Betriebssysteme Vorlesung 7

Scheduling bei Einprozessor- und Multiprozessorarchitekturen

Realtime Scheduling

Literatur: Stallings Kapitel 9,10

Übersicht

- Scheduling für Einprozessorarchitekturen
  - Unterscheidung nach Zeitskalen
  - Algorithmen für kurzfristiges Scheduling (Dispatching)
    - FCFS, RR, SPN, SRT, HRRN, FB
  - Bsp: traditionelles Unix Scheduling
- Scheduling für Mehrprozessorarchitekturen
- Realtime Scheduling
- Bsp
  - Linux, Unix SVR4, Windows 2000

Nutzen

- Sie lernen Algorithmen zum Scheduling kennen, die in realen Betriebssystemen eingesetzt werden.
- Sie können dadurch die Möglichkeiten der Einflußnahme auf die Laufzeit eines Prozesses einschätzen und gezielt nutzen.

CPU Scheduling

- Langfristiges Scheduling
  - entscheidet, ob neue Prozesse begonnen werden, d.h. in die Menge der laufenden Prozesse aufgenommen werden selten aufgerufen
- Mittelfristiges Scheduling
  - entscheidet, welche Prozesse teilweise oder vollständig im Hauptspeicher liegen. Teil der Swapping Funktion. manchmal aufgerufen

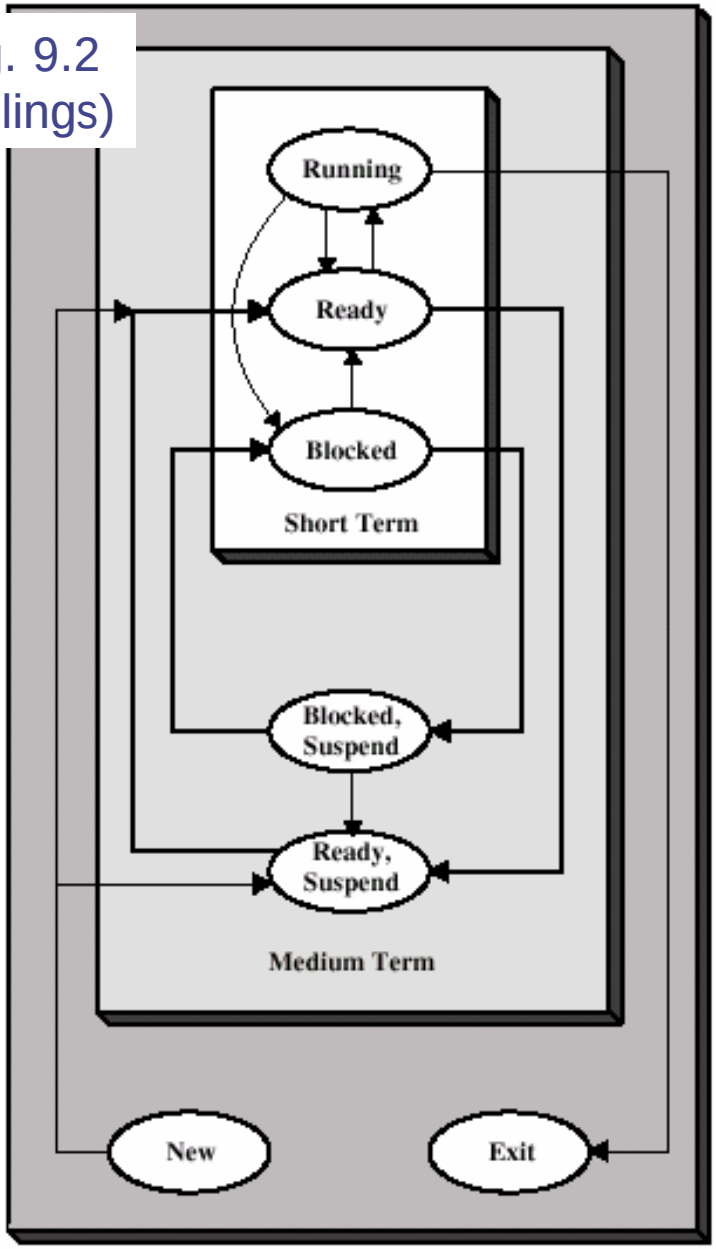
Lang- & mittelfristiges Scheduling bestimmt den Grad des Multiprogramming und hängt im wesentlichen von der Speicherbelegungssituation und der CPU Auslastung ab.

- Kurzfristiges Scheduling, Dispatching häufig aufgerufen
  - entscheidet, welche Prozesse durch die CPU bearbeitet werden
- E/A Scheduling
  - entscheidet, welcher von allen Prozessen mit E/A Anfragen durch ein E/A Gerät bearbeitet werden soll

Scheduling ist im wesentlichen die Aufgabe eine Reihe von Warteschlangen so zu verwalten, das Wartezeiten minimal werden und Performance / Durchsatz maximal wird.

CPU Scheduling: Aufteilung der Zustände nach Scheduling

(Fig. 9.2 Stallings)



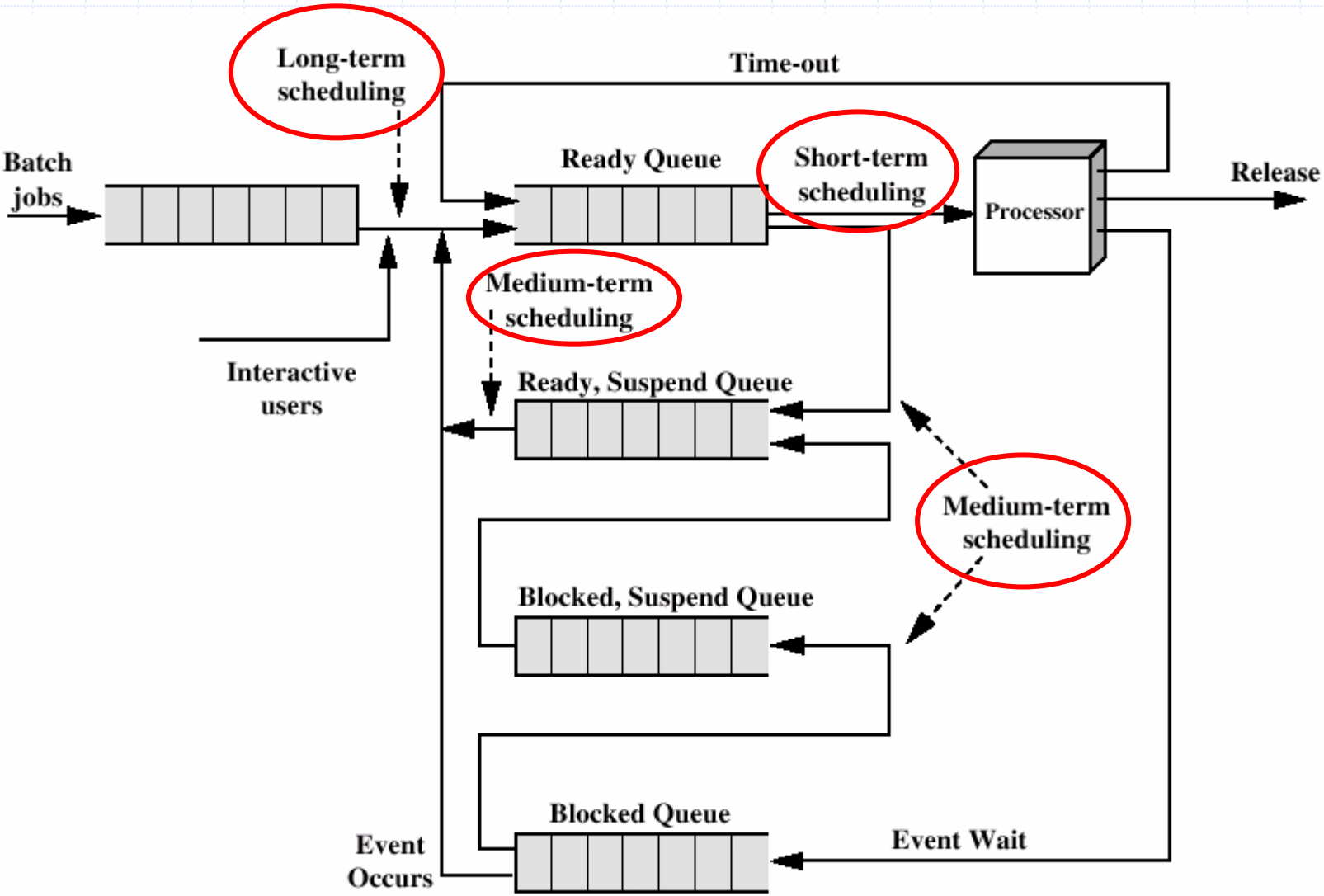
Der langfristige Scheduler steuert den Grad des Multiprogramming. Bei Batch Processing ist die Menge anstehender Aufgaben vorab bekannt, bzw neue Jobs werden zunächst auf der Festplatte gelagert ohne ein Prozeßimage zu erzeugen. Er wird aktiv, wenn neue Jobs eintreffen, ein bestehender terminiert oder die CPU einige Zeit idle ist.

