2.8 Skripte/Shell-Hilfen

2.8.1 Umleitungen bei Verbundanweisungen

- Die folgenden beiden Verbundanweisungen sind ähnlich:

```
for((x=0; x<3; ++x))
do
   echo 123 > x.txt
done
```

Ergebnis: In x.txt steht einmal 123.

```
for((x=0; x<3; ++x))
do
   echo 123
done > x.txt
```

- Ergebnis: In x.txt steht dreimal 123.
- Grund: Im ersten Fall wird x.txt mehrmals zum Überschreiben geöffnet und wieder geschlossen. Im zweiten Fall wirkt die Umleitung auf die Schleife als Ganzes.
- Spannend bei: Umleitung innen und außen gleichzeitig!

```
for((x=0;x<2;++x))
> do
>    echo ABC 1>&2
> done 2> x.txt
```

- Wirkt offenbar so, als wenn die äußere Umleitung (für die Verbundanweisung) zuerst auf der Kommandozeile gestanden hätte und die innere Umleitung (für den echo-Befehl) dahinter.
- Ebenso bei der Eingabeumleitung:

```
echo abc > x.txt; echo cde >> x.txt; echo fgh >> x.txt
for((x=0; x<3; ++x))
do
   read A < x.txt
   echo "$A"
done
```

- Ergebnis: Es wird dreimal die erste Zeile ausgegeben.

```
echo abc > x.txt; echo cde >> x.txt; echo fgh >> x.txt
for((x=0; x<3; ++x))
do
   read A
   echo "$A"
done < x.txt
```

- Ergebnis: Es werden alle drei Zeilen ausgegeben.
- Grund: Im ersten Fall wird x.txt dreimal geöffnet und geschlossen, im zweiten Fall nur einmal.

2.8.2 Umleitungen bei Funktionen

- Auch einen Funktionsaufruf kann man umleiten:

```
function x()
{
   echo 123
   echo 456
}
x > x.txt
```

- Ergebnis: Wie bei Verbundanweisung. Umleitung (Öffnen und Schließen) wirkt auf gesamte Funktion.

2.8.3 Umleitungen bei Skripten

- Auch ein ganzes Skript kann man umleiten
- Ergebnis: Wie bei Verbundanweisung und Funktion. Umleitung wirkt auf das ganze Skript

2.8.4 exec-Funktion für Umleitungen

- `exec` allgemein: neues Programm ersetzt bash, gleiche PID(!)
- `exec bash`: Skript wird beendet, neue bash ersetzt aktuelle
- `exec x.sh` (Shell-Skript oder eingebauter Befehl): Skript wird beendet, neue bash ersetzt aktuelle
- `exec` ohne Parameter: es geht mit der aktuellen Shell weiter, aber: eventuell angegebene Umleitungen werden ab jetzt verwirklicht.
  Das Skript (oder die Funktion) wird nicht beendet!
- Merke: `exec bash` $\neq$ `exec`

2.8.5 Datei zur Ausgabe öffnen und schließen

- Dauerhaftes Öffnen von Ausgabe-Datei:

```
exec 8> aus.txt # 8 zum Schreiben oeffnen
echo 123 1>&8   # ausgeben nach 8
echo 456 1>&8   # ausgeben nach 8
exec 8>&-       # 8 schliessen (Operator >& mit Minuszeichen)
```

- Schreiben mit Schleife:

```
exec 8> aus.txt  # 8 zum Schreiben oeffnen
for((x=0; x<3; ++x))
do
   echo 123 1>&8 # ausgeben nach 8
done             # ohne Umleitung dahinter, die ist dauerhaft
exec 8>&-        # 8 schliessen (Operator >& mit Minuszeichen)
```

2.8.6 Datei zur Eingabe öffnen und schließen

- Dauerhaftes Öffnen von Eingabe-Datei:

```
echo 123 > ein.txt; echo 456 > ein.txt echo 789 >ein.txt
exec 9< ein.txt # 9 zum Lesen oeffnen
read 0<&4       # eingeben von 9, 1. Zeile. Oder read -u 4
echo $REPLY     # Ergebnis
read 0<&4       # eingeben von 9, 2. Zeile. Oder read -u 4
echo $REPLY     # Ergebnis
exec 9<&-       # 9 schliessen (Operator <& mit Minuszeichen)
```

- Hinweis: Beim read-Befehl kann man anstatt der Umleitung die Option -u (=use Kanal x) benutzen. Dann sieht dieselbe Befehlsfolge so aus:

```
echo 123 > ein.txt; echo 456 > ein.txt echo 789 >ein.txt
exec 9< ein.txt # s.o.
read -u 4       # neue Schreibweise, sonst wie oben
echo $REPLY     # s.o.
read -u 4       # neue Schreibweise, sonst wie oben
echo $REPLY     # s.o.
exec 9<&-       # s.o.
```

