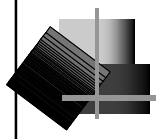
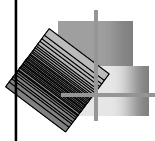


Kontrollflüsse

Planen (Scheduling)

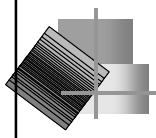
1



Motivation

- Schedulingstrategie hängt vom Einsatzgebiet ab
  - Batch-Betrieb: Ausführung von nicht-interaktiven Anwendungen mit langen Rechenzeiten
  - Echtzeitbetrieb: Einhaltung vorgegebener Fristen
  - Interaktiver Betrieb: Benutzer wollen schnell bedient werden
- Scheduling für Mono- und Multiprozessorsysteme
- Häufig zweistufige Verfahren:
  - Schedulingentscheidung muß schnell getroffen werden
  - Short-Term-Scheduler: Auswahl des nächsten Prozesses
  - Long-Term-Scheduler: Längerfristige Aktualisierung der Schedulingkriterien und -parameter
- Unnötige Kontextwechsel vermeiden
  - Sichern des Prozessorzustandes kostet Zeit
  - Problematik kalter Caches

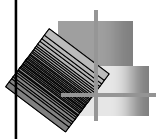
2



Scheduling-Kriterien

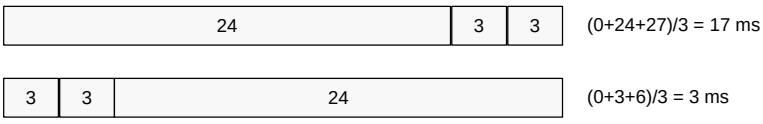
- CPU-Auslastung
  - möglichst viele CPU-Zyklen sinnvoll nutzen
- Durchsatz
  - Anzahl fertiggestellter Aufträge pro Zeiteinheit maximieren
- Turnaround-Zeit
  - Zeit zwischen zwei Prozeßaktivierungen minimieren
- Wartezeit
  - Minimierung der Zeit in Bereit-Liste
- Antwortzeit
  - Schnelle Reaktion auf interaktive Eingaben (Maus, Tastatur, ...)
- Realzeit
  - Einhalten vorgegebener Fristen (Echtzeitprogrammierung)

3



First-Come, First-Served (FCFS)

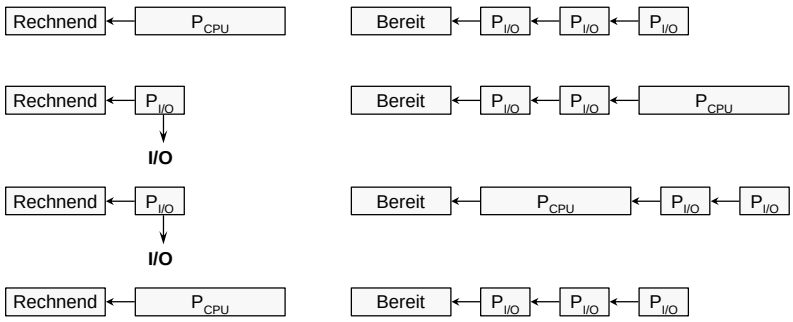
- CPU-Zuteilung in der Reihenfolge des Auftrageingangs
- Nicht-preemptives Scheduling
- Einfache Realisierung über Schlange
  - Prozeß am Kopf der Schlange wird CPU zugeteilt
  - bei freiwilliger Abgabe oder E/A-Operation wieder ans Ende
- Mittlere Wartezeit variiert stark (bzgl. CPU-Bursts):



4

Der Konvoi-Effekt

- Ein Prozeß mit langem CPU-Burst “kann den ganzen Betrieb aufhalten”
  - Rechnerauslastung nicht optimal



5

Shortest-Job-First (SJF)

- CPU geht an Prozeß mit dem kleinsten nächsten CPU-Burst
  - bei mehreren Prozessen mit gleich langem Burst FCFS
- Problem: Länge des nächsten CPU-Burst unbekannt
- SJF ist optimal bzgl. Minimierung der Wartezeit
- Approximation:
  - exponentielle Mittel der gemessenen Längen vergangener Bursts:

$$Burst_{Geschätzt,n+1} = a * Burst_{Gemessen,n} + (1-a)\,Burst_{Geschätzt,n}$$

  - Warum exponentiell?
- Preemptive und nicht-preemptive Variante
  - Prozeß mit kleinerem Burst kommt in die Bereit-Liste

6

Prioritätsbasiertes Scheduling

- CPU wird dem Prozeß mit höchster Priorität zugeteilt
- SJF ist Spezialfall für prioritätsbasiertes Scheduling
  - Priorität ist umgekehrt proportional zur geschätzten Länge des nächsten CPU-Burst:

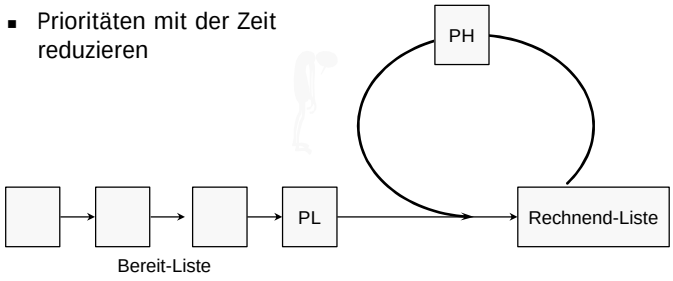
$$p = \frac{1}{Burst_{Geschätzt,n}}$$

- Prioritätszuordnung
  - Intern: Betriebssystem weist Priorität gemäß bestimmter Kriterien zu
  - Extern: Prioritäten werden von der Anwendung vorgegeben
- Preemptive und nicht-preemptive Variante
  - Prozeß mit höherer Priorität wird rechenbereit
    - Neu entstandener Prozeß
    - Beendete E/A-Operation o.ä.

7

Aushungern von Prozessen

- Fairness bei prioritätsbasierten Verfahren nicht garantiert
- Beispiel:
  - Hoch-priorer Prozeß PH mit kleinem CPU-Burst
  - Nieder-priorer Prozeß PL mit längerem CPU-Burst
- Aging
  - Prioritäten mit der Zeit reduzieren



8

Round-Robin (RR)

- RR = FCFS mit Preemption
  - Das Scheduling-Verfahren in Multitasking-Systemen
  - Zeitscheibe 10 ms bis 20 ms
- Mittlere Wartezeit
  - Zusätzliche Kontextwechsel
  - bestimmt durch Prozeßanzahl: bei n Prozessen und Zeitscheibe zs

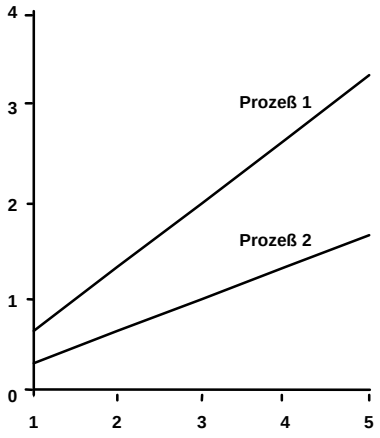
$$WZ_{mittel} \leq zs*(n-1)$$

  - Länge der CPU-Bursts hat keinen Einfluß
- Wahl der Zeitscheibe kritisch für Gesamtleistung
  - zu groß: FCFS
  - zu klein: zuviele Kontextwechsel

9

Benachteiligung I/O-intensiver Prozesse

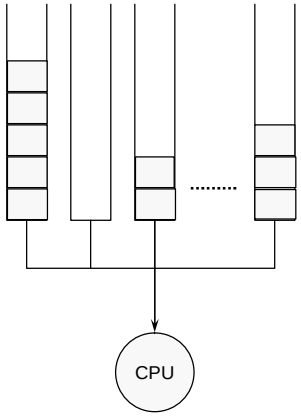
- CPU-intensive Prozesse werden von RR bevorzugt
- Beispiel:
  - Zeitscheibe 10 ms
  - Prozeß 1 (rechenintensiv): 10 ms CPU-Bursts
  - Prozeß 2 (I/O-intensiv): 5 ms CPU-Bursts
- Lösungsansatz:
  - spezielle Bereit-Liste für Prozesse die Zeitscheibe nicht aufgebraucht haben



10

Multilevel Scheduling

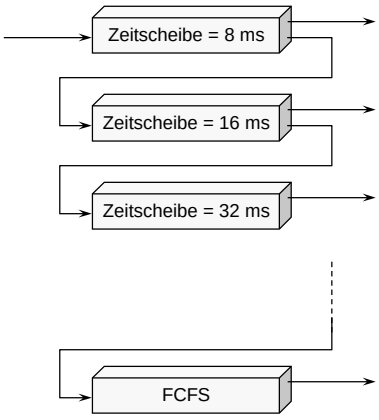
- Kombination mehrerer Scheduling-verfahren
- Gängige Kombinationen
  - Hintergrund- (Batch) und Vordergrund-aufträge (Interaktiv)
  - Echtzeitaufträge und zeitunkritische Aufträge
- Unterteilung der Bereit-Liste
  - Jede Teilliste verwendet eigene Schedulingstrategie
  - Prozesse werden permanent einer Liste zugeordnet
- Zusätzliches Auswahlverfahren bei mehreren nicht-leeren Teillisten
  - feste Prioritäten
  - Zeitmultiplexen



11

Multilevel Feedback Scheduling

- Bevorzugung interaktiver (I/O-intensiver) Prozesse
- Aufteilung der Prozesse z.B. nach Länge der CPU-Bursts
  - Prozesse wechseln zwischen den einzelnen Teillisten
  - Prozesse mit langen CPU-Bursts in der Vergangenheit werden in der Priorität reduziert
- Komplexere Feedback-Mechanismen möglich
  - Wann sinkt ein Prozeß ab?
  - Wann steigt ein Prozeß auf?



