

Betriebssysteme und nebenläufige Programmierung

SoSe 2026

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 20. März 2026

Betriebssysteme und nebenläufige Programmierung

SoSe 2026


7 Scheduling

Inhalt:

- ➔ Einführung
- ➔ Kriterien für das Scheduling
- ➔ Scheduling-Verfahren: allgemeine Aspekte
- ➔ Scheduling-Algorithmen

- ➔ Tanenbaum 2.5
- ➔ Stallings 9
- ➔ Nehmer/Sturm 5.3

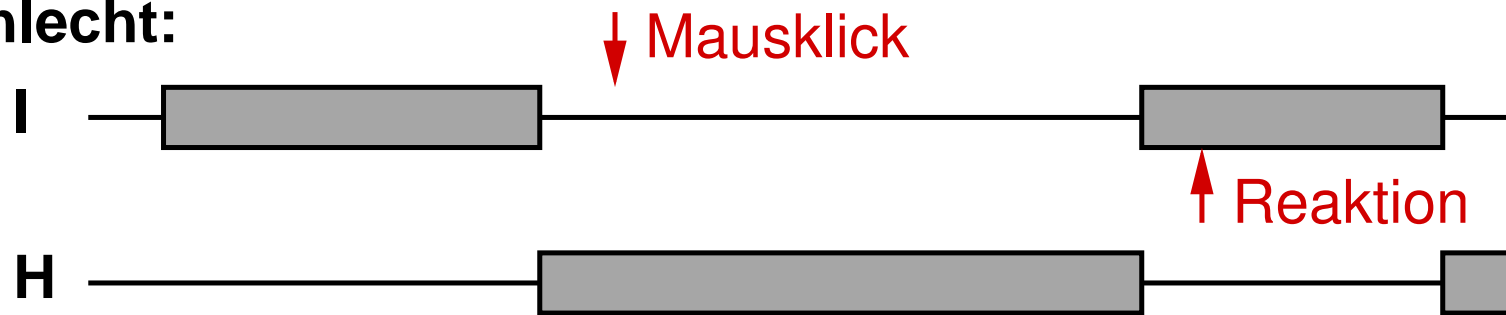
Teilaufgabe des BS: Mehrprogrammbetrieb

- ➔ „Gleichzeitige“ Abarbeitung mehrerer Threads (und damit auch mehrerer Programme) im schnellen Wechsel
- ➔ Bisher behandelt:
 - ➔ Mechanismus des Threadwechsels
- ➔ Jetzt: **Thread-Scheduling**
 - ➔ Auswahlstrategie: welcher Thread darf als nächstes wie lange (und ggf. auf welcher CPU) rechnen?
 - ➔ Verwaltung des Betriebsmittels Prozessor
- ➔ (Daneben: weitere Scheduling-Aufgaben,  **8.3.3, 9.3**)

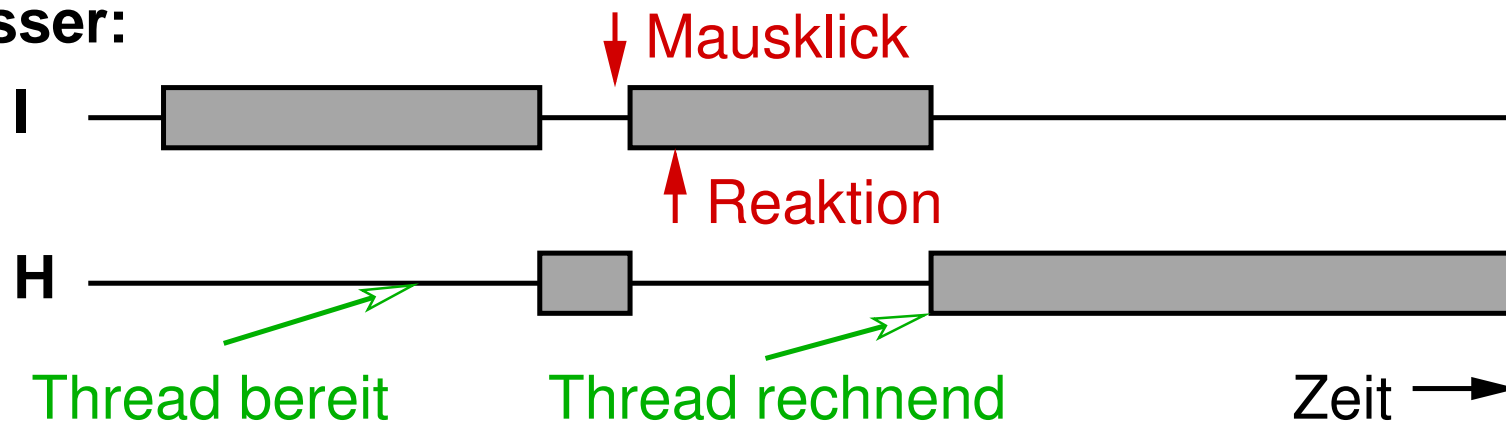
Scheduling beeinflusst Nutzbarkeit des Systems

- ➔ Beispiel: interaktiver Thread (I) und Hintergrundthread (H)
- ➔ Darstellung als **Gantt-Diagramm**

Schlecht:



Besser:

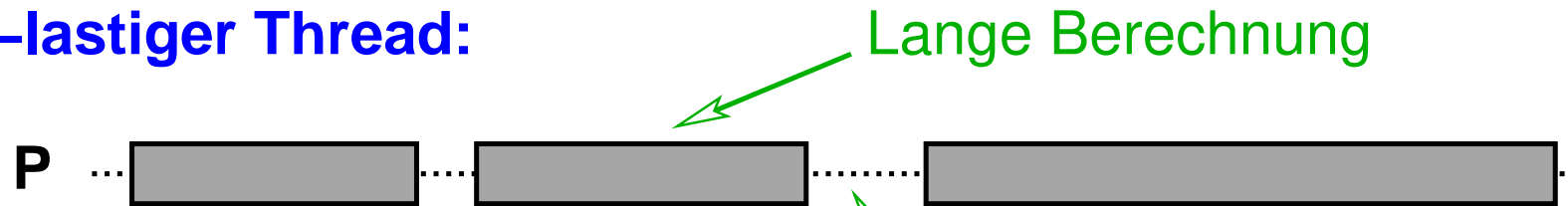


Vorbemerkung: Betriebsarten eines Systems

- ➔ Stapelverarbeitungs-Betrieb
 - ➔ System arbeitet nicht-interaktive Aufträge ab
 - ➔ häufig bei Großrechnern
- ➔ Interaktiver Betrieb
 - ➔ typisch für Arbeitsplatzrechner, Server
- ➔ Echtzeitbetrieb
 - ➔ Steueraufgaben, Multimedia-Anwendungen

Vorbemerkung: Threadcharakteristiken

CPU-lastiger Thread:



Lange Berechnung

E/A-lastiger Thread:



Kurze Berechnung

Warten auf E/A

Zeit →

Kriterien für das Scheduling (Benutzersicht)

- ➔ Minimierung der Durchlaufzeit (Stapelbetrieb)
 - ➔ Zeit zwischen Eingang und Abschluß eines Jobs (inkl. aller Wartezeiten)
- ➔ Minimierung der Antwortzeit (Interaktiver Betrieb)
 - ➔ Zeit zwischen Eingabe einer Anfrage und Beginn der Ausgabe
- ➔ Einhaltung von Terminen (Realzeitbetrieb)
 - ➔ Ausgabe muß nach bestimmter Zeit erfolgt sein
- ➔ Vorhersehbarkeit
 - ➔ Durchlaufzeit / Antwortzeit i.W. unabhängig von Auslastung des Systems

Kriterien für das Scheduling (Systemsicht)

- ➔ Maximierung des Durchsatzes (Stapelbetrieb)
 - ➔ Anzahl der fertiggestellten Jobs pro Zeiteinheit
- ➔ Optimierung der Prozessorauslastung (Stapelbetrieb)
 - ➔ prozentualer Anteil der Zeit, in der Prozessor beschäftigt ist
- ➔ Balance
 - ➔ gleichmäßige Auslastung aller Ressourcen
- ➔ Fairness
 - ➔ vergleichbare Threads sollten gleich behandelt werden
- ➔ Durchsetzung von Prioritäten
 - ➔ Threads mit höherer Priorität bevorzugt behandeln



