2405 Betriebssysteme
IV. Scheduling von Prozessen

Thomas Staub, Markus Anwander
Universität Bern

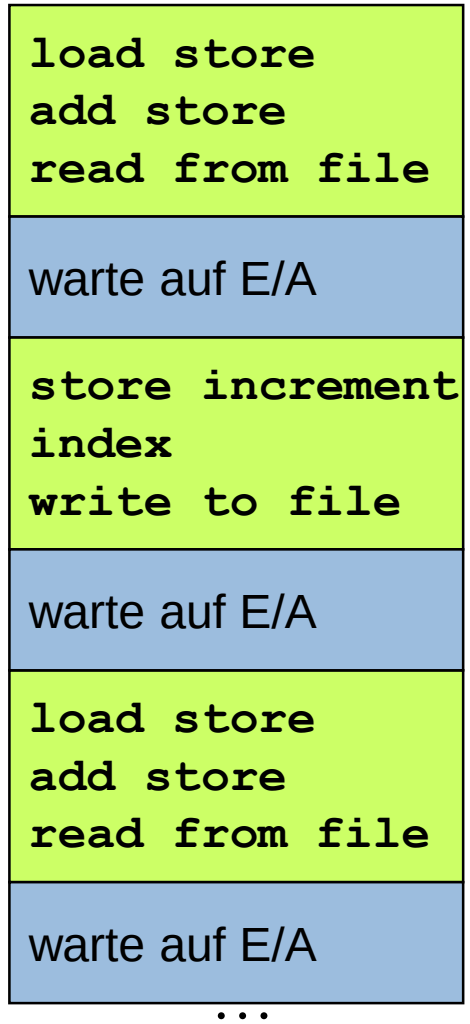
b
UNIVERSITÄT
BERN

Inhalt

1.1 CPU- und E/A-Bursts

Prozessausführung besteht aus einer Folge von Instruktionen

- CPU Bursts
  - Sequenz von CPU-Zyklen
- E/A-Bursts
  - Ein-/Ausgabe



1.2 CPU-Scheduler

- Aufgabe: Auswahl des nächsten Prozesses aus der Ready-Queue, welcher in den Status *rechnend* wechseln darf (kurzfristiges Scheduling).
- Scheduling-Entscheidungen in folgenden Situationen
  - Prozess wechselt von *rechnend* nach *blockiert*
  - Prozess wechselt von *rechnend* nach *bereit*
  - Prozess wechselt von *blockiert* nach *bereit*
  - Prozess terminiert
- nicht präemptives Scheduling können rechnendem prozess CPU nicht wegnehmen
- präemptives Scheduling

1.3 Scheduling-Kriterien

- Fairness
- CPU-Auslastung
  - Mass für Prozessorauslastung bei Anwendungsausführung
- Durchsatz
  - Anzahl verarbeiteter Aufträge pro Zeiteinheit
- Verweilzeit
  - Zeit zwischen Starten und Beenden eines Prozesses
- Wartezeit
  - Zeit in Ready-Queue
- Antwortzeit
  - Zeit zwischen Anforderung und der ersten Antwort
- Realzeitverhalten
  - Einhaltung der von den Anwendungen vorgegebenen Realzeitgarantien

2.1 First Come First Serve

- nicht präemptiv
- Beispiel: 3 Prozesse starten zum Zeitpunkt 0
  P_1(24 Zeiteinheiten (ZE)), P_2 (3 ZE), P_3 (3 ZE)

a) Wartezeit: (0 + 24 + 27) ZE / 3 = 17 ZE
Konvoi-Effekt: schnelle Prozesse hinter einem langsamen

b) Wartezeit: (0 + 3 + 6) ZE / 3 = 3 ZE

2.2 Shortest Job First

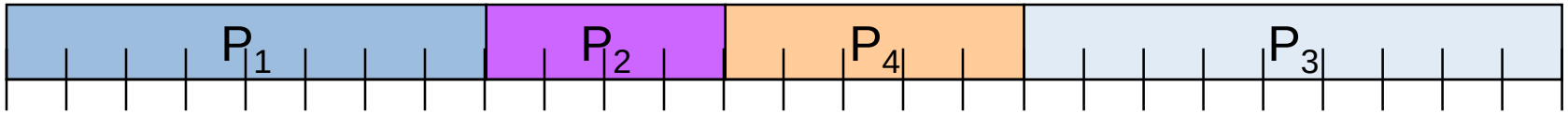
- Auswahl des Prozesses mit dem kürzesten nächsten CPU-Burst
- Optimierung der Wartezeit
- Optionen
  - nicht präemptiv
    Ein einmal rechnender Prozess wird nicht verdrängt, bevor CPU-Burst beendet ist.
  - präemptiv (Shortest Remaining Time First)
    Ein rechnender Prozess kann sofort verdrängt werden,
    falls ein neuer Prozess ankommt und es gilt:
    [CPU-Burst-Länge des neuen Prozesses < verbleibende Länge des aktuellen CPU-Bursts].

