mehrere Prozesse beteiligt. Das Betriebssystem unterstützt dies mit Prozessgruppen.

Jede interaktive Sitzung hat ein Kontrollterminal (*/dev/tty*), das aus einer Tastatur und einer Ausgabekonsole besteht. Dies können reale oder virtuelle Geräte sein. Dem „Vordergrund"-Job ist die Tastatur des Kontrollterminals zugeordnet. Daneben kann es beliebige Hintergrund-Jobs geben. Der Job-Control-Mechnismus erlaubt dem Benutzers, mit verschiedenen Signalen und Kommandos Programme anzuhalten und im Vorder- oder Hintergrund fortzusetzen:

- Abbruch des Vordergrund-Jobs mit STRG-C
- Anhalten des Vordergrund-Jobs mit STRG-Z
- Fortsetzen eines angehaltenen Jobs mit *fg* oder *bg*

- Fortsetzung eines angehaltenen Jobs im Vordergrund

  Syntax: `fg` *`PGID`*

  Der Prozessgruppe wird dazu mit *kill* das Fortsetzungs-Signal SIGCONT geschickt.

- Fortsetzung eines angehaltenen Jobs im Hintergrund

  Syntax: `bg` *`PGID`*

Damit die richtigen Prozesse die Signale erhalten, müssen Prozessgruppen verwaltet werden: Für jeden Programmaufruf wird eine neue Prozessgruppe erzeugt (`setpgid`). Die Prozessgruppen-ID (PGID) ist identisch mit der PID des Prozesses. Damit die richtigen Prozesse abgebrochen bzw. unterbrochen werden, muss die Shell dem Systemkern eine Änderung der Vordergrundprozessgruppe mitteilen (`tcsetpgrp`). Nach Ende eines Vordergrund-Kommandos muss sich die Shell selbst wieder in den Vordergrund bringen.

Eine Pipeline wird als ein Kommando, d.h. als eine Gruppe implementiert: Die PID des ersten Pipeline-Teilnehmers ist die PGID aller Pipeline-Teilnehmer. Mit STRG-Z wird also die gesamte Pipeline angehalten.

Wenn ein Programm aus dem Hintergrund von der Tastatur lesen will, wird es ebenfalls angehalten (SIGTTIN). Schreiben aus dem Hintergrund auf das Kontrollterminal ist normalerweise erlaubt. Wenn man es verbietet, z.B. mit dem Kommando „stty tostop", wird ein Hintergrundprozess beim Schreibversuch mit SIGTTOU angehalten.

Unabhängig davon, mit welchem Signal ein Prozess angehalten wurde, erfolgt die Weiterführung mit SIGCONT.

2 Sonstige Hinweise

1. Unter dem URL (https://m-jaeger.de/lv/bs/aufgaben/ha1/shell-src.tar.gz) gibt es ein als Basis verwendbare Shell-Skelett, in dem schon das komplette Front-End, d.h. Syntaxanalyse und Zerlegung der Kommandozeile, sowie einfache Kommandos und Sequenzen realisiert sind. Sie müssen dann noch:
   (a) die interne Darstellung der Kommandos in „kommandos.h“ und „listen.h“ verstehen lernen,
   (b) die Struktur des Interpretierers „interpretiere.c“ studieren,
   (c) die noch nicht vorhandenen Kommandos in „interpretiere.c“ ergänzen,
   (d) die Signal-Handler in „shell.c“ ergänzen.
2. Fehlgeschlagene Systemaufrufe müssen erkannt und geeignet behandelt werden.
3. Vor Anzeige der Prozessliste muss diese durch Abruf der aktuellen Prozesszustände aller Subprozesse aktualisiert werden.
4. Testen Sie die Shell sorgfältig. Testen Sie bei der Pipeline vor allem auch die Szenarien „Terminierung des Lesers bei blockiertem Schreiber“, z.B.

   ```
   $ od -x /bin/bash | head -1
   ```

   und „Terminierung des Schreibers bei blockiertem Leser“, z.B.

