

# Systemprogrammierung

## Prozessverwaltung: Einplanungsverfahren

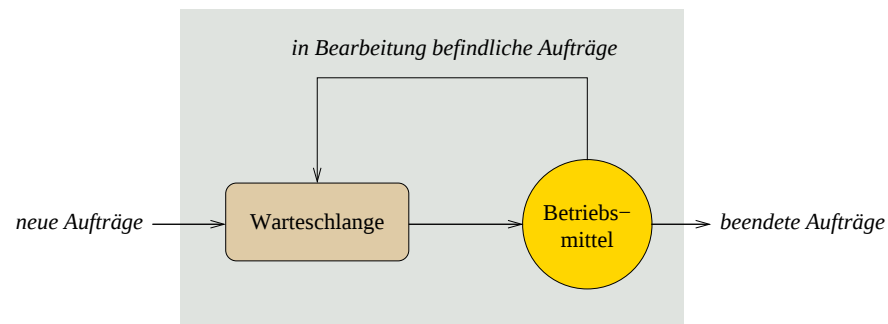
Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

30. April 2012

## Zur Erinnerung (SP1, VI Prozesse, S. 11)

Prinzipielle Funktionsweise von Einplanungsalgorithmen



Ein einzelner Einplanungsalgorithmus charakterisiert sich durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden. [4]

## Gliederung

- 1 Einordnung
  - Klassifikation
- 2 Verfahrensweisen
  - Kooperativ
  - Verdrängend
  - Mehrstufig
  - Priorisierend
  - Vergleich
- 3 Zusammenfassung
- 4 Anhang
  - Fallstudien

## Kooperativ vs. Präemptiv

Souverän ist die Anwendung oder das Betriebssystem

*cooperative scheduling* voneinander abhängiger Prozesse

- „unkooperative“ Prozesse können die CPU monopolisieren
- während der Programmausführung müssen Systemaufrufe erfolgen
  - Endlosschleifen ohne Systemaufrufe im Anwendungsprogramm verhindern Prozesse anderer Anwendungsprogramme
- alle Systemaufrufe müssen den Scheduler durchlaufen

*preemptive scheduling* voneinander unabhängiger Prozesse

- Prozessen wird die CPU entzogen, zugunsten anderer Prozesse
- der laufende Prozess wird ereignisbedingt von der CPU verdrängt
  - Endlosschleifen beeinträchtigen andere Prozesse nicht (bzw. kaum)
- die Ereignisbehandlung aktiviert (direkt/indirekt) den Scheduler
- Monopolisierung der CPU ist nicht möglich: **CPU-Schutz**

## Deterministisch vs. Probabilistisch

Mit oder ohne *à priori* Wissen

*deterministic scheduling* bekannter, exakt vorberechneter Prozesse

- alle CPU-Stoßlängen und ggf. auch **Termine** sind bekannt
  - bei (strikten) Echtzeitsystemen mindestens die Stoßlänge des „schlimmsten Falls“ (engl. *worst-case execution time*, WCET)
- die genaue Vorhersage der CPU-Auslastung ist möglich
- das System stellt die Einhaltung von **Zeitgarantien** sicher
- die Zeitgarantien gelten unabhängig von der jeweiligen Systemlast

*probabilistic scheduling* unbekannter Prozesse

- exakte CPU-Stoßlängen sind unbekannt, ggf. auch Termine
- die CPU-Auslastung kann lediglich abgeschätzt werden
- das System kann Zeitgarantien weder geben noch einhalten
- Zeitgarantien sind durch die Anwendung sicherzustellen

## Asymmetrisch vs. Symmetrisch

An eine CPU gebundene oder ungebundene Programmausführung

*asymmetric scheduling* ist abhängig von Eigenschaften der Ebene<sub>2/3</sub>

- obligatorisch in einem asymmetrischen Multiprozessorsystem
  - Rechnerarchitektur mit **programmierbare Spezialprozessoren**
  - z.B. Grafik- und/oder Kommunikationsprozessoren einerseits und ein Feld konventioneller (gleichartiger) CPUs andererseits
  - Prozesse sind an bestimmte Prozessoren gebunden
- optional in einem symmetrischen Multiprozessorsystem (s.u.)
  - das Betriebssystem hat freie Hand über die Prozessorvergabe
- Prozesse in funktionaler Hinsicht ungleich verteilen (müssen)

*symmetric scheduling* ist abhängig von Eigenschaften der Ebene<sub>2</sub>

- identische Prozessoren, alle geeignet zur Programmausführung
- Prozesse werden gleich auf die Prozessoren verteilt: **Lastausgleich**

