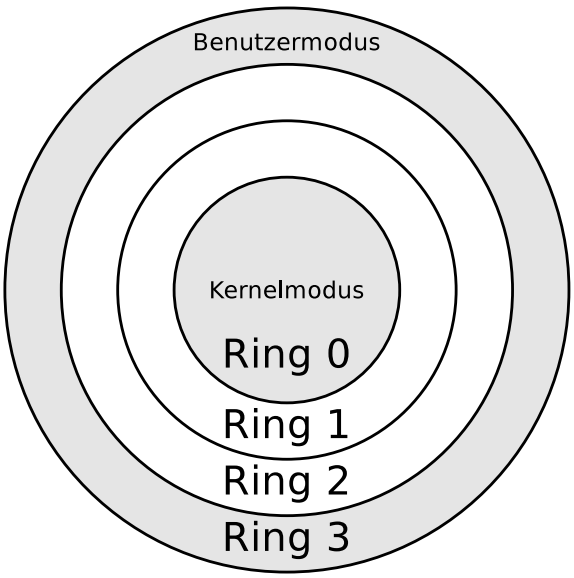
Lösung von Übungsblatt 4

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, in 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. Beschreiben Sie die Alternative, wenn Prozesse im Benutzermodus nicht direkt Systemaufrufe aufrufen sollen.

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

Aufgabe 2 (Prozesse)

1. Im Mehrprogramm-Betrieb (Multitasking) wechseln sich die Prozesse ständig ab. Beschreiben Sie, wie es möglich ist, dass ein Prozess die Ausführung in demselben Zustand fortsetzt, in dem er unterbrochen wurde.

   Das Betriebssystem implementiert einen Prozesskontrollblock für jeden Prozess. Wenn eine Unterbrechung oder ein Systemaufruf erfolgt, werden der Hardwarekontext und der Systemkontext im Prozesskontrollblock gespeichert. Sobald dem Prozess die CPU wieder zugewiesen wird, werden der Hardwarekontext und der Systemkontext aus seinem Prozesskontrollblock wiederhergestellt.

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

   Benutzerkontext, Hardwarekontext und Systemkontext.

3. Nennen Sie die Prozesskontextinformation(en), die nicht im Prozesskontrollblock gespeichert sind.

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

4. Warum sind nicht alle Prozesskontextinformationen im Prozesskontrollblock gespeichert?

   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.

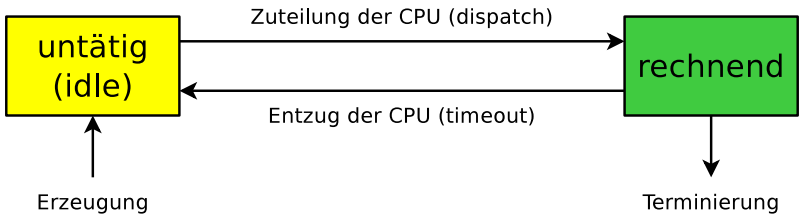
5. Beschreiben Sie die Aufgabe des Dispatchers.

   Aufgabe des Dispatchers ist die Umsetzung der Zustandsübergänge der Prozesse.

