HOCHSCHULE FULDA
UNIVERSITY OF APPLIED SCIENCES

# Betriebssysteme

## Stage 5 – Scheduling

# Was bisher geschah ...

- Geschichte der Rechner und Betriebssysteme
- Von-Neumann-Architektur
- Bootprozess und Sicherheitskonzept
- Interaktion mit dem Betriebssystem
  - Systemaufrufe (*system calls*)
  - Unterbrechungen (*interrupts*)
- Aufbau von Betriebssystemen
- Betriebssystemstrukturen

- Prozesse und Threads
  - Prozesszustände
  - Kontrollstrukturen
  - Prozesserzeugung

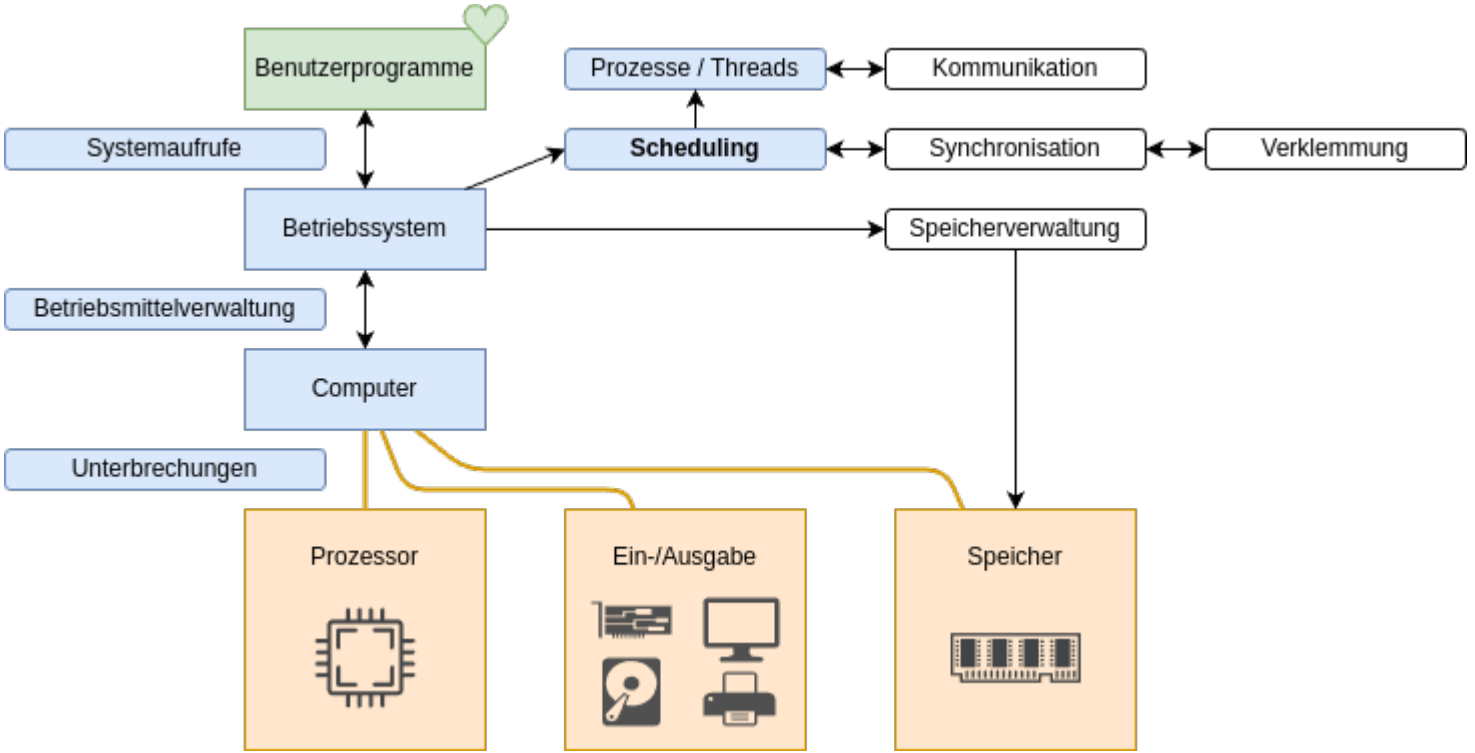
# Nachtrag: Threads

- Monolithische Betriebssysteme
  - Ein einziges großes Programm, alles im selben Adressraum

- Überlegung: Interrupts
  - Blockieren die CPU
  - Einige sind ggf. zeitkritischer als andere

- Linux nutzt Multithreading im Kernel
  - Erlaubt asynchrone Ausführung kann mittels Workqueue (kworker) geschehen (https://docs.kernel.org/core-api/workqueue.html)

# Übersicht


Benutzerprogramme
Systemaufrufe
Betriebssystem
Betriebsmittelverwaltung
Computer
Unterbrechungen
Prozesse / Threads
Kommunikation
Scheduling
Synchronisation
Verklemmung
Speicherverwaltung
Prozessor
Ein-/Ausgabe
Speicher