## Statisch vs. Dynamisch

Entkoppelt von oder gekoppelt mit der Programmausführung

*offline scheduling* statisch, vor der Programmausführung

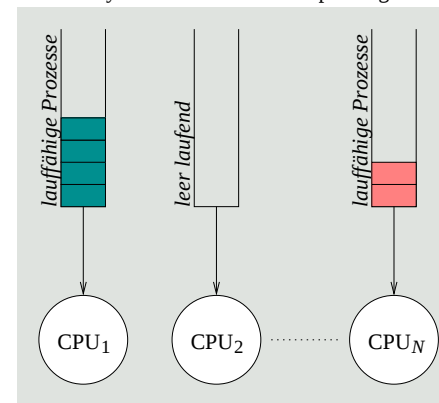
- Komplexität verbietet Ablaufplanung im laufenden Betrieb
  - zu berechnen, ob die Einhaltung aller Zeitvorgaben garantiert werden kann, ist ein NP-vollständiges Problem
  - die Berechnungskomplexität wird zum kritischen Faktor, wenn auf jede abfangbare katastrophale Situation zu reagieren ist
- Ergebnis der Vorbereitung ist ein **vollständiger Ablaufplan**
  - u.a. erstellt per Quelltextanalyse spezieller „Übersetzer“
  - oft zeitgesteuert abgearbeitet als Teil der Prozesseinlastung
- die Verfahren sind zumeist beschränkt auf **strikte Echtzeitsysteme**

*online scheduling* dynamisch, während der Programmausführung

- Stapelsysteme, interaktive Systeme, verteilte Systeme
- schwache und feste Echtzeitsysteme

## Asymmetrisch vs. Symmetrisch (Forts.)

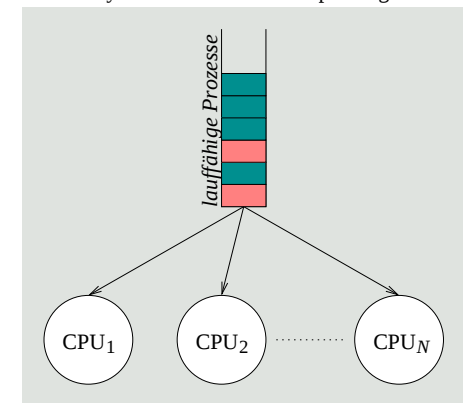
asymmetrische Prozesseinplanung



separate Bereitlisten

- lokale Bereitliste
- ggf. ungleichmäßige Auslastung

symmetrische Prozesseinplanung



gemeinsame Bereitliste

- globale Bereitliste
- ggf. gleichmäßige Auslastung

## Gliederung

- 1 Einordnung
  - Klassifikation
- 2 Verfahrensweisen
  - Kooperativ
  - Verdrängend
  - Mehrstufig
  - Priorisierend
  - Vergleich
- 3 Zusammenfassung
- 4 Anhang
  - Fallstudien

## FCFS (engl. *first come, first served*)

Fair, einfach zu implementieren (FIFO Queue), . . . , dennoch problematisch

Prozesse werden nach ihrer **Ankunftszeit** (engl. *arrival time*) eingeplant und in der sich daraus ergebenden Reihenfolge auch verarbeitet

- nicht-verdrängendes Verfahren, setzt kooperative Prozesse voraus

Gerechtigkeit zu Lasten hoher Antwortzeit und niedrigem E/A-Durchsatz

- suboptimal bei einem Mix von kurzen und langen CPU-Stößen

- Prozesse mit  $\left\{ \begin{array}{l} \text{langen} \\ \text{kurzen} \end{array} \right\}$  CPU-Stößen werden  $\left\{ \begin{array}{l} \text{begünstigt} \\ \text{benachteiligt} \end{array} \right\}$

### Problem: Konvoieffekt

- kurze Prozesse bzw. CPU-Stöße folgen einem langen. . .

## Klassische Einplanungs- bzw. Auswahlverfahren

- kooperativ FCFS gerecht
  - wer zuerst kommt, mahlt zuerst. . .
- verdrängend RR, VRR reihum
  - jeder gegen jeden. . .
- probabilistisch SPN (SJF), SRTF, HRRN priorisierend
  - die Kleinen nach vorne. . .
- mehrstufig MLQ, FB (MLFQ)
  - Rasterfahndung. . .

### Weiterführende Literatur

- *Theory of Scheduling* [2]
- *Operating System Theory* [1]
- *Queuing Systems* [3]
- *Real-Time Systems* [5]

## FCFS: Konvoieffekt

Durchlaufzeit kurzer Prozesse im Mix mit langen Prozessen

| Prozess | Zeiten  |       |       |      |       | $T_q/T_s$ |
|---------|---------|-------|-------|------|-------|-----------|
|         | Ankunft | $T_s$ | Start | Ende | $T_q$ |           |
| A       | 0       | 1     | 0     | 1    | 1     | 1.00      |
| B       | 1       | 100   | 1     | 101  | 100   | 1.00      |
| C       | 2       | 1     | 101   | 102  | 100   | 100.00    |
| D       | 3       | 100   | 102   | 202  | 199   | 1.99      |
| Ø       |         |       |       |      | 100   | 26.00     |

