



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

7.1 Einführung



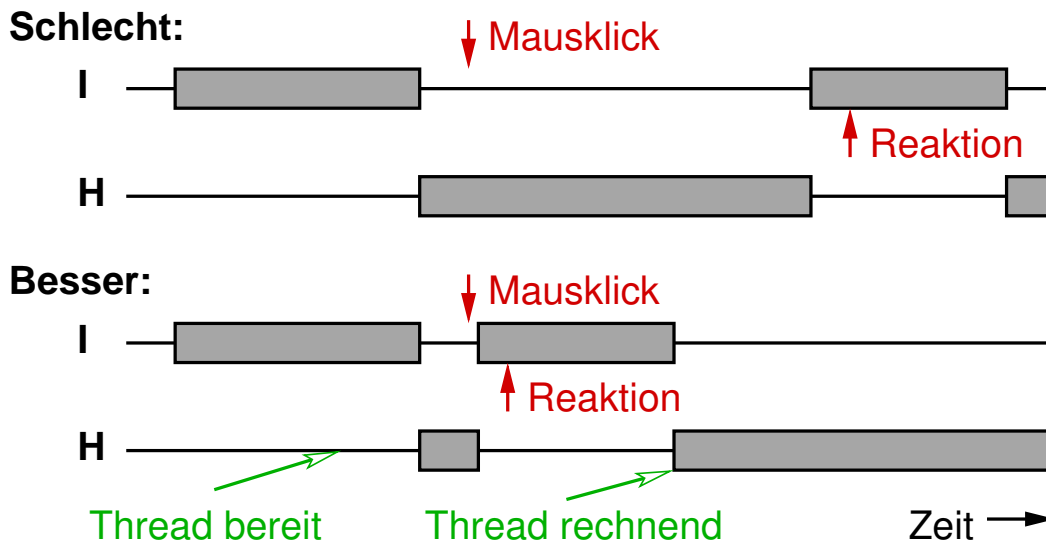
Teilaufgabe des BS: Mehrprogrammbetrieb

- ➔ „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**



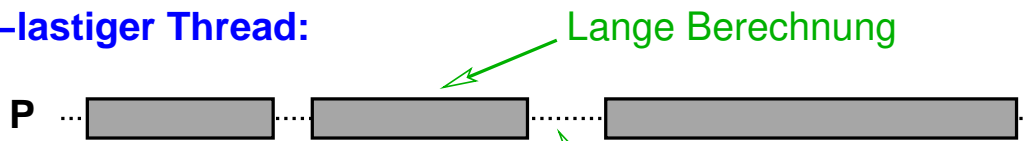
7.2 Kriterien für das Scheduling

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

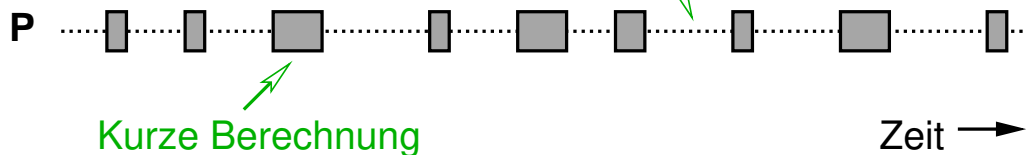
Vorbemerkung: Threadcharakteristiken

CPU-lastiger Thread:



Warten auf E/A

E/A-lastiger Thread:



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 ↔ 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

7.3 Scheduling-Verfahren: allgemeine Aspekte ...



★★

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 ⇒ kürzere Blockierung
 - ➔ Bei Erzeugung eines Threads
 - ➔ Bei E/A-Interrupt
 - evtl. werden blockierte Threads rechenbereit
 - ➔ Regelmäßig bei Timer-Interrupt
- } nur bei prä-emptivem Scheduling

7.4 Scheduling-Algorithmen



- ➔ 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
- ➔ Detailspekt: **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

Anmerkungen zu Folie 340:

Etwas mehr zum Thema Multiprozessor-Scheduling findet sich im Tanenbaum-Buch, Kap. 8.1.4.