6. Beschreiben Sie die Aufgabe des Schedulers.

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

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

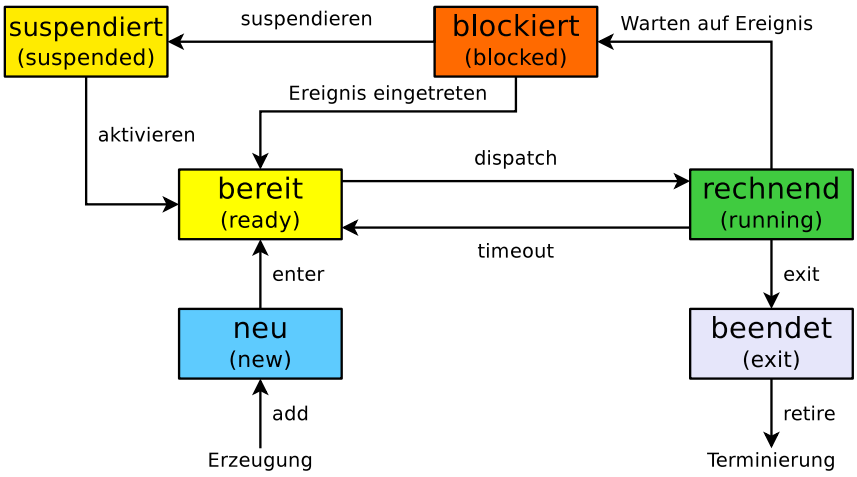


8. Ist das 2-Zustands-Prozessmodell sinnvoll? Begründen Sie kurz ihre Antwort.

   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.*

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.

11. Beschreiben Sie die Aufgabe der Prozesstabelle.

    In der Prozesstabelle speichert das Betriebssystem alle Informationen zu allen Prozessen. Einzige Ausnahme: Der Inhalt des Adressraums (Benutzerkontext) wird nicht in der Prozesstabelle gespeichert. Die Prozesstabelle wird als Array oder verkettete Liste realisiert. Für jeden Prozess existiert eine eigene Tabelle, der Prozesskontrollblock.

12. Geben sie an, wie viele Zustandslisten für Prozesse im Zustand „blockiert“ das Betriebssystem verwaltet.

    Für jedes Ereignis existiert eine eigene Liste 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 (PID) mehr gibt.

    Dann kann kein neuer Prozess erstellt werden.

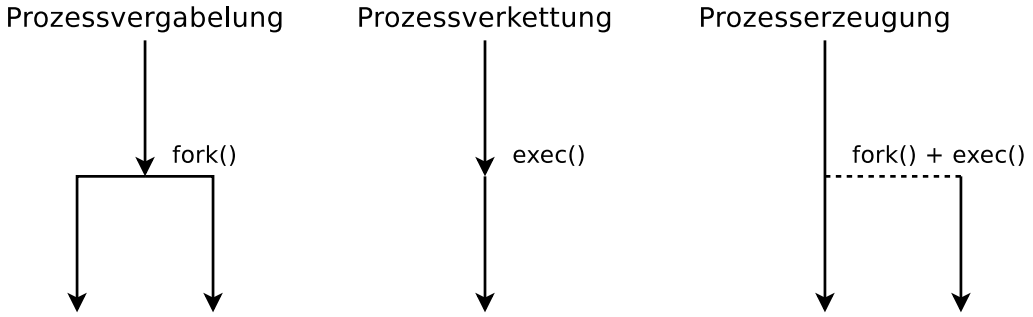
14. Beschreiben Sie was der Systemaufruf `fork()` macht.

    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 ist/sind, um die gezeigte Prozesserzeugung zu realisieren.



17. Beschreiben Sie, was passiert, wenn Sie dieses Programm ausführen:

```
while(true){
    fork()
}
```

Es ist eine Fork-Bombe. Das Programm erstellt Kopien des Prozesses in einer Endlosschleife. Das Programm erstellt so lange Kopien des Prozesses, bis es keinen freien Speicher mehr gibt und das System unbenutzbar ist.

18. 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.

19. Beschreiben Sie was `init` ist und was seine Aufgabe 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. Beschreiben Sie was einen Kindprozess kurz nach der Erzeugung vom Elternprozess unterscheidet.

Die PID 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 die Daten, die das Textsegment enthält.

Den ausführbaren Programmcode (Maschinencode).