$T_s$  = Bedienzeit,  $T_q$  = Durchlaufzeit

normalisierte Durchlaufzeit ( $T_q/T_s$ ): vergleichsweise sehr schlecht bei C

- sie steht in einem extrem schlechten Verhältnis zur Bedienzeit  $T_s$
- typischer Effekt im Falle von kurzen Prozessen, die langen folgen

## RR (engl. *round robin*)

Verdrängendes FCFS, Zeitscheiben, CPU-Schutz

Prozesse werden nach ihrer **Ankunftszeit** ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant

- verdrängendes Verfahren, nutzt **periodische Unterbrechungen**
  - Zeitgeber (engl. *timer*) liefert asynchrone Programmunterbrechungen
- jeder Prozess erhält eine **Zeitscheibe** (engl. *time slice*) zugeteilt
  - obere Schranke für die CPU-Stoßlänge eines laufenden Prozesses

Verringerung der bei FCFS auftretenden Benachteiligung von Prozessen mit kurzen CPU-Stößen

- die **Zeitscheibenlänge** bestimmt die Effektivität des Verfahrens
  - zu lang, Degenierung zu FCFS; zu kurz, sehr hoher Mehraufwand
- Faustregel: etwas länger als die Dauer eines „typischen CPU-Stoßes“

## VRR (engl. *virtual round robin*)

RR mit Vorzugswarteschlange und variablen Zeitscheiben

Prozesse werden mit Beendigung ihres E/A-Stoßes **bevorzugt eingeplant**, jedoch nicht (zwingend) bevorzugt/sofort eingelastet

- Einreihung in eine der Bereitliste vorgeschalteten **Vorzugsliste**
  - FIFO  $\leadsto$  evtl. Benachteiligung hoch-interaktiver Prozesse; daher...
  - aufsteigend sortiert nach dem **Zeitscheibenrest** eines Prozesses
- **Umplanung** bei Beendigung des jeweils laufenden CPU-Stoßes
  - die Prozesse auf der Vorzugsliste werden zuerst eingelastet
  - sie bekommen die CPU für die Restdauer ihrer Zeitscheibe zugeteilt
  - bei Ablauf dieser Zeitscheibe werden sie in die Bereitliste eingereiht

### Vermeidung der bei RR möglichen Ungleichverteilung von CPU-Zeiten

- bevorzugt werden interaktive Prozesse mit kurzen CPU-Stößen
- erreicht durch strukturelle Maßnahmen — nicht durch analytische

## RR: Konvoieffekt

Leistungsprobleme bei einem Mix von Prozessen

**E/A-intensive Prozesse** schöpfen ihre Zeitscheibe selten voll aus

- sie beenden ihren CPU-Stoß freiwillig
  - vor Ablauf der Zeitscheibe

**CPU-intensive Prozesse** schöpfen ihre Zeitscheibe meist voll aus

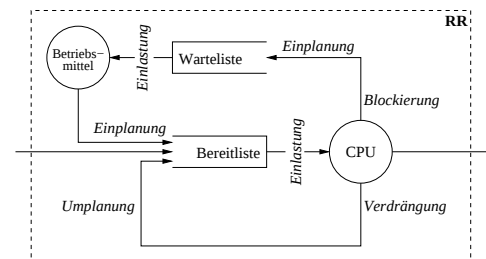
- sie beenden ihren CPU-Stoß unfreiwillig
  - durch Verdrängung

**Problem: kurze CPU-Stöße folgen einem langen...**

CPU-Zeit ist zu Gunsten CPU-intensiver Prozesse ungleich verteilt

- E/A-intensive Prozesse werden schlechter bedient
- E/A-Geräte sind schlecht ausgelastet
- Varianz der Antwortzeit E/A-intensiver Prozesse ist groß

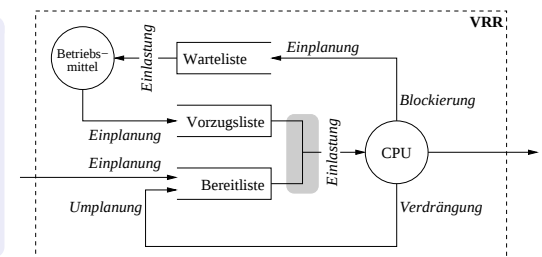
## RR vs. VRR



- Bereitliste**
- lauffähiger Fäden<sup>a</sup>
  - dreiseitig bestückt
    - 2 × Einplanung
    - 1 × Umplanung
  - **unbedingt** bedient
- Warteliste**
- blockierter Fäden

<sup>a</sup>CPU „Warteliste“

- Bereitliste**
- wie bei RR
  - 2-seitig bestückt
    - 1 × Einplanung
    - 1 × Umplanung
  - **bedingt** bedient
- Warteliste**
- wie bei RR
- Vorzugsliste**
- **unbedingt** bedient