340-1

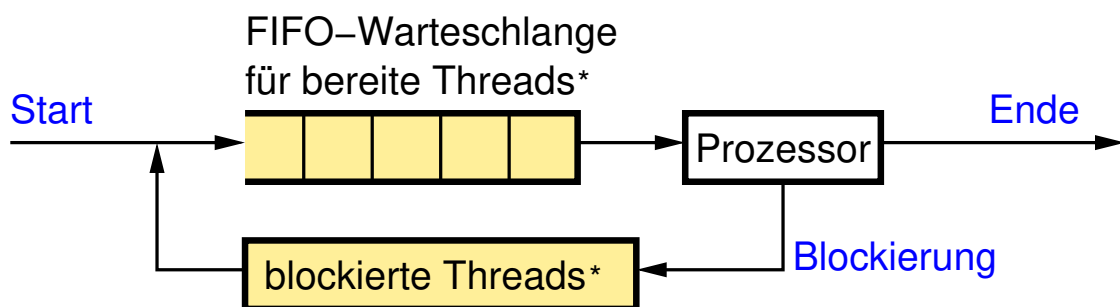
7.4 Scheduling-Algorithmen ...



★★

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

7.4.1 First Come First Served (FCFS) ...

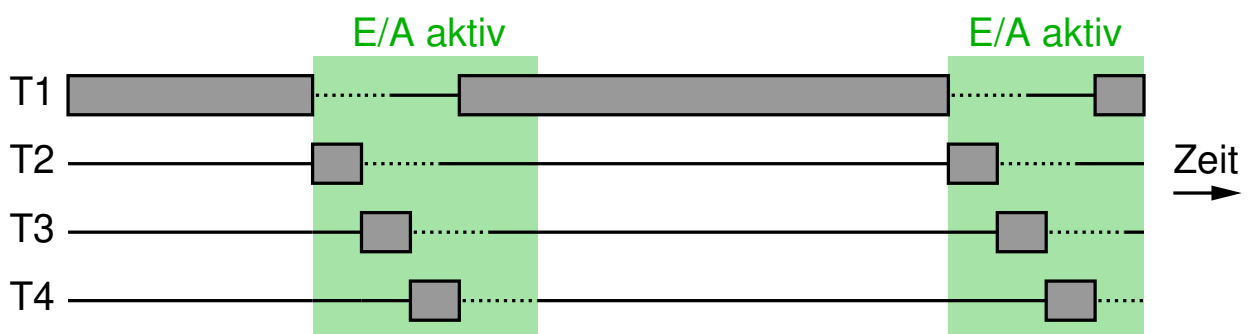


(Animierte Folie)

★

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 Scheduling-Algorithmen ...



★★★

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 **Restlaufzeiten**
- ➔ Variante für interaktive Systeme:
 - ➔ betrachte CPU-Burst als Job, mit Schätzung der Dauer

Anmerkungen zu Folie 343:

SJF für interaktive Systeme

Die CPU wird dabei an den Thread mit dem kürzesten CPU-Burst vor der nächsten Blockierung (Interaktion) zugewiesen. Da die Dauer dieses CPU-Bursts aber vorher nicht bekannt ist, werden Schätzungen verwendet, die auf gemessenen Zeiten der bisherigen CPU-Bursts basieren.

Meist wird ein exponentieller, gleitender Mittelwert eingesetzt (*Aging*):

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n \quad \text{mit } \alpha \in [0, 1]$$

Dabei ist S_{n+1} der neue Mittelwert, T_n die gemessene Länge des letzten CPU-Burst und S_n der bisheriger Mittelwert.

Ein Meßwert wirkt sich mit zunehmendem Alter immer weniger auf den Mittelwert aus, wobei der Faktor α bestimmt, wie stark das Gewicht des Meßwerts mit seinem Alter abnimmt: ein Meßwert T_i geht in den Mittelwert S_n mit dem Gewicht $\alpha(1 - \alpha)^{n-i}$ ein ($i \leq n$).

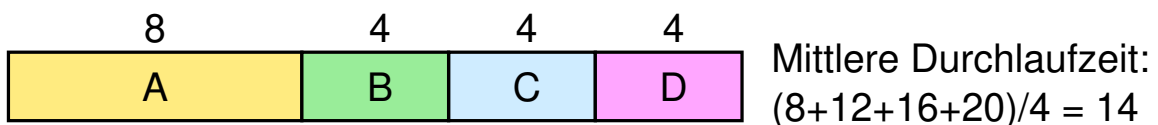
343-1

7.4.2 Shortest Job First (SJF) ...

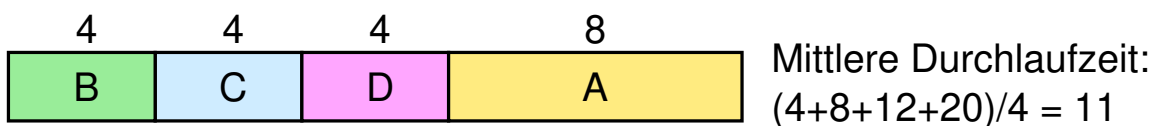


Beispiel (alle Jobs liegen gleichzeitig vor)

