




Tobias Lauer

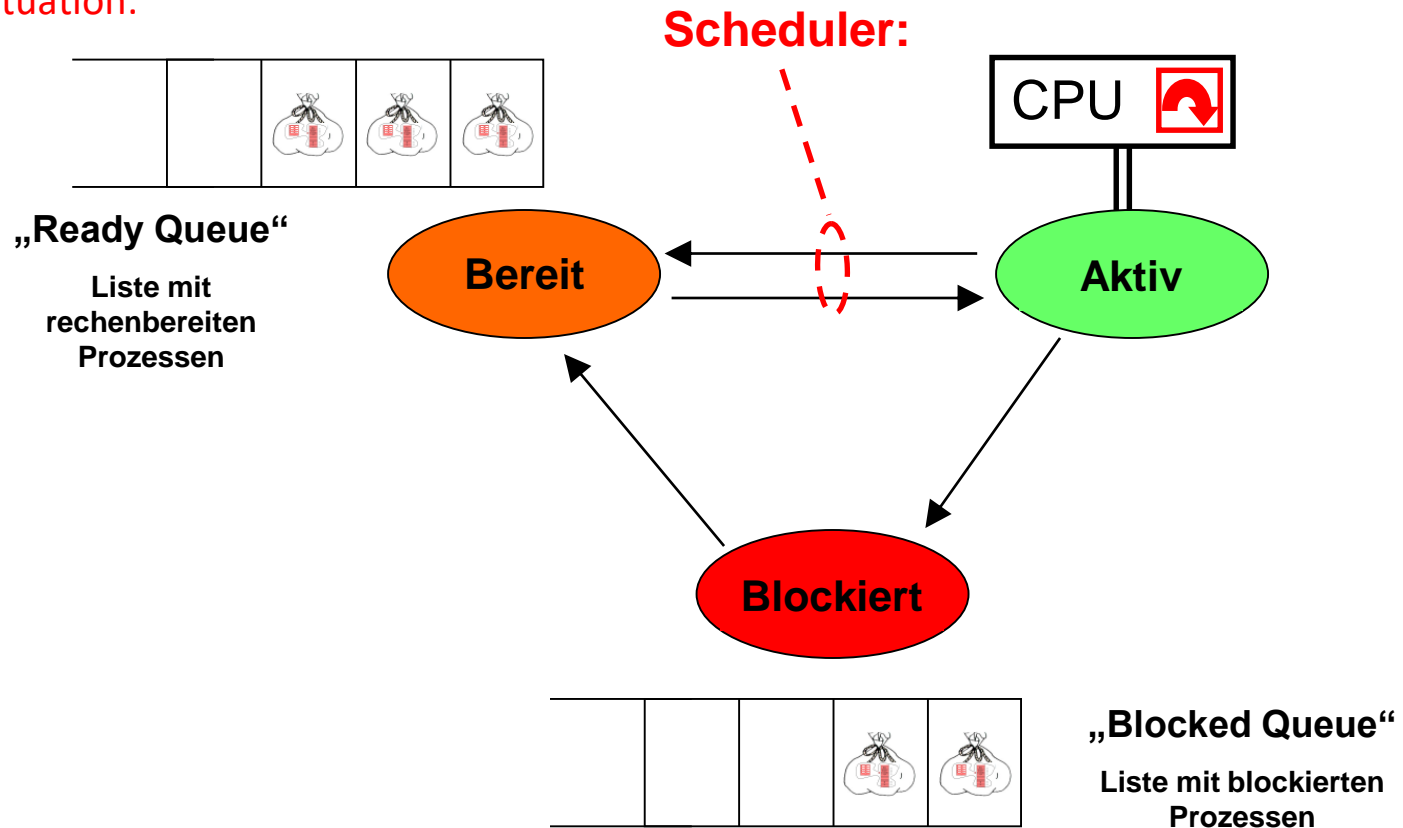
Scheduling

- 
- Einführung
 - Grundlegende Schedulingverfahren
 - Echtzeitscheduling
 - Scheduling in Mehrprozessorumgebungen
 - Scheduling in Windows
 - Scheduling in Linux

Der Scheduler

- Komponente, die bestimmt, welcher Prozess als nächstes die CPU erhält

Ausgangssituation:

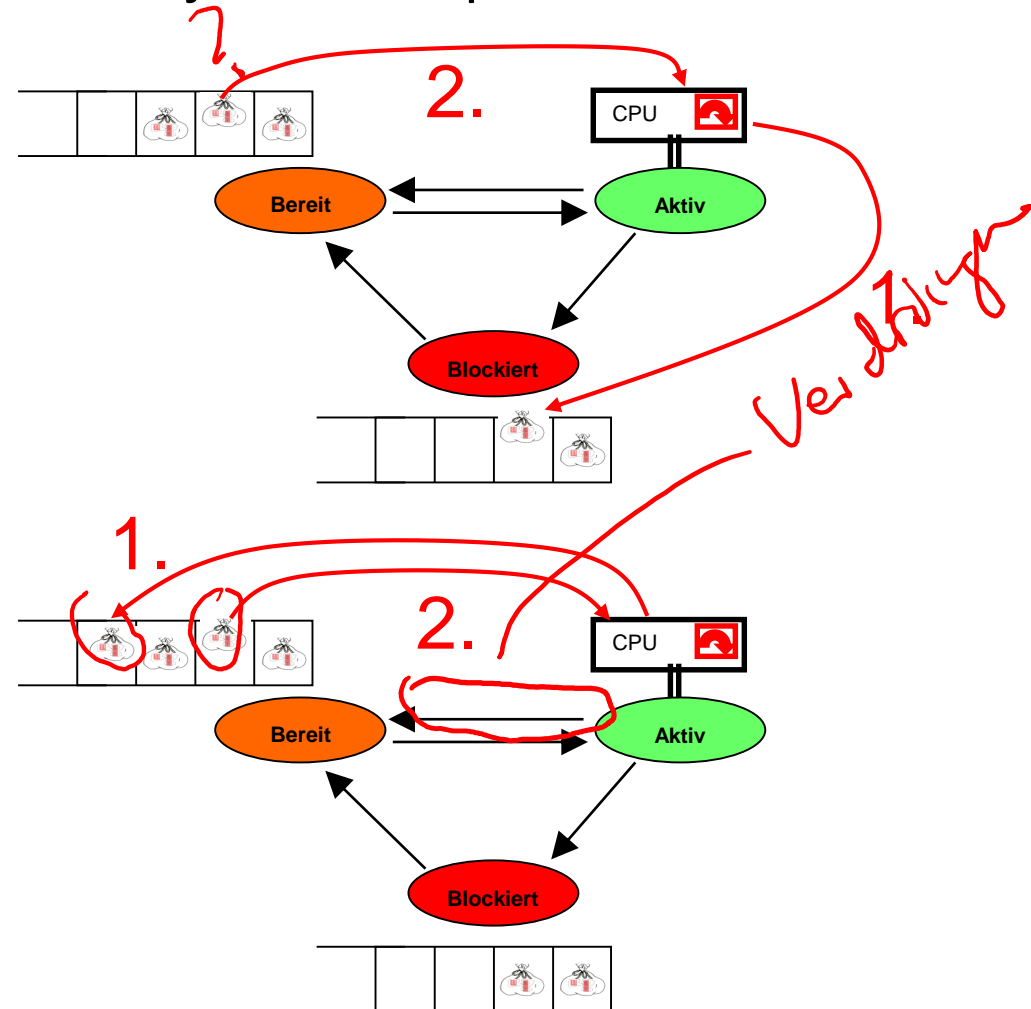


Aufgaben des Schedulers - 1

- Entscheider (Schiedsrichter) wer zu jedem Zeitpunkt rechnen darf

– Wenn ein bisher aktiver Prozess blockiert wird, entscheidet der Scheduler, welcher der bereiten Prozesse die CPU erhält

– Wenn ein Prozess schon für längere Zeit die CPU hat, entscheidet der Scheduler, ob bzw. wann dieser verdrängt werden soll.
(z.B. zyklische Überprüfung alle 100 ms)

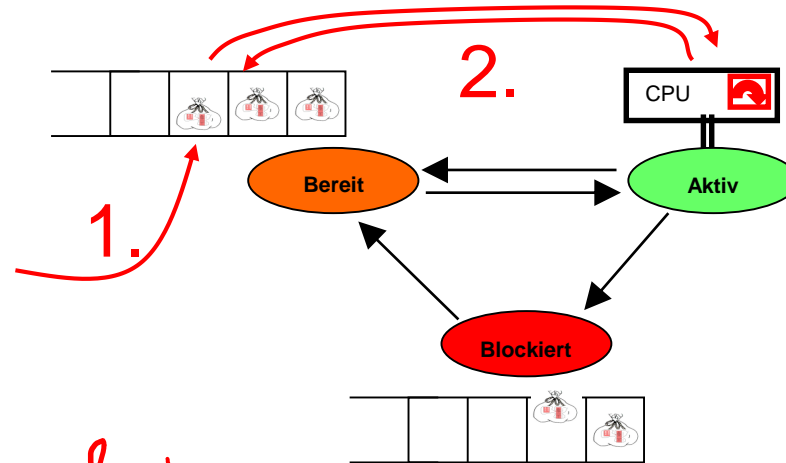


Aufgaben des Schedulers - 2

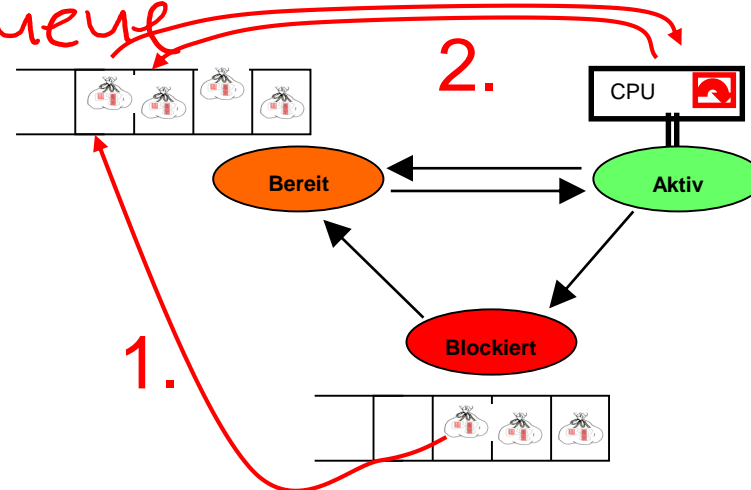
■ Weitere „Verdrängungs“-Entscheidungen:

- Wenn ein neuer Prozess dazukommt, entscheidet der Scheduler, ob der laufende Prozess verdrängt werden soll

