

## Aufgabe 1

- a) Die Zustände *idle*, *runnable*, *running* und *sleeping* entsprechen den Zuständen *idle*, *ready*, *running* und *blocked* aus der Vorlesung. Der Zustand *traced* entspricht in etwa dem Zustand *blocked* aus der Vorlesung. Der Prozeß wartet auf einer Resource des Prozesses selbst und die Verfügbarkeit dieser Resource kann durch den Benutzer über Signale kontrolliert wird.

Der Zustand *zombie* hat keine Entsprechung im vereinfachten Prozeßzustandsmodell. Dieser Zustand existiert in erster Linie zum Erhalten von Prozessinformationen beim Terminieren des Prozesses (wie z.B. Exit-Code, Ursache der Terminierung, PID des Prozesses) bis diese vom System oder anderen Prozessen nicht mehr benötigt werden.

- b) Das System ist gesund. Ein UNIX-Rechner bietet typischerweise viele Dienste (mail, http, rlogin, X-Window-Oberfläche,..) von denen die wenigsten ununterbrochen durch Benutzerinteraktion in Benutzung sind. Diese Dienste sind gestartet und *warten* darauf benutzt zu werden. Ein sauber programmierter Dienst verbringt seine Wartezeit im Prozeßzustand *sleeping* oder *blocked*. Dadurch wird erreicht, daß die wenigen laufenden Prozesse kurze Antwortzeiten haben. Ein System mit vielen laufenden Prozessen steht unter Last und hat schlechte Antwortzeiten.

- c) Wir modellieren das Programm durch folgende Anweisungen:

```
read a; read b; read c; s=a+b+c; write s; exit
```

Es ergibt sich folgende Abfolge von Zuständen.

*idle*, (*runnable*, *running*)<sup>+</sup>

[read a] ⇒ *sleeping*

[Ereignis:Benutzereingabe] ⇒ (*runnable*, *running*)<sup>+</sup>

[read b] ⇒ *sleeping*

[Ereignis:Benutzereingabe] ⇒ (*runnable*, *running*)<sup>+</sup>

[read c] ⇒ *sleeping*

[Ereignis:Benutzereingabe] ⇒ (*runnable*, *running*)<sup>+</sup>

[s=a+b+c] *runnable*, (*runnable*, *running*)<sup>\*</sup>

[write s] ⇒ *sleeping*

[Ereignis: Ausgabe abgeschlossen] ⇒ (*runnable*, *running*)<sup>+</sup>

[exit] ⇒ *zombie*

[Ereignis: Vaterprozeß registriert das Terminieren] ⇒ Prozeß wird aus dem System entfernt.

Dabei bedeutet  $(a, b)^*$  eine Sequenz aus beliebig vielen (0,1,2,...) Wiederholungen von  $a, b$ . Die Notation  $(a, b)^+$  bedeutet eine oder beliebig viele Wiederholungen.

## Aufgabe 2

- a) Wichtigste Unterschiede zwischen Programm und Prozeß:

Programm	Prozeß
Algorithmus, Folge von Anweisungen	<u>eine</u> Ausführung eines Programms
passiv, zustandslos	aktiv, Zustand = Prozeßkontext (Programmzähler, Register, Programmvariablen,...)
kein exaktes Timing, lediglich die Reihenfolge der Anweisungen festgelegt	exaktes Timing, welches vom Scheduler berechnet wird.
kann immer wieder neu gestartet werden	„lebt“ nur einmal

- b) Diese Trennung wurde eingeführt um kritischen Code (Ansteuerung von Hardware/Geräten und sicherheitsrelevanter Code) dem Einfluß des Benutzers zu entziehen.