Für eine gleichmäßige Ressourcennutzung kann eine Mischung aus E/A-lastigen und CPU-lastigen Prozessen erwünscht sein.

Der mittelfristige Scheduler ist Teil der Swap Funktion und reguliert ebenfalls den Grad des Multiprogramming.

Der kurzfristige Scheduler (Dispatcher) wird häufig aktiviert: bei Clock, E/A Interrupts, Systemaufrufen und Signalen.

Warteschlangennetz



(Stallings,Fig. 9.3)

Scheduling Algorithmen für kurzfristiges Scheduling

Kriterien

anwenderorientiert

- Turnaround Time, Umlaufzeit (für Batch Jobs)
- Antwortzeiten bei interaktiven Prozessen, z.B. < 2 sec
- Deadlines/Endtermine
- Vorhersagbarkeit

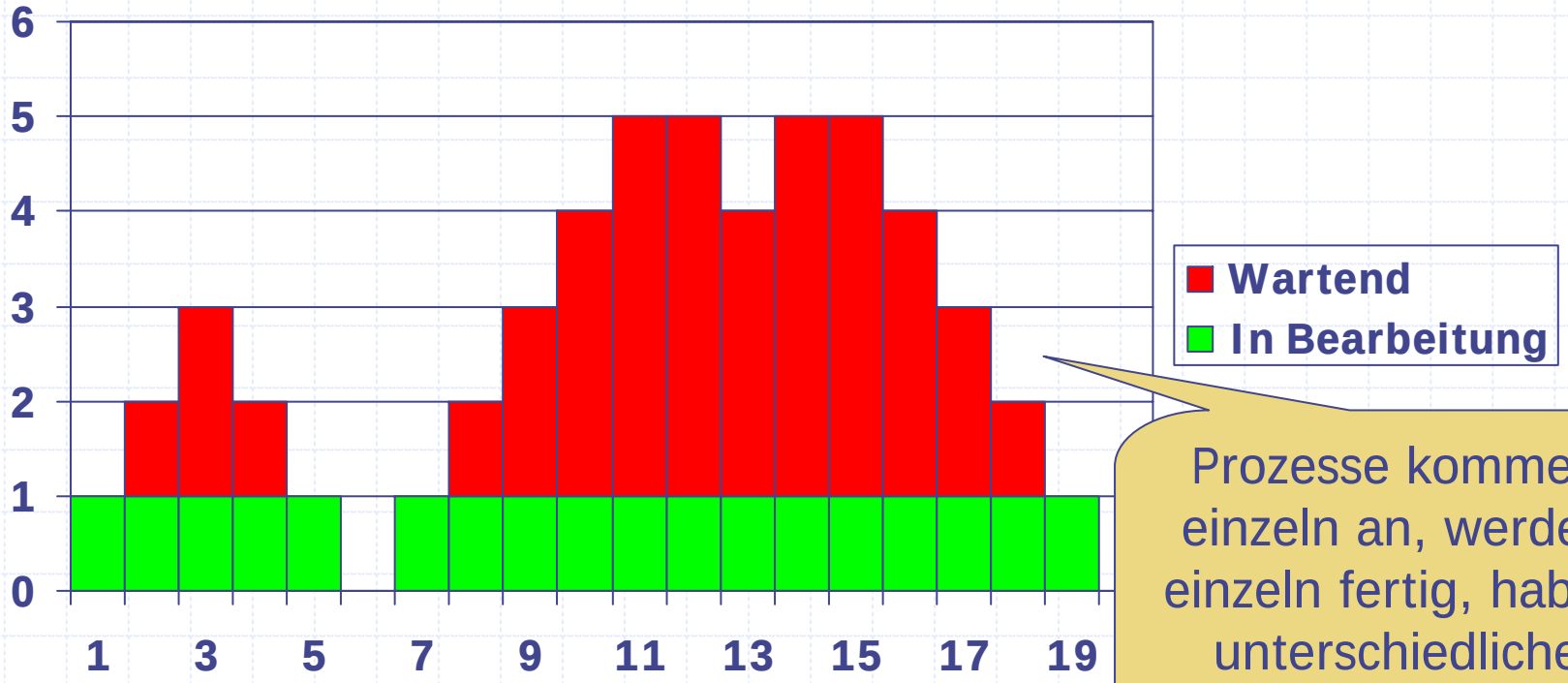
systemorientiert (bei Mehrbenutzerbetrieb interessant)

- Durchsatz
- CPU Auslastung
- Fairness
- Berücksichtigung von Prioritäten
- Ausgeglichene Nutzung von Ressourcen

Möglichkeiten

- Prioritäten
- Unterbrechungen

Was können Scheduling Algorithmen beeinflussen ?



Prozesse kommen einzeln an, werden einzeln fertig, haben unterschiedliche Bearbeitungsdauern.

Beobachtung:

- 1 Prozess wird jeweils bearbeitet, n-1 Prozesse warten parallel.
- Anzahl grüner Felder ist stets gleich der Gesamtmenge Arbeit.

Ziel: Wartezeit insgesamt (rote Fläche) sollte klein sein.