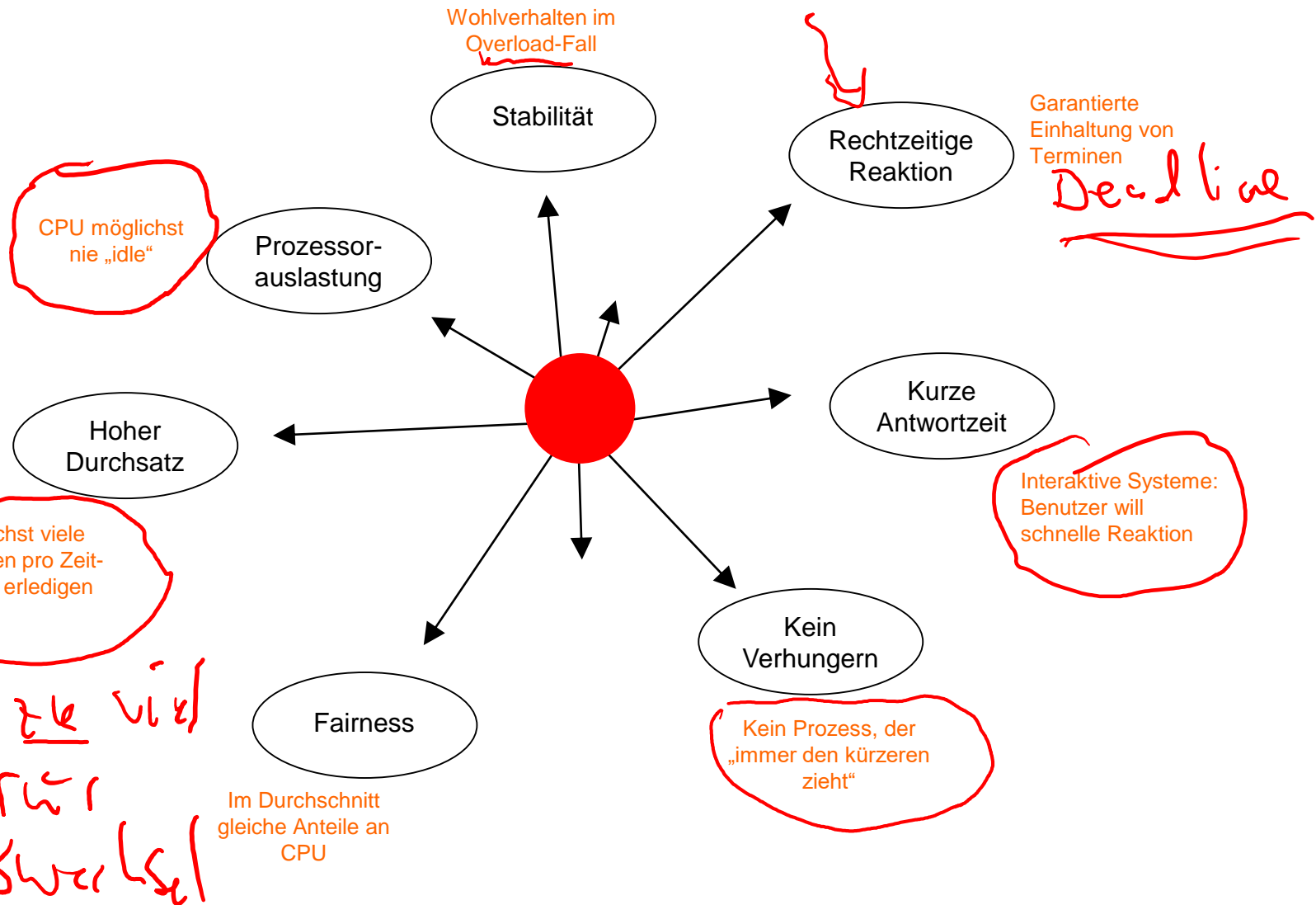
- Wenn ein blockierter Prozess wieder bereit wird, entscheidet der Scheduler, ob der laufende Prozess verdrängt werden soll



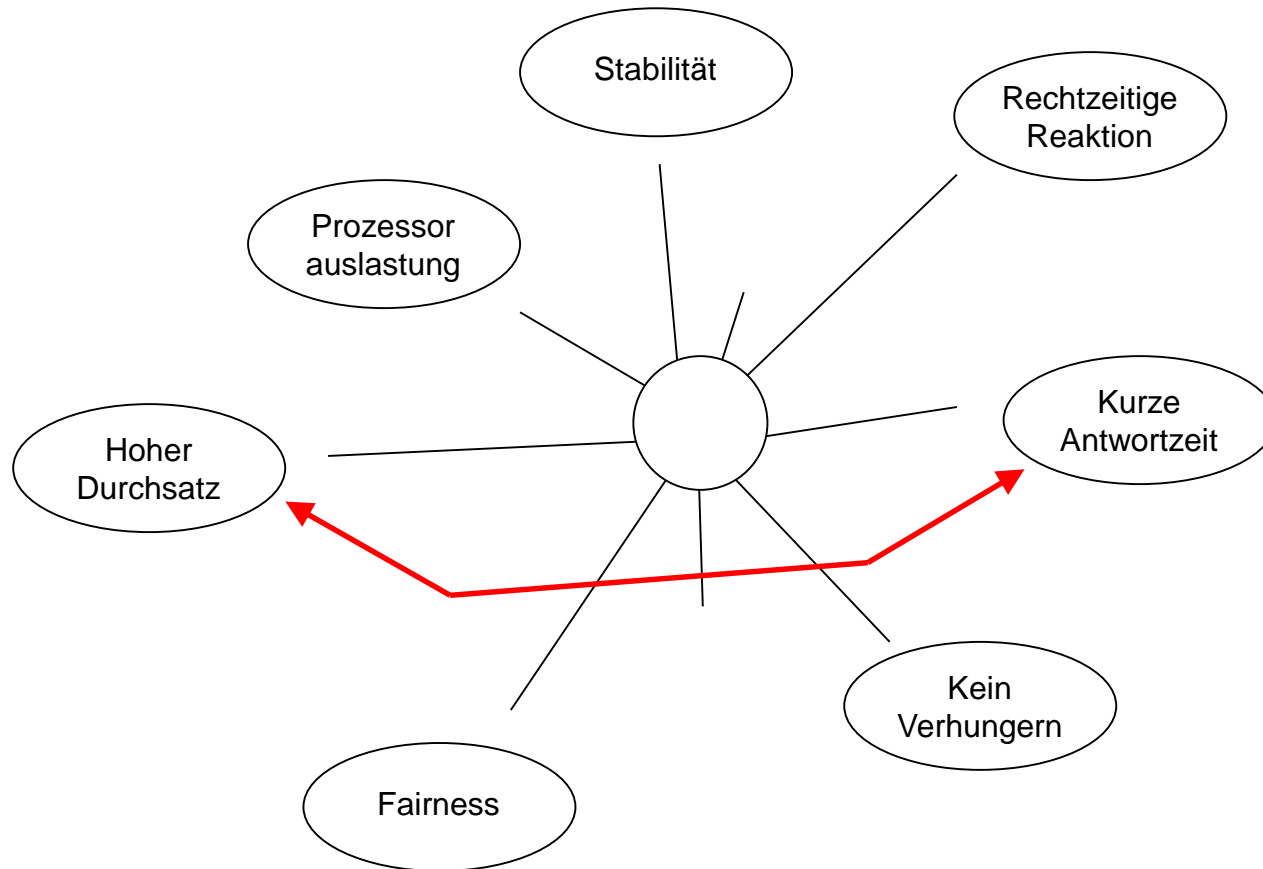
Ready-Queue



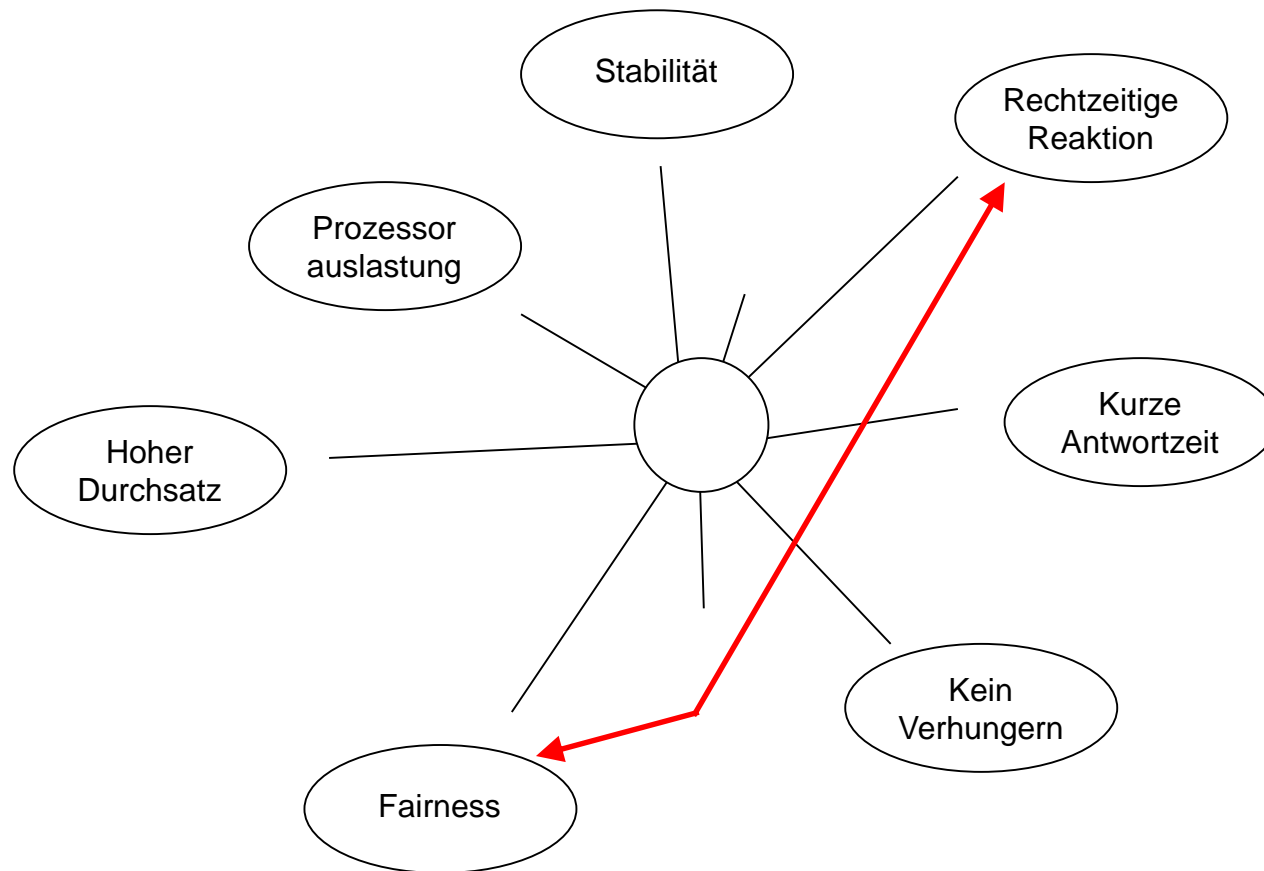
Ein Scheduler hat es schwer....



Beispiele für Zielkonflikte im Scheduling



Beispiele für Zielkonflikte im Scheduling



Was noch dazu kommt....

Scheduling selbst kann teuer sein:

rechen-
speicherintensiv

- Bestimmen eines Kandidaten und Verwaltung der Queues kostet Rechenzeit

- Je komplexer die Strategie, desto höher der Aufwand

- Prozesswechsel kostet Rechenzeit

- Übergang in Kernel Mode → Disput
- Sichern des alten Prozesskontextes + Laden des neuen (Register, Prozess-Handles, etc.)
- Verlust von Cache-Informationen (neuer Prozess muss Cache erst wieder neu laden)
- Hauptspeicherinhalte

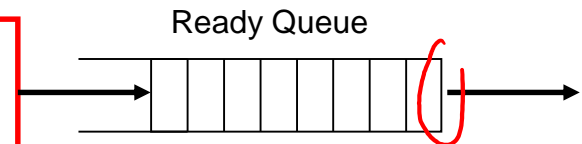
→ Scheduling-Strategien müssen sorgfältig ausgewählt werden

Beispiel für ein einfaches Scheduling-Verfahren

- FCFS (First Come First Serve)

- SCHEDULING-Regel:

Gib demjenigen Prozess die CPU,
der am längsten in der Ready Queue ist



- Hat ein Prozess die CPU, gibt er sie erst wieder ab, wenn er z.B. von einer I/O Operation blockiert wird (keine Verdrängung).

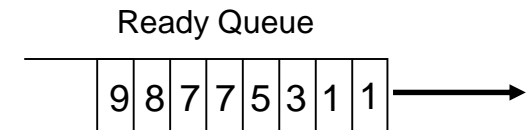
Was sind Vor- und Nachteile?