   ```
   $ echo hallo | cat
   ```

2.1 Abgabe der Lösung

Die Hausübung muss in Einzelarbeit oder in Zweiergruppen bearbeitet und abgegeben werden. Die Lösungen müssen in den Übungsgruppen präsentiert und abgenommen werden. Zusätzlich müssen die Lösungen per E-Mail wie folgt abgegeben werden:

- Sie erzeugen ein Verzeichnis, dessen Name mit der Matrikelnummer des (bzw. eines) Bearbeiters übereinstimmt. Dort gibt es ein Unterverzeichnis „src“ und eine Datei „team.txt“. In „team.txt“ stehen die Autoren mit folgenden Angaben: Nachname, Vorname(n), Matrikelnummer.
- Im Verzeichnis „src“ stehen alle Quelltexte und ein „Makefile“ zur Erzeugung der Shell. Das Verzeichnis wird in eine komprimierte tar-Archivdatei (*Matrikelnummer*`.tgz`) gepackt.
- Beispiel:

  Hansi Hacker und Gundula Guru bearbeiten die Aufgabe als Zweiergruppe. Hansi hat die Matrikelnummer 123456, Gundula die Matrikelnummer 765432. Hansi bietet an, die Aufgabe unter seiner Matrikelnummer abzugeben.

  – Zuerst wird das Verzeichnis für die Abgabe und das Unterverzeichnis *src* erstellt:

    ```
    $ mkdir 123456
    $ mkdir 123456/src
    ```

  – Der Inhalt von `123456/team.txt`:

    ```
    Guru, Gundula, 765432
    Hacker, Hansi, 123456
    ```

  - Inhalt von „123456/src“:

    ```
    frontend.h  interpretiere.c  kommandos.c  kommandos.h  listen.c  listen.h
    Makefile  parser.y  scanner.l  shell.c  utils.c  utils.h  variablen.h
    wortspeicher.c  wortspeicher.h
    ```

- Archiv erzeugen:

  ```
  tar cvfz 123456.tgz 123456
  ```

- Per E-Mail senden an: *michael.jaeger@mni.thm.de*

  Betreff: `BS-HA1`
  Inhalt der Mail: leer
  Anhang: `123456.tgz`

Anhang: Shell Front-End und Interpreter-Schnittstelle

Aufbau der Shell

Die Shell ist eine Konsolenapplikation, die in einer Endlosschleife von der Standardeingabe Kommandos einliest und ausführt. Die eingelesenen Kommandos werden von der Parser-Komponente analysiert und auf Fehler geprüft. Der Parser baut aus jedem Kommando eine interne Darstellung („abstrakter Syntaxbaum“) auf. Diese interne Kommandorepäsentation wird dann dem Interpreter übergeben, der das Kommando ausführt.

Front-End

Als Frond-End wird der Teil der Shell bezeichnet, der die Kommandos liest und die interne Repräsentation aufbaut. Die Einleseschleife steht in der Datei `shell.c`. Die Syntaxanalyse für die Kommandos besteht aus zwei Unterkomponenten:

- Der Scanner erkennt lexikalische Tokens wie Strings, Schlüsselwörter, Operatoren, Trennzeichen usw. Er übermittelt dem Parser bei Aufruf die Information über das nächste Token innerhalb des eingelesenen Kommandos. Der Scanner wird aus einer formalen Spezifikation der Tokensyntax durch reguläre Ausdrücke (`scanner.l`) mit dem Scannergenerator *flex* generiert.
- Der Parser erkennt die verschiedenen Kommandotypen wie einfache Kommandos oder Pipelines. Er verarbeitet die Eingabe nicht zeichenweise, sondern benutzt den Scanner als Hilfsfunktion. Er baut nach der Analyse des Kommanods die passende interne Darstellung auf. Drr Parser wird aus einer formalen Spezifikation der Kommandosyntax (`parser.y`) mit dem Parsergenerator *bison* generiert. Diese formale Spezifikation besteht aus einer kontextfreien Grammatik, deren Ableitungsregeln die Kommandostruktur beschreiben, und semantischen Aktionen (C-Code) zum Aufbau der internen Darstellung.

