

# Betriebssysteme 1

SS 2017

**Prof. Dr.-Ing. Hans-Georg Eßer**  
Fachhochschule Südwestfalen

**Foliensatz D:**

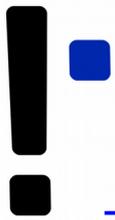
- Scheduler

v1.0, 2016/05/20

# Scheduler: Gliederung

---

- Was ist Scheduling? Motivation
- Kooperatives / präemptives Scheduling
- CPU- und I/O-lastige Prozesse
- Ziele des Scheduling  
(abhängig vom BS-Typ)
- Standard-Scheduling-Verfahren



# Einführung

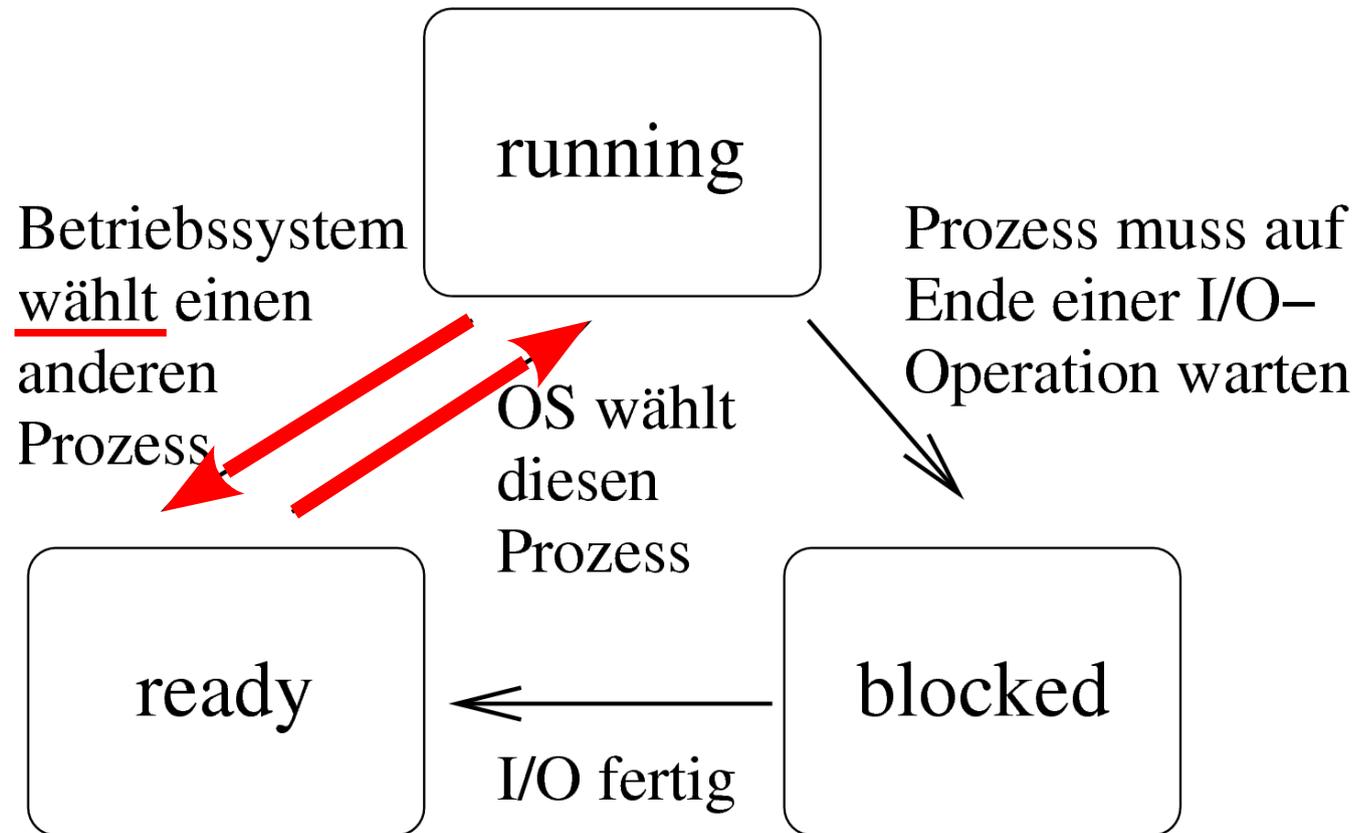
# Scheduling – worum geht es?

---

- Multitasking: Mehrere Prozesse konkurrieren um Betriebsmittel
- Betriebssystem verwaltet die Betriebsmittel
- Rechenzeit auf dem Prozessor
- Scheduler entscheidet:  
Welchen Prozess wann ausführen?
- Ausführreihenfolge entscheidend für  
Gesamt-Performance des Betriebssystems

# Scheduling: Prozess auswählen

## Zustandsübergänge



# Wann wird Scheduler aktiv?

---

- Neuer Prozess entsteht (fork)
- Aktiver Prozess blockiert wegen I/O-Zugriff
- Blockierter Prozess wird bereit
- Aktiver Prozess endet (exit)
- Prozess rechnet schon zu lange
- Interrupt tritt auf

# ■ Unterbrechendes Scheduling

---

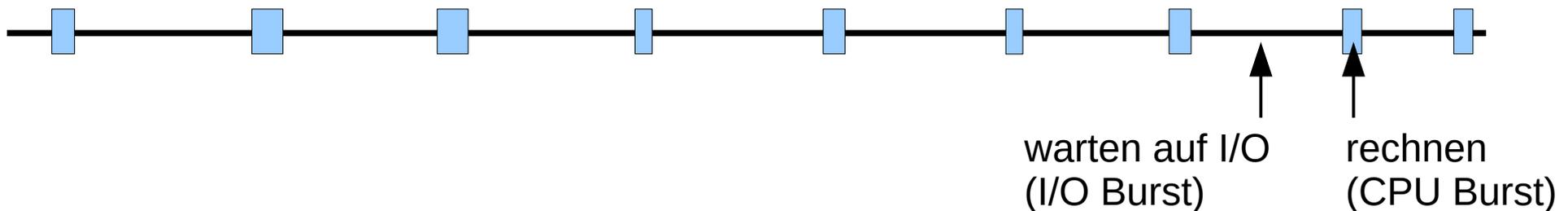
## Prozess-Unterbrechung möglich?

- **Kooperatives Scheduling:**
  - Prozess rechnet so lange, wie er will; bis zum nächsten I/O-Aufruf oder bis `exit()`
  - Scheduler wird nur bei Prozess-Blockieren oder freiwilliger CPU-Aufgabe aktiv
- **Präemptives (unterbrechendes) Scheduling:**
  - Timer aktiviert regelmäßig Scheduler, der neu entscheiden kann, „wo es weiter geht“

# Prozesse: I/O- oder CPU-lastig

## I/O-lastig:

- Prozess hat zwischen I/O-Phasen nur kurze Berechnungsphasen (CPU)



## CPU-lastig:

- Prozess hat zwischen I/O-Phasen lange Berechnungsphasen



# Häufige Prozesswechsel?

---

## Faktoren

- **Zeit für Kontext-Switch:** Scheduler benötigt Zeit, um Prozesszustand zu sichern  
→ verlorene Rechenzeit
- **Wartezeit der Prozesse:** Häufigere Wechsel erzeugen stärkeren Eindruck von Gleichzeitigkeit