- + Einfacher Algorithmus; schnell, effizient
- + Optimaler Durchsatz; fair; kein Verhungern
- Schlechte Antwortzeiten
- Keine Gewährleistung einer rechtzeitigen Reaktion

Eine weitere einfache Scheduling-Strategie

- „Shortest Job First“ (SJF) Scheduling
- Job = (Teil-)Aufgabe eines Prozesses (auch „Task“)
- Anwendbar, wenn Bearbeitungszeit einer (Teil-)Aufgabe vorher bekannt

- SCHEDULING-Regel:
 Gib demjenigen Prozess die CPU,
 der die geringste Bearbeitungszeit hat



- Möglich **ohne** Verdrängung oder **mit** Verdrängung
 (z.B. wenn ein neuer kurzer Job hereinkommt)

- + Einfach zu realisieren, relativ billig (aber teurer als FCFS!)
- + Optimal in Bezug auf durchschnittliche Job-Wartezeiten
- + Potentiell gutes Verhalten für interaktive Nutzer und Echtzeitanwendungen

- Nicht fair

- Potentielles Verhungern

- i.d.R. Bearbeitungszeit vorab nicht bekannt

→ Monitoring der Prozess-Historie

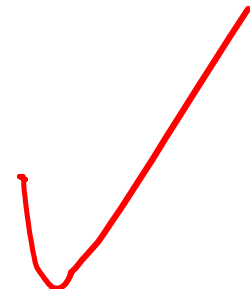
Shortest Remaining Time (SRT)

- Variante von Shortest Job First
 - Statt Gesamtlaufzeit wird „Restlaufzeit“ eines Jobs verwendet
(Remaining_time = Total_time - Consumed_time)
 - Mit Verdrängungsstrategie, d.h. neue kurze Jobs verdrängen längere Jobs, die noch eine höhere Restlaufzeit haben.
- + Erfüllt noch bessere durchschnittliche Job-Wartezeiten
- Ebenfalls nicht fair
 - Laufzeitverhalten muss bekannt sein
 - Gefahr des „Verhungerns“ von langen Jobs
im Falle von vielen statistisch gleichverteilten Kurzläufnern

Prioritätsbasiertes Scheduling

- Jedem Prozess wird eine **Priorität** zugeordnet
- SCHEDULING-Regel:
 1. Gib demjenigen Prozess mit der höchsten Priorität die CPU
 2. Verdränge dabei ggfs. Prozesse mit niedrigerer Priorität
 3. Arbeite innerhalb derselben Priorität nach FCFS oder SJF
- In unterschiedlichen Varianten in vielen heutigen Betriebssystemen eingesetzt (Linux, Windows,..)
- + Relativ einfache Struktur, effizient realisierbar
- + Gute Antwortzeiten (bei richtiger Priorisierung)
- + Fair (im Rahmen der Priorisierung)
 - ~~Gefahr des Verhungerns~~
 - Potentiell lange Wartezeiten für niedrigpriorisierte Prozesse
- Varianten
 - Statische Prioritäten (bei Prozessinitialisierung festgelegt)
 - Dynamische Prioritäten (passen sich zur Laufzeit an)

ähnlich wie bei SJF

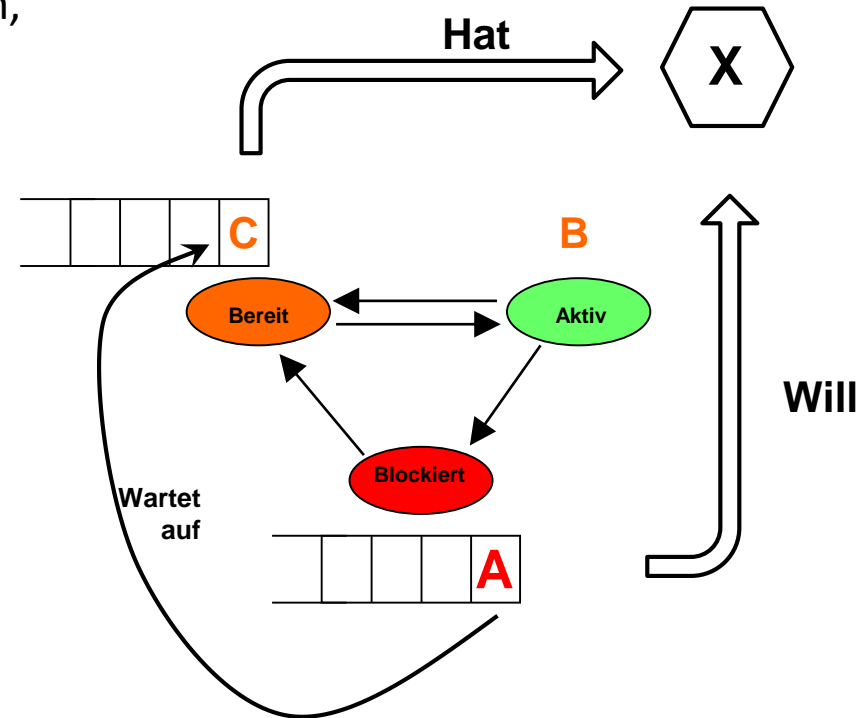


Problem „Prioritätsinvertierung“

- Zwischen Prozessen bestehen Abhängigkeiten, die dem Scheduler nicht bekannt sind

- Beispiel:

- Prozess **C** (niedrige Priorität) hat Betriebsmittel X, z.B. E/A-Gerät, ist in **Ready-Queue** und „bereit“, X wieder freizugeben
- Prozess **B** (mittlere Priorität) ist Langläufer und ist „aktiv“, d.h. hat die CPU
- Prozess **A** (hohe Priorität) will Betriebsmittel X, ist aber „blockiert“, bis **C** dieses freigibt.

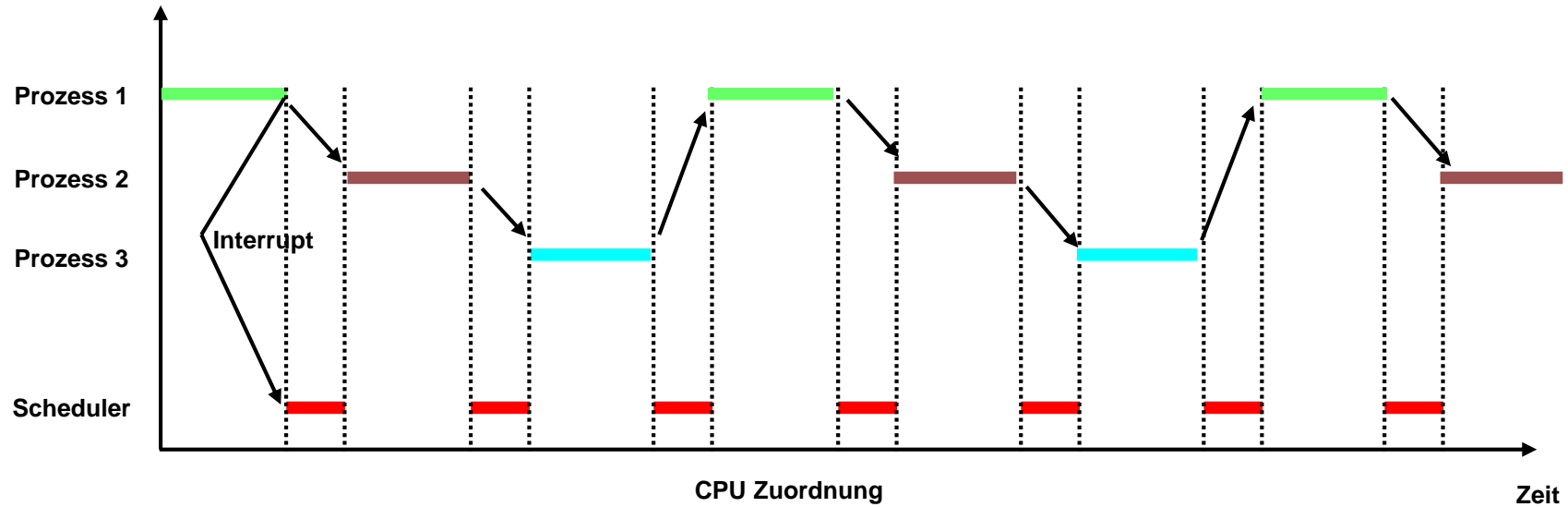


- **A** kommt trotz höherer Priorität nicht vor **B** und **C** an die Reihe
= „Prioritätsinvertierung“

Round Robin

- CPU „Time Sharing“ zwischen Prozessen
- Prozesse sind in einer Runde angeordnet und erhalten reihum jeweils eine Zeit lang die CPU (Zeitdauer = „Quantum“ oder „time-slice“)
- SCHEDULING-Regel:
 - Gib demjenigen Prozess die CPU, welcher der nächste im Zyklus ist.
 - Verdränge den Prozess, wenn sein CPU-Quantum überschritten ist.
- + Fair, kein Verhungern
- + Potentiell gutes Verhalten für interaktive Nutzer und Echtzeitanwendungen
- Verdrängung erzeugt in jedem Fall Overhead
- Effizienz hängt ab von Anzahl der Prozesswechsel
(kleines Quantum: größere Fairness; großes Quantum: größere Effizienz)
- Sinnvolle Werte: z.B. 10-100 ms
 - **Praktikum Versuch 4:** Standardwert(e) für Windows bestimmen!

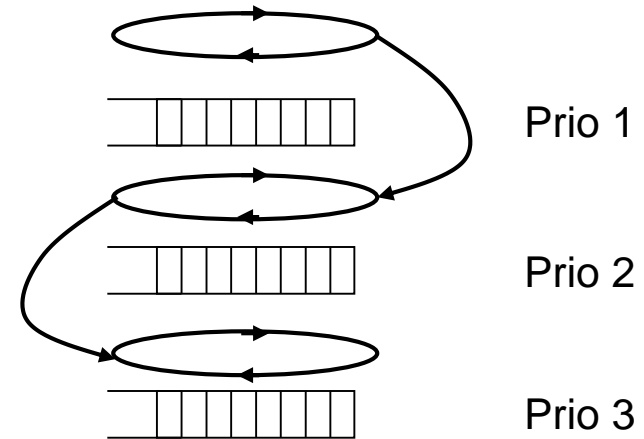
Round Robin im Zeitdiagramm



Prioritätsbasiertes Scheduling mit Round Robin

- **Kombination** von 2 Scheduling-Verfahren (häufig in der Praxis)
- Prozessen wird statisch oder dynamisch eine Priorität zugeordnet (**Prioritätsklassen** = Prozesse mit gleicher Priorität)

- **SCHEDULING-Regel:**
 1. Gib demjenigen Prozess mit der höchsten Priorität die CPU
 2. Verdränge dabei ggf. Prozesse mit niedrigerer Priorität;
 3. Arbeite innerhalb derselben Prioritätsklasse nach Round-Robin.



