Prozesse II - Scheduling

Reihenfolgeplanung (Scheduling)

- Planung der Ausführungsreihenfolge von Prozessen ist Aufgabe des **Schedulers**
- In modernen Systemen Unterscheidung zwischen **Job-Scheduling** (Langzeit) und **CPU-Scheduling** (Kurzzeit)
  - **Job-Scheduling:** Langfristige Planung, welche Prozesse aus- und eingelagert werden sollen
  - **CPU-Scheduling:** Planung der Ausführungsreihenfolge der eingelagerten Prozesse
- Unterscheidung **Scheduler/Dispatcher:**
  - Scheduler plant Ausführungsreihenfolge,
  - Dispatcher führt Kontextwechsel aus

Scheduling – Ziele

Scheduler können unterschiedliche, ggf. konkurrierende Ziele verfolgen:

- **Auslastung** der CPU maximieren
- **Durchsatz** (Jobs/Zeiteinheit) maximieren
- mittlere **Ausführungszeit** minimieren
- **Wartezeit** bereiter Prozesse minimieren
- **Antwortzeit** minimieren
- mittlere **CPU-Zeit** je Prozess ausbalancieren
- …

Hierzu wird ggf. eine Abschätzung der zu erwartenden Rechenzeit für jeden Prozess benötigt (nicht immer möglich)

Scheduling – Zielkonflikte

Problem: Ziele weder vollständig noch konsistent!

Beispiele:

1. Optimierung auf Durchsatz bevorzugt kurze Prozesse,
   - dadurch allerdings häufige Prozesswechsel
   - effektive CPU-Auslastung geringer
   - lange Jobs benachteiligt
2. Optimierung auf hohe Auslastung durch möglichst wenige Unterbrechungen,
   - Antwortzeit erhöht sich
   - kurze Jobs benachteiligt

→ **Es gibt keinen idealen Schedulingalgorithmus !**

Scheduling – Zielkonflikte

Fazit: Schedulingalgorithmen je nach Einsatzgebiet auswählen:

- in Batchsystemen liegt Fokus auf Auslastung
- in interaktiven Systemen (Bsp.: PC) sind kurze Antwortzeiten wichtig
- in Echtzeitsystemen*) zählt Einhaltung der Zeitschranken

*) Echtzeitsysteme sind in der Lage, Jobs garantiert innerhalb bestimmter Zeitschranken zu bearbeiten. Sie werden u.A. in Steuerungssystemen verwendet.

Kooperatives und präemptives Scheduling

Zwei grundsätzlich unterschiedliche Schedulingansätze:

- **Kooperativ (nicht-präemptiv):** Job erhält CPU, bis er sie wieder freigibt (Bsp.: Windows 3.11, aber auch auf Mainframes nicht unüblich), Einsatz v.a. in Batchsystemen
- **Präemptiv:** Prozessunterbrechung ist von vornherein vorgesehen (Bsp.: Windows 7, Linux), Einsatz in interaktiven und Echtzeitsystemen

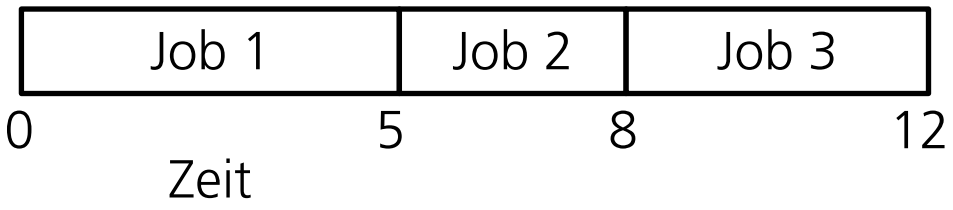
Kooperative Schedulingstrategien

First-Come-First-Serve (FCFS):

- Jobs werden in der Reihenfolge ihres Eintreffens bearbeitet.
- **Beispiel:**

Job	Dauer
1	5 ZE
2	3 ZE
3	4 ZE

Ausführungsreihenfolge:



Job	Ausführungs-zeit
1	5 ZE
2	8 ZE
3	12 ZE

$$\emptyset = 8\frac{1}{3}\ \text{ZE}$$

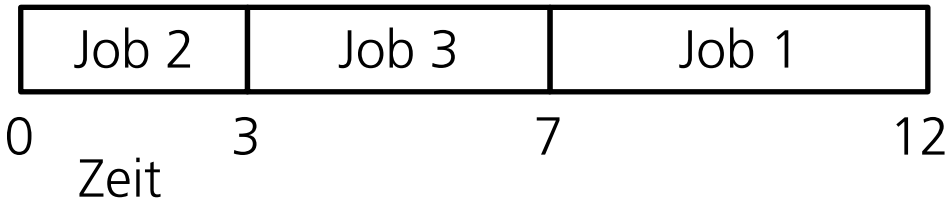
Kooperative Schedulingstrategien

Shortest-Job-First (SJF):

- Kürzester Job wird zuerst ausgeführt
- **Beispiel:**

Job	Dauer
1	5 ZE
2	3 ZE
3	4 ZE

Ausführungsreihenfolge:



Job	Ausführungs-zeit
1	12 ZE
2	3 ZE
3	7 ZE

$$\emptyset = 7\frac{1}{3}\ \text{ZE}$$

Kooperative Schedulingstrategien

Prioritätenscheduling:

- Jobs erhalten Prioritäten
- Jobs mit höchster Priorität*) werden zuerst ausgeführt

Problem: Sowohl SJF als auch Prioritätenscheduling können zum ständigen Verdrängen von langen/niedrig priorisierten Jobs führen, wenn stetig neue, kürzere/höher priorisierte Jobs eintreffen!

Die Jobs „verhungern“, der Effekt wird **Starvation** genannt.

*) In der Praxis kommt es vor, dass die höchste Priorität dem niedrigsten Zahlenwert zugeordnet ist, sodass Prio(0) > Prio (1)

Starvation vermeiden

Vermeidung von Starvation durch **Aging**:

- für jeden Job wird protokolliert, **wie lange** er sich bereits in der Warteschlange befindet (in Zeiteinheiten oder in der Anzahl der Jobs vor ihm),
- mit steigendendem Alter wird die **Priorität** des Jobs erhöht, bis seine Priorität höher ist als die aller anderen Jobs,
- Ansatz ist sowohl bei **SJF** als auch bei **Prioritätenscheduling** anwendbar

Präemptives Scheduling

Jobs nacheinander abzuarbeiten, ist oft nicht praktikabel

- **Antwortzeit** bei interaktiven Systemen leidet,
- **Multitasking** nur mit Multiprozessorsystemen möglich,
- **„gierige" Jobs** binden CPU.

Lösung: Zeitscheibenverfahren

- Jeder Job erhält für eine definierte Zeit die CPU (Zeitscheibe).
- Nach Ablauf der Zeitscheibe erfolgt Unterbrechung und erneute Einreihung des Jobs **am Ende** der Warteschlange.

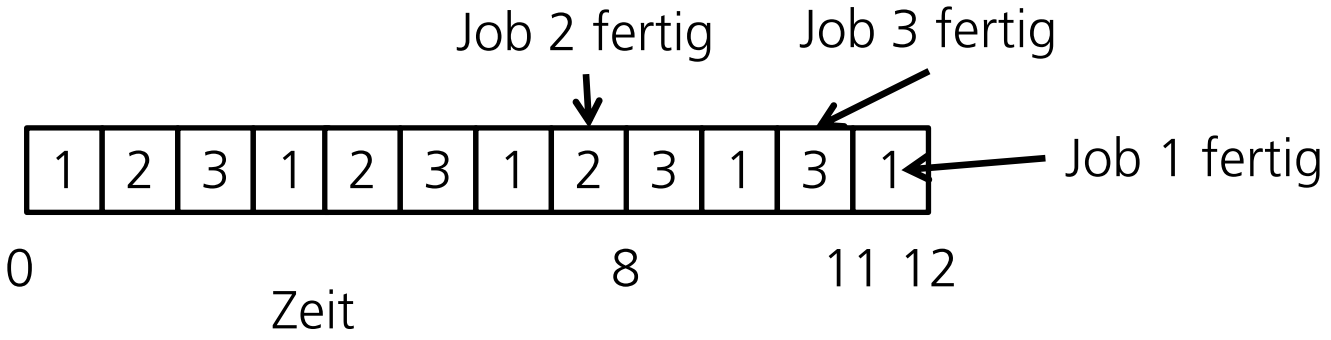
→ Jobs werden **reihum** bedient (engl.: **Round Robin**)

