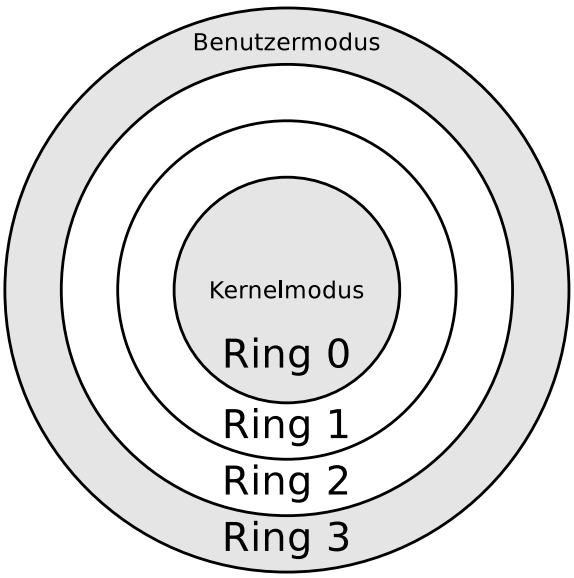
# Lösung von Übungsblatt 7

## Aufgabe 1 (Systemaufrufe)

1. x86-kompatible CPUs enthalten 4 Privilegienstufen („Ringe“) für Prozesse. Markieren Sie in der Abbildung (*deutlich erkennbar!*) den Kernelmodus und den Benutzermodus.




2. Nennen Sie den Ring, dem der Betriebssystemkern zugeordnet ist.

   *In Ring 0 (= Kernelmodus) läuft der Betriebssystemkern.*

3. Nennen Sie den Ring, dem die Anwendungen der Benutzer zugeordnet sind.

   *In Ring 3 (= Benutzermodus) laufen die Anwendungen.*

4. Nennen Sie den Ring, bei dem Prozesse vollen Zugriff auf die Hardware haben.

   *Prozesse im Kernelmodus (Ring 0) haben vollen Zugriff auf die Hardware.*

5. Nennen Sie einen Grund für die Unterscheidung von Benutzermodus und Kernelmodus.

   *Verbesserung von Stabilität und Sicherheit.*

6. Beschreiben Sie, was ein Systemaufruf ist.

   *Ein Systemaufruf ist ein Funktionsaufruf im Betriebssystem, der einen Sprung vom Benutzermodus in den Kernelmodus auslöst ($\Longrightarrow$ Moduswechsel)*

7. Beschreiben Sie, was ein Moduswechsel ist.

*Ein Prozess gibt die Kontrolle über die CPU an den Kernel ab und wird unterbrochen bis die Anfrage fertig bearbeitet is. Nach dem Systemaufruf gibt der Kernel die CPU wieder an den Prozess im Benutzermodus ab. Der Prozess führt seine Abarbeitung an der Stelle fort, an der der Kontextwechsel zuvor angefordert wurde.*

8. Nennen Sie zwei Gründe, warum Prozesse im Benutzermodus Systemaufrufe nicht direkt aufrufen sollten.

   *Direkt mit Systemaufrufen arbeiten ist unsicher und schlecht portabel.*

9. Damit Prozesse im Benutzermodus nicht immer Systemaufrufe aufrufen müssen, gibt es eine alternative Vorgehensweise. Beschreiben Sie diese.

   *Verwendung einer Bibliothek, die zuständig ist für die Kommunikationsvermittlung der Benutzerprozesse mit dem Kernel und den Moduswechsel zwischen Benutzermodus und Kernelmodus.*

# Aufgabe 2 (Prozesse)

1. Nennen Sie die drei Arten von Prozesskontextinformationen, die das Betriebssystem speichert.

   *Benutzerkontext, Hardwarekontext und Systemkontext.*

2. Geben Sie an, welche Prozesskontextinformationen nicht im Prozesskontrollblock gespeichert sind.

   *Der Benutzerkontext, also die Daten im zugewiesenen Adressraum (virtuellen Speicher).*

3. Beschreiben Sie, warum nicht alle Prozesskontextinformationen im Prozesskontrollblock gespeichert sind.

   *Weil der virtuelle Speicher jedes Prozesses je nach verwendeter Architektur mehrere GB oder mehr groß sein kann. Der Benutzerkontext ist damit einfach zu groß, um ihn doppelt zu speichern.*