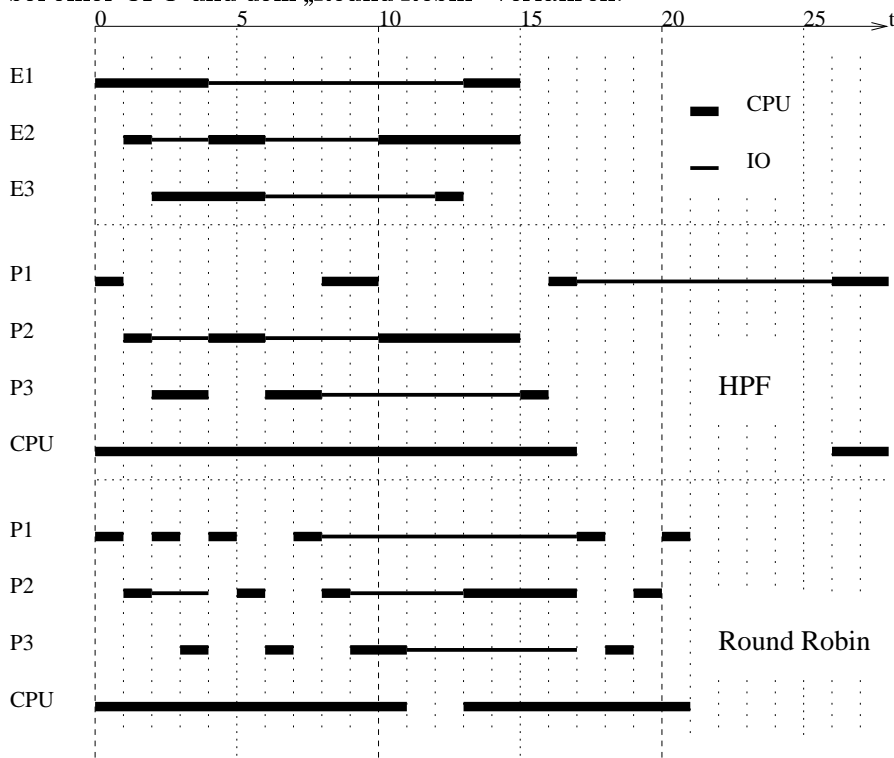
Vorteile	Nachteile
Stabilität des Betriebssystemkerns	Zeitbedarf zum Umschalten des Modus
klare Trennung der Zuständigkeiten: Gerätetreiber und Anwendungsprogramm	Treiberprogrammierung erschwert, kein einfacher Zugriff

Moderne CPUs bieten Hardwareunterstützung zur Realisierung dieses Konzepts, z.B. Intel x86 CPUs im „Protected Mode“.

### Aufgabe 3

Die Aufgabenstellung ist leider nicht eindeutig. Daher machen wir folgende zusätzliche Annahmen: der Scheduler wird zu den Zeitpunkte 0,1,2,... aktiv und bestimmt zum Zeitpunkt  $t$  welcher Prozess die CPU für die „Zeitscheibe“  $(t, t + 1)$  zugeteilt bekommt. Sämtliche Ressourcenanforderungen in dieser Zeitscheibe seien zur Zeit  $t$  bekannt. Beim Round-Robin Verfahren nehmen wir zusätzlich an, daß ein Prozeß, der neu im System ist oder die Blockierung einer E/A-Operation verläßt, vor einem Prozeß der gerade in Ausführung war in die Warteschlange eingereiht wird.

Das folgende Bild zeigt die Zuteilung der CPU bei 1. Vorhandensein von 3 CPUs (2 würden zufällig auch reichen), 2. bei einer CPU und der Strategie „Highest Priority First“ sowie 3. bei einer CPU und dem „Round Robin“-Verfahren.



- a) aus dem oberen Drittel des Bildes kann man die Zeitpunkte der frühest möglichen Terminierung ablesen:  $t_1 = 15, t_2 = 15, t_3 = 13$ .

- b) Im mittleren Teil der Abbildung ist die CPU-Zuteilung bei Verwendung der Strategie „Highest Priority First“ zu sehen. Diese ist anhand des Bildes nachzuvollziehbar, da nur die Anforderungen der Prozesse, eventuelle Blockierung und die Prioritäten zu berücksichtigen sind. Auffällig ist hier: der Prozeß  $P_2$  mit der höchsten Priorität bekommt die

CPU genau so zugeteilt, wie es im Programm  $E_2$  vorgesehen ist. Als Zeitpunkte der Terminierung ergeben sich hier:  $t_1 = 23$ ,  $t_2 = 15$ ,  $t_3 = 16$ .