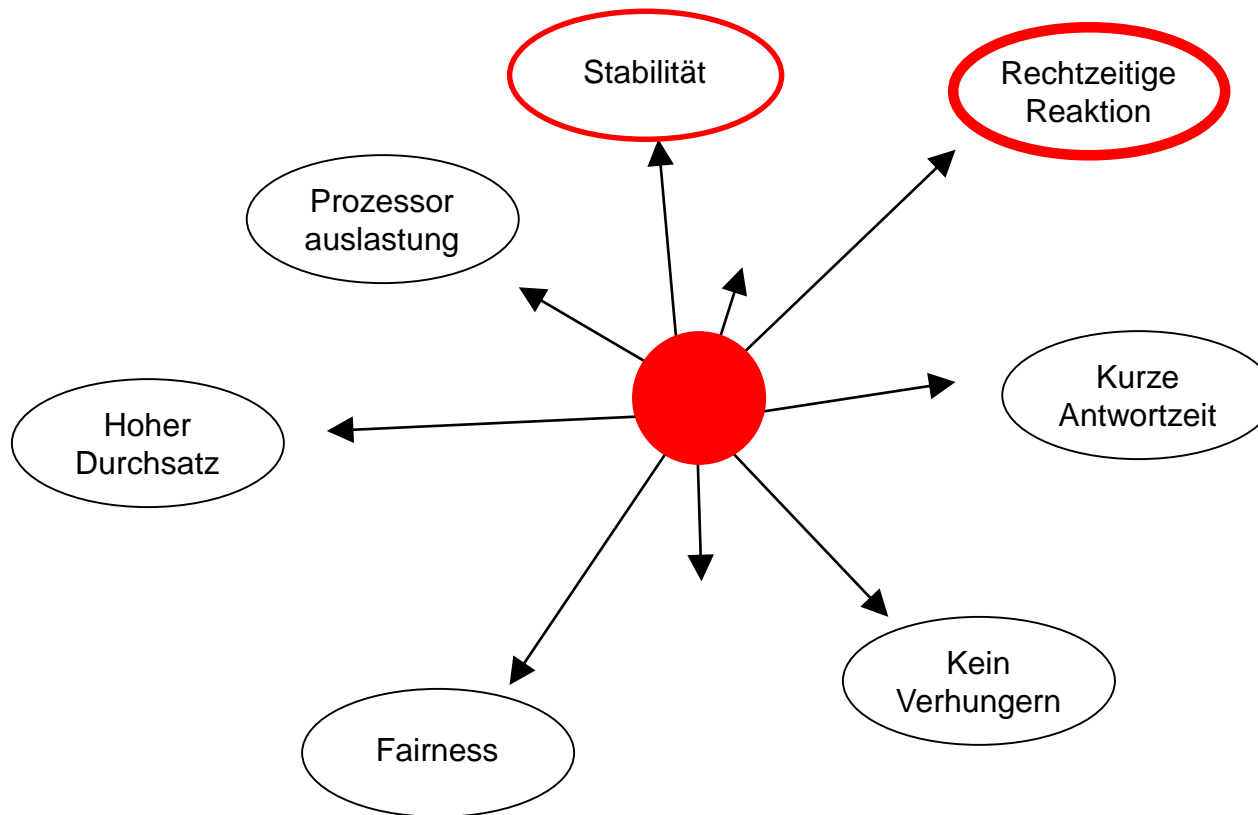
- + kombiniert Vorteile von beiden Strategien
- + Potentiell gutes Verhalten für interaktive Nutzer und Echtzeitanwendungen
- Gefahr von Verhungern

Echtzeit-Scheduling

- **Weiche** Echtzeit: Verletzung von Rechtzeitigkeit
→ **Verminderter Nutzen**
- **Harte** Echtzeit: Verletzung von Rechtzeitigkeit
→ **Kritischer Schaden**
- Aktionen werden bis zu festgelegten Zeitpunkten („**Deadlines**“) oder innerhalb festgelegter Zeitintervalle („**Periods**“) benötigt
- System muss auf externe Ereignisse (Events) rechtzeitig reagieren
- System muss auf Zeitereignisse (Timeouts) rechtzeitig reagieren
- Zentrales Lösungselement:
Echtzeitbetriebssystem mit **Echtzeit-Scheduler**

Anforderungen an Echtzeit-Scheduler - I

- Andere Gewichtung als bei Nicht-Echtzeitsystemen

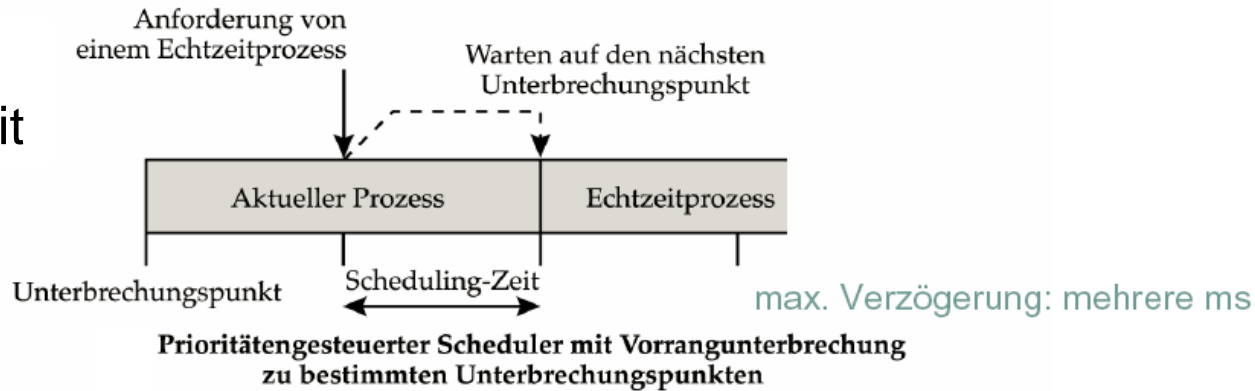


Anforderungen an Echtzeit-Scheduler - II

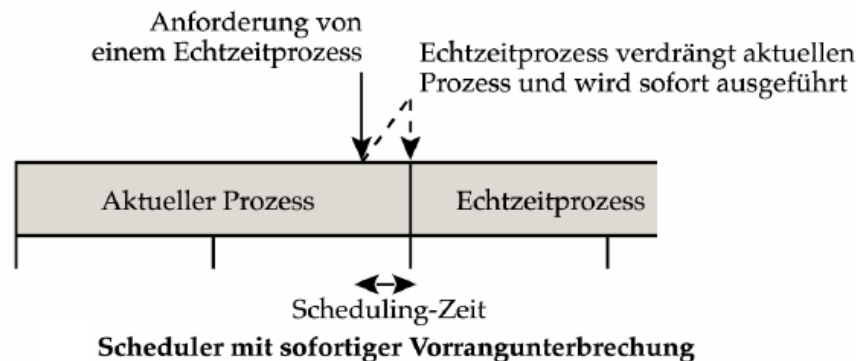
- Klein und kompakt!
 - Schnelle Prozessumschaltung (wenig Kontext)
 - Schnelle Interruptbehandlung (kurze Sperren)
- Unterstützung von **periodisch wiederkehrenden Aufgaben** (z.B. Sensorabfragen)
- Stärkere Einflussnahme des Benutzers (Programmierers) auf die Kontrolle der Ausführungsabfolge (Vorgabe von Prioritäten, Deadlines, Periodendauer, etc.)
- Stabiles Verhalten auch im Overload-Bereich und bei Teilausfällen des Systems wichtig

Anforderungen an Echtzeit-Scheduler - III

nicht
Echtzeit



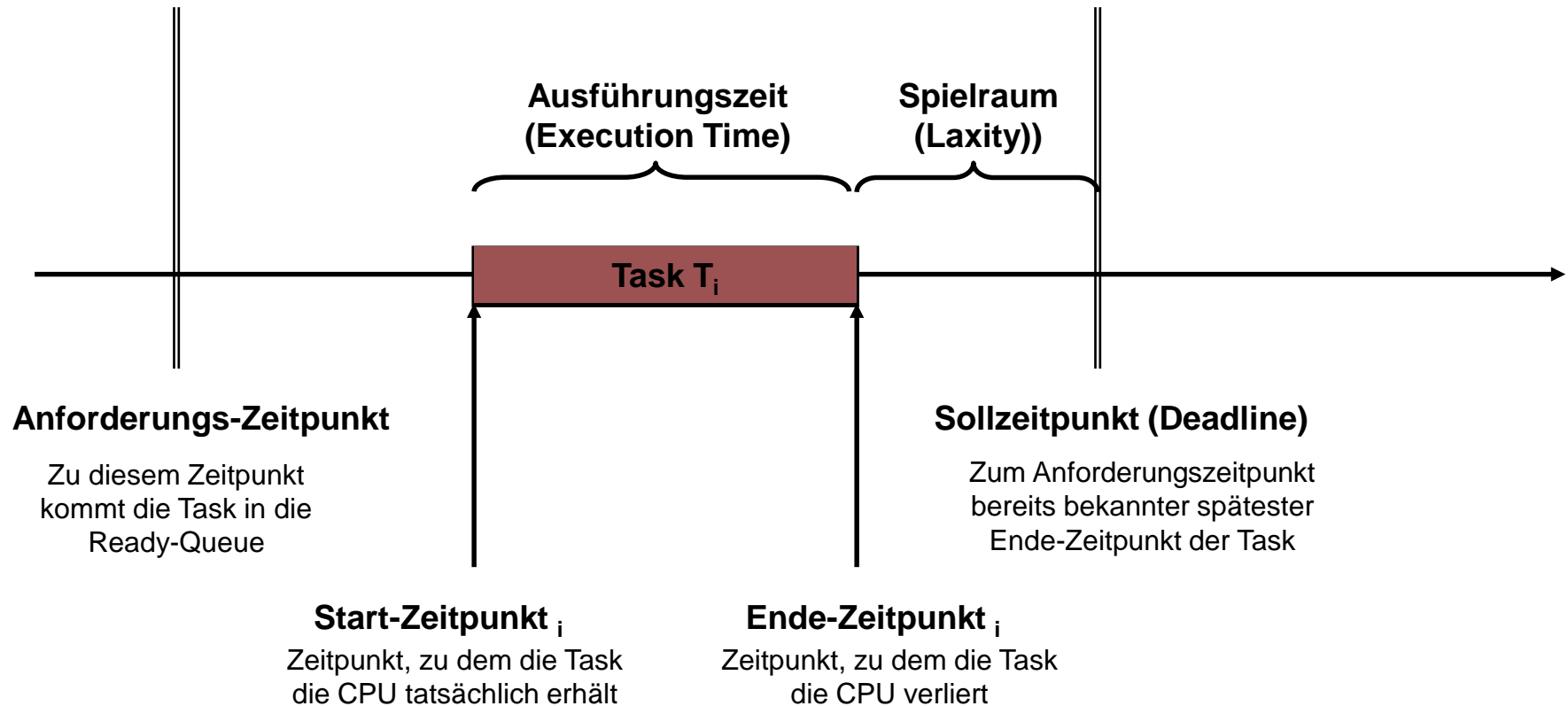
Echtzeit



max. Zuteilungsverzögerung 100 μ s

Quelle: [Stal03]

Modell einer Echtzeittask

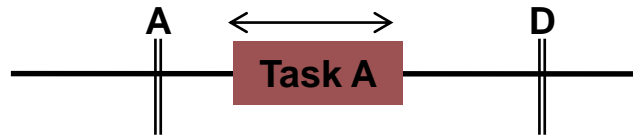


- Periodische Tasks: zyklisch wiederkehrend (Periode P)
- Aperiodische Tasks: ereignisgesteuert, unvorhersehbar

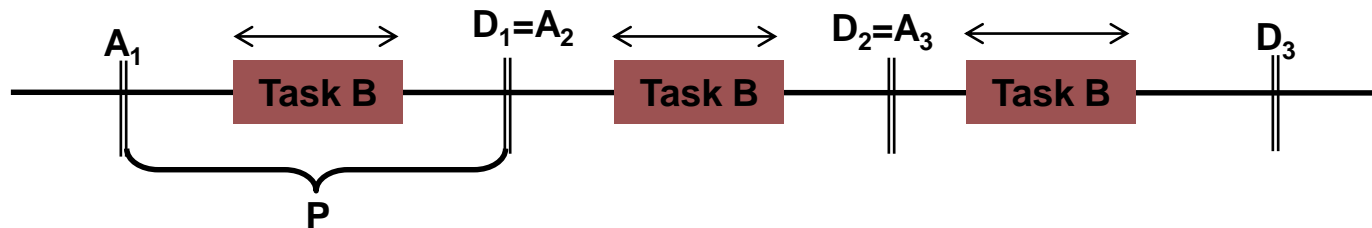
Anforderungen an Echtzeit-Scheduler - II

■ Unterstützung von aperiodischen und periodischen „Tasks“

- Aperiodische Tasks: ereignisgesteuert, unvorhersehbar
(Anforderungszeit A, Deadline D)



- Periodische Tasks: zyklisch wiederkehrend (z.B. Sensorabfrage)
(Periode P, $D_i = A_i + P$, $A_{i+1} = D_i$)



Echtzeit-Scheduling – Unterscheidungsmerkmale

■ Statisches Scheduling vs. Dynamisches Scheduling

Statisches Scheduling	<ul style="list-style-type: none"> • A-priori Wissen über Prozessanforderungen (Startzeiten, Ausführungszeiten, ..) vorhanden • Vorabplanung aller Entscheidungen möglich (z.B. tabellenorientiert)
Dynamisches Scheduling	<ul style="list-style-type: none"> • Prozessanforderungen erst zur Laufzeit bekannt • Ad-Hoc Entscheidungen in Realzeit nötig

■ Scheduling mit oder ohne Verdrängung

Scheduling mit Verdrängung (Preemption)	<ul style="list-style-type: none"> • Einem „aktiven“ Prozess kann die CPU weggenommen werden, um diese einem anderen „bereiten“ Prozess zuzuordnen
Scheduling ohne Verdrängung (no Preemption)	<ul style="list-style-type: none"> • Hat ein Prozess die CPU, so wird sie ihm nicht weggenommen, solange er sich nicht selbst blockiert (z.B. durch I/O Wait)

Echtzeit Scheduling – Unterscheidungsmerkmale

■ Scheduling und Prioritäten

Scheduling ohne Prioritäten	Scheduling erfolgt nicht prioritätsgesteuert
Statische Prioritäten	Prioritäten eines Prozesses werden einmal zu Beginn der Ausführung festgelegt und zur Laufzeit nicht verändert
Dynamische Prioritäten	Prioritäten können zur Laufzeit an neue Anforderungen angepasst werden.