Scheduling-Kriterien nicht gleichzeitig erfüllbar

- ➔ z.B. Prozessorauslastung \leftrightarrow kurze Antwortzeit
- ➔ Scheduling-Ziele und -Verfahren von Betriebsart abhängig
 - ➔ Stapelverarbeitung: hoher Durchsatz, gute Auslastung
 - ➔ bevorzuge Aufträge, die freie Ressourcen nutzen
 - ➔ Interaktiver Betrieb: kurze Antwortzeiten
 - ➔ bevorzuge Threads, die auf E/A (Benutzereingabe) gewartet haben und nun rechenbereit sind
 - ➔ Echtzeitbetrieb: Einhaltung von Zeitvorgaben
 - ➔ bevorzuge Threads, deren Ausführungsfristen ablaufen

Ebenen des Scheduling

- ➔ Langfristiges Scheduling (Eingangs-Scheduler)
 - ➔ bei Systemen mit Stapelverarbeitung von Jobs:
 - ➔ welcher Job wird als nächstes zugelassen?
D.h. wann werden Prozesse für den Job erzeugt?
 - ➔ Ziel z.B. Mischung aus CPU- und E/A-lastigen Prozessen
- ➔ Mittelfristiges Scheduling (Speicher-Scheduler)
 - ➔ Auslagern und Suspendieren von Prozessen (z.B. bei Speicherengpässen)
 - ➔ legt Multiprogramming-Grad fest
- ➔ Kurzfristiges Scheduling (CPU-Scheduler)
 - ➔ Zuteilung der CPU(s) an bereite Threads



Nicht-präemptives und präemptives Scheduling

➔ Nicht-präemptives Scheduling

- ➔ Thread darf so lange rechnen, bis er freiwillig die CPU aufgibt oder blockiert
- ➔ sinnvoll bei Stapelverarbeitung und teilw. Echtzeitsystemen

➔ Präemptives Scheduling

- ➔ Scheduler kann einem Thread die CPU zwangsweise entziehen
 - ➔ wenn ein anderer Thread (mit höherer Priorität) rechenbereit geworden ist
 - ➔ nach Ablauf einer bestimmten Zeit
- ➔ unterstützt interaktive Systeme und Echtzeitsysteme

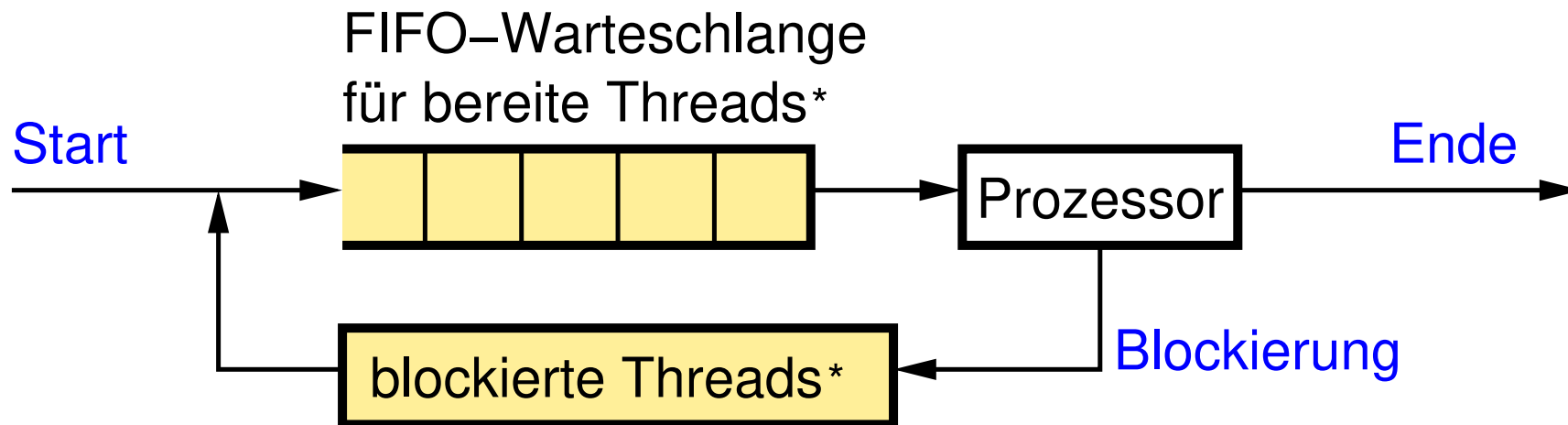
Scheduling-Zeitpunkte (CPU-Scheduler)

- ➔ Bei Beendigung eines Threads
 - ➔ Bei freiwilliger CPU-Aufgabe eines Threads
 - ➔ Bei Blockierung eines Threads
 - ➔ Grund der Blockierung kann/sollte berücksichtigt werden
 - ➔ z.B. bei P()-Operation: weise CPU dem Thread zu, der Semaphor belegt \Rightarrow kürzere Blockierung
 - ➔ Bei Erzeugung eines Threads
 - ➔ Bei E/A-Interrupt
 - ➔ evtl. werden blockierte Threads rechenbereit
 - ➔ Regelmäßig bei Timer-Interrupt
- } nur bei prä-emptivem Scheduling

- ➔ Legen fest, welcher Thread auf einer CPU als nächstes ausgeführt wird und wie lange er rechnen darf
- ➔ Bei Multiprozessor-Systemen prinzipiell:
 - ➔ jede CPU führt Algorithmus unabhängig von den anderen aus
 - ➔ Zugriffe auf Bereit-Warteschlange unter wechselseitigem Ausschluß
- ➔ Detailaspekt: **Cache-Affinity** / **Affinity Scheduling**
 - ➔ ein einmal auf einer CPU ausgeführter Thread sollte nach Möglichkeit auf dieser CPU bleiben
 - ➔ Daten des Threads sind im Cache dieser CPU!
 - ➔ jeder Thread erhält eine (oder mehrere) bevorzugte CPU(s)
 - ➔ einfache Realisierung: getrennte Warteschlangen für jede CPU
 - ➔ falls leer: Warteschlangen anderer CPUs inspizieren

7.4.1 *First Come First Served (FCFS)*

➔ Nicht-präemptives Verfahren



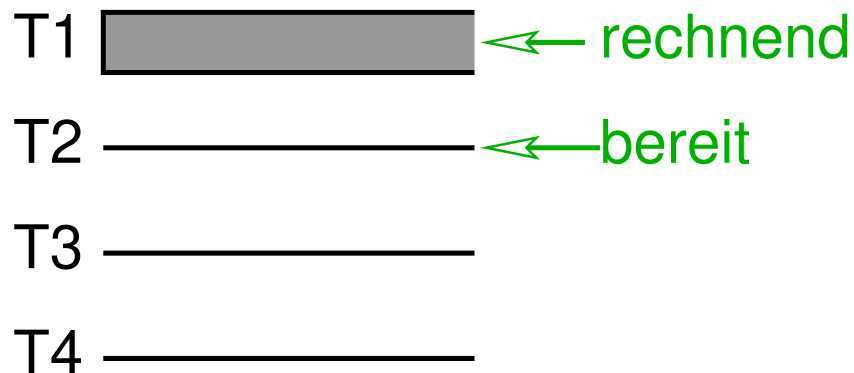
➔ Der am längsten wartende bereite Thread* darf als nächstes rechnen

➔ nach Aufhebung einer Blockierung reißt sich ein Thread* wieder hinten in die Warteschlange ein

