

## 5. Foliensatz

# Betriebssysteme und Rechnernetze

Prof. Dr. Christian Baun

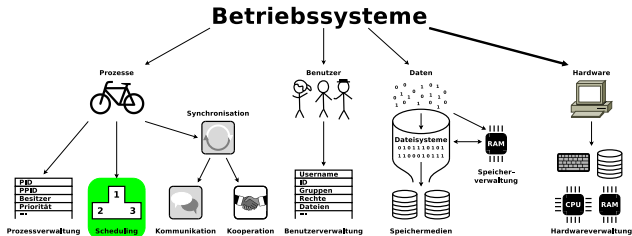
Frankfurt University of Applied Sciences  
(1971–2014: Fachhochschule Frankfurt am Main)  
Fachbereich Informatik und Ingenieurwissenschaften  
[christianbaun@fb2.fra-uas.de](mailto:christianbaun@fb2.fra-uas.de)

# Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - die Schritte des **Dispatchers** (Prozessumschalter) beim Prozesswechsel
  - was **Scheduling** ist
    - wie **präemptives** und **nicht-präemptives Scheduling** funktioniert
    - die Arbeitsweise verschiedener **Scheduling-Verfahren**
    - wie das **Scheduling moderner Betriebssysteme** im Detail funktioniert

Im SS2019 habe ich alle Scheduling-Algorithmen (SJF/SRTF/LJF/LRTF/HRRN) von meinen Vorlesungsmaterialien gelöscht, bei denen für jeden Prozess bekannt sein muss, wie lange er bis zu seiner Terminierung braucht, also wie lange seine Abarbeitungszeit ist. Das ist in der Realität praktisch nie der Fall (⇒ **unrealistisch**)

Übungsblatt 5 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes



# Prozesswechsel – Der Dispatcher (1/2)

- Aufgaben von Multitasking-Betriebssystemen sind u.a.:
  - **Dispatching**: Umschalten des Prozessors bei einem Prozesswechsel
  - **Scheduling**: Festlegen des Zeitpunkts des Prozesswechsels und der Ausführungsreihenfolge der Prozesse
- Der **Dispatcher** (Prozessumschalter) führt die Zustandsübergänge der Prozesse durch

## Wir wissen bereits...

- Beim Prozesswechsel entzieht der Dispatcher dem rechnenden Prozess die CPU und teilt sie dem Prozess zu, der in der Warteschlange an erster Stelle steht
- Bei Übergängen zwischen den Zuständen `bereit` und `blockiert` werden vom Dispatcher die entsprechenden Prozesskontrollblöcke aus den Zustandslisten entfernt und neu eingefügt
- Übergänge aus oder in den Zustand `rechnend` bedeuten immer einen Wechsel des aktuell rechnenden Prozesses auf der CPU

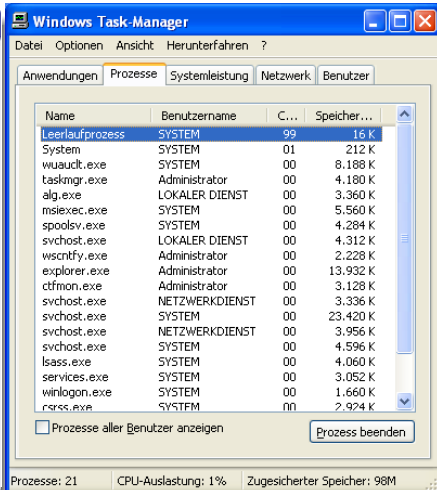
## Beim Prozesswechsel in oder aus dem Zustand `rechnend`, muss der Dispatcher...

- den Kontext, also die Registerinhalte des aktuell ausgeführten Prozesses im Prozesskontrollblock speichern (retten)
- den Prozessor einem anderen Prozess zuteilen
- den Kontext (Registerinhalte) des jetzt auszuführenden Prozesses aus seinem Prozesskontrollblock wieder herstellen