# Ziele des Scheduling (1)

---

## Aus Anwendersicht

- **[A1] Ausführdauer:** Wie lange läuft der Prozess insgesamt?
- **[A2] Reaktionszeit:** Wie schnell reagiert der Prozess auf Benutzerinteraktion?
- **[A3] Deadlines** einhalten
- **[A4] Vorhersehbarkeit:** Gleichartige Prozesse sollten sich auch gleichartig verhalten, was obige Punkte angeht
- **[A5] Proportionalität:** „Einfaches“ geht schnell

# Ziele des Scheduling (2)

---

## Aus Systemsicht

- **[S1] Durchsatz:** Anzahl der Prozesse, die pro Zeit fertig werden
- **[S2] Prozessorauslastung:** Zeit (in %), die der Prozessor aktiv war
- **[S3] Fairness:** Prozesse gleich behandeln, keiner darf „verhungern“
- **[S4] Prioritäten** beachten
- **[S5] Ressourcen** gleichmäßig einsetzen

# [A1] Ausführdauer

---

## Wie viel Zeit vergeht vom Programmstart bis zu seinem Ende?

- $n$  Prozesse  $p_1$  bis  $p_n$  starten zum Zeitpunkt  $t_0$  und sind zu den Zeitpunkten  $t_1$  bis  $t_n$  fertig
- Durchschnittliche Ausführdauer:  
$$1/n \cdot \sum_i (t_i - t_0)$$
- Abhängig von konkreten Prozessen;  
Berechnung nur für Vergleich verschiedener Scheduling-Verfahren sinnvoll

### Wie schnell reagiert das System auf Benutzereingaben?

- Benutzer drückt Taste, klickt mit Maus etc. und wartet auf eine Reaktion
- Wie lang ist die Zeit zwischen Auslösen des Interrupts und Aktivierung des Prozesses, der die Eingabe auswertet?
- Toleranz bei langen Wartezeiten gering; schon 2–4 Sekunden kritisch, darüber inakzeptabel

## Hält das System Deadlines ein?

- Realtime-Systeme: besondere Ansprüche
- Prozesse müssen in vorgegebener Zeit ihre Aufgaben erledigen, also ausreichend und rechtzeitig Rechenzeit erhalten
- Wie oft werden Deadlines nicht eingehalten?
- Optimiere (prozentualen) Anteil der eingehaltenen Deadlines

## [A4] Vorhersehbarkeit

---

### Ähnliches Verhalten ähnlicher Prozesse?

- Intuitiv: Gleichartige Prozesse sollten sich auch gleichartig verhalten, d. h.
  - ♦ Ausführdauer und Reaktionszeit immer ähnlich
  - ♦ Unabhängig vom sonstigen Zustand des Systems
- Schwierig, wenn das System beliebig viele Prozesse zulässt → Beschränkungen?

# [A5] Proportionalität

---

## Vorgänge, die „einfach“ sind, werden schnell erledigt

- Es geht um das (evtl. falsche) Bild, das Anwender sich von technischen Abläufen machen
- Benutzer akzeptiert Wartezeit eher, wenn er den zugrunde liegenden Vorgang als komplex einschätzt

## Terminierende Prozesse

- Anzahl der Prozesse, die pro Zeiteinheit (z. B. pro Stunde) fertig werden, sollte hoch sein
- Misst, wie viel Arbeit erledigt wird
- Abhängig von konkreten Prozessen; Berechnung nur für Vergleich verschiedener Scheduling-Verfahren sinnvoll

# [S2] Prozessorauslastung

---

## **CPU immer gut beschäftigt halten**

- Anteil der Taktzyklen, in denen die CPU nicht „idle“ war
- Interessanter Faktor, wenn Rechenzeit sehr wertvoll ist (kommerzielles Rechenzentrum)

## Alle Prozesse haben gleiche Chancen

- Jeder Prozess sollte mal drankommen (kein „Verhungern“, engl. *process starvation*)
- Keine großen Abweichungen bei den Wartezeiten und Ausführdauern
- Falls Prozess-Prioritäten:  
→ „manche sind gleicher“

## Verschieden wichtige Prozesse auch verschieden behandeln

- Prioritätsklassen: Prozesse mit hoher Priorität bevorzugt behandeln
- Dabei verhindern, dass nur noch Prozesse mit hoher Priorität laufen (und alles andere steht)

# [S5] Ressourcen-Balance

---

## „BS verwaltet die Betriebsmittel...“

- Grundidee des BS: alle Ressourcen gleichmäßig verteilen und gut auslasten
- CPU-Scheduler hat auch Einfluss auf (un)gleichmäßige Auslastung der I/O-Geräte
- Prozesse bevorzugen, die wenig ausgelastete Ressourcen nutzen wollen

# Anforderungen an das Betriebssystem (1)

---

## **Drei Kategorien**

- Stapelverarbeitung
- Interaktives System
- Echtzeitsystem

## **Immer wichtig:**

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance

## Stapelverarbeitung

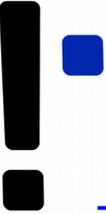
- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- S1 Durchsatz
- A1 Ausführdauer
- S2 Prozessor-Auslastung

## Interaktives System

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- A2 Reaktionszeit
- A5 Proportionalität

## Echtzeitsystem

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- A3 Deadlines
- A4 Vorhersehbarkeit



---

# Scheduler für Stapelverarbeitung (Batch-Systeme)

# Stapelverarbeitung (Batch)

---

## Eigenschaften der Stapelverarbeitung

- Nicht interaktives System  
(keine normalen Benutzerprozesse)
- Jobs werden über Job-Verwaltung abgesetzt;  
System informiert über Fertigstellung
- Typische Aufgaben: Lange Berechnungen,  
Kompilervorgänge

# Stapelverarbeitung (Batch)

## Historisch:

- Batch-Betrieb mit Lochkarten
- Programm-Code und Daten auf Lochkarten
- Keine I/O (keine Geräte außer Kartenleser, Drucker)
- Kartenstapel (engl.: batch) legt Reihenfolge fest
  - Programm-Code von Karten lesen
  - Daten von Karten lesen
  - Berechnung durchführen
  - Ergebnis auf Karten stanzen oder drucken
  - Nächster Job

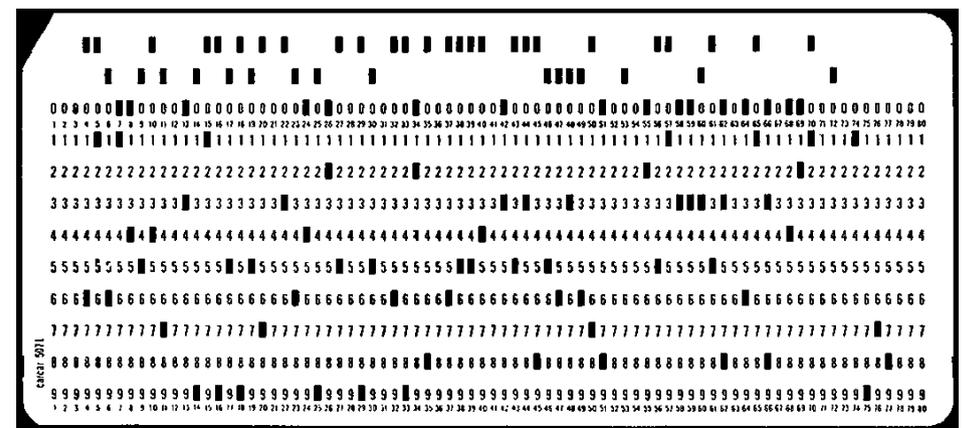


Bild: <http://www.fao.org/docrep/X5738E/x5738e0h.htm>

# Stapelverarbeitung (Batch)

---

## Moderne Batch-Systeme

- Normale Rechner (mit Platten, Netzwerk etc.)
- Kein interaktiver Betrieb (kein Login etc.)
- Job-Management-Tool nimmt Jobs an
- Long term scheduler entscheidet, wann ein Job gestartet wird – evtl. basierend auf Informationen über Ressourcenverbrauch und erwartete Laufzeit des Programms