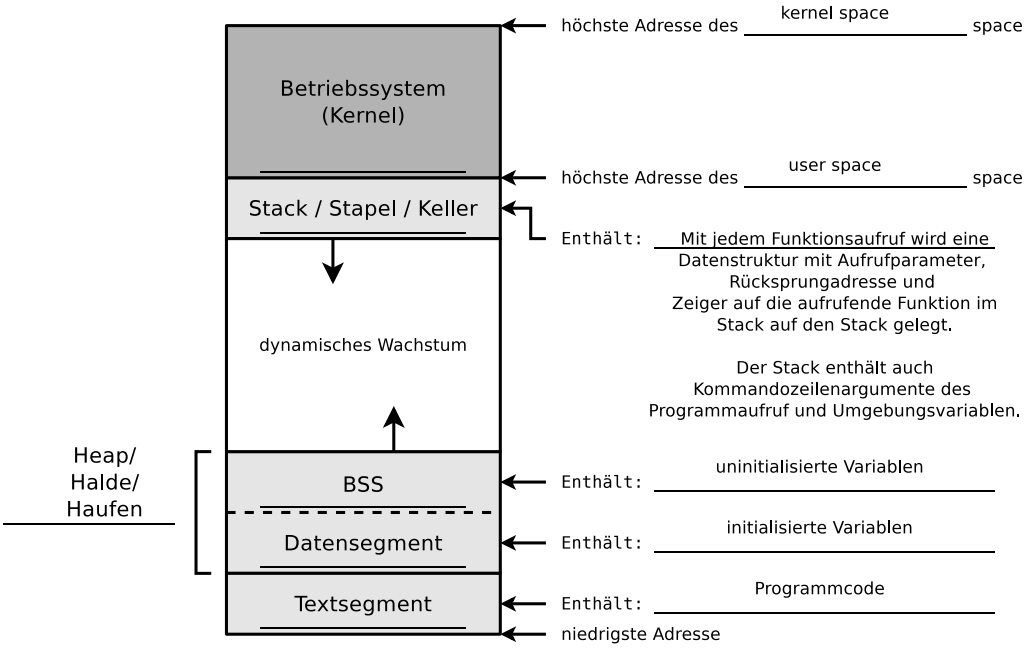
23. Nennen sie die Daten, die der Heap enthält.

Konstanten und Variablen die außerhalb von Funktionen deklariert sind.

24. Nennen sie die Daten, die der Stack enthält.

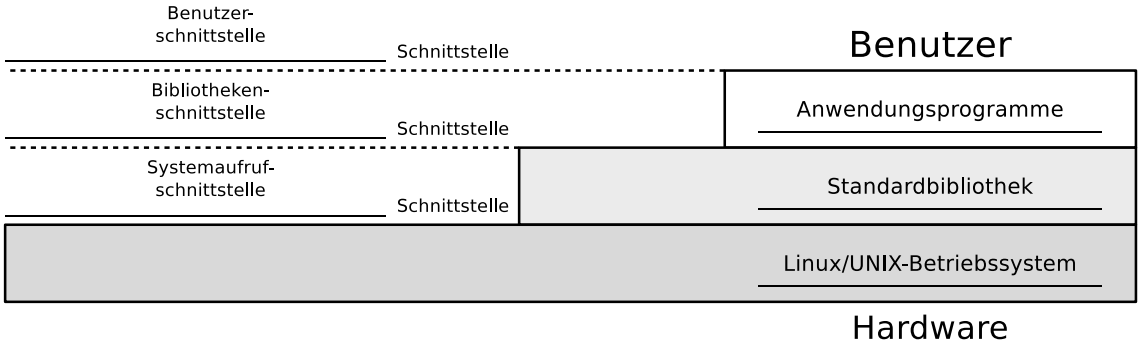
Kommandozeilenargumente des Programmaufrufs, Umgebungsvariablen, Aufrufparameter und Rücksprungadressen der Funktionen, lokale Variablen der Funktionen.

25. 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 (Schnittstellen des Betriebssystems)

Die Benutzer können nicht direkt mit der Hardware kommunizieren. Zwischen der Hardware und den Benutzern können drei Schichten unterschieden werden. Jede dieser Schichten implementiert eine Schnittstelle. Nennen Sie die Schichten und die Schnittstellen in der Abbildung.



Aufgabe 4 (Kontrollstrukturen)

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
```

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
```

```
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. 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.

```
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. 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 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."
  else
    echo "Die Datei lock.txt ist nicht vorhanden."
  fi
  sleep 10
done
```