2.2.1 Beispiel: (nicht) präemptives Shortest Job First (SJF)

Prozess	Ankunftszeit	Burst-Zeit
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

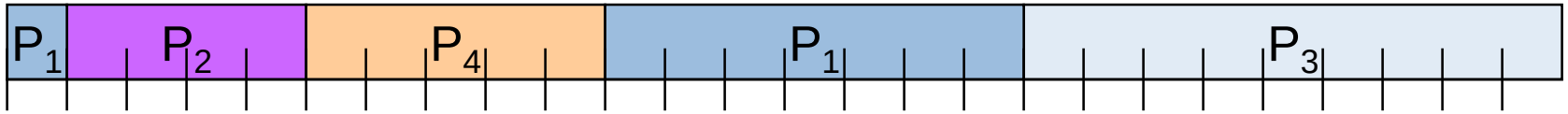
nicht präemptiv

Wartezeit: (0+7+15+9)/4=7.75



präemptiv

Wartezeit: (9+0+15+2)/4=6.5



2.2.2 CPU-Burst-Bestimmung

- nur Schätzung basierend auf vorhergehenden Bursts möglich
- z.B. exponentielle Mittelwertbildung

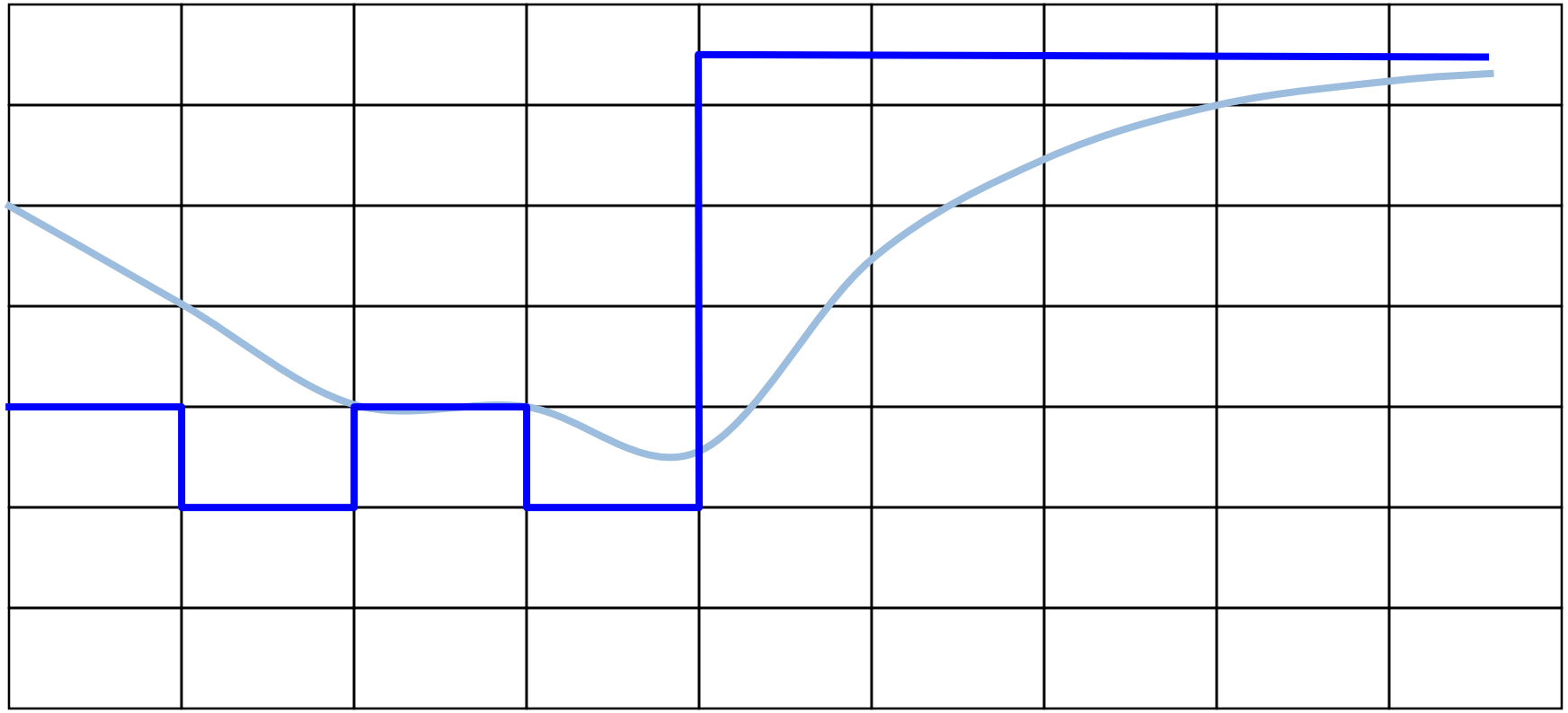
$$\tau_{n+1} = \alpha\ t_n + (1-\alpha)\ \tau_n$$

