# Hausübungen 1 Betriebssysteme

Andre Rein <andre.rein@mni.thm.de>

Version v1.0, 03.05.2023: Final

# Inhalt

Lesen Sie das gesamte Dokument bevor Sie anfangen! Das könnte Ihnen unangenehme Überraschungen ersparen!

# 1. Allgemeines zu den Hausübungen

In Betriebssystem gibt es 2 anerkannte Hausübungen. Beide Hausübungen müssen **abgegeben** und **akzeptiert** werden, um eine Zulassung zur Klausur zu erhalten. Dieses Dokument beschreibt die Details zur **Hausübung 1**. Das Bestehen der Hausübung 1 ist Voraussetzung zur Teilnahme an Hausübung 2. Die Abgabe kann als Gruppenabgabe erfolgen. Es sind maximal 3 Studierende pro Gruppe erlaubt!

# 2. Aufgabenstellung

Ihre Aufgabe ist es, eine Shell in der Programmiersprache C zu implementieren.

Die Hausübung ist in **zwei Teilaufgaben** aufgeteilt:

1. **Teilaufgabe 1**: `basicsh` Programmieren Sie eine Shell (`basicsh`) in der Programmiersprache C, die *einfache* und *komplexe* Kommandos verschiedener Typen interpretiert und ausführt, z.B. einfache Kommandos, Kommandosequenzen, bedingte Ausführung von Kommandos und Pipelines.
2. **Teilaufgabe 2**: `advancedsh` Erweitern Sie die basicsh um eine Prozessliste. Die erweiterte Shell (`advancedsh`) soll in einer Liste verschiedene Informationen zu ihren Kindprozessen speichern und mit dem Kommando `status` anzeigen. Bei Subprozessterminierung muss ein Signalhandler für `SIGCHLD` aktiviert werden, der in der Prozessliste den Terminierungsstatus einträgt.

# 3. Zeitplan und Vorstellung der Hausübung 1

Die Abgabe der Hausübung erfolgt in **zwei** Schritten. Sie implementieren zuerst die Teilaufgabe 1 *basicsh*, demonstrieren diese bei einem Dozent oder Tutor in einer der Übungsstunden und reichen diese digital ein. Nur wenn die Teilaufgabe erfolgreich vorgeführt wurde, gilt die digitale Abgabe als bestanden.

Wurde Ihre Teilaufgabe 1 akzeptiert und rechtzeitig von Ihnen digital abgegeben, dann werden Sie für die Teilaufgabe 2 freigeschaltet und es folgt Schritt 2.

Für die Teilaufgabe 2 erweitern Sie die *basicsh* und implementieren die zusätzlichen Funktionen der *advancedsh*, demonstrieren diese bei einem Dozent oder Tutor in einer der Übungsstunden und reichen diese dann digital ein. Die Hausübung gilt nur dann als bestanden, wenn Sie von einem Dozent oder Tutor akzeptiert wurde und rechtzeitig von Ihnen digital abgegeben wurde.

Wenn Ihre Teilaufgabe 2 vollständig akzeptiert und rechtzeitig digital abgegeben wurde, werden Sie für die Abgabe der Hausübung 2 freigeschaltet.

Bei der Vorstellung der Teilaufgaben in der Übung ist jeder Teilnehmer der Gruppe anwesend und bereit Fragen zur Teilaufgabe zu beantworten! **Nicht anwesende Gruppenmitglieder erhalten keine Zulassung zur Klausur**!

| **Teilaufgabe** | **Fertigstellung bis** |
|---|---|
| *basicsh* | 01.06.2023 |
| *advancedsh* | 29.06.2023 |

Weitere Hinweise zur digitalen Abgabe der Hausübung finden sie in Section 7

# 4. Shell mit Basisfunktionen: basicsh

## 4.1. 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**
   a. **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.

   b. **Ausgabeumlenkung mit Anfügen**: Syntax: `Programmaufruf >> Dateipfad`
      - Falls die Datei schon vorhanden ist, wird der neue Inhalt hinter den alten geschrieben.

   c. **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 Standardeingabe des nächsten.