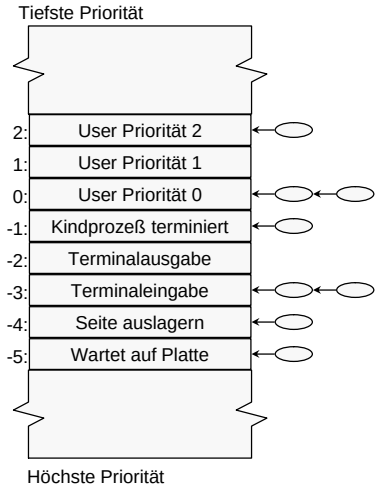
12

Beispiel: UNIX

- Multilevel-Feedback-Scheduling
- Prioritätszuordnung
  - < 0: Prozeß wartet im Kern-Modus
  - >= 0: Prozeß wartet im User-Modus
- Strategie
  - Suche von hoher zu niedriger Priorität
  - RR innerhalb einer Liste
  - Prozesse schnell aus dem Kern
- Feedback
  - 1 pro Sekunde: Neuberechnung der Prozeßpriorität

$$CPU_{Neu} = \frac{CPU_{Alt}}{2}$$

$$P_{Neu} = (0 \; oder \; renice) + CPU_{Neu}$$



13

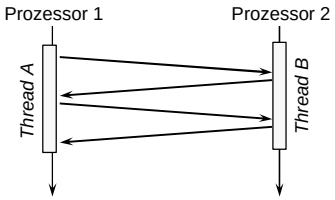
Multiprozessorscheduling

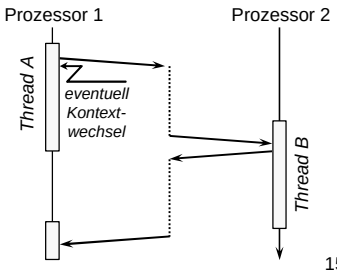
- Realisierung des Zustandsmodells
  - Wo sind die Teillisten plaziert?
  - Wer darf auf die Listen zugreifen?
- Zuordnung Thread-Prozessor
  - statisch: Thread bleibt auf einem einmal zugeordneten Prozessor
  - dynamisch: Prozessor wird bei jeder Threadaktivierung neu gewählt
- Schedulingziel
  - traditionell hohe Auslastung
  - bei “ausreichend vielen” Prozessoren Auslastung nicht mehr primär
  - Unterstützung für kooperierende Threads?

14

Mehrere Threads / Mehrere Prozessoren ✓

- Unabhängige Kontrollflüsse
- Kooperierende Kontrollflüsse
  - in einem Adreßraum: Gemeinsamer Speicher
  - in verschiedenen Adreßräumen: Austausch von Nachrichten
- Ziel
  - Ausnutzung vorhandener Parallelität
  - Gleichzeitige Ausführung kooperierender Threads erreichen
  - Keine unnötigen Kontextwechsel
- Anwendungsspezifische Thread-Gruppe für Scheduling sichtbar
  - Kennzeichnung?

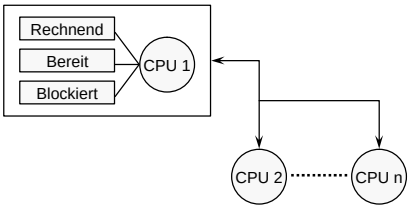


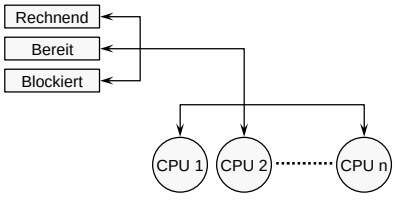


15

Zustandslisten: Plazierung und Zugriff

- Zentrale Listen, ausgezeichneter Prozessor (Master/Slave)
  - Zustandsänderungen u.ä. werden an Master weitergeleitet
  - einfache Monoprozessorerweiterung
  - Ausfall des zentralen Prozessors fatal
  - Master ist schnell Leistungsengpaß
- Zentrale Listen, alle Prozessoren (Peer)
  - Koordinierter Zugriff auf die einzelnen Teillisten (Synchronisation)
- (Lokale Listen für jeden Prozessor)

