

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – IX.2 Prozessverwaltung: Einplanungsverfahren

Wolfgang Schröder-Preikschat

8. November 2022



Agenda

Einführung

Einordnung
Klassifikation

Verfahrensweisen
Kooperativ
Verdrängend
Probabilistisch
Mehrstufig

Zusammenfassung



© wosch SP (WS 2022/23, C – IX.2) 1. Einführung

IX.2/2

Gliederung

Einführung

Einordnung
Klassifikation

Verfahrensweisen
Kooperativ
Verdrängend
Probabilistisch
Mehrstufig

Zusammenfassung



© wosch SP (WS 2022/23, C – IX.2) 1. Einführung

IX.2/3

Lehrstoff

- gängige Klassen der **Ein-/Umplanung** von Prozessen kennenlernen und in ihrer Bedeutung einschätzen können
 - jedes Verfahren einer Klasse hat bestimmte **Gütemerkmale** im Fokus
 - bei mehreren Merkmalen müsste ein **Kompromiss** gefunden werden
 - scheidet Konfliktlösung aus, ist eine geeignete **Priorisierung** vorzunehmen
- die Verfahren auf **nicht-funktionale Eigenschaften** untersuchen, so Gemeinsamkeiten und Unterschiede erfassen
 - Gerechtigkeit, minimale Antwort- oder Durchlaufzeit
 - maximaler Durchsatz, maximale Auslastung
 - Termineinhaltung, Dringlichkeiten genügend, Vorhersagbarkeit
- erkennen, dass es die „ **Eierlegende Wollmilchsau**“ auch in einer virtuellen Welt nicht geben kann. . .

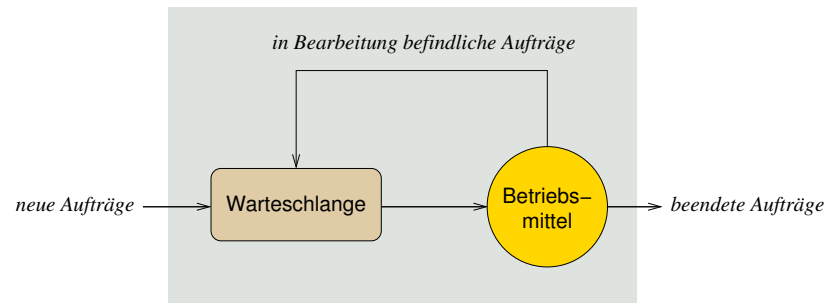
Kein einziges Verfahren zur Ein-/Umplanung von Prozessen hat nur Vorteile, befriedigt alle Bedürfnisse, genügt allen Ansprüchen.



© wosch SP (WS 2022/23, C – IX.2) 1. Einführung

IX.2/4

Verwaltung von (betriebsmittelgebundenen) **Warteschlangen**



Ein einzelner Einplanungsalgorithmus ist charakterisiert durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden. [12]



- die Charakterisierung von **Einplanungsalgorithmen** macht glauben, Betriebssysteme fokussiert „mathematisch“ studieren zu müssen
 - R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
 - E. G. Coffman, P. J. Denning. *Operating System Theory*.
 - L. Kleinrock. *Queuing Systems, Volume I: Theory*.
- praktische Umsetzung offenbart jedoch einen **Querschnittsbelang** (*cross-cutting concern*), der sich kaum modularisieren lässt
 - spezifische Betriebsmittelmerkmale stehen ggf. Bedürfnissen der Prozesse, die Aufträge zur Betriebsmittelnutzung abgesetzt haben, gegenüber
 - dabei ist die Prozessreihenfolge in Warteschlangen (bereit, blockiert) ein Aspekt, die Auftragsreihenfolge dagegen ein anderer Aspekt
 - Interferenz**¹ bei der Durchsetzung der Strategien kann die Folge sein
- Einplanungsverfahren stehen und fallen mit den Vorgaben, die für die jeweilige **Zieldomäne** zu treffen sind
 - die „Eier-legende Wollmilchsau“ kann es nicht geben
 - Kompromisslösungen sind geläufig — aber nicht in allen Fällen tragfähig

¹lat. *inter* zwischen und *ferire* von altfrz. *s'entreferir* sich gegenseitig schlagen.