t_n = Länge des n. CPU-Bursts

τ_{n+1} = vorhergesagter Wert für den nächsten CPU-Burst

$0 \leq \alpha \leq 1$

2.2.3 Beispiel: CPU-Burst-Bestimmung



6	4	6	4	13	13	13	13	CPU-Burst (t_i)
10	8	6	6	5	9	11	12	Schätzung (τ_i)

$\alpha = 0.5$

2.3 Prioritäts-Scheduling

- Jeder Prozess erhält eine Prioritätsnummer.
- Selektion des Prozesses mit der höchsten Priorität (niedrigste Prioritätsnummer) aus Ready-Queue ready queue: prozesse in bereit
- SJF ist Prioritäts-basiertes Verfahren mit Priorität = erwartete Burst-Zeit
- Problem: Aushungern,
  d.h. Prozesse mit niedrigerer Priorität werden unter Umständen nie bedient
- Lösung: Altern (Aging), Priorität steigt mit der Wartezeit
- Prioritäts-Scheduling kann präemptiv oder nicht präemptiv sein.

2.3.1 Beispiel: Prioritäts-Scheduling

Prozess	Burst-Zeit	Priorität
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

2.3.2 Interne und externe Prioritäten

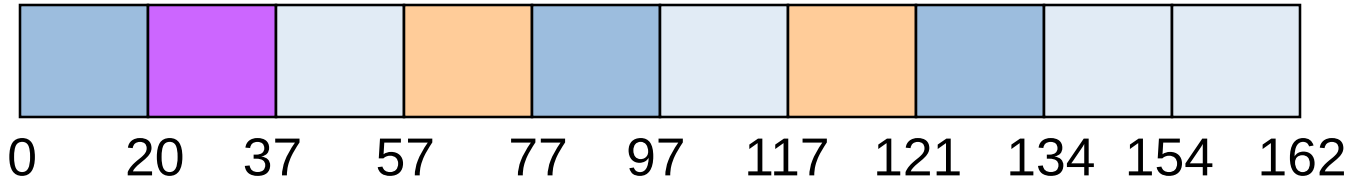
- Interne Prioritäten basieren auf messbaren Prozesseigenschaften.
  - Zeitbegrenzungen
  - Speicherbedarf
  - Zahl offener Dateien
  - E/A-Tätigkeit
- Externe Prioritäten basieren auf Kriterien ausserhalb des Betriebssystems.
  - Wichtigkeit des Benutzers
  - bezahlte Gebühren

2.4 Round Robin

- geeignet für Time-Sharing
- Jeder Prozess erhält eine kleine Einheit CPU-Zeit (Zeitquantum), z.B. 10 – 100 ms.
- Einreihung des Prozesses nach Ablauf des Zeitquantums in die Ready-Queue
- Bei n Prozessen und einem Zeitquantum von q:
  - Jeder Prozess erhält 1/n der CPU-Zeit mit höchstens q Zeiteinheiten.
  - Maximale Wartezeit: q (n-1)
  - für grosse q: FIFO
  - für kleine q: zu grosser Overhead
- präemptiv