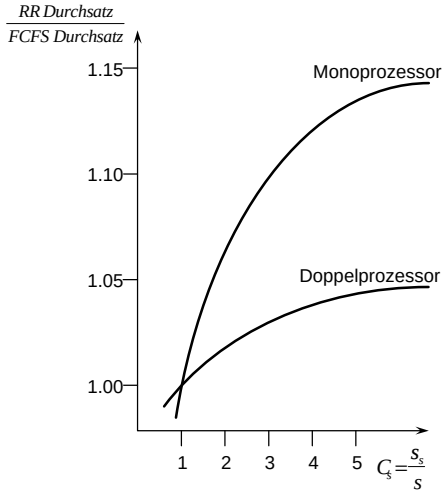




16

Auswirkungen der Schedulingstrategie

- Zusätzliche Effekte durch Verfügbarkeit mehrerer Prozessoren
  - mehr Auswahl, dadurch z.B. Vermeidung von Konvoi-Effekten
- Einfache Strategien häufig ausreichend
- Beispiel
  - C. Sauer, K. Chandy (1981), *Computer Systems Performance Modelling*, Prentice Hall
    - σ_s Standardabweichung der Bedienzeit
    - s Mittlere Bedienzeit



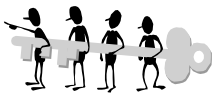
17

Load Sharing

- Globale Listen
  - Untätige Prozessoren wählen Thread aus der globalen Bereit-Liste
- Vorteile:
  - Gleichmäßige Verteilung der Rechenlast
  - Kein zentraler Scheduler
  - Alle Schedulingstrategien für Monoprozessoren anwendbar
    - Einfache Strategien u.U. ausreichend
- Nachteile:
    - FCFS hat sich in vielen Fällen als völlig ausreichend erwiesen
  - Alle Prozessoren greifen auf die zentralen Listen zu (ab einer bestimmten Prozessoranzahl kritisch)
  - Wiederanlauf eines unterbrochenen Thread auf anderem Prozessor hat Folgekosten (kalter Cache, kalter TLB, etc.)
  - Gleichzeitig Ausführung bestimmter Threads nicht garantiert

18

Gang Scheduling



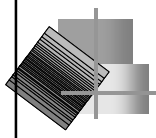
- Eine Menge kooperierender Threads wird auf mehrere Prozessoren verteilt und “gleichzeitig” ausgeführt
  - Andere Begriffe: Gruppenscheduling, Coscheduling
- Vorteile:
  - Leistungssteigerung (Parallele und verteilte Programmierung)
  - Weniger Kontextwechsel
  - Reduzierter Schedulingaufwand: eine Schedulingentscheidung gilt für viele Prozessoren
- Nachteile:
  - Kennzeichnung zusammengehöriger Kontrollflüsse
  - Problem: Leistungssteigerung einer Anwendung vs. Auslastung des Gesamtsystems
    - Beispiel: N Prozessoren, M Anwendungen mit T_k < N Threads ($k \in M$) und T_m+T_n > N für alle m und n

19

Dedizierte Prozessorzuordnung

- Mehrere Prozessoren einer Anwendung für die gesamte Ausführungszeit fest zuordnen
  - Anwendung = kooperierende Threads in mehreren Adreßräumen
- Vorteile:
  - Keine Kontextwechsel
  - Keine kalten Caches
  - Einfache Realisierung
- Nachteile:
  - Verschwendung von Prozessoren?
    - Bei Systemen mit sehr vielen Prozessoren akzeptabel
  - Sinnvolle Zuordnung Thread-Prozessor
    - nur eingeschränkt automatisch durch Compiler
  - Nicht alle Anwendungen so modellierbar

20

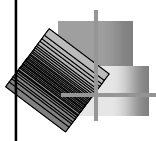


Activity Working Set

- Problematik: Wieviele Prozessoren werden einer Anwendung zugeordnet?
- Analogie zur Working-Set-Theorie:
  - E. Gehringer, D. Siewiorek, Z. Segall (1987), *Parallel Processing: The Cm* Experience*, Digital Press

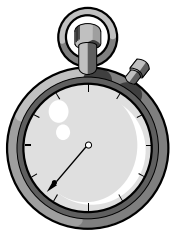
- AWS: Prozessoren
  - Anwendung kann potentiell sehr viele Threads erzeugen
  - Aktivitätslokalität?
  - Prozessorflattern
  - Prozessorfragmentierung
- WS: Speicherseiten
  - Anwendung kann potentiell sehr viele Seiten referenzieren
  - Referenzlokalität?
  - Seitenflattern
  - Speicherfragmentierung

21