Echtzeitscheduling: Earliest Deadline First (EDF)

- Voraussetzungen:
 - Aperiodische oder periodische Tasks
 - Für jeden Prozess in der Ready-Queue ist die nächste Deadline bekannt

- SCHEDULING-Regel:

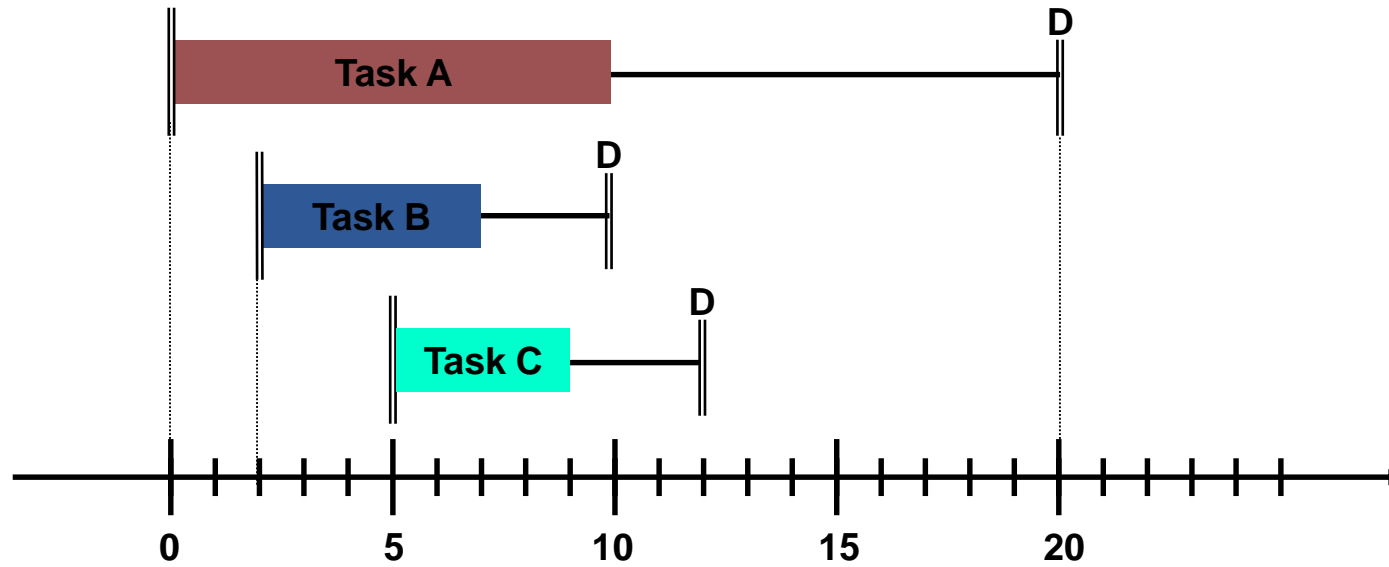
Gib demjenigen Prozess die CPU, der die nächste Deadline hat;
verdränge dabei ggfs. den gerade laufenden Prozess

- Klassifizierung:

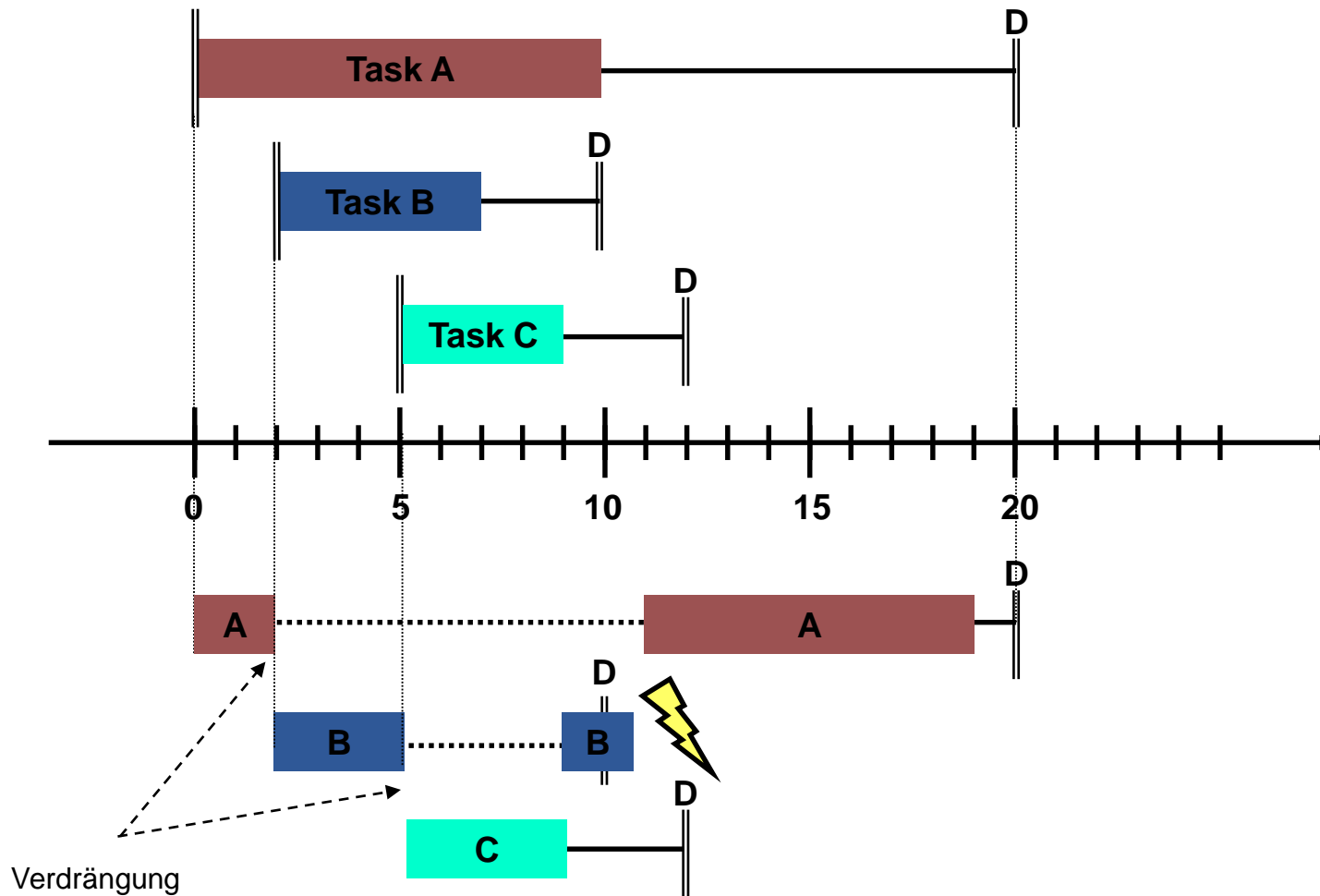
Statisch	Ohne Verdrängung	Keine Prioritäten
Dynamisch	Mit Verdrängung	Statische Prioritäten
		Dynamische Prioritäten

- Mittelgroßer Aufwand zur Implementierung der Strategie
 - Verwalten einer Liste aller Prozesse mit (aufsteigenden) Deadlines
 - Neue Prozesse müssen „einsortiert“ werden
 - Ggf. Verdrängung, wenn neu in die Ready-Queue aufgenommene Prozesse/Tasks frühere Deadlines haben.

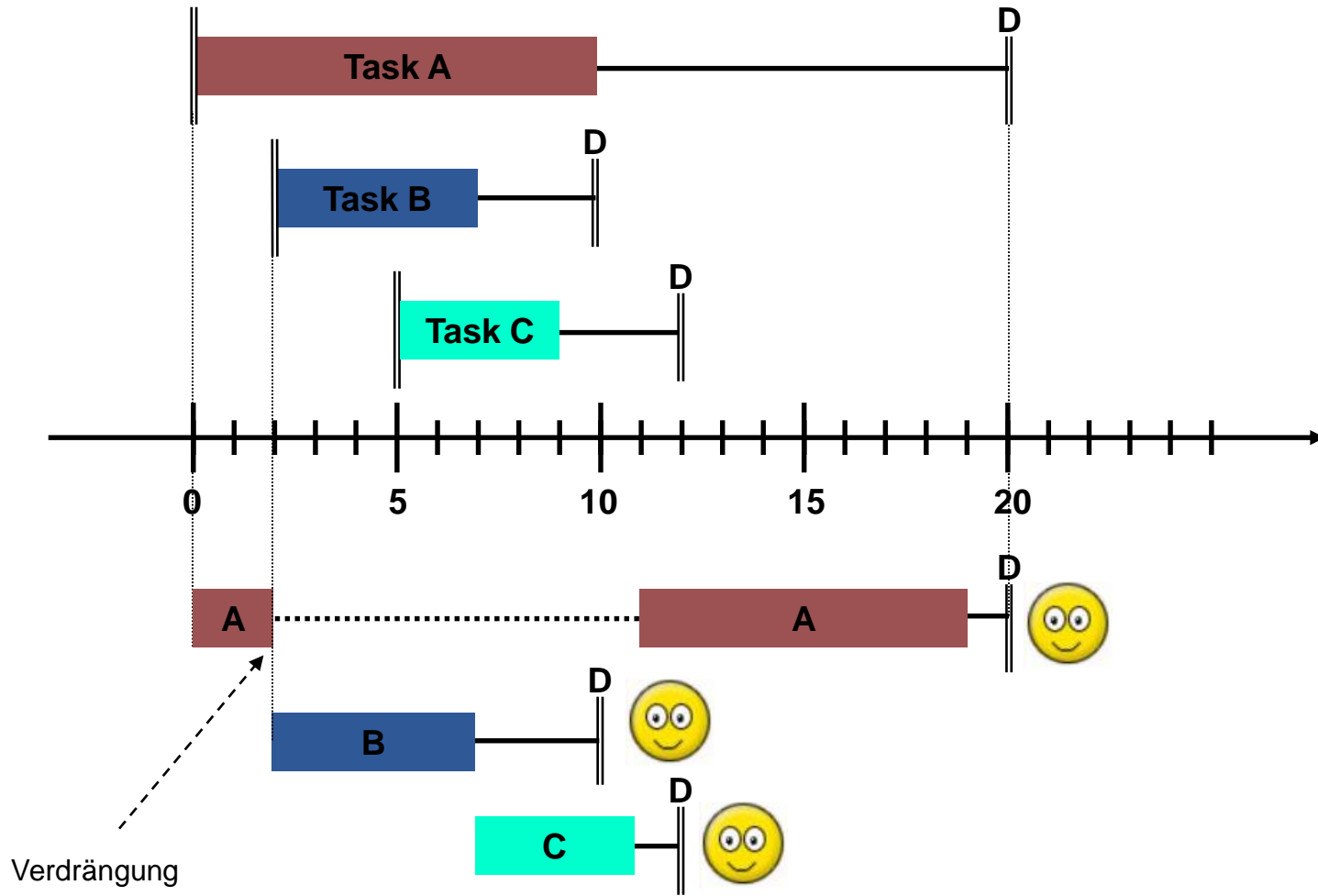
Beispiel



Beispiel – Statische Prioritäten ($A < B < C$)



Beispiel – EDF



EDF – Bewertung

+ EDF ist in der Lage, auf einem Einprozessorsystem ein

„OPTIMALES“ Scheduling

zu implementieren, d.h.