Vier Jobs in FCFS-Reihenfolge



Dieselben Jobs in SJF-Reihenfolge



7.4.2 Shortest Job First (SJF) ...

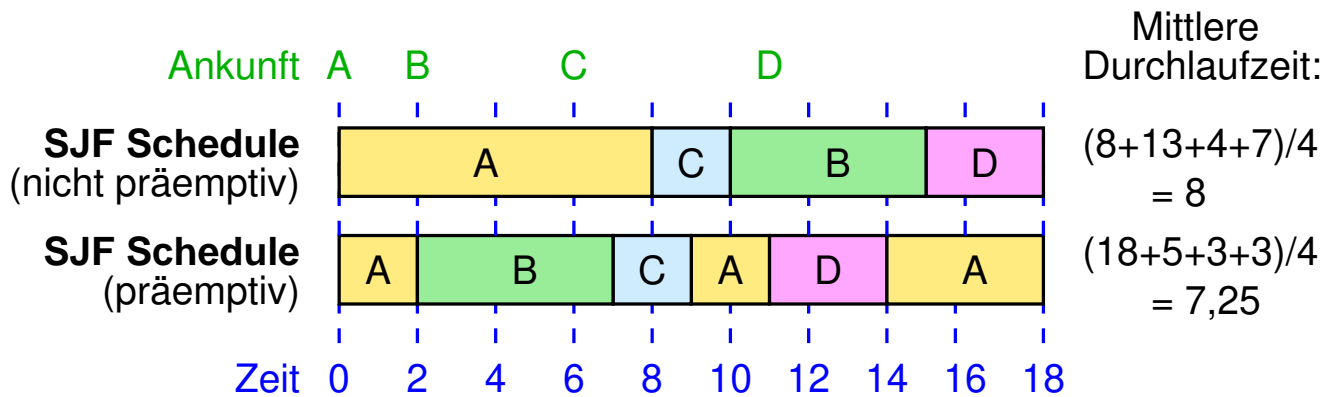


(Animierte Folie)



Beispiel (die Jobs treffen nacheinander ein)