Gliederung

Einführung

Einordnung
Klassifikation

Verfahrensweisen
Kooperativ
Verdrängend
Probabilistisch
Mehrstufig

Zusammenfassung



Kooperativ vs. Präemptiv

... bereit zur Zusammenarbeit

- Souverän** ist die Anwendung oder das Betriebssystem verhält sich Entwicklungen gegenüber zuvorkommend, vorsorglich, vorbeugend
 - kooperative Planung** (*cooperative scheduling*)
 - Ein-/Umplanung voneinander abhängiger Prozesse
 - Prozessen wird die CPU nicht zugunsten anderer Prozesse entzogen
 - der laufende Prozess gibt die CPU nur mittels Systemaufruf ab
 - die Systemaufrufbehandlung aktiviert (direkt/indirekt) den Scheduler
 - systemaufruffreie Endlosschleifen beeinträchtigen andere Prozesse
 - CPU-Monopolisierung** ist möglich: *run to completion*
 - präemptive Planung** (*preemptive scheduling*)
 - Ein-/Umplanung voneinander unabhängiger Prozesse
 - Prozessen kann die CPU entzogen werden, zugunsten anderer Prozesse
 - der laufende Prozess wird **ereignisbedingt** von der CPU **verdrängt**
 - die Ereignisbehandlung aktiviert (direkt/indirekt) den Scheduler
 - Endlosschleifen beeinträchtigen andere Prozesse nicht (bzw. kaum)
 - Monopolisierung der CPU ist nicht möglich: **CPU-Schutz**
- Synergie**: auf Maschinenprogrammenebene kooperative *user threads*, auf Betriebssystemebene präemptive *kernel threads* [8, vgl. S. 27]. ...



Deterministisch vs. Probabilistisch

- alle Abläufe durch **à priori Wissen** eindeutig festlegen könnend oder die **Wahrscheinlichkeit** berücksichtigend
 - deterministische Planung** (*deterministic scheduling*)
 - alle Prozesse (Rechenstoßlängen)² und ggf. auch **Termine** sind bekannt
 - die genaue Vorhersage der CPU-Auslastung ist möglich
 - das System stellt die Einhaltung von **Zeitgarantien** sicher
 - die Zeitgarantien gelten unabhängig von der jeweiligen Systemlast
 - probabilistische Planung** (*probabilistic scheduling*)
 - Prozesse (exakte Rechenstoßlängen) 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
- dabei fällt die Abgrenzung nicht immer so scharf aus: wahrscheinliche Abläufe vorherzusagen, ist sehr nützlich. . .

²Bei (strikten) Echtzeitsystemen mindestens die Stoßlänge des „schlimmsten Falls“ (*worst-case execution time, WCET*).



Statisch vs. Dynamisch

- Abläufe entkoppelt von oder gekoppelt mit der Programmausführung bestimmen und entsprechend entwickeln
 - statische Planung** (*off-line scheduling*), vorlaufend
 - vor Betrieb des Prozess- oder Rechensystems
 - die Berechnungskomplexität verbietet Planung im laufenden Betrieb
 - z.B. die Berechnung, dass alle Zeitvorgaben garantiert eingehalten werden
 - unter Berücksichtigung jeder abfangbaren katastrophalen Situation
 - Ergebnis der Vorberechnung 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**
 - dynamische Planung** (*on-line scheduling*), mitlaufend
 - während Betrieb des Prozess- oder Rechensystems
 - Stapelsysteme, interaktive Systeme, verteilte Systeme
 - schwache und feste Echtzeitsysteme
- auch hier ist die Abgrenzung nicht immer so scharf: von vorläufigen Ablaufplänen ausgehen zu können, ist sehr hilfreich. . .

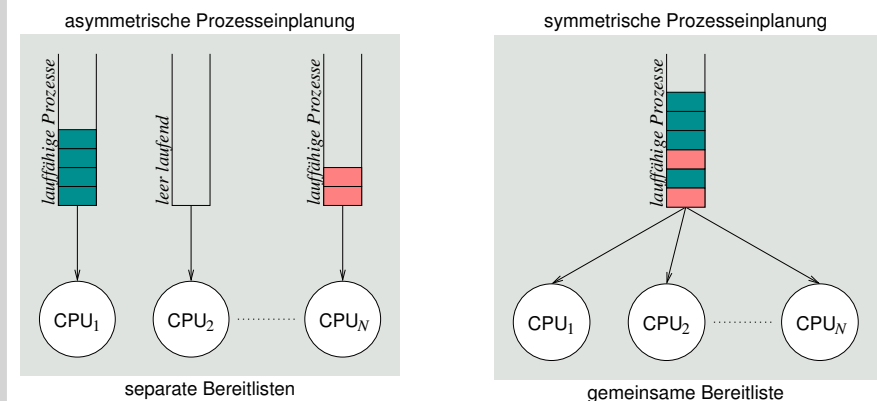


Asymmetrisch vs. Symmetrisch (1)

- für **mehrere Prozessoren** Abläufe nach verschiedenen Kriterien oder ihren wechselseitigen Entsprechungen festlegen
 - asymmetrische Planung** (*asymmetric scheduling*)
 - je nach den Prozesseigenschaften der **Maschinenprogrammebene**
 - obligatorisch in einem asymmetrischen Multiprozessorsystem
 - Rechnerarchitektur mit **programmierbare Spezialprozessoren**
 - z.B. Grafik- und/oder Kommunikationsprozessoren einerseits
 - ein Feld konventioneller (gleichartiger) Prozessoren andererseits
 - optional in einem symmetrischen Multiprozessorsystem (s.u.)
 - das Betriebssystem hat freie Hand über die Prozessorvergabe
 - Prozesse in funktionaler Hinsicht ungleich verteilen (müssen)
 - symmetrische Planung** (*symmetric scheduling*)
 - je nach den Prozesseigenschaften der **Befehlssatzebene**
 - identische Prozessoren, alle geeignet zur Programmausführung
 - Prozesse möglichst gleich auf die Prozessoren verteilen: **Lastausgleich**
- dabei kann jedem Prozessor eine eigene Bereitliste zugeordnet sein oder (Gruppen von) Prozessoren teilen sich eine Bereitliste