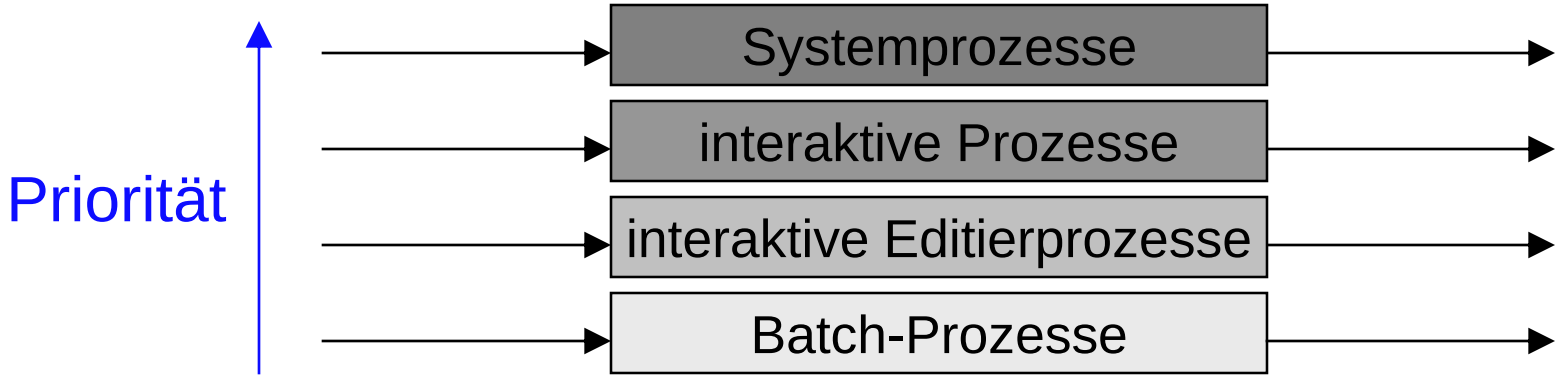
2.4.1 Beispiel: Round Robin

Prozess	Burst-Zeit
P_1	53
P_2	17
P_3	68
P_4	24



2.5 Multilevel Queue

- Ready-Queue wird in verschiedene einzelne Queues unterteilt.
- Jede einzelne Queue hat eigenen Scheduling-Algorithmus.
- Scheduling zwischen den einzelnen Queues
  - feste (absolute) Priorität
  - Zeitscheiben (z.B. 40:30:20:10)

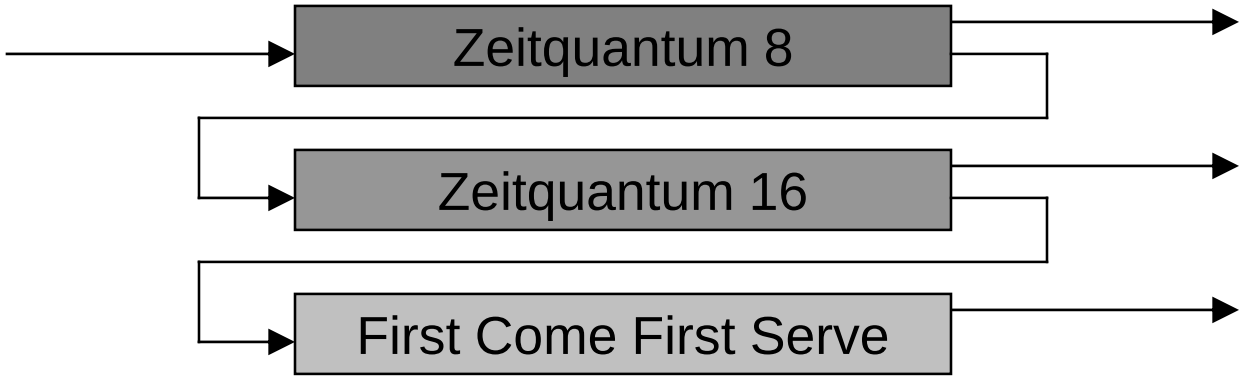


2.6 Multilevel Feedback Queue

- Migration zwischen verschiedenen Queues,
  z.B. wenn Prozess zu viel CPU-Zeit beansprucht

→ hohe Priorität für I/O-gebundene und interaktive Prozesse

- präemptiv



2.7 Lotterie-Scheduling

- Prozesse haben Lose.
- Betriebssystem führt Verlosung durch.
- Preis des „Gewinners“: CPU-Zeit
- Beispiel
  - 50 Verlosungen pro Sekunde
  - Preis = 20 ms CPU-Zeit
- Wichtige Prozesse können Extra-Lose erwerben.
- Austausch von Losen
  - Client-Prozess sendet Nachricht an Server, blockiert und übergibt Lose an Server-Prozess.
  - Server-Prozess verarbeitet Anfrage und gibt restliche Lose an Client zurück.

