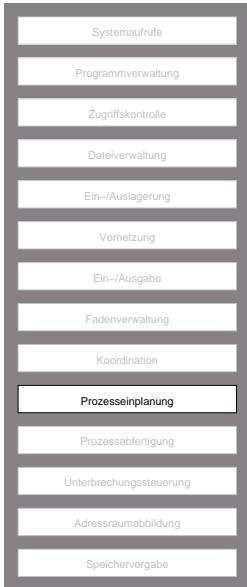


# Prozesseinplanung



## Uniprozessor-Scheduling für den *Allgemeinzweckbetrieb*

- Abfertigungszustände und Zustandsübergänge
- grundlegende Verfahrensweisen
- Fallstudien

Multiprozessor  
Echtzeit } -Scheduling im Hauptstudium

☞ Klassifikation, Kriterien: Folien 6-65 bis 6-75

## Abfertigungszustände

Jedem Prozess ist in Abhängigkeit von der Einplanungsebene ein *logischer Zustand* zugeordnet, der den Abfertigungszustand zu einem Zeitpunkt angibt:

### kurzfristig (*short-term scheduling*)

- bereit, laufend, blockiert

### mittelfristig (*medium-term scheduling*)

- schwebend bereit, schwebend blockiert

### langfristig (*long-term scheduling*)

- erzeugt, gestoppt, beendet

## Kurzfristige Einplanung

**bereit** (*ready*) zur Ausführung durch den Prozessor (die CPU)

- der Prozess ist auf der Warteliste für die CPU-Zuteilung (*ready list*)
- seine Listenposition bestimmt sich durch das Einplanungsverfahren

**laufend** (*running*), Zuteilung des Betriebsmittels „CPU“ ist erfolgt

- der Prozess führt Berechnungen durch, er vollzieht seinen CPU-Stoß
- für jeden Prozessor gibt es zu einem Zeitpunkt nur einen laufenden Prozess

**blockiert** (*blocked*) auf ein bestimmtes Ereignis

- der Prozess führt „Ein-/Ausgabe“ durch, er vollzieht seinen E/A-Stoß
- er erwartet die Erfüllung mindestens einer Bedingung (✗ Kap. 7/8)

## Mittelfristige Einplanung

Ein Prozess ist komplett ausgelagert, d. h., der Inhalt seines gesamten Adressraums wurde in den Hintergrundspeicher verschoben (*swap-out*) und der von dem Prozess belegte Vordergrundspeicher wurde freigegeben. Die Einlagerung (*swap-in*) des Adressraums ist abzuwarten:

**schwebend bereit** (*ready suspend*)

- die CPU-Zuteilung (☞ „bereit“) ist außer Kraft gesetzt
- der Prozess ist auf der Warteliste für die Speicherzuteilung

**schwebend blockiert** (*blocked suspend*)

- der Prozess erwartet weiterhin ein Ereignis (☞ „blockiert“)
- tritt das Ereignis ein, wird der Prozess „schwebend bereit“

# Langfristige Einplanung

**erzeugt** (created) und fertig zur Programmverarbeitung

☞ `fork(2)`

- der Prozess ist instanziert, ihm wurde ein Programm zugeordnet
  - ggf. steht die Zuteilung des Betriebsmittels „Speicher“ jedoch noch aus

**gestoppt** (*stopped*)

☞ signal(3)