- c) Die CPU-Zuteilung durch das „Round Robin“-Verfahren ist nicht unmittelbar anhand der Abbildung (unteres Drittel) nachzuvollziehen, da zusätzlich zu den Anforderungen der Prozesse auch noch die ready-Warteschlange berücksichtigt werden muß. In der folgenden Tabelle bedeutet  $Q_{ready}$  der Inhalt der Ready-Warteschlange (links Eingang, rechts Ausgang) zu Beginn der jeweiligen Zeitscheibe,  $Blocked$  die Menge der blockierten Prozesse.

$t$	$Q_{ready}$	CPU	Blocked
(0,1)	$[P_1]$	$P_1$	$\{\}$
(1,2)	$[P_1, P_2]$	$P_2$	$\{\}$
(2,3)	$[P_3, P_1]$	$P_1$	$\{P_2\}$
(3,4)	$[P_1, P_3]$	$P_3$	$\{P_2\}$
(4,5)	$[P_3, P_2, P_1]$	$P_1$	$\{\}$
(5,6)	$[P_1, P_3, P_2]$	$P_2$	$\{\}$
(6,7)	$[P_2, P_1, P_3]$	$P_3$	$\{\}$
(7,8)	$[P_3, P_2, P_1]$	$P_1$	$\{\}$
(8,9)	$[P_3, P_2]$	$P_2$	$\{P_1\}$
(9,10)	$[P_3]$	$P_3$	$\{P_1, P_2\}$
(10,11)	$[P_3]$	$P_3$	$\{P_1, P_2\}$
(11,12)	$[\ ]$	–	$\{P_1, P_2, P_3\}$
(12,13)	$[\ ]$	–	$\{P_1, P_2, P_3\}$
(13,14)	$[P_2]$	$P_2$	$\{P_1, P_3\}$
(14,15)	$[P_2]$	$P_2$	$\{P_1, P_3\}$
(15,16)	$[P_2]$	$P_2$	$\{P_1, P_3\}$
(16,17)	$[P_2]$	$P_2$	$\{P_1, P_3\}$
(17,18)	$[P_2, P_3, P_1]^*$	$P_1$	$\{\}$
(18,19)	$[P_1, P_2, P_3]$	$P_3$	$\{\}$
(19,20)	$[P_1, P_2]$	$P_2$	$\{\}$
(20,21)	$[P_1]$	$P_1$	$\{\}$
(21,22)	$[\ ]$	–	$\{\}$

\*: Die Reihenfolge der Prozesse  $P_1$  und  $P_3$  in der Warteschlange zum Zeitpunkt  $t = 17$  ist nicht eindeutig. Es wurde eine der beiden Alternativen gewählt.

Als Zeitpunkte für die Terminierung ergibt sich  $t_1 = 21$ ,  $t_2 = 20$ ,  $t_3 = 19$ .

#### Aufgabe 4

- a) Java Realisierung eines Semaphors mit Hilfe von Monitoren:

```
public class Semaphore {
    int ctr;
    public Semaphore(int initial) {
        ctr = initial;
    }
    synchronized public void P() throws InterruptedException {
        ctr--;
        if (ctr < 0) wait();
    }
    synchronized public void V() {
        ctr++;
        if (ctr <= 0) notify();
    }
}
```

- b) Das Konzept der UNIX-Signale entspricht noch am ehesten den Monitoren. Die Funktion `pause()` ähnelt der `wait()`-Funktion des Monitoren. Die `signal()` Funktionen sind sogar namentlich gleich. Es sind jedoch deutliche Unterschiede zu erkennen:
- Beim Versenden eines UNIX-Signals muß der Empfängerprozeß bekannt sein. Wartelisten stehen nicht implizit zur Verfügung und müssen mit anderen Mitteln nachgebildet werden. Monitore übernehmen im Gegensatz dazu selbständig die Auswahl eines zu weckenden Prozesses.
  - Die begrenzte Anzahl von frei benutzbaren Signalen begrenzt die Anzahl der Synchronisationsobjekte. Die Anzahl von Monitor-Objekten ist praktisch unbegrenzt.