2.8 Garantiertes Scheduling

- Berechnung der vorgesehenen CPU-Zeit jedes Prozesses:
  (aktuelle Zeit – Erzeugungszeitpunkt) / n,
  n = # Prozesse
- Berechnung Verhältnis (verbrauchte CPU-Zeit / vorgesehene CPU-Zeit)
- Beispiel:
  - 0.5: Prozess hat nur halb soviel Zeit verbraucht wie geplant.
  - 2.0: Prozess hat doppelt soviel Zeit verbraucht wie geplant.
- Prozess mit dem geringsten Verhältnis wird solange ausgeführt bis ein anderer Prozess das geringste Verhältnis aufweist.

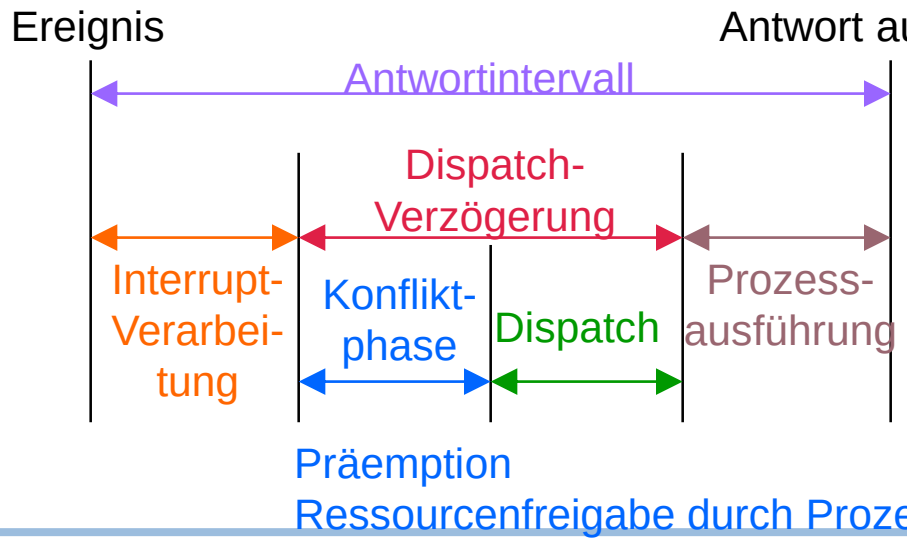
kann präemtiv und nicht präemntiv gemacht werden

3. Echtzeitsysteme

- Echtzeitsysteme müssen in begrenzter Zeit auf Ereignisse reagieren.
- Harte Echtzeitsysteme
  - Beenden der Prozesse in garantiertem Zeitintervall
  - Mechanismen
    - Zugangskontrolle (Admission Control), Ressourcenreservierung und Scheduling
    - kein Sekundärspeicher weil schwer zu beherschen und zu langsam
- Weiche Echtzeitsysteme
  - Versuch, Zeitüberschreitungen durch Prioritäten zu vermeiden

3.1 Implementierung von Echtzeitsystemen

- Prioritäts-Scheduling
  - hohe Priorität für Realzeitprozesse
- Minimierung von Verzögerungen
  - begrenzte Dispatch-Verzögerung
  - Präemptive Kerne: Präemption von Prozessen im Systemmodus, z.B. durch Preemption Points in längeren Systemaufrufen
  - Prozesse mit hoher Priorität warten auf Prozesse mit kleiner Priorität (Priority Inversion) → Vererben von Prioritäten



3.2 Echtzeit-Scheduling

- Ein System ist planbar, falls gilt: $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$

genug freie prozesszeit

  - m: Anzahl periodischer Ereignisse
  - Ereignis i tritt mit Periodendauer P_i auf und erfordert C_i CPU-Zeit.
- Beispiel
  - 3 periodische Ereignisse mit Periodendauern 100, 200 und 500 ms sowie CPU-Zeit pro Ereignis von 50, 30 und 100 ms.
  - 0.5 + 0.15 + 0.2 = 0.85 ≤ 1
  - Weiteres Ereignis mit Periodendauer von 1 s darf nicht mehr als 150 ms CPU-Zeit erfordern.