Zeitscheibenverfahren (Round Robin)

Beispiel:

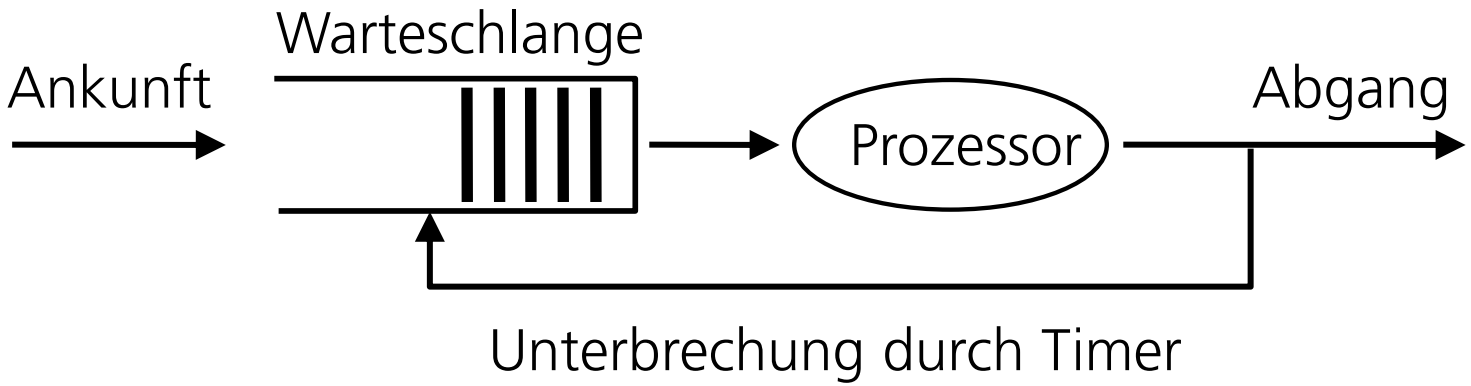
Job	Dauer
1	5 ZE
2	3 ZE
3	4 ZE

Reihenfolge:



Job	**1**	**2**	**3**	∅
Ausführungs-zeit	12	8	11	$\mathbf{10\frac{1}{3}\ ZE}$

Präemptives Scheduling – Zeitscheibenverfahren



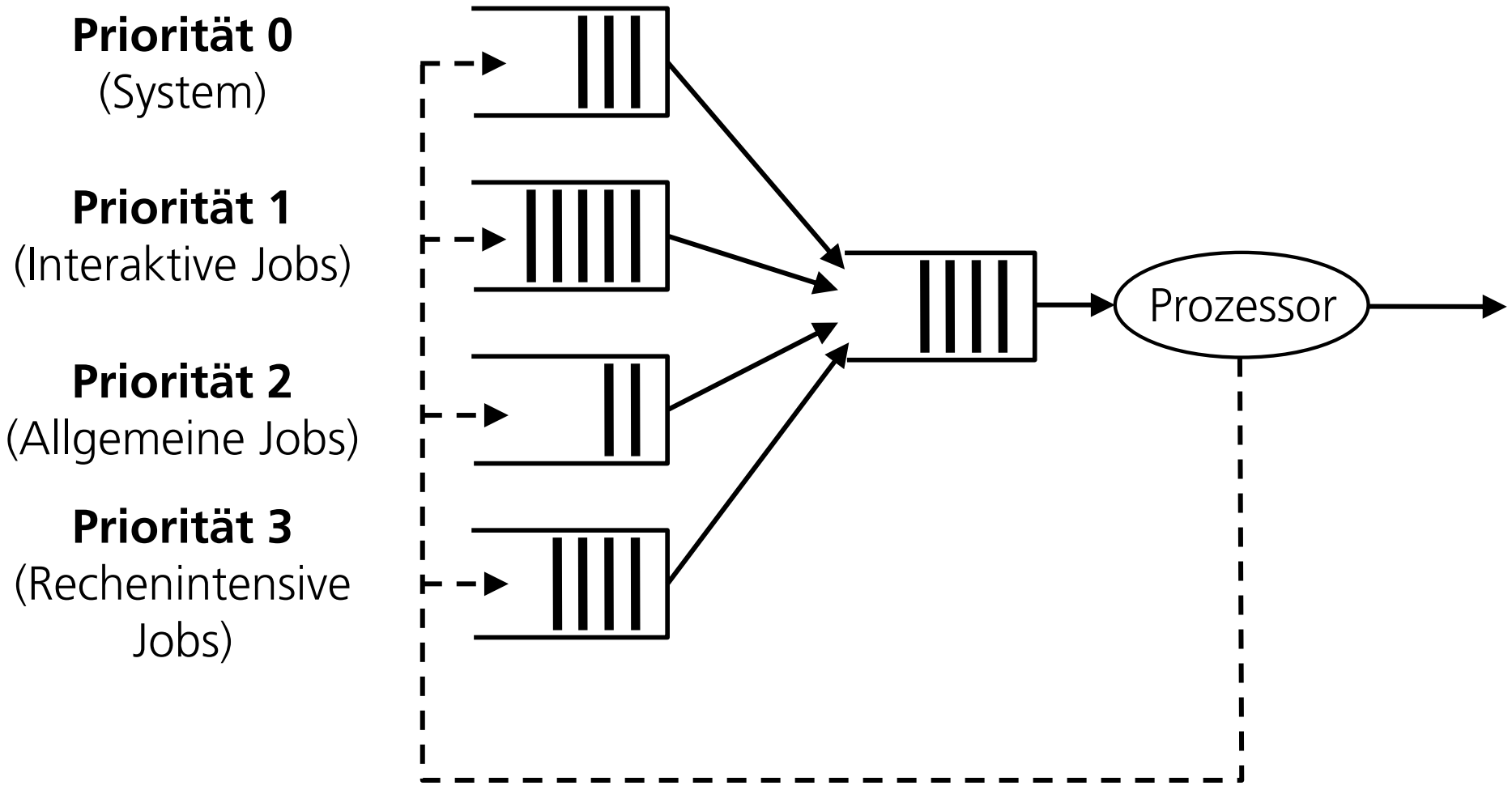
- **Zeitscheibengröße** *T* im Verhältnis zur Umschaltzeit beeinflusst stark die Performance des Systems
- **Richtwert:** *T* > mittlerer CPU-Bedarf zwischen zwei I/O-Vorgängen (CPU burst) von 80% der Jobs
- **Linux (Kernel 2.6):** 4 ms Standard, bis 1 ms möglich

Modifizierte Zeitscheibenverfahren

Dynamic Priority Round Robin (DPRR)

- Round-Robin mit Vorstufe
- Vurstufe enthält priorisierte Warteschlangen
- Priorität der Jobs in den Warteschlangen wächst mit Verweildauer
- Berücksichtigung bei CPU-Zuteilung erst bei Überschreiten eines Prioritätenschwellwertes
- Einsatz in Windows (31 Prioritätsstufen mit eigenen Warteschlangen) und UNIX (bis zu 255 Stufen)

Dynamic Priority Round Robin (DPRR) – Konzept



Hinweis: Der Übersicht halber wurden nur 4 Prioritätsstufen dargestellt. In der Praxis werden 31 (Windows) bis 255 (UNIX) Stufen verwendet.