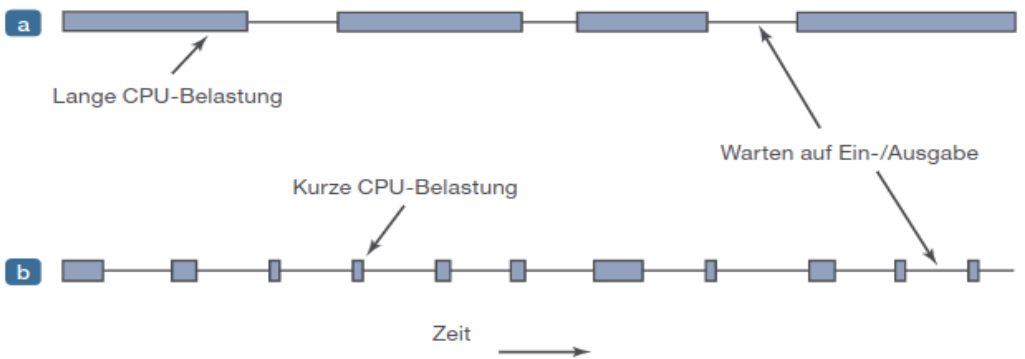
# Scheduling (Tanenbaum Kapitel 2.4)

- Grundlagen
  - Prozessverhalten
  - Bedientheorie
- Schedulingstrategien
  - Nicht-Preemptive Verfahren
  - Preemptive Verfahren
  - Echtzeit Scheduling
- Beispiele
  - Linux
  - Windows

# Grundlagen: Prozessverhalten

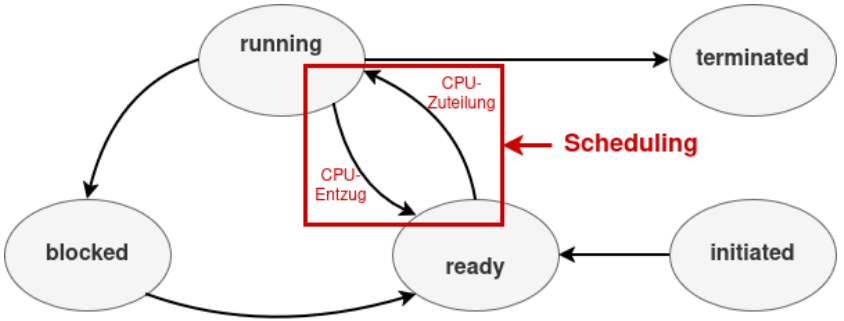
- Sind mehrere Prozesse oder Threads rechenbereit, so konkurrieren sie um die CPU

- Das Verhalten unterscheidet sich vor allem in
  - CPU-Intensiv (CPU bound): Häufige und ggf. auch lange CPU-Belastung
  - E/A-Intensiv (I/O bound): Häufige Blockierung durch warten auf eine I/O-Operation




(a) Ein CPU-intensiver Prozess (b) Ein E/A-intensiver Prozess
(Tanenbaum, Fig. 2.38)

# Scheduling: Grundlagen




- Scheduling besteht aus zwei Aspekten
  - Auswahl des nächsten Prozesses
    (→ Schedulingstrategie)
  - Wechsel von einem Prozess zu einem anderen
    (→ Kontextwechsel)

- Scheduler und Dispatcher

  Der Scheduler nutzt einen Scheduling-Algorithmus zur Auswahl des Prozesses, der als nächstes ausgeführt werden soll. Der Dispatcher führt den Kontextwechsel durch. Dieser Vorgang „verursacht Kosten“ (*dispatch delay*)

- Gründe für einen Kontextwechsel
  - Der Prozess wird beendet (ist fertig)
  - Der Prozess führt eine langsame I/O-Operation aus (z.B. lesen von der Festplatte)
  - Die Hardware benötigt die Hilfe des Betriebssystems und gibt einen Interrupt aus
  - Das Betriebssystem beschließt, den aktuellen Prozess zugunsten eines anderen zu pausieren

# Scheduling: Grundlagen

- Strategien der Ablaufsteuerung
  - Nicht-preemptives Scheduling: Ein rechnender Prozess bleibt solange aktiv, bis er endet oder sich selbst blockiert
    - Solche Verfahren sind für General Purpose Systeme mit interaktiven Benutzern i.d.R. nicht geeignet, können sich aber z.B. bei Stapelverarbeitung anbieten
  - Preemptives Scheduling: Rechnende Prozesse können suspendiert werden (Prozessorentzug) obwohl diese weiter rechenwillig sind
    - preemptive-resume: Fortsetzung ohne Verlust
    - preemptive-repeat: Beginn von Vorne
- Unterscheidung
  - Prioritäten-basierte Scheduling-Verfahren
    - Ordnen Prozessen Prioritäten zu (relative Wichtigkeit)
    - Prioritäten können extern vorgegeben, oder intern durch das Betriebssystem bestimmt werden
    - Prioritäten können statisch (ändern sich während der Laufzeit nicht) oder dynamischen (Änderung während der Laufzeit möglich) sein
  - Zeitscheiben-basierte Scheduling-Verfahren
    - Teilen Prozessen Zeitscheiben zu
    - Entzug des Prozessors mittels Unterbrechungen nach Ablauf der Zeitscheibe

# Ziele von Schedulingstrategien

- **Alle Systeme**
  - Fairness: "Gerechte" Behandlung aller Aufträge
    (z.B. alle rechenwilligen Prozesse erhalten den gleichen Anteil an der zur Verfügung stehenden Prozessorzeit)
  - Balance: Alle Teile des Systems werden ausgelastet, es entwickelt sich kein Flaschenhals