## SPN (engl. *shortest process next*)

Zeitreihen bilden, analysieren und verwerten

Prozesse werden nach ihrer **erwarteten Bedienzeit** eingeplant

- Grundlage dafür ist *a priori* Wissen über die **Prozesslaufzeiten**:
  - Stapelbetrieb** Programmierer setzen **Frist** (engl. *time limit*)
  - Produktionsbetrieb** Erstellung einer **Statistik** durch Probeläufe
  - Dialogbetrieb** **Abschätzung** von CPU-Stoßlängen zur Laufzeit
- Abarbeitung einer aufsteigend nach Laufzeiten sortierten Bereitsliste
  - Abschätzung erfolgt vor (statisch) oder zur (dynamisch) Laufzeit

Verkürzung von Antwortzeiten und Steigerung der Gesamtleistung des Systems auf Kosten länger laufender Prozess

- ein **Verhungern** (engl. *starvation*) dieser Prozesse ist möglich

## SPN: DämpfungsfILTER (engl. *decay filter*)

Wichtung der CPU-Stöße

**Dämpfung** (engl. *decay*) der am weitesten zurückliegenden CPU-Stöße:

$$S_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot S_n$$

- für den konstanten Wichtungsfaktor  $\alpha$  gilt dabei:  $0 < \alpha < 1$
- drückt die **relative Wichtung** einzelner CPU-Stöße der Zeitreihe aus
- teilweise Expansion der Gleichung führt zu:
  - $S_{n+1} = \alpha T_n + (1 - \alpha)\alpha T_{n-1} + \dots + (1 - \alpha)^i \alpha T_{n-i} + \dots + (1 - \alpha)^n S_1$
- Beispiel der Entwicklung für  $\alpha = 0.8$ :
  - $S_{n+1} = 0.8 T_n + 0.16 T_{n-1} + 0.032 T_{n-2} + 0.0064 T_{n-3} + \dots$

## SPN: Abschätzung der Dauer eines CPU-Stoßes

**Mittelwertbildung** über alle CPU-Stoßlängen eines Prozesses:

$$S_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^n T_i = \frac{1}{n} \cdot T_n + \frac{n-1}{n} \cdot S_n$$

- Problem dieser Berechnung ist die **gleiche Wichtung** aller CPU-Stöße
- jüngere CPU-Stöße eine größere Wichtung geben: **Lokalität**

**Messung** der Dauer eines CPU-Stoßes geschieht bei Prozesseinlastung:

- Stoppzeit  $T_2$  von  $P_x$  entspricht (in etwa) der Startzeit  $T_1$  von  $P_y$ 
  - gemessen in **Uhrzeit** (engl. *clock time*) oder **Uhrtick** (engl. *clock tick*)
- Akkumulation der Differenzen  $T_2 - T_1$  für jeden Prozess  $P_i$

## SRTF (engl. *shortest remaining time first*)

Verdrängendes SPN, Verhungersgefahr, Effektivität von VRR

Prozesse werden nach ihrer **erwarteten Bedienzeit** eingeplant und in unregelmäßigen Zeitabständen **sporadisch** umgeplant

- sei  $T_{et}$  die erwartete CPU-Stoßlänge eines eintreffenden Prozesses
- sei  $T_{rt}$  die verbleibende CPU-Stoßlänge des laufenden Prozesses
- der laufende Prozess wird verdrängt, wenn gilt:  $T_{et} < T_{rt}$

**Umplanung** erfolgt ereignisbedingt und (ggf. voll) verdrängend

- z.B. bei Beendigung des E/A-Stoßes eines wartenden Prozesses
- allgemein: bei Aufhebung der Wartebedingung für einen Prozess

**Verdrängung führt zu besseren Antwort- und Durchlaufzeiten:**

- gegenüber VRR steht der **Overhead** zur CPU-Stoßlängenabschätzung

## HRRN (engl. *highest response ratio next*)

SRTF ohne Verhungern der Prozesse

Prozesse werden nach ihrer **erwarteten Bedienzeit** eingeplant und periodisch unter Berücksichtigung ihrer **Wartezeit** umgeplant

- in regelmäßigen Zeitabständen wird ein Verhältniswert  $R$  berechnet:

$$R = \frac{w + s}{s}$$

$w$  aktuell abgelaufene Wartezeit eines Prozesses

$s$  erwartete (d.h., abgeschätzte) Bedienzeit eines Prozesses

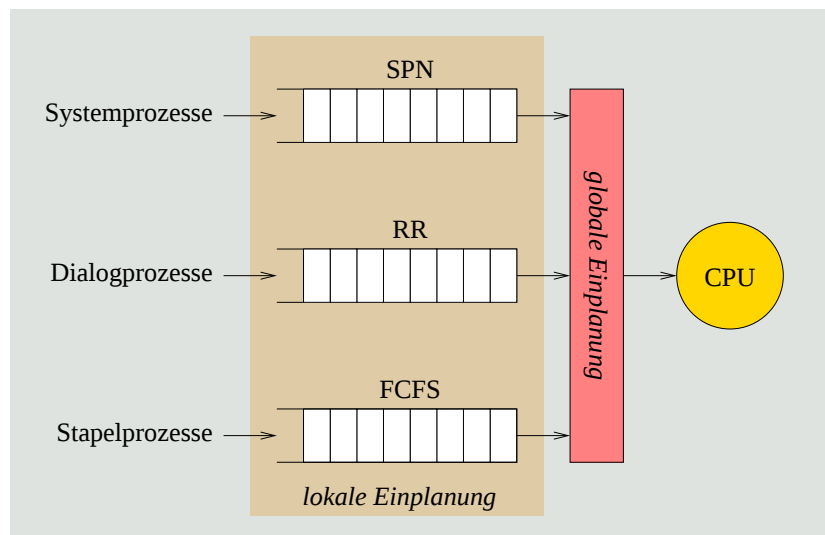
- periodische Aktualisierung aller Einträge in der Bereitliste
- ausgewählt wird der Prozess mit dem größten Verhältniswert  $R$

**Alterung** (engl. *aging*) von Prozessen meint Anstieg der Wartezeit

- Alterung entgegenwirken (engl. *anti-aging*) beugt Verhungern vor

## MLQ: Mischbetrieb

System-, Dialog- und Stapelprozesse im Mix



## MLQ (engl. *multilevel queue*)

Unterstützt Mischbetrieb: Vorder- und Hintergrundbetrieb

Prozesse werden nach ihrem **Typ** (d.h., nach den für sie zutreffend geglaubten Eigenschaften) eingeplant

- Aufteilung der Bereitliste in separate („getypte“) Listen
  - z.B. für System-, Dialog- und Stapelprozesse
- mit jeder Liste eine **lokale Einplanungsstrategie** verbinden
  - z.B. SPN, RR und FCFS
- zwischen den Listen eine **globale Einplanungsstrategie** definieren
  - statisch** Liste einer bestimmten Prioritätsebene fest zuordnen
    - Verhungersgefahr für Prozesse tiefer liegender Listen
  - dynamisch** die Listen im Zeitmultiplexverfahren wechseln
    - z.B. 40 % System-, 40 % Dialog-, 20 % Stapelprozesse

Prozessen Typen zuordnen ist eine statische Entscheidung

- sie wird zum Zeitpunkt der Prozesserschöpfung getroffen

## FB (engl. *feedback*)

Begünstigt kurze/interaktive Prozesse, ohne die relativen Stoßlängen kennen zu müssen

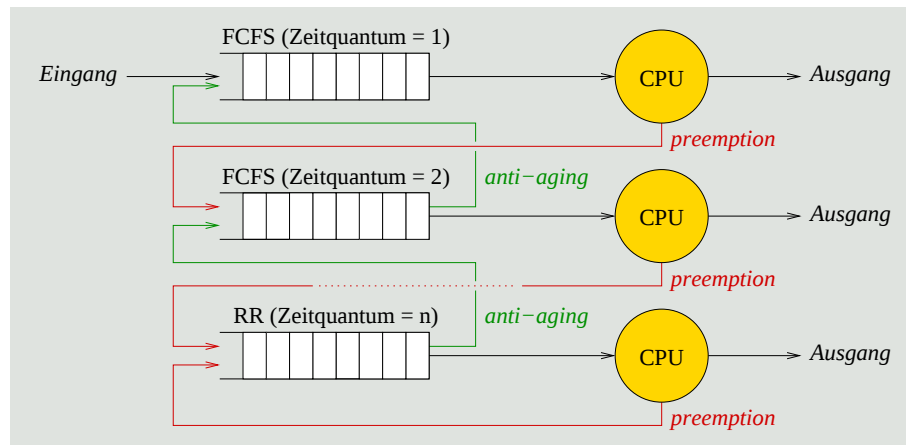
Prozesse werden nach ihrer **Ankunftszeit** ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant

- Hierarchie von Bereitlisten, je nach Anzahl der **Prioritätsebenen**
  - erstmalig eintreffende Prozesse steigen oben ein
  - Zeitscheibenablauf drückt den laufenden Prozess weiter nach unten
- je nach Ebene verschiedene Einreihungsstrategien und -parameter
  - unterste Ebene arbeitet nach RR, alle anderen (höheren) nach FCFS
  - die Zeitscheibengrößen nehmen von oben nach unten zu

**Bestrafung** (engl. *penalization*)

- Prozesse mit langen CPU-Stößen fallen nach unten durch
  - ggf. wird der Alterung entgegengewirkt: Prozesse wieder anheben
- Prozesse mit kurzen CPU-Stößen laufen relativ schnell durch

## FB: Bestrafung und Bewährung



multilevel feedback queue (MLFQ)