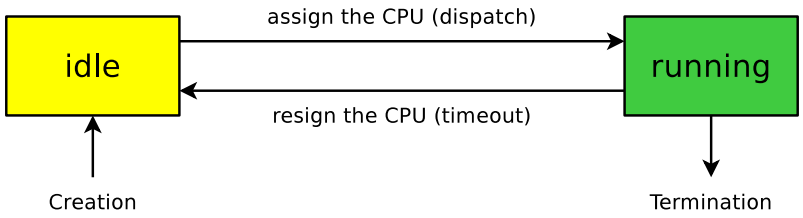
4. Beschreiben Sie die Aufgabe des Dispatchers.

   *Aufgabe des Dispatchers ist die Umsetzung der Zustandsübergänge der Prozesse.*

5. Beschreiben Sie die Aufgabe des Schedulers.

   *Er legt die Ausführungsreihenfolge der Prozesse mit einem Scheduling-Algorithmen fest.*

6. Das 2-Zustands-Prozessmodell ist das kleinste, denkbare Prozessmodell. Tragen Sie die Namen der Zustände in die Abbildung des 2-Zustands-Prozessmodells ein.




7. Ist das 2-Zustands-Prozessmodell sinnvoll? Begründen Sie kurz ihre Antwort.

   *Das 2-Zustands-Prozessmodell geht davon aus, dass alle Prozesse immer zur Ausführung bereit sind. Das ist aber unrealistisch, denn es gibt fast immer Prozesse, die blockiert sind. Die untätigen Prozesse müssen in mindestens zwei Gruppen unterschieden werden:*

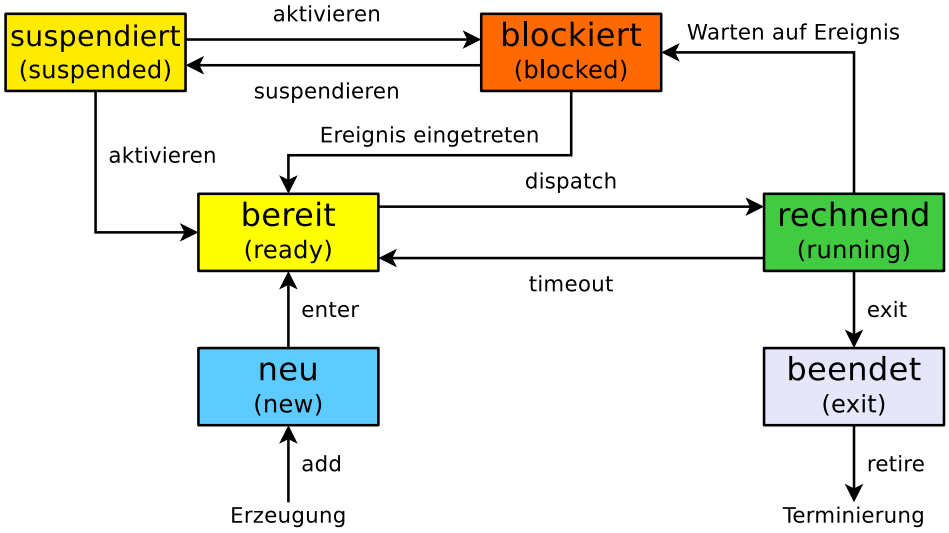
   - *Prozesse die im Zustand bereit (ready) sind.*
   - *Prozesse die im Zustand blockiert (blocked) sind.*

8. Begründen Sie, warum wir in der Vorlesung das 3-Zustands-Prozessmodell um die Zustände `neu` und `beendet` erweitert haben.

