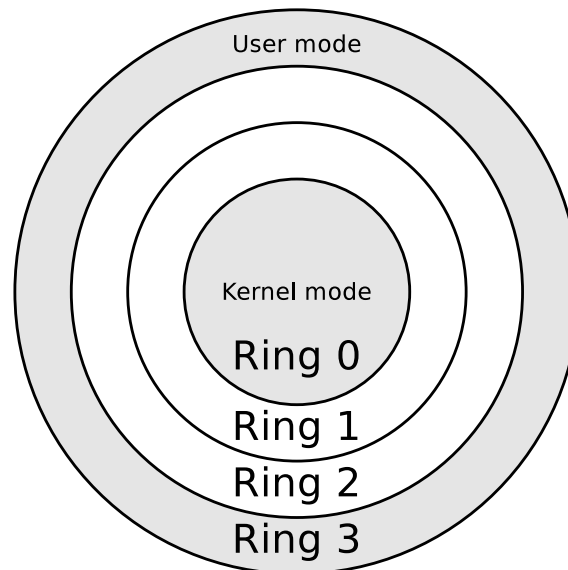


Solution of Exercise Sheet 7

Exercise 1 (System Calls)

1. x86-CPU's contain 4 privilege levels („rings“) for processes. Mark in the diagram (*clearly visible!*) the kernel mode and the user mode.



2. Name the ring to which the operating system is assigned.

In ring 0 (= kernel mode) runs the kernel.

Name the ring to which the user applications are assigned.

In ring 3 (= user mode) run the applications.

3. Name the ring to which processes are assigned that have full access to the hardware.

Processes in kernel mode (ring 0) have full access to the hardware.

4. Name a reason for the differentiation between user mode and kernel mode.

It improves stability and security.

5. Explain what a system call is.

If a user-mode process must carry out a higher privileged task (e.g. accessing hardware), it can access this the kernel via a system call. A system call is a function call in the operating system, which triggers a switch from user mode to kernel mode (\implies context switch).

6. Explain what a context switch is.

A process passes the control over the CPU to the kernel and is suspended until the request is completely processed. After the system call, the kernel passes the control over the CPU back to the user-mode process. The process continues its execution at the location, where the context switch was previously requested.

7. Name two reasons why user mode processes should not call system calls directly.

Working directly with system calls is unsafe and the portability is poor.

8. Name an alternative if user mode processes shall not call system calls directly.

Modern operating systems provide a library, which is logically located between the user mode processes and the kernel.

Exercise 2 (Processes)

1. Name the three sorts of process context information the operating system stores.

User context, hardware context and system context.

2. Name the process context information that are not stored in the process control block.

The user context, which is the allocated address space (virtual memory).

3. Explain why the process control block does not store all process context information.

Depending on the architecture, the virtual memory of each process may be several GB in size. Therefore, the user context is just too big in size to store it twice.

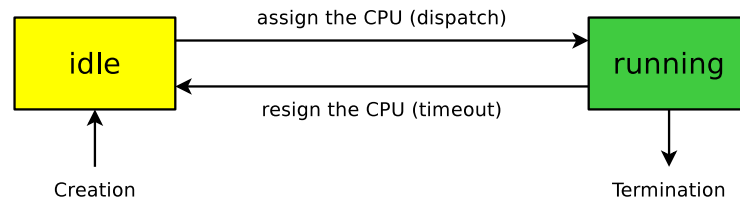
4. Explain the task of the dispatcher.

It carries out the state transitions of the processes.

5. Explain the task of the scheduler.

It specifies the execution order of the processes.

6. The process state model with 2 states is the smallest possible process model. Enter the names of the states in the diagram of the process state model with 2 states.