## Gegenüberstellung von Strategien und Verfahrensweisen

kooperativ/verdrängend vs. probabilistisch/deterministisch

|                 | FCFS                                                    | RR | VRR | SPN | SRTF | HRRN | FB |
|-----------------|---------------------------------------------------------|----|-----|-----|------|------|----|
| kooperativ      | ✓                                                       |    |     | ✓   |      |      |    |
| verdrängend     |                                                         | ✓  | ✓   |     | ✓    | ✓    | ✓  |
| probabilistisch |                                                         |    |     | ✓   | ✓    | ✓    |    |
| deterministisch | keine bzw. nicht von sich aus allein $\leadsto$ EZS [5] |    |     |     |      |      |    |

MLQ umfasst Eigenschaften der in dem Verfahren vereinten Strategien

- Priorisierung von Strategien liefert Nuancen im Laufzeitverhalten
- speziellen Anwendungsanforderungen (teilweise) entgegenkommen:
  - z.B. FCFS priorisieren  $\leadsto$  „number crunching“ fördern

## Prioritäten setzende Verfahren

Statische Prioritäten (MLQ) vs. dynamische Prioritäten (VRR, SPN, SRTF, HRRN, FB)

**Prozessvorrang** bedeutet die bevorzugte Einlastung von Prozessen mit höherer Priorität und wird auf zwei Arten bestimmt:

**statisch** zum Zeitpunkt der **Prozesserzeugung**  $\leadsto$  Laufzeitkonstante

- wird im weiteren Verlauf nicht mehr verändert
- erzwingt eine deterministische Ordnung zw. Prozessen

**dynamisch** zum Zeitpunkt der **Prozessausführung**  $\leadsto$  Laufzeitvariable

- die Berechnung erfolgt durch das Betriebssystem
  - ggf. in Kooperation mit den Anwendungsprogrammen
- erzwingt keine deterministische Ordnung zw. Prozessen

### Echtzeitverarbeitung bedingt Prioritäten setzende Verfahren

- jedoch nicht jedes solcher Verfahren eignet sich zum Echtzeitbetrieb
- Einplanung muss ein **deterministisches Laufzeitverhalten** liefern
  - entsprechend der jeweiligen Anforderungen der Anwendungsdomäne

## Gliederung

- 1 Einordnung
  - Klassifikation
- 2 Verfahrensweisen
  - Kooperativ
  - Verdrängend
  - Mehrstufig
  - Priorisierend
  - Vergleich
- 3 Zusammenfassung
- 4 Anhang
  - Fallstudien

## Resümee

- Prozesseinplanung unterliegt einer breit gefächerten **Einordnung**
  - kooperativ/verdrängend
  - deterministisch/probabilistisch
  - entkoppelt/gekoppelt
  - asymmetrisch/symmetrisch
- die entsprechenden **Verfahrensweisen** sind z.T. sehr unterschiedlich
  - FCFS: kooperativ
  - RR, VRR: verdrängend
  - SPN, SRTF, HRRN: probabilistisch
  - MLQ, FB (MLFQ): mehrstufig
- Prioritäten setzende Verfahren legen einen **Prozessvorrang** fest
  - dies betrifft die behandelten probabilistischen, mehrstufigen Verfahren
    - die allesamt nichtdeterministisch und damit nicht echtzeitfähig sind
  - echtzeitfähige Prozesseinplanung ist vor allem deterministisch
- die Fallstudien (s. Anhang) planen Prozesse probabilistisch ein...

## Gliederung

- 1 **Einordnung**
  - Klassifikation
- 2 **Verfahrensweisen**
  - Kooperativ
  - Verdrängend
  - Mehrstufig
  - Priorisierend
  - Vergleich
- 3 **Zusammenfassung**
- 4 **Anhang**
  - Fallstudien

## Literaturverzeichnis

- [1] COFFMAN, E. G. ; DENNING, P. J.:  
*Operating System Theory*.  
Prentice Hall, Inc., 1973
- [2] CONWAY, R. W. ; MAXWELL, L. W. ; MILLNER, L. W.:  
*Theory of Scheduling*.  
Addison-Wesley, 1967
- [3] KLEINROCK, L. :  
*Queuing Systems*. Bd. I: Theory.  
John Wiley & Sons, 1975
- [4] LISTER, A. M. ; EAGER, R. D.:  
*Fundamentals of Operating Systems*.  
The Macmillan Press Ltd., 1993. –  
ISBN 0-333-59848-2
- [5] LIU, J. W. S.:  
*Real-Time Systems*.  
Prentice-Hall, Inc., 2000. –  
ISBN 0-13-099651-3

## UNIX klassisch

Zweistufiges Verfahren, Antwortzeiten minimierend, Interaktivität fördernd

**low-level** kurzfristig; präemptiv, MLFQ, **dynamische Prozessprioritäten**

- einmal pro Sekunde:  $prio = cpu\_usage + p\_nice + base$
- CPU-Nutzungsrecht mit jedem „Tick“ (1/10 s) verringert
  - Prioritätswert kontinuierlich um „Tickstand“ erhöhen
  - je höher der Wert, desto niedriger die Priorität
- über die Zeit gedämpftes CPU-Nutzungsmaß:  $cpu\_usage$ 
  - der Dämpfungsfaktor variiert von UNIX zu UNIX