Scheduling: Auswahl des bearbeiteten (grünen) Prozesses

Chance:

bei Auswahl eines kurzen Prozesses wird die Anzahl Wartender schneller kleiner, Gesamtwartezeit sinkt

Scheduling Algorithmen für kurzfristiges Scheduling

- FCFS: First-come-first-served
  - der Prozeß mit längster Wartezeit in der Ready Queue wird solange ausgeführt, bis er blockiert
- RR: Round robin
  - Zeitscheibenverfahren, Prozesse werden reihum jeweils für eine festgelegte Zeitspanne (Zeitquantum) bearbeitet
- SPN: Shortest process next
  - der Prozeß mit der kürzesten zu erwartenden Bearbeitungszeit wird ohne Unterbrechung abgearbeitet
- SRN: Shortest remaining time
  - unterbrechende Version von SPN, wählt Prozeß mit der kürzesten zu erwartenden Restbearbeitungszeit
- HRRN: Highest response ratio next
  - wählt den Prozeß mit größtem Wert für (w+s)/s, wobei w=Wartezeit, s=erwartete Bedienzeit ist
- FB: Feedback
  - je länger Prozesse laufen, desto geringer wird ihre Priorität. FB wählt innerhalb von Prioritätsklassen nach FCFS aus, auf unterster Stufe dann RR, um Verhungern auszuschliessen.

FCFS Besonderheiten

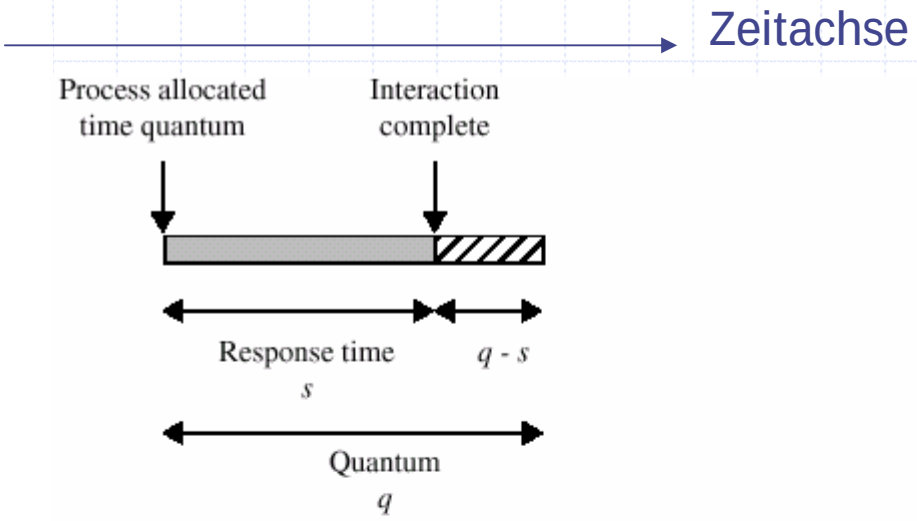
- im täglichen Leben üblicherweise eingesetzte Entscheidungsregel, Prioritäten/Klassen widersprechen dem Menschenbild (Liberte,Egalite,Fraternite)
- in Rechensystemen zeigen sich folgende Effekte
  - die normalisierte Umlaufzeit: 1 + Wartezeit / Rechenzeit zeigt, dass arbeitsaufwändige Prozesse gegenüber Kurzläufern begünstigt werden.
  - CPU-lastige Prozesse werden gegenüber E/A-lastigen bevorzugt, weil bei E/A Aufrufen eine Unterbrechung und Prozesswechsel erfolgt
  - dadurch werden E/A Geräte auch bei ausreichender Menge an Prozessen nur unzureichend genutzt

daher: FCFS für CPU Scheduling nur in Verbindung mit Prioritätsscheduling und weiteren Regeln,

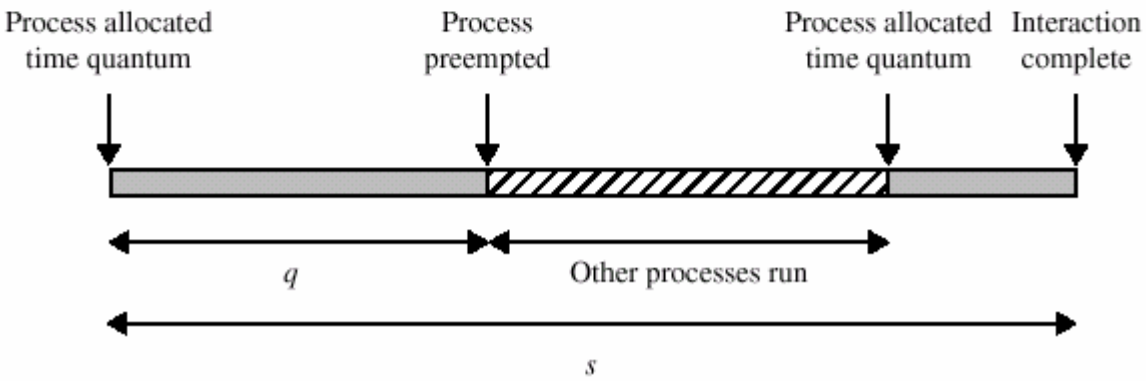
-> z.B. Feedback Scheduling (mit Unterbrechungen, später)

Round Robin Besonderheiten

- Prozesse werden spätestens nach K Zeiteinheiten unterbrochen, aus der Ready Queue wird ein Nachfolger gemäß FCFS ausgewählt: Zeitscheibenverfahren
- bewirkt kurze Antwortzeiten für Kurzläufer
- erfordert passende Wahl des Zeitquantums, Wert sollte nah an typischer Interaktionszeit liegen
- CPU-lastige Prozesse werden bevorzugt
- Fairness wird ggfs durch dynamische Prioritäten ausgeglichen



a) Zeitquantum > Interaktionszeit
b) Zeitquantum < Interaktionszeit



Shortest Process Next (SPN) : Besonderheiten

- führt zu besseren mittleren Umlaufzeiten als FCFS, weil kurze Jobs bevorzugt werden, dafür steigt jedoch die Varianz (Vorhersagbarkeit wird dadurch schlechter)
- Problem: Kenntnis der Bearbeitungszeit
  - Klassenbildung je nach Art der Anwendung
  - je Klasse statistischer Schätzwert durch „exponentielle Glättung“

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

Bearbeitungszeit des n-ten Jobs dieser Klasse
Gewichtungsfaktor zwischen 0 und 1
Schätzwert nach n+1 Beobachtungen

- Risiko: Verhungern besonders aufwändiger Jobs
- Wg der fehlenden Unterbrechungen ist SPN für interaktive und transaction-processing Systeme ungünstig. Abhilfe: Shortest Remaining Processing Time (SRT) ist unterbrechende Variante von SPN bzgl geschätzter Restbearbeitungszeit.

Highest Response Ratio Next: Besonderheiten

- wählt den Prozeß mit der größten Response Ratio, d.h. der größten Wartezeit im Verhältnis zur erwarteten Bedienzeit aus $R=(w+s)/s=1+(w/s)$ wenn der aktuelle Prozeß terminiert oder blockiert

  w: Wartezeit
  s: erwartete Bedienzeit

- kürzere Jobs werden ebenso bevorzugt wie aufwändige, die lange gewartet haben

- SPN, SRN und HRRN erfordern es Bedienzeiten zu schätzen

Feedback: Besonderheiten

Strategie betrachtet bereits verbrauchte Rechenzeit

- es gibt n+1 Warteschlangen mit fallender Priorität
- neue Prozesse beginnen mit der höchsten Priorität i=0 in Schlange 0
- nach jeder Unterbrechung sinkt ein Prozeß in der Priorität von i auf i+1 ab, wandert also in die Warteschlange mit Index i+1
- Prozesse werden nach Priorität und FCFS in den Warteschlangen 0,1,..., n-1 ausgewählt und für Zeitquantum bearbeitet (preempt)
- die Schlange mit niedrigster Priorität wird mit RR bearbeitet

- kurze Prozesse werden bevorzugt
- je größer die Bearbeitungszeit, desto länger wird die Wartezeit
- in letzter Schlange extreme Wartezeiten verhindert, falls mit fallender Priorität die Zeitscheiben größer werden
- Um Verhungern zu verhindern, kann ein Prozeß in der letzten Schlange nach Überschreiten eines Grenzwertes in seiner Priorität wieder ansteigen.