# 5. Shell mit Prozessliste: advancedsh

## 5.1. Prozessliste und Prozesszustände

Die Shell verwaltet eine Liste der Subprozesse. Das Kommando `status` gibt zu jedem Subprozess aus: `PID`, `PGID`, `STATUS` und `PROG`. 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 ein Signal oder `exit`-Aufruf terminiert wurde. Dabei ist `n` der Exit-Wert und `s` die Signalnummer.

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

Beispiel für die Verwendung von `status`:

```
$bshell> pwd
/home/ar
$bshell> /opt/firefox/bin/firefox &
PID=1454 PGID=1454
$bshell> urxvt &
PID=1455 PGID=1455
$bshell> kill -15 1455
$bshell> ls -l /xyz
Datei nicht gefunden
$bshell> status
PID PGID STATUS      PROG
1412 1412 exit(0)    pwd
1454 1454 running    /opt/firefox/bin/firefox
1455 1455 signal(15) urxvt
1461 1461 exit(0)    kill
1464 1464 exit(2)    ls
$bshell> status
PID  PGID STATUS     PROG
1454 1454 running    /opt/firefox/bin/firefox
$bshell>
```

## 5.2. 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.

### 5.2.1. Implementierungshinweis signalhandler

Es ist zwar über Umwege möglich zur Registrierung des Signalhandlers hier die Funktion `signal` zu verwenden, es wird aber empfohlen stattdessen die Funktion `sigaction` (`man 2 sigaction`) zu verwenden. Ein Beispiel finden Sie in der Datei `shell.c` (hier) und in der manpage. Als `FLAGS` können Sie `SA_RESTART|SA_NOCLDSTOP` verwenden.

## 5.3. Implementierungshinweis Liste

Sie können die Prozessliste durch ein Array oder eine verkettete Liste implementieren. Hierzu können Sie die Datenstrukturen aus `list.h` verwenden. Bedenken Sie hierbei, dass sie selbstständig noch weitere Verwaltungsfunktionen implementieren müssen. U.a. sind dies Funktionen zum *löschen* und *aktualisieren* von Einträgen. 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 Betriebssystemkernel geliefert wird und bei der Ausgabe der Liste dann daraus eine lesbare Statusinformation zu generieren.

# 6. Sonstige Hinweise

## 6.1. Herunterladen und Zugriff auf die Hausübung

Clonen Sie sich das GIT Repository von https://git.thm.de/arin07/betriebssysteme_public.

Das geht entweder mit:

```
$git clone git@git.thm.de:arin07/betriebssysteme_public.git
```

oder

```
$git clone https://git.thm.de/arin07/betriebssysteme_public.git
```

Wechseln Sie in das Verzeichnis: `cd betriebssysteme_public/hausuebung/hu1/`

Dort finden Sie alles was Sie benötigen.

1. Im Verzeichnis `bshell/src` finden Sie ein verwendbares Shell-Skelett, in dem schon das komplette Frontend, d.h. Syntaxanalyse und Zerlegung der Kommandozeile, sowie die Ausführung einfache Kommandos (im Vordergrund und Hintergrund) implementiert sind. Sie müssen dann noch:
   a. die interne Darstellung der Kommandos in `command.h` und `list.h` verstehen lernen,
   b. die Struktur der Ausführung von Kommandos `execute.c` studieren,
   c. die noch nicht vorhandenen Kommandos in `execute.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`

## 6.2. Compilieren der Hausübung

### 6.2.1. Verwendung von `make`

```
$ cd betriebssysteme_public/hausuebung/hu1/bshell/src
$ make
$ ./shell
```

### 6.2.2. Verwendung von `cmake`

$ cd betriebssysteme_public/hausuebung/hu1/ $ mkdir build $ cd build $ cmake ../ $ make $ ./shell

### 6.2.3. Mögliche Probleme

Wenn die Kompilierung fehl schlägt, fehlt Ihnen vermutlich ein benötigtes Programm oder eine Bibliothek. Stellen Sie sicher, dass auf Ihrem System folgende Pakete installiert sind: `bison flex libreadline-dev gcc make cmake`. Unter Ubuntu oder Debian sollten sie diese Pakete mittels: `apt-get install bison flex libreadline-dev gcc make cmake` installieren können.