# Stapelverarbeitung (Batch)

---

## Scheduling-Verfahren für Batch-Betrieb

- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time Next

# First Come, First Served (FCFS)

---

## Einfache Warteschlange

- Neue Prozesse reihen sich in Warteschlange ein
- Scheduler wählt jeweils nächsten Prozess in der Warteschlange
- Prozess arbeitet, bis er fertig ist (kooperatives Scheduling)

# FCFS-Beispiel

Drei Prozesse mit  
Rechendauern

T1: 15 Takte

T2: 4 Takte

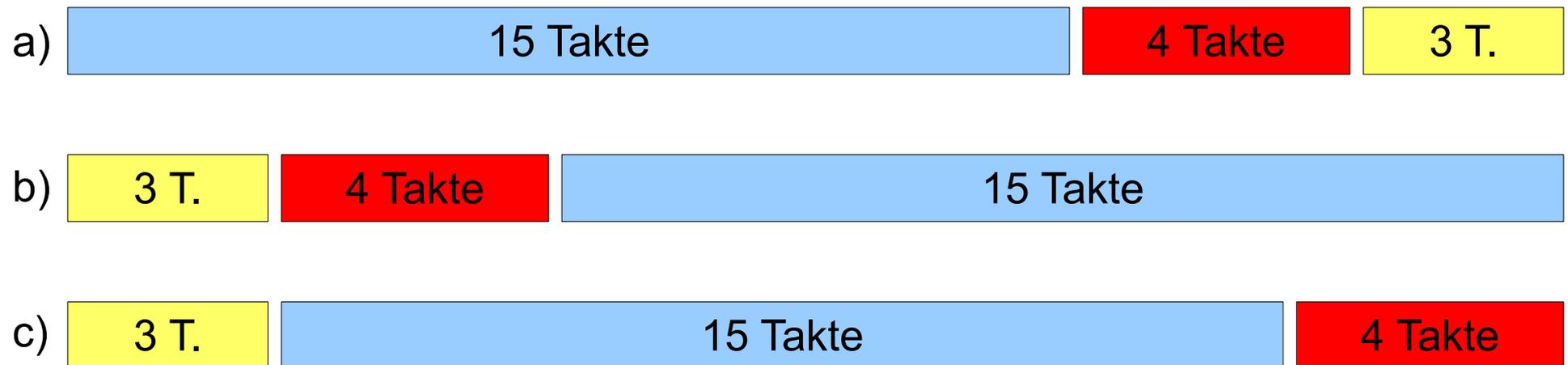
T3: 3 Takte

Durchschnittliche  
Ausföhrdauer:

a)  $(15+19+22) / 3 = 18,67$

b)  $(3+7+22) / 3 = 10,67$

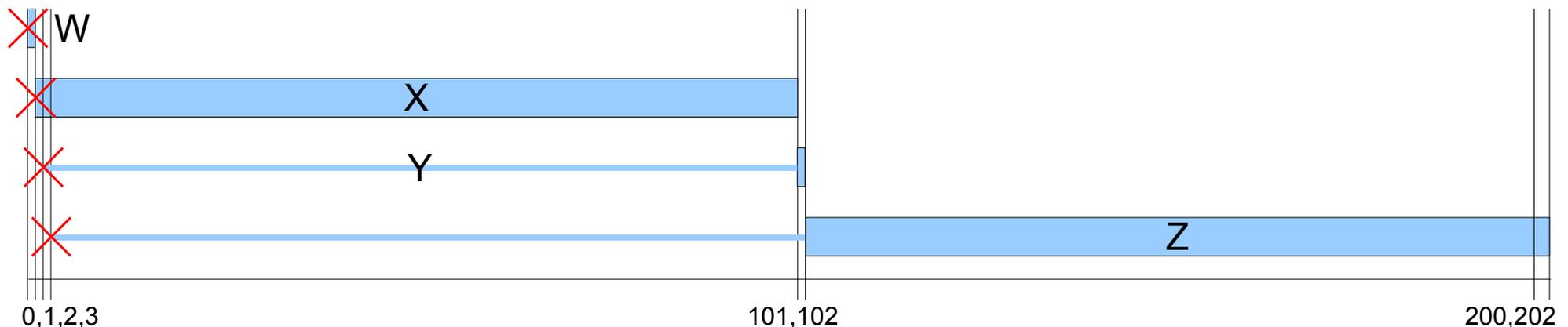
c)  $(3+18+22) / 3 = 14,33$



# FCFS: Gut für lange Prozesse

- FCFS bevorzugt lang laufende Prozesse
- Beispiel: 4 Prozesse W, X, Y, Z

Prozess	Ankunftszeit	Service Time $T_s$ (Rechenzeit)	Startzeit	Endzeit	Turnaround $T_r$ (Endzeit-Ankunftszeit)	$T_r/T_s$
<b>W</b>	0	1	0	1	1	1,00
<b>X</b>	1	100	1	101	100	1,00
<b>Y</b>	2	1	101	102	100	<b>100,00</b>
<b>Z</b>	3	100	102	202	199	1,99



# FCFS: CPU- vs. I/O-lastige Prozesse

---

## FCFS bevorzugt CPU-lastige Prozesse

- Während CPU-lastiger Prozess läuft, müssen alle anderen Prozesse warten
- I/O-lastiger Prozess kommt irgendwann dran, läuft nur sehr kurz und muss sich dann wieder hinten anstellen
- Ineffiziente Nutzung sowohl der CPU als auch der I/O-Geräte

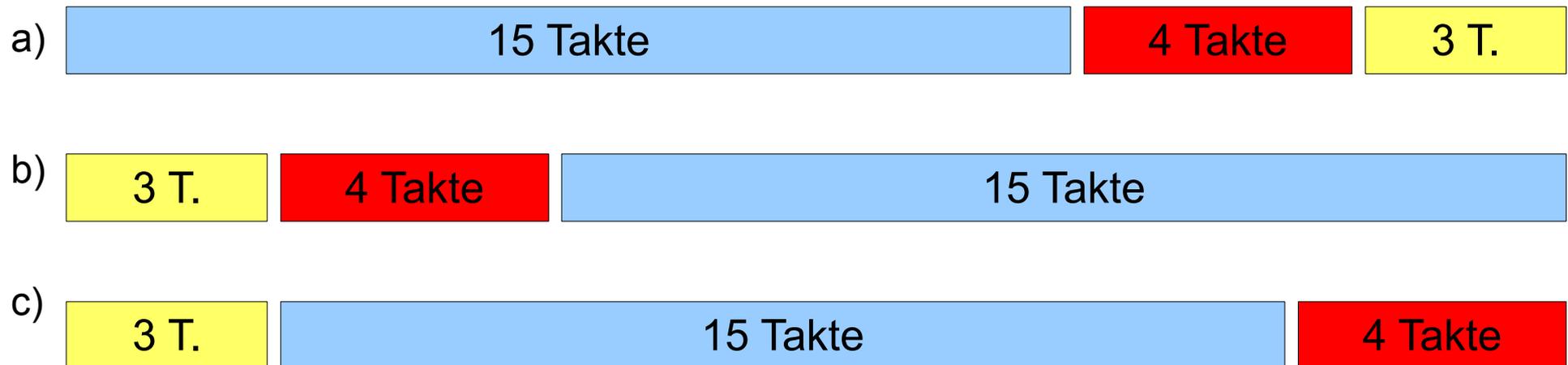
# Shortest Job First (SJF)