# Prozesswechsel – Der Dispatcher (2/2)

## Der Leerlaufprozess (*System Idle Process*)

- Bei Windows-Betriebssystemen seit Windows NT erhält die CPU zu jedem Zeitpunkt einen Prozess
- Ist kein Prozess im Zustand bereit, kommt der **Leerlaufprozess** zum Zug
- Der Leerlaufprozess ist immer aktiv und hat die niedrigste Priorität
- Durch den Leerlaufprozesses muss der Scheduler nie den Fall berücksichtigen, dass kein aktiver Prozess existiert
- Seit Windows 2000 versetzt der Leerlaufprozess die CPU in einen stromsparenden Modus
- Für jeden CPU-Kern (in Hyperthreading-Systemen für jede logische CPU) existiert ein Leerlaufprozess



<https://unix.stackexchange.com/questions/361245/what-does-an-idle-cpu-process-do>

„In Linux, one idle task is created for every CPU and locked to that processor; whenever there's no other process to run on that CPU, the idle task is scheduled. Time spent in the idle tasks appears as "idle" time in tools such as top. . .“

# Scheduling-Kriterien und Scheduling-Strategien

- Beim Scheduling legt das Betriebssystem die Ausführungsreihenfolge der Prozesse im Zustand bereit fest
- **Keine Scheduling-Strategie...**
  - ist für jedes System optimal geeignet
  - kann alle Scheduling-Kriterien optimal berücksichtigen
    - Scheduling-Kriterien sind u.a. CPU-Auslastung, Antwortzeit (Latenz), Durchlaufzeit (*Turnaround*), Durchsatz, Effizienz, Echtzeitverhalten (Termineinhaltung), Wartezeit, Overhead, Fairness, Berücksichtigen von Prioritäten, Gleichmäßige Ressourcenauslastung...
- Bei der Auswahl einer Scheduling-Strategie muss immer ein **Kompromiss** zwischen den Scheduling-Kriterien gefunden werden

# Nicht-präemptives und präemptives Scheduling

- 2 Klassen von Schedulingverfahren existieren:

- ① **Nicht-präemptives Scheduling** bzw. **Kooperatives Scheduling** (nicht-verdrängendes Scheduling)

- Ein Prozess, der vom Scheduler die CPU zugewiesen bekommen hat, behält die Kontrolle über diese bis zu seiner vollständigen Fertigstellung oder bis er die Kontrolle freiwillig wieder abgibt
    - Problematisch: Ein Prozess kann die CPU so lange belegen wie er will

Beispiele: Windows 3.x, MacOS 8/9, Windows 95/98/Me (für 16-Bit-Prozesse)

- ② **Präemptives Scheduling** (verdrängendes Scheduling)

- Einem Prozess kann die CPU vor seiner Fertigstellung entzogen werden
    - Wird einem Prozess die CPU entzogen, pausiert er so lange in seinem aktuellen Zustand, bis der Scheduler ihm erneut die CPU zuteilt
    - Nachteil: Höherer Overhead als nicht-präemptives Scheduling
    - Die Vorteile von präemptivem Scheduling, besonders die Beachtung von Prozessprioritäten, überwiegen die Nachteile

Beispiele: Linux, MacOS X, Windows 95/98/Me (für 32-Bit-Prozesse), Windows NT (inkl. XP/Visa/7/8/10/11), FreeBSD

# Einfluss auf die Gesamtleistung eines Computers

- Wie groß der Einfluss des verwendeten Schedulingverfahrens auf die Gesamtleistung eines Computers sein kann, zeigt dieses Beispiel
  - Die Prozesse  $P_A$  und  $P_B$  sollen nacheinander ausgeführt werden
    - Läuft ein Prozess mit kurzer Laufzeit vor einem Prozess mit langer Laufzeit, **verschlechtern** sich Laufzeit und Wartezeit des langen Prozesses **wenig**
    - Läuft ein Prozess mit langer Laufzeit vor einem Prozess mit kurzer Laufzeit, **verschlechtern** sich Laufzeit und Wartezeit des kurzen Prozesses **stark**

Prozess	CPU-Laufzeit
A	24 ms
B	2 ms

Reihenfolge	Laufzeit		Durchschnittliche Laufzeit	Wartezeit		Durchschnittliche Wartezeit
	A	B		A	B	
$P_A, P_B$	24 ms	26 ms	$\frac{24+26}{2} = 25 \text{ ms}$	0 ms	24 ms	$\frac{0+24}{2} = 12 \text{ ms}$
$P_B, P_A$	26 ms	2 ms	$\frac{2+26}{2} = 14 \text{ ms}$	2 ms	0 ms	$\frac{0+2}{2} = 1 \text{ ms}$