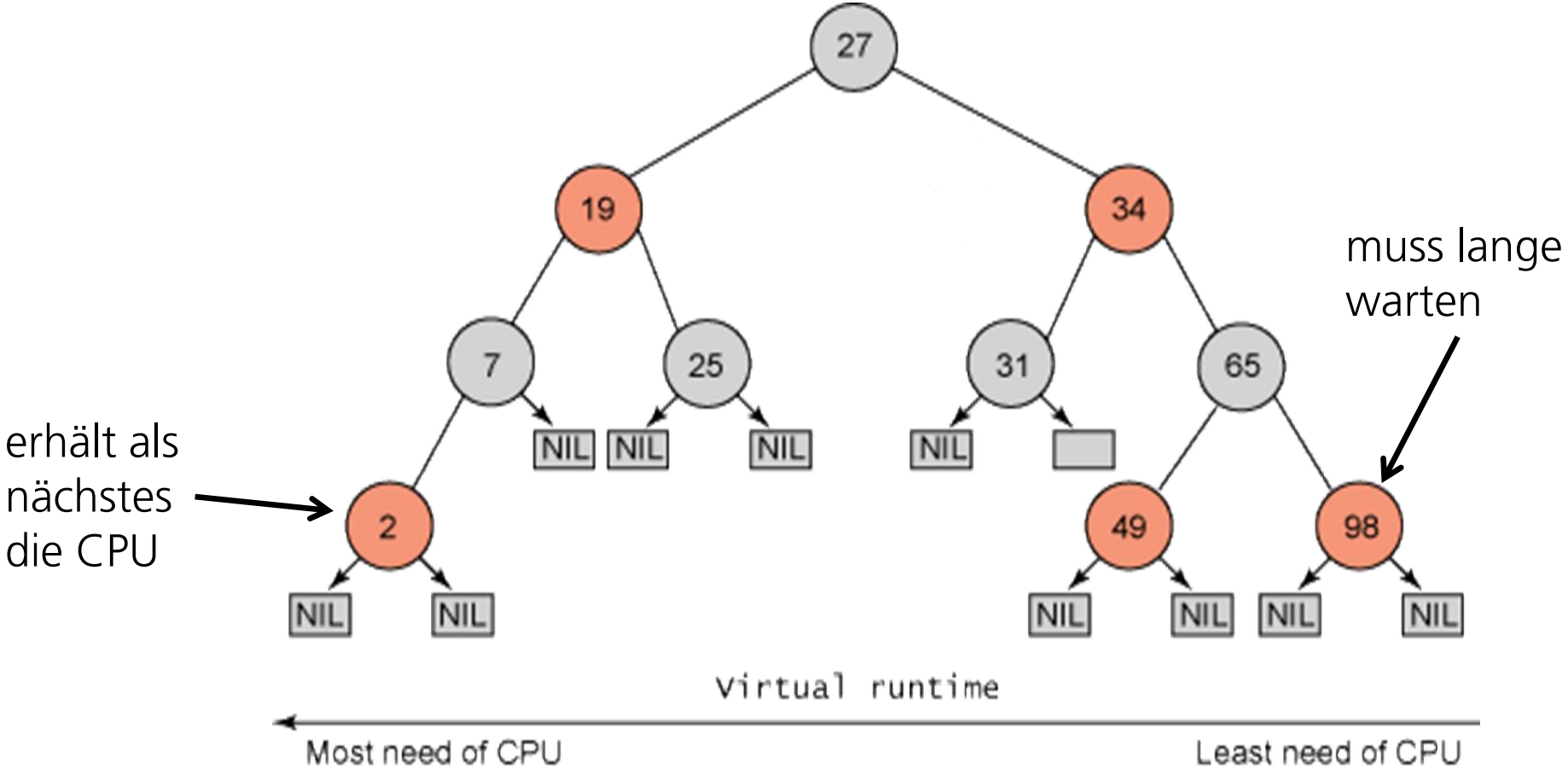
Praxisbeispiel: Completely-Fair-Scheduler (CFS)

Completely-Fair-Scheduler (CFS):

- Standardscheduler in Linux seit Kernel 2.6.23,
- sortiert anstehende Prozesse nach dem Anteil ihrer Laufzeit, in der sie tatsächlich die CPU erhalten haben (*virtual_runtime*),
- Prozess mit geringster *virtual_runtime* erhält CPU für eine Zeitscheibe,
- nach Ablauf der Zeitscheibe wird *virtual_runtime* des Prozesses aktualisiert und der Prozess neu einsortiert,
- nächster Prozess wird ausgewählt

Praxisbeispiel: Completely-Fair-Scheduler (CFS)

Prozesse im CFS nach *virtual_runtime* sortiert in einem Rot/Schwarz-Baum:



Quelle: IBM

Praxisbeispiel: Completely-Fair-Scheduler (CFS)

Completely-Fair-Scheduler (CFS) - Diskussion:

- tatsächlich gleichmäßige Verteilung der CPU-Zeit auf Prozesse,
- sichert, dass lange blockierte Prozesse (wartend auf I/O) die CPU erhalten, sobald sie sie brauchen,
- im Multiuserbetrieb jedoch ggf. unfair gegenüber Usern mit geringer Anzahl an Prozessen,
- nicht vollständig fair, sondern garantiert nur, dass Unfairniss in $O(N)$ bei N Prozessen ist

Scheduling in Echtzeitbetriebssystemen

Definition: Ein Echtzeitbetriebssystem ist ein Betriebssystem mit Echtzeitfunktionen, die die Einhaltung von Zeitbedingungen und die Vorhersagbarkeit des Systemverhaltens gewährleisten.