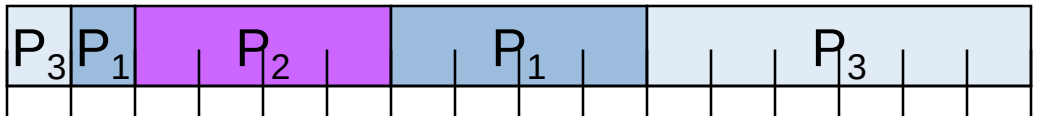
3.2.1 Offline-Scheduling

- (Statisches) Scheduling vor der eigentlichen Programmausführung zur Vermeidung von Scheduling-Overhead
- Vorberechnung eines vollständigen Ausführungsplans in Tabellenform
- Einfacher Tabellenzugriff während der Ausführung
- Voraussetzung: periodische Aktivitäten

3.2.2 Earliest Deadline First

- Prozesse mit Ausführungsfristen
- Prozess mit engster Frist wird selektiert.
- (nicht) präemptiv beides möglich

Prozess	Ankunftszeit	Ausführungszeit	Frist
P_1	1	5	10
P_2	2	4	7
P_3	0	7	17



3.2.3 Rate Monotonic Scheduling

- für periodische Systeme
- Statisches, präemptives Prioritäts-Scheduling
- Aktivitäten mit hoher Frequenz (= kleine Periode): hohe Priorität, z.B. P1
- Aktivitäten mit niedriger Frequenz (= grosse Periode): niedrige Priorität, z.B. P2

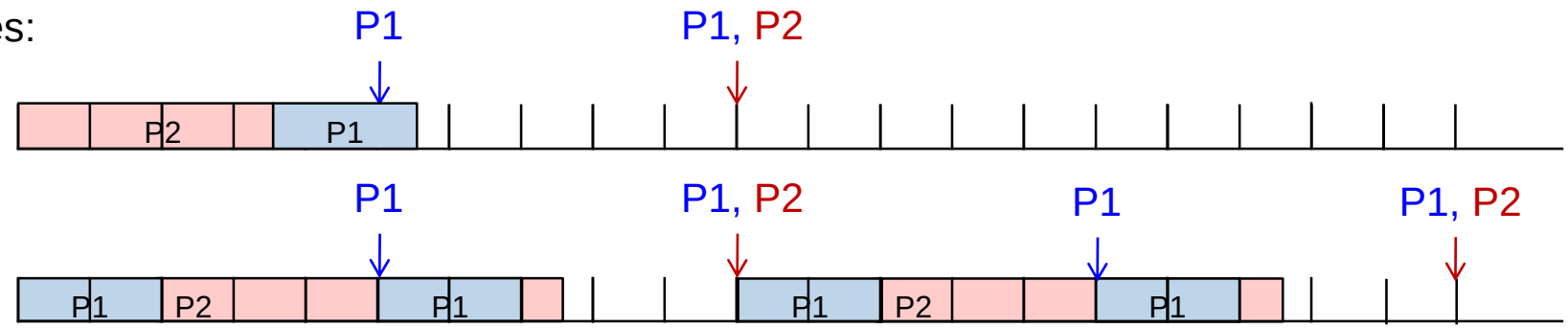
→ minimale Verzögerung von Aktivitäten mit hoher Frequenz

→ geringe Wahrscheinlichkeit für deren Fristverletzung

→ aber: Zerstückelung von Aktivitäten niedriger Frequenz wegen höherer Anzahl von Kontextwechseln zu Prozessen mit höherer Priorität

geht davon aus, dass prozesse mit hoher frequenz tiefe deadlines haben

Deadlines:

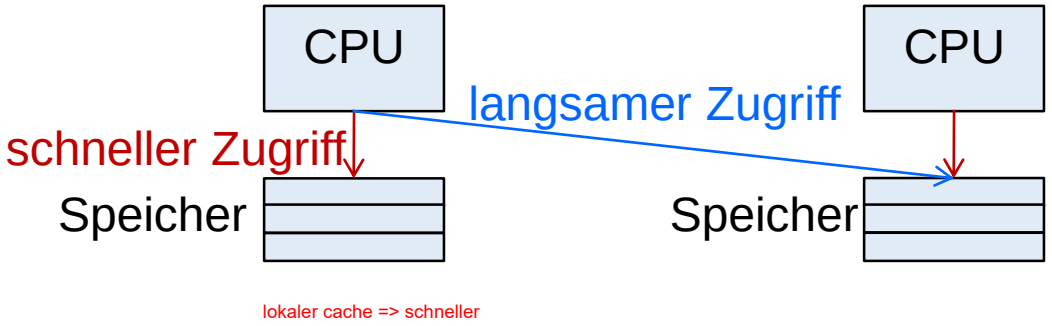


4.1 Asymmetrisches und Symmetrisches Multiprocessing

- Asymmetrisches Multiprocessing
  - Scheduling durch einen Prozessor (Master)
  - Andere Prozessoren führen Benutzercode aus.
- Symmetrisches Multiprocessing
  - Jeder Prozessor führt eigenes Scheduling durch.
  - Scheduling-Datenstrukturen, z.B. Ready-Queue
    - 1 pro System
      - automatische Lastverteilung
      - Zugriffskonflikte bei vielen CPUs
    - 1 pro CPU