Das Front-End wird im Rahmen der Hausübung vorgegeben und muss nicht erweitert oder modifiziert werden.

Interne Kommandorepräsentation

Die interne Repräsentation ist die Ausgangsbasis für die Ausführung durch den Interpreter. Die zugrunde liegende Datenstruktur ist in `kommandos.h` definiert. Es handelt sich dabei um einen abstrakten Syntaxbaum, in dessen Wurzelknoten (`struct kommando`) der Kommandotyp und ein Flag für Vorder- oder Hintergrundaufruf stehen. Der Kern der Kommandodarstellung wird durch eine Union-Komponente *u* repräsentiert, die je nach Kommandotyp ein *EinfachKommando* oder ein *Sequenz-Kommando* ist.

Einfache Kommandos

Ein einfaches Kommando kann ein Shell-internes Kommando wie `cd` oder einen Programmaufruf wie ”`ls -l /tmp`” repräsentieren. Die interne Repräsentation besteht aus einer Liste von Wörtern (*worte*) mit Längenangabe (*laenge*) und einer Liste von Ein- Ausgabeumlenkungen.

Beispiel: Betrachten Sie das Shell-Kommando

```
wc -l > wc.out < wc.in
```

Sei *k* ein Zeiger auf die interne Repräsentation.

```
k->typ = K_EINFACH
k->endeabwarten = 1
k->u.einfach.wortanzahl = 2
k->u.einfach.worte[0] = "wc"
k->u.einfach.worte[1] = "-l"
```

Sei `uml = k->u.einfach.umlenkungen` die Liste der Umlenkungen für das Beispielkommando. Die Liste ist zweielementig. Das erste Listenelement

```
Umlenkung u1 = listeKopf(uml)
```

repräsentiert die Ausgabeumlenkung:

```
u1.filedeskriptor = 1
u1.modus = WRITE
u1.pfad = "wc.out"
```

Das zweite Element der Umlenkungsliste

```
Umlenkung u2 = listeKopf(listeRest(uml))
```

die Eingabeumlenkung:

```
u2.filedeskriptor = 0
u2.modus = READ
u2.pfad = "wc.in"
```

Die in `listen.h` definierten einfach verketteten Listen werden für die Umlenkungen und auch für die Unterkommandos komplexer Kommandos verwendet.

Komplexe Kommandos

Ein komplexes Kommando enthält Unterkommandos und wird immer unter Verwendung einer Liste dieser Unterkommandos repräsentiert. Ob diese Liste eine Pipeline, eine Sequenz oder eine andere Form repräsentiert ist ausschließlich am Kommandotyp zu erkennen. Die Listenelemente sind in der selben Reihenfolge wie die Unterkommandos in der Kommandozeile angeordnet.

Beispiel:

Die interne Repräsentation von

```
if <Programmaufruf 1>; then <Programmaufruf 2>; else <Programmaufruf 3>; fi
```

besteht aus dem Typ `K_IF` und einer Liste mit den internen Repräsentationen der drei Unterkommandos.

Die interne Repräsentation von

```
<Programmaufruf 1> | <Programmaufruf 2> | <Programmaufruf 3>
```

besteht aus dem Typ `K_PIPE` und einer Liste mit den internen Repräsentationen der drei Unterkommandos.

IF-Anweisungen ohne ELSE werden intern als "`&&`" -Kommandos dargestellt, denn

```
if <Programmaufruf 1>; then <Programmaufruf 2>; fi
```

ist äquivalent zu

```
<Programmaufruf 1> && <Programmaufruf 2>
```

Anhang: Testfälle für die Shell

1. Status-Kommando

   Starten Sie 4 Prozesse im Hintergrund und prüfen Sie, ob „status“ die Zustände korrekt anzeigt:

   ```
   ls -l xyzxyz &
   xterm &
   xterm &
   ps &
   ```

   Der `ls`-Befehl terminiert mit exit(2), wenn Die Datei „xyzxyz“ nicht existiert. Der `ps`-Befehl terminiert mit exit(0). Beide `xterm`-Prozesse sollten im Zustand „Running“ sein. Beenden Sie einen der `xterm`-Prozesse mit „kill -9“ und prüfen Sie erneut mit „status“, ob die Zustände richtig angezeigt werden.