   - `neu`: *Der Prozess (Prozesskontrollblock) ist erzeugt, wurde aber vom Betriebssystem noch nicht in die Warteschlange für Prozesse im Zustand* `bereit` *eingefügt. Motivation: Auf manchen Systemen ist die Anzahl der ausführbaren Prozesse limitiert, um Speicher zu sparen und den Grad des Mehrprogrammbetriebs festzulegen*

   - `exit`: *Der Prozess ist fertig abgearbeitet oder wurde beendet, aber sein Prozesskontrollblock existiert aus verschiedenen Gründen noch. In der Regel sind Ressourcen noch nicht freigegeben oder der Elternprozess hat den Rückgabewert des Kindprozesses noch nicht die angenommen.*

9. Tragen Sie die Namen der Zustände in die Abbildung des 6-Zustands-Prozessmodells ein.




10. Beschreiben Sie, was ein Zombie-Prozess ist.

    *Ein Zombie-Prozess ist fertig abgearbeitet (via Systemaufruf `exit`), aber sein Eintrag in der Prozesstabelle existiert so lange, bis der Elternprozess den Rückgabewert (via Systemaufruf `wait`) abgefragt hat. Seine PID kann noch nicht an einen neuen Prozess vergeben werden.*

11. Beschreiben Sie die Aufgabe der Prozesstabelle.

    *Zur Verwaltung der Prozesse führt das Betriebssystem die Prozesstabelle. Es ist eine Liste aller existierenden Prozesse. Sie enthält für jeden Prozess einen Eintrag, den Prozesskontrollblock.*

12. Wie viele Warteschlangen für Prozesse im Zustand „blockiert“ verwaltet das Betriebssystem?

    *Für jedes Ereignis existiert eine eigene Warteschlange mit den Prozessen, die auf das Ereignis warten.*

13. Beschreiben Sie was passiert, wenn ein neuer Prozess erstellt werden soll, es aber im Betriebssystem keine freie Prozessidentifikation (PIDs) mehr gibt.

    *Dann kann kein neuer Prozess erstellt werden.*

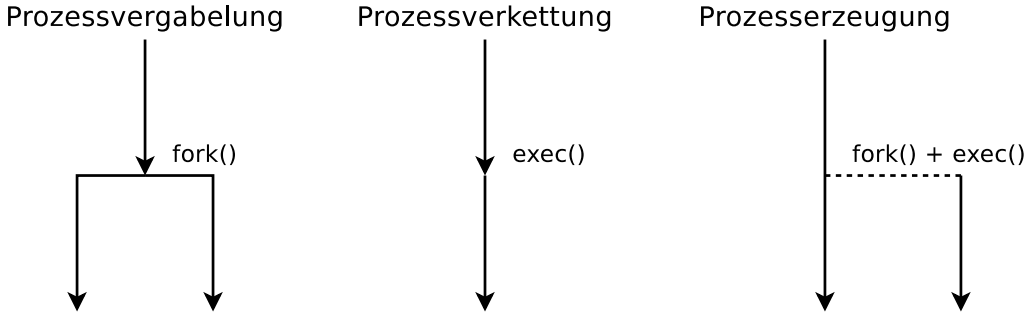
14. Beschreiben Sie, was der Systemaufruf `fork()` macht.

    *Ruft ein Prozess `fork()` auf, wird eine identische Kopie als neuer Prozess gestartet.*

15. Beschreiben Sie, was der Systemaufruf `exec()` macht.

    *Der Systemaufruf `exec()` ersetzt einen Prozess durch einen anderen.*

16. Die drei Abbildungen zeigen alle existierenden Möglichkeiten, einen neuen Prozess zu erzeugen. Schreiben Sie zu jeder Abbildung, welche(r) Systemaufruf(e) nötig sind, um die gezeigte Prozesserzeugung zu realisieren.




17. Ein Elternprozess (PID = 75) mit den in der folgenden Tabelle beschriebenen Eigenschaften erzeugt mit Hilfe des Systemaufrufs `fork()` einen Kindprozess (PID = 198). Tragen Sie die vier fehlenden Werte in die Tabelle ein.

| | **Elternprozess** | **Kindprozess** |
|---|---|---|
| PPID | 72 | *75* |
| PID | 75 | 198 |
| UID | 18 | *18* |
| Rückgabewert von `fork()` | *198* | *0* |

*Erklärung: Hat die Erzeugung eines Kindprozesses mit* `fork()` *geklappt, ist der Rückgabewert von* `fork()` *im Elternprozess die PID des neu erzeugten Kindprozesses. Im Kindprozess ist der Rückgabewert von* `fork()` *0. Die Benutzer-Identifikation (UID) von Elternprozess und Kindprozess ist identisch. Die Parent Process ID (PPID) des Kindprozesses ist die PID des Elternprozesses.*