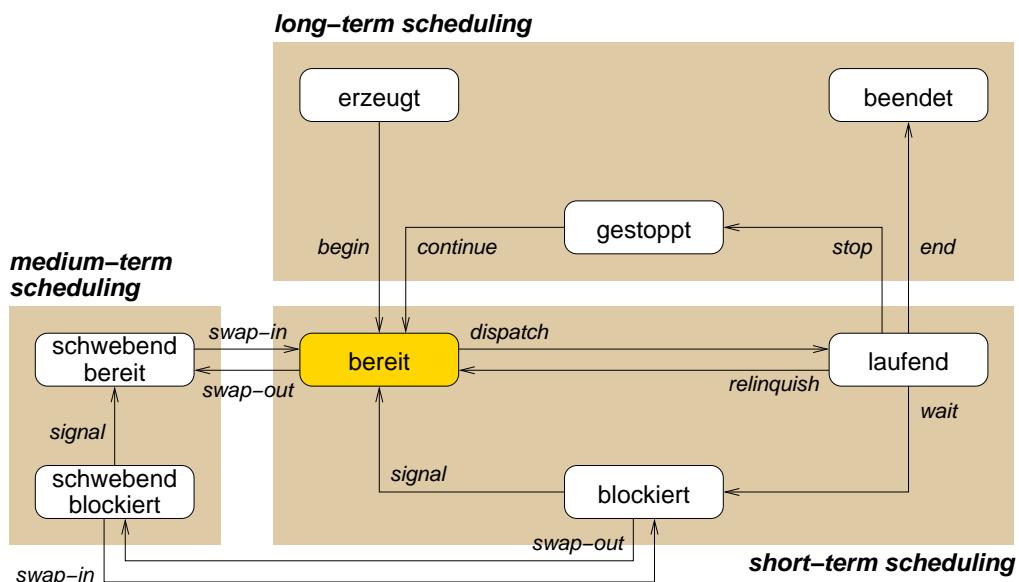
- der Prozess wurde angehalten, automatisch oder manuell (z. B.  $\sim Z$ )
  - Ursachen können z. B. sein Überlast und Verklemmungsvermeidung

**beendet** (ended) und erwartet die Entsorgung

☞ `exit(2) → wait(2)`

- der Prozess ist terminiert, seine Betriebsmittel werden freigegeben
  - ggf. muss ein anderer Prozess den „Kehraus“ vollenden (z. B. UNIX)

## Zustandsübergänge



## Einplanungs-/Auswahlzeitpunkt

- jeder Übergang in den Zustand „bereit“ aktualisiert die CPU-Warteschlange
  - eine Entscheidung über die Einreihung des Prozessdeskriptors wird getroffen
  - das Ergebnis dieser Entscheidung ist eine Funktion der Einplanungsstrategie
- Einplanung (*scheduling*) bzw. Umplanung (*rescheduling*) erfolgt, . . .
  - nachdem ein Prozess erzeugt worden ist *begin*
  - wenn ein Prozess freiwillig die Kontrolle über die CPU abgibt *relinquish*
  - sofern das von einem Prozess erwartete Ereignis eingetreten ist *signal*
  - sobald ein Prozess wieder aufgenommen werden kann *continue*
- ein Prozess kann dazu gedrängt werden, die CPU freiwillig abzugeben [Wodurch?]

## Einplanungsverfahren

- klassische Einplanungs-/Auswahlstrategien:
  - FCFS ..... *gerecht*
  - RR, VRR ..... *zeitscheibenbasiert*
  - SPN (SJF), SRTF, HRRN ..... *prioritätssetzend*
  - FB (MLQ, MLFQ) ..... *mehrstufig*
- Fallstudien:
  - UNIX (4.3 BSD, Solaris), NT, Linux (2.4, 2.5/2.6), MacOS X/Mach

## FCFS

## First Come, First Serve

- ein einfaches und gerechtes Verfahren: „wer zuerst kommt, mahlt zuerst“
  - Einreihungskriterium ist die *Ankunftszeit* eines Prozesses
  - arbeitet nicht-verdrängend und setzt kooperative Prozesse voraus
- der Ansatz ist suboptimal bei einem Mix von kurzen und langen CPU-Stößen
  - Prozesse mit  $\begin{cases} \text{langen} \\ \text{kurzen} \end{cases}$  CPU-Stößen werden  $\begin{cases} \text{begünstigt} \\ \text{benachteiligt} \end{cases}$
- Problem „Konvoi(d)effekt“  $\Rightarrow$  hohe Antwortzeit, niedriger E/A-Durchsatz

## FCFS

## Durchlaufzeit

Prozess	Zeiten					$T_q/T_s$
	Ankunft	Bedienung ( $T_s$ )	Start	Ende	Durchlauf ( $T_q$ )	
A	0		1	0	1	1
B	1		100	1	101	100
C	2		1	101	102	100
D	3		100	102	202	199
$\emptyset$					100	26.00

- die *normalisierte Durchlaufzeit* ( $T_q/T_s$ ) von C ist vergleichsweise sehr schlecht
  - sie steht in einem extrem schlechten Verhältnis zur Bedienzeit  $T_s$
- mit dem Problem sind immer kurze Prozesse konfrontiert, die langen folgen

- verringert die bei FCFS auftretende Benachteiligung kurzer CPU-Stöße
  - Basis für *CPU-Schutz*: ein Zeitgeber bewirkt periodische Unterbrechungen
  - die Periodenlänge entspricht typischerweise einer **Zeitscheibe** (*time slicing*)
- mit Ablauf der Zeitscheibe erfolgt ggf. ein Prozesswechsel
  - der unterbrochene Prozess wird ans Ende der Bereitliste verdrängt
  - der nächste Prozess wird gemäß FCFS der Bereitliste entnommen
- die Zeitscheibenlänge bestimmt maßgeblich die Effektivität des Verfahrens
  - zu lang, Degenerierung zu FCFS; zu kurz, sehr hoher *Overhead* [Warum?]
  - Faustregel: etwas länger sein als die Dauer einer „typischen Interaktion“

**E/A-intensive Prozesse** beenden ihren CPU-Stoß innerhalb ihrer Zeitscheibe

☞ sie blockieren und kommen mit Ende ihres E/A-Stoßes in die Bereitliste

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

☞ sie werden verdrängt und kommen sofort wieder in die Bereitliste

- die CPU-Zeit ist zu Gunsten CPU-intensiver Prozesse ungleich verteilt
  - E/A-intensive Prozesse werden schlecht bedient, Geräte schlecht ausgelastet
  - die Varianz der Antwortzeit E/A-intensiver Prozesse erhöht sich

## VRR

## Virtual Round Robin

- vermeidet die bei RR mögliche ungleiche Verteilung der CPU-Zeiten
  - Prozesse kommen mit Ende ihrer E/A-Stöße in eine **Vorzugsliste**
  - diese Liste wird vor der Bereitliste abgearbeitet
- das Verfahren arbeitet mit Zeitscheiben unterschiedlicher Längen
  - Prozesse der Vorzugsliste bekommen keine volle Zeitscheibe zugeteilt
  - ihnen wird die Restlaufzeit ihrer vorher nicht voll genutzten Zeit gewährt
  - sollte ihr CPU-Stoß länger dauern, werden sie in die Bereitliste verdrängt
- die Prozessabfertigung ist dadurch im Vergleich zu RR etwas aufwendiger

## SPN

## Shortest Process Next

- verringert die bei FCFS auftretende Benachteiligung kurzer CPU-Stöße
  - Grundlage dafür ist die Kenntnis über die Prozesslaufzeiten
- das Hauptproblem besteht darin, die Laufzeiten vorhersagen zu können
  - beim Stapelbetrieb geben Programmierer das erforderliche *time limit*<sup>50</sup> vor
  - im Produktionsbetrieb läuft der Job mehrfach nur zu statistischen Zwecken
  - im Dialogbetrieb wird ein Mittelwert der Stoßlängen eines Prozesses gebildet
- Antwortzeiten werden wesentlich verkürzt und die Gesamtleistung steigt

<sup>50</sup>Die Zeitdauer, innerhalb der der Job (wahrscheinlich/hoffentlich) beendet wird, bevor er abgebrochen wird.

- Basis ist die 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 sind jedoch von größerer Bedeutung als ältere
  - sie sollten daher auch mit größerer Wichtung berücksichtigt werden
- das Lokalitätsprinzip erfordert eine stärkere Einbeziehung jüngerer CPU-Stöße

- die am weitesten zurückliegenden CPU-Stöße sollen weniger Gewicht erhalten:

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

- für den konstanten Wichtungsfaktor  $\alpha$  gilt dabei:  $0 < \alpha < 1$
- er drückt die relative Wichtung einzelner CPU-Stöße der Zeitreihe aus

- teilweise Expansion der Gleichung führt zu:

$$S_{n+1} = \alpha T_n + (1 - \alpha) \alpha T_{n-1} + \dots + (1 - \alpha)^i \alpha T_{n-i} + \dots + (1 - \alpha)^n S_1$$

- für  $\alpha = 0.8$ :  $S_{n+1} = 0.8 T_n + 0.16 T_{n-1} + 0.032 T_{n-2} + 0.0064 T_{n-3} + \dots$

## SRTF

### *Shortest Remaining Time First*

- lässt den SPN-Ansatz geeignet erscheinen für den Dialogbetrieb: **Verdrängung**
  - sei  $T_{et}$  die erwartete CPU-Stoßlänge eines eintreffenden Prozesses und  $T_{rt}$  die verbleibende CPU-Stoßlänge des laufenden Prozesses
  - der laufende Prozess wird verdrängt, wenn gilt:  $T_{et} < T_{rt}$
- wie SPN kann auch SRTF Prozesse zum „Verhungern“ (*starvation*) bringen
  - dafür führt das verdrängende Verhalten zu besseren Durchlaufzeiten
  - dem RR-Overhead steht Overhead zur Stoßlängenabschätzung gegenüber

## HRRN

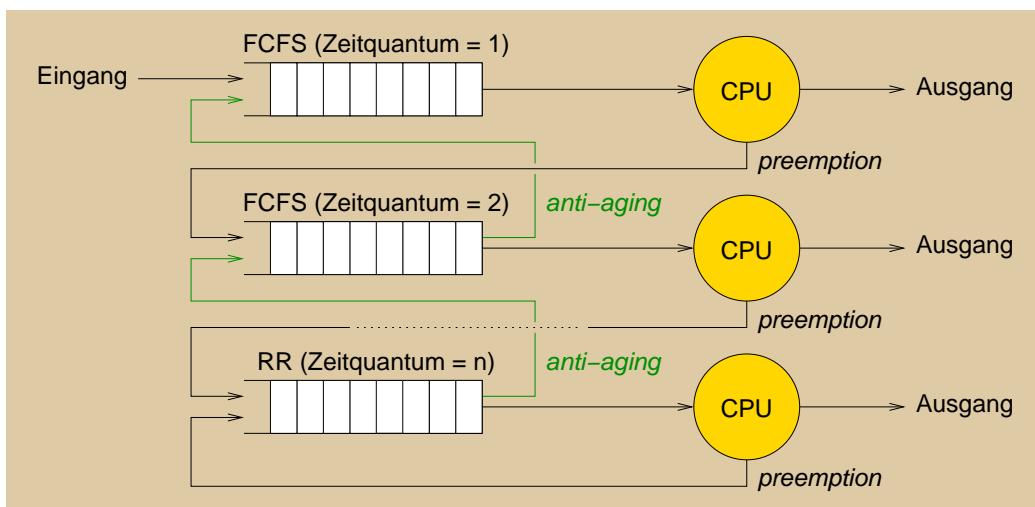
### *Highest Response Ratio Next*

- vermeidet das bei SRTF mögliche Verhungern von Prozessen langer CPU-Stöße
  - das **Altern** (*aging*), d. h., die *Wartezeit* von Prozessen wird berücksichtigt

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

- mit  $w$  = „Wartezeit des Prozesses“ und  $s$  = „erwartete Bedienzeit“
- ausgewählt wird immer der Prozess mit dem größten Verhältniswert  $R$

- begünstigt kurze Prozesse, ohne die relativen Längen der Prozesse zu kennen
  - Grundlage ist die „**Bestrafung**“ (*penalization*) lange gelaufener Prozesse
  - Prozesse unterliegen dem Verdrängungsprinzip
- mehrere Bereitlisten kommen zum Einsatz, je nach Anzahl von *Prioritätsebenen*
  - wenn ein Prozess erstmalig eintrifft, läuft er auf höchster Prioritätsebene
  - mit Ablauf seiner Zeitscheibe, wird er in die nächst niedrige Ebene verdrängt
  - die unterste Ebene arbeitet nach RR, alle anderen (höheren) nach FCFS
- kurze Prozesse laufen relativ schnell durch, lange Prozesse können verhungern
  - die Wartezeit kann berücksichtigt werden, um höhere Ebenen zu erreichen



## Prioritäten

- ein Prozess-„Vorrang“, der Zuteilungsentscheidungen maßgeblich beeinflusst
  - *Echtzeitverarbeitung* bedingt prioritätssetzende Verfahren, nicht umgekehrt

**statische Prioritäten** werden zum Zeitpunkt der Prozesserzeugung festgelegt

- der Wert kann im weiteren Verlauf nicht mehr verändert werden
- das Verfahren erzwingt eine deterministische Ordnung zwischen Prozessen

**dynamische Prioritäten** werden während der Prozesslaufzeit aktualisiert

- die Aktualisierung erfolgt im Betriebssystem, aber auch vom Benutzer aus
- SPN, SRTF, HRRN und FB sind z. B. Spezialfälle dieses Verfahrens

## Kombinierte Verfahren

## Multilevel Scheduling

- mehrere Betriebsformen lassen sich nebeneinander („gleichzeitig“) betreiben
  - z. B. gleichzeitige Unterstützung von 

Dialog- und Hintergrundbetrieb
Echtzeit- und sonstigem Betrieb
  - dialogorientierte bzw. zeitkritische Prozesse werden bevorzugt bedient
- die technische Umsetzung erfolgt typischerweise über mehrere Bereitlisten
  - jeder Bereitliste ist eine bestimmte Zuteilungsstrategie zugeordnet
  - die Listen werden typischerweise nach Priorität, FCFS oder RR verarbeitet
  - ein höchst komplexes Gebilde ↗ **multi-level feedback** (MLFB)
- FB kann als Spezialfall dieses Verfahrens aufgefasst werden

# UNIX

- zweistufiges präemptives Verfahren mit dem Ziel, *Antwortzeiten* zu minimieren
  - low-level** kurzfristig; präemptiv, MLFB, dynamische Prozessprioritäten
    - einmal pro Sekunde:  $prio = cpu\_usage + p\_nice + base$
    - jeder „Tick“ (1/10 s) verringert das Nutzungsrecht über die CPU
    - der „Tickstand“ wird zur Priorität addiert: hohe Zahl → niedrige Priorität
    - das Maß für der CPU-Nutzung ( $cpu\_usage$ ) wird über die Zeit gedämpft
    - die Dämpfungs-/Glättungsfunktion variiert von UNIX zu UNIX
  - high-level** mittelfristig; mit Ein-/Auslagerung (*swapping*) arbeitend
- Prozesse können relativ zügig den Betriebssystemkern verlassen [Wozu ist das gut?]
  - gesteuert über die beim Schlafenlegen einstellbare *Aufweckpriorität*

# UNIX

## 4.3 BSD (1)

- jeden vierten Tick (40 ms) erfolgt die Berechnung der *Benutzerpriorität*:

$$p\_usrpri = PUSER + \left[ \frac{p\_cpu}{4} \right] + 2 \cdot p\_nice$$

- $p\_cpu$  nimmt mit jedem Tick zu und wird einmal pro Sekunde geglättet:

$$p\_cpu = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p\_cpu + p\_nice$$

- Glättung für erwachte Prozesse, die länger als eine Sekunde blockiert waren:

$$p\_cpu = \left[ \frac{2 \cdot load}{2 \cdot load + 1} \right]^{p\_slptime} \cdot p\_cpu$$

**Glättung (decay filter)** Bei einer angenommenen mittleren Auslastung (*load*) von 1 gilt  $p\_cpu = 0.66 \cdot p\_cpu + p\_nice$ . Ferner sei angenommen, ein Prozess sammelt  $T_i$  Ticks im Zeitintervall  $i$  an und  $p\_nice = 0$ :

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

☞ Nach fünf Sekunden gehen nur noch 13 % „alte“ Auslastung ein.

- FB, 60 Ebenen (Warteschlangen)
  - hohe Ebene  $\equiv$  hohe Priorität
- Tabellensteuerung:
 

<i>quantum</i>	<i>tqexp</i>	<i>slpret</i>	<i>maxwait</i>	<i>lwait</i>	Ebene
200	0	50	0	50	0
200	0	50	0	50	1

  - quantum* Zeitscheibe (ms)
  - tqexp* Ebene bei „Bestrafung“
  - slprt* Ebene nach Deblockierung
  - maxwait* ohne Bedienung (s)
  - lwait* Ebene bei „Beförderung“
- rechenintensive Prozesse steigen ab, interaktive steigen auf

<i>quantum</i>	<i>tqexp</i>	<i>slpret</i>	<i>maxwait</i>	<i>lwait</i>	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

**Beispiel:** 1 CPU-Stoß (1000 ms), 5 E/A-Stöße und 5 CPU-Stöße (1 ms)

#	Ebene	CPU-Stoß	Prozesswechsel
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:** Der Prozess wird von einem anderen Prozess (mit höherer Priorität) verdrängt und muss warten. Droht er zu „verhungern“, wird er in der Priorität wieder angehoben („befördert“).

#	Ebene	CPU-Stoß	Prozesswechsel
...			
6	9	200	Zeitscheibe
7	0	20	Beförderung
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
...			

420 ms

## NT (1)

## Prioritätsklassen

- präemptive, prioritäts- und zeitscheibenbasierte Einplanung von Fäden
  - Verdrängung erfolgt auch dann, wenn der Faden sich im Kern befindet
    - ☞ nicht so bei UNIX & Co
  - RR bei gleicher Priorität: 0 reserviert, 1–15 variabel, 16–31 Echtzeit
- die Prozessart (Vorder-/Hintergrund) bestimmt das **Zeitquantum** eines Fadens
  - vermindert sich mit jedem Tick (10 bzw. 15 ms) um 3 oder um 1, falls der Faden in den Wartezustand geht
  - die **Zeitscheibenlänge** variiert mit den Prozessen: 20 – 120 ms
- variable Priorität: *process priority class + relative thread priority + boost*

- Fadenprioritäten werden in bestimmten Situationen dynamisch angehoben

– Abschluss von Ein-/Ausgabe (Festplatten)	+1
– Mausbewegung, Tastatureingabe	+6
– Deblockierung, Betriebsmittelfreigabe (Semaphor, Event, Mutex)	+1
– andere Ereignisse (Netzwerk, <i>Pipe</i> , . . . )	+2
– Ereignis im Vordergrundprozess	+2

☞ die *dynamic boosts* werden mit jedem Tick wieder verbraucht

**Fortschrittsgarantie** alle 3–4 s erhalten bis zu 10 „benachteiligte“ Fäden für zwei Zeitscheiben die Priorität 15

- der Scheduler unterteilt die CPU-Zeit in **Epochen**, die . . .
   
*beginnen*, wenn jeder lauffähige Prozess ein Zeitquantum erhalten hat
   
*enden*, wenn alle lauffähigen Prozesse ihre Zeitquanten verbraucht haben
- **Zeitquanten** (Zeitscheiben) variieren mit den Prozessen und Epochen
  - mit jedem Tick nimmt das Zeitquantum des unterbrochenen Prozesses ab
  - jeder Prozess besitzt eine einstellbare *Zeitquantumbasis*: 20 Ticks  $\approx$  210 ms
  - beide Werte addiert liefert die **dynamische Priorität** eines Prozesses
  - dynamische Anpassung:  $quantum = quantum/2 + (20 - nice)/4 + 1$
- ein *Echtzeitprozess* (☞ schwache EZ) besitzt eine **statische Priorität** (1–99)

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

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

 ↝ eine Bereitliste
- eine **Gütfunktion** liefert den besten Kandidaten aller lauffähigen Prozesse  $O(n)$ 

$v = -1000$	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	++
- teilt ein Prozess mit seinem Vorgänger den Adressraum, hat er einen Bonus

- konstante Berechnungskomplexität** bei der Einplanung lauffähiger Prozesse
  - pro CPU zwei *Prioritätsfelder*: *active*, *expired*
  - pro Feld 140 *Prioritätsebenen*: 1–100 Echtzeit-, 101–140 sonstige Prozesse
  - pro Ebene eine (doppelt verkettete) Bereitliste
- Prioritäten gewöhnlicher Prozesse skalieren je nach Grad der Interaktivität
  - Bonus* (−5) für interaktive Prozesse, *Strafe* (+5) für rechenintensive
  - berechnet am Zeitscheibenende:  $prio = MAX\_RT\_PRIO + nice + 20$
- „abgelaufene Prozesse“ wandern in das „*expired*“-Feld, interaktive bleiben
  - Epochenwechsel:  $aux = active$ ;  $active = expired$ ;  $expired = aux$ ;

- ursprünglich konzipiert für **symmetrische Multiprozessorsysteme (SMP)**
  - jede CPU ist einer *Prozessormenge* (*processor set*) zugeordnet
  - die Prozessormenge definiert eine Einplanungsdomäne (*scheduling domain*)
- Uniprozessorsysteme werden als „Spezialisierung“ aufgefasst
  - in dem Fall gibt es nur genau eine Prozessormenge mit genau einer CPU

**Ziel:** einer CPU einer Prozessormenge einen Faden fair und effektiv zuweisen

- für jede Prozessormenge steht eine eigene *globale Bereitliste* zur Verfügung
- eine *lokale Bereitliste* pro CPU verwaltet nur „prozessoraffine Fäden“<sup>51</sup>

<sup>51</sup>Fäden, die während ihrer gesamten Lebensdauer oder auch nur zeitweise mit bestimmten Prozessoren bzw. prozessorgebundenen Betriebsmitteln (Geräte, Recheneinheiten, . . . ) assoziiert sind.

- **prioritätsorientierte Fadeneinplanung:** 0 (hoch) – 31 (bzw. 127, niedrig)
  - pro Prioritätsebene eine Bereitliste, pro Faden drei Prioritätswerte:
    1. eine Basispriorität, initial übernommen von der „Task-Priorität“
    2. ein niedrigster Wert, den sich der Faden selbst zuweisen kann
    3. die aktuelle Priorität, berechnet sich aus 1. und der CPU-Auslastung
  - konfigurierbare Verfahren: *time sharing* (TS), *gang scheduling*, *background; earliest deadline first* (EDF), *rate monotonic* (RM), *fixed priority* (FP) EZ
- Prozesse erhalten Zeitquanten zugewiesen (☞ TS)  $\approx$  UNIX/4.3 BSD

**handoff scheduling** Priorität eines Fadens wird kurzzeitig auf den niedrigsten Wert (2.) herabgesetzt, um einem anderen Faden Vorrang zu gewähren

# UNIX $\iff$ NT $\iff$ Linux $\iff$ Mach

## Einplanungseinheit „Gewichtsklasse“

- Prozess  $\Leftrightarrow$  4.3 BSD
- Faden  $\Leftrightarrow$  Solaris, MacOS X/Mach, NT, Linux

## Echtzeitfähigkeit Linux, NT, MacOS X/Mach: „schwach echtzeitfähig“

- das Einhalten von Zeitschränken kann nicht garantiert werden
- problematisch bis unzureichend für Steuerungsanwendungen

## Ablaufinvarianz (reentrance) NT, MacOS X/Mach

- Fäden sind insbesondere auch innerhalb des Kerns verdrängbar
- der Kern von „UNIX & Co“ verhält sich eher wie ein Monitor

## Zusammenfassung

- Betriebssysteme müssen drei Arten von Zuteilungsentscheidungen treffen:
  1. *long-term scheduling* von Prozessen, die zum System zugelassen werden
  2. *medium-term scheduling* von aus- oder einzulagernden Prozessen
  3. *short-term scheduling* von Prozessen, die die CPU zugeteilt bekommen
- alle hier betrachteten Verfahren werden dem *short-term scheduling* zugerechnet
  - benutzer- und systemorientierte Kriterien sind schwer zu vereinheitlichen
  - die Auswahl des geeigneten Verfahrens kommt einer Gratwanderung gleich
- kombinierte Verfahren bieten Flexibilität — gegen Implementierungskomplexität