# Scheduling-Verfahren

- Zahlreiche Scheduling-Verfahren (Algorithmen) existieren
  - Jedes Scheduling-Verfahren versucht unterschiedlich stark, die bekannten Scheduling-Kriterien und -Grundsätze einzuhalten
- Bekannte Scheduling-Verfahren:
  - **Prioritätengesteuertes Scheduling**
  - **First Come First Served (FCFS)** bzw. **First In First Out (FIFO)**
  - ~~Last Come First Served (LCFS)~~
  - **Round Robin (RR)** mit Zeitquantum
  - ~~Shortest/Longest Job First (SJF/LRTF)~~
  - ~~Shortest/Longest Remaining Time First (SRTF/LRTF)~~
  - ~~Highest Response Ratio Next (HRRN)~~
  - **Earliest Deadline First (EDF)**
  - **Fair-Share-Scheduling**
  - ~~Statisches Multilevel-Scheduling~~
  - **Multilevel-Feedback-Scheduling**
  - **Completely Fair Scheduler (CFS)**



## Betriebssysteme implementieren häufig mehrere Schedulingverfahren

- In Linux z.B. gehört jeder Prozess zu einer bestimmten Prozessklasse
- Für „**Echtzeitprozesse**“...
  - SCHED\_FIFO (Prioritätengesteuertes Scheduling, nicht-unterbrechend)
  - SCHED\_RR (unterbrechend)
  - SCHED\_DEADLINE (EDF-Scheduling, unterbrechend)
- Für „**normale**“ Prozesse...
  - SCHED\_OTHER (Time-Sharing-Betrieb normaler Prozesse) realisiert als...
    - Multilevel Feedback Scheduling (bis Kernel 2.4)
    - O(1)-Scheduler (Kernel 2.6.0 bis 2.6.22)
    - Completely Fair Scheduler (seit Kernel 2.6.23)

```
$ ps a | grep okular
359675 pts/2    Sl          0:04 okular  bts_WS2122_slideset_08_en.pdf
$ chrt -p 361443
pid 361443's current scheduling policy: SCHED_OTHER
pid 361443's current scheduling priority: 0
```

```
SCHED_OTHER:      chrt -o -p PRIO PID
SCHED_FIFO:       chrt -f -p PRIO PID
SCHED_RR:         chrt -r -p PRIO PID
SCHED_DEADLINE:  chrt -d -sched-runtime NS -sched-deadline NS -sched-period NS 0 PID
```

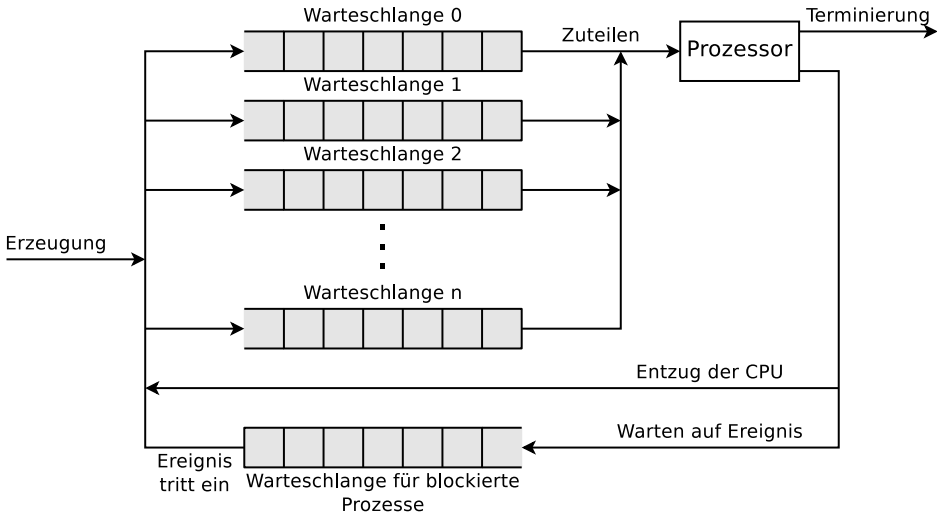
„A SCHED\_DEADLINE task should receive *runtime* microseconds of execution time every *period* microseconds, and these *runtime* microseconds are available within *deadline* microseconds from the beginning of the period.“