Charakterisierung von Scheduling Algorithmen

	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
FCFS	max[w]	Nonpreemptive	Not emphasized	May be high, especially if there is a large variance in process execution times	Minimum	Penalizes short processes; penalizes I/O bound processes	No
Round Robin	constant	Preemptive (at time quantum)	May be low if quantum is too small	Provides good response time for short processes	Minimum	Fair treatment	No
SPN	min[s]	Nonpreemptive	High	Provides good response time for short processes	Can be high	Penalizes long processes	Possible
SRT	min[$s - e$]	Preemptive (at arrival)	High	Provides good response time	Can be high	Penalizes long processes	Possible
HRRN	$\max\left(\frac{w+s}{s}\right)$	Nonpreemptive	High	Provides good response time	Can be high	Good balance	No
Feedback	(see text)	Preemptive (at time quantum)	Not emphasized	Not emphasized	Can be high	May favor I/O bound processes	Possible

w = time spent in system so far, waiting and executing
e = time spent in execution so far
s = total service time required by the process, including e

Fair share Scheduling

- Prozesse werden nach Anwendern gruppiert
  - Prozesse eines einzelnen Anwenders, einer Anwendergruppe
- fair = gerecht in Bezug auf eine Gruppe
  - z.B. 4 Gruppen, je Gruppe 25% der CPU Zeit
- wird mit dynamischen Prioritäten und Zeitscheiben realisiert
  - jeder Prozeß hat eine Basispriorität $Base_j$
  - der aktive Prozeß erhöht den Prioritätswert für sich und seine Gruppe nach folgenden Formeln

$$CPU_j(i) = CPU_j(i-1)/2$$

$$GCPU_k(i) = GCPU_k(i-1)/2$$

$$\mathrm{Prio}_j(i) = \mathrm{Base}_\mathrm{j} + CPU_j(i)/2 + GCPU_k(i)/2 \cdot W_k$$

Werte am Ende der Phase i-1

Werte zu Beginn einer Phase i, Prozeß j gehört zu Gruppe k

Beispiel für Fair Scheduling

jede Gruppe erhält Gewicht w=0,5

Werte in Phase 3

Prozeß A:

$CPU_j(i)$: 75 / 2 =37

$GCPU_j(i)$:75 / 2=37

Prio:

60 + 37/2 + 37 / (4*0,5)= 97

Prozesse werden reihum A B A C A B … bedient, so dass {A} 50% CPU, {B,C} 50% CPU bekommen

Time	Process A			Process B			Process C		
	Priority	Process	Group	Priority	Process	Group	Priority	Process	Group
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		•	•						
		•	•						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					•	•			•
					•	•			•
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		•	•						
		•	•						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						•		•	•
						•		•	•
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		•	•						
		•	•						
		78	78						
5	98	39	39	70	3	18	76	15	18

Group 1 Group 2

Zwischenfazit

- Scheduling nach Zeitskalen
  - langfristig, mittelfristig, kurzfristig
- Dispatching mit unterschiedlichen Algorithmen
  - FCFS
  - RR
  - SPN, SRT
  - HRRN
  - FB

  legen Vorzug auf Prozesse mit langen Laufzeiten (FCFS) oder kurzen Laufzeiten (SRT) oder behandeln alle gleichmäßig (RR)

  erfordern Kenntnis/Schätzung von Laufzeiten (SPN,SRT,HRRN)
- im folgenden
  - Beispiel: traditionelles UNIX Scheduling
  - Multiprozessorscheduling
  - Realtime Scheduling

Traditionelles Unix Scheduling (Unix SV R3, BSD 4.3)

- auf interaktive Time-Sharing Umgebungen abgestimmt
- Ziel: gute Antwortzeiten für interaktive Anwender
  - Nebenbedingung: Hintergrund Jobs mit niedriger Priorität dürfen nicht verhungern
- Algorithmus: Multi-Level Feedback mit RR je Ebene
  - 1s Zeitquantum
  - Priorität hängt von der Art des Prozesse und der Historie ab

$$CPU_j(i) = CPU_j(i-1)/2$$

$$\mathrm{Prio}_j(i) = \mathrm{Base}_j + CPU_j(i)/2 + nice_j$$

kann vom Anwender gesetzt werden

Die Basispriorität dient der Einteilung in Prozeßklassen „Bänder“:
1) Swapper, 2) Blockweise E/A Gerätesteuerung,
3) Dateimanipulation, 4) Zeichenweise E/A Gerätesteuerung,
5) Anwenderprozesse

Beispiel-ablauf bei traditionellem Unix Scheduling, E/A-lastige Prozesse werden gegenüber CPU-lastigen leicht bevorzugt, dadurch wird eine bessere E/A Geräte Auslastung erzielt.

Time	Process A		Process B		Process C	
	Priority	CPU Count	Priority	CPU Count	Priority	CPU Count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

0.5*60

60+0.5*30

Multiprozessorscheduling

- Unterschiede in Architekturen mit mehreren Prozessoren
  - loose gekoppelte Systeme, verteilte Systeme, Cluster
    (wird später unter „verteilte Systeme“ behandelt)
  - Architekturen mit Spezialprozessoren, z.B. für E/A
  - Architekturen mit eng-gekoppelten CPUs, die sich einen gemeinsamen Hauptspeicher teilen und unter der Kontrolle eines einzigen Betriebssystems stehen
- Multiprozessorarchitekturen sind von Interesse

  insbesondere bei der Bearbeitung paralleler Prozesse, die nicht völlig unabhängig voneinander sind

  Unterscheidung nach Häufigkeit der Synchronisation HS
  - fein: Parallelität innerhalb einer Instruktion, HS < 20 Instrukt.
  - medium: parallele Prozesse, Multithreading, HS 20-200 Instrukt.
    HIER hat Scheduling einen besonderen Einfluß!
  - grob: parallele Prozesse, Multiprogramming, HS 200-2000 Instr.
  - sehr grob: verteilte Prozesse in einem Netzwerk, HS 2000-1M
  - unabhängig: Prozesse, die nichts miteinander zutun haben

Design Aspekte

- Zuordnung von Prozessen zu Prozessoren
  - CPUs als Ressourcen Pool, nicht mehr so einzigartig wie zuvor
  - Prozesse können einer CPU statisch zugeordnet werden oder aus 1 Warteschlange stets neu verteilt werden
  - Master/Slave Design: 1 CPU nur für OS, andere leisten Arbeit