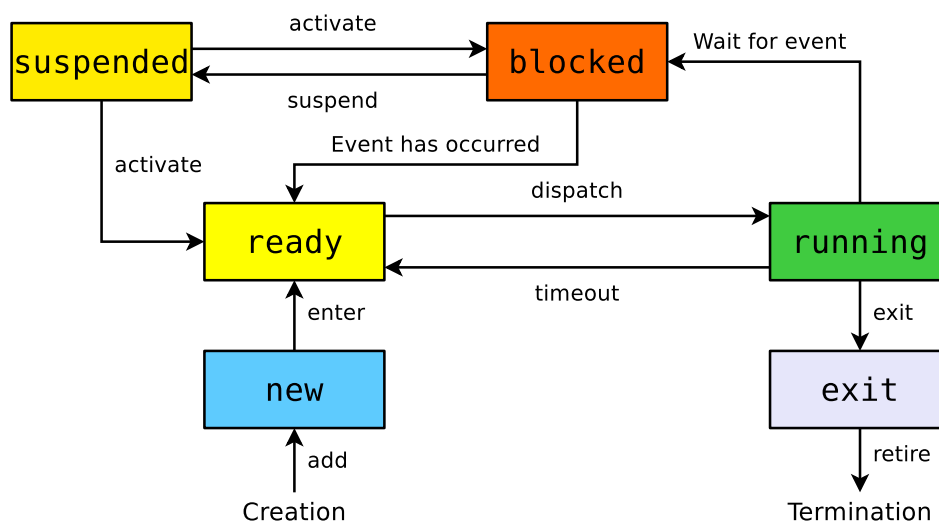


7. Does the process state model with 2 states make sense? Explain your answer shortly.

The process state model with two states assumes that all processes are ready to run at any time. This is unrealistic because almost always do processes exist, which are blocked. The idle processes must be categorized into two groups:

- *Processes, which are ready.*
- *Processes, which are blocked.*

8. Enter the names of the states in the diagram of the process state model with 6 states.



9. Explain what a zombie process is.

A zombie process has completed execution (via the system call `exit`) but its entry in the process table exists until the parent process has fetched (via the system call `wait`) the exit status (return code). Its PID cannot yet be assigned to a new process.

10. Explain the task of the process table.

For managing the processes, the operating system implements the process table. It contains for each process a record which is called process control block.

11. Give the number of status lists, the operating system manages for processes in „blocked“ state.

For each event, there is a separate list of processes waiting for this event.

12. Explain what happens if a new process is to be created, but the operating system has no more free process IDs (PID) left.

In this case, no new process can be created.

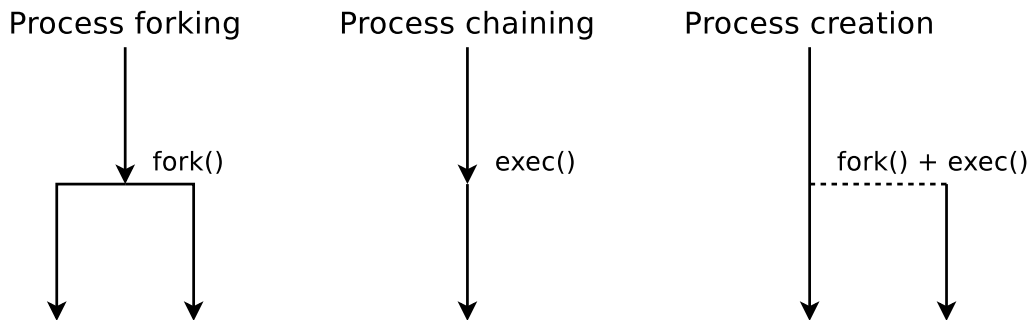
13. Describe the effect of calling the system call `fork()`.

If a process calls `fork`, an identical copy is started as a new process.

14. Describe the effect of calling the system call `exec()`.

The system call `exec` replaces a process with another one.

15. The three diagrams below show all existing ways of creating a new process. Specify for each diagram, which system call(s) are required to implement the illustrated way of process creation.



16. A parent process (PID = 75) with the characteristics, described in the table below, creates a child process (PID = 198) by using the system call `fork()`. Enter the four missing values into the table.

	Parent Process	Child Process
PPID	72	75
PID	75	198
UID	18	18
Return value of <code>fork()</code>	198	0

17. The following C source code creates a child process. Give the value of the `returnvalue` variable for the child process and for the parent process. In your answer, explain the importance of the return value in the parent process.

```

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 }
```

In the child process, `fork()` has the return value 0.

In the parent process `fork()` has a positive return value. The return value then is equal to the PID of the newly created child process. This return value allows the parent process to identify the child process.

18. Describe what `init` is and what its task is.

`init` is the first process in Linux/UNIX. It has PID 1. All running processes originate from `init`. `init` is the father of all processes.

19. Name the differences of a child process from the parent process shortly after its creation.

The PID, the PPID, and the memory areas.

20. Describe the effect, when a parent process is terminated before the child process.

If a parent process terminates before the child process, it gets `init` as the new parent process assigned. Orphaned processes are always adopted by `init`. The PPID of the child process then becomes value 1.

21. Describe what data the Text Segment contains.

It contains the program code (machine code).