Quelle: <https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt>

# Prioritätengesteuertes Scheduling

- Prozesse werden nach ihrer Priorität (= Wichtigkeit bzw. Dringlichkeit) abgearbeitet
- Es wird immer dem Prozess im Zustand bereit die CPU zugewiesen, der die höchste Priorität hat
  - Die Priorität kann von verschiedenen Kriterien abhängen, z.B. statische (zugewiesene) Prioritätsstufe, benötigte Ressourcen, Rang des Benutzers, geforderte Echtzeitkriterien, usw.
- Kann **präemptiv** (verdrängend) und **nicht-präemptiv** (nicht-verdrängend) sein
- Die Prioritätenvergabe kann **statisch** oder **dynamisch** sein
  - Statische Prioritäten ändern sich während der gesamten Lebensdauer eines Prozesses nicht und werden häufig in Echtzeitsystemen verwendet
  - Dynamische Prioritäten werden von Zeit zu Zeit angepasst  
⇒ **Multilevel-Feedback Scheduling** (siehe Folie 21)
- Gefahr beim (statischen) prioritätengesteuertem Scheduling: Prozesse mit niedriger Priorität können verhungern (⇒ **nicht fair**)
- Prioritätengesteuertes Scheduling eignet sich für interaktive Systeme

# Prioritätengesteuertes Scheduling

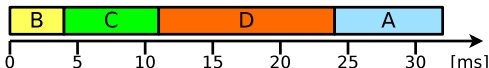


Quelle: William Stallings. Betriebssysteme. 4. Auflage. Pearson (2003). S.465

# Beispiel zum Prioritätengesteuerten Scheduling

- Auf einem Einprozessorrechner (mit nur einem CPU-Kern) sollen 4 Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand bereit
- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)

Prozess	Rechenzeit	Priorität
A	8 ms	3
B	4 ms	15
C	7 ms	8
D	13 ms	4



- Die Rechenzeit ist die Zeit, die der Prozess Zugriff auf die CPU benötigt, um komplett abgearbeitet zu werden
- Laufzeit = „Lebensdauer“ = Zeitspanne zwischen dem Anlegen und Beenden eines Prozesses = (Rechenzeit + Wartezeit)

Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	32	4	11	24

Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	24	0	4	11

$$\text{Durchschn. Laufzeit} = \frac{32+4+11+24}{4} = 17,75 \text{ ms}$$

$$\text{Durchschn. Wartezeit} = \frac{24+0+4+11}{4} = 9,75 \text{ ms}$$

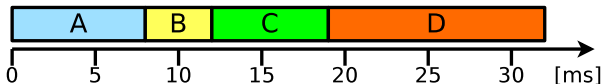
# First Come First Served (FCFS)

- Funktioniert nach dem Prinzip **First In First Out** (FIFO)
- Die Prozesse bekommen die CPU entsprechend ihrer Ankunftsreihenfolge zugewiesen
- Laufende Prozesse werden nicht unterbrochen
  - Es handelt sich um **nicht-präemptives** (nicht-verdrängendes) Scheduling
- FCFS ist **fair**
  - Alle Prozesse werden berücksichtigt
- Die **mittlere Wartezeit kann unter Umständen sehr hoch sein**
  - Prozesse mit kurzer Abarbeitungszeit müssen eventuell lange warten, wenn vor ihnen Prozesse mit langer Abarbeitungszeit eingetroffen sind
- FCFS/FIFO eignet sich für Stapelverarbeitung ( $\implies$  Foliensatz 1)
- FIFO wird in Linux für nicht-präemptive „Echtzeitprozesse“ verwendet

# Beispiel zu First Come First Served

- Auf einem Einprozessorrechner (mit nur einem CPU-Kern) sollen 4 Prozesse verarbeitet werden
- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)

Prozess	Rechenzeit	Ankunftszeit
A	8 ms	0 ms
B	4 ms	1 ms
C	7 ms	3 ms
D	13 ms	5 ms



- Die Rechenzeit ist die Zeit, die der Prozess Zugriff auf die CPU benötigt, um komplett abgearbeitet zu werden
- Laufzeit = „Lebensdauer“ = Zeitspanne zwischen dem Anlegen und Beenden eines Prozesses = (Rechenzeit + Wartezeit)

## Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	8	11	16	27

$$\text{Durchschn. Laufzeit} = \frac{8+11+16+27}{4} = 15,5 \text{ ms}$$

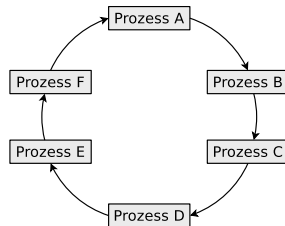
## Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	0	7	9	14

$$\text{Durchschn. Wartezeit} = \frac{0+7+9+14}{4} = 7,5 \text{ ms}$$

# Round Robin (RR) – Zeitscheibenverfahren (1/2)

- Es werden Zeitscheiben (*Time Slices*) mit einer festen Dauer festgelegt
- Die Prozesse werden in einer zyklischen Warteschlange nach dem FIFO-Prinzip eingereiht
  - Der erste Prozess der Warteschlange erhält für die Dauer einer Zeitscheibe Zugriff auf die CPU
  - Nach dem Ablauf der Zeitscheibe wird diesem der Zugriff auf die CPU wieder entzogen und er wird am Ende der Warteschlange eingereiht
  - Wird ein Prozess erfolgreich beendet, wird er aus der Warteschlange entfernt
    - Neue Prozesse werden am Ende der Warteschlange eingereiht
- Die Zugriffszeit auf die CPU wird **fair** auf die Prozesse aufgeteilt
- RR mit Zeitscheibengröße  $\infty$  verhält sich wie FCFS



## Round Robin (RR) – Zeitscheibenverfahren (2/2)

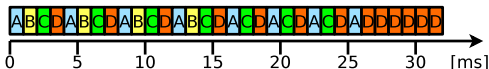
- Je länger die Bearbeitungsdauer eines Prozesses ist, desto mehr Runden sind für seine vollständige Ausführung nötig
- Die Größe der Zeitschlitze ist wichtig für die Systemgeschwindigkeit
  - Je kürzer sie sind, desto mehr Prozesswechsel müssen stattfinden  
⇒ Hoher Overhead
  - Je länger sie sind, desto mehr geht die Gleichzeitigkeit verloren  
⇒ Das System hängt/*ruckelt*
- Die Größe der Zeitschlitze liegt üblicherweise im ein- oder zweistelligen Millisekundenbereich
- **Bevorzugt Prozesse, die eine kurze Abarbeitungszeit haben**
- **Präemptives (verdrängendes) Scheduling-Verfahren**
- Round Robin Scheduling eignet sich für interaktive Systeme
- Round Robin wird in Linux für präemptive „Echtzeitprozesse“ verwendet



# Beispiel zu Round Robin

- Auf einem Einprozessorrechner (mit nur einem CPU-Kern) sollen 4 Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand bereit
- Zeitquantum  $q = 1$  ms
- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)

Prozess	Rechenzeit
A	8 ms
B	4 ms
C	7 ms
D	13 ms



- Die Rechenzeit ist die Zeit, die der Prozess Zugriff auf die CPU benötigt, um komplett abgearbeitet zu werden
- Laufzeit = „Lebensdauer“ = Zeitspanne zwischen dem Anlegen und Beenden eines Prozesses = (Rechenzeit + Wartezeit)

## Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	26	14	24	32

$$\text{Durchschn. Laufzeit} = \frac{26+14+24+32}{4} = 24 \text{ ms}$$

## Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	18	10	17	19

$$\text{Durchschn. Wartezeit} = \frac{18+10+17+19}{4} = 16 \text{ ms}$$

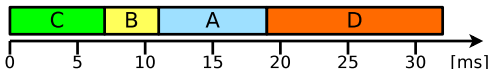
# Earliest Deadline First (EDF)

- Ziel: Prozesse sollen nach Möglichkeit ihre Termine zur Fertigstellung (Deadlines) einhalten
- Prozesse im Zustand bereit werden aufsteigend **nach ihrer Deadline geordnet**
  - Der Prozess, dessen Deadline am nächsten ist, bekommt die CPU zugewiesen
- Eine Überprüfung und gegebenenfalls Neuorganisation der Warteschlange findet statt, wenn...
  - ein neuer Prozess in den Zustand bereit wechselt
  - oder ein aktiver Prozess terminiert
- Kann als **präemptives und nicht-präemptives Scheduling** realisiert werden
  - Präemptives EDF eignet sich für Echtzeitbetriebssysteme
  - Nicht-präemptives EDF eignet sich für Stapelverarbeitung
- EDF wird in Linux für präemptive „Echtzeitprozesse“ verwendet