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

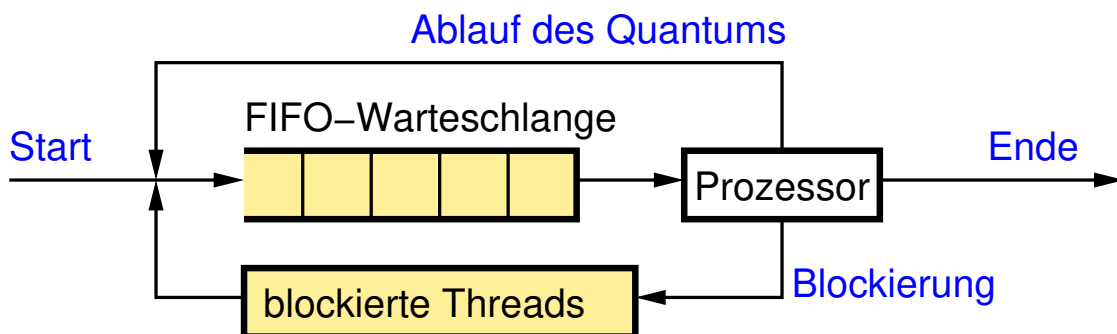


7.4 Scheduling-Algorithmen ...



7.4.3 Round Robin (RR), Zeitscheibenverfahren

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

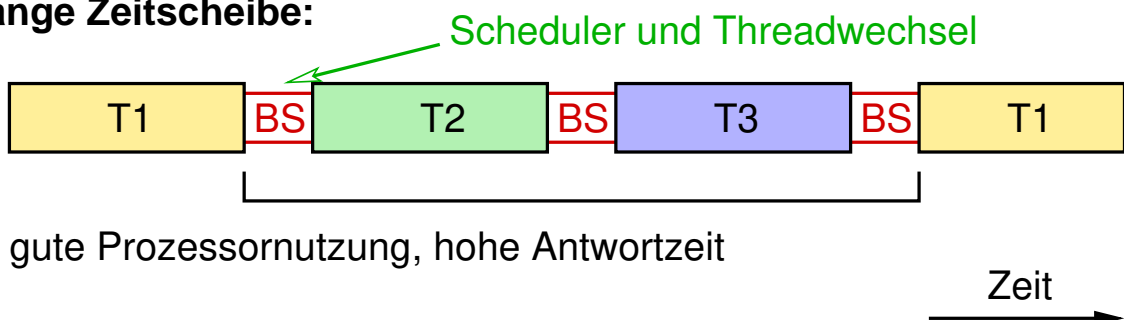


Diskussion

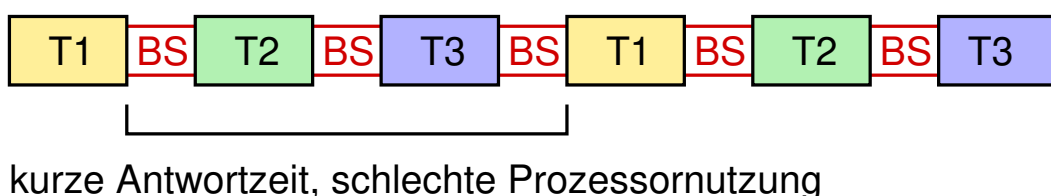
- ➔ Einfach zu realisieren
 - FIFO-Warteschlange aller rechenbereiten Threads und Timer-Interrupt
- ➔ Frage: Länge des Quantum
 - kurzes Quantum:
 - kurze Antwortzeiten
 - schlechte CPU-Nutzung durch häufige Threadwechsel
 - langes Quantum: umgekehrt
 - Praxis: 10 - 100 *ms*
- ➔ RR ist nicht fair:
 -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

Zeitscheibenlänge

Lange Zeitscheibe:

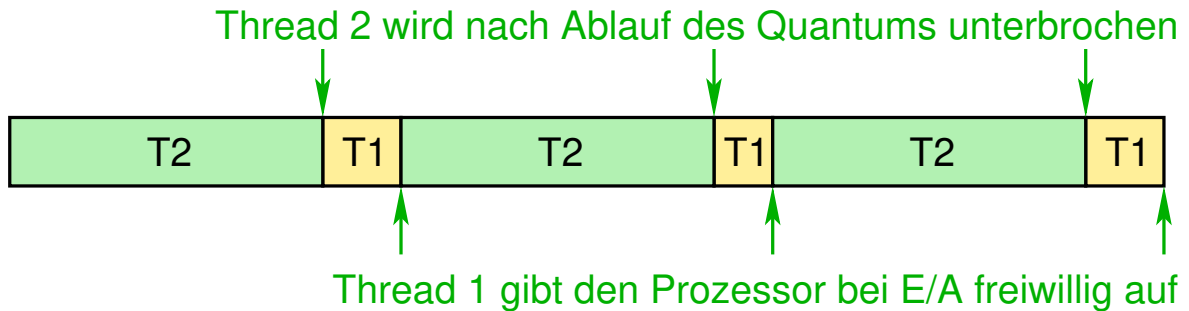


Kurze Zeitscheibe:



Fairness

- T1 E/A-lastiger Thread
- T2 CPU-lastiger Thread



- ➔ Thread 1 ist gegenüber Thread 2 benachteiligt

7.4 Scheduling-Algorithmen ...

7.4.4 Prioritätenbasiertes Scheduling

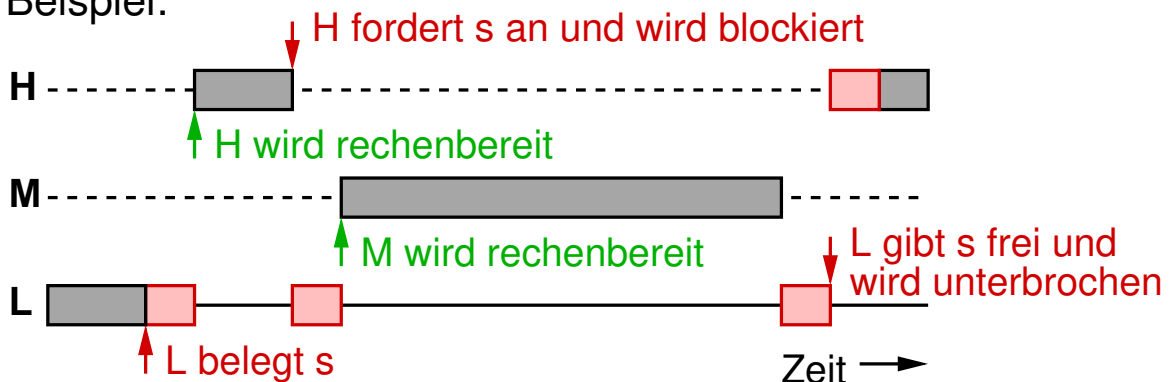
- ➔ Jeder Thread erhält eine (unterschiedliche) Priorität
 - ➔ CPU wird an den Thread mit der höchsten Priorität zugeteilt
- ➔ I.d.R. präemptiv
- ➔ Statische Prioritäten
 - ➔ Priorität eines Threads bei Erzeugung festgelegt
 - ➔ häufig bei Realzeit-Systemen
- ➔ Dynamische Prioritäten
 - ➔ Priorität kann 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