- **Stapelverarbeitungssysteme**
  - Durchsatz: Anzahl der erledigten Aufträge pro Zeiteinheit
  - Auslastung: Anteil der Zeit in der die Bedienstation sich im Zustand belegt befindet

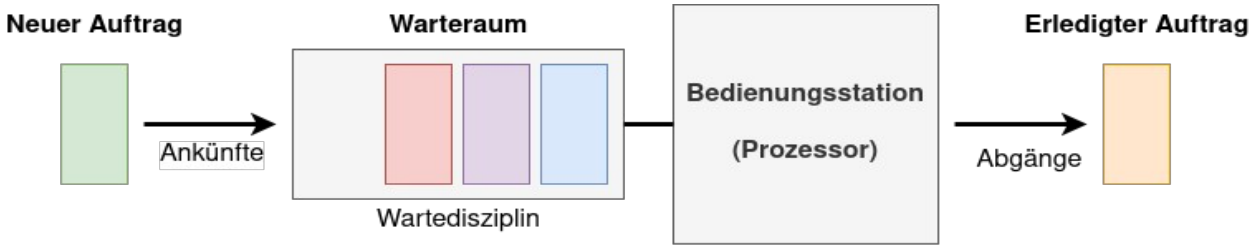
- **Interaktive Systeme (Dialogsysteme)**
  - Antwortzeit: Minimierung der Zeitdauer vom Eintreffen eines Auftrags bis Fertigstellung

- **Echtzeitsysteme**
  - Einhalten von Deadlines
  - Vorhersagbarkeit ist hier eines der wichtigsten Kriterien

# Warteschlangentheorie bzw. Bedienungstheorie

**Auftrag**

- Einheit zur Bearbeitung
  (z.B. Stapeljob, Dialogschritt)




**Wartedisziplin**

- FCFS (First-Come-First-Served)
- LIFO (Last-In-First-Out)
- Random (zufällig ausgewählt)

> **Bedienzeit**

- Zeitdauer für die reine Bearbeitung eines Auftrags durch die Bedienstation (den Prozessor)

- Hinweise
  - Ankünfte und Bedienungszeiten werden bei Bedienungsmodellen häufig durch stochastische Prozesse modelliert (es existieren sogar KI-basierte Verfahren, die aber eher keine Praxisrelevanz besitzen)
  - komplexere Bedienmodelle können mehrere Bedienstationen (Multiprozessorsystem), mehrere Warteräume, mehrphasige Bedienung und Rückführung teilweise bearbeiteter Aufträge enthalten

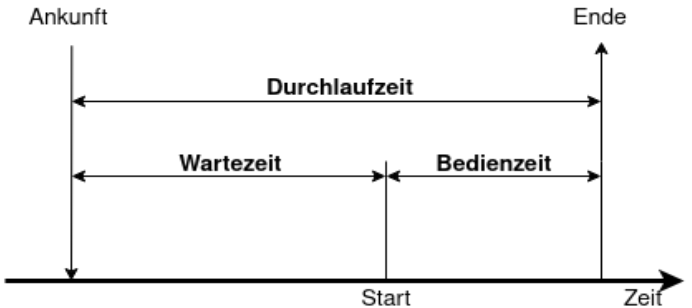
# Warteschlangentheorie bzw. Bedienungstheorie

- Durchlaufzeit (Antwortzeit / Verweilzeit)
  - Zeitdauer vom Eintreffen eines Auftrags bis Fertigstellung (→ $T_{Durchlaufzeit} = T_{Ende} - T_{Ankunft}$)

    Beispiel bei Dialogaufträgen: Die Zeitdauer zwischen der Eingabe eines Benutzers (z.B. Drücken der Return-Taste) bis zur Erzeugung einer zugehörigen Ausgabe (z.B. auf dem Bildschirm).

- Wartezeit
  - Zeitdauer vom Eintreffen eines Auftrags bis zur Bearbeitung
    (→ $T_{Durchlaufzeit} - T_{bedienzeit}$)
    (→ Durchschnitt: Summe der Wartezeiten / Anzahl Prozesse)

- Normalisierte Durchlaufzeit
  - Verhältnis zwischen Wartezeit und Bedienzeit
    (→ $T_{Durchlaufzeit} / T_{Bedienzeit}$)




| Prozess | Bedienzeit | Ankunft | Start | Ende | Durchlaufzeit | Wartezeit | Normalisiert |
|---|---|---|---|---|---|---|---|
| **A** | **1** | **0** | *0* | *1* | *1* | *1 - 1 = 0* | *1 / 1 = 1* |
| **B** | **100** | **1** | *1* | *101* | *100* | *100 - 100 = 0* | *100 / 100 = 1* |
| **C** | **1** | **2** | *101* | *102* | *100* | *100 - 1 = 99* | *100 / 1 = 100* |

# Schedulingverfahren

- Non-preemptive Scheduling (Tanenbaum Kapitel 2.4.2)
  - First Come First Serve (FCFS / FIFO)
  - Shortest Job First (SJF)
  - Shortest Remaining Time Next (SRTN)
  - Prioritätsscheduling (Prio)

- Preemptive Scheduling (Tanenbaum Kapitel 2.4.3)
  - Unterbrechendes Prioritätsscheduling
  - Round-Robin-Scheduling (RR) und Virtual Round Robin (VRR)
  - Mehrschlangen-Feedback-Scheduling (MLFQ)
  - Completely Fair Scheduling (CFS)