# Beispiel zu Earliest Deadline First

- Auf einem Einprozessorrechner (mit nur einem CPU-Kern) sollen 4 Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand bereit
- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)

Prozess	Rechenzeit	Deadline
A	8 ms	25
B	4 ms	18
C	7 ms	9
D	13 ms	34



- Die Rechenzeit ist die Zeit, die der Prozess Zugriff auf die CPU benötigt, um komplett abgearbeitet zu werden
- Laufzeit = „Lebensdauer“ = Zeitspanne zwischen dem Anlegen und Beenden eines Prozesses = (Rechenzeit + Wartezeit)

## Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	19	11	7	32

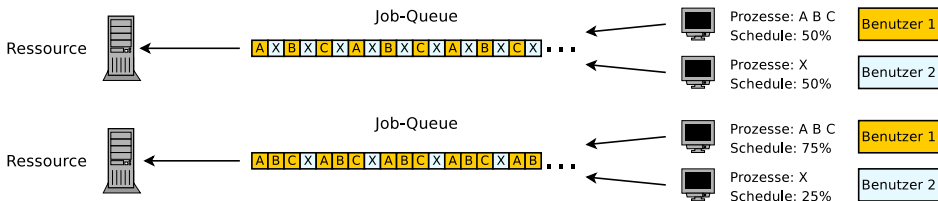
$$\text{Durchschn. Laufzeit} = \frac{19+11+7+32}{4} = 17,25 \text{ ms}$$

## Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	11	7	0	19

$$\text{Durchschn. Wartezeit} = \frac{11+7+0+19}{4} = 9,25 \text{ ms}$$

# Fair-Share



- Bei **Fair-Share** werden Ressourcen zwischen Gruppen von Prozessen in einer fairen Art und Weise aufgeteilt
- Besonderheit:
  - Die Rechenzeit wird den Benutzern und nicht den Prozessen zugeteilt
  - Die Rechenzeit, die ein Benutzer erhält, ist unabhängig von der Anzahl seiner Prozesse
- Die Ressourcenanteile, die die Benutzer erhalten, heißen Shares

Fair-Share wird häufig in Cluster- und Grid-Systemen eingesetzt

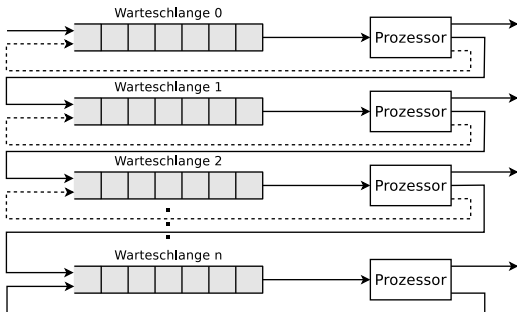
Fair-Share wird in Job-Schedulern und Meta-Schedulern (z.B. SUN/Oracle/Univa/Altair Grid Engine) zur Verteilung der Aufträge auf Ressourcen in Grid-Standorten und zwischen den Standorten in Grids eingesetzt

# Multilevel-Feedback-Scheduling (1/2)

- Es ist **unmöglich, die Rechenzeit verlässlich im voraus zu kalkulieren**
  - Lösung: Prozesse, die schon länger aktiv sind, werden **bestraft**
- **Multilevel-Feedback-Scheduling** arbeitet mit mehreren Warteschlangen
  - Jede Warteschlange hat eine andere Priorität oder Zeitmultiplex (z.B. 70%:15%:10%:5%)
- Jeder neue Prozess kommt in die oberste Warteschlange
  - Damit hat er die höchste Priorität
- Innerhalb jeder Warteschlange wird Round Robin eingesetzt
  - Gibt ein Prozess die CPU freiwillig wieder ab, wird er wieder in die selbe Warteschlange eingereiht
  - Hat ein Prozess seine volle Zeitscheibe genutzt, kommt er in die nächst tiefere Warteschlange mit einer niedrigeren Priorität
    - Die Prioritäten werden bei diesem Verfahren also **dynamisch** vergeben
- Multilevel-Feedback-Scheduling ist **unterbrechendes Scheduling**

