

### Klassen von Schedulingverfahren:

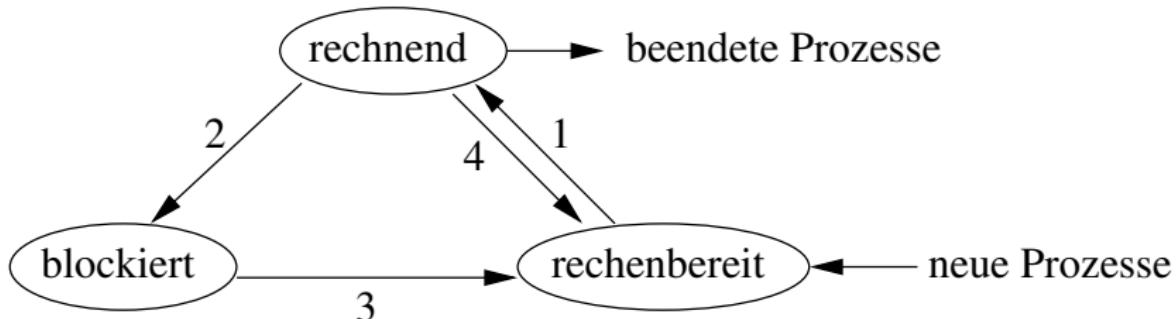
- ① deterministisch ↔ nicht deterministisch

auch: Offline- versus Online-Verfahren

Beim deterministischen Scheduling ist der Rechenzeitbedarf einer Anwendung bzw. eine Schätzung für die Rechenzeit bekannt. Häufig werden auch alle Ankunftszeiten der Jobs im voraus als bekannt angenommen.

- ② verdrängend ↔ nicht verdrängend

(preemptive ↔ non-preemptive)



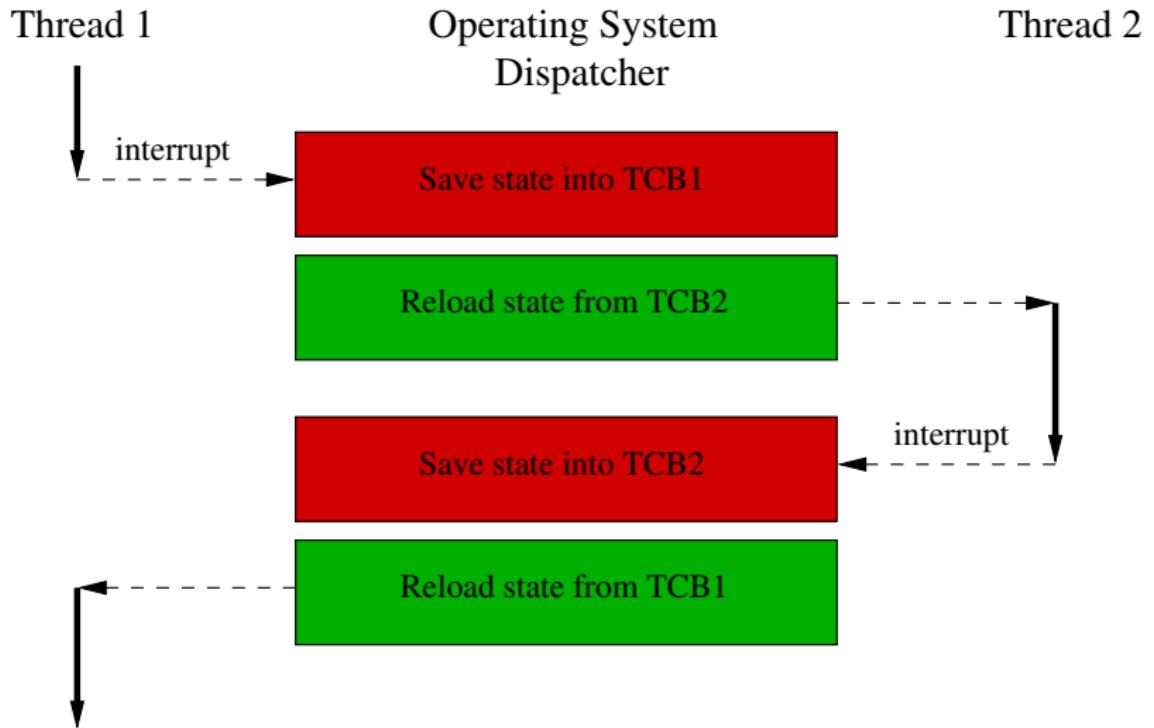
Bei verdrängenden Verfahren darf ein laufender Prozeß unterbrochen werden (z.B. Timesharing, Prioritäten-Verfahren mit Unterbrechung).