* bzw. Job(s) bei Stapelbetrieb

Diskussion

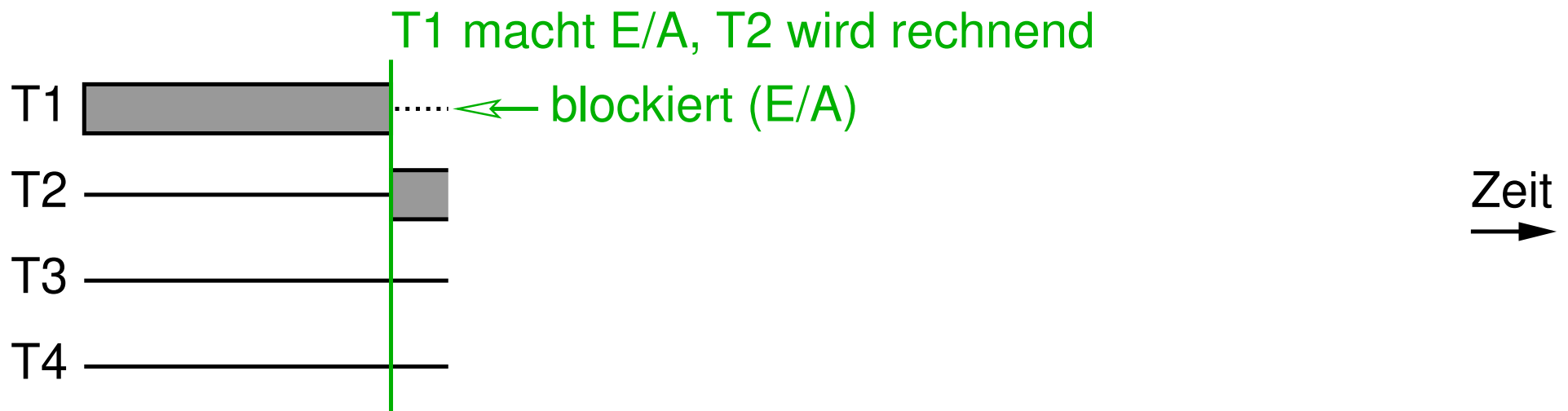
- ➔ Sehr einfacher Algorithmus
- ➔ Stellt gute CPU-Auslastung sicher
- ➔ Vorwiegend für Stapelverarbeitung
- ➔ Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- ➔ Balance evtl. schlecht durch Convoy-Effekt



Zeit
→

Diskussion

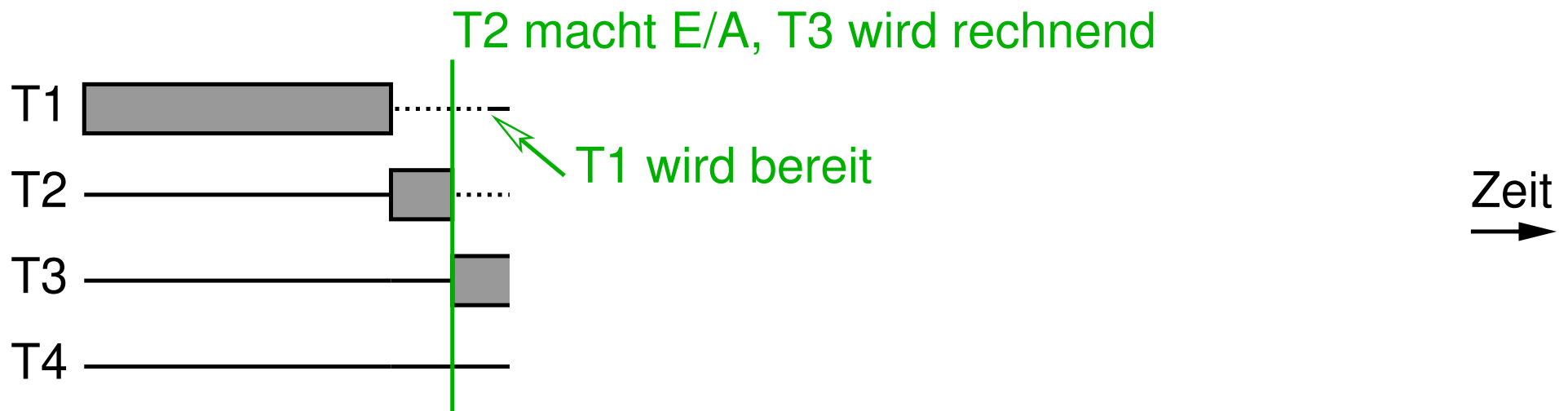
- ➔ Sehr einfacher Algorithmus
- ➔ Stellt gute CPU-Auslastung sicher
- ➔ Vorwiegend für Stapelverarbeitung
- ➔ Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- ➔ Balance evtl. schlecht durch Convoy-Effekt





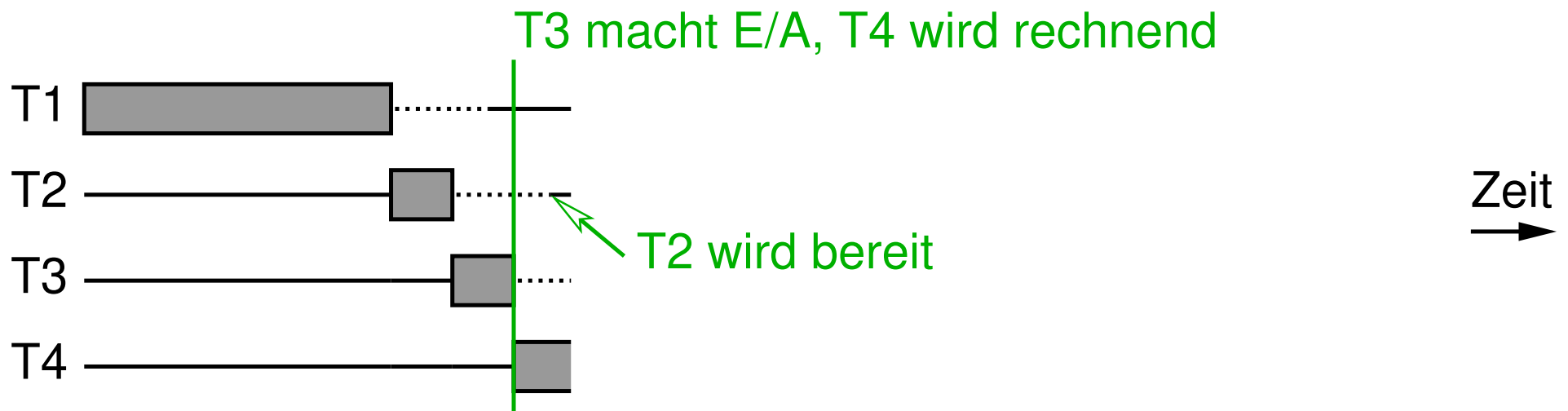
Diskussion

- ➔ Sehr einfacher Algorithmus
- ➔ Stellt gute CPU-Auslastung sicher
- ➔ Vorwiegend für Stapelverarbeitung
- ➔ Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- ➔ Balance evtl. schlecht durch Convoy-Effekt



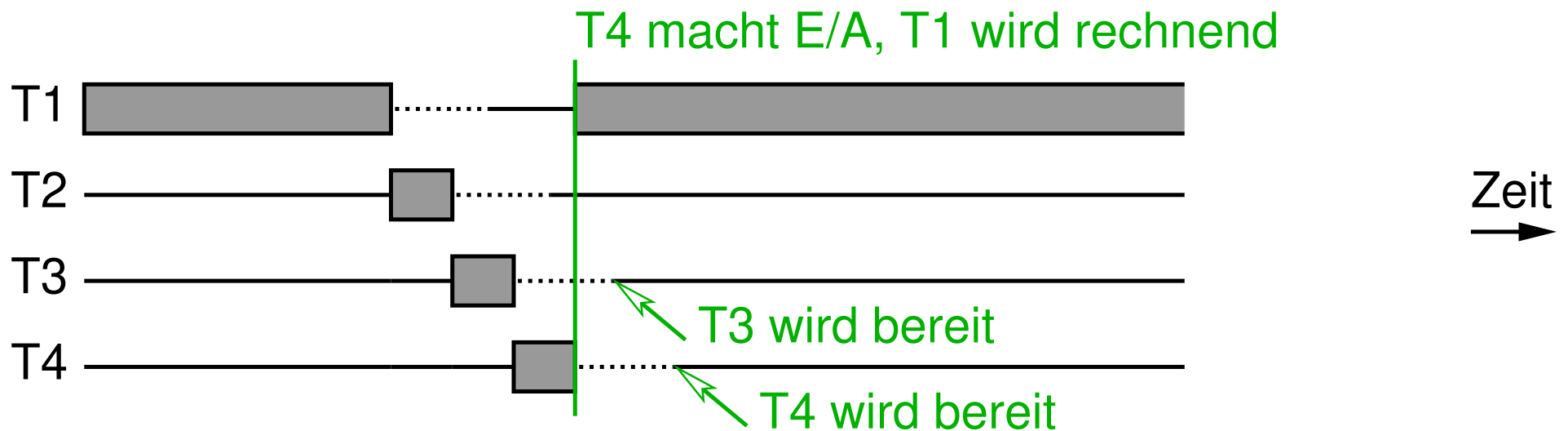
Diskussion

- ➔ Sehr einfacher Algorithmus
- ➔ Stellt gute CPU-Auslastung sicher
- ➔ Vorwiegend für Stapelverarbeitung
- ➔ Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- ➔ Balance evtl. schlecht durch Convoy-Effekt