## Multilevel-Feedback-Scheduling (2/2)

- Vorteil:
  - **Keine komplizierten Abschätzungen!**
    - Neue Prozesse werden schnell in eine Prioritätsklasse eingeordnet
  - **Bevorzugt neue Prozesse gegenüber älteren** (länger laufenden) Prozessen
  - Prozesse mit vielen Ein-/Ausgabeoperationen werden bevorzugt, weil sie nach einer freiwilligen Abgabe der CPU wieder in die ursprüngliche Warteliste eingeordnet werden  $\Rightarrow$  Dadurch behalten Sie ihre Priorität
  - Ältere, länger laufende Prozesse werden verzögert



Quelle: William Stallings. Betriebssysteme. 4. Auflage. Pearson (2003). S.479

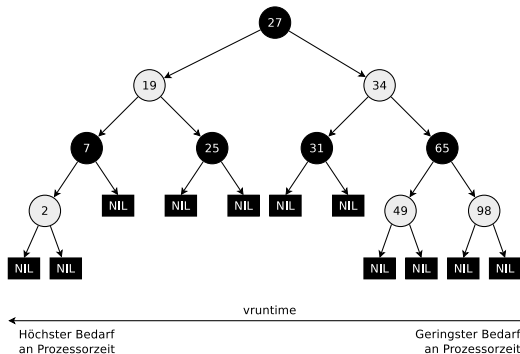
Viele moderne Betriebssysteme verwenden für das Scheduling der Prozesse Varianten des Multilevel-Feedback-Scheduling. Beispiele: Linux für „normale“ Prozesse (bis Kernel 2.4), Mac OS X, FreeBSD und die Windows NT-Familie

# Completely Fair Scheduler (Linux seit 2.6.23) – Teil 1/4

- Der Kernel realisiert für jeden CPU-Kern einen CFS-Scheduler und verwaltet für jeden `SCHED_OTHER` Prozess eine Variable ***vruntime*** (virtual runtime)

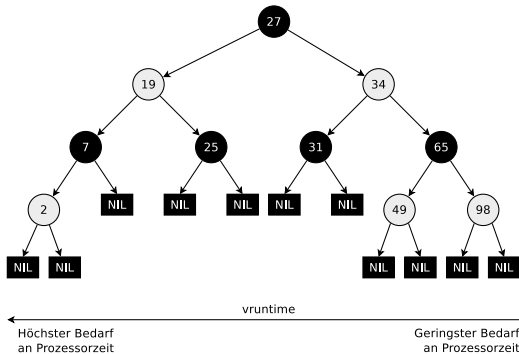
- Der Wert repräsentiert eine virtuelle Prozessorlaufzeit in Nanosekunden

- `vruntime` sagt aus, wie lange der jeweilige Prozess schon gerechnet hat
  - Der Prozess mit der niedrigsten `vruntime` bekommt als nächstes Zugriff auf den CPU-Kern
- Die Verwaltung der Prozesse geschieht mit Hilfe eines **Rot-Schwarz-Baums** (selbstbalancierender binärer Suchbaum)
  - Die Prozesse sind anhand der `vruntime`-Werte einsortiert



# Completely Fair Scheduler (Linux seit 2.6.23) – Teil 2/4

- Ziel: Alle Prozesse, die einem CPU-Kern zugeordnet sind, sollen einen ähnlich großen (fairen) Anteil Rechenzeit erhalten  
 $\Rightarrow$  Bei  $n$  Prozessen soll jeder Prozess  $1/n$  der Rechenzeit erhalten



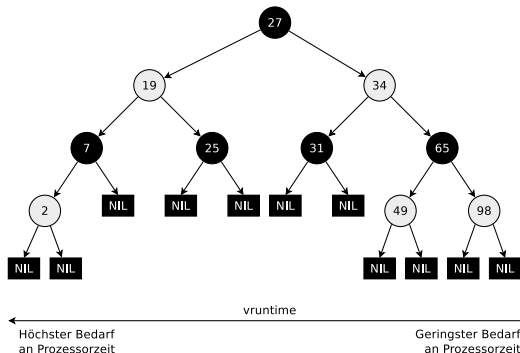
- Hat ein Prozess Zugriff auf den CPU-Kern, darf er so lange rechnen, bis sein vruntime-Wert sich dem angestrebten Anteil von  $1/n$  der verfügbaren Rechenzeit angenähert hat
- Der Scheduler strebt einen gleichen vruntime-Wert für alle Prozesse an