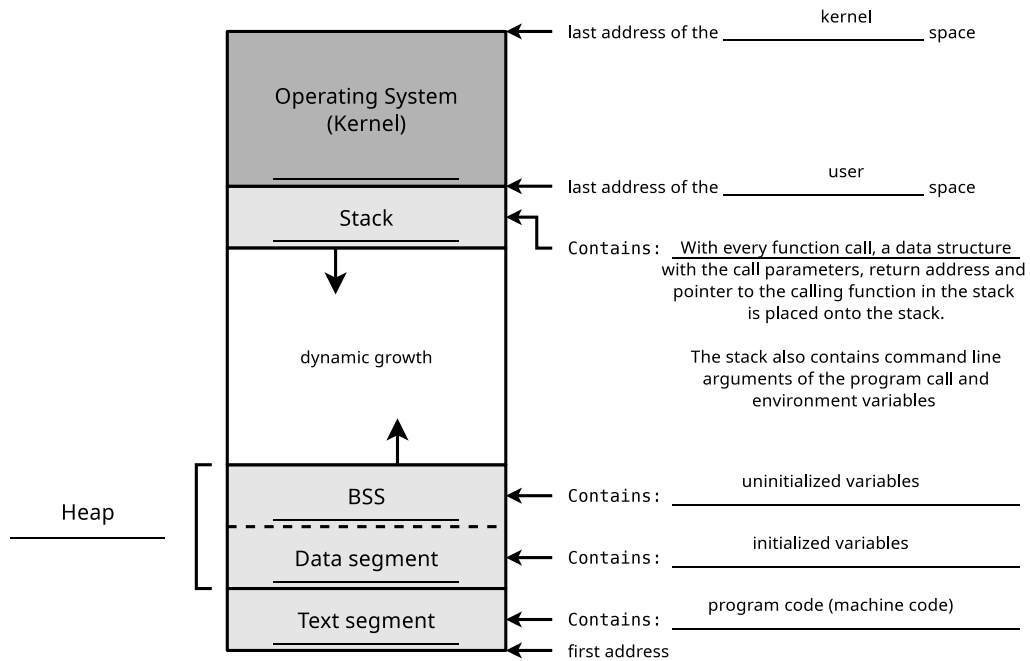
22. Describe what data the Heap contains.

Constants and variables, which get values assigned in global declarations (outside of functions).

23. Describe what data the Stack contains.

Command line arguments, environment variables of the program call, call parameters and return address of functions, local variables of functions.

24. The figure shows the structure of a UNIX process in memory. Fill in the missing labels (technical terms) of the process-related data and the missing information about the content of this data.



Exercise 3 (Information about Processes in the Operating System)

The output of the `ps` command contains helpful information about the processes in the operating system.

```
$ 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. Explain the content of the column UID.

User ID of the owner of the process.

2. Explain the content of the column PID.

The unique process ID.

3. Explain the content of the column PPID.

The unique process ID of the parent process.

4. Explain the content of the column **C**.

CPU utilization of the process in percent.

5. Explain the content of the column **SZ**.

virtual process size = text segment, heap and stack.

6. Explain the content of the column **RSS**.

Resident Set Size = occupied physical memory (without swap) in kB.

7. Explain the content of the column **PSR**.

Number of the CPU core assigned to the process.

8. Explain the content of the column **STIME**.

Start time of the process

9. Explain the content of the column **TTY**.

Teletypewriter = control terminal. Usually a virtual device: pts (pseudo terminal slave)

10. Explain the content of the column **TIME**.

Consumed CPU time of the process (HH:MM:SS).

11. Name the parent process of the process that has printed this overview of the processes in the command-line interface.

*The **bash** process with PID 3568 is the parent process of the **ps** process with PID 94414.*

Exercise 4 (Time-based Command Execution, Control Structures, Archiving)

1. Program a shell script, which reads two numbers as command line arguments. The script should check whether the numbers are identical, and print out the result of the check.

```
1 #!/bin/bash
2 #
3 # Skript: vergleich.bat
4 #
5 echo "Geben sie zwei Zahlen ein"
6 read -p "Zahl1:" zahl1
```

```
7 read -p "Zahl2:" zahl2
8 if [ $zahl1 -eq $zahl2 ] ; then
9     echo "Die beiden Zahlen sind gleich groß."
10 else
11     echo "Die beiden Zahlen sind nicht gleich groß."
12 fi
```

2. Extend the shell script in a way that if the numbers are not identical, it is checked, which one of the two numbers is the larger one. The result of the check should be printed out.