Asymmetrisch vs. Symmetrisch (2)



- lokale Bereitliste
- ggf. ungleichmäßige Auslastung
- ohne gegenseitige Beeinflussung
- keine Multiprozessorsynchronisation
- globale Bereitliste
- ggf. gleichmäßige Auslastung
- gegenseitige Beeinflussung
- Multiprozessorsynchronisation



Gliederung

Einführung

Einordnung
Klassifikation

Verfahrensweisen
Kooperativ
Verdrängend
Probabilistisch
Mehrstufig

Zusammenfassung



Klassische Planungs- bzw. Auswahlverfahren

- betrachtet werden grundlegende Ansätze für **Uniprozessorsysteme**, je nach Klassifikationsmerkmal bzw. nichtfunktionaler Eigenschaft:
 - 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, MLFQ (FB)
 - Multikulti...
- dabei steht die Fähigkeit zur **Interaktion** mit „externen Prozessen“ (insb. dem Menschen) als Gütemerkmal im Vordergrund
 - d.h., Auswirkungen auf die **Antwortzeit** von Prozessen



FCFS

first-come, first-served

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

- Prozesse werden nach ihrer **Ankunftszeit** (*arrival time*) eingeplant und in der sich daraus ergebenden Reihenfolge auch verarbeitet
 - nicht-verdrängendes Verfahren, setzt kooperative Prozesse voraus
- gerechtes Verfahren auf Kosten einer im Mittel höheren Antwortzeit und niedrigerem E/A-Durchsatz
 - suboptimal bei einem Mix von kurzen und langen Rechenstößen

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

- Problem: **Konvoieffekt**
 - kurze Prozesse bzw. Rechenstöße folgen einem langen...



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
∅					100	26.00

T_s = Bedienzeit, T_q = Durchlaufzeit

normalisierte Durchlaufzeit (T_q/T_s)

- ideal für A und B, unproblematisch für D
- schlecht für C
 - sie steht in einem extrem schlechten Verhältnis zur Bedienzeit T_s
 - typischer Effekt im Falle von kurzen Prozessen, die langen folgen



Verdrängendes FCFS, Zeitscheiben, CPU-Schutz.

- Prozesse werden nach ihrer **Ankunftszeit** ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant
 - verdrängendes Verfahren, nutzt **periodische Unterbrechungen**
 - Zeitgeber (*timer*) liefert asynchrone Programmunterbrechungen
 - jeder Prozess erhält eine **Zeitscheibe** (*time slice*) zugeteilt
 - obere Schranke für die Rechenstoßlänge eines laufenden Prozesses
- Verringerung der bei FCFS auftretenden Benachteiligung von Prozessen mit kurzen Rechenstößen
 - 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 Rechenstoßes“
- Problem: **Konvoieffekt**
 - Prozesse kürzer als die Zeitscheibe folgen einem, der verdrängt wird. . .