---

- Keine Unterbrechungen (wie FCFS)
- Nächste Rechendauer (Burst) aller Prozesse bekannt oder wird geschätzt
- Strategie: Führe zunächst den Prozess aus, der am kürzesten laufen wird
- Minimiert die durchschnittliche Laufzeit aller Prozesse
- Prinzip war schon in FCFS-Beispiel erkennbar

# SJF-Beispiel

Im Beispiel von der FCFS-Folie (Folie 32):  
Ausführreihenfolge b) entspricht SJF



# SJF-Eigenschaften

---

Generelles Problem:

**Woher wissen, wie lange die Prozesse laufen?**

- Batch-System; Programmierer muss Laufzeit schätzen  
→ Bei grober Fehleinschätzung: Job abbrechen
- System, auf dem immer die gleichen / ähnliche Jobs laufen → Statistiken führen
- Interaktive Prozesse: Durchschnitt der bisherigen Burst-Längen berechnen

Ohne diese Information ist dieses Scheduling-Verfahren nur ein theoretisches

# Burst-Dauer-Prognose (1)

---

## Einfachste Variante: Mittelwert

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

mit:

$T_i$ : Dauer des  $i$ -ten CPU-Burst des Prozess

$S_i$ : Vorausgesagte Dauer des  $i$ -ten CPU-Burst

$S_1$ : Vorausgesagte Dauer des 1. CPU-Burst  
(nicht berechnet)

# Burst-Dauer-Prognose (2)

---

## Exponentieller Durchschnitt

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n \quad \alpha: \text{Gewicht zwischen 0 und 1}$$

Beispiel:  $\alpha=0,8$ :