EDF **findet** für eine gegebene Menge von Tasks eine **Abfolge**, welche **alle Deadlines einhält**, **sofern es prinzipiell eine solche Abfolge gibt**.

Nachteile:

- Aufwand für Entscheidungsfindung
- Nicht optimal, falls auch Startzeit wichtig (auf Endezeit optimiert)
- Nicht optimal in Überlastsituationen (nahe 100% Auslastung);
instabil, d.h. Überschreiten *einer* Zeitschranke bleibt nicht isoliert

Rate Monotonic Scheduling (RMS)

- Voraussetzungen:
 - Periodische Tasks
 - Initiale Zuordnung von Prioritäten zu Prozessen gemäß ihrer Task-Periode
 - Kürzere Periode \leftrightarrow Höhere Priorität ($PRIO_i = 1 / P_i$)

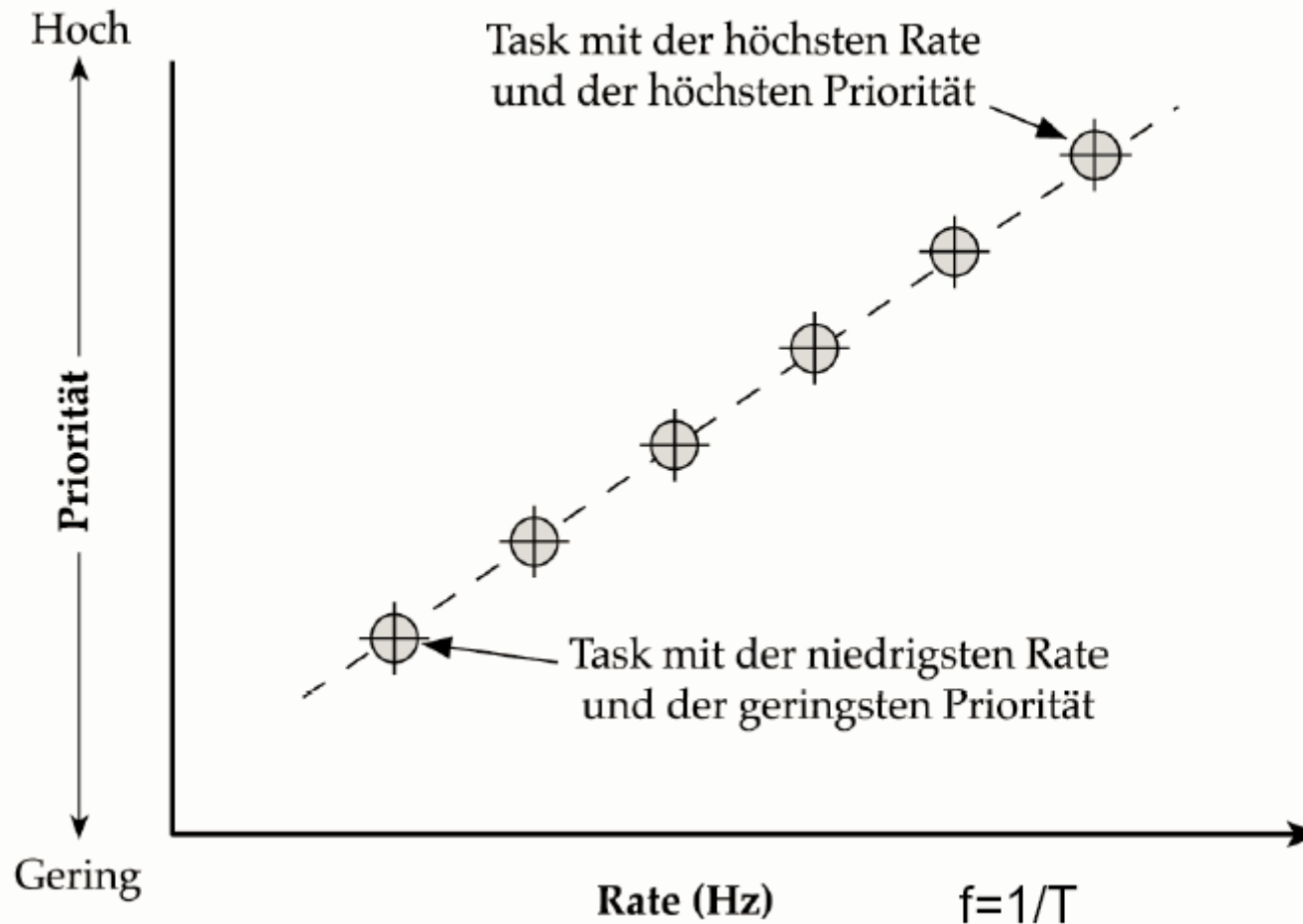
- SCHEDULING-Regel:
 Gib demjenigen lauffähigen Prozess die CPU, der die kürzeste Periode hat; verdränge dabei ggfs. den gerade laufenden Prozess

- Klassifizierung:

Statisch	Ohne Verdrängung	Keine Prioritäten
Dynamisch	Mit Verdrängung	Statische Prioritäten
		Dynamische Prioritäten

- Geringer Aufwand zur Implementierung der Strategie
 - Priorität wird pro Prozess nur einmal berechnet (Startup)
 - Ggfs. Verdrängung, wenn neu in die Ready-Queue aufgenommene Prozesse höhere Priorität haben.

RMS Prioritäten



Quelle: [Stal03]

RMS – Optimalitätssatz

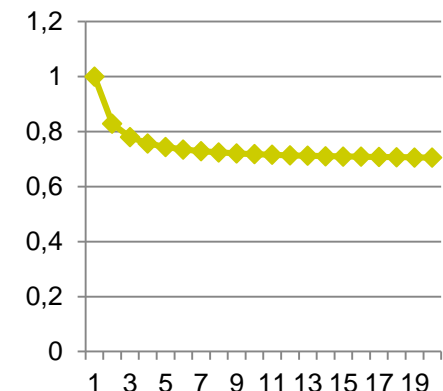
- Annahme: n periodische Prozesse
- Prozess i hat Periode P_i und Ausführungszeit C_i
- **Notwendiges** (aber **nicht hinreichendes**!) Kriterium für ein erfolgreiches Scheduling (d.h. Einhalten aller Deadlines) ist, dass

$$C_1/P_1 + C_2/P_2 + \dots C_n/P_n \leq 1$$

- Man kann nachweisen, dass RMS optimal ist, d.h. **alle Deadlines einhält**, **wenn die Tasks** die CPU nicht zu 100% beanspruchen, sondern **eine gewisse Reserve lassen** (**hinreichendes Kriterium**).
- Genauer: RMS ist optimal, wenn

$$C_1/P_1 + C_2/P_2 + \dots C_n/P_n < n \cdot (2^{1/n} - 1)$$

- Beispiel:
 - bei 2 Prozessen: wenn $< 82,8 \%$
 - bei 3 Prozessen: wenn $< 77,9 \%$
 - bei ∞ Prozessen: wenn $< \sim 69,3 \%$ ($\ln 2$)



RMS – Bewertung

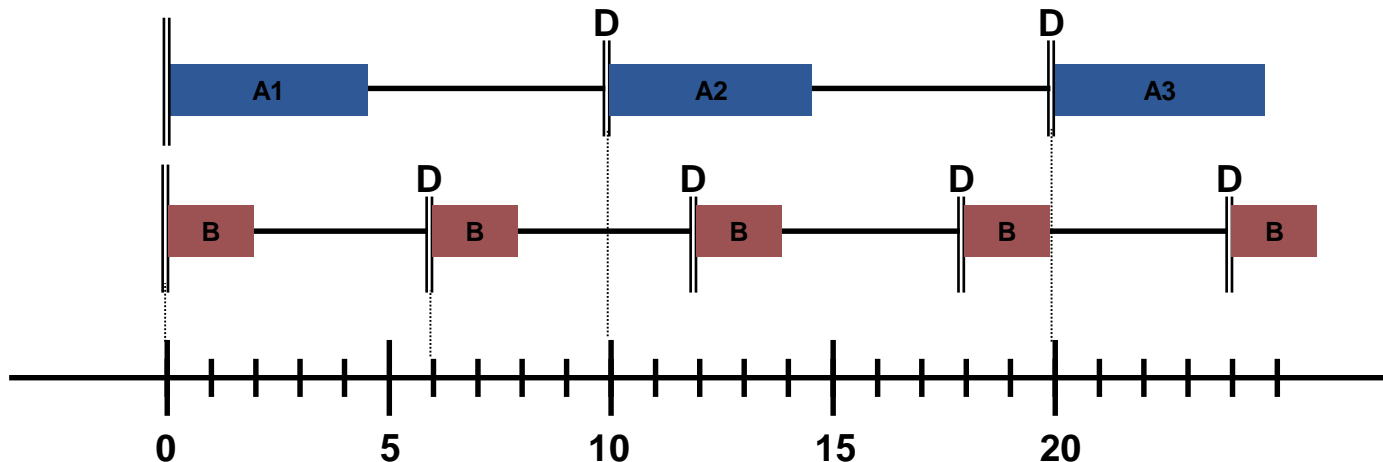
- + RMS ist sehr einfach und effizient
(nur 1x Berechnung der Priorität für jeden Prozess)
- + RMS ist optimal, d.h. hält in jedem Fall alle Deadlines ein,
wenn die Gesamtauslastung der CPU nicht zu hoch ist
(siehe vorige Folie)
- RMS kann versagen bei hoher Prozessorauslastung ($> 69,3\%$)
Aber: Häufig hat eine Anwendung auch „weiche Tasks“
Diese kann man niedrigst priorisieren, d.h. sie nehmen die Reserve nur in Anspruch wenn die harten Tasks sie nicht brauchen („Lückenfüller“).
- RMS ist **nur für periodische Prozesse**

RMS – Beispiel

Prozess	Periode	Ausführungszeit	PRIORITÄT
A	10 msec	4,5 msec	$1/10 = 0,1$
B	6 msec	2,0 msec	$1/6 = 0,17$