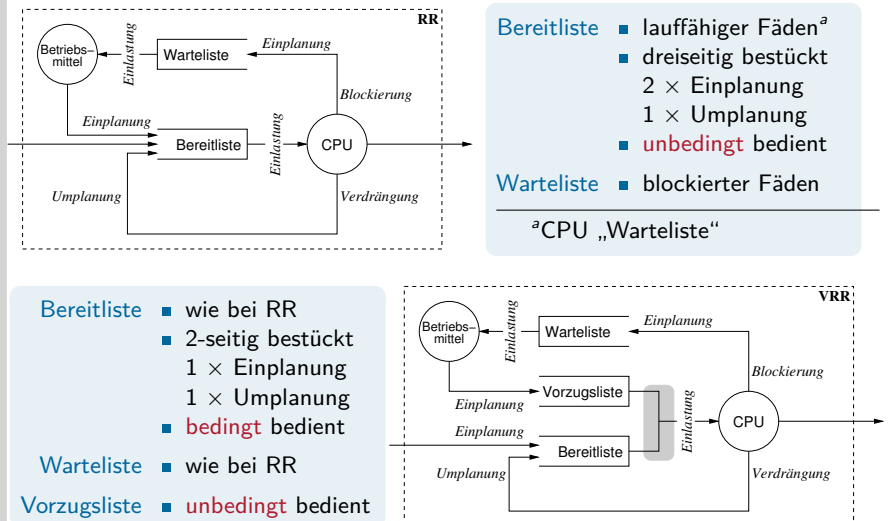


- da die Zuteilung der Zeitscheiben an Prozesse nach FCFS geschieht, werden kurze Prozesse nach wie vor benachteiligt:
 - E/A-intensive Prozesse** schöpfen ihre Zeitscheibe selten voll aus
 - sie beenden ihren Rechenstoß freiwillig
 - vor Ablauf der Zeitscheibe
 - CPU-intensive Prozesse** schöpfen ihre Zeitscheibe meist voll aus
 - sie beenden ihren Rechenstoß unfreiwillig
 - durch Verdrängung
- unabhängig davon werden jedoch alle Prozesse immer reihum bedient
- wird eine Zeitscheibe durch einen Prozess nicht ausgeschöpft, verteilt sich die CPU-Zeit zu Ungunsten E/A-intensiver Prozesse
 - E/A-intensive Prozesse werden schlechter bedient
 - E/A-Geräte sind schlecht ausgelastet
 - Varianz der Antwortzeit E/A-intensiver Prozesse kann beträchtlich sein
 - in Abhängigkeit vom jeweiligen Mix von Prozessen



RR mit Vorzugswarteschlange und variablen Zeitscheiben, um interaktive (d.h., E/A-intensive) Prozesse nicht zu benachteiligen.

- auf E/A wartende Prozesse werden mit Beendigung ihres jeweiligen Ein-/Ausgabestoßes bevorzugt eingeplant (d.h., bereit gestellt)
 - **Einplanung** mittels einer der Bereitliste vorgeschalteten **Vorzugsliste**
 - FIFO \leadsto evtl. Benachteiligung hoch-interaktiver Prozesse; daher. . .
 - aufsteigend sortiert nach dem **Zeitscheibenrest** eines Prozesses
 - **Umplanung** bei Ablauf der aktuellen Zeitscheibe
 - 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
- erreicht durch strukturelle Maßnahmen — nicht durch analytische
- kein voll-verdrängendes Verfahren
 - die Einlastung auch des kürzesten bereitgestellten Prozesses erfolgt nicht zum Zeitpunkt seiner Bereitstellung
 - sondern frühestens nach Ablauf der aktuellen Zeitscheibe



Zeitreihen bilden, analysieren und verwerten: nicht verdrängend.

- jeder Prozess wird entsprechend der für ihn im Durchschnitt oder maximal **erwarteten Bedienzeit** eingeplant
 - Grundlage dafür ist *a priori* Wissen über die **Prozesslaufzeiten**:
 - Stapelbetrieb** Programmierer setzen **Frist** (*time limit*)
 - Produktionsbetrieb** Erstellung einer **Statistik** durch Probeläufe
 - Dialogbetrieb** **Abschätzung** von Rechenstoßlängen 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 bei Benachteiligung längerer Prozesse
 - ein **Verhungern** (*starvation*) dieser Prozesse ist möglich
- ohne Konvoi-Effekt — jedoch ist als praktikable Implementierung nur die **näherungsweise Lösung** möglich
 - da die Rechenstoßlängen nicht exakt im Voraus bestimmbar
 - die obere Grenze einer Stoßlänge nicht selten auch unvorhersagbar ist



- ein **heuristisches Verfahren**, das für jeden Prozess den **Mittelwert** über seine jeweiligen Rechenstoßlängen bildet
 - damit ist die erwartete Länge des nächsten Rechenstoßes eines Prozesses:

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

- arithmetisches Mittel aller gemessenen Rechenstoßlängen des Prozesses
- Problem dieser Berechnung ist die **gleiche Wichtigkeit** aller Rechenstöße
- jüngere Rechenstöße machen jedoch die **Lokalität** eines Prozesses aus
 - diesen Stößen sollte eine größere Wichtigkeit gegeben werden (vgl. S. 23)
- die **Messung** der Dauer eines Rechenstoßes geschieht im Moment der Prozesseinlastung (d.h., der Prozessumschaltung)
 - Stoppzeit T_2 von P_j entspricht (in etwa) der Startzeit T_1 von P_{j+1}
 - gemessen in **Uhrzeit** (*clock time*) oder **Uhrtick** (*clock tick*)
 - dann ergibt $T_2 - T_1$ die gemessene Rechenstoßlänge für jeden Prozess P_i
 - der Differenzwert wird im jeweiligen Prozesskontrollblock akkumuliert



- mittels **Dämpfungsfiler** (*decay filter*), d.h., der **Dämpfung** (*decay*) der am weitesten zurückliegenden Rechenstöße:

$$\begin{aligned} S_{n+1} &= \frac{1}{n} \cdot T_n + \frac{n-1}{n} \cdot S_n \\ &= \alpha \cdot T_n + (1-\alpha) \cdot S_n \end{aligned}$$