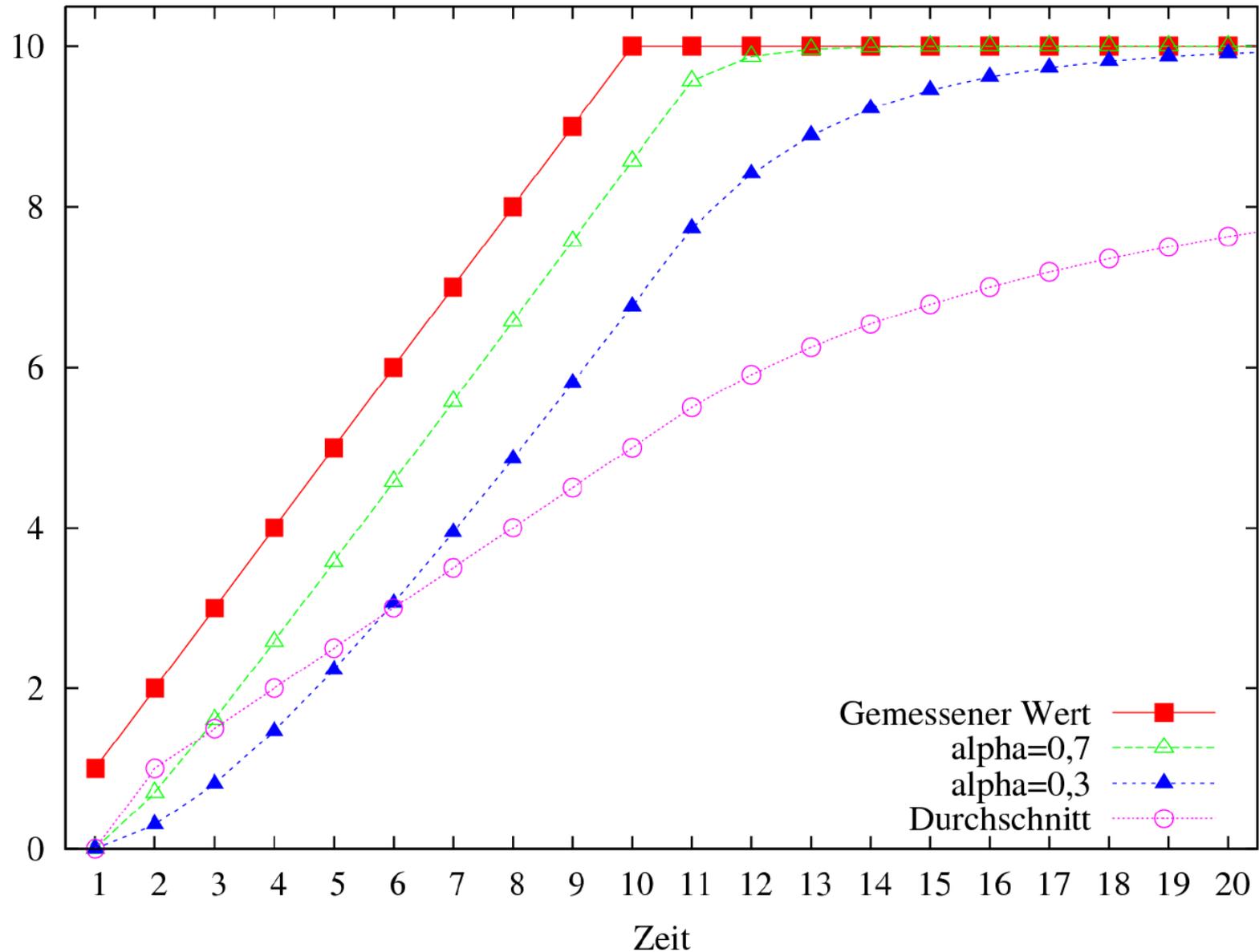
$$S_2 = 0,8 T_1 + 0,2 S_1$$

$$S_3 = 0,8 T_2 + 0,2 S_2 = 0,8 T_2 + 0,2 (0,8 T_1 + 0,2 S_1)$$

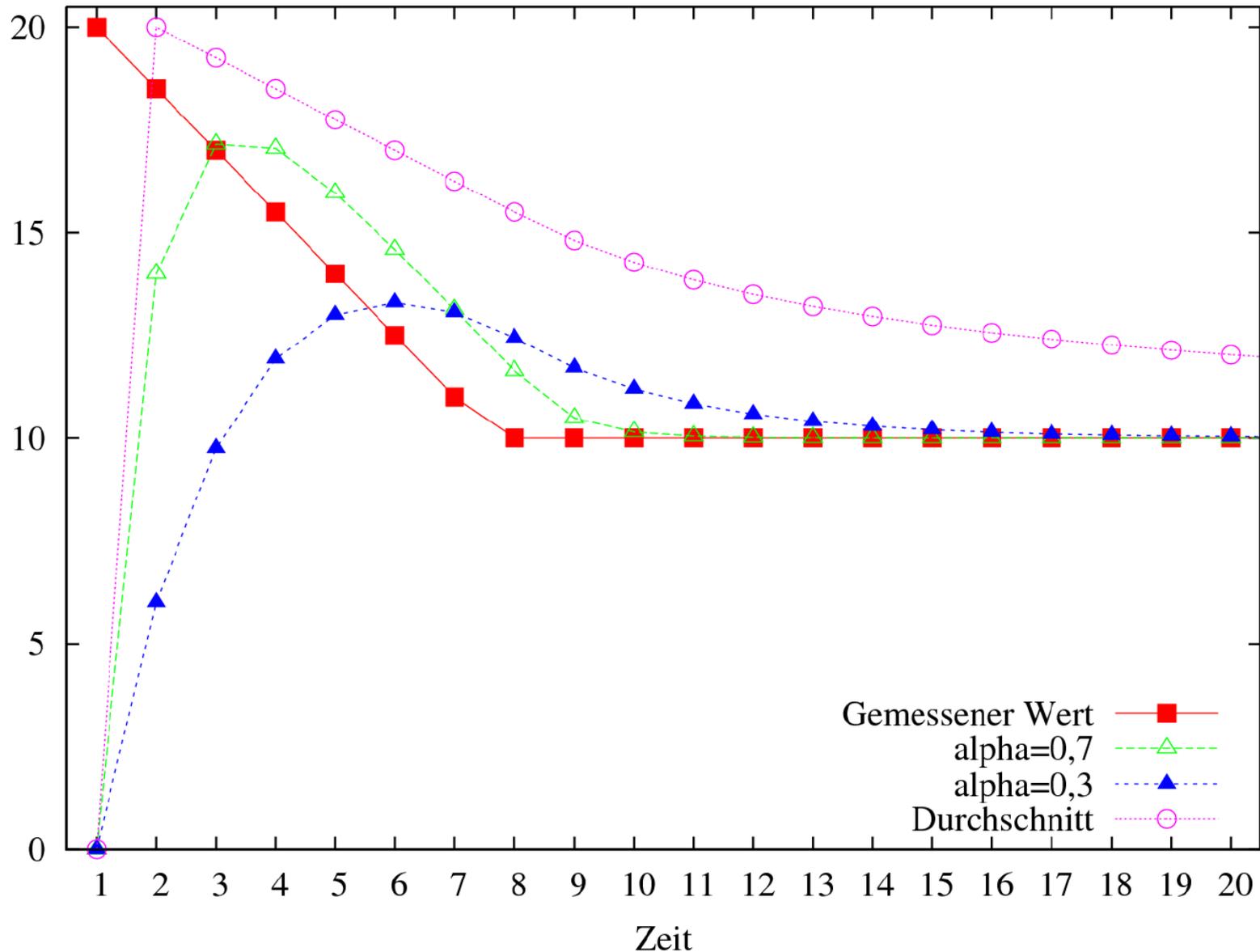
$$\dots = 0,8 T_2 + 0,16 T_1 + 0,04 S_1$$

$$S_{n+1} = \sum_{i=0}^n (1 - \alpha)^{n-i} \alpha T_i \quad \text{mit} \quad T_0 := S_1$$

# Burst-Dauer-Prognose (3)



# Burst-Dauer-Prognose (4)



# Shortest Remaining Time (SRT)

---

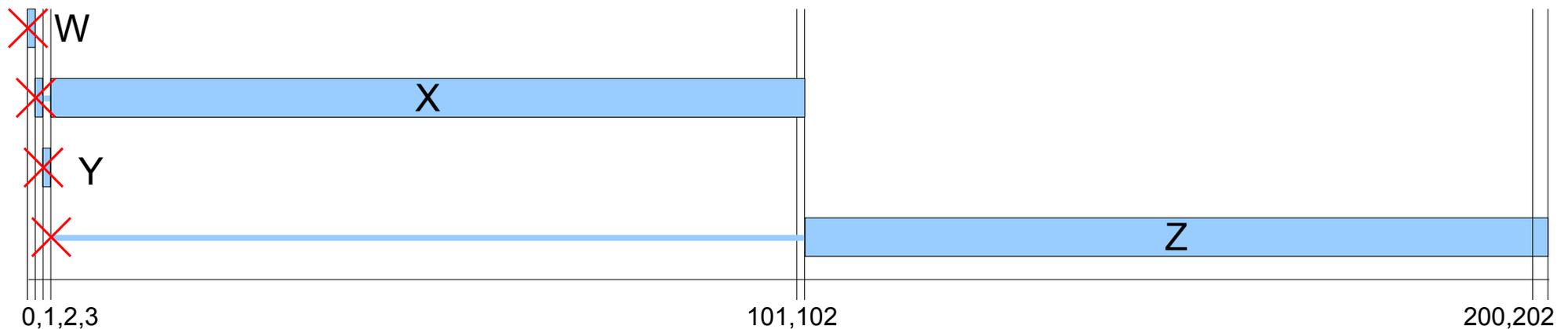
SRT ähnelt SJF, aber:

- präemptiv (mit Unterbrechungen)
- Regelmäßig Neuberechnung, wie viel Restzeit die Prozesse noch benötigen werden
- Scheduler prüft Reihenfolge immer, wenn ein neuer Job erzeugt wird
- Für kürzeren (auch neuen) Job wird der aktive unterbrochen
- Wie bei SJF gute Laufzeitprognose nötig

# SRT-Beispiel

Altes FCFS-Beispiel: SRT unterbricht jetzt X:  
Denn Y kommt zwar später, ist aber kürzer

Prozess	Ankunftszeit	Service Time $T_s$ (Rechenzeit)	Startzeit	Endzeit	Turnaround $T_r$ (Endzeit-Ankunftszeit)	$T_r/T_s$
<b>W</b>	0	1	0	1	1	1,00
<b>X (1)</b>	1	100	1	2 (*)		
<b>Y</b>	2	1	2	3	1	1,00
<b>X (2)</b>			3	102	102-1=101	1,01
<b>Z</b>	3	100	102	202	199	1,99



# SJF vs. SRT

---

- Vorsicht (auch für Prüfung): SJF und SRT werden oft verwechselt.
- „SJF < SRT“
  - (SJF ist i.d.R. schlechter als SRT, weil es keine Unterbrechungen kennt)
  - dieser Vergleich gilt auch lexikalisch (Eselsbrücke)



---

# Scheduler für interaktive Systeme

# Interaktive Systeme

---

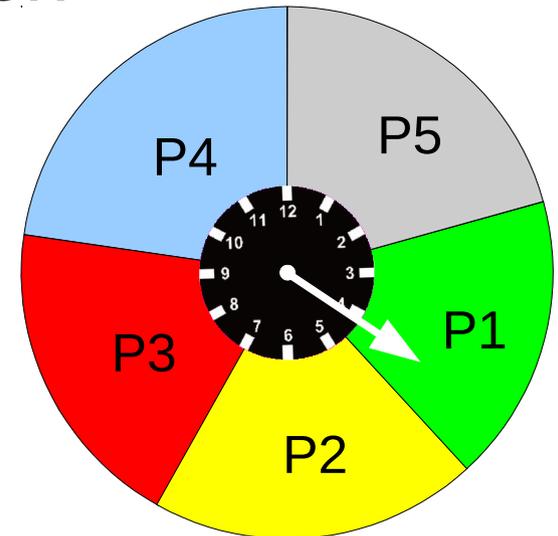
- Typisch: Interaktive und Hintergrund-Prozesse
- Desktop- und Server-PCs
- Eventuell mehrere / zahlreiche Benutzer, die sich die Rechenkapazität teilen
- Scheduler für interaktive Systeme prinzipiell auch für Batch-Systeme brauchbar (aber nicht umgekehrt)

## Scheduling-Verfahren für interaktive Systeme

- Round Robin
- Prioritäten-Scheduler
- Lotterie-Scheduler

# Round Robin / Time Slicing (1)

- Wie FCFS – aber mit Unterbrechungen
- Alle bereiten Prozesse in einer Warteschlange
- Jedem Thread eine Zeitscheibe (quantum, time slice) zuordnen
- Ist Prozess bei Ablauf der Zeitscheibe noch aktiv, dann:
  - Prozess verdrängen (preemption), also in den Zustand „bereit“ versetzen
  - Prozess ans Ende der Warteschlange hängen
  - Nächsten Prozess aus Warteschlange aktivieren