# 7. Abgabe der Lösung

Sollten Sie Probleme bei der digitalen Abgabe haben, melden Sie sich einfach in einer der Übungen.

Die Hausübung muss in Einzelarbeit, in Zweiergruppen oder in Dreiergruppen bearbeitet und abgegeben werden. Die Lösungen zu den Teilaufgaben 1 und 2 müssen in den Übungsgruppen präsentiert und abgenommen werden. Zusätzlich müssen die Lösungen digital abgegeben werden:

Hierzu rufen Sie im Stammverzeichnis `betriebssysteme_public/hausuebung/hu1/` das Shellskript `submit.sh` auf.

Dann folgen Sie den Anweisungen auf dem Bildschirm. Beim erstmaligen Start müssen Sie zuerst Daten zu sich und Ihrem Team angeben. Das Python-Skript führt Sie durch diese Schritte. Eine Abgabe kann mehrmals erfolgen, d.h. wenn Sie ein korrigierte Version abgeben möchten, führen Sie das Python Skript nochmals aus.

Nachfolgend ein Beispiel zur Abgabe der Teilaufgabe 1:

```
ar@lunar:[~/betriebssysteme_public/hausuebung/hu1]$ sh-5.1$ ./submit.sh
Identified Python 3.9.2 at /usr/bin/python3.9
[Bootstrap] Identified pip 20.3.1 from /usr/lib/python3.9/site-packages/pip (python 3.9)' at /usr/bin/python3.9 -m
pip
[Bootstrap] Identified 'pipenv, version 2020.11.15' at /usr/bin/python3.9 -m pipenv
[Bootstrap] Running '/usr/bin/python3.9 -m pipenv install' to ensure the existence of a virtual environment.
[Bootstrap] This may take a moment.
Installing dependencies from Pipfile.lock (da6003)...
  □   □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 0/0 — 00:00:00
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
[Bootstrap] Running the script in a virtual environment.

=== Welcome ===
There is no team.json saved yet. You will need to provide information about your team before you can submit
solutions.
A new team.json will now be created.
Please enter details of team member 1.
Last Name:  Hacker
First Name:  Hansi
Matriculation number:  1245784547
First name: Hansi; Last name: Hacker; Matriculation number: 1245784547
Is this correct? [y/N] y
Another? [Y/n] Y
Please enter details of team member 2.
Last Name:  Root
First Name:  Roberta
Matriculation number:  47545648711
First name: Roberta; Last name: Root; Matriculation number: 47545648711
Is this correct? [y/N] y
Another? [Y/n] n
team.json successfully created.

Do you want to submit a solution now? [Y/n] y

=== Submitting ===
Please choose the assignment you want to submit.
 1 - shell-basic
```

```
 2 - shell-advanced
Please enter the id of your selection: 1
Selected variant 'bshell' for assignment 'shell-basic'.
=== Building submission ===
Running 'cmake . && make'.
=== Building submission finished ===
=== Cleaning directory for submission ===
=== Cleaning directory finished ===
=== Starting Upload ===
=== Upload finished ===
=== BEGIN Upload Log
[2021-04-22 14:48:45.709294] Received submission b9e9f610-42d2-4bcd-816d-58fb1513f277
[2021-04-22 14:48:45.709410] Ip: 94.114.202.133
[2021-04-22 14:48:45.709440] Team members: [('Hacker', 'Hansi', '1245784547'), ('Root', 'Roberta', '47545648711')]
[2021-04-22 14:48:45.719162] Your submission was ACCEPTED for course 'Betriebssysteme-ss21', assignment 'shell-
basic'.
[2021-04-22 14:48:45.719195] Your submission has the id b9e9f610-42d2-4bcd-816d-58fb1513f277.
[2021-04-22 14:48:45.719363] This assignment is currently not automatically evaluated.
[2021-04-22 14:48:45.719379] You will receive more information about the evaluation from your lecturer

=== END Upload Log
Successfully submitted your solution. Please see the upload log above for more details.
```

Für die Abgabe der Teilaufgabe 2 wählen Sie einfach die Nummer `2 shell-advanced` aus.