**high-level** mittelfristig; mit **Umlagerung** arbeitend

Prozesse können relativ zügig den Betriebssystemkern verlassen

- gesteuert über die beim Schlafenlegen einstellbare **Aufweckpriorität**

## UNIX 4.3 BSD

MLFQ (32 Warteschlangen, RR), dynamische Prioritäten (0–127)

**Berechnung** der **Benutzerpriorität** bei jedem vierten Tick (40 ms)

- $p_{usrpri} = PUSER + \left\lfloor \frac{p\_cpu}{4} \right\rfloor + 2 \cdot p\_nice$ 
  - mit  $p\_cpu = p\_cpu + 1$  bei jedem Tick (10 ms)
  - **Gewichtungsfaktor**  $-20 \leq p\_nice \leq 20$  (vgl. nice(2))
- Prozess mit Priorität  $P$  kommt in Warteschlange  $P/4$

**Glättung** des Wertes der **Prozessornutzung** ( $p\_cpu$ ) jede Sekunde

- $p\_cpu = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p\_cpu + p\_nice$
- **Sonderfall:** Prozesse schließen länger als eine Sekunde
  - $p\_cpu = \left[ \frac{2 \cdot load}{2 \cdot load + 1} \right]^{p\_slptime} \cdot p\_cpu$

## UNIX Solaris

MLQ (4 Klassen) und MLFQ (60 Ebenen, Tabellensteuerung)

| quantum | tqexp | slpret | maxwait | lwait | Ebene |
|---------|-------|--------|---------|-------|-------|
| 200     | 0     | 50     | 0       | 50    | 0     |
| 200     | 0     | 50     | 0       | 50    | 1     |
| ...     |       |        |         |       |       |
| 40      | 34    | 55     | 0       | 55    | 44    |
| 40      | 35    | 56     | 0       | 56    | 45    |
| 40      | 36    | 57     | 0       | 57    | 46    |
| 40      | 37    | 58     | 0       | 58    | 47    |
| 40      | 38    | 58     | 0       | 58    | 48    |
| 40      | 39    | 58     | 0       | 59    | 49    |
| 40      | 40    | 58     | 0       | 59    | 50    |
| 40      | 41    | 58     | 0       | 59    | 51    |
| 40      | 42    | 58     | 0       | 59    | 52    |
| 40      | 43    | 58     | 0       | 59    | 53    |
| 40      | 44    | 58     | 0       | 59    | 54    |
| 40      | 45    | 58     | 0       | 59    | 55    |
| 40      | 46    | 58     | 0       | 59    | 56    |
| 40      | 47    | 58     | 0       | 59    | 57    |
| 40      | 48    | 58     | 0       | 59    | 58    |
| 20      | 49    | 59     | 32000   | 59    | 59    |

/usr/sbin/dispatch -c TS -g

| MLQ (Klasse)    | Priorität |
|-----------------|-----------|
| time-sharing TS | 0–59      |
| interactive IA  | 0–59      |
| system SYS      | 60–99     |
| real time RT    | 100–109   |

MLFQ in Klasse TS bzw. IA:

|         |                          |
|---------|--------------------------|
| quantum | Zeitscheibe (ms)         |
| tqexp   | Ebene bei Bestrafung     |
| slprt   | Ebene nach Deblockierung |
| maxwait | ohne Bedienung (s)       |
| lwait   | Ebene bei Bewährung      |

**Besonderheit:** *dispatch table* (TS, IA) kapselt alle Entscheidungen

- kunden-/problemspezifische Lösungen durch verschiedene Tabellen

## UNIX 4.3 BSD: Glättung durch DämpfungsfILTER

**Annahme 1:**  $\emptyset$  Auslastung ( $load$ ) sei  $1 \rightsquigarrow p\_cpu = 0.66 \cdot p\_cpu + p\_nice$

**Annahme 2:** Prozess sammelt  $T_i$  Ticks im Zeitintervall  $i$  an,  $p\_nice = 0$ :

$$\begin{aligned}
 p\_cpu &= 0.66 \cdot T_0 \\
 &= 0.66 \cdot (T_1 + 0.66 \cdot T_0) = 0.66 \cdot T_1 + 0.44 \cdot T_0 \\
 &= 0.66 \cdot T_2 + 0.44 \cdot T_1 + 0.30 \cdot T_0 \\
 &= 0.66 \cdot T_3 + \dots + 0.20 \cdot T_0 \\
 &= 0.66 \cdot T_4 + \dots + 0.13 \cdot T_0
 \end{aligned}$$

- nach fünf Sekunden gehen nur noch etwa 13 % der „Altlast“ ein

## UNIX Solaris: Bestrafung vs. Bewährung nach Verdrängung

Beispiel:

- $1 \times$  CPU-Stoß à 1000 ms
- $5 \times$  E/A-Stoß → CPU-Stoß à 1 ms

| #  | Ebene | CPU-Stoß | Ereignis    |
|----|-------|----------|-------------|
| 1  | 59    | 20       | Zeitscheibe |
| 2  | 49    | 40       | Zeitscheibe |
| 3  | 39    | 80       | Zeitscheibe |
| 4  | 29    | 120      | Zeitscheibe |
| 5  | 19    | 160      | Zeitscheibe |
| 6  | 9     | 200      | Zeitscheibe |
| 7  | 0     | 200      | Zeitscheibe |
| 8  | 0     | 180      | E/A-Stoß    |
| 9  | 50    | 1        | E/A-Stoß    |
| 10 | 58    | 1        | E/A-Stoß    |
| 11 | 58    | 1        | E/A-Stoß    |
| 12 | 58    | 1        | E/A-Stoß    |

Variante: nach 640 ms...

- der Prozess wird verdrängt und muss auf die erneute Einlastung warten
- der Alterung des wartenden Prozesses wird durch Anhebung seiner Priorität entgegengewirkt (*anti-aging*)
- die höhere Ebene erreicht, steigt der Prozess im weiteren Verlauf wieder ab

|    |    |     |                   |
|----|----|-----|-------------------|
| 7  | 0  | 20  | <i>anti-aging</i> |
| 8  | 50 | 40  | Zeitscheibe       |
| 9  | 40 | 40  | Zeitscheibe       |
| 10 | 30 | 80  | Zeitscheibe       |
| 11 | 20 | 120 | Zeitscheibe       |
| 12 | 10 | 80  | E/A-Stoß          |
| 13 | 50 | 1   | E/A-Stoß          |

## Linux 2.4

### Epochen und Zeitquanten

Prozessen zugewiesene Prozessorzeit ist in **Epochen** unterteilt:

**beginnen** alle lauffähige Prozess haben ihr Zeitquantum erhalten

**enden** alle lauffähigen Prozesse haben ihr Zeitquantum verbraucht

**Zeitquanten** (Zeitscheiben) variieren mit den Prozessen und Epochen:

- jeder Prozess besitzt eine einstellbare **Zeitquantumbasis** (`nice(2)`)
  - 20 Ticks  $\approx$  210 ms
  - das Zeitquantum eines Prozesses nimmt periodisch (Tick) ab
- beide Werte addiert liefert die **dynamische Priorität** eines Prozesses
  - dynamische Anpassung:  $quantum = quantum/2 + (20 - nice)/4 + 1$

**Echtzeitprozesse** (schwache EZ) besitzen **statische Prioritäten**: 1–99

## Linux 2.5

### Deterministische Prozesseinplanung: $O(1)$

Einplanung von Prozessen hat **konstante Berechnungskomplexität**:

**Prioritätsfelder** zwei Tabellen pro CPU: *active*, *expired*

**Prioritätsebenen** 140 Ebenen pro Tabelle

- 1–100 für Echtzeit-, 101–140 für sonstige Prozesse
- pro Ebene eine (doppelt verkettete) Bereitliste

**Prioritäten** gewöhnlicher Prozesse skalieren je nach **Interaktivitätsgrad**

- **Bonus** (–5) für interaktive Prozesse, **Strafe** (+5) für rechenintensive
- berechnet am Zeitscheibenende:  $prio = MAX\_RT\_PRIO + nice + 20$

Ablauf des Zeitquantums befördert aktiven Prozess ins „*expired*“-Feld

- zum Epochwechsel werden die Tabellen ausgetauscht
  - `void *aux = active; active = expired; expired = aux;`

## Linux 2.4 (Forts.)

### Einplanungsklassen und Gütefunktion

Prozesseinplanung unterscheidet zwischen drei **Scheduling-Klassen**:

|              |                                                  |                    |
|--------------|--------------------------------------------------|--------------------|
| <b>FIFO</b>  | verdrängbare, kooperative Echtzeitprozesse       | } eine Bereitliste |
| <b>RR</b>    | Echtzeitprozesse derselben Priorität             |                    |
| <b>other</b> | konventionelle („ <i>time-shared</i> “) Prozesse |                    |

Prozessauswahl greift auf eine **Gütefunktion** zurück:  $O(n)$

|                |                                                   |    |
|----------------|---------------------------------------------------|----|
| $v = -1000$    | der Prozess ist <i>Init</i>                       | –  |
| $v = 0$        | der Prozess hat sein Zeitquantum verbraucht       | –  |
| $0 < v < 1000$ | der Prozess hat sein Zeitquantum nicht verbraucht | +  |
| $v \geq 1000$  | der Prozess ist ein Echtzeitprozess               | ++ |

Prozesse können bei der Auswahl einen **Bonus** („*boost*“) erhalten

- sofern sie sich mit dem Vorgänger den Adressraum teilen