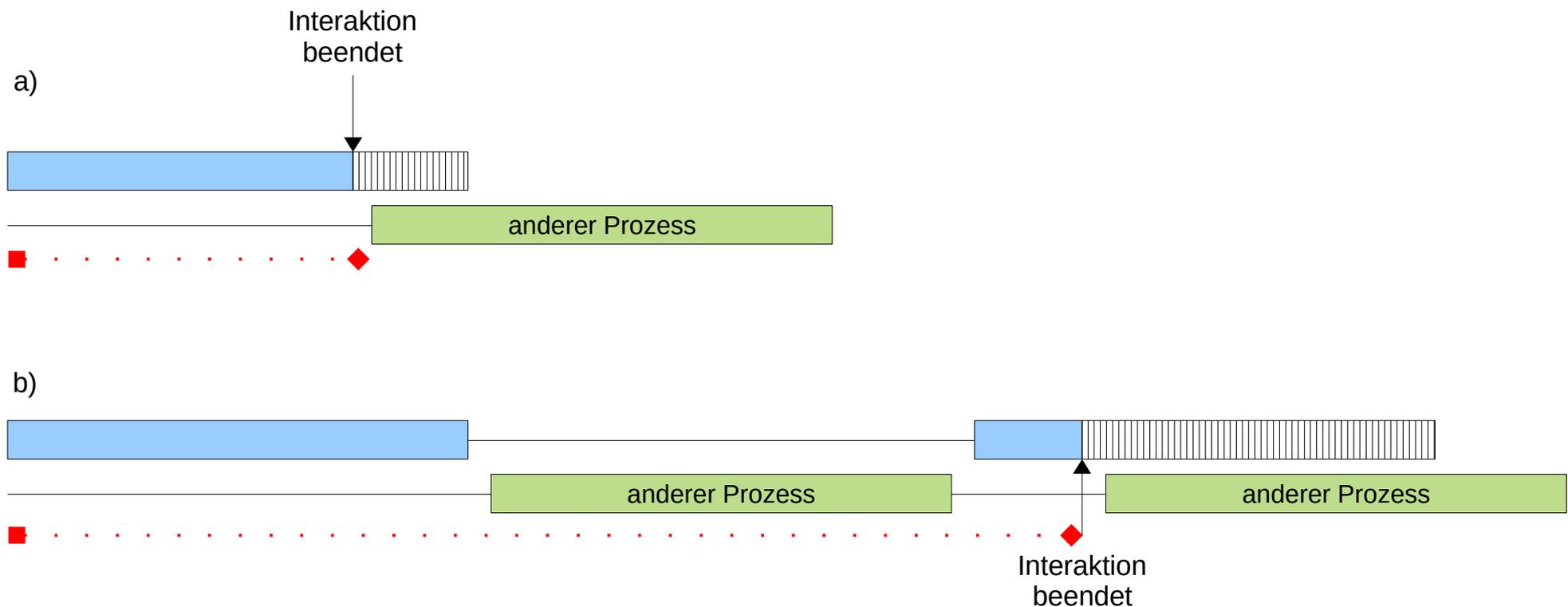
## Round Robin (2)

---

- Blockierten Prozess, der wieder bereit wird, hinten in Warteschlange einreihen
- Kriterien für Wahl des Quantum:
  - Größe muss in Verhältnis zur Dauer eines Context Switch stehen
  - Großes Quantum: evtl. lange Verzögerungen
  - Kleines Quantum: kurze Antwortzeiten, aber Overhead durch häufigen Context Switch

# Round Robin (3)

- Oft: Quantum  $q$  etwas größer als typische Zeit, die das Bearbeiten einer Interaktion benötigt

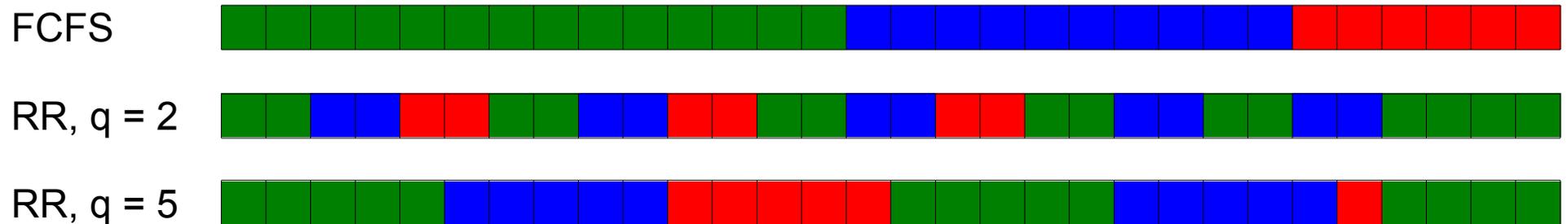


# Round-Robin-Beispiel

---

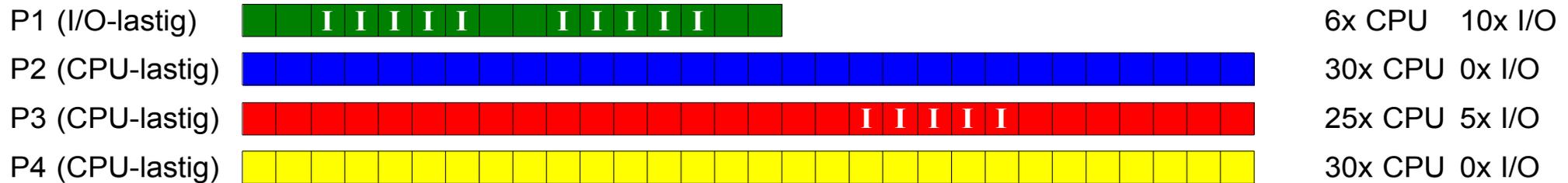
## Szenario: Drei Prozesse

- FCFS (einfache Warteschlange, keine Unterbrechung)
- Round Robin mit Quantum 2
- Round Robin mit Quantum 5

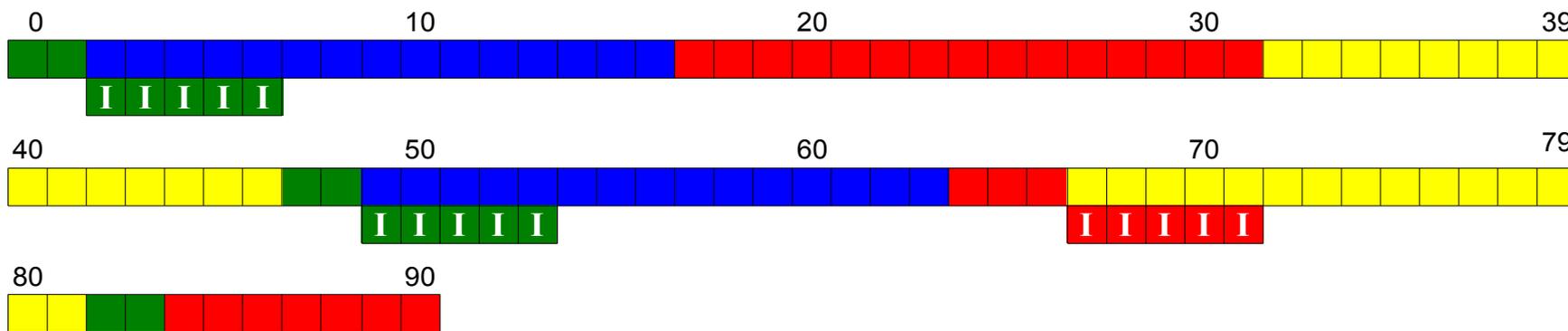


# Round Robin: I/O- vs. CPU-lastig

Idealer Verlauf (wenn jeder Prozess exklusiv läuft)