- mit zuletzt gemessener (T_n) und geschätzter (S_n) Rechenstoßlänge
- für den konstanten **Wichtungsfaktor** α gilt dabei: $0 < \alpha < 1$
 - drückt die **relative Wichtigkeit** einzelner Rechenstöße der Zeitreihe aus
- um die Wirkung des Wichtungsfaktors zu verdeutlichen, die teilweise Expansion der Gleichung wie folgt:
 - $S_{n+1} = \alpha T_n + (1-\alpha)T_{n-1} + \dots + (1-\alpha)^i T_{n-i} + \dots + (1-\alpha)^n S_1$
- Beispiel der Entwicklung für $\alpha = 0.8$:
 - $S_{n+1} = 0.8 T_n + 0.16 T_{n-1} + 0.032 T_{n-2} + 0.0064 T_{n-3} + \dots$
 - zurückliegende Rechenstöße des Prozesses verlieren schnell an Gewicht



Hungerfreies SPN.

- 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
- ausgewählt wird der Prozess mit dem größten Verhältniswert R
- die periodische Aktualisierung betrifft alle Einträge in der Bereitliste und findet im Hintergrund des aktuellen Prozesses statt
 - ausgelöst durch einen **Uhrtick** (*clock tick*)
- Anmerkung: ein Anstieg der Wartezeit eines Prozesses bedeutet seine **Alterung** (*aging*)
 - der Alterung entgegenwirken beugt Verhungern (*starvation*) vor



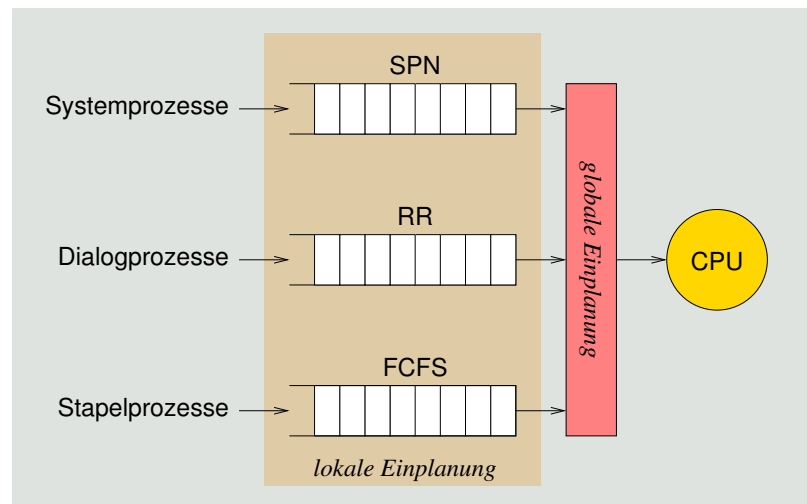
Verdrängendes SPN, Hungergefahr, Effektivität von VRR.

- Prozesse werden nach ihrer **erwarteten Bedienzeit** eingeplant und in unregelmäßigen Zeitabständen **spontan** umgeplant
 - sei T_{et} die erwartete Rechenstoßlänge eines eintreffenden Prozesses
 - sei T_{rt} die verbleibende Rechenstoßlänge des laufenden Prozesses
 - der laufende Prozess wird verdrängt, wenn gilt: $T_{et} < T_{rt}$
- die **Umplanung** erfolgt ereignisbedingt und (ggf. voll) verdrängend im Moment der Ankunftszeit eines Prozesses
 - z.B. bei Beendigung des Ein-/Ausgabestoßes eines wartenden Prozesses
 - allgemein: bei Aufhebung der Wartebedingung für einen Prozess
- bei **Verdrängung** kommt der betreffende Prozess entsprechend der Restdauer seiner erwarteten Rechenstoßlänge auf die Bereitliste
 - führt allgemein zu besseren Antwort- und Durchlaufzeiten
 - gegenüber VRR steht der Aufwand zur Rechenstoßlängenabschätzung



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
 - Hungergefahr für Prozesse tiefer liegender Listen
 - dynamisch** – die Listen im Zeitmultiplexverfahren wechseln
 - z.B. 40 % System-, 40 % Dialog-, 20 % Stapelprozesse
- dem Prozess einen Typen zuzuordnen ist eine statische Entscheidung
 - sie wird zum Zeitpunkt der Prozesszeugung getroffen

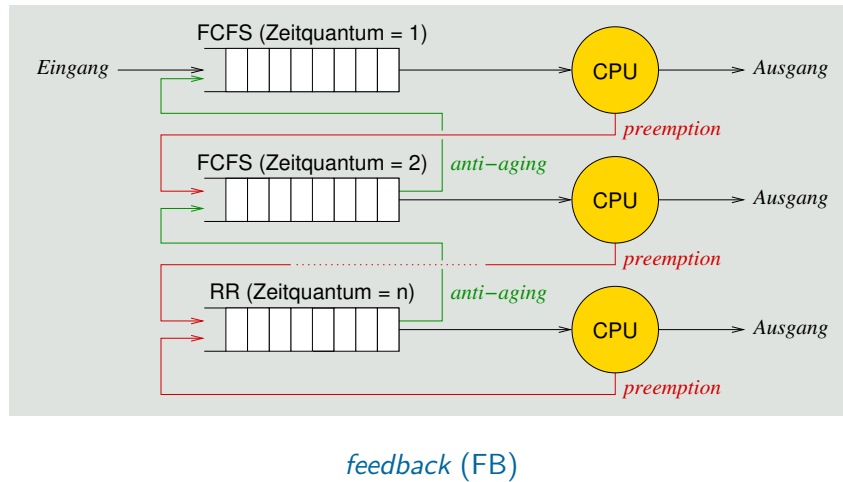


Begünstigt kurze/interaktive Prozesse, ohne die relativen Stoßlängen kennen zu müssen.