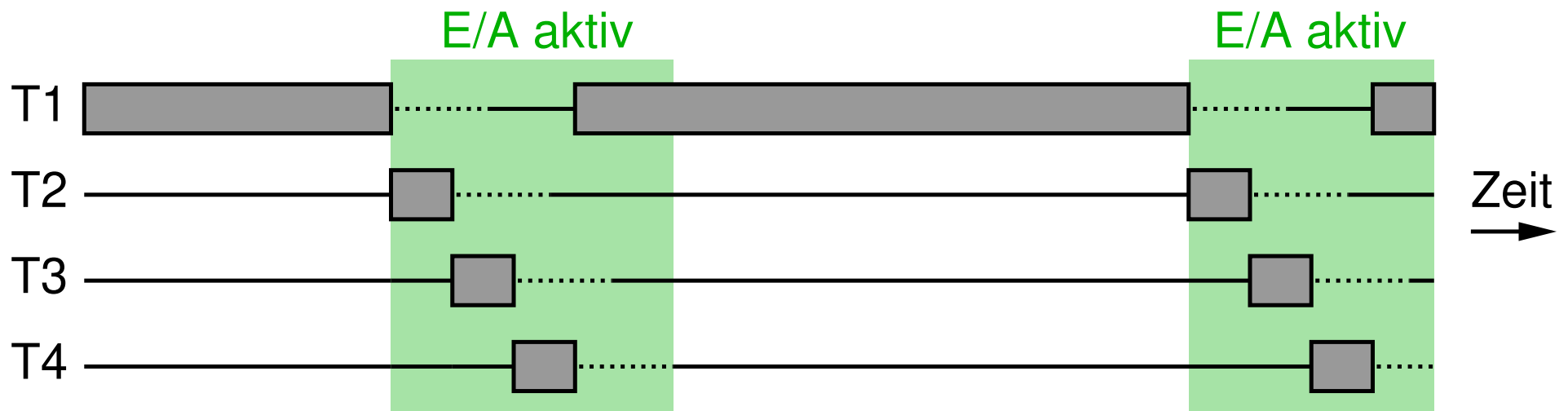
Diskussion

- ➔ Sehr einfacher Algorithmus
- ➔ Stellt gute CPU-Auslastung sicher
- ➔ Vorwiegend für Stapelverarbeitung
- ➔ Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- ➔ Balance evtl. schlecht durch Convoy-Effekt



Diskussion

- ➔ Sehr einfacher Algorithmus
- ➔ Stellt gute CPU-Auslastung sicher
- ➔ Vorwiegend für Stapelverarbeitung
- ➔ Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- ➔ Balance evtl. schlecht durch Convoy-Effekt



7.4.2 *Shortest Job First (SJF)*

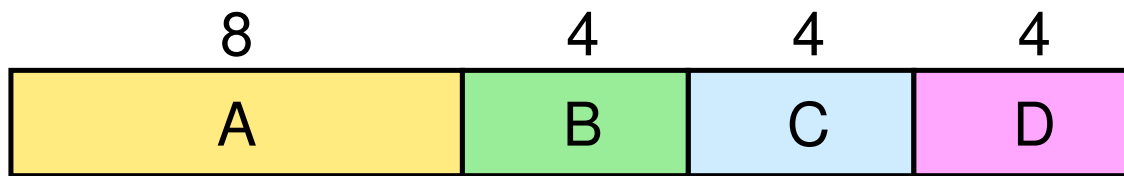
- ➔ Ziel: Optimierung der Durchlaufzeit
- ➔ Strategie: kürzester Job wird zuerst gerechnet
 - ➔ Dauer muß vorab bekannt sein (bei Stapelbetrieb i.a. erfüllt)
- ➔ SJF ist beweisbar optimal, falls alle Jobs gleichzeitig vorliegen
 - ➔ in der Regel: Jobs treffen nacheinander ein, Scheduler kann nur die Jobs berücksichtigen, die bereits bekannt sind
- ➔ Einfachster Fall: nicht-präemptiv, keine Blockierungen durch E/A
- ➔ In präemptiver Variante und bei Blockierungen durch E/A:
 - ➔ betrachte **Rest**laufzeiten
- ➔ Variante für interaktive Systeme:
 - ➔ betrachte CPU-Burst als Job, mit Schätzung der Dauer

7.4.2 Shortest Job First (SJF) ...



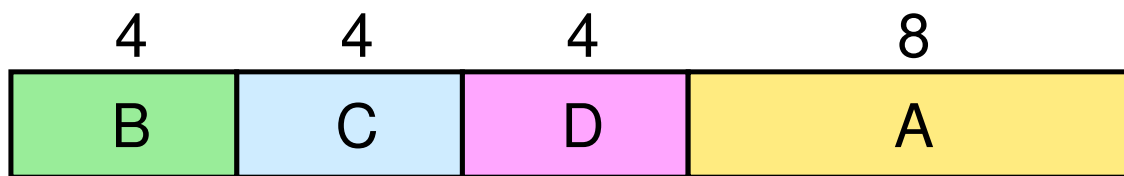
Beispiel (alle Jobs liegen gleichzeitig vor)

Vier Jobs in FCFS-Reihenfolge



Mittlere Durchlaufzeit:
 $(8+12+16+20)/4 = 14$

Dieselben Jobs in SJF-Reihenfolge



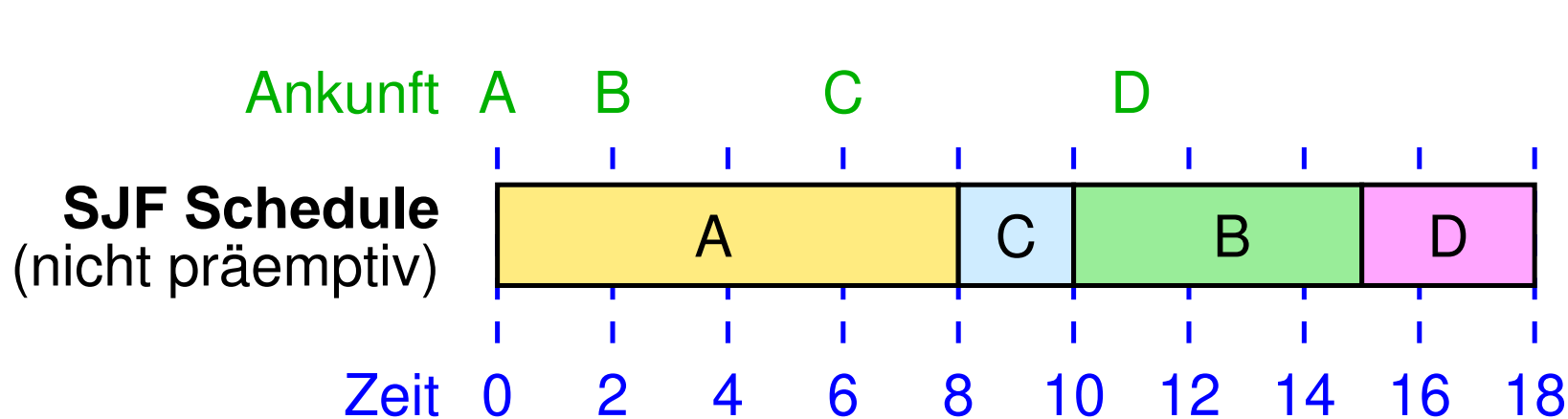
Mittlere Durchlaufzeit:
 $(4+8+12+20)/4 = 11$

7.4.2 Shortest Job First (SJF) ...



Beispiel (die Jobs treffen nacheinander ein)

Job	Ankunftszeit	Bedienzeit
A	0	8
B	2	5
C	6	2
D	11	3



Mittlere
Durchlaufzeit:

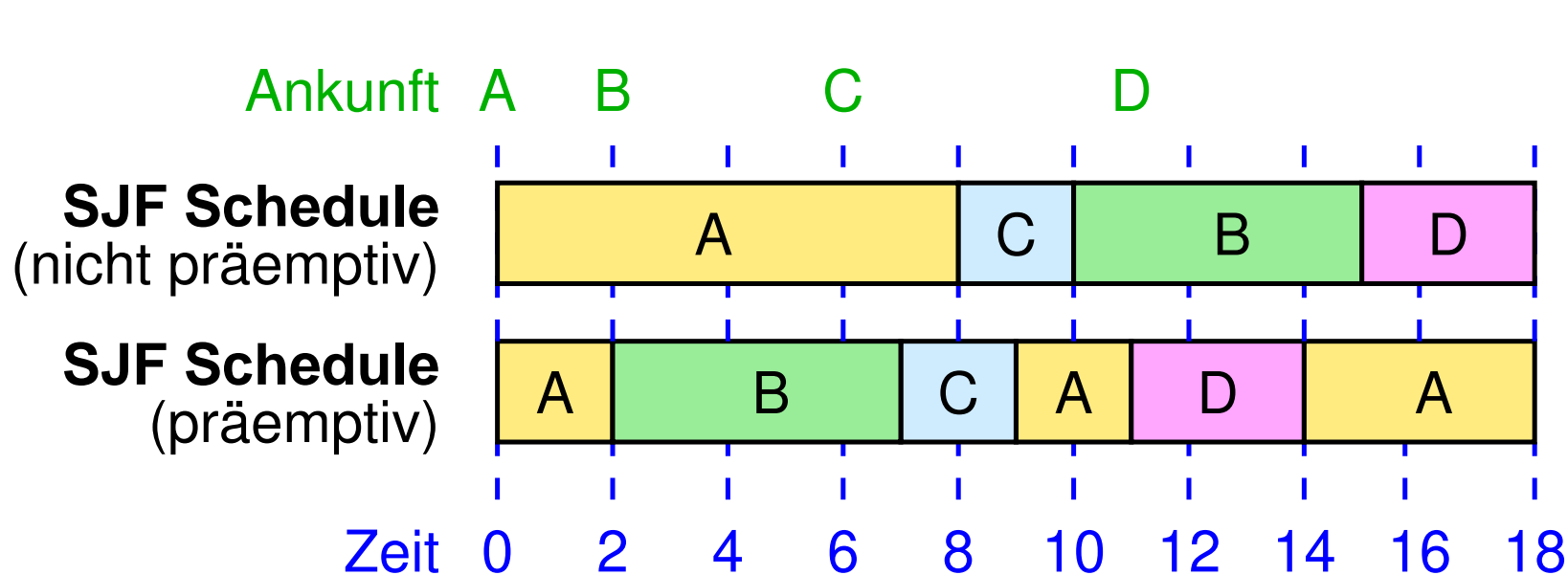
$$(8+13+4+7)/4 = 8$$

7.4.2 Shortest Job First (SJF) ...



Beispiel (die Jobs treffen nacheinander ein)

Job	Ankunftszeit	Bedienzeit
A	0	8
B	2	5
C	6	2
D	11	3



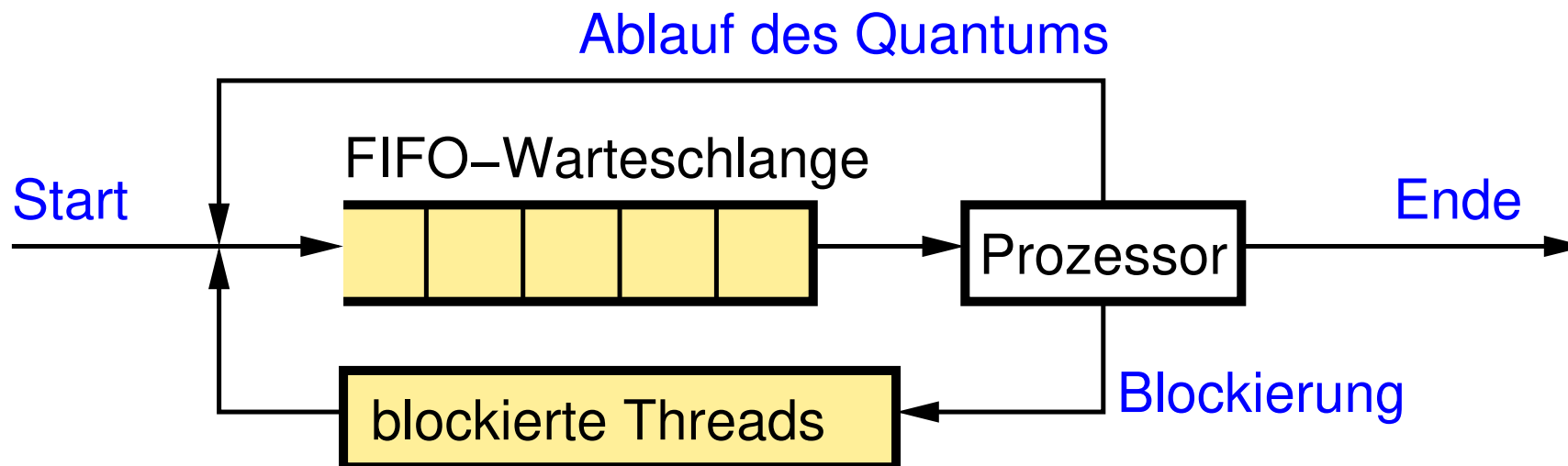
Mittlere
Durchlaufzeit:

$$(8+13+4+7)/4 = 8$$

$$(18+5+3+3)/4 = 7,25$$

7.4.3 Round Robin (RR), Zeitscheibenverfahren

- ➔ Präemptive Variante von FCFS
- ➔ Jeder Thread darf höchstens eine bestimmte Zeit (**Quantum**, **Zeitscheibe**, **Time Slice**) rechnen



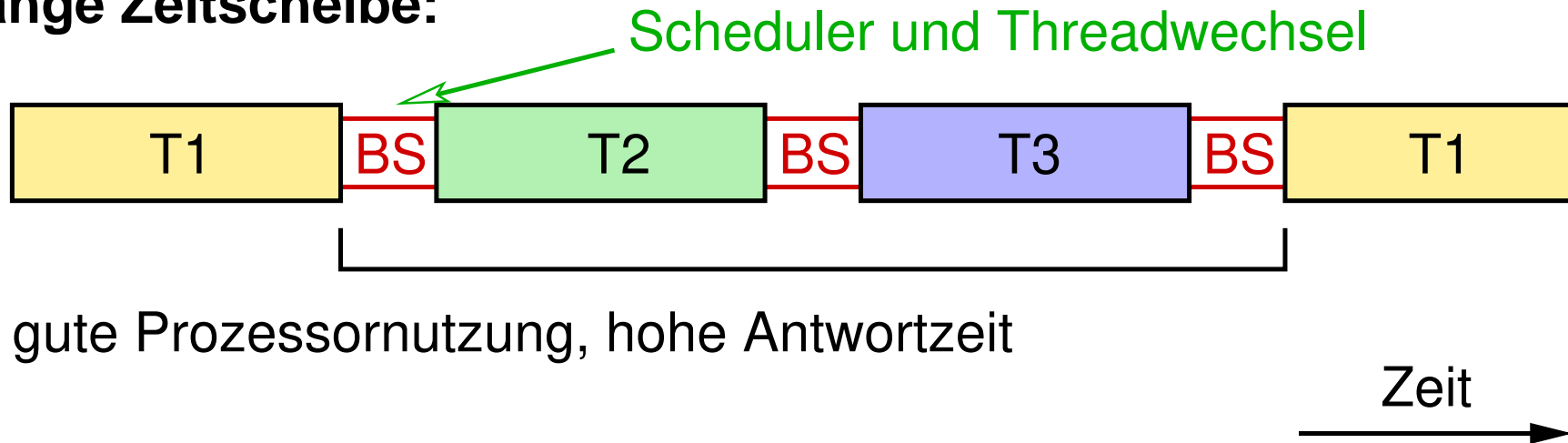


Diskussion

- ➔ Einfach zu realisieren
 - ➔ FIFO-Warteschlange aller rechenbereiten Threads und Timer-Interrupt
- ➔ Frage: Länge des Quantums
 - ➔ kurzes Quantum:
 - ➔ kurze Antwortzeiten
 - ➔ schlechte CPU-Nutzung durch häufige Threadwechsel
 - ➔ langes Quantum: umgekehrt
 - ➔ Praxis: 10 - 100 *ms*
- ➔ RR ist nicht fair:
 - ➔ E/A-lastige Threads werden benachteiligt