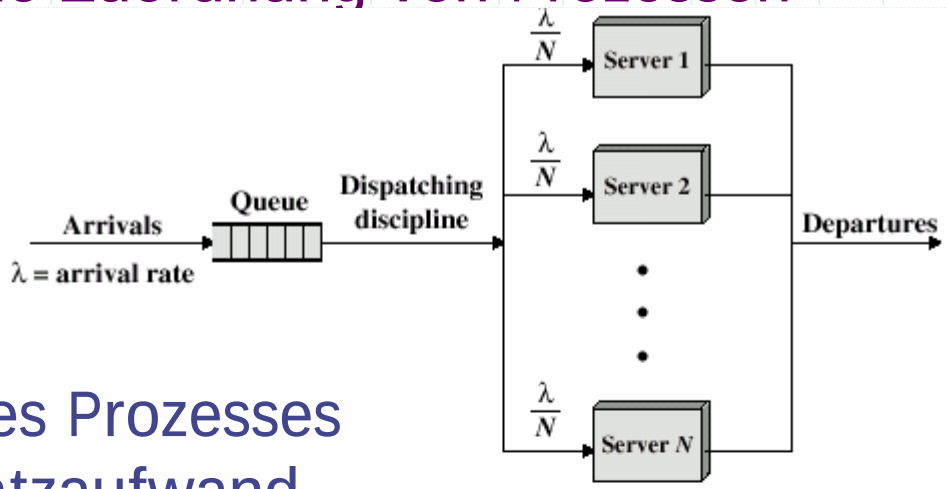
    pro: einfach, auch bzgl Ressourcen, contra: Ausfall, potent. Engpass
  - Peer Design: jede CPU kann OS ausführen, jede CPU betreibt eigenes Scheduling, wg Konflikten und verteilter Steuerung schwieriger
- Durchführung von Multiprogramming auf einzelnen CPUs
  - Ist dies bei mehreren CPUs noch sinnvoll ?

    bei grober Granularität sicher, aber bei mittlerer ?

    Paralleles Programm mit Threads erreicht gewünschten Speedup sicher nur bei echter Parallelverarbeitung!
- Dispatching von Prozessen
  - lohnt der Aufwand wie beim Uniprozessor oder sollte man Rechenzeit für Scheduling einsparen ? Nach Sauer,Chandy`81 nimmt Bedeutung von Scheduling mit CPU Zahl ab, FCFS ok.

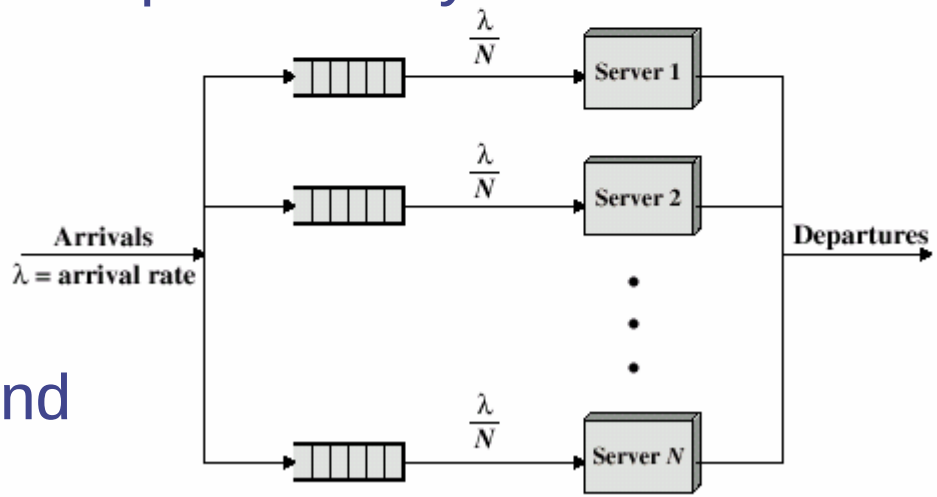
Statische oder dynamische Zuordnung von Prozessen

Dynamische Zuordnung:
- CPUs aus 1 Schlange versorgt
- bei Shared Memory: Kontextinformation eines Prozesses geteilt, daher kein Zusatzaufwand durch CPU-Wechsel eines Prozesses
- typisch für traditionelle Multiprozessorsysteme



(a) Multiserver queue

Feste Zuordnung:
- je CPU 1 Schlange
- ungleichmäßige Last 1 CPU überlastet während andere frei sind
- Gruppen oder Gang Scheduling möglich



(b) Multiple Single-server queues

Thread Scheduling

Scheduling Einfluß auf Speedup ist bei mittlerer Granularität hoch

Ansätze für Multiprozessor Thread Scheduling sind:

- Load sharing:
  - CPUs entnehmen bereite Prozesse/Threads aus 1 gemeinsamen Schlange,
  - Load Sharing vs Load Balancing, dort eher dauerhafte Zuordnung
  - gleichmäßige CPU Belastung, kein zentraler Scheduler,
  - Scheduling Strategien wie beim Uniprozessor plus weitere:
  - „FCFS“, alle Threads eines neuen Prozesses stellen sich am Ende an, besser als Varianten wie „Kleinste Anzahl unscheduled Threads zuerst“, mit oder ohne Unterbrechungen
  - <u>übliches Verfahren(!)</u>, mögliche Probleme: Schlange ist Bottleneck wg Synchronisation, Cacheing wg Threadwechsel zwischen CPUs sabotiert, nicht alle Threads eines Prozesses werden zusammen bearbeitet
  - Varianten: globale und zusätzliche lokale Run-Queues, z.B. in Mach
- Gang Scheduling: eine Menge zusammengehöriger Threads wird gemeinsam einer Menge CPUs 1-1 zugeordnet
- feste CPU Zuordnung: jeder Prozeß erhält Anzahl CPUs passend zur Anzahl Threads für die Laufzeit des Prozesses exklusiv
- dynamisches Scheduling: Anzahl Threads eines Prozesses darf zur Laufzeit variieren

Thread Scheduling

- Gang Scheduling
  - alle Threads eines Prozesses zusammen, 1-1 Prozessoren zugeordnet
  - parallele Bearbeitung von Threads kann Blockierungen wg Synchronisation verringern, Anzahl Thread/Prozeßwechsel reduzieren, Performance verbessern
  - Zeitscheiben bgzl Prozessen kann bei unterschiedlichem Bedarf an CPUs sehr ineffizient sein, weil CPUs dann frei bleiben! Daher möglichst auch Teilmengen von CPUs zuordnen.
- feste CPU Zuordnung als Extremform von Gangscheduling:
  - jeder Prozeß erhält Anzahl CPUs passend zur Anzahl Threads <u>für die Laufzeit</u> des Prozesses exklusiv, bei E/A Blockierung bleibt CPU idle
    - bei Massiv Parallelen Architekturen ist CPU Auslastung nicht so heikel
    - keine Prozeßwechsel -> weniger Overhead, effektiveres Cacheing
  - Problem der CPU Zuordnung erinnert an Virtual Memory und Bestimmung der Größe des Working Sets
- dynamisches Scheduling: Anzahl Threads eines Prozesses darf zur Laufzeit variieren, OS teilt CPUs zu, Prozess ist in der Lage mehr oder weniger CPUs mit Threads zu nutzen
  - Zukunftsmusik, Forschungsthema

Real-Time Scheduling, Scheduling mit Echtzeitanforderungen

- Echtzeitanforderungen treten bei der Steuerung realer Systeme auf, z.B. Laborexperimente, Produktionsanlagen, Robotersteuerungen, Überwachung des Luftraums, Explorationssysteme (Deep Space Mission, U-Boote...), Fly-by-wire, Drive-by-wire, ...
- Man unterscheidet
  - hard real-time tasks: Termine/Deadlines müssen eingehalten werden. Eine Weiterbearbeitung nach Verpassen der Deadline ist nicht möglich / sinnvoll.
  - soft real-time tasks: Termine sollten möglichst eingehalten werden; Aufgaben können auch nach Überschreiten einer Deadline noch sinnvoll abgeschlossen werden.
  - aperiodische Aufgaben mit Anfangs- oder Endterminen
  - periodische Aufgaben mit Anforderungen der Art: „jeweils einmal in T Zeiteinheiten“ oder „genau im Abstand von T Zeiteinheiten“