- Echtzeit Verfahren (Tanenbaum Kapitel 2.4.4)
  - *Beispiel: Rate-Monotonic Scheduling (RMS)*
  - *Beispiel: Earliest-Deadline-First (EDF)*

# Verfahren: First Come First Served

- Algorithmus
  - Prozesse werden ausschließlich nach ihrem Ankunftszeitpunkt ausgewählt
  - Vollständige Abarbeitung eines Auftrags bevor der nächste begonnen wird

- Vor- und Nachteile
  + Minimierung der Kontextwechsel (z.B. gut bei Stapelverarbeitung)
  - Erfordert Bereitschaft von Prozessen die CPU auch mal selbständig abzugeben (z.B. durch I/O)
  - Führt zu Konvoi-Effekt
    - Prozesse mit langen CPU-Stößen werden begünstigt
    - Prozesse mit kurzen CPU-Stößen werden benachteiligt

- Implementierung
  - Die Ready-Queue wird als FIFO-Liste verwaltet

# Verfahren: First Come First Served

- Rechenbeispiel

| Prozess | Ankunft | Bedienzeit |
|---|---|---|
| A | 1 | 6 |
| B | 3 | 10 |
| C | 4 | 3 |

- Resultierender Zeitplan

| 1 | | | | 5 | | | | | 10 | | | | | 15 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | B | B | B | B | B | B | B | B | B | B | C | C | C |
| A | | B | C | | | | | | | | | | | | | | | |

- Wartezeit

| Prozess | Bedienzeit | Wartezeit | Durchlaufzeit |
|---|---|---|---|
| **A** | **6** | *0* | *6* |
| **B** | **10** | *4* | *14* |
| **C** | **3** | *13* | *16* |

- Durchschnittliche Wartezeit (hängt von der Reihenfolge der Ankunft ab!)

  (0 + 4 + 13) / 3 = 17/3 = 5.66

# Verfahren: Shortest Job First

- Algorithmus
  - Von allen rechenwilligen Prozessen wird der mit der kleinsten Bedienzeitanforderung ausgewählt
  - Vollständige Abarbeitung des Auftrags bevor der nächste begonnen wird
  - Bei gleicher Bedienzeitanforderung mehrerer Aufträge wird nach FCFS gewählt

- Vor- und Nachteile
  - \+ Shortes Job First ist in dem Sinne *optimal* in der Menge aller möglichen Algorithmen, dass er die kürzeste mittlere Wartezeit für alle Aufträge sichert
  - \- Gefahr des Verhungerns langlaufender Prozesse

- Implementierung
  - Nur bei vorheriger Kenntnis der Bedienzeit möglich
  - Entscheidung auf Grund vorheriger Beobachtungen möglich

# Verfahren: Shortest Job First

- Rechenbeispiel

| Prozess | Ankunft | Bedienzeit |
|---|---|---|
| A | 1 | 6 |
| B | 3 | 10 |
| C | 4 | 3 |

- Resultierender Zeitplan

| 1 | | | | 5 | | | | | 10 | | | | | 15 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | C | C | C | B | B | B | B | B | B | B | B | B | B |
| A | | B | C | | | | | | | | | | | | | | | |

- Wartezeit

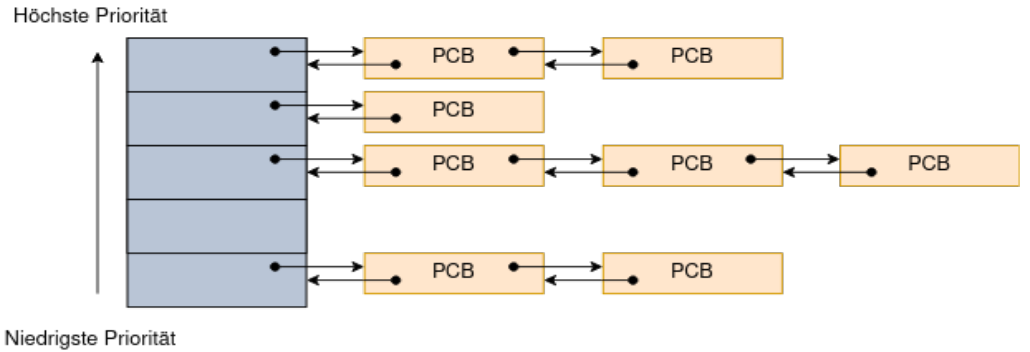
| Prozess | Ankunft | Bedienzeit | Durchlaufzeit | Wartezeit |
|---|---|---|---|---|
| **A** | **1** | **6** | *6 - 1 (+ 1) = 6* | *6 - 6 = 0* |
| **B** | **3** | **10** | *19 - 3 (+ 1) = 17* | *17 - 10 = 7* |
| **C** | **4** | **3** | *9 - 4 (+ 1) = 6* | *6 - 3 = 3* |

- Durchschnittliche Wartezeit: (0 + 7 + 3) / 3 = 10 / 3 = 3,3

# Verfahren: Prioritätsscheduling

- Mehrere Warteschlangen unterschiedlicher Priorität

- Algorithmus
  - Jeder Auftrag besitze eine statische Priorität
  - Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt
  - Bei gleicher Priorität wird z.B. nach FCFS ausgewählt




Mehrere Ready-Listen nach Priorität sortiert

# Verfahren: Prioritätsscheduling

Im Beispiel: **Hohe Zahl = Hohe Priorität**

In der Realität ist dies eine Frage der Implementierung und kann sich unterscheiden.