4.2 Prozessor-Affinität

- Zuweisen der gleichen CPU für Threads
  → Ausnutzen von Verfügbarkeit der Daten in Caches und lokalem Speicher
- Weiche Affinität
  - System versucht, Prozess auf dem gleichen Prozessor zu belassen, gibt aber keine Garantie, z.B. Solaris: Processor Sets
- Harte Affinität
  - Prozess kann spezifizieren, dass er immer auf dem selben Prozessor ausgeführt werden will, z.B. Linux



4.3 Gruppen-Scheduling

- Zuweisen von Threads auf mehrere CPUs

→ Parallelität, Kooperation über gemeinsamen Speicher

- Gleichzeitige Zuweisung von Prozessoren erlaubt effiziente Interprozesskommunikation über gemeinsamen Speicher.
- Kennzeichnung kooperierender Threads durch Anwendung
- Thread-Gruppe kommt nur bei genügend vielen freien Prozessoren zur Ausführung, so dass alle Threads gleichzeitig auf einem eigenen Prozessor zur Ausführung kommen (Gang-Scheduling).
  Dadurch werden bei eng kooperierenden Threads Blockierungen reduziert, die Leistung erhöht und das Scheduling vereinfacht.
  (Scheduling der Thread-Gruppe statt einzelner Threads).
  Problematisch, falls nur grössere Thread-Gruppen existieren.

4.4 Lastausgleich

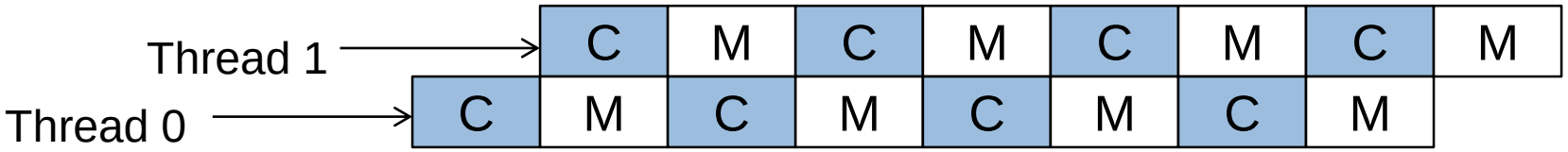
- Push Migration
  - Spezifischer Task prüft periodisch Last auf allen Prozessoren und verteilt Prozesse.
- Pull Migration
  - Untätige Prozessoren fordern Prozesse von ausgelasteten an.
- Oft: Kombination von Push und Pull Migration

4.5 Multithreading

- Grobgranular: Ausführung eines Threads bis zum Eintreten eines „Memory Stall“ (z.B. bei Cache Miss)