Besonderheiten bei Echtzeit-Betriebssystemen

Determiniertheit

- Durchführung von Aufgaben zu festgelegten Zeitpunkten / Zeiträumen. Bei mehreren Prozessen, die ihrerseits von externen Ereignissen abhängen, hängt Determiniertheit von den Reaktionszeiten des OS auf Ereignisse und den Ressourcen zu Bearbeitung von Ereignissen ab. Typische aber unzureichende Metrik: Reaktionszeit, Zeit zwischen Interrupt und Bestätigung

Responsiveness

- Bearbeitungszeit eines Interrupts, hängt davon ab, ob für die Interruptroutine ein Prozeß/Modewechsel erforderlich ist, wie lange die Routine bearbeitet wird, ob Interrupts geschachtelt auftreten können (Interrupt Nesting)

Einfluß des Anwenders, wesentlich umfangreicher

- auf Scheduling, Prioritäten, Paging/Swapping, Rechte

Zuverlässigkeit, Fail-soft Operation

- höhere Anforderung an Zuverlässigkeit (keine Systemabstürze), bei Ausfällen müssen Prozesse nach Bedeutung gewichtet werden.

Besonderheiten bei Echtzeit-Betriebssystemen

Übliche Feature

- schnelle Prozeß/Threadwechsel
- geringe Größe bzgl minimaler Funktionalität
- kurze Reaktionszeiten auf externe Interrupts
- Multitasking für Interprozeßkommunikation (Semaphore, Signale.
- Spezielle sequentielle Dateien zur schnellen Datenaufnahme
- unterbrechendes Scheduling mit Prioritäten
- minimale Zeitintervalle in den Interrupts disabled sind
- Basisoperationen zum Verzögern von Tasks für eine feste Zeit
- Spezielle Alarmsignale und Time-out Signale

Dispatcher agiert bei Realtime anders als bei Time-sharing

- für deterministische Reaktionszeiten im Millisekundenbereich und darunter wird entweder mit regelmäßigen Preemption Points sogar der OS Kernel unterbrochen oder mit sofortiger Unterbrechung auf Interrupts reagiert (Ausnahme: Kernel ist in kritischem Bereich, „critical-code lockout section“)

Scheduling bei Vorhandensein von Deadlines

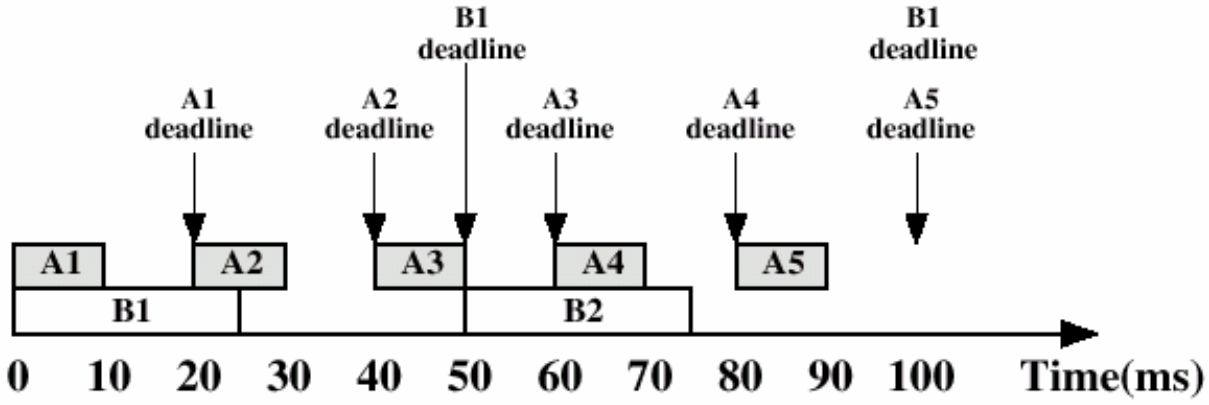
- Welcher Task soll als nächster bearbeitet werden ?
  - Sinnvolle Regel: „earliest deadline first“
- Sollen Unterbrechungen stattfinden ?
  - bei Endterminen sind Unterbrechungen sinnvoll
  - bei Anfangsterminen sind Unterbrechungen unnötig, insbesondere wenn Prozesse vorab bekannt sind

Wir betrachten 2 Beispiele und kommen dann zu einem häufig eingesetzten Verfahren für periodische Tasks: Rate Monotonic Scheduling

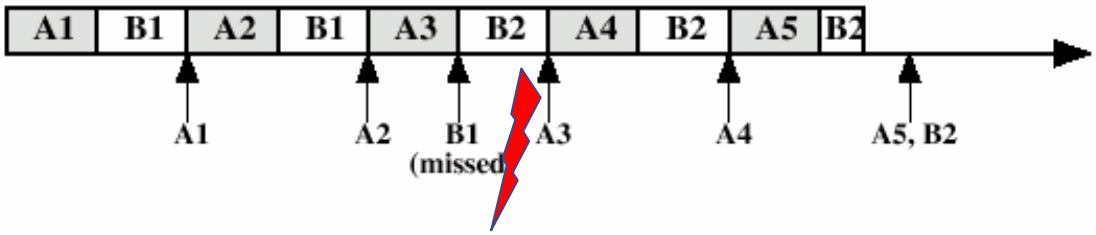
Beispiel: Periodische Tasks mit Endterminen

A alle 20 ms
B alle 50 ms

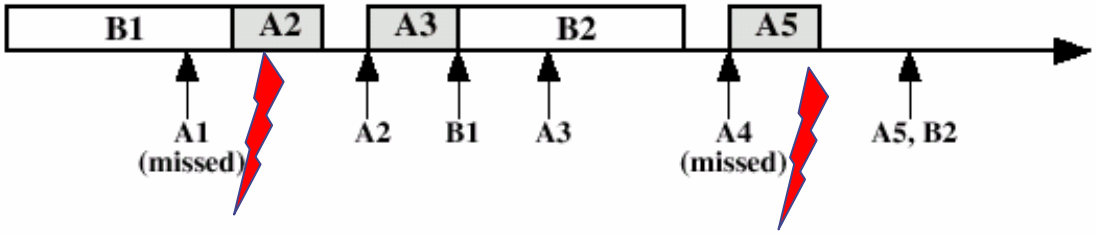
Arrival times, execution times, and deadlines



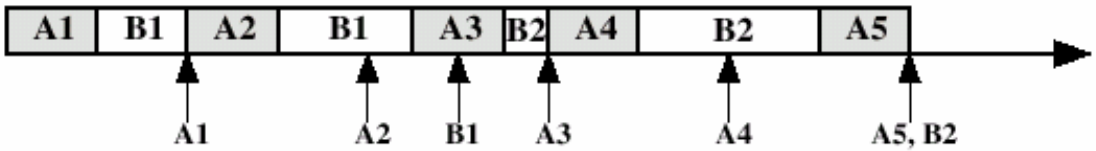
Fixed-priority scheduling; A has priority