- 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 (penalisation)**
 - Prozesse mit langen Rechenstößen fallen nach unten durch
 - Prozesse mit kurzen Rechenstößen laufen relativ schnell durch
- **Alterung (ageing)**
 - nach unten durchfallende Prozesse finden seltener statt
 - durchgefallene Prozesse nach einer **Bewährungsfrist** wieder anheben



MLFQ: Bestrafung und Bewährung



feedback (FB)

Gliederung

Einführung

Einordnung
Klassifikation

Verfahrensweisen
Kooperativ
Verdrängend
Probabilistisch
Mehrstufig

Zusammenfassung

Gegenüberstellung

	FCFS	RR	VRR	SPN	HRRN	SRTF
kooperativ	✓			(✓)	(✓)	
verdrängend		✓	✓			✓
probabilistisch				✓	✓	✓
deterministisch	keine bzw. nicht von sich aus allein ~ EZS [13]					

- MLQ und MLFQ erlauben eine Kombination dieser Verfahren, jedoch abgestuft und nicht alle zusammen auf derselben Ebene
 - dadurch wird letztlich eine **Priorisierung** der Strategien vorgenommen
 - entsprechend der globalen Strategie, die den Ebenenwechsel steuert
 - teilweise wird so speziellen Anwendungsbedürfnissen entgegengekommen
 - z.B. FCFS priorisieren ~ „number crunching“ fördern
- jedes dieser Verfahren stellt bestimmte **Gütemerkmale** [7] in den Vordergrund und vergibt damit indirekt Prioritäten an Prozesse

Prioritäten setzende Verfahren

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

- Prozessvorrang** bedeutet die bevorzugte Einlastung von Prozessen mit höherer Priorität und wird auf zwei Arten bestimmt:
 - statisch**
 - Zeitpunkt der **Prozesserzeugung** ~ Laufzeitkonstante
 - wird im weiteren Verlauf nicht mehr verändert
 - erzwingt die deterministische Ordnung zw. Prozessen
 - dynamisch**
 - „jederzeit“ im **Prozessintervall** ~ Laufzeitvariable
 - die Berechnung erfolgt durch das Betriebssystem
 - ggf. in Kooperation mit den Anwendungsprogrammen
 - erzwingt keine deterministische Ordnung zw. Prozessen
- damit ist allerdings noch nicht **Echtzeitverarbeitung** garantiert, bei der Prozessvorrang eine maßgebliche Rolle spielt
 - die **Striktheit von Terminvorgaben** ist einzuhalten: weich, fest, hart
 - entsprechend der jeweiligen Anforderungen der Anwendungsdomäne
 - keines der behandelten Verfahren sichert dies dem Anwendungssystem zu

- Prozesseinplanung unterliegt einer breit gefächerten **Einordnung**
 - kooperativ/verdrängend
 - deterministisch/probabilistisch
 - statisch/dynamisch
 - asymmetrisch/symmetrisch
- die entsprechenden **Verfahrensweisen** sind z.T. sehr unterschiedlich
 - FCFS: kooperativ
 - RR, VRR: verdrängend
 - SPN, HRRN, SRTF: probabilistisch
 - MLQ, MLFQ (FB): mehrstufig
- Prioritäten setzende Verfahren legen einen **Prozessvorrang** fest
 - FCFS: Ankunftszeit
 - RR: Ankunftszeit, VRR: Ankunftszeit nach Ein-/Ausgabestoß
 - SPN: Rechenstoß, HRRN: Verhältniswert, SRTF: Rechenstoßrest
- eine weitere Dimension ist die **Striktheit von Terminvorgaben**
 - die jedoch keins der behandelten Verfahren an sich berücksichtigt. ...