- Rechenbeispiel

| Prozess | Ankunft | Bedienzeit | Priorität |
|---|---|---|---|
| A | 1 | 6 | 2 |
| B | 3 | 10 | 3 |
| C | 4 | 3 | 4 |

- Resultierender Zeitplan

| 1 | | | | 5 | | | | | 10 | | | | | 15 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | C | C | C | B | B | B | B | B | B | B | B | B | B |
| A/2 | | B/3 | C/4 | | | | | | | | | | | | | | | |

- Wartezeit

| Prozess | Ankunft | Bedienzeit | Priorität | Durchlaufzeit | Wartezeit |
|---|---|---|---|---|---|
| **A** | **1** | **6** | **2** | *6 – 1 (+ 1) = 6* | *6 – 6 = 0* |
| **B** | **3** | **10** | **3** | *19 – 3 (+ 1) = 17* | *17 – 10 = 7* |
| **C** | **4** | **3** | **4** | *9 – 4 (+ 1) = 6* | *6 – 3 = 3* |

- Die Durchschnittliche Wartezeit hängt von der Priorität der wartenden Prozesse ab!
  - (0 + 7 + 3) / 3 = 10 / 3 = 3,33

# Schedulingverfahren

- Non-preemptive Scheduling (Tanenbaum Kapitel 2.4.2)
  - First Come First Serve (FCFS / FIFO)
  - Shortest Job First (SJF)
  - Shortest Remaining Time Next (SRTN)
  - Prioritätsscheduling (Prio)

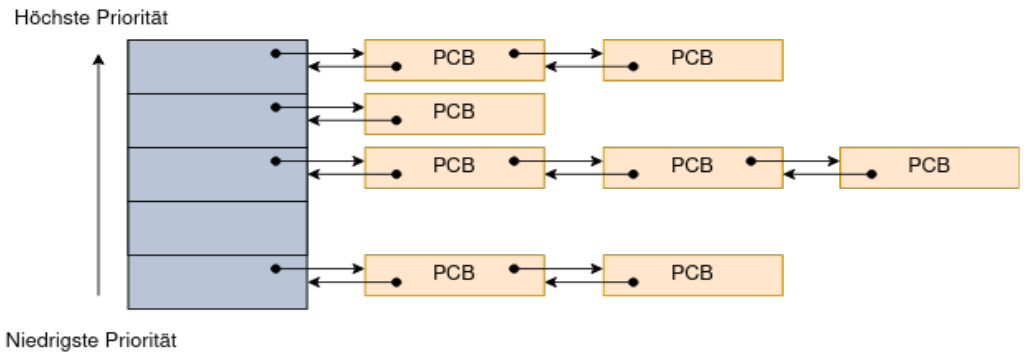
- Preemptive Scheduling (Tanenbaum Kapitel 2.4.3)
  - Unterbrechendes Prioritätsscheduling
  - Round-Robin-Scheduling (RR) und Virtual Round Robin (VRR)
  - Mehrschlangen-Feedback-Scheduling (MLFQ)
  - Completely Fair Scheduling (CFS)

- Echtzeit Verfahren (Tanenbaum Kapitel 2.4.4)
  - *Beispiel: Rate-Monotonic Scheduling (RMS)*
  - *Beispiel: Earliest-Deadline-First (EDF)*

# Verfahren: Unterbrechendes Prioritäts-Scheduling

- Mehrere Warteschlangen unterschiedlicher Priorität
  - Prozesswechsel, falls ein höher priorisierter Prozess rechenwillig wird

- Algorithmus
  - Jedem Prozess ist ein statische (oder dynamische) Priorität zugeordnet
  - Prozesse werden gemäß ihrer Priorität in eine Warteschlange eingereiht
  - Es wird jeweils der rechenwillige Prozess mit der höchsten Priorität ausgewählt und bedient
  - Wird ein Prozess höherer Priorität rechenwillig (z.B. nach Beendigung einer Blockierung), so wird der laufende Prozess unterbrochen (*preemption*) und wieder in die *Ready Queue* eingefügt



# Schedulingverfahren: Round Robin

- Round Robin ist ein Zeitscheibenverfahren

- Algorithmus
  - Die Menge der rechenwilligen Prozesse wird linear geordnet
  - Jeder rechenwillige Prozess erhält den Prozessor nacheinander, jeweils für eine feste Zeitdauer q, die Zeitscheibe (*time slice*) oder Quantum genannt wird
  - Nach Ablauf des Quantums wird der Prozessor entzogen und dem nächsten zugeordnet (*preemptive-resume*)
  - Tritt vor Ende des Quantums Blockierung oder Prozessende ein, erfolgt der Prozesswechsel sofort
  - Neu eintreffende Aufträge werden am Ende der Warteschlange eingefügt

- Implementierung
  - Die Zeitscheibe wird durch einen Timer-Interrupt realisiert
  - Die Ready Queue wird i.d.R. als lineare Liste verwaltet
    - Erreicht ein Prozess das Ende seines Quantums, so wird er am Ende der Liste wieder eingefügt

# Verfahren: Round Robin (Quantum = 4)

- Rechenbeispiel

| Prozess | Ankunft | Bedienzeit |
|---|---|---|
| A | 1 | 6 |
| B | 3 | 10 |
| C | 4 | 3 |