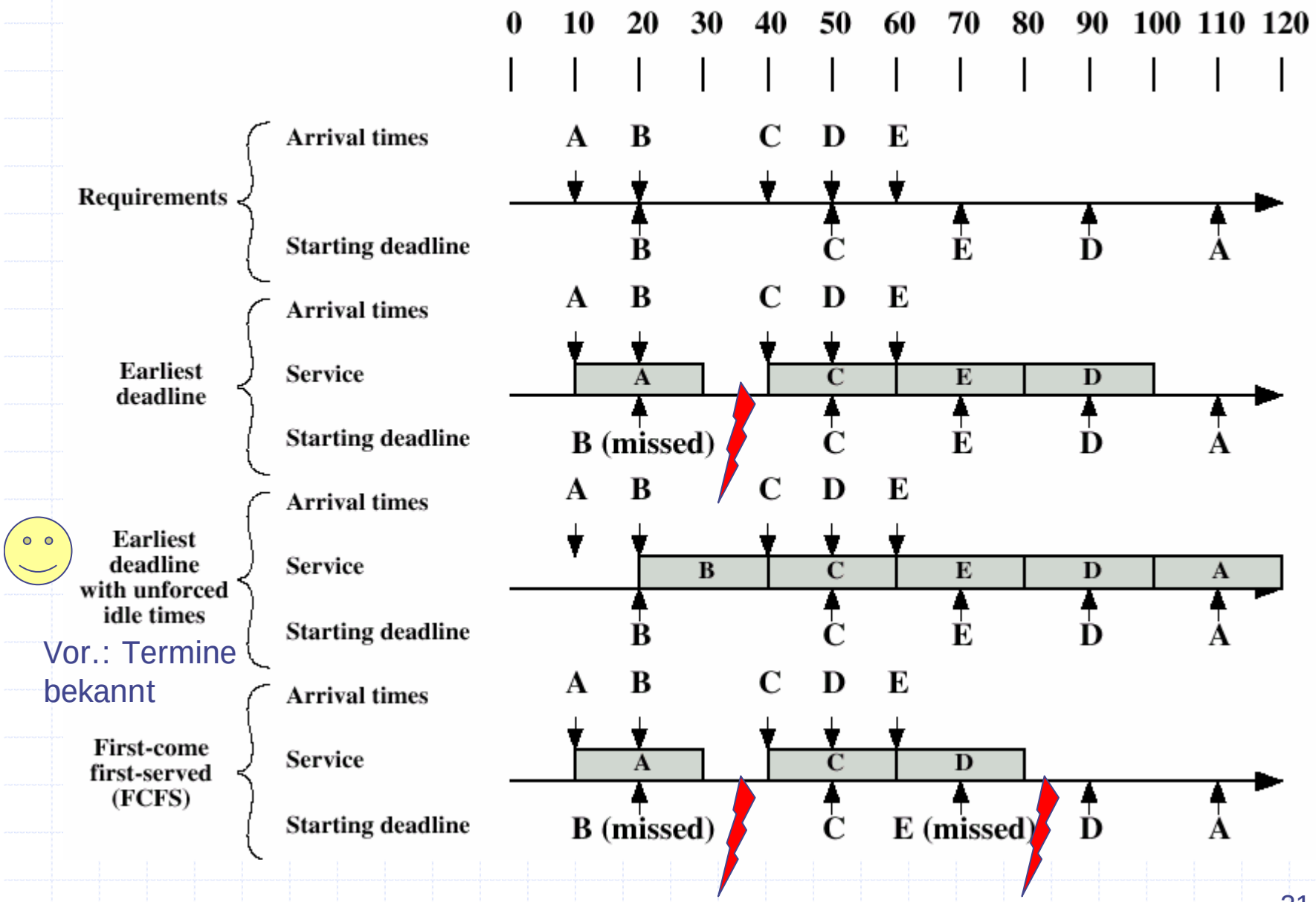
Fixed-priority scheduling; B has priority



Earliest deadline scheduling using completion deadlines



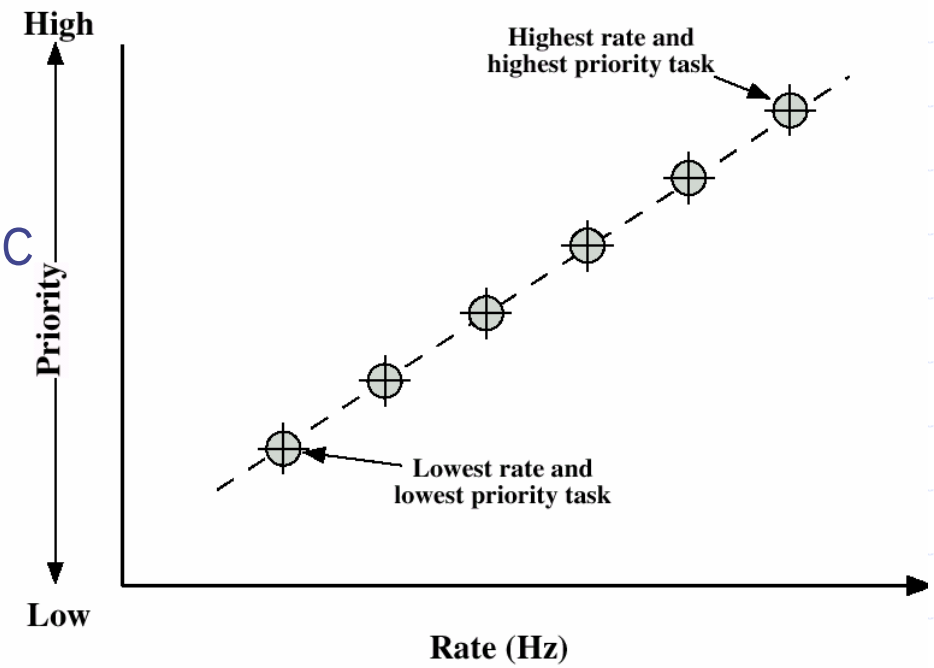
Beispiel: Aperiodische Tasks mit Anfangsterminen



Vor.: Termine bekannt

Rate Monotonic Scheduling (RMS)

Tasks haben Anfangsabstände T (Periode, Zeiteinheit Sek), und damit Rate 1/T (in Hertz), z.B. 50ms -> 20 Hz.
Tasks haben Bearbeitungszeit C
RMS: Task mit größter Rate erhält höchste Priorität.



Bei n Tasks gilt allgemein eine Obergrenze für die CPU Auslastung von 1

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq 1$$

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

liegt zwischen 0,693 und 1

Bei RMS ist die Obergrenze niedriger. Falls Ungleichung gilt, ist gesichert, dass RMS funktioniert.