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

\*) im Zustand  
*bereit*, nicht  
*blockiert!*

# Virtual Round Robin (1)

---

## Beobachtung:

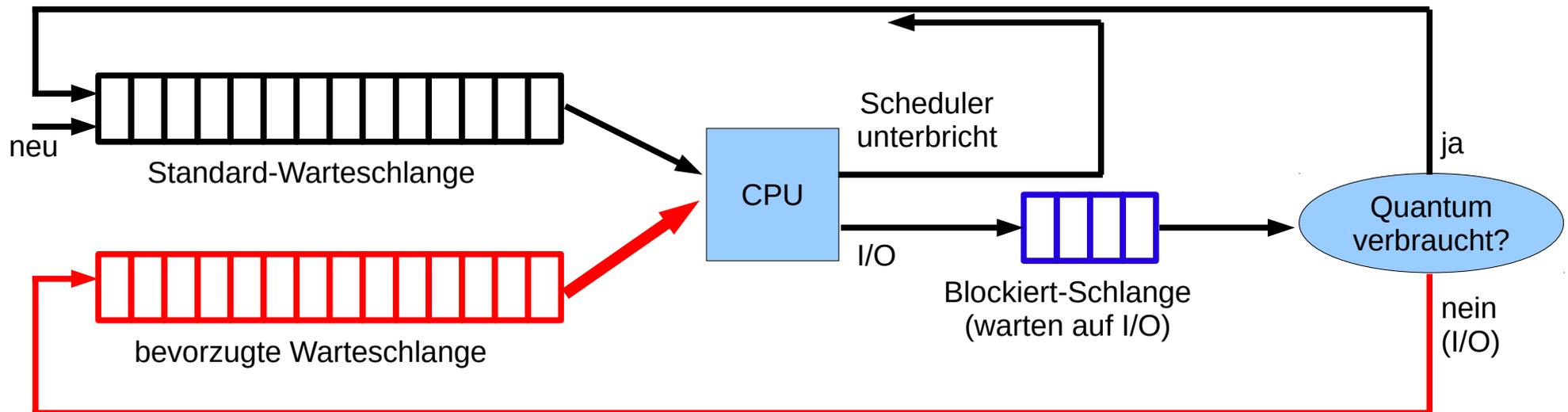
- Round Robin unfair gegenüber I/O-lastigen Prozessen:
  - CPU-lastige nutzen ganzes Quantum,
  - I/O-lastige nur einen Bruchteil

## Lösungsvorschlag:

- Idee: Nicht verbrauchten Quantum-Teil als „Guthaben“ des Prozesses merken
- Sobald blockierter Prozess wieder bereit ist (I/O-Ergebnis da): Restguthaben sofort aufbrauchen

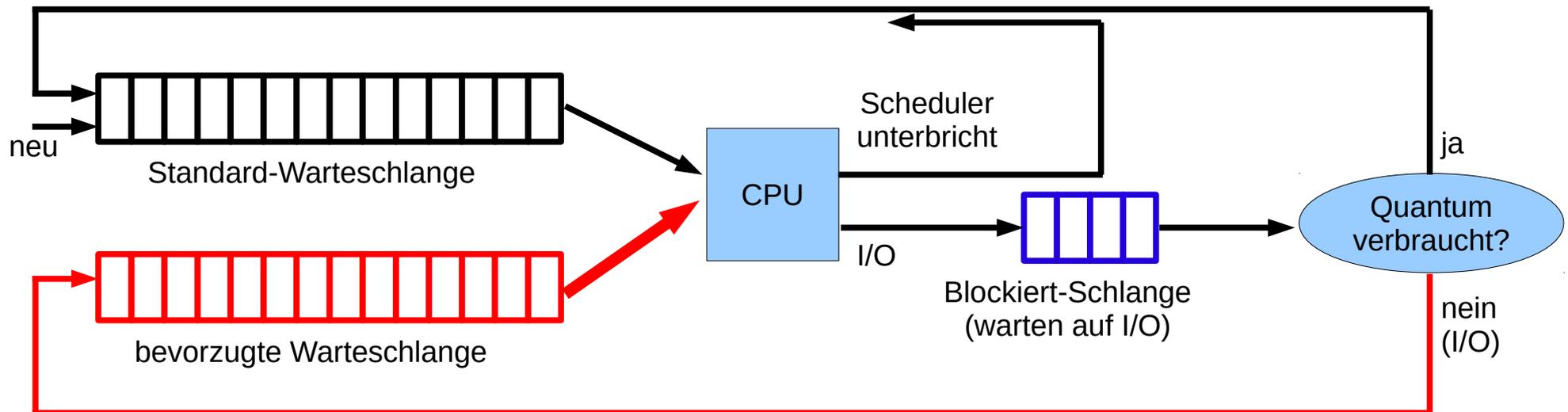
# Virtual Round Robin (2)

- Prozesse, die Zeitquantum verbrauchen, wie bei normalem Round Robin behandeln: zurück in Warteschlange
- Prozesse, die wegen I/O blockieren und nur Zeit  $u < q$  ihres Quantums verbraucht haben, bei Blockieren in Zusatzwarteschlange stecken



# Virtual Round Robin (3)

- Scheduler bevorzugt Prozesse in Zusatzschlange
- Quantum für diesen Prozess:  $q-u$   
(kriegt nur das, was ihm „zusteht“, was er beim letzten Mal nicht verbraucht hat)



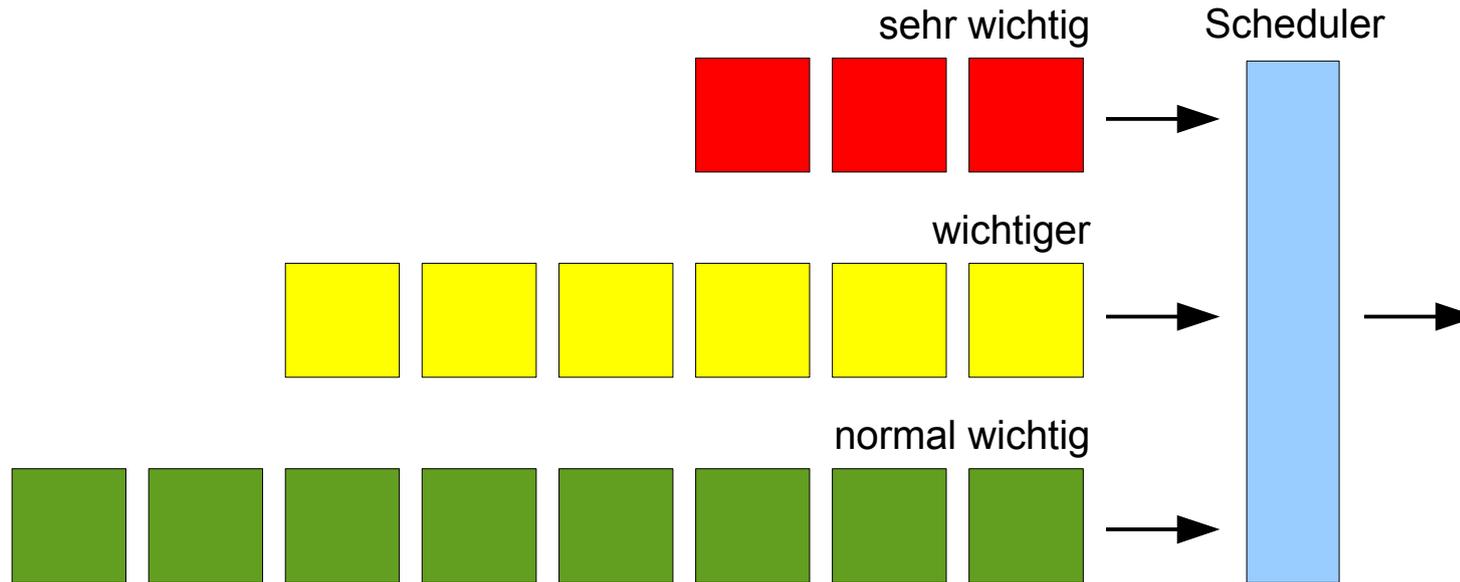
# Prioritäten-Scheduler (1)