# 8. Anhang: Shell Frontend und Interpreter-Schnittstelle

## 8.1. Aufbau der Shell

Die Shell ist ein Terminalprogramm, das 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* (AST)) auf. Diese interne Kommandorepräsentation wird dann dem Interpreter (Ihre Shell) übergeben, der das Kommando ausführt.

## 8.2. Frontend

Als Frondend 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 sog. Tokensyntax durch reguläre Ausdrücke (`tokenscanner.l`) mit dem Scannergenerator `flex` generiert.
- Der Parser erkennt die verschiedenen Kommandotypen und verarbeitet die Eingabe nicht zeichenweise, sondern benutzt den Scanner als Hilfsfunktion. Er baut nach der Analyse des eigegebenen Kommandos die passende interne Darstellung auf. Der Parser wird aus einer formalen Spezifikation der Kommandosyntax (`tokenparser.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 Frontend wird im Rahmen der Hausübung vorgegeben und muss nicht erweitert oder modifiziert werden.**

Versuchen Sie nicht Ihre Shell ohne die Verwendung des bereitgestellten Frontends selbst zu entwickeln. Diese Versuche sind in der Vergangenheit spätestens bei komplexen Aufrufen gescheitert! Verwenden Sie einfach das bereitgestellte Frontend!

## 8.3. Interne Kommandorepräsentation

Die interne Repräsentation ist die Ausgangsbasis für die Ausführung durch den Interpreter der Shell. Die zugrunde liegende Datenstruktur ist in `command.h` definiert. Es handelt sich dabei um einen abstrakten Syntaxbaum, in dessen Wurzelknoten (`struct Command`) der *Kommandotyp* bestimmt wird. Jedes Kommando wird durch eine Kommandosequenz `CommandSequence`

repräsentiert. Bei einer Kommandosequenz handelt es sich um eine einfach verkettete Liste, in der die einzelnen einfachen Kommandos `SimpleCommand` verwaltet werden. Diese einfachen Kommandos repräsentieren das Kommando, das vom Interpreter ausgeführt wird.

Je nachdem um welchen Kommandotyp es sich handelt, hat die Kommandosequenz einen oder mehrere einfache Kommandos als Einträge. Die Anzahl der Einträge wird in `CommandSequence→command_list_len` verwaltet.

### 8.3.1. Einfache Kommandos

Ein einfaches Kommando `SimpleCommand` kann ein Shell-internes Kommando wie `cd` oder einen Programmaufruf wie `ls -l -a /tmp` repräsentieren. Die interne Repräsentation besteht aus:

- einer Liste von Wörtern (`command_tokens`) mit Längenangabe (`command_token_counter`)
- einer Liste von Ein- Ausgabeumlenkungen `redirections` und
- einem Wert `background`, der angibt ob das Kommando im Hintergrund oder Vordergrund ausgeführt werden soll.

Beispiel: Betrachten Sie das Shell-Kommando

```
ls -l -a /tmp
```

Sei `cmd` ein Zeiger auf die interne Repräsentation `Command`.

```
cmd->command_type = C_SIMPLE
cmd->command_sequence->command_list_len=1
(SimpleCommand *) cmd_s=cmd->command_sequence->command_list->head
cmd_s->command_tokens[0]="ls"
cmd_s->command_tokens[1]="-l"
cmd_s->command_tokens[2]="-a"
cmd_s->command_tokens[3]="/tmp"
cmd_s->command_token_counter=4
cmd_s->background=0
cmd_s->redirections=NULL
```

`cmd_s→redirections` repräsentiert eine Liste der Umlenkungen für Kommandos. Die Umlenkungen sind abermals eine Liste und kann 1, 2 oder 3 Elemente enthalten.