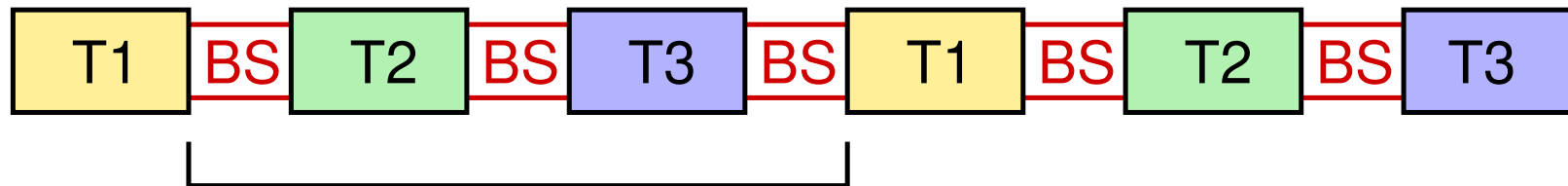
Zeitscheibenlänge

Lange Zeitscheibe:



gute Prozessornutzung, hohe Antwortzeit

Kurze Zeitscheibe:

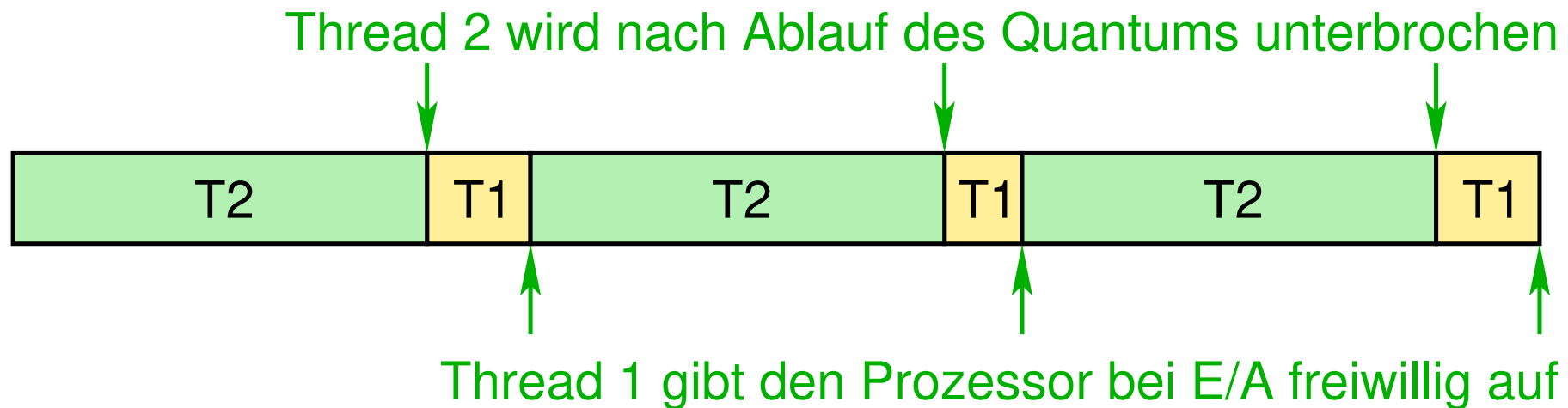


kurze Antwortzeit, schlechte Prozessornutzung

Fairness

T1 E/A-lastiger Thread

T2 CPU-lastiger Thread



➔ Thread 1 ist gegenüber Thread 2 benachteiligt

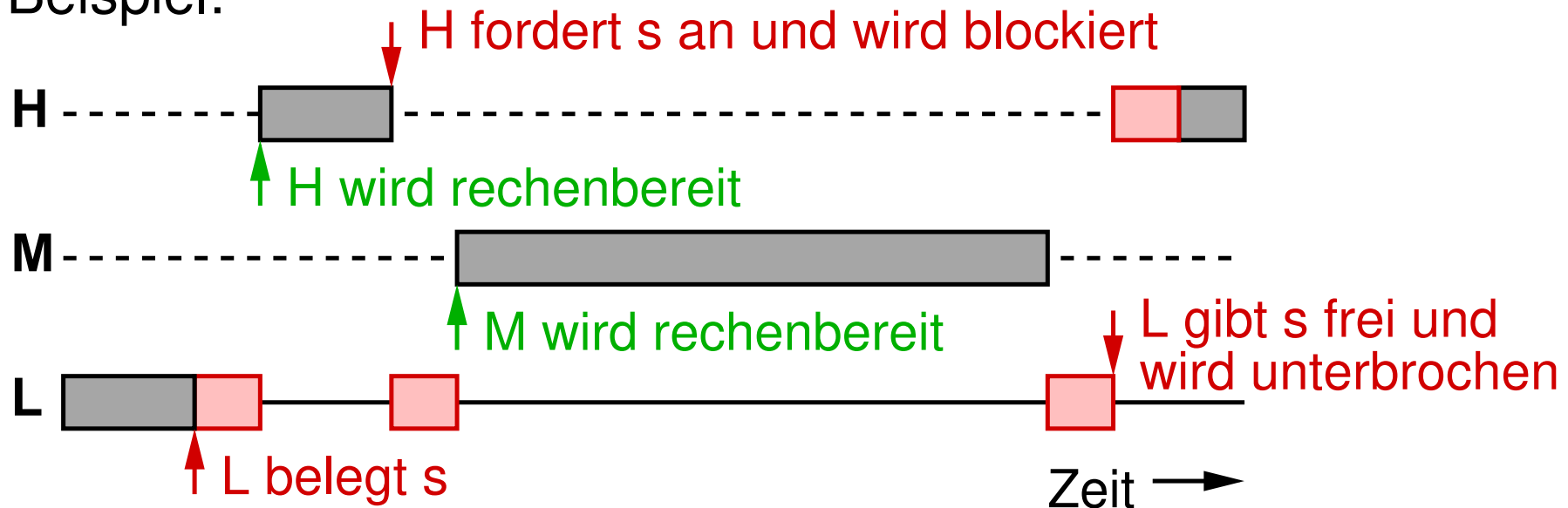
7.4.4 Prioritätenbasiertes Scheduling

- ➔ Jeder Thread erhält eine (unterschiedliche) Priorität
 - ➔ CPU wird an den Thread mit der höchsten Priorität zugeteilt
- ➔ I.d.R. präemptiv
- ➔ Statische Prioritäten
 - ➔ Priorität eines Threads bei Erzeugung festgelegt
 - ➔ häufig bei Realzeit-Systemen
- ➔ Dynamische Prioritäten
 - ➔ Priorität kann laufend geändert werden, abhängig vom Verhalten des Threads (**Feedback Scheduling**)
 - ➔ z.B.: Priorität bestimmt durch Länge des letzten CPU-Bursts des Threads (\sim SJF)

Prioritätsinversion

- ➔ Problem bei wechselseitigem Ausschluß:
 - ➔ hoch priorisierter Thread **H** muss ggf. warten, bis ein Thread **L** mit niedriger Priorität ein Semaphor **s** freigibt
 - ➔ Thread **M** mit mittlerer Priorität kann **L** (und damit auch **H**) beliebig lange aufhalten

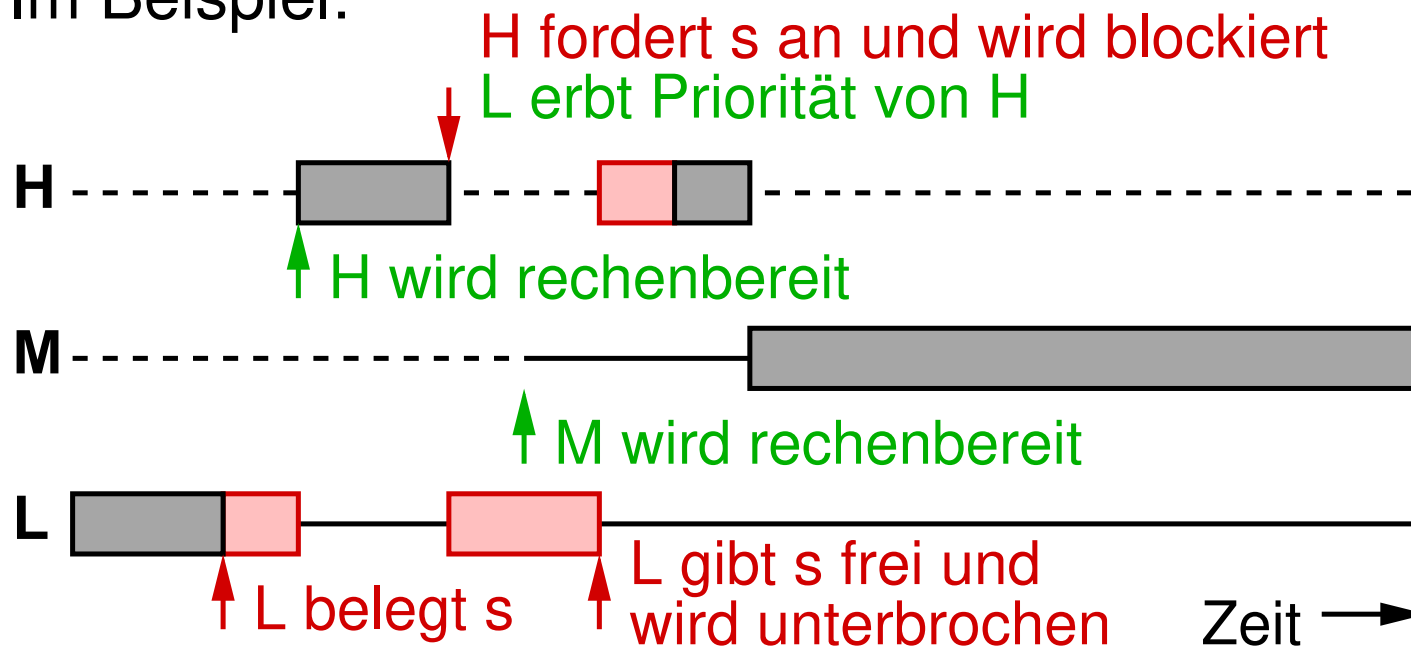
➔ Beispiel:



Prioritätsinversion ...

- ➔ Lösung: Prioritätsvererbung
- ➔ Wenn ein hoch priorisierter Thread durch einen niedriger priorisierten Thread blockiert ist, erbt dieser für die Dauer der Blockierung die Priorität des blockierten Threads

➔ Im Beispiel:



7.4.5 *Multilevel Scheduling*

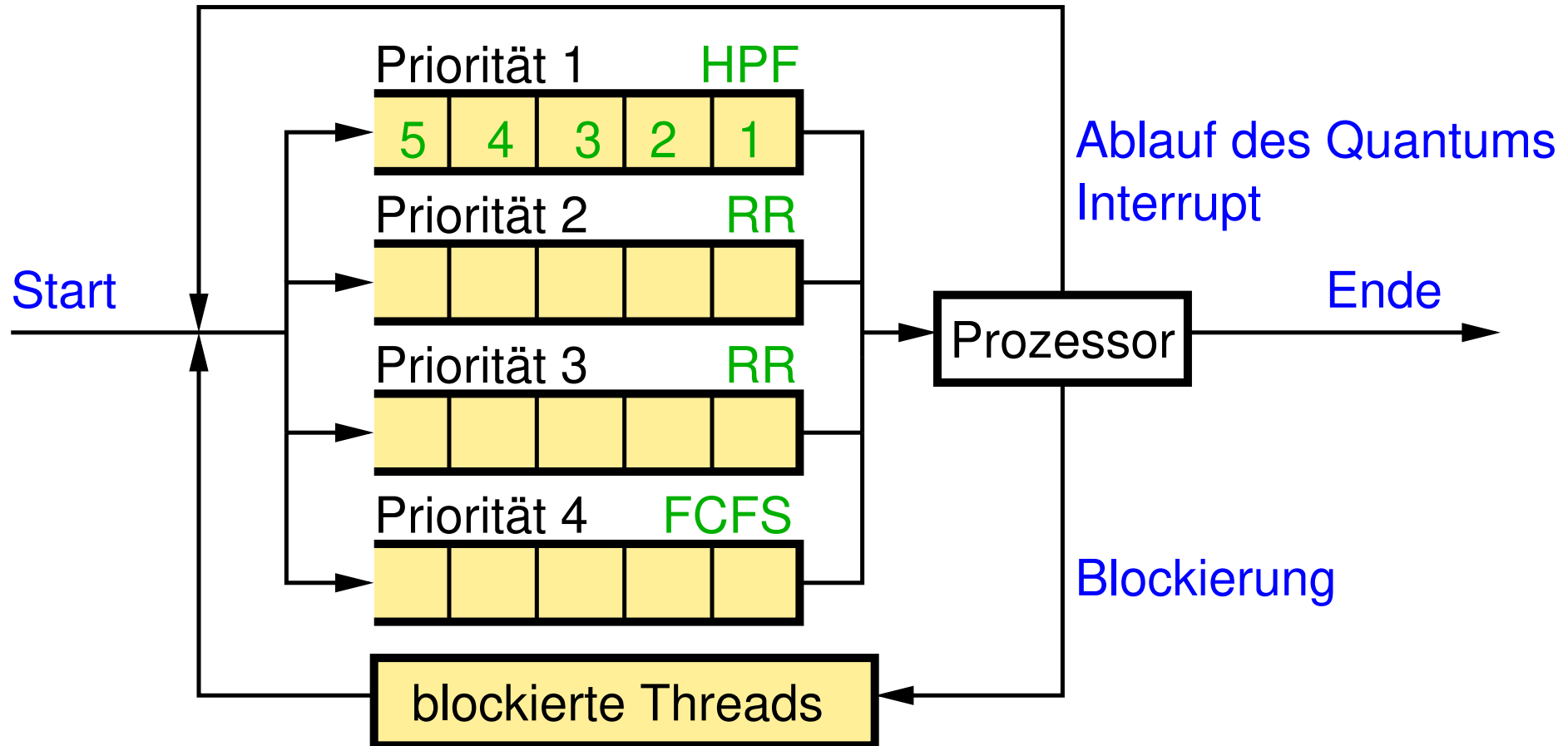
- ➔ Mehrere Warteschlangen für bereite Threads
- ➔ Jede Warteschlange kann eigene Auswahlstrategie besitzen
- ➔ Zusätzlich: Strategie zur Auswahl der **aktuellen** Warteschlange
 - ➔ z.B. Prioritäten oder Round-Robin
- ➔ Statische Verfahren:
 - ➔ Thread wird fest einer Warteschlange zugeordnet
- ➔ Dynamische Verfahren:
 - ➔ Thread kann in Abhängigkeit seines Verhaltens zwischen Warteschlangen wechseln

Beispiel: Statisches *Multilevel Scheduling*

- ➔ Unterstützung verschiedener Betriebsarten in einem System
- ➔ Threads werden in Prioritäts**klassen** eingeteilt
 - ➔ pro Prioritätsklasse eine Warteschlange
 - ➔ unterschiedliche Scheduling-Strategien innerhalb der Warteschlangen

Priorität	Threadklasse	Strategie
1	Echtzeitthreads (Multimedia)	Prioritäten
2	Interaktive Threads	RR
3	E/A-intensive Threads	RR
4	Rechenintensive Stapel-Jobs	FCFS

Beispiel: Statisches Multilevel Scheduling ...