- [1] BAYER, R. :
Symmetric binary B-Trees: Data structure and maintenance algorithms.
In: *Acta Informatica* 1 (1972), Dezember, S. 290–306
- [2] COFFMAN, E. G. ; DENNING, P. J.:
Operating System Theory.
Prentice Hall, Inc., 1973
- [3] CONWAY, R. W. ; MAXWELL, L. W. ; MILLNER, L. W.:
Theory of Scheduling.
Addison-Wesley, 1967
- [4] GUIBAS, L. J. ; SEDGEWICK, R. :
A dichromatic framework for balanced trees.
In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS 1978)*, IEEE, 1978, S. 8–21
- [5] HÖNIG, T. :
Der O(1)-Scheduler im Kernel 2.6.
In: *Linux Magazin* (2004), Februar, Nr. 2



- [6] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Dialog- und Echtzeitverarbeitung.
In: [9], Kapitel 7.2
- [7] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Einplanungsgrundlagen.
In: [9], Kapitel 9.1
- [8] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Prozesse.
In: [9], Kapitel 6.1
- [9] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)
- [10] KLEINROCK, L. :
Queuing Systems. Bd. I: Theory.
John Wiley & Sons, 1975
- [11] KORNAI, J. :
Economics of Shortage.
North-Holland Publishing Company, 1980



- [12] LISTER, A. M. ; EAGER, R. D.:
Fundamentals of Operating Systems.
The Macmillan Press Ltd., 1993. –
ISBN 0–333–59848–2
- [13] LIU, J. W. S.:
Real-Time Systems.
Prentice-Hall, Inc., 2000. –
ISBN 0–13–099651–3



UNIX klassisch

- zweistufig, Antwortzeiten minimierend, Interaktivität fördernd:

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

- 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 Dämpfungsfiler variiert von UNIX zu UNIX

high-level mittelfristig; mit **Umlagerung** (*swapping*) 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)

- $p_usrpri = PUSER + \lceil \frac{p_cpu}{4} \rceil + 2 \cdot p_nice$
 - 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), sekundlich

- $p_cpu = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p_cpu + p_nice$
- **Sonderfall:** Prozesse schließen länger als eine Sekunde
 - $p_cpu = \lceil \frac{2 \cdot load}{2 \cdot load + 1} \rceil^{p_slptime} \cdot p_cpu$



UNIX 4.3 BSD

Glättung durch Dämpfungsfiler

- Annahme 1:
 - mittlere Auslastung ($load$) sei 1

$$\frac{2 \cdot load}{2 \cdot load + 1} = \frac{2}{3} = 0.66 \leadsto p_cpu = 0.66 \cdot p_cpu + p_nice$$

- 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

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

quantum	tgexp	slprt	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
<i>time-sharing</i>	TS	0–59
<i>interactive</i>	IA	0–59
<i>system</i>	SYS	60–99
<i>real time</i>	RT	100–109

MLFQ in Klasse TS bzw. IA:

- quantum* Zeitscheibe (ms)
- tgexp* Ebene bei Bestrafung
- slprt* Ebene nach Deblokierung
- maxwait* ohne Bedienung (s)
- lwait* Ebene bei Bewährung

- Besonderheit: *dispatch table* (TS, IA) kapselt alle Entscheidungen
 - kunden-/problemspezifische Lösungen durch verschiedene Tabellen



Beispiel:

- 1 × CPU-Stoß à 1000 ms
- 5 × 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...

- Prozess wird verdrängt, muss auf die erneute Einlastung warten
- Alterung des wartenden Prozesses wird durch *Prioritätsanhebung* entgegengewirkt (*anti-aging*)
- die höhere Ebene erreicht, sinkt der Prozess danach 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



- 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 Prozessen und Epochen:
 - jeder Prozess besitzt eine einstellbare *Zeitquantumbasis* (*nice(2)*)
 - 20 Ticks ≈ 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** besitzen *statische Prioritäten*: 0–99
 - je kleiner der Wert, desto höher die Priorität
 - schwache Echtzeit (vgl. [6, S. 16])



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

FIFO	verdrängbare, kooperative Echtzeitprozesse	} eine Bereitliste
RR	Echtzeitprozesse derselben Priorität	
other	konventionelle („time-shared“) Prozesse	
- Prozessauswahl greift auf eine **Gütefunktion** zurück: $O(n)$
 - $v = -1000$ der Prozess ist *Init*
 - $v = 0$ 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



- Prozessplanung hat **konstante Berechnungskomplexität** [5]:
 - Prioritätsfelder** zwei Tabellen pro CPU: *active, expired*
 - jedes Feld eine Bitkarte (*bitmap*) von n Einträgen
 - mit n gleich der Anzahl von Prioritätsebenen
 - Prioritätsebenen** 140 Ebenen = Einträge pro Tabelle
 - 0–99 für Echtzeit-, 100–139 für sonstige Prozesse
 - pro Ebene eine (doppelt verkettete) Bereitliste
 - ist Bitkartenposition i gesetzt (1, *true*), dann ist wenigstens ein Prozess auf der Bereitliste von Ebene i verzeichnet
 - zur Listenauswahl wird die Bitkarte von Anfang ($i = 0$) an abgesucht
 - ggf. unter Zuhilfenahme spezieller Bitoperationen des Prozessors (x86: BSF)
- Prioritäten gemeiner 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 Epochenswechsel werden die Tabellen ausgetauscht: Zeigerwechsel