Die Umleitungen für das Kommando `ls -l > /tmp/foo` würde z.B. so aussehen

```
(Redirection *) redirection=cmd_s->redirections->head
redirection.r_type = R_FILE
redirection.r_mode = R_WRITE
redirection.u.r_file = "/tmp/foo"
```

Erarbeiten Sie sich die restlichen Strukturen selbstständig.

### 8.3.2. Listen

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

Generell arbeiten nahezu alle Datenstrukturen mit diesen Listen und erfordern, dass Sie über diese Listen iterieren.

Eine einfache Iteration über eine solche Liste, wird hier beispielhaft für die Kommandosequenzen gezeigt. Die komplette Implementierung finden Sie in der Datei `execute.c` in der Funktion `execute(Command* cmd)` :

```
int execute(Command * cmd){
    List * lst=NULL;
    ...
    ...
    lst = cmd->command_sequence->command_list;
    while (lst !=NULL){
        cmd_s=(SimpleCommand *)lst->head;
        ...
        ...
    lst=lst->tail;
    }

}
```

*Für die Umleitungen funktioniert die Iteration nahezu genauso. Sie müssen nur die korrekte Liste auswählen und auf den entsprechenden korrekten Datentyp casten.*

### 8.3.3. Komplexe Kommandos

Die oben gezeigte Implementierung von Kommandos mittels Kommandosequenzen via Listen sollte Ihnen die Verwendung von komplexen Kommandos vereinfachen. Der einzige Unterschied zwischen einem einfachen und einem komplexen Kommando ist nämlich nur, dass die Kommandosequenz mehr als ein `SimpleCommand` enthält.

Das o.a. Codebeispiel für die Listen führt bereits eine Iteration über alle in einer Kommandosequenz enthaltenen Kommandos aus. Sie müssen hierbei nur noch auf den Typ des Komplexen Kommandos achten. hierbei sind für Sie nur die folgenden Typen relevant:

- `C_SEQUENCE`
- `C_PIPE`
- `C_AND`

- `C_OR`

Diese müssen sie einzeln behandeln, je nachdem welcher Typ angegeben ist. Hierbei gilt, dass Typen nicht untereinander vermischt werden. D.h. eine Kommandosequenz ist jeweils nur einem Typ (`C_PIPE, C_AND, C_OR, C_SEQUENCE`) zugeordnet.

### 8.3.4. Anzeigen von Kommandos

Wenn Sie die Shell mit dem Parameter `--print-commands` aufrufen, wird Ihnen die Struktur der Kommandos angezeigt. Das sollte Ihnen helfen, die internen Strukturen der Kommandos besser zu verstehen.

Beispiel für den Aufruf des Kommandos `cat /bin/bash | od -x | head -1`:

```
$ ./shell --print-commands
bshell> cat /bin/bash | od -x | head -1
--- COMMANDS ---
<PIPE_COMMAND [3]>
     <SIMPLE_COMMAND> {command: "cat /bin/bash", background: "no"}
     <SIMPLE_COMMAND> {command: "od -x", background: "no"}
     <SIMPLE_COMMAND> {command: "head -1", background: "no"}
</PIPE_COMMAND>
<<<< COMMANDS >>>>
0000000 457f 464c 0102 0001 0000 0000 0000 0000
```

# 9. Anhang: Testfälle für die Shell

## 9.1. Status-Kommando

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

```
$bshell> ls -l xyzxyz &
$bshell> xterm &
$bshell> xterm &
$bshell> 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.

### 9.1.1. Beispielhafte Ausgabe:

```
$bshell> status
     PID     PGID     STATUS                NAME
  169832   169832    exit(0)                  ps
  169726   169726    running               xterm
  169636   169636    running               xterm
  169600   169600    exit(2)                  ls
$bshell> kill -9 169726
$bshell> status
     PID     PGID     STATUS                NAME
  170109   170109    exit(0)                kill
  169726   169726  signal(9)               xterm
  169636   169636    running               xterm
```

## 9.2. Umlenkung der Ein- und Ausgabe

Prüfen Sie die Umlenkungen einzeln und in Kombination:

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

Prüfen Sie die Fehlerbehandlung bei Umlenkungen:

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

## 9.3. Pipelines

Prüfen Sie Pipelines mit folgenden Tests:

- unproblematische Pipeline

`cat|cat|cat`

- 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`?

- 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.

Sollten Sie das Programm `xterm` nicht installiert haben, holen Sie dies nach oder verwenden Sie einen beliebigen anderen Terminalemulator, wie z.B. `urxvt`.

# 10. ANHANG Checkliste Hausübung 1

Nachfolgend erhalten Sie die Checkliste, die während der Überprüfung Ihrer Abgabe in den Übungsstunden verwendet wird.

Testen Sie vor der Abgabe jedes einzelne Kommando. Wenn ein Kommando nicht korrekt ausgeführt wird, gilt die Abgabe als nicht bestanden!

Die Teilaufgabe 1 *basicsh* muss alle Tests bis *Status* bestehen! Die Teilaufgabe 2 *advancedsh* muss alle Tests vollständig bestehen.

Bedenken Sie das die Änderungen aus Teilaufgabe 2 sich auch auf die Teilausgabe 1 auswirken kann. Sie sollten also vor der finalen Abgabe abermals die gesamte Checkliste durchgehen und testen!

## 10.1. cd

- ☐ cd in existierendes Verzeichnis
- ☐ cd in nicht existierendes Verzeichnis
- ☐ cd in Verzeichnis mit fehlenden Rechten
- ☐ cd ohne Argumente

```
1 $ cd /tmp
2 $ pwd
3 /tmp
4 $ cd xx
5 xx: No such file or directory
6 $ cd
7 $ pwd
8 /home/student
```

## 10.2. Ausführung bei Erfolg/Misserfolg

- ☐ Und-Verknüpfung
- ☐ Oder-Verknüpfung

```
$ true && echo yay
yay
$ false && echo nope
$ true || echo nope
$ false || echo yay
yay
$ cat && echo nope
^C
$ cat || echo yay
^Cyay
```

^C entspricht STRG+C

## 10.3. Umlenkungen

- ☐ stdout-Umleitung (> und >>)
- ☐ stdin-Umleitung
- ☐ Fehlerbehandlung

```
$ 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
```

```
$ cat < xyz
xyz: No such file or directory
$ touch outfile
$ chmod 000 outfile
$ ls >> outfile
outfile: Permission denied
```

## 10.4. Pipelines

- ☐ Unproblematische Pipelines
- ☐ Warten auf alle Pipeline-Teilnehmer
- ☐ Keine Verklemmung bei voller Pipeline
- ☐ Zustände korrekt erfasst, wenn eine Pipeline mit >10 Prozessen mit STRG-C abgebrochen wird

```
$ cat | cat | cat
$ ls -l | wc
$ xterm | cat
$ cat /bin/bash | od -x | head -1
$ cat | cat | cat | cat | ... | cat | cat | cat
^C
$ status
```

^C entspricht STRG+C

## 10.5. Status

- ☐ Rückgabewerte korrekt
- ☐ Signale korrekt
- ☐ Alte Prozesse entfernt

```
$ ls -l xzy &
$ xterm &
$ xterm &
$ ps &

$ status
$ kill -9 xtermpid
$ status
```

## 10.6. Allgemeines

- ▢ Codestruktur und Codequalität
- ▢ Signalhandler korrekt angemeldet
- ▢ SIGINT wird nicht von der Shell abgefangen
- ▢ Prozessliste sicher vor gemeinsamem Zugriff

Viel Erfolg! :)