18. Der folgende C-Quellcode erzeugt einen Kindprozess. Welchen Wert hat die Variable `returnvalue` beim Kindprozesses und welchen Wert hat sie beim Elternprozess. Gehen Sie bei Ihrer Antwort auch auf die Bedeutung des Rückgabewerts beim Elternprozess ein und erklären Sie, warum der Rückgabewert für den Elternprozess sehr wichtig ist.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  void main() {
6    int returnvalue = fork();
7
8    if (returnvalue < 0) {
9      printf("Error.\n");
10     exit(1);
11   }
12   if (returnvalue > 0) {
13     printf("Parent Process.\n");
14     exit(0);
```

```
15   }
16   else {
17     printf("Child Process.\n");
18     exit(0);
19   }
20 }
```

*Beim Kindprozess hat `fork()` den Rückgabewert 0.*

*Beim Elternprozess hat `fork()` einen positiven Rückgabewert. Der Rückgabewert entspricht dann der PID des neu erzeugten Kindprozesses. Durch diesen Rückgabewert kann der Elternprozess den Kindprozess identifizieren.*

19. Erklären Sie, was `init` ist.

    *`init` ist der erste Prozess unter Linux/UNIX. Er hat die PID 1. Alle laufenden Prozesse stammen von `init` ab. `init` ist der Vater aller Prozesse.*

20. Nennen Sie den Unterschied eines Kindprozess vom Elternprozess kurz nach der Erzeugung.

    *Die PID, die PPID und die Speicherbereiche.*

21. Beschreiben, Sie was passiert, wenn ein Elternprozess vor dem Kindprozess beendet wird.

    *`init` adoptiert den Kind-Prozess. Die PPID des Kind-Prozesses hat dann den Wert 1.*

22. Nennen Sie den Inhalt des Textsegments.

    *Den ausführbaren Programmcode (Maschinencode).*

23. Nennen Sie den Inhalt des Datensegments.

    *Initialisierte Variablen. Enthält globale Variablen (Deklaration ist außerhalb von Funktionen), und lokale statische Variablen, denen ein Anfangswert zugewiesen wird.*

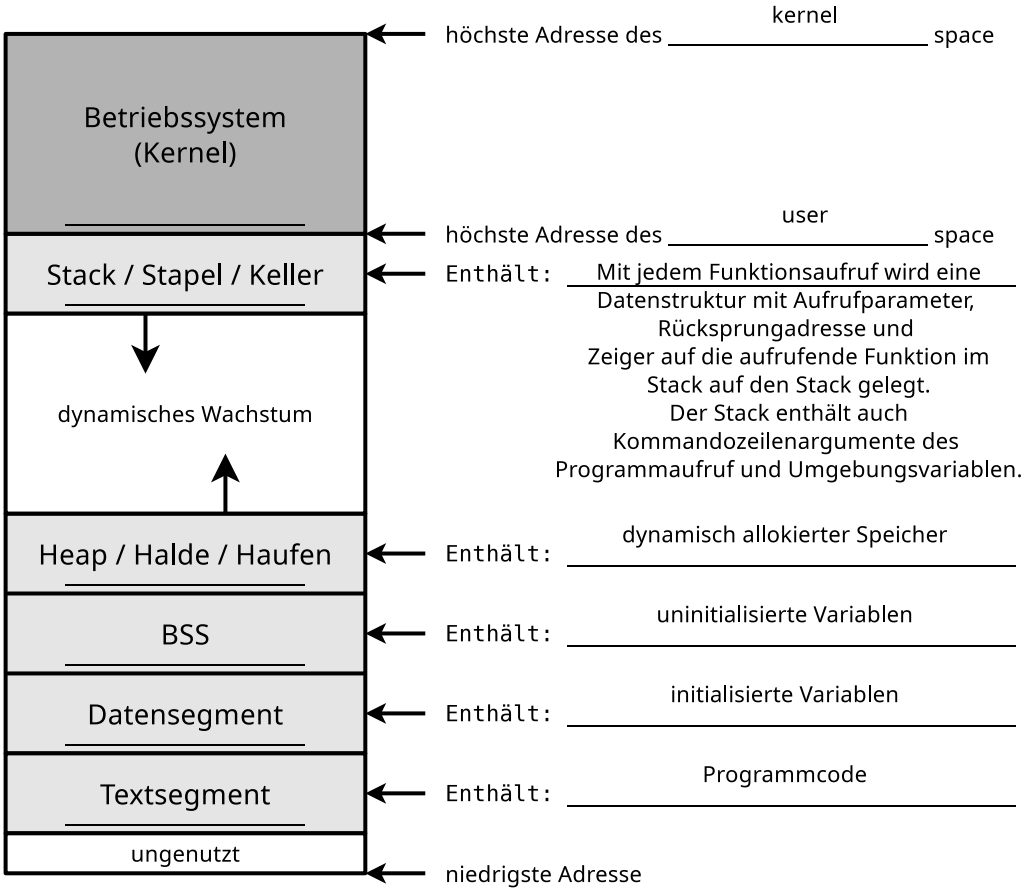
24. Nennen Sie den Inhalt des BSS.

    *Nicht initialisierte Variablen. Enthält globale Variablen (Deklaration ist außerhalb von Funktionen) und lokale statische Variablen, denen kein Anfangswert zugewiesen wird.*

25. Nennen Sie den Inhalt des Stack.

    *Aufrufparameter und Rücksprungadressen der Funktionen, lokale Variablen der Funktionen.*

26. Die Abbildung zeigt die Struktur eines UNIX-Prozesses im Speicher. Ergänzen Sie die fehlenden Bezeichnungen (Fachbegriffe) der prozessbezogenen Daten und die fehlenden Informationen zum Inhalt dieser Daten.




# Aufgabe 3 (Informationen zu Prozessen im Betriebssystem)

In der Ausgabe des Kommandos `ps` finden Sie hilfreiche Informationen zu den Prozessen im Betriebssystem.

