## Aufgabe 1a)

Schreiben Sie ein Shellskript, welches zwei Parameter akzeptiert. Der erste Parameter soll eine ID z.B. 1 oder 2 und der zweite eine Sekundenzahl zwischen 2 und 10 enthalten.
Es soll die im zweiten Parameter angegebene Anzahl von Sekunden lang nichts tun ( siehe `man sleep` ), dann eine Meldung, welche die ID enthält ausgeben (z.B. "Hier ist Prozess 1." bzw. "Hier ist Prozess 2."), dann nochmals dieselbe Zeit schlafen und zum Abschluss erneut seine Meldung ausgeben. Rufen Sie das Skript zweimal mit unterschiedlicher ID auf.

Um ein Programm/Skript im Hintergrund ablaufen zu lassen, wird dem Aufruf ein „`&`“ angehängt.

Beispiel: `$ ./skript &`

Das Skript an sich könnte folgendermaßen aussehen:

```
#!/bin/bash
nummer=${1:-"(keine Nummer angegeben)"}
zeit=${2:-1}
sleep $zeit
echo "Hier ist Prozess $nummer."
sleep $zeit
echo "Hier ist Prozess $nummer."
```

Der Aufruf dieses Skriptes (für die Übungsaufgabe) als Hintergrundprozesse:

```
$ ./skript 1 5 &; ./skript 2 5 &
```

Bei diesem Aufruf würde das Skript jeweils 5 Sekunden „schlafen“ und sich als Prozess 1 bzw. 2 melden.
Diese Aufrufe können in ein Skript verpackt werden, bei dem zunächst eine Überprüfung von `$2` (auf 2<=`$2`<=10) gemacht wird, um anschließend die Aufrufe mit `$1` und `$2` zu parametrisieren und mit `&` zu versehen.

## Aufgabe 1b)

Geben Sie den Prozessstatus des Systems aus, während die Prozesse schlafen (siehe man ps).

Der Befehl „`ps`“ listet alle Prozesse auf, die aktuell auf dem System laufen.
Ohne Parameter aufgerufen listet er nur die Prozesse auf, die im Kontext der aktuellen Shell laufen. Mit „`-u`“ bzw. „`u`“ wird auch der Prozessstatus angezeigt.
Mit dem Parameter „`-A`“ lassen sich beispielsweise alle auf dem System laufenden Prozesse anzeigen.

Die Ausgabe (von „`$ ps -u`“) könnte z.B. so aussehen:

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
andreas   3550  0.1  0.3   5760  3196 pts/2    Ss   09:42   0:00 bash
andreas   3863  0.0  0.1   4504  1532 pts/2    SN   09:47   0:00 ./skript
andreas   3864  0.0  0.1   4504  1528 pts/2    SN   09:47   0:00 ./skript
andreas   3867  0.0  0.0   2960   632 pts/2    SN   09:47   0:00 sleep 20
andreas   3870  0.0  0.0   2960   628 pts/2    SN   09:47   0:00 sleep 20
andreas   3872  0.0  0.1   2648  1020 pts/2    R+   09:47   0:00 ps u
```

In der zweiten Spalte sieht man die sogenannte PID, also die Prozess-ID. Diese ist für die nächste Aufgabe von Bedeutung.

## Aufgabe 1c)

Beenden Sie einen Prozess vorzeitig ( siehe man kill ).

Mit „`kill`“ sendet man sogenannte Signale an einen Prozess. Wird keine Signalnummer angegeben, wird das Signal SIGTERM (Nummer 15) an den als Parameter angegebenen Prozess gesendet. Dieses Signal fordert einen Prozess auf, sich zu beenden.
Folgender Befehl würde z.B. den Prozess „skript“ mit der PID 3863 beenden:

```
$ kill 3863
```

Möchte man das Signal explizit angeben, sieht ein Aufruf folgendermaßen aus:

```
$ kill -9 3863
```

Hier wird das Signal 9 (SIGKILL) gesendet, welches den Prozess etwas „rücksichtsloser“ unmittelbar beendet.

## Aufgabe 1d)

Starten Sie einen Hintergrundprozess, der seinerseits drei neue Hintergrundprozesse startet. Jeder dieser drei Prozesse soll unmittelbar hintereinander 250mal die Meldung „Hallo, hier ist Prozess x.“ ausgeben, wobei x je nach Prozess „A“ „B“ oder „C“ ist. Auch hier lassen sich gut Shellskripte einsetzen. Was beobachten Sie bezüglich der Reihenfolge der Prozessausführungen?

Eine Lösungsmöglichkeit mit 2 kleinen Skripten könnte folgendermaßen aussehen:
Datei „`unterskript`“:

```
#!/bin/bash
prozess=${1:-"(kein Buchstabe angegeben)"}
for i in {1..250}
do
     echo "Hallo, hier ist Prozess $prozess."
done
```

Datei „`hauptskript`“:

```
#!/bin/bash
./unterskript A &
./unterskript B &
./unterskript C &
```

Das Hauptskript wird dann folgendermaßen aufgerufen:

```
$ ./hauptskript &
```

Die Reihenfolge der Prozessausführungen und somit die Reihenfolge/Mischung der Ausgaben ist von Ausführung zu Ausführung und von System zu System etwas verschieden. (Es kann also keine „eindeutige Lösung“ für die Ausgabe angegeben werden.)

### Aufgabe 2a)

```
#!/bin/bash
while [ 1 -eq 1 ]
do
  echo -n "."
  sleep 1
done
```

Speichern Sie obiges Skript in eine Textdatei trap_example.sh und führen Sie es in einer Shell aus. Installieren Sie einen Signalhandler, der das Beenden durch <strg>+<c> verhindert.

<strg>+<c> sendet das Signal SIGINT.

Es lässt sich mit folgender Zeile abfangen:

```
trap "echo SIGINT, Signal 2 abgefangen" INT
```

### Aufgabe 2b)

Stoppen Sie das Skript mit <strg>+<z>, und sehen Sie sich das Ergebnis mit "ps u" an. Beenden Sie das Skript endgültig (kill), damit es nicht mehr in der Ausgabe von "ps u" auftaucht. Installieren Sie einen Signalhandler für <strg>+<z> (TSTP). Versuchen Sie, das Skript mit <strg>+<z> zu stoppen. Warum funktioniert das nicht wie erwartet (Kontrollieren Sie den Prozessstatus mit "ps u")? Unterscheiden sich die Ergebnisse, wenn Sie TSTP mittels kill an das Skript senden? Warum?

<strg>+<z> sendet das Signal SIGTSTP

Ergebnis von ps -u für <strg>+<z> ohne Signalhandler:

```
USER       PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
uname    21929  0.0  0.0   6088  1460 pts/3 T    12:56   0:00 sh trap_example.sh
uname    21932  0.0  0.0   4720   500 pts/3 T    12:56   0:00 sleep 1
```

Das Skript lässt sich mittels kill $PID (in diesem Fall 21929) beenden. Es wird aus der Prozesstabelle gelöscht und taucht daher auch nicht mehr in der Ausgabe von ps -u auf.

Handler für SIGTSTP:

```
trap "echo SIGTSTP abgefangen" TSTP
```

Ergebnis von ps -u für <strg>+<z> nach Installieren eines Handlers für SIGTSTP:

```
USER       PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
uname    21940  0.0  0.0   6088  1460 pts/3 S+   12:59   0:00 sh trap_example.sh
uname    21946  0.0  0.0   4720   504 pts/3 T+   12:59   0:00 sleep 1
```

Das Signal hat nicht trap_example.sh, sondern nur den sleep-Prozess gestoppt.
trap_example.sh wartet jedoch auf sleep, weshalb keine Ausgabe mehr erfolgt.
Man erhält auch nicht, wie normalerweise, die Kommandozeile zurück;
das Skript "hängt“ in einem wartenden Zustand.

Wird SIGTSTP über kill direkt an trap_example.sh gesendet, arbeitet der Signalhandler wie erwartet: das Signal wird ignoriert, es erfolgt eine Ausgabe.

**Aufgabe 2c)**

Installieren Sie einen Signalhandler für das Standardsignal von kill, um die Standardaktion (welche war das noch einmal?) zu unterbinden.

Das Standardsignal von kill ist SIGTERM (Signal 15), die Standardaktion darauf ist exit.

Handler für SIGTERM:

```
trap "echo SIGTERM abgefangen" TERM
```

**Aufgabe 2d)**

Was passiert, wenn Sie einen Handler für das Signal 9 installieren?

Da sich Signal 9 nicht abfangen lässt, wird der Prozess trotzdem beendet.

Das fertige Skript mit allen Signalhandlern:

```
#!/bin/bash

# Handler für Signal 2 (<strg>+<c>)
trap "echo SIGINT, Signal 2 abgefangen" SIGINT

# Handler für Signal 15
trap "echo SIGTERM, Signal 15 abgefangen" SIGTERM

# <strg>+<z> (Signalnr, architekturabhängig, 20 auf i386)
trap "echo SIGTSTP abgefangen" SIGTSTP

# Handler für Signal 9 - ohne Wirkung
trap "echo SIGKILL, Signal 9 abgefangen" SIGKILL
```

```
# Handler für SIGSTOP - ebenfalls ohne Wirkung
# (Signalnummer, architekturabhängig, 19 auf i386)
trap 'echo SIGSTOP abgefangen' SIGSTOP

while [ 1 -eq 1 ]
do
 echo -n "."
 sleep 1
done
```