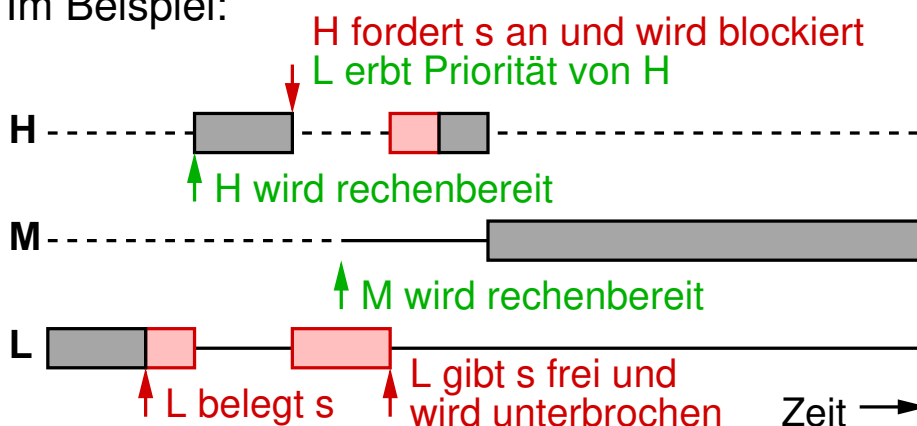
➔ Beispiel:



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:



Anmerkungen zu Folie 352:

Mehr dazu finden Sie u.a. in

- ➔ William Stallings. *Operating Systems – Internals and Design Principles*. Eighth Edition. Pearson, 2015. Kap. 10.2, S. 488 ff.

352-1

7.4 Scheduling-Algorithmen ...



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

7.4.5 Multilevel Scheduling ...



Beispiel: Statisches Multilevel Scheduling

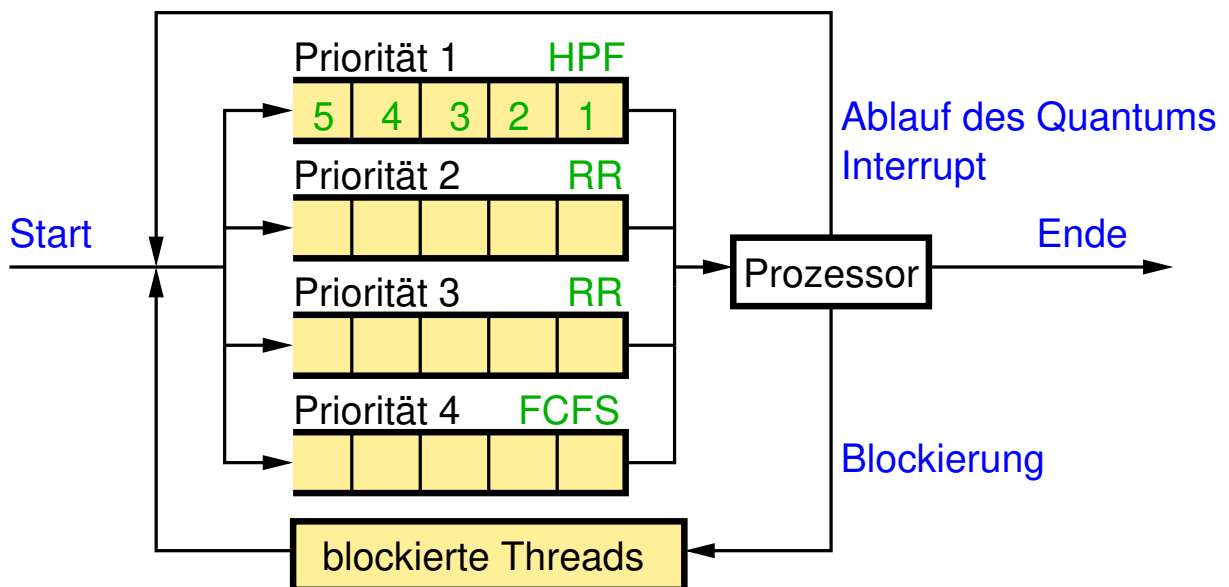
- ➔ Unterstützung verschiedener Betriebsarten in einem System
- ➔ Threads werden in **Prioritätsklassen** eingeteilt
 - pro Prioritätsklasse eine Warteschlange
 - unterschiedliche Scheduling-Strategien innerhalb der Warteschlangen

