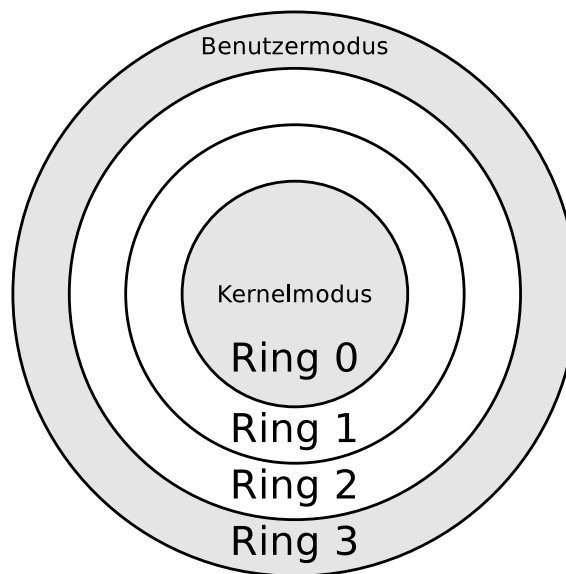


Lösung von Übungsblatt 4

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. In welchem Ring befindet sich der Betriebssystemkern?
In Ring 0 (= Kernelmodus) läuft der Betriebssystemkern.
3. In welchem Ring befinden sich die Anwendungen der Benutzer?
In Ring 3 (= Benutzermodus) laufen die Anwendungen.
4. Prozesse in welchem Ring haben vollen Zugriff auf die Hardware?
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. Was ist ein Systemaufruf?
Ein Systemaufruf ist ein Funktionsaufruf im Betriebssystem, der einen Sprung vom Benutzermodus in den Kernelmodus auslöst (\implies Moduswechsel)
7. Was ist ein Moduswechsel?

Ein Prozess gibt die Kontrolle über die CPU an den Kernel ab und wird unterbrochen bis die Anfrage fertig bearbeitet ist. 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. Welche Alternative gibt es, wenn Prozesse im Benutzermodus nicht direkt Systemaufrufe aufrufen sollen?

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. Welche drei Arten Prozesskontextinformationen speichert das Betriebssystem?

Benutzerkontext, Hardwarekontext und Systemkontext.

2. Welche Prozesskontextinformationen sind nicht im Prozesskontrollblock gespeichert?

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

3. 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.

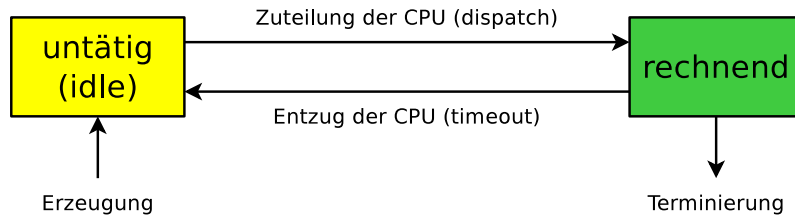
4. Was ist die Aufgabe des Dispatchers?

Aufgabe des Dispatchers ist die Umsetzung der Zustandsübergänge der Prozesse.

5. Was ist die Aufgabe des Schedulers?

Er legt die Ausführungsreihenfolge der Prozesse mit einem Scheduling-Algorithmus 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.

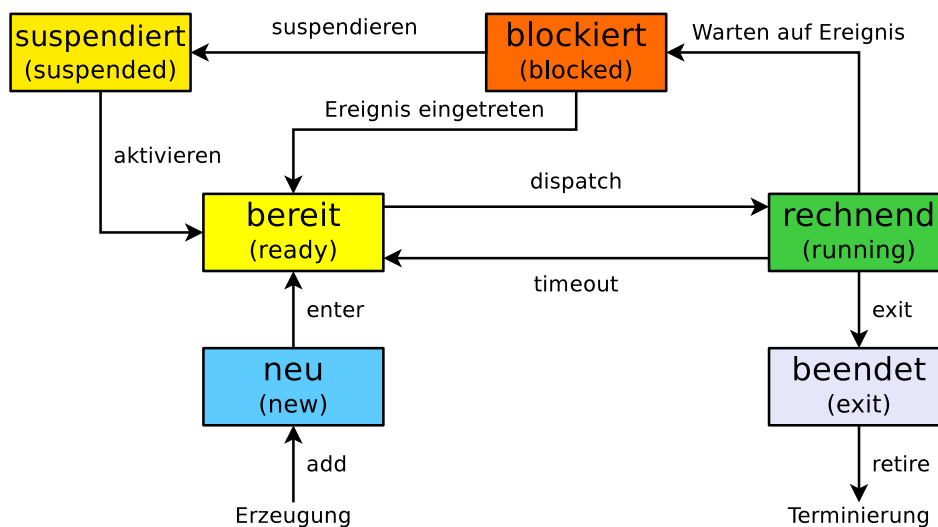


7. Ist das 2-Zustands-Prozessmodell sinnvoll? Begründen Sie kurz ihre Antwort.

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



9. Was ist ein Zombie-Prozess?

Ein beendeter Prozess, der noch in der Prozesstabelle auftaucht. Seine PID kann noch nicht an einen neuen Prozess vergeben werden.

10. Welche Aufgabe hat die 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.

11. Wie viele Zustandslisten für Prozesse im Zustand „blockiert“ verwaltet das Betriebssystem?

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

12. 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.

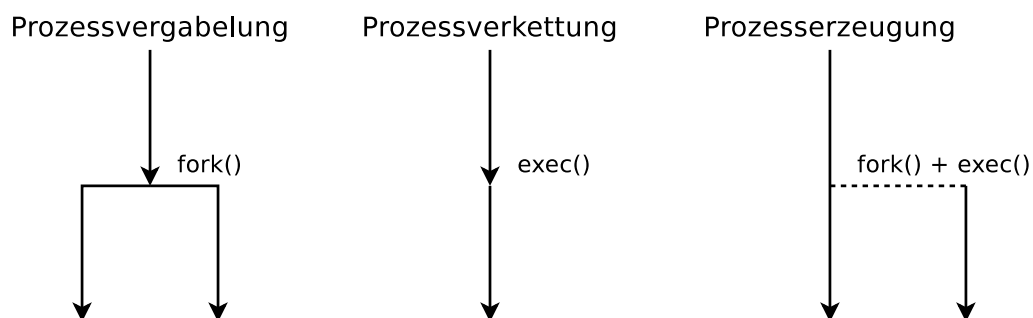
13. Was macht der Systemaufruf `fork()`?

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

14. Was macht der Systemaufruf `exec()`?

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

15. 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 Prozesserschöpfung zu realisieren.



16. 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 <code>fork()</code>	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.

17. Was ist `init` und was ist seine Aufgabe?

`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.

18. Was unterscheidet einen Kindprozess vom Elternprozess kurz nach der Erzeugung?

Die PID und die Speicherbereiche.

19. 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.

20. Welche Daten enthält das Textsegment?

Den ausführbaren Programmcode (Maschinencode).

21. Welche Daten enthält der Heap?

Konstanten und Variablen die außerhalb von Funktionen deklariert sind.

22. Welche Daten enthält der Stack?

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

Aufgabe 3 (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.

```
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. 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.

```
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
```

Aufgabe 4 (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.

```
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. Erweitern Sie das Shell-Skript aus Teilaufgabe 1 dahingehend, dass wenn die als Argument angegebene Datei existiert, soll festgestellt werden, ob diese ausgeführt 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.

```
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. 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.

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