bei eintreten wechsel

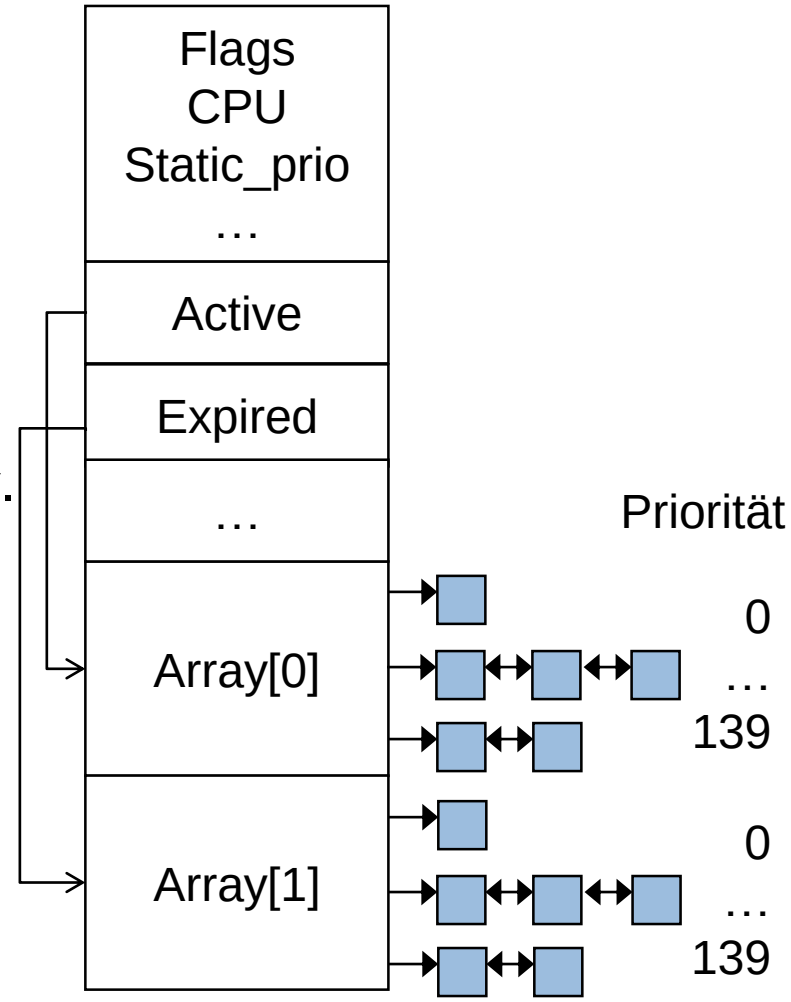
- Feingranular: Prozesse alternieren nach einzelner Instruktion (Logik für Thread-Wechsel), mehrere Hardware-Threads pro Core



C: Compute
M: Memory Stall

5.1.1 Beispiel: Linux Scheduling im 2.5 Kern

- Scheduling mit Kernel Threads
- Klassen
  - Realzeit-FIFO (ohne Zeitquantum) dürfen solange rechnen wie sie müssen
  - Realzeit-Round-Robin (mit Zeitquantum)
  - Timesharing (Standard, Priorität > 99)
- Modifikation der Priorität mit nice Kommando
- Runqueue-Datenstruktur für jede CPU
- Scheduler wählt nicht-leere Queue mit höchster Priorität.
- Threads kommen nach Ende des Zeitquantums in Expired Array.
- Vertauschen von Expired- und Active-Zeiger, falls keine Prozesse im aktiven Array existieren.
- Höhere Zeitquanten für Threads hoher Priorität
- Dynamische Neuberechnung der Priorität: Bonus (-5 … +5) für interaktive Threads
- Scheduler versucht in Multiprozessorsystemen Last auszugleichen und Threads auf früher benutzter CPU zuzuweisen.



5.1.2 Beispiel: Linux Scheduling im 2.6 Kern

- Scheduling-Klassen mit spezifischen Prioritäten und ggf. individuellen Algorithmen
- Auswahl des Prozesses mit der höchsten Priorität aus der Klasse mit der höchsten Priorität
- Standard-Linux: 2 Klassen
  - Completely Fair Scheduling (CFS)
  - Real-time: unterschiedliche Prioritäten für Realzeit- (0-99) und normale (100-139) Prozesse

5.1.3 Completely Fair Scheduler

- Zuweisen von CPU-Zeit-Anteilen für jeden Prozess basierend auf
  - nice-Wert (-20 – 19),
  - Ziel-Verzögerung (Zeit in der Prozess mindestens einmal ausgeführt werden sollte) und
  - Gesamtzahl der Prozesse
- Prozesse mit niedrigeren nice-Werten (default: 0) erhalten höhere Anteile.
- Steuerung über Variable **`vruntime`**
  (virtual run-time, zeichnet Laufzeit eines Prozesses auf)
  - **`vruntime`** ist mit Verfallsfaktor verbunden, welcher von der Priorität abhängt.
    - Prozesse mit hoher Priorität: **`vruntime`** < reale Laufzeit
    - Prozesse mit Default-Priorität: **`vruntime`** = reale Laufzeit
    - Prozesse mit niedriger Priorität: **`vruntime`** > reale Laufzeit
  - Auswahl des Prozesses mit geringstem **`vruntime`**-Wert

5.2 Scheduling von Threads

- 2 Scheduling-Ebenen: Prozesse und Threads
- Scheduling von User Threads innerhalb eines Prozesses (transparent für Betriebssystem, ggf. anwendungsspezifisch)
- Bei Kernel Threads kann auch zwischen Threads verschiedener Prozesse gewechselt werden.
- Problem bei Kernel Threads:
  aufwändige Wechsel zwischen Threads verschiedener Prozesse
- Aufwand für Kontextwechsel kann für Scheduling-Entscheidung berücksichtigt werden.