Prof. Dr. M. Jäger
FB MNI

8. November 2016

Hausübung 1 - Betriebssysteme I - WS2015/16

Bearbeitungszeit: bis 13.12.2016 24 Uhr

Präsentation der Lösung: in den Übungsstunden UND per Upload (siehe unten)

1 Aufgabenstellung

Programmieren Sie in C eine Shell, die verschiedene Kommandos verschiedener Typen interpretiert. Kommandos sollen im Vordergrund und im Hintergrund ausgeführt werden können. Die Shell soll eine Liste der Subprozesse verwalten. Folgende interaktive Eingriffe in die Programme sollen möglich sein:

- Abbruch mit STRG-C
- Anhalten mit STRG-Z
- Fortsetzen eines angehaltenen Programms mit *fg* oder *bg*

1.1 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. Status der Subprozesse ausgeben, Syntax: status

   Die Shell zeigt für alle terminierten Subprozesse, deren Status noch nicht angezeigt wurde, folgendes an: PID, PGID, Terminierungsinformation und Programmname an. Danach werden diese Daten gelöscht. Für alle noch laufenden Hintergrundprozesse werden PID, PGID, Programmname und “running” angezeigt, für angehaltene Prozesse der Zustand *stopped*.

   Beispiel:

```
>> 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
>>
```

4. Terminieren der Shell

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

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

5. Verzeichnis wechseln

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

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

6. Fortsetzung eines angehaltenen Kommandos im Vordergrund

   Syntax: `fg` *PGID*

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

7. Fortsetzung eines angehaltenen Kommandos im Hintergrund

   Syntax: `bg` *PGID*

8. Umlenkungen der Ein- und Ausgabe

   Umlenkungen der Eingabe und Ausgabe sind einzeln oder kombiniert möglich.

   - 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*
     Wie oben, aber 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.

9. Sequenz, Syntax: *Programmaufruf* ; *Programmaufruf* ; ... ; *Programmaufruf*

   Die Programme werden in Shell-Subprozessen nacheinander ausgeführt.

10. Ausführung bei Erfolg,

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

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

11. Ausführung bei Misserfolg

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

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

12. Pipeline

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

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

13. Bedingte Ausführung

    Syntax:

    ```
    if Programmaufruf then Programmaufruf else Programmaufruf fi
    if Programmaufruf then Programmaufruf fi
    ```

1.2 Signalbehandlung

Der Shell-Hauptprozess soll sich nicht mit STRG-C abbrechen lassen, sondern stattdessen einen Hinweis ausgeben, dass er mit dem *exit*-Kommando beendet wird. Damit das Betriebssystem nicht unnötig viele Zombies aufbewahrt, sollte die Shell einen Signalhandler für das SIGCHLD-Signal haben, der den Status terminierter Prozesse in der Prozessliste aktualisiert. Die Behandlung der Job-Control-Signale wird im nächsten Abschnitt beschrieben.

1.3 Prozessgruppen (Jobs), Vordergrund- und Hintergrundausführung

Zu jeder interaktiven Sitzung gehört ein Kontrollterminal, das im Dateisystem unter dem Pfad */dev/tty* zu finden ist. Ein Kommando („Job“) läuft im „Vordergrund“, wenn ihm die Tastatur des Kontrollterminals zugeordnet ist. In den meisten Fällen entspricht ein Job einem Prozess. Bei Pipelines gehören allerdings mehrere Prozesse zu dem Job. Mit der Tastenkombination STRG-C kann man den Vordergrund-Job abbrechen, mit STRG-Z anhalten. Genauer gesagt, bekommen die zum Job gehörenden Prozesse die Signale SIGINT bzw. SIGTSTP.

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.

1.4 Prozessliste

Für das Status-Kommando muss eine Prozessliste verwaltet werden. Sie können diese Liste einfach durch ein Array 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.

1.5 Sonstige Hinweise

1. Als Basis können Sie ein Shell-Skelett verwenden, in dem schon das komplette Front-End, d.h. Syntaxanalyse und Zerlegung der Kommandozeile, sowie einfache Kommandos und Sequenzen realisiert sind. Sie finden das Skelett auf dem Rechner `saturn.mni.thm.de` im Verzeichnis `~hg52/bs/shellsource`. Der Zugriff über das Internet ist beispielsweise per *rsync*, *scp* oder *sftp* möglich.
2. Bei allen Kommandos ist eine Fehlerbehandlung für fehlgeschlagene Systemaufrufe durchzuführen.
3. Für das status-Kommando muss eine Prozessliste geführt werden. 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
   ```

1.6 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 auf den MNI-Server *saturn* hochgeladen werden, wo mit moss (http://theory.stanford.edu/ aiken/moss) noch eine Plagiatsprüfung erfolgt. Dabei sind folgende Konvebtionen einzuhalten:

- Sie erzeugen auf dem Rechner „saturn.mni.thm.de“ ein Verzeichnis, dessen Name mit der Matrikelnummer des (bzw. eines) Bearbeiters übereinstimmt. Dort gibt es ein Unterverzeichnis „prog“ und eine Datei „autor.txt“. In „autor.txt“ steht der Autor (bzw. die beiden Autoren) mit folgenden Angaben: Nachname, Vorname(n), Matrikelnummer.
- Im Verzeichnis „prog“ stehen die Quelltexte, ein auf Naiade mit „make“ getestetes Makefile zur Erzeugung der Shell und eine auf „saturn“ lauffähige und getestete Shell mit dem Programmnamen „shell“.
- Das Verzeichnis wird in eine mit komprimierte tar-Archivdatei (*Matrikelnummer*`.tgz`) gepackt, die nur für den Besitzer lesbar ist, und dann auf den Rechner „saturn“ in das Verzeichnis „/home/hg52/bs/ha1/moss’ kopiert.
- 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 seinem Benutzerkonto abzugeben.

  Er meldet sich auf Saturn an und legt die für die Abgabe vorgesehenen Verzeichnisse in seinem Home-Verzeichnis an:

  ```
  $ cd
  $ mkdir 123456
  $ mkdir 123456/prog
  ```

Inhalt von „123456/autor.txt“:

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

Inhalt von „123456/prog“: Quelltextmodule: `Makefile, shell.c, shell.h, ...`
Auf Saturn lauffähiges Programm: `shell`

Archiv erzeugen, Zugriffsrechte setzen, abgeben:

```
cd
tar cvfz 123456.tgz 123456
chmod 600 123456.tgz
cp 123456.tgz ~hg52/bs/ha1/moss
```

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 `main.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

```
ls -l /tmp > ls.out
```

Sei *k* ein Zeiger auf die interne Repräsentation.

```
k->typ = K_EINFACH
k->endeabwarten = 1
k->u.einfach.wortanzahl = 3
k->u.einfach.worte[0] = "ls"
k->u.einfach.worte[1] = "-l"
k->u.einfach.worte[2] = "/tmp"
```

Sei `uml = k->u.einfach.umlenkungen` die Liste der Umlenkungen für das Beispielkommando. Die Liste ist einelementig und das Listenelement repräsentiert die Umlenkung der Standardausgabe wie folgt:

```
uml -> Kopf.filedeskriptor = 1
uml -> Kopf.modus = WRITE
uml -> Kopf.pfad = "ls.out"
```

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 <Kommando 1> then <Kommando 2> else <Kommando 3> fi
```

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

Die interne Repräsentation von

```
<Kommando 1> | <Kommando 2> | <Kommando 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 <Kommando 1> then <Kommando 2> fi
```

ist äquivalent zu

```
<Kommando 1> && <Kommando 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.