Bei nicht-verdrängenden Verfahren bestehen asynchrone Unterbrechungen nur aus Device Interrupts. Nach der Unterbrechungsbehandlung rechnet der unterbrochene Prozeß weiter.

## Betriebssystemkomponenten

### Definition 16:

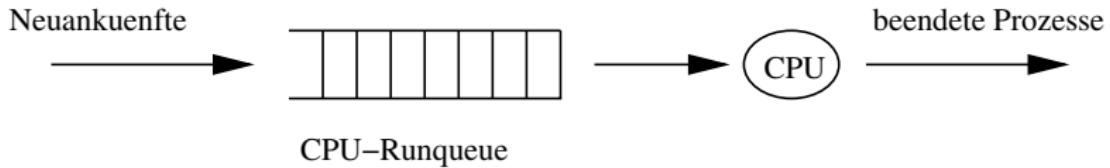
- ① Die **Scheduler-Komponente** implementiert die Schedulingstrategie des Systems und bestimmt, welcher Prozeß als nächster rechnen darf.
- ② Die **Dispatcher-Komponente** ist für den Prozeßwechsel zuständig: Der Zustand des bisher laufenden Prozesses muß gesichert werden und der neu ausgewählte Prozeß muß *rechenfähig* gesetzt werden.  
(analog für den Threadwechsel)



TCB: Thread Control Block

## FIFO (First In First Out)

auch: FCFS (First Come First Served)



### Eigenschaften:

- + non-preemptive
- + einfache Verwaltung
- + fair

Beispiel:

Prozeß	Ankunftzeit	Rechenzeitbedarf
$P_1$	0	24
$P_2$	1	3
$P_3$	2	3

## Bezeichnungen

$W_i := \text{Startzeit} - \text{Ankunftzeit}$  Wartezeit von Prozeß  $P_i$   
(nicht-verdrängende Verfahren)

$t_i :=$  Rechenzeit von Prozeß  $P_i$

$V_i := W_i + t_i$  Verweilzeit (Laufzeit, Ausführungszeit) von  
Prozeß  $P_i$

Im Falle von Dialogbetrieb ist die Reaktions- bzw. Antwortzeit relevant:

Die Aufenthaltszeit eines Auftrags im System, bis zum Eintreffen des ersten Resultats an einer Ausgabeschnittstelle, heißt Reaktionszeit.

**Beispiel:** Zeit bis zur Ausgabe des ersten Zeichens auf dem Bildschirm

## Gütekriterien für Scheduling-Verfahren

### ① CPU-Auslastung

Optimierungsstrategien:

- Ausnutzen von I/O-Wartezeiten durch Multitasking
- Mehrprozessorssysteme  $\Rightarrow$  Lastausgleich

### ② $\bar{W}$ mittlere Wartezeit

Ziel:  $\bar{W}$  minimieren

### ③ Dialogbetrieb

$\Rightarrow$  Reaktionszeit/Antwortzeit minimieren

### ④ Fairness-Prinzip: Prozesse mit langer Rechenzeit sollen länger warten als kurze Prozesse!

$$\Rightarrow \frac{1}{n} \sum_{i=1}^n \frac{W_i}{t_i} \text{ minimieren}$$

aber: Kein Prozeß darf verhungern!

## Echtzeitsysteme

**Ziel:** Einhalten von Fristen

$r_i$  (ready time): frühester Zeitpunkt, zu dem der Prozessor an  $P_i$  zugeteilt werden darf.