- Resultierender Zeitplan

| 1 | | | | 5 | | | | | 10 | | | | | 15 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | B | B | B | B | C | C | C | A | A | B | B | B | B | B | B |
| A | | B | C | | | | | | | | | | | | | | | |

- Wartezeit

| Prozess | Ankunft | Bedienzeit | Durchlaufzeit | Wartezeit |
|---|---|---|---|---|
| **A** | **1** | **6** | *13 - 1 (+ 1) = 13* | *13 - 6 = 7* |
| **B** | **3** | **10** | *19 - 3 (+ 1) = 17* | *17 - 10 = 7* |
| **C** | **4** | **3** | *11 - 4 (+ 1) = 8* | *8 - 3 = 5* |

- Durchschnittliche Wartezeit: (7 + 7 + 5) / 3 = 19 / 3 = 6,3

# Verfahren: Round Robin

- Round-Robin ist einfach und weit verbreitet
  - Alle Prozesse werden als gleich wichtig angenommen und fair bedient
  - Idealfall: Jeder rechenwillige Prozess n erfährt 1/n der Prozessorleistung
    (Zeiten für Kontextwechsel als Null angenommen)

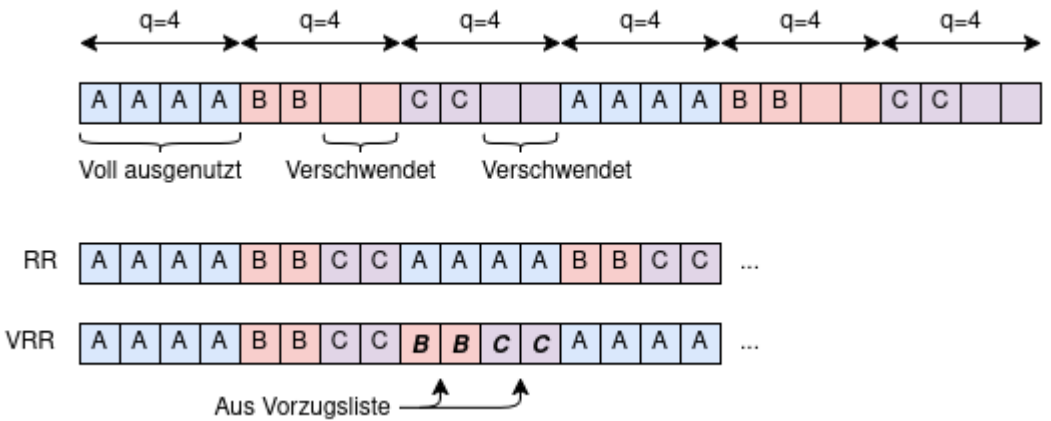
- Gerechtigkeit
  - Bevorteilt CPU-Lastige Prozesse
    (kommen nach voller Nutzung ihrer Zeitscheibe wieder in die Ready-Liste)
  - Benachteiligt IO-Lastige Prozesse
    (kommen wieder in die Ready-Liste, jedoch ohne die vorherige Zeitscheibe aufgebraucht zu haben)

- Kritisch für die Effizienz ist die Wahl der Zeitscheibenlänge (des Quantums q)
  - Quantum zu klein → häufige Prozesswechsel, sinnvolle Prozessornutzung sinkt
    (q → 0: Der Overhead für Kontextwechsel überwiegt)
  - Quantum zu groß → schlechte Antwortzeiten bei kurzen interaktiven Aufträgen
    (q → unendlich: Round-Robin verhält sich wie FCFS)

# Verfahren: Virtual Round Robin

- Erweiterung des klassischen Round Robin
  - Vermeiden der Bevorteilung CPU-intensiver Prozesse

- Nutzung einer Vorzugsliste
  - Prozesse die ihre Zeitscheibe nicht aufgebraucht haben kommen in eine Vorzugsliste
  - Die Vorzugsliste wird immer vor der „normalen“ Liste abgearbeitet
  - Prozesse in der Vorzugsliste erhalten den Rest ihrer nicht verbrauchten Zeitscheibe mit Priorität




**Beispielhafte Darstellung RR vs. VRR**

Hinweise

Zeile 1 ist nur zur Veranschaulichung. Es existieren natürlich keine leeren Slots.

Wann, wo und ob ein Prozess aus der Vorzugsliste eingefügt werden kann, hängt stark vom realen Verhalten der Prozesse ab.

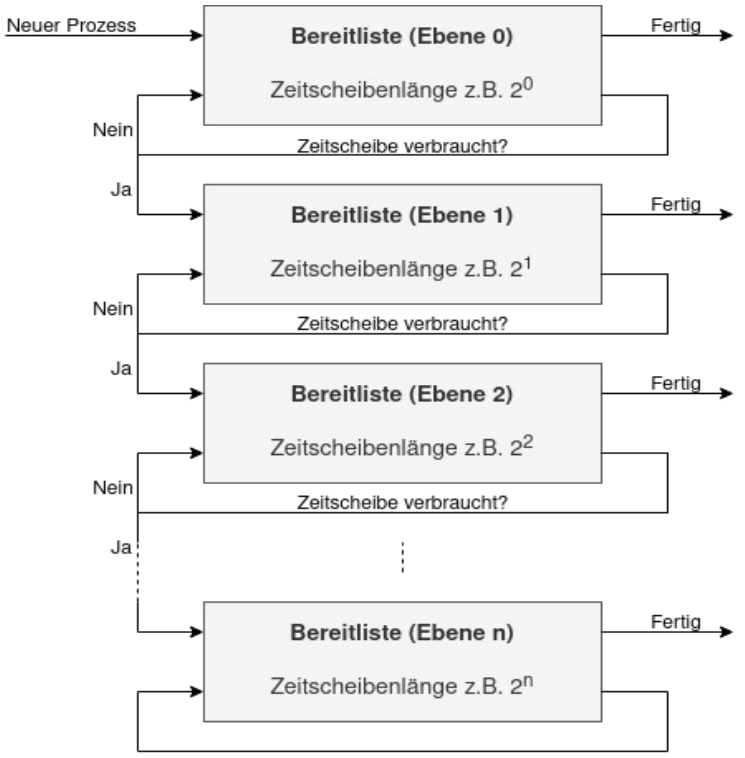
# Verfahren: Multi Level Feedback Queue (MLFQ)