```
$ ps -eFw
UID      PID   PPID  C      SZ    RSS PSR STIME TTY        TIME CMD
root       1      0  0   42090  12820   0 Aug29 ?      00:00:03 /sbin/initroot
root       2      0  0       0      0   4 Aug29 ?      00:00:00 [kthreadd]
...
bnc     2149   1782  1  258958 133484   7 Aug29 ?      00:11:20 xfwm4 --display :0.0  ...
bnc     2474   1782  0  137013  54512   8 Aug29 ?      00:03:28 xfce4-panel --display :0.0 ...
bnc     2478   1782  0  166034 138652  15 Aug29 ?      00:00:20 xfdesktop --display :0.0 ...
bnc     3252   2474  3 8590107 577484   9 Aug29 ?      00:51:07 /opt/google/chrome/chrome
bnc     3530   1721  0  157125  62824   0 Aug29 ?      00:00:44 /usr/libexec/gnome-terminal-server
bnc     3568   3530  0    3271   9556  15 Aug29 pts/0  00:00:01 bash
root    6706      1  0    7087  10556   3 Aug29 ?      00:00:00 /usr/sbin/cupsd -l
root    6737      1  0   44549  18680  12 Aug30 ?      00:00:00 /usr/sbin/cups-browsed
bnc    72577  72539  0    2773   7224   4 Aug31 pts/1  00:00:00 /bin/bash
bnc    90775  72577  1  279130 187352   9 09:39 pts/1  00:00:04 okular thesis.pdf
```