- Einlesen aller Zeilen mit Schleife:

```
echo 123 > ein.txt; echo 456 > ein.txt echo 789 >ein.txt
exec 9< ein.txt # 9 zum Lesen oeffnen
while read -u 4 # eingeben von 9, 1. bis letzte Zeile
do              #
   echo $REPLY  # Ergebnis
done            # ohne Umleitung dahinter, die ist ja dauerhaft
exec 9<&-       # 9 schliessen (Operator <& mit Minuszeichen)
```

2.8.7 Datei zur Ein- und Ausgabe öffnen und schließen

- Der Operator <> erlaubt es, dieselbe Datei zum Lesen und zum Schreiben zu öffnen.
- Beim Schließen ist es egal, ob man >&- oder <&- angibt: die Datei wird geschlossen (für beide Richtungen). Nur <>&- ist nicht möglich (wurde so festgelegt).

```
echo "Hallo, Welt" > daten.txt # Datei vorbereiten
exec 7<> daten.txt # Datei fuer E+A oeffnen
read -n 8 -u7      # Lies 8 Bytes aus Kanal 7
echo -n a >&7      # Schreibe ein a an diese Stelle
exec 7>&-          # Datei schliessen
cat daten.txt      # Ergebnis testen
```

Ergebnis: Hallo, Walt

2.8.8 Signale behandeln

Signale dienen zur Kommunikation zwischen Prozessen: Ein Prozess schickt ein Signal, um einem anderen Prozess etwas mitzuteilen. Der andere Prozess sollte auf das Signal angemessen reagieren. Standardmäßig reagieren Prozesse auf ein Signal mit einem Standardverhalten, das eins von zwei Extremen ist:

a) Ignorieren (=weitermachen wie bisher)

b) Abbrechen

Für jedes Signal ist festgelegt, welches Verhalten erfolgt.

Aber oft möchte man ein anderes Verhalten erreichen, etwa das Neu-Einlesen der Konfiguration oder irgendetwas Spezielles. Ein Programm, das direkt mit dem Kernel kommuniziert, kenn dafür den API-Aufruf `signal()`.

Auf der Shell-Ebene gibt es stattdessen den eingebauten Befehl `trap`, der das Gleiche leistet. `trap` hat zwei Parameter:

a) der erste Parameter ist die Befehlszeile (eventuell maskiert)

b) die folgenden Parameter sind die Nummern der Signale, die auf diese Weise abgefangen werden sollen.

Diesen Befehl kann man direkt auf der Befehlszeile einsetzen:

Terminal

```
schueler@debian964:~$ for x in {1..31}; do \
 trap "echo _Signal Nr. $x_" "$x"; done
schueler@debian964:~$ # Drücke Strg-C
_Signal Nr. 2_n:~$ ^C
schueler@debian964:~$ # Drücke Strg-AltGr-ß
_Signal Nr. 2_n:~$ ^\
schueler@debian964:~$ # Drücke Strg-Z
_Signal Nr. 20_:~$ ^Z
schueler@debian964:~$ uname
Linux
_Signal Nr. 17_
schueler@debian964:~$ sleep 10 # danach Strg-C
^C
_Signal Nr. 17_
schueler@debian964:~$ sleep 10 # danach Strg-AltGr-ß
^\Verlassen
_Signal Nr. 17_
schueler@debian964:~$ sleep 10 # danach Strg-Z
^Z
[1]+  Angehalten              sleep 10
_Signal Nr. 17_
```

Mit der Option `-p` kann man sehen, welche Befehlszeile momentan für welches Signal eingestellt ist:

Terminal

```
schueler@debian964:~$ trap "echo hallo" 16
schueler@debian964:~$ trap -p
trap -- 'echo hallo' SIGSTKFLT
```

Sinnvoll ist der Befehl aber vor allem in Skripten:

```
1 #!/bin/bash
2 trap "echo␣Probe␣1"          10 # Signal 10 abfangen
3 trap "echo␣Probe␣2;␣exit␣1" 15 # Signal 12 abfangen+exit
4
5 while true;
6 do
7    sleep 1
8 #  echo "warte ... "
9 done
```

[title=traptest.sh] Wenn man dieses Skript im Hintergrund aufgerufen hat, kann man versuchen, ihm eins der beiden Signale zu schicken:

Terminal

```
schueler@debian964:~$ traptest.sh &
[1] 3723
schueler@debian964:~$ kill -10 %1
schueler@debian964:~$  Benutzerdefiniertes Signal 1
Probe 1
schueler@debian964:~$
schueler@debian964:~$ kill -12 %1
schueler@debian964:~$
[1]+ Benutzerdefiniertes Signal 2   traptest
schueler@debian964:~$
```