- Erweiterung des Mehrschlangen-Scheduling
  - Rechenwillige Prozesse werden dynamisch in Warteschlangen verschiedener Priorität eingeordnet. Die Zuordnung wird während der Laufzeit abhängig vom Verhalten immer wieder angepasst.

- Algorithmen zur Neubestimmung der Priorität sind wesentlich
  - Beispiel 1: Wenn ein Prozess blockiert, wird die Priorität nach Ende der Blockierung um so größer, je weniger er von seinem Quantum verbraucht hat (Bevorzugung von I/O-intensiven Prozessen)
  - Beispiel 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von CPU-intensiven Prozessen)
  - Beispiel 3: Wenn ein Prozess lange nicht bedient worden ist, wird seine Priorität verbessert (Altern bzw. Verhungern vermeiden)

# Verfahren: MLFQ (Beispiel einer Implementierung)

- Mehrere Prioritätsebenen
  - Je höher die Priorität, umso kürzer die Zeitscheibe
  - Je niedriger die Priorität, umso länger die Zeitscheibe
- Unterschiedliche Verfahren je Ebene möglich
  - Zum Beispiel Round Robin, First Come First Served, …
- Neue Prozesse beginnen mit höchster Priorität
  - Zeitscheibe verbraucht?
    → Prozess wandert in die nächste Stufe
  - Zeitscheibe nicht verbraucht?
    → Prozess bleibt in gleicher Stufe
    (jedoch wird die Zeitscheibe nicht wieder aufgefüllt)
- Regelmäßiger “Boost” gegen das Verhungern
  - Prozesse aus unterster Ebene werden nach oben verschoben



# Bewertung: Multi Level Feedback Queue (MLFQ)

- Mit wachsender Bedienzeit sinkt die Priorität
  - → Kurzläufer werden bevorzugt, Langläufer werden herabgesetzt
- Wachsende Länge des Quantums mit fallender Priorität
  - → verringert die Anzahl der notwendigen Kontextwechsel
    (Einsparen von Overhead bei Prozessen, die sowieso lange laufen möchten)
- Verbesserung der Priorität nach Beendigung einer Blockierung
  - → berücksichtigt das I/O-Verhalten
    (Bevorzugung von I/O-intensiven Prozessen, die nur kurz auf die CPU möchten)
- Unterscheidung von Terminal I/O und sonstigem I/O
  - → Prozesse, die I/O im Terminal (Tastatur/Bildschirm) durchführen, können bevorzugt werden, da diese interaktiven Prozesse ggf. kurze Antwortzeiten bieten sollen

*Das Verfahren ist sehr flexibel und kann – je nach Implementierung – unterschiedliche Metriken zur Bewertung der Priorität von Prozessen heranziehen.*

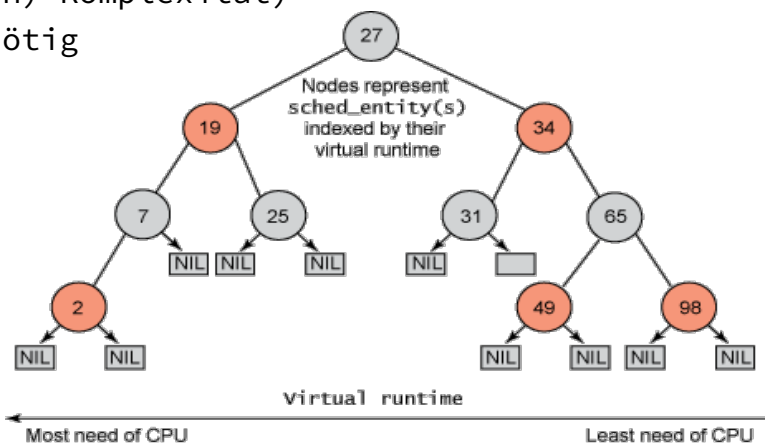
# Completely Fair Scheduling (CFS)

- Grundidee
  - Gerechte Verteilung von Rechenzeit auf alle Tasks (auch solche, die aktuell blockiert sind)
    → Jeder Task erhält genau 1 / num_tasks Prozent der CPU
  - Abkehr von statischen Zeitscheiben

- Tasks (Prozesse und Threads) erhalten ein virtuelles Laufzeitkonto
  - Tasks im Zustand *running* verbrauchen Zeit auf dem Konto
  - Tasks, im Zustand *ready/blocked* verbrauchen keine Zeit (sondern *sparen sich Rechenzeit an*)