```
bnc    94414   3568  0    2861   4952   6 11:19 pts/0  00:00:00 ps -eFw
```

1. Nennen Sie den Inhalt der Spalte `UID`.

   *User-ID. Die Benutzerkennung des Besitzers des Prozesses.*

2. Nennen Sie den Inhalt der Spalte `PID`.

   *Prozess-ID. Die eindeutige Kennung des Prozesses.*

3. Nennen Sie den Inhalt der Spalte `PPID`.

   *Parent Prozess-ID. Die eindeutige Kennung des Elternprozesses.*

4. Nennen Sie den Inhalt der Spalte `C`.

   *CPU-Belastung des Prozesses in Prozent.*

5. Nennen Sie den Inhalt der Spalte `SZ`.

   *Virtuelle Prozessgröße = Textsegment, Heap und Stack.*

6. Nennen Sie den Inhalt der Spalte `RSS`.

   *Resident Set Size = Belegter physischer Speicher (ohne Swap) in kB.*

7. Nennen Sie den Inhalt der Spalte `PSR`.

   *Nummer des Prozess zugewiesenen CPU-Kerns.*

8. Nennen Sie den Inhalt der Spalte `STIME`.

   *Startzeitpunkt des Prozesses*

9. Nennen Sie den Inhalt der Spalte `TTY`.

   *Teletypewriter = Steuerterminal. Meist ein virtuelles Gerät: pts (pseudo terminal slave)*

10. Nennen Sie den Inhalt der Spalte `TIME`.

    *Bisherige Rechenzeit des Prozesses auf der CPU (HH:MM:SS).*

11. Nennen Sie den Elternprozess des Prozesses, der diese Übersicht der Prozesse in der Kommandozeile ausgegeben hat.

    *Der `bash`-Prozess mit der PID 3568 ist der Elternprozess des `ps`-Prozesses mit der PID 94414.*

# Aufgabe 4 (Zeitgesteuerte Kommandoausführung, Kontrollstrukturen, Archivierung)

1. Schreiben Sie ein Shell-Skript, das zwei Zahlen als Kommandozeilenargumente einliest. Das Skript soll prüfen, ob die Zahlen identisch sind und das Ergebnis der Überprüfung ausgeben.

```
#!/bin/bash
#
# Skript: vergleich.bat
#
echo "Geben sie zwei Zahlen ein"
read -p "Zahl1:" zahl1
read -p "Zahl2:" zahl2
if [ $zahl1 -eq $zahl2 ] ; then
  echo "Die beiden Zahlen sind gleich groß."
else
  echo "Die beiden Zahlen sind nicht gleich groß."
fi
```

2. Erweitern Sie das Shell-Skript dahingehend, dass wenn die Zahlen nicht identisch sind, überprüft wird, welche der beiden Zahlen die Größere ist. Das Ergebnis der Überprüfung soll ausgegeben werden.

```
#!/bin/bash
#
# Skript: vergleich2.bat
#
echo "Geben sie zwei Zahlen ein"
read -p "Zahl1:" zahl1
read -p "Zahl2:" zahl2
if [ $zahl1 -eq $zahl2 ] ; then
  echo "Die beiden Zahlen sind gleich groß."
elif [ $zahl1 -gt $zahl2 ]; then
  echo "Zahl 1 mit Wert $zahl1 ist größer."
else
  echo "Zahl 2 mit Wert $zahl2 ist größer."
fi
```

3. Schreiben Sie ein Shell-Skript, das ein Verzeichnis Ihrer Wahl sichert. Von dem Verzeichnis soll eine Archivdatei mit der Endung `.tar.bz2` erzeugt werden. Diese Datei soll im Verzeichnis `/tmp` abgelegt werden. Der Name der Archivdatei entsprechen soll folgendem Benennungsschema:

   `Backup_<USERNAME>_<JAHR>_<MONAT>_<TAG>.tar.bz2`

   Die Felder `<USERNAME>`, `<JAHR>`, `<MONAT>` und `<TAG>` sollen durch die aktuellen Werte ersetzt werden.

```
#!/bin/bash
```

```
2 #
3 # Skript: archiv_erstellen.bat
4 #
5 ARCHIVNAME="Backup_`whoami`_`date +"%Y_%m_%d"`.tar.bz2"
6 VERZEICHNIS="/tmp/testverzeichnis"
7
8 # Archivdatei mit bz2-Kompression erstellen
9 # c => create an archive file.
10 # j => bz2 compression.
11 # v => show detailed output of command.
12 # f => filename of archive file.
13 tar -cjvf $ARCHIVNAME $VERZEICHNIS
14
15 # Archivdatei nach /tmp verschieben
16 mv $ARCHIVNAME /tmp
```

4. Schreiben Sie ein Shell-Skript, das testet, ob heute schon eine Archivdatei gemäß dem Benennungsschema aus Teilaufgabe 3 angelegt wurde. Das Ergebnis der Überprüfung soll in der Shell ausgegeben werden.

