

## Testkommandos für die Shell zu Hausübung 1 - Betriebssysteme I

### 1. cd-Kommando

```
$ cd /tmp
$ pwd
/tmp
$ cd xx
xx: Datei oder Verzeichnis nicht gefunden
$ pwd
/tmp
$ cd
$ pwd
/home/jaeger
```

### 2. 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.

### 3. 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
```

#### 4. Ausführung bei Erfolg/Misserfolg

Prüfen Sie dies mit verschiedenen exit-Werten und mit dem Signal SIGINT:

```
$ true && pwd
/home/jaeger
$ true || pwd
$ false && pwd
$ false || pwd
/home/jaeger
$ cat && pwd
STRG-C
$ cat || pwd
STRG-C
/home/jaeger
```

#### 5. 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) Werden die Zustände korrekt erfasst, wenn durch viele gleichzeitige Ereignisse SIGCHLD-Signale verloren gehen? Starten Sie eine Pipeline mit 10 Prozessen und brechen Sie sie dann mit STRG-C ab. Sind für alle Prozesse die Zustände korrekt?

```
$ cat|cat|cat|cat|cat|cat|cat|cat|cat|cat
STRG-C
$ status
...
```

## 6. 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   PROGRAMM
7835     7835    stopped  cat
>> cat|cat|cat
abc
abc
^Z
>> status
PID      PGID    STATUS   PROGRAMM
7872     7870    stopped  cat
7871     7870    stopped  cat
7870     7870    stopped  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.