- Task mit geringster Virtual Runtime (größter Rückstand) bekommt den Prozessor
  - Zeit-geordneter Rot-Schwarz-Baum für Taskverwaltung (→ O(log n)-Komplexität)
  - Keine Heuristiken zur Beurteilung von CPU- vs. I/O-intensiv nötig

    I/O-intensive Prozesse, besitzen ganz natürlich eine niedrige *vruntime*,
    und befinden sich daher eher links im Baum → Erhalten die CPU öfter



# Completely Fair Scheduling (CFS)

- Virtual runtime (vruntime)
  - Zuweisung einer dynamischen *timeslice* mittels high resolution timer interrupt
  - Länge der Zeitscheibe ist von vielen Faktoren abhängig

- CPF Prioritäten (0 – 139 / −99 – +19) und Niceness
  - Nice (−20 – +19) führt zu längerer bzw. kürzerer *vruntime*
    (z.B. vruntime += t * weight)

- Linux
  - Scheduling classes / policies
    - Mehrere Strategien verfügbar und parallel nutzbar
      - Real-time (SCHED_FIFO, SCHED_RR)
        → POSIX fixed-priority real-time scheduling (Interessantes Paper)
      - Fair (SCHED_NORMAL) → CFS
      - Idle (SCHED_IDLE)
  - Scheduling Groups
    - Erlaubt die CPU-Zeit fair über Gruppen zu verteilen
    - Zum Beispiel: Prozess-Gruppen, Benutzergruppen, ...

# Beispiel: Scheduling in Linux

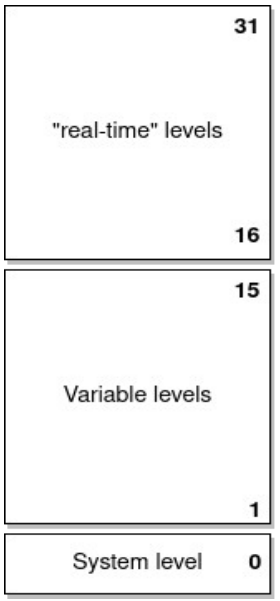
- Linux 1.2
  - Zyklische Liste, Round-Robin
- Linux 2.2
  - Scheduling-Klassen (Echtzeit, Non-Preemptive, Nicht-Echtzeit) und Unterstützung für Multiprozessoren
- Linux 2.4: Round Robin
  - O(n)-Komplexität (jeder Task-Kontrollblock muss angefasst werden)
  - Teilweiser Ausgleich bei nicht verbrauchter Zeitscheibe
- Linux 2.6: O(1)-Scheduler (Ingo Molnár)
  - O(1)-Komplexität (konstanter Aufwand für Auswahl unabhängig von Anzahl Tasks)
  - Mehrere Run Queues, eine je Priorität
  - Zahlreiche Heuristiken für Entscheidung I/O-intensiv oder rechenintensiv
- Seit Linux Kernel 2.6.23: CFS (Ingo Molnár) (Ankündigung)
  - Basis: Rotating Staircase Deadline Scheduler (RSDL) (Ankündigung Con Kolivas)
- Ganz neu: „Earliest Eligible Virtual Deadline First" (EEVDF) (Ankündigung Jonathan Corbet)

# Beispiel: Windows NT

- Preemptiv und Prioritätsbasiert, Round-Robin bei Threads gleicher Priorität
  - Thread-Prioritäten 0 bis 31 aufgeteilt in drei Bereiche
    → system, variable, real-time

- Art des Prozesses bestimmt das Zeitquantum
  - Länge des Quantums ist verschieden für Desktop- bzw. Server-Variante (Server: 6-fach länger), Quantum wird für Vordergrundthreads verdoppelt

- Temporäre Prioritätsanhebung (Priority Boost) in den Levels 1-15
  - z.B. bei Komplettierung einer I/O-Operation, einem Fensterthread der den Focus erhält
    - Dynamic Boost (process_priority_class + relative_thread_priority + boost)
      → Abschluss von Ein-/Ausgabe (Festplatten) +1
      → Mausbewegung, Tastatureingabe +6
      ...




- Fortschrittsgarantie (verhindert das Aushungern von Threads)
  - Alle 3-4 s erhalten bis zu 10 „benachteiligte“ Threads für zwei Zeitscheiben die Priorität 15.

- Die Verdrängung erfolgt auch dann, wenn es sich um einen Thread des Kernel handelt (nicht so bei UNIX / Linux)

# Idle loop / Leerlaufprozess

- Was tut eine CPU, wenn sie nichts zutun hat?
  - Es ist nicht trivial für eine CPU, nichts zu tun :)

- Überlegungen
  - CPU kann nicht einfach nichts tun
    Moderne CPUs besitzen Mechanismen um CPU-Cores in energiersparende “Schlafzustände” zu versetzen
  - Schedulder soll nicht “leerlaufen”
    Einsparung einer gesonderten Spezialfallbehandlung bei jeder Scheduling-Entscheidung

- Implementierung
  - Von Windows bekannter “Leerlaufprozess”
  - Linux nutzt einen sog. Idle Loop

- Herausforderung
  - Versetzen der CPU in leichte/tiefe Schlafzustände ist sebst “kostspielig”
  - Statistische Abschätzung im Idle Loop, ob bzw. welcher Schlafzustand sich lohnt

Mehr dazu: https://docs.kernel.org/admin-guide/pm/cpuidle.html