- der Planer verzichtet auf Prozessheuristik und feste Zeitscheiben
 - vielmehr garantiert er jedem Prozess einen bestimmten **Prozessoranteil**
 - $1/N$ Zeiteinheiten, mit N gleich der Anzahl der lafbereiten Prozesse
 - der Gesamtanteil wächst und schrumpft mit der Gesamtzahl dieser Prozesse
 - der relative Anteil variiert mit der Priorität (`nice(2)`) eines Prozesses
 - in Bezug auf diesen Anteil gilt dabei für einen laufenden Prozess:
 - (a) er kann den Prozessor früher abgeben (blockiert: E/A-intensiver Prozess)
 - (b) er darf den Prozessor länger behalten (rechenintensiver Prozess)
 - (c) er wird verdrängt, sobald er sein $1/N$ -tel Zeitanteil konsumiert hat und ein anderer Prozess wartet, er länger als der kürzeste lafbereite Prozess läuft
 - Fall (c) wird regelmäßig überprüft, durch einen Zeitgeber (*clock tick*)
- Rechenstoßlängen der Prozesse haben eine **minimale Granularität**

Definition (*minimum granularity, MG*)

Die Zeitspanne, die einem Prozess auf dem Prozessor zuzubilligen ist, bevor dem Prozess der Prozessor wieder entzogen werden kann.

- 1–4 ms, die **Periodenlänge** des Zeitgebers; Vorgabe 4 ms



- die **Ziellatenz** des Planers bestimmt den effektiven Prozessoranteil

Definition (*target latency, TL*)

Die Mindestzeit, die für jeden lafbereiten Prozess zur Verfügung zu stellen ist, um wenigstens eine Runde des Prozessors zu erhalten.

- auch die untere Grenze der **Periodenlänge** des Planers; Vorgabe 20 ms
- $$P_{sched} = \begin{cases} TL & \text{wenn } N \leq TL/MG \\ N \times MG & \text{sonst} \end{cases}$$

- wartende Prozesse geben den Beginn ihres nächsten Zeitschlitzes vor

Definition (*virtual runtime, VR*)

Die aus den jeweiligen Rechenstoßlängen akkumulierte Laufzeit eines Prozesses in *ns*, gewichtet mit seiner „Nettigkeit“ (*niceness*).

- wird in jeder Zeitgeberperiode oder bei der Prozessorabgabe aktualisiert
- bestimmt den Moment des Prozessorentzugs und den Bereitlistenplatz
 - der laufende Prozess wird verdrängt, wenn $VR_{running} > VR_{ready}$



- Prozesse vom Prozessor zu verdrängen ist **Mangelwirtschaft** [11] ☹
 - es besteht ein Mangel an „Waren“ (Prozessoren)
 - während genug „Geld“ (Prozesse) zum Kauf dieser Waren vorhanden ist

Definition (*idealer Prozessor*)

Ein Prozessor, der jeden Prozess mit genau gleicher Geschwindigkeit parallel ausführen kann, jeweils mit $1/N$ -tel Zeiteinheiten (*ns*).

- alle Prozesse erhalten den Prozessor für die gleiche relative Laufzeit

Definition (*maximum execution time, MET*)

Die Zeit, die ein Prozess auf einen idealen Prozessor erwarten würde.

- für den nächsten Prozess auf der Bereitliste ist dies die Wartezeit auf den Prozessor, normalisiert auf die Gesamtzahl der lauffähigen Prozesse
- nach $MET = P_{sched}/N$ Zeiteinheiten wird der Prozess mit der kürzesten *VR* den Prozessor zugeteilt erhalten



- die Bereitliste ist ein **balancierter binärer Suchbaum** [1]:³ $O(\log n)$
 - n gleicht der Anzahl der Knoten im Baum, d.h., der lafbereiten Prozesse
 - sortiert nach der „**Vorkaufszeit**“ eines Prozesses für den Prozessor in *ns*

Definition (*preemption time*)

Der Zeitpunkt eines (auf der Bereitliste) wartenden Prozesses zur Ausübung des Vorkaufsrechts, den Prozessor als erster angeboten zu bekommen.

- die **gewichtete virtuelle Laufzeit** eines vormals gelaufenen Prozesses
- diese variiert von Prozess zu Prozess auf der Bereitliste
- ein zunehmender Wert von links nach rechts innerhalb des RS-Baums
- Prozesse sind nach dem Beginn ihres nächsten Rechenstoßes gelistet
 - frühestens am Ende der laufenden Zeitgeberperiode wird verdrängt
 - erhält der ganz links im RS-Baum verzeichnete Prozess den Prozessor und
 - der verdrängte Prozess wird gemäß seiner *VR* im RS-Baum eingetragen
 - ein erzeugter Prozess erhält die **minimale aktuelle virtuelle Laufzeit**
 - die kleinste vom System festgestellte virtuelle Laufzeit eines Prozesses

³Rot-Schwarz-Baum (RS-Baum: *red-black tree* [4])