```
1 #!/bin/bash
2 #
3 # Skript: archiv_untersuchen.bat
4 #
5 DATEI="/tmp/Backup_`whoami`_`date +"%Y_%m_%d"`.tar.bz2"
6
7 if [ ! -f $DATEI ] ; then
8   echo "Die Datei $DATEI existiert nicht.";
9 else
10   echo "Die Datei $DATEI existiert.";
11 fi
```

5. Schreiben Sie zwei `cron`-Jobs. Der erste `cron`-Job soll an jedem Tag (außer am Wochenende) um 6:15 Uhr das Shell-Skript aus Teilaufgabe 3 aufrufen, das die Archivdatei mit dem Backup erzeugt.

   Der zweite `cron`-Job soll jeden Tag (außer am Wochenende) um 11:45 Uhr das Shell-Skript aus Teilaufgabe 4 aufrufen, das testet, ob heute schon eine Archivdatei angelegt wurde.

   Die Ausgabe der Shell-Skripte soll in eine Datei `/tmp/Backup-Log.txt` angehängt werden. Wenn die Archivdatei `Backup...tar.bz2` erfolgreich erzeugt wurde, soll dieses in der Log-Datei `/tmp/Backup-Log.txt` vermerkt werden.

   Vor jedem neuen Eintrag in die Datei sollen Zeilen nach folgendem Muster (mit aktuellen Werten) in die Log-Datei `/tmp/Backup-Log.txt` eingefügt werden:

   ```
   **************************
   20.11.2013 --- 21:39:51 Uhr
   ```

   *Um die Lösung zu verstehen, hier im Vorfeld einige wichtige Informationen zur Crontabelle.*

*Jeder Auftrag / jede Zeile in der crontab besteht aus sechs Feldern. Die ersten fünf Felder werden benutzt, um den Ausführungszeitpunkt des Auftrags zu bestimmen. Im sechsten und letzten Eintrag wird das Programm, Skript oder Kommando festgelegt, das zu dem Ausführungszeitpunkt gestartet werden soll.*

- *Spalte 1: Minute (`0-59` oder `*`)*
- *Spalte 2: Stunde (`0-23` oder `*`)*
- *Spalte 3: Tag (`1-31` oder `*`)*
- *Spalte 4: Monat (`1-12`, `Jan-Dec` oder `jan-dec` oder `*`)*
- *Spalte 5: Wochentag (`0-6`, `Sun-Sat` oder `sun-sat` oder `*`)*
- *Spalte 6: Auszuführender Befehl (Programmname und Pfad)*

*Ein Eintrag in der Crontabelle darf auf keinen Fall einen Zeilenumbruch enthalten und nicht länger als 1024 Zeichen sein. Kommentare beginnen in der Crontabelle immer mit einer Raute (`#`). Es ist nicht nur möglich, einen Wert pro Zeitspalte anzugeben. Es können auch mehrere Werte pro Spalte angeben werden. Diese werden durch Kommas voneinander getrennt.*

*Zum Ausgeben und Bearbeiten der eigenen Crontabelle auf der Shell existiert der Befehl `crontab`:*

- *`crontab -l`: Die eigene crontab ausgeben.*
- *`crontab -e`: Die eigene crontab bearbeiten.*
- *`crontab -r`: Die eigene crontab löschen.*

*Der Editor, den der Befehl crontab aufruft, ist standardmäßig `vi`. Um die Crontabelle mit einem anderen Editor zu bearbeiten, muss die Shellvariable `EDITOR` erzeugt werden. Diese muss den Namen und wenn nötig noch den Pfad des bevorzugten Editors enthalten. Mit dem folgenden Befehl auf der Shell wird in Zukunft die Crontabelle immer mit dem Editor `nano` gestartet:*

```
export EDITOR=/usr/bin/nano
```

*Eine mögliche Lösung:*

```
$ export EDITOR=/usr/bin/joe
$ crontab -e
```

*Einträge in der Crontabelle passend zur Aufgabenstellung:*

```
# 1. Spalte: 15. Minute der Stunde
# 2. Spalte: 6. Stunde des Tages
# 3. Spalte: An jedem Tag des Monates
# 4. Spalte: In jedem Monate des Jahres
# 5. Spalte: An den Wochentage Montag bis Freitag
# 6. Spalte: Kommando
15 6 * * 1-5 echo -e "**********************\n`date
+%d.%m.%Y\ ---\ %X`" >> /tmp/Backup-Log.txt &&
/pfad/zu/archiv_erstellen.bat >> /tmp/Backup-Log.txt
```