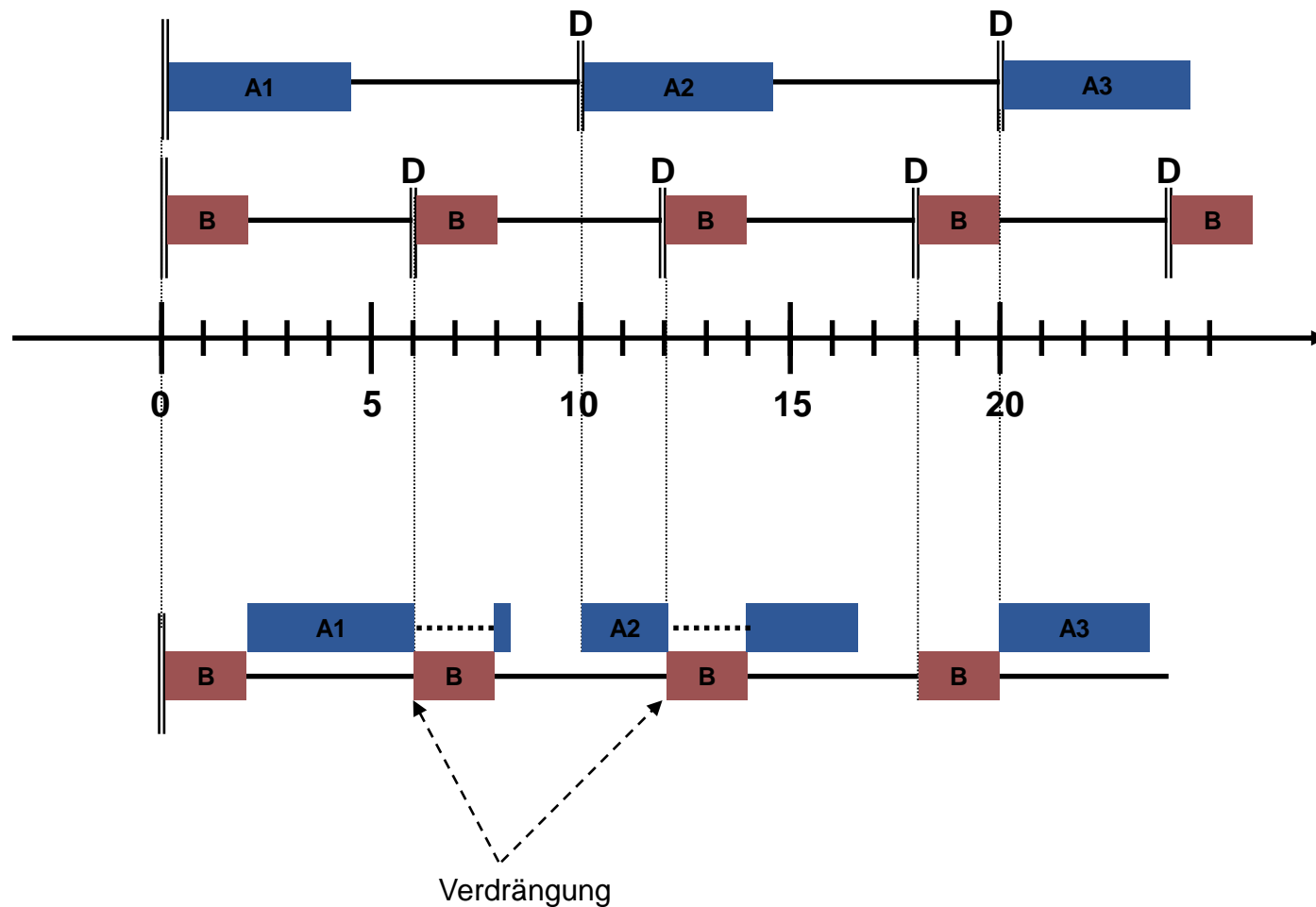
← Prio 2

← Prio 1



- Sind Echtzeitbedingungen prinzipiell erfüllbar? Reicht Rechenkapazität?
 - A: $4,5/10 \rightarrow 45\%$, B: $2/6 \rightarrow 33\%$; $45\% + 33\% = 78\% < 100\%$ **OK**
- Findet RMS-Algorithmus in jedem Fall ein passendes Scheduling?
 - $78\% < 82,8\%$ (2 Prozesse) **OK**

RMS - Beispiel



Weitere Echtzeit-Scheduling-Verfahren

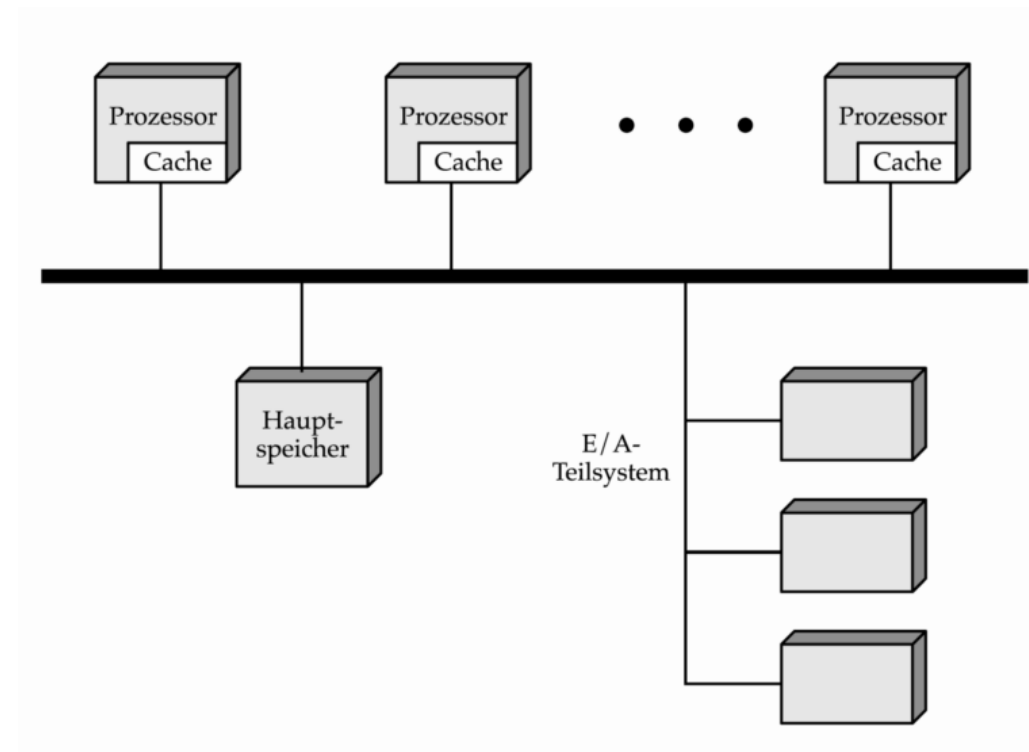
- **Least-Laxity-First Scheduling**
 - Laxity = „Spielraum“, Reserve einer Task
 - Wie EDF, aber es wird nicht nur die nächste Deadline, sondern auch die noch anstehende Rechenzeit eingerechnet.
 - Besseres Überlastverhalten als EDF, aber auch mehr Prozesswechsel und höherer Rechenaufwand zum Aktualisieren der Laxity
- **Time-Slice-Scheduling**
 - Prozesse kommen zyklisch reihum mit kurzen Zeitscheiben an die Reihe
 - Zeitscheibenlänge proportional zum Anteil der Prozessorauslastung durch jeweiligen Prozess
- Viele Verfahren aus Nicht-Echtzeitbetriebssystemen sind ebenfalls bedingt tauglich für Echtzeitbetriebssysteme
 - Round-Robin Verfahren
 - Shortest-Job-First Verfahren
- Häufig: **Kombinationen** von Verfahren (z.B. prioritätenbasiert, adaptiv)
- Weitere Möglichkeit in der Praxis: Standardbetriebssysteme verwenden, aber genügend Reserven einplanen. (CPU < 50%).
 - v.a. für **weiche Echtzeit**

Scheduling in Mehrprozessorumgebungen

- Typen von Mehrprozessorumgebungen:
 - Fall I: N -Prozessor-Systeme
 - Fall II: Hyperthreading-Prozessoren
 - Fall III: Multi-Core-Prozessoren
- II+III simulieren CPUs → „Virtuelle CPUs“
- **Zusätzliche Anforderungen** an das Scheduling in Mehrprozessorumgebungen:
 - Möglichst optimale Ausnutzung **aller** CPU-Ressourcen
 - Möglichst wenig Prozessorwechsel (Verlust des Kontextes, z.B. Registerinhalte, Cacheinhalte, etc.) „**Prozessoraffinität**“
 - **Prozessfamilien** können/sollen optimaler unterstützt werden
- Es kann in einer Mehrprozessorumgebung sinnvoll sein, CPUs kurzzeitig ungenutzt zu lassen, um einen teuren Kontextwechsel zu vermeiden und Prozessfamilien nicht zu trennen

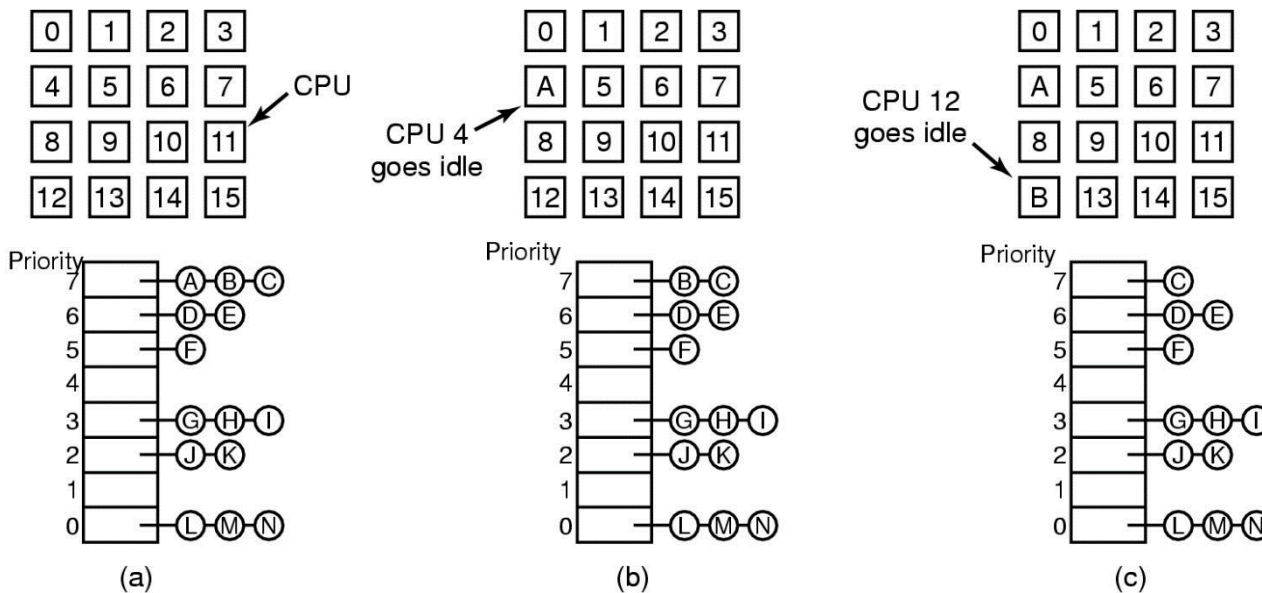
Symmetrisches Mehrprozessorsystem

- Alle Prozessoren sind gleichberechtigt (im Gegensatz zu: Master/Slave)
- Alle Prozessoren können Betriebssystemcode ausführen
- Prozessoren haben individuelle Register + CPU Cache (→ „dirty reads“!)
- Prozessoren teilen sich Hauptspeicher, E/A und externen Bus



Standard Time-Sharing Scheduling

- Ähnlicher Ansatz wie beim Einprozessorbetrieb
- **SCHEDULING-Regel:**
 Falls eine CPU frei wird, vergib sie an denjenigen bereiten Prozess/Thread mit der aktuell höchsten Priorität.
 Verdränge einen Prozess/Thread nach einer vollen Zeitscheibe



Quelle: [Tan02]

Standard Time-Sharing Scheduling

- + einfaches Verfahren, leicht erweiterbar von Einprozessorbetrieb
- + stellt sicher, dass keine CPU „idle“ ist, während gleichzeitig Prozesse/Threads bereit wären
- Prozesse werden beliebig zwischen Prozessoren verschoben („job hopping“) → sie verlieren jedes Mal ihren kompletten Kontext (CPU Cache, ..) und müssen diesen neu aufbauen.