---

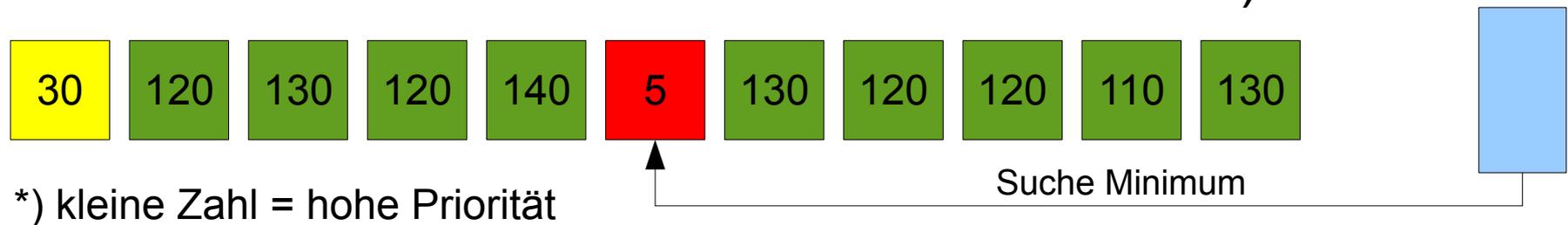
- Idee:
  - a) Prozesse in Prioritätsklassen einteilen oder
  - b) jedem Prozess einen Prioritätswert zuordnen
- Scheduler bevorzugt Prozesse mit hoher Prior.
- Priorität
  - bei Prozesserzeugung fest vergeben
  - oder vom Scheduler regelmäßig neu berechnen lassen
- Scheduling kooperativ oder präemptiv

# Prioritäten-Scheduler (2)

a) Mehrere Warteschlangen für Prioritätsklassen



b) Scheduler sucht Prozess mit höchster Priorität \*)



# Prioritäten-Scheduler (3)

---

## Mehrere Warteschlangen

- Prozesse verschiedenen Prioritätsklassen zuordnen und in jeweilige Warteschlangen einreihen
- Scheduler aktiviert nur Prozesse aus der höchsten nicht-leeren Warteschlange
- Präemptiv: Prozesse nach Zeitquantum unterbrechen
- Innerhalb der Warteschlangen: Round Robin

# Prioritäten-Scheduler (4)

---

## Keine Hierarchien, sondern individuelle Prozess-Prioritäten

- Alle Prozesse stehen in einer Prozessliste
- Scheduler wählt stets Prozess mit der höchsten Priorität
- Falls mehrere Prozesse gleiche (höchste) Priorität haben, diese nach Round Robin verarbeiten

# Prioritäten-Scheduler (5)

---

**Prozesse können „verhungern“ → Aging**

## **Prioritätsinversion:**

- Prozess hoher Priorität ist blockiert (benötigt ein Betriebsmittel)
- Prozess niedriger Priorität besitzt dieses Betriebsmittel, wird aber vom Scheduler nicht aufgerufen (weil es höher-prioritäre Prozesse gibt)
- Beide Prozesse kommen nie dran, weil immer Prozesse mittlerer Priorität laufen
- Ausweg: Aging

# Prioritäten-Scheduler (6)

---

## Aging:

- Priorität eines Prozesses, der bereit ist und auf die CPU wartet, wird regelmäßig erhöht
- Priorität des aktiven Prozesses und aller nicht-bereiten (blockierten) Prozesse bleibt gleich
- Ergebnis: Lange wartender Prozess erreicht irgendwann ausreichend hohe Priorität, um aktiv zu werden

# Prioritäten-Scheduler (7)

---

## Verschiedene Quantenlängen

- Mehrere Prioritätsklassen:
  1. Priorität = 1 Quantum, 2. Priorität = 2 Quanten,
  3. Priorität = 4 Quanten, 4. Priorität = 8 Quanten
- Prozesse mit hoher Priorität erhalten kleines Quantum.
- Geben sie die CPU vor Ablauf des Quantums zurück, behalten sie hohe Priorität
- Verbrauchen sie Quantum, verdoppelt Scheduler die Quantenlänge und stuft die Priorität runter – solange, bis Prozess sein Quantum nicht mehr aufbraucht

# Prioritäten-Scheduler (8)

---

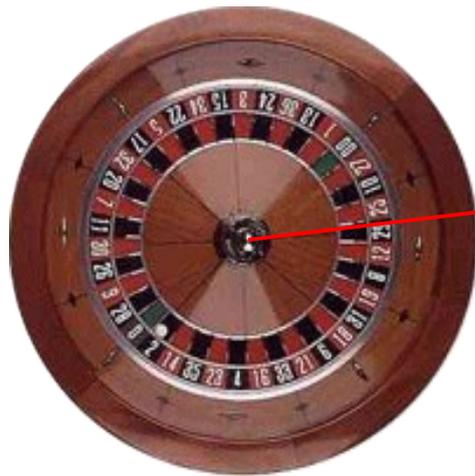
- Linux-Praxis: `nice`, `renice`  
siehe Folien B-58 bis B-62  
(aus Foliensatz zu Prozessen)

# Lotterie-Scheduler (1)

---

- Idee: Prozesse erhalten „Lotterie-Lose“ für die Verlosung von Ressourcen
- Scheduler zieht ein Los und lässt den Prozess rechnen, der das Los besitzt
- Priorisierung: Einige Prozesse erhalten mehr Lose als andere

# Lotterie-Scheduler (2)



Scheduler zieht Los **Nr. 5**

Prozess 1  
Lose 1,2,3,4

Prozess 2  
Lose **5,6**

Prozess 3  
Lose 7,8,9

Prozess 4  
Los 10

# Lotterie-Scheduler (3)

---

- Gruppenbildung und Los-Austausch:
  - Zusammenarbeit Client / Server
  - Client stellt Anfrage an Server, gibt ihm seine Lose und blockiert
  - Nach Bearbeitung gibt Server die Lose an den Client zurück und weckt ihn auf
  - Keine Clients vorhanden?
    - Server erhält keine Lose, rechnet nie

# Lotterie-Scheduler (4)

---

- Aufteilung der Rechenzeit nur statistisch korrekt
- In konkreten Situationen verschieden lange Wartezeiten möglich
- Je länger mehrere Prozesse laufen, desto besser ist erwartete CPU-Aufteilung

# Scheduling auf Multi-CPU-Systemen

---

- Multitasking auf einzelnen CPUs (oder nicht?)
- CPUs gleich-behandeln oder Master/Slaves?
- Zuordnung Prozess  $\leftrightarrow$  CPU: fest/variabel?
- BS-Instanz auf jeder CPU (was passiert, wenn zwei Scheduler denselben Prozess auswählen?)
- Gang Scheduling

Literatur: William Stallings, „Operating Systems – Internals and Design Principles“, Kapitel 10