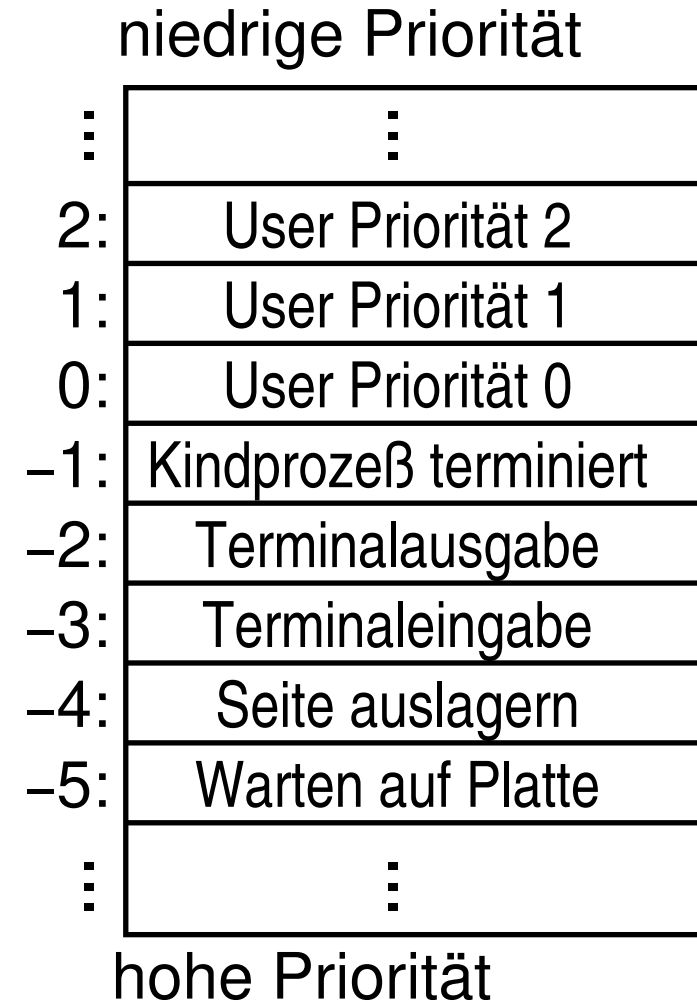


Beispiel: *Multilevel Feedback Scheduling*

- ➔ Dynamische Prioritäten und variable Zeitscheibenlänge
- ➔ Mehrere Warteschlangen mit unterschiedlicher Priorität
 - ➔ innerhalb einer Warteschlange: *Round Robin*
 - ➔ bei niedrigerer Priorität: längeres Quantum
- ➔ Falls Thread Quantum aufbraucht: erniedrigen der Priorität
 - ➔ CPU-lastiger Thread erhält längeres Quantum, wird seltener unterbrochen
- ➔ Sonst: Priorität nicht verändern bzw. wieder erhöhen
 - ➔ E/A-lastiger (bzw. interaktiver) Thread erhält bevorzugt die CPU, aber nur für kurze Zeit

7.5.1 Scheduling in BSD-Unix

- ➔ Priorisierte Warteschlangen mit *Round Robin*
- ➔ Threads, die durch BS-Aufruf im BS-Kern blockiert sind, erhalten hohe (negative) Priorität
 - ➔ nach Ende der Blockierung: Thread erhält bevorzugt die CPU, Priorität wird dann wieder auf alten Wert gesetzt
- ➔ Anpassung der normalen User-Priorität je nach CPU-Nutzung (Aging-Verfahren)





7.5.2 Scheduling in Windows NT / Vista

- ➔ 32 Prioritätsklassen
 - ➔ Priorität 16-31 (höchste)
 - ➔ Echtzeitklasse, statische Priorität
 - ➔ Priorität 1-15
 - ➔ normale Threads, dynamische Priorität
 - ➔ starke Prioritätserhöhung beim Warten auf Benutzereingabe, moderatere Erhöhung beim Warten auf E/A,
 - ➔ danach schrittweise Reduktion zum Ausgangswert
 - ➔ Priorität 0 (niedrigste)
 - ➔ *Idle*-Thread
- ➔ Innerhalb einer Klasse: *Round-Robin*



7.5.3 Scheduling in Linux (O(1) Scheduler)

- ➔ Jeder Thread wird in Linux einer Scheduling-Klasse zugeordnet:
 - ➔ SCHED_FIFO: „Echtzeit“-Threads mit FIFO Scheduling
 - ➔ SCHED_RR: „Echtzeit“-Threads mit Round Robin Scheduling
 - ➔ SCHED_OTHER: normale Threads
- ➔ „Echtzeit“-Threads besitzen lediglich höhere Priorität (0-99)
 - ➔ Priorität normaler Threads: 100-139
- ➔ Für jede Priorität: je zwei Warteschlangen pro CPU
 - ➔ *active*: Threads, die Zeitscheibe noch nicht aufgebraucht haben
 - ➔ *expired*: Threads, deren Quantum abgelaufen ist
- ➔ Zusätzliches Bit-Set zeigt an, welche Warteschlangen nicht leer sind



Vorgehensweise

- ➔ Scheduler wählt Thread mit höchster Priorität in *active* Warteschlange
 - ➔ falls Quantum aufgebraucht wird: Einreihen in *expired* Warteschlange
 - ➔ falls Thread blockiert: nach Blockierung wieder in *active* Warteschlange (mit entsprechend reduziertem Quantum)
- ➔ Falls *active* Warteschlange leer: Tausch der Warteschlangen
- ➔ Prioritäten der normalen Threads werden durch komplexe Heuristiken angepasst
 - ➔ interaktive (E/A-lastige) Threads sollen höhere Priorität erhalten



7.5.4 Linux: *Completely Fair Scheduler* (CFS)

- ➔ Ziel: jeder Thread bekommt einen fairen Anteil der CPU-Leistung
 - ➔ ohne aufwendige und problematische Heuristiken
- ➔ CFS simuliert eine ideale Multitasking-CPU
 - ➔ bei n Threads erhält jeder den Anteil $1/n$ der CPU-Leistung
- ➔ Dazu: die bisher erhaltene CPU-Zeit eines Threads wird ns -genau erfasst (*virtual runtime*)
- ➔ Rechenbereite Threads werden nach *virtual runtime* sortiert in einen balancierten Baum (*Red-Black-Tree*) eingetragen
- ➔ Der Thread mit der kleinsten *virtual runtime* erhält die CPU
- ➔ Neu erzeugte Threads erhalten als *virtual runtime* den Mittelwert aller anderen Threads



Red-Black-Tree

- ➔ Balancierter, binärer Suchbaum
 - ➔ Knoten sind entweder rot oder schwarz eingefärbt
 - ➔ jeder interne Knoten hat zwei Kinder
 - ➔ interne Knoten tragen Informationen
-
- ➔ Forderungen:
 - ➔ Wurzel und Blätter sind schwarz
 - ➔ rote Knoten haben zwei schwarze Kinder
 - ➔ jeder Pfad von einem Knoten zu einem Blatt hat dieselbe Zahl schwarzer Knoten
 - ➔ Damit Baumhöhe bei n inneren Knoten: $\mathcal{O}(\log n)$
 - ➔ Einfügen / Suchen / Löschen in $\mathcal{O}(\log n)$



Details

- ➔ Länge der Zeitscheibe:
 - ➔ Systemparameter *latency*: Zeit, bis jeder Thread einmal „drankommt“
 - ➔ Quantum = $latency / N$, wobei N = Zahl der rechenbereiten Threads
 - ➔ ggf. Sonderbehandlung, falls N zu groß
- ➔ Behandlung der Prioritäten (*nice*-Werte):
 - ➔ Quantum wird entsprechend der Priorität gewichtet
- ➔ Gruppen-Scheduling
 - ➔ erlaubt Definition von Gruppen von Threads
 - ➔ jede Gruppe erhält dann denselben Anteil der CPU-Leistung



Multi-Core Scheduling

- ➔ Jeder Core hat eigene Thread-Warteschlangen (bzw. RB-Tree)
 - ➔ damit: Skalierbarkeit, keine Synchronisation notwendig
- ➔ Warteschlangen müssen zwischen Cores balanciert sein
 - ➔ z.B.: nicht alle hochprioren Threads bei einem Core
- ➔ Lastbalancierung ist aber teuer (Synchronisation, Caches)
 - ➔ seltene Durchführung; gute Verfahren notwendig
- ➔ Last berücksichtigt Prioritäten und durchschnittliche CPU-Nutzung
- ➔ Lastverteilung erfolgt hierarchisch
 - ➔ wegen hierarchischer Cache-/Speicherstruktur

➔ Scheduling

- ➔ Entscheidung welcher Thread wann wie lange (und ggf. auch welcher CPU) rechnen darf
- ➔ Unterschiedliche Anforderungen, je nach Sichtweise und Betriebsmodus
- ➔ Nicht-präemptives und präemptives Scheduling
 - ➔ präemptiv: BS kann einem Thread die CPU zwangsweise entziehen

➔ Scheduling-Algorithmen

- ➔ FCFS: FIFO-Warteschlange rechenbereiter Threads, nicht-präemptiv
- ➔ SJF: *Shortest Job First*
 - ➔ optimiert Durchlaufzeit von Jobs

- ➔ Scheduling-Algorithmen ...
 - ➔ *Round Robin* (RR): präemptive Version von FCFS
 - ➔ Threads dürfen nur bestimmte Zeit rechnen
 - ➔ Prioritätenbasiertes Scheduling:
 - ➔ nur der Thread mit höchster Priorität bekommt CPU (bzw. die n höchstpriorären Threads bei n CPUs)
 - ➔ *Multilevel* Scheduling
 - ➔ mehrere Warteschlangen mit unterschiedlicher Auswahlstrategie
 - ➔ statisches *Multilevel* Scheduling
 - ➔ feste Zuordnung Thread → Warteschlange
 - ➔ *Multilevel Feedback* Scheduling
 - ➔ dynamische Zuordnung Thread → Warteschlange