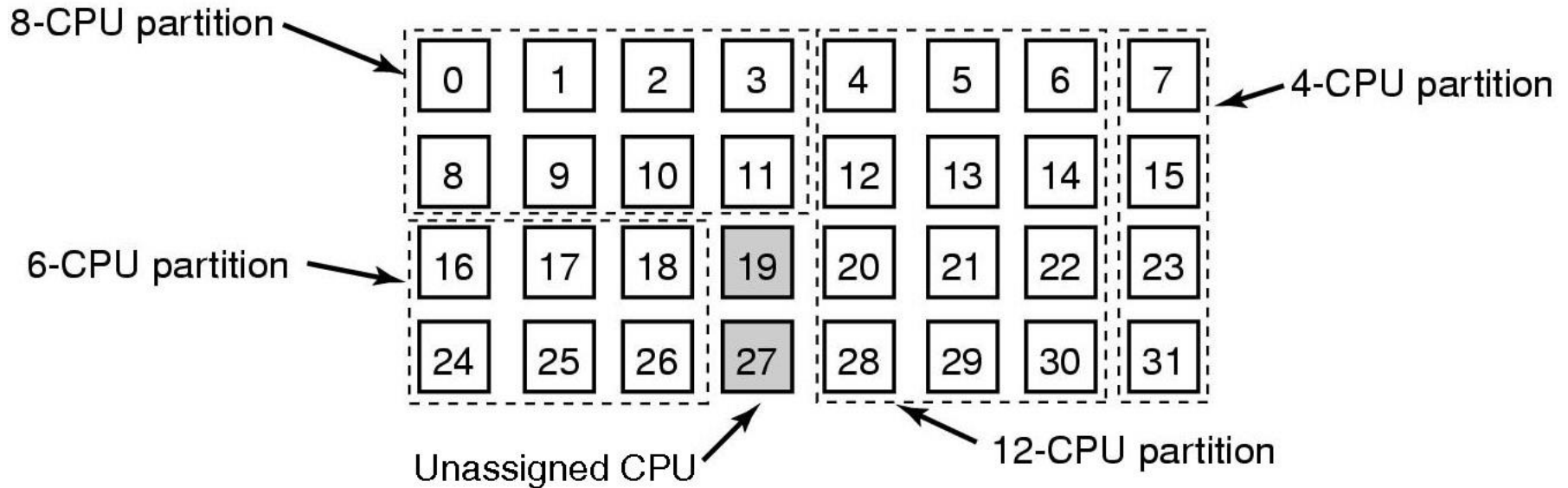
⇒ „Affinity Scheduling“, „Two-Level Scheduling“:

- First Level: Prozessor hat Menge von „preferred“ Prozessen (Scheduling zwischen diesen Prozessen pro Prozessor)
- Second Level: Hat ein Prozessor keine eigenen Prozesse/Threads mehr, die bereit wären, „holt“ er sich Prozesse/Threads von anderen Prozessoren
- + Lastverteilung
- + Cache-Affinität wird berücksichtigt
- + geringere Konkurrenz um Prozessliste
- - komplexeres Verfahren

Gang Scheduling

- Motivation: Häufig arbeitet Gruppe von Prozessen oder Threads eng zusammen: „Prozessfamilien“, „Threadgruppen“
- Wenn die Einzelprozesse bzw. (-threads) unabhängig ge-scheduled werden, so ist dies ineffizient → **Beispiel Tafel**
- Idee: Alle Prozesse/Threads einer Prozessfamilie zusammen (gleichzeitig) laufen lassen
- **SCHEDULING-Regel:**
Falls eine Prozessfamilie mit k Mitgliedern „bereit“ ist und genügend Prozessoren frei sind, vergib k dieser Prozessoren als Partition für eine Zeitscheibe lang an die komplette Prozessfamilie.
Verdränge die komplette Prozessfamilie nach Ablauf der Zeitscheibe
- Variante: „Space Sharing“: Partitionen werden zugeordnet, ohne Zeitgleichheit zu erzwingen (Raum aber nicht Raum/Zeit Sharing)

Beispiel Gang Scheduling – Raum-Achse



- Zu jedem Zeitpunkt ist die Menge der CPUs partitioniert in (verschieden große) Teilmengen.
- Scheduling sucht nach nicht zugeordneten Teilmengen hinreichender Größe

Quelle: [Tan02]

Beispiel Gang Scheduling – Zeit-Achse

		CPU					
		0	1	2	3	4	5
Time slot	0	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
	1	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
	2	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀
	3	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆
	4	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
	5	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
	6	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀
	7	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆

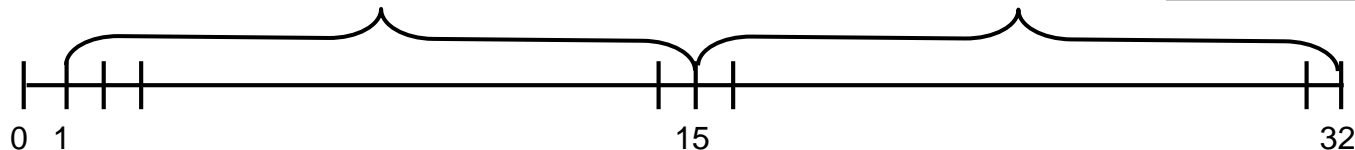
Quelle: [Tan02]

- In jedem Zeitintervall (Zeitscheibe) sind entweder alle oder keine Elemente einer Prozess(Thread-)familie aktiv
- Wenn einzelne Prozesse während dieser Zeitscheibe blockiert werden, so wird die CPU nicht re-scheduled
- Ineffizienzen im lokalen Bereich werden überkompensiert durch höhere Effizienz in Bezug auf Anzahl Prozesswechsel, Caching, etc.

Scheduling in Windows XP - I

- Windows XP verwendet ein **prioritätsbasiertes, verdrängendes** Scheduling-Verfahren
- Scheduling ist thread-basiert, nicht prozess-basiert (Thread-Prozess-Zuordnung sekundär)
- Jeder Thread kann eine Priorität von 0..32 haben
 - 16-32: Echtzeitprioritäten (i.d.R. nur vom System)
 - 1-15: Nichtechtzeitprioritäten (User-Threads)
 - 0: reserviert für System

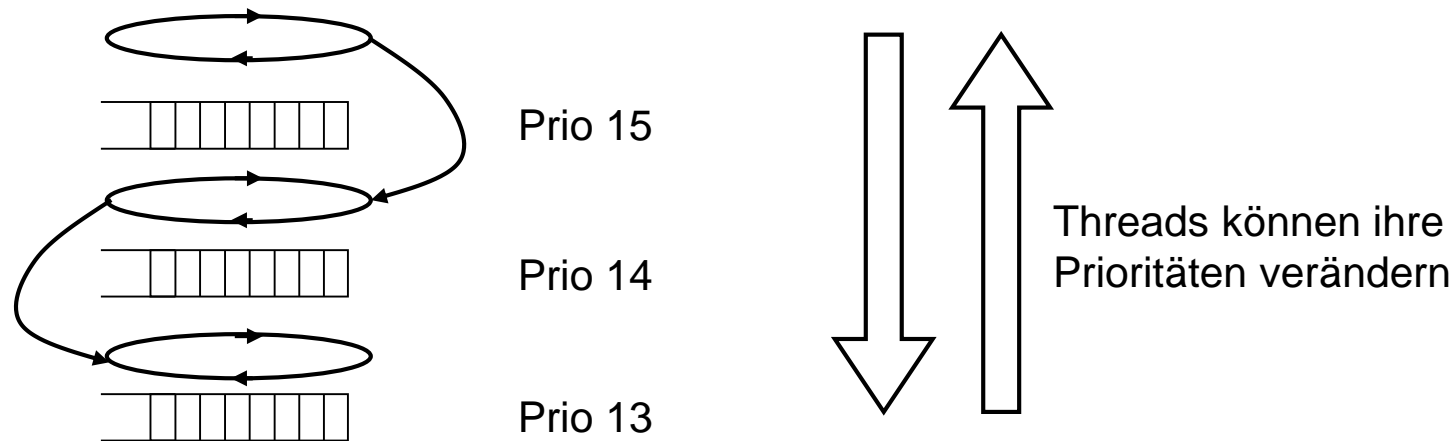
- Echtzeit - Priorität 24
- Hoch - Priorität 13
- Höher als normal - Priorität 9
- Normal - Priorität 8
- Niedriger als normal - Priorität 7
- Niedrig - Priorität 4



- Über WIN32API kann Prozess-Startpriorität (Basispriorität) und Threadpriorität (+-2) festgelegt werden (aber nur im Bereich 1..15)
- User-Priorität (Bereich 1..15) ist dynamisch, d.h. verändert sich zur Laufzeit

Scheduling in Windows XP - II

- Schedulingverfahren: Dynamische Prioritäten mit Round-Robin



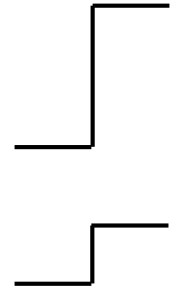
- Zeitscheibenlängen (typisch)

- Workstation: 20-30 ms
- Server: 120-180 ms

Dynamik im Windows XP Scheduling

■ Prioritätserhöhung z.B.

- nach Abschluss einer E/A-Operation
(Beispiel: Platte: +1, Netzwerk: +2, Tastatur: +6)
- nach Beendigung einer Warteoperation auf andere Prozesse
(+1, +2)



■ Prioritätserniedrigung

- wenn ein Prozess seine Zeitscheibe komplett aufgebraucht hat
- jedesmal: -1, aber höchstens bis zur Basispriorität des Threads



■ Modifikation der Zeitscheibenlänge

- Erhöhung der Zeitscheibenlänge, wenn Thread zu einem Fenster gehört, das in den Vordergrund kommt (z.B. x 3)

Lösung für Prioritätsinvertierung in Windows XP

- (Wdh:) Prioritätsinvertierung = Verhungern eines hochprioren Prozesses, weil niedrigpriorer Prozess, von dem dieser abhängig ist, nicht zur Ausführung kommt.
- Lösung:
Windows XP überprüft 1x pro Sekunde, ob ein Prozess für mehr als 4 Sekunden in der Ready-Queue war.

→ Wenn ja:

Prozess bekommt zwei Zeitscheiben lang die maximale Priorität (15). Danach wird er wieder auf seine Basispriorität zurückgesetzt. „Priority Boost“

Seit Windows Vista

- Round-Robin verbessert durch exaktere CPU-Messung
 - Verwendung des Zykluszählers moderner Prozessoren
 - verbrauchte CPU-Zeit pro Thread wird besser abgeschätzt (z.B. keine "gestohlene" Zeit durch Interrupts)
 - gerechtere Zuteilung
- Multimedia-Scheduler
 - Multimedia-Anwendungen registrieren sich beim Betriebssystem unter einer bestimmten Multimedia-Klasse (Audio, Capture, ..)
 - Betriebssystem erhöht für diese Threads die Priorität
 - aber: es ist gewährleistet, dass 20% der CPU-Zeit für andere Threads übrig bleibt (z.B. 8 ms Windows Media Player, 2 ms Virens Scanner)
- Konzept von "E/A-Prioritäten" (zusätzlich zu CPU-Priorität)
 - 6 Prioritätsstufen
 - z.B. Multimedia-Player mit hoher Prio; Virens Scanner mit niedriger

Scheduling in Linux

- Linux verwendet wie Windows ein prioritätsbasiertes, verdrängendes Scheduling-Verfahren
- Scheduling ist ebenfalls thread-basisert,
- Linux verwaltet 3 (Windows: 2) Zuteilungsklassen („policies“) für Prozesse/Threads:

Echtzeit FIFO	<ul style="list-style-type: none">• Eine Thread-Bereit-Queue pro Priorität• Pro Queue: FCFS (First Come First Serve) -Verfahren• Prioritätsbasierte Verdrängung, aber keine Zeitscheiben• Threads können CPU freiwillig abgeben („yield“)
Echtzeit Round Robin	<ul style="list-style-type: none">• Zusätzlich noch Zeitquantum, d.h. Thread wird vom System (nicht freiwillig) der Prozessor entzogen
Nicht-Echtzeit	<ul style="list-style-type: none">• Round-Robin-ähnliches Verfahren (verdrängend)• Dynamische Prioritätenanpassung (ähnlich Windows XP)• Einflussnahme des Users beschränkt möglich

Eigenschaften des Linux Scheduling

- Als „Echtzeit-Threads“ deklarierte Threads werden vor allen anderen bevorzugt
- CPU-lastige Threads werden in einer Art Zeitscheibenverfahren abgearbeitet (Quantum pro Quantum). Allerdings ist die Länge einer Zeitscheibe nicht fest.
- Konkurrieren mehrere CPU-lastige Prozesse, erhält derjenige mehr Rechenzeit, der die höhere Basispriorität hat. (Rechenzeit \sim proportional zur Priorität)
- E/A-lastige Threads, die deblockiert werden, werden bevorzugt (erhalten automatisch ein höheres Quantum)