RMS

- RMS wird in industriellen Anwendungen häufig eingesetzt

  (obwohl mit „earliest deadline first“ höhere CPU Auslastung möglich)

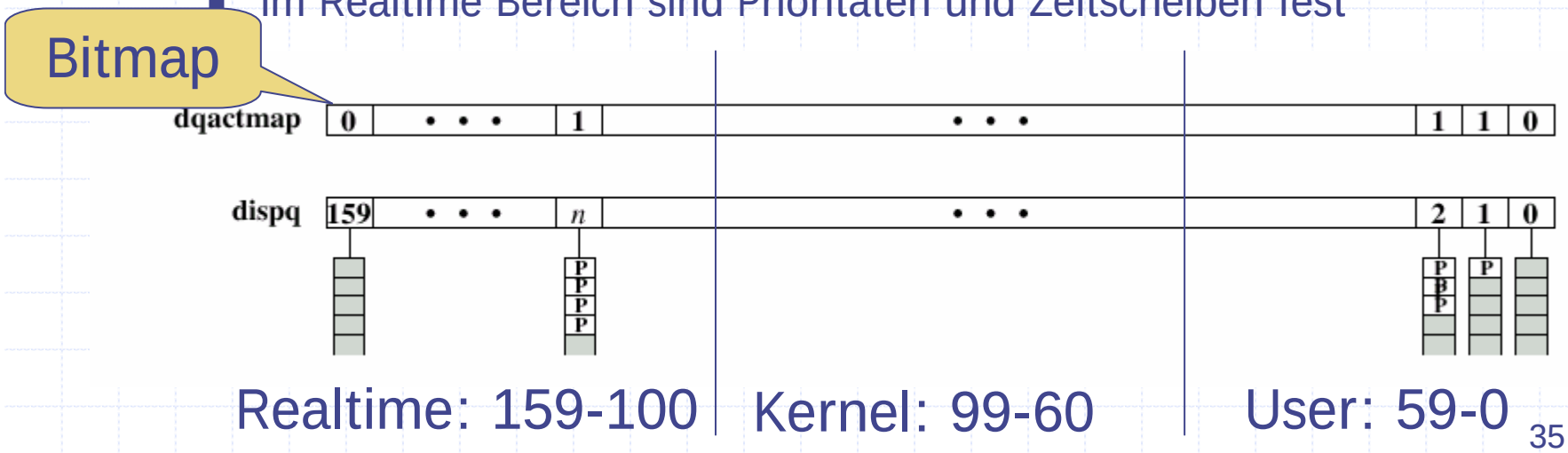
  - Leistungsunterschiede sind in der Praxis gering, die Obergrenze ist recht konservativ, Auslastung von 90% wird durchaus erreicht
  - vielfach zusätzlich zu hard- auch soft-realtime Tasks, daher kann übrige CPU Kapazität mit soft-realtime tasks genutzt werden
  - Stabilität mit RMS leichter zu gewährleisten, bei Überlastung sichern statische Prioritäten Bearbeitung zu. (bei Earliest Deadline Scheduling variieren Prioritäten).

Beispiele für Scheduling: LINUX

- basiert auf traditionellem UNIX Scheduling mit 3 Klassen wg Echtzeitanforderungen
  - SCHED_FIFO: Realtime Threads mit FIFO Scheduling und Prioritäten, Unterbrechungen nur bei höherpriorigen Threads, Blockierung oder freiwilliger Aufgabe.
  - SCHED_RR: Realtime Threads mit RR Scheduling und Prioritäten wie FIFO nur mit Zeitscheiben und entsprechenden Interrupts.
  - SCHED_OTHER: alle anderen Prozesse/Threads werden mit Multilevel Feedback und RR behandelt. Prozesse haben eine Basispriorität und können vom Anwender zusätzlich noch mit nice innerhalb des zulässigen Prioritätsbereichs herabgestuft werden. Natürlich werden Betriebssystemtasks bevorzugt behandelt.

Beispiele für Scheduling: UNIX SV R4

- Rangordnung:
  - Realtime Prozesse vor Kernel- vor Usermode Prozessen behandelt
- Statischer Prioritätsscheduler mit Unterbrechungen und 160 Prioritätsstufen, in 3 Gruppen unterteilt
- Preemption Points, um Kernelunterbrechungen zu ermöglichen, (safe places in Kernelroutinen)
- RR für jede Prioritätsstufe
  - im Bereich der Userprozesse sind Prioritäten dynamisch um CPU lastige Prozesse nicht zu bevorzugen, Zeitscheiben für niederpriorige Prozesse werden darüberhinaus größer 10ms -> 100ms
  - im Realtime Bereich sind Prioritäten und Zeitscheiben fest

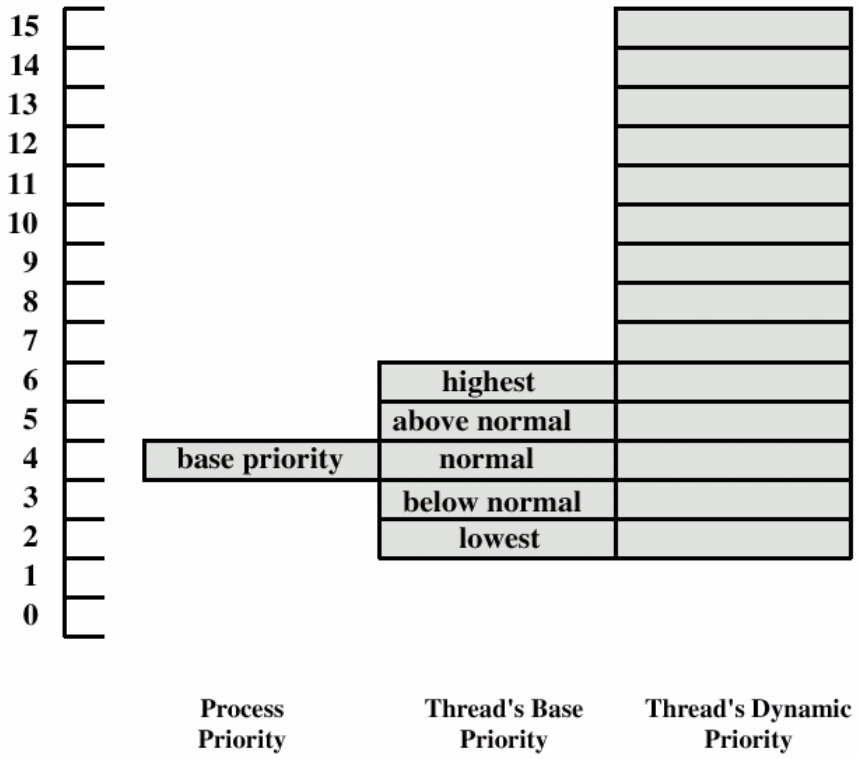


Beispiele für Scheduling: Windows 2000

Unterbrechender Scheduler mit Prioritätsstufen

- 2 Gruppen: Realtime Bereich mit festen Prioritäten,

  zusätzlicher Bereich mit variablen Prioritäten
- je Gruppe 16 Stufen, je Stufe RR Scheduling
- im variablen Bereich werden Prioritäten zugunsten von E/A lastigen Threads variiert, insbesondere bei Blockierung wg Interaktion mit dem Anwender



Besonderheit beim Multiprozessorscheduling mit N CPUs

- N-1 CPUs bearbeiten exklusiv die N-1 höchstpriorigen Threads
- 1 CPU übernimmt die restlichen niederpriorigen Threads

Zusammenfassung

- Einprozessorscheduling
  - Ziel: geringe (normalisierte) Turnaroundzeiten
  - Schedulingalgorithmen: FCFS, RR, SPN, SRT, HRRN, FB
  - in praxi:
    - FB mit FCFS, RR je Ebene,
    - Prioritätsstufen, variable Prioritäten um E/A lastige Prozesse nicht zu benachteiligen, je Stufe FCFS oder RR
- Mehrprozessorscheduling
  - Granularität, Prozeß-Scheduling
  - Thread Scheduling, Gang Scheduling
- Realtimescheduling
  - Deadline Scheduling
  - Rate Monotonic Scheduling
- Beispiele: Unix SVR3, Linux, Unix SVR4, Windows 2000