2. Umlenkung der Ein- und Ausgabe

   Prüfen Sie die Umlenkungen einzeln und in Kombination:

   ```
   >> echo hallo > f
   >> cat f
   hallo
   >> echo hallo >>f
   >> cat f
   hallo
   hallo
   >> cat <f
   hallo
   hallo
   >> cat <f >>f1
   >> cat >>f1 <f
   >> cat f1
   hallo
   hallo
   hallo
   hallo
   ```

   Prüfen Sie die Fehlerbehandlung bei Umlenkungen:

   ```
   >> cat < yyyy
   No such file or directory
   Fehler beim Öffnen der Datei: yyyy
   >> touch outfile
   >> chmod 000 outfile
   >> ls >>outfile
   Permission denied
   Fehler beim Öffnen der Datei: outfile
   ```

3. Pipelines

   Prüfen Sie Pipelines mit folgenden Tests:

   1) unproblematische Pipeline

      ```
      cat|cat|cat
      ```

2) Wartet die Shell auf *alle* Pipeline-Teilnehmer?

   `xterm|cat`

   Lassen Sie sich im xterm-Fenster mit „ps -u“ die Prozessnummern anzeigen und beenden Sie den cat-Prozess mit *kill*. Wartet die Shell bis zur Terminierung von `xterm`?

3) Gibt es eine Verklemmung, wenn ein Schreiber wegen einer vollen Pipe blockiert und der letzte Pipeline-Teilnehmer die Pipe nicht vollständig liest?

   ```
   cat /bin/bash | od -x | head -1
   ```

   Bei Terminierung von `head` sollte `od` ein SIGPIPE erhalten und dadurch terminieren. Dieses wiederum erzeugt ein weiteres SIGPIPE für `cat`, das dann ebenfalls terminiert. SIGPIPE wird dem Pipe-Schreiber aber dann nicht geschickt, wenn noch ein weiterer Leser existiert, d.h. wenn der Elternprozess oder ein anderer Pipeline-Teilnehmer unnötigerweise einen Lesedeskriptor offen hat.

4. Job-Control

   1) Prüfen Sie bei einzelnen Programmen und bei Pipelines, ob das Abbrechen mit STRG-C und das Unterbrechen mit STRG-Z funktioniert:

      ```
      >> cat
      ^Z
      >> status
      PID      PGID    STATUS   INFO   PROGRAMM
        7835    7835   stopped   20    cat
      >> cat|cat|cat
      abc
      abc
      ^Z
      >> status
      PID      PGID    STATUS   INFO   PROGRAMM
        7872    7870   stopped    20     cat
        7871    7870   stopped    20     cat
        7870    7870   stopped    20     cat
      ```

   2) Prüfen Sie bei einzelnen Programmen und bei Pipelines, ob die Fortsetzung im Vordergrund und Hintergrund funktioniert.

   3) Prüfen Sie, ob sich ein Texteditor wie `vi` im Hintergrund starten lässt. Wenn ja, gibt es ein Problem:

      `vi` setzt ein Terminalattribut namens TOSTOP (Stoppen bei versuchtem Terminal-Output), welches dazu führt, dass jeder Hintergrundprozess beim Terminal-Schreibzugriff ein SIGTTOU bekommt. Damit erreicht er zweierlei: a) Ein anderer Hintergrundprozess zerschiesst sein Editorfenster nicht b) Wenn er selbst im Hintergrund ist, wird er beim Fensteraufbau mit SIGTTOU gestoppt.

      Die Kindprozesse der Shell, speziell Programme wie `vi`, dürfen daher SIGTTOU nicht ignorieren. Dazu müssen sie vor dem exec die Behandlung wieder von SIG_IGN auf SIG_DFL setzen. SIG_IGN wird beim exec sonst an das neue Programm vererbt.