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

7.4.5 Multilevel Scheduling ...



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 Scheduling: Beispiele



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)

niedrige Priorität

⋮	⋮
2:	User Priorität 2
1:	User Priorität 1
0:	User Priorität 0
-1:	Kindprozeß terminiert
-2:	Terminalausgabe
-3:	Terminaleingabe
-4:	Seite auslagern
-5:	Warten auf Platte
⋮	⋮

hohe Priorität

Anmerkungen zu Folie 357:

Die normale User-Priorität bestimmt sich aus dem exponentiellen gleitenden Mittelwert der CPU-Nutzung: je höher die CPU-Nutzung desto niedriger die Priorität.

357-1

7.5 Scheduling: Beispiele ...



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

7.5.3 Scheduling in Linux (O(1) Scheduler) ...



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 $n.s$ -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

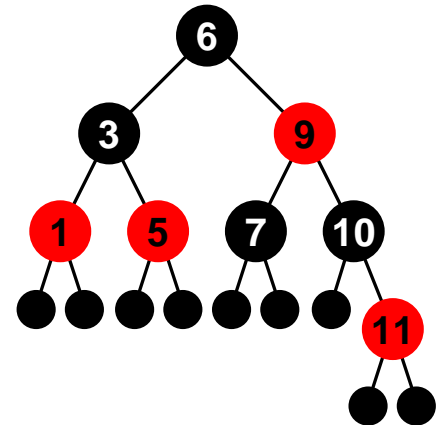
Anmerkungen zu Folie 361:

Der *Completely Fair Scheduler (CFS)* wird seit Linux Version 2.6.23 verwendet. Er ersetzt den $O(1)$ Scheduler für die Scheduling-Klasse `SCHED_OTHER`. Die Klassen `SCHED_FIFO` und `SCHED_RR` werden im Wesentlichen noch wie vorher implementiert.

CFS realisiert eine Variante des *Weighted Fair Queueings*, das z.B. auch in Netzwerk-Geräten (Router/Switches) zum Paket-Scheduling eingesetzt wird.

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 $\mathcal{O}(\log n)$



Anmerkungen zu Folie 362:

Red-Black-Trees sind u.a. auch die Grundlage für die Klassen `TreeSet` und `TreeMap` der Java Klassenbibliothek.

Eine detaillierte Beschreibung von *Red-Black-Trees* finden Sie u.a. in

- ➔ Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms* (Third Edition), MIT Press, 2009
[https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_\(2009\).pdf](https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf)
- ➔ <https://de.wikipedia.org/wiki/Rot-Schwarz-Baum>



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 hochpriorigen 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

Anmerkungen zu Folie 364:

Mehr zum Thema „Linux Scheduling“ finden Sie u.a. in

- ➔ William Stallings. *Operating Systems – Internals and Design Principles*. Eighth Edition. Pearson, 2015. Kap. 10.3
- ➔ Andrew S. Tanenbaum, Herbert Bos. *Moderne Betriebssysteme*. 4. Auflage. Pearson, 2016. Kap. 10.3.4

Vertiefte Informationen zum Thema „Completely Fair Scheduler“ finden Sie u.a. unter

- ➔ https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- ➔ <https://www.linuxjournal.com/node/10267>
- ➔ <https://developer.ibm.com/tutorials/l-completely-fair-scheduler>
- ➔ <https://opensource.com/article/19/2/fair-scheduling-linux>
- ➔ <https://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf>

364-1

7.6 Zusammenfassung / Wiederholung



- ➔ 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