$t_i$  : Schätzung für maximale Ausführungszeit

$d_i$  (deadline) : Zeitpunkt, an dem die Ausführung von  $P_i$  beendet sein muß

$s_i$  : Startzeit von Prozeß  $P_i$

Zeitbedingungen:

$$r_i \leq s_i \quad \wedge \quad s_i + t_i \leq d_i$$

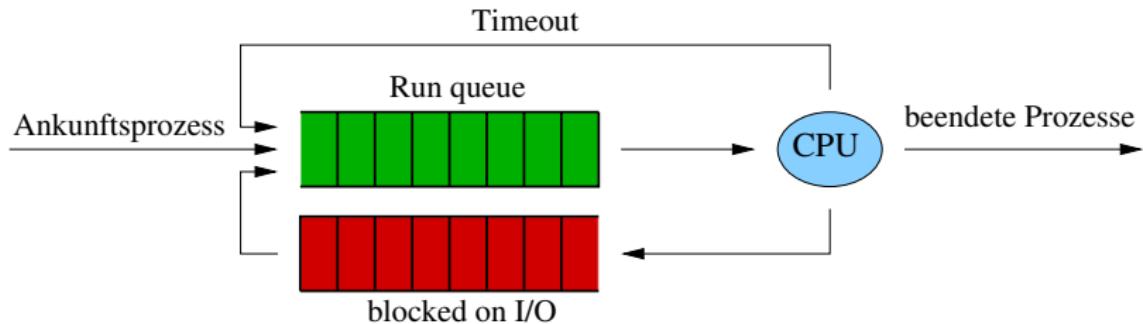
**Definition 17:** Ein Vergabeplan für eine Prozeßmenge  $P = \{P_1, P_2, \dots, P_n\}$  mit gegebenen  $r_i, t_i, d_i$  heißt **gültig**, falls sich keine Ausführungszeiten auf einem Prozessor überlappen und sämtliche Zeitbedingungen eingehalten werden.

## Wie wählt man ein geeignetes Schedulingverfahren aus?

- ① **Beschreibung der Betriebsart**  
(Stapelverarbeitung, Dialogsystem, Echtzeitsystem, ... )
- ② **Beschreibung der Arbeitslast**  
(interaktiv, CPU-intensiv, periodische Datenströme, Verteilung der Ankunftszeiten, Verteilung der Rechenzeiten, ... )  $\Rightarrow$   
Gütekriterium wird festgelegt
- ③ **Bewertung:** analytische Modellierung mittels  
Warteschlangentheorie bzw. ereignis-basierte Simulation  
(siehe Master-VL „Leistungsanalyse“)

## Beispiel: Modellierung mittels Warteschlangentheorie

Die ankommenden (vom BS zu verwaltenden) Prozesse werden als *stochastischer Prozeß* modelliert, d.h. die Ankunftszeiten und Rechenzeiten der (vom BS zu verwaltenden) Prozesse werden durch Verteilungsfunktionen beschrieben (Poissonverteilung u. a.).



**gegeben:** (Annahmen über) Verteilung der Prozeßankünfte und Rechenzeiten

**gesucht:**  $E(V)$

$E(N)$ ,  $N =$  Länge der CPU-Runqueue

## Probleme:

- Rechenzeiten  $t_i$  sind i.a. nicht im voraus bekannt,
- Schedulingverfahren verursachen Overhead!

⇒ Einsatz von einfachen **Heuristiken**, die die optimale Lösung annähern sollen

## Shortest Job First (SJF)

**Gegeben:** Prozesse  $P_1, P_2, \dots, P_n$   
(geschätzte) Rechenzeiten  $t_1, t_2, \dots, t_n$

**Beispiel:**  $n = 5$

$$t_1 = 2, \quad t_2 = 4, \quad t_3 = t_4 = t_5 = 1$$

**Theorem 1:** Bei gegebener Prozeßmenge ist SJF optimal bzgl.  $\bar{W}$ .

Variante: Shortest next CPU burst

manchmal auch SJF genannt!

Prozeßverhalten: Wechsel von CPU- und I/O-intensiven Phasen  
(sogenannte CPU- bzw. I/O-Bursts)

## Approximation der CPU-Burst-Länge

$t_n :=$  Länge des beobachteten n-ten CPU-Bursts

$\tau_{n+1} :=$  geschätzte Länge des  $(n + 1)$ -ten CPU-Bursts

Wähle  $\alpha$  mit  $0 \leq \alpha \leq 1$

$$\tau_0 = \bar{t}$$

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \tau_n$$

$$\alpha = 0 \implies \tau_{n+1} = \tau_n = \bar{t}$$

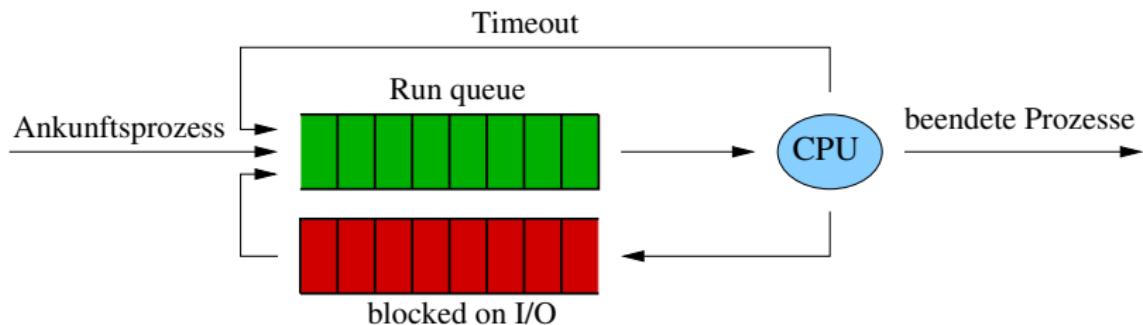
$$\alpha = 1 \implies \tau_{n+1} = t_n$$

üblich:  $0 < \alpha < 1$

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) [\alpha t_{n-1} + (1 - \alpha) \tau_{n-1}] \\ &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^n \alpha t_0 + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

## Approximation für SJF: Zeitscheibenverfahren

Jedem Prozeß wird ein Zeitquantum (sog. **Zeitscheibe**) an Rechenzeit zugeteilt. Die rechenbereiten Prozesse werden in einer FIFO-Warteschlange verwaltet. Nach Ablauf der Zeitscheibe werden nicht-beendete Prozesse wieder hinten in die Warteschlange einsortiert (**Round-Robin**).



Die Wartezeit eines Prozesses hängt ab:

- von der Anzahl Prozesse im System und
- von seiner Rechenzeitanforderung.

**Vorteil:** Zeitscheiben reduzieren die Antwortzeit:

Sind bei Ankunft des Prozesses  $n$  Prozesse im System, bekommt der Prozeß nach maximal  $n$  Zeitscheiben die CPU zugeteilt.

**Nachteil:** Zusätzlicher Aufwand für den Prozeßwechsel.

**Beispiel:** Prozeßwechsel unter SunOS4.03 ca. 1 ms.

Zeitscheibenlänge	# CS pro Sekunde	Overhead pro Sekunde	Overhead in %
1 s	1	1 ms	0,1 %
0.1 s	10	10 ms	1 %
10 ms	100	0.1 s	10 %

### Kompromiß:

Zeitscheibenlänge  $\delta t \rightarrow 0 \Rightarrow$  kurze Antwortzeiten

Zeitscheibenlänge  $\delta t \rightarrow \infty \Rightarrow$  geringerer Overhead

**Beispiel:** Linux 2.4 benutzt eine Zeitscheibenlänge von 10 ms,

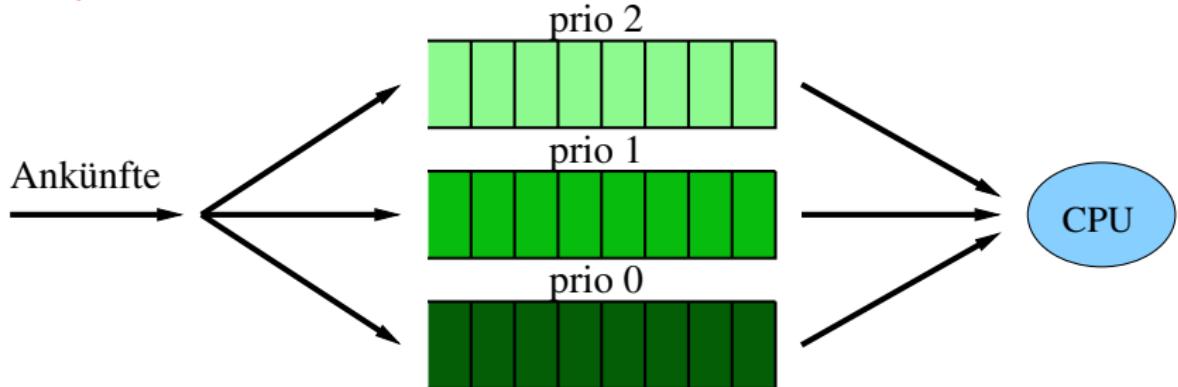
ab Linux 2.6 werden unterschiedlich lange Zeitscheiben vergeben:  
10 ms (I/O-Prozesse) - 5120 ms (rechenintensive Prozesse)

## Prioritäten-basiertes Scheduling

verdrängend / nicht verdrängend

**Verfahren:** Jedem Prozeß wird eine Priorität zugeordnet.  
Prozesse einer Priorität werden in derselben  
Warteschlange verwaltet. Innerhalb der Warteschlange  
wird FIFO angewandt.  
Die CPU wird dem ersten Prozeß in der nichtleeren  
Warteschlange höchster Priorität zugeordnet.

## Beispiel:



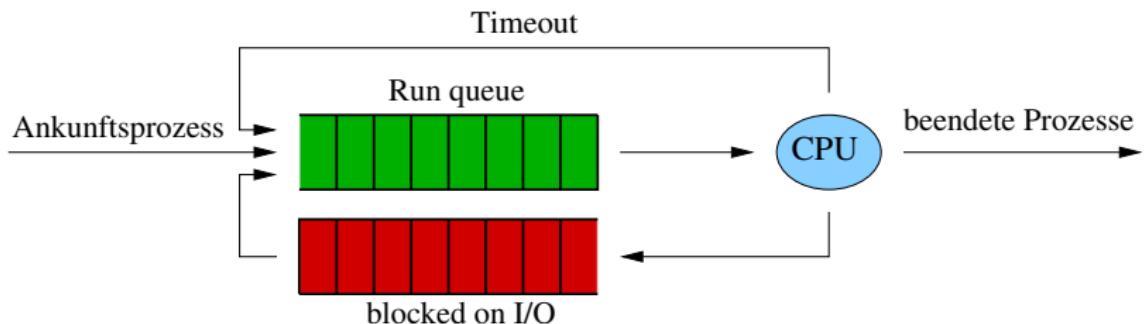
	$t$	prio	Ankunftszeit
$P_1$	2	1	0
$P_2$	1	0	1
$P_3$	3	2	2
$P_4$	1	1	3
$P_5$	1	2	4

## Bemerkung:

- SJF:  $prio = t_i^{-1}$ ,
- **Nachteil:** Prozesse geringer Priorität können verhungern (Starvation)!
- **Vorteil:** Prio-Scheduling unterstützt Ablaufsteuerung!
  - ① Wichtige Prozesse, die Betriebssystemaufgaben ausführen, können höhere Priorität bekommen.
  - ② Prozesse, die längere Zeit nicht gerechnet haben, können kurzfristig eine höhere Priorität erhalten.  
⇒ sogenanntes **Priority Boosting**

**Beispiel für Priority Boosting:** Erhöhen der Priorität, nachdem der Prozeß längere Zeit blockiert war, weil er auf ein Ereignis gewartet hat:

Tritt das Ereignis ein und wird der Prozess wieder rechenbereit gesetzt, wird seine Priorität *kurzfristig* erhöht.



**Definition 18:** Priority Inversion bezeichnet eine Scheduling-Anomalie: Ein Prozeß niedriger Priorität wird gerechnet und ein Prozeß höherer Priorität wartet auf ein Ereignis, daß nur der andere Prozeß auslösen kann.