- Echtzeitverhalten über die normalen Aufgaben eines Betriebssystems hinaus,
- in der Regel kompakter als andere Betriebssysteme, weil hauptsächlich in eingebetteten Systemen verwendet,
- typisch sind wiederkehrende Jobs, oft auch periodisch (bspw. in Regelsystemen)

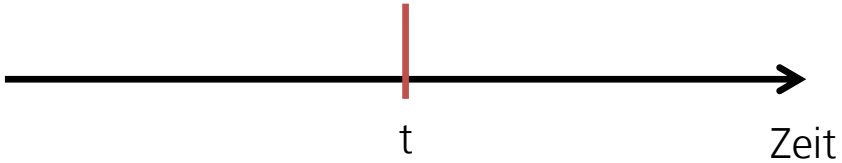
Echtzeitanforderungen

- **Weiche Echtzeitanforderungen**
  - Verzögerung nicht optimal, aber tolerierbar
  - Beispiel: Bankautomat
- **Harte Echtzeitanforderungen**
  - Verzögerung nicht tolerierbar
  - Beispiel: Steuerungssystem bei Flugzeugen

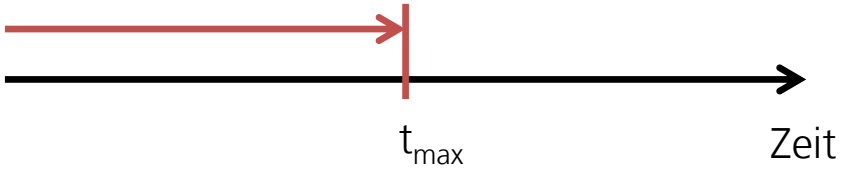
Die zentrale Komponente von Echtzeitbetriebssystemen ist der **Scheduler**!

Rechtzeitigkeit – Varianten

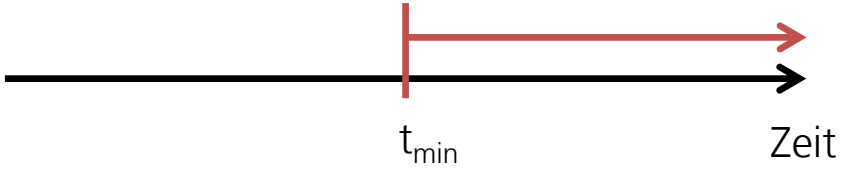
exakter Zeitpunkt
(sehr aufwändig)



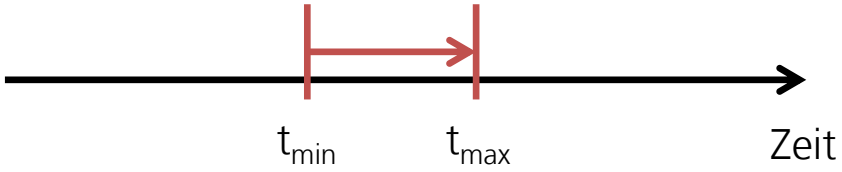
obere Schranke
(Regelfall)



untere Schranke
(leicht einzuhalten)



Bereich



Scheduling in Echtzeitbetriebssystemen

- **Earliest Deadline First (EDF):** Derjenige Job, dessen obere Schranke als nächstes erreicht wird, erhält die CPU
- **Least Laxity First (LLF):** Der Job mit dem geringsten zeitlichen *Spielraum* erhält die CPU

Spielraum: $l_i = tD - (t + t_C)$

t: aktueller Zeitpunkt

t_C: verbleibende benötigte Rechenzeit

t_D: Deadline des Jobs

Der Spielraum des aktuell bearbeiteten Jobs bleibt konstant, der der anderen Jobs sinkt

Zusammenfassung

- Scheduler = Planung der Ausführungsreihenfolge
  - kooperativ oder präemptiv
  - optimiert auf Einsatzzweck
- besondere Anforderungen bei Echtzeitsbetriebssystemen durch Zeitgarantien