```
1 #!/bin/bash
2 #
3 # Skript: vergleich2.bat
4 #
5 echo "Geben sie zwei Zahlen ein"
6 read -p "Zahl1:" zahl1
7 read -p "Zahl2:" zahl2
8 if [ $zahl1 -eq $zahl2 ] ; then
9     echo "Die beiden Zahlen sind gleich groß."
10 elif [ $zahl1 -gt $zahl2 ]; then
11     echo "Zahl 1 mit Wert $zahl1 ist größer."
12 else
13     echo "Zahl 2 mit Wert $zahl2 ist größer."
14 fi
```

3. Program a shell script, which creates a backup of a directory of your choice. The script should create an archive file with the file extension `.tar.bz2` from the directory. The archive file should be stored in the directory `/tmp`. The name of the archive file should correspond to the following naming scheme:

Backup_<USERNAME>_<YEAR>_<MONTH>_<DAY>.tar.bz2

The fields <USERNAME>, <YEAR>, <MONTH> and <DAY> should be replaced by the current values.

```
1 #!/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. Program a shell script, which checks if already today an archive file was created according to the naming scheme of subtask 3. The result of the check should be printed out in the shell.

```
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. Write two cron jobs. The first cron job should execute at 6:15 am on every day (except on weekends) the shell script from subtask 3, which creates the archive file with the backup.

The second cron job should execute at 11:45 am on every day (except on weekends) the shell script from subtask 4, which checks, whether already today an archive file was created.

The output from the shell scripts should be appended to a file /tmp/Backup-Log.txt. If the archive file Backup...tar.bz2 has been created successfully, this should be noted in the log file /tmp/Backup-Log.txt.

Before each new entry in the file, lines according to the following pattern (with current values) should be inserted into the log file /tmp/Backup-Log.txt.

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

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 angegeben 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 Monats
# 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 Monats
# 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
```

Exercise 5 (Shell Scripts)

1. Program a shell script, which checks for a file, which is specified as an argument, whether it exists and if it is a file, a directory, a symbolic link, a socket or a named pipe.
 - The script should print out the result of the check.

```
1 #!/bin/bash
2 #
3 # Skript: datei_testen.bat
4 #
5 if test -e $1 ; then
6
7     echo "Die Datei existiert."
8
9     if test -d $1 ; then
10         echo "Die Datei ist ein Verzeichnis."
11     elif test -L $1 ; then
12         echo "Die Datei ist ein symbolischer Link."
13     elif test -S $1 ; then
14         echo "Die Datei ist ein Socket."
15     elif test -p $1 ; then
16         echo "Die Datei ist eine benannte Pipe (FIFO)."
17     fi
18 fi
```

2. Extend the shell script from subtask 1 in a way that if the file, which is specified as an argument, exists, it is checked, if the file could be executed and if write access would be possible.

```
1 #!/bin/bash
2 #
3 # Skript: datei_testen2.bat
4 #
5 if test -e $1 ; then
6
7     echo "Die Datei existiert."
8
9     if test -x $1 ; then
10         echo "Datei ist ausführbar"
11     else
12         echo "Datei ist nicht ausführbar"
13     fi
14
15     if test -w $1 ; then
16         echo "Datei ist schreibbar"
17     else
18         echo "Datei ist nicht schreibbar"
19     fi
20
21     if test -d $1 ; then
22         echo "Die Datei ist ein Verzeichnis."
23     elif test -L $1 ; then
```

```
24 echo "Die Datei ist ein symbolischer Link."  
25 elif test -S $1 ; then  
26 echo "Die Datei ist ein Socket."  
27 elif test -p $1 ; then  
28 echo "Die Datei ist eine benannte Pipe (FIFO)."  
29 fi  
30 fi
```

3. Program a shell script, which reads text from the command line, until the string END occurs.

- The script should convert the text, which is read in from the command line, to uppercase.

```
1 #!/bin/bash  
2 #  
3 # Skript: einlesen.bat  
4 #  
5 while [ true ]  
6 do  
7 read EINGABE  
8 if [ $EINGABE == "ENDE" ] ; then  
9 exit  
10 else  
11 echo $EINGABE | tr '[:lower:]' '[:upper:]'  
12 fi  
13 done
```

4. Program a shell script, which prints out the number of running processes for all logged in users.

5. Extend the shell script from subtask 4 in a way that that the output is sorted.

- The user with most processes should stand at the beginning.

6. Program a shell script, which checks after start every 10 seconds, if a file /tmp/lock.txt exists.

- Each time after the script has checked the existence of the file, it should output an appropriate message on the shell.
- Once the file /tmp/lock.txt exists, the script should terminate itself.

```
1 #!/bin/bash  
2 #  
3 # Skript: lock_testen.bat  
4 #  
5 while [ true ]  
6 do  
7 if test -f "/tmp/lock.txt" ; then  
8 echo "Die Datei lock.txt ist vorhanden."  
9 exit 0  
10 else  
11 echo "Die Datei lock.txt ist nicht vorhanden."  
12 fi
```

```
13  sleep 10  
14  done
```