```
# 1. Spalte: 45. Minute der Stunde
# 2. Spalte: 11. Stunde des Tages
# 3. Spalte: An jedem Tag des Monates
# 4. Spalte: In jedem Monate des Jahres
# 5. Spalte: An den Wochentage Montag bis Freitag
# 6. Spalte: Kommando
45 11 * * 1-5 echo -e "**********************\n`date
+%d.%m.%Y\ ---\ %X`" >> /tmp/Backup-Log.txt &&
/pfad/zu/archiv_untersuchen.bat >> /tmp/Backup-Log.txt
```

# Aufgabe 5 (Shell-Skripte)

1. Schreiben Sie ein Shell-Skript, das für eine als Argument angegebene Datei feststellt, ob die Datei existiert und ob es sich um eine ein Verzeichnis, einen symbolischen Link, einen Socket oder eine benannte Pipe (FIFO) handelt.

   - Das Skript soll das Ergebnis der Überprüfung ausgeben.

```
#!/bin/bash
#
# Skript: datei_testen.bat
#
if test -e $1 ; then

  echo "Die Datei existiert."

  if test -d $1 ; then
    echo "Die Datei ist ein Verzeichnis."
  elif test -L $1 ; then
    echo "Die Datei ist ein symbolischer Link."
  elif test -S $1 ; then
    echo "Die Datei ist ein Socket."
  elif test -p $1 ; then
    echo "Die Datei ist eine benannte Pipe (FIFO)."
  fi
fi
```

2. Erweitern Sie das Shell-Skript aus Teilaufgabe 1 dahingehend, dass wenn die als Argument angegebene Datei existiert, soll feststellt werden, ob diese ausführt werden könnte und ob schreibend darauf zugegriffen werden könne.

```
#!/bin/bash
#
# Skript: datei_testen2.bat
#
if test -e $1 ; then

  echo "Die Datei existiert."

  if test -x $1 ; then
    echo "Datei ist ausführbar"
```

```
  else
    echo "Datei ist nicht ausführbar"
  fi

  if test -w $1 ; then
    echo "Datei ist schreibbar"
  else
    echo "Datei ist nicht schreibbar"
  fi

  if test -d $1 ; then
    echo "Die Datei ist ein Verzeichnis."
  elif test -L $1 ; then
    echo "Die Datei ist ein symbolischer Link."
  elif test -S $1 ; then
    echo "Die Datei ist ein Socket."
  elif test -p $1 ; then
    echo "Die Datei ist eine benannte Pipe (FIFO)."
  fi
fi
```

3. Schreiben Sie ein Shell-Skript, das so lange auf der Kommandozeile Text einliest, bis es durch die Eingabe von `ENDE` beendet wird.

   - Die eingelesenen Daten soll das Skript in Großbuchstaben konvertieren und ausgeben.

```
#!/bin/bash
#
# Skript: einlesen.bat
#
while [ true ]
do
  read EINGABE
  if [ $EINGABE == "ENDE" ] ; then
    exit
  else
    echo $EINGABE | tr '[:lower:]' '[:upper:]'
  fi
done
```

4. Schreiben Sie ein Shell-Skript, das für alle eingeloggten Benutzer die Anzahl der laufenden Prozesse ausgibt.

5. Erweitern Sie das Shell-Skript aus Teilaufgabe 4 dahingehend, dass die Ausgabe sortiert ausgegeben wird.

   - Der Benutzer mit den meisten Prozessen soll am Anfang stehen.

6. Schreiben Sie ein Shell-Skript, das nach dem Start alle 10 Sekunden überprüft, ob eine Datei `/tmp/lock.txt` existiert.

   - Jedes Mal, nachdem das Skript das Vorhandensein der Datei überprüft hat, soll es eine entsprechende Meldung auf der Shell ausgeben.

- Sobald die Datei `/tmp/lock.txt` existiert, soll das Skript sich selbst beenden.

```
#!/bin/bash
#
# Skript: lock_testen.bat
#
while [ true ]
do
  if test -f "/tmp/lock.txt" ; then
    echo "Die Datei lock.txt ist vorhanden."
    exit 0
  else
    echo "Die Datei lock.txt ist nicht vorhanden."
  fi
  sleep 10
done
```