Der CFS-Scheduler kümmert sich nur um die Ablaufplanung der „normalen“ Prozesse, die der Prozessklasse `SCHED_OTHER` zugeordnet sind



# Completely Fair Scheduler (Linux seit 2.6.23) – Teil 3/4

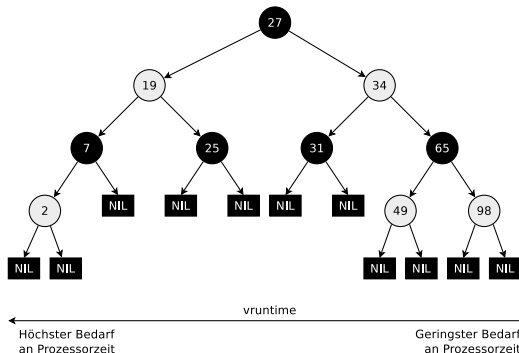
- Die Werte sind die Schlüssel der inneren Knoten
- Blattknoten (NIL-Knoten) haben keine Schlüssel und enthalten keine Daten
- NIL steht für *none*, *nothing*, *null*, also für einen Null-Wert oder Null-Pointer



- Aus Fairnessgründen weist der Scheduler dem Prozess ganz links im Baum als nächstes den CPU-Kern zu
- Wird ein Prozess vom CPU-Kern verdrängt, erhöht sich der vruntime-Wert um die Zeit, die der Prozess auf dem CPU-Kern gelaufen ist

# Completely Fair Scheduler (Linux seit 2.6.23) – Teil 4/4

- Die Knoten (Prozesse) im Baum wandern kontinuierlich von rechts nach links  
 $\Rightarrow$  Das gewährleistet die faire Verteilung der Rechenleistung



- Der Scheduler berücksichtigt die statischen Prozessprioritäten (nice-Werte) der Prozesse
- Die vruntime-Werte werden abhängig vom nice-Wert unterschiedlich gewichtet
  - Anders gesagt: Die virtuelle Uhr kann unterschiedlich schnell laufen

# Klassische und moderne Scheduling-Verfahren

	Scheduling NP	P	Fair	Rechenzeit muss bekannt sein	Berücksichtigt Prioritäten
Prioritätengesteuertes Scheduling	X	X	nein	nein	ja
First Come First Served = FIFO	X		ja	nein	nein
<del>Last Come First Served</del>	<del>X</del>	<del>X</del>	<del>nein</del>	<del>nein</del>	<del>nein</del>
Round Robin		X	ja	nein	nein
<del>Shortest/Longest Job First</del>	<del>X</del>		<del>nein</del>	<del>ja</del>	<del>nein</del>
Shortest Remaining Time First		X	nein	ja	nein
Longest Remaining Time First		X	nein	ja	nein
Highest Response Ratio Next	X		ja	ja	nein
Earliest Deadline First	X	X	ja	nein	nein
Fair-Share		X	ja	nein	nein
<del>Statisches Multilevel-Scheduling</del>		<del>X</del>	<del>nein</del>	<del>nein</del>	<del>ja (statisch)</del>
Multilevel-Feedback-Scheduling		X	ja	nein	ja (dynamisch)
<del>O(1)-Scheduler</del>		<del>X</del>	<del>ja</del>	<del>nein</del>	<del>ja</del>
Completely Fair Scheduler		X	ja	nein	ja

- NP = Nicht-präemptives Scheduling, P = Präemptives Scheduling
- Ein Schedulingverfahren ist „fair“, wenn jeder Prozess irgendwann Zugriff auf die CPU erhält
- Es ist unmöglich, die Rechenzeit verlässlich im voraus zu kalkulieren

Linux 2.6.0 bis 2.6.22 verwendet den **O(1)-Scheduler**. Dieser spielt aus Zeitgründen hier keine Rolle  
<https://www.ibm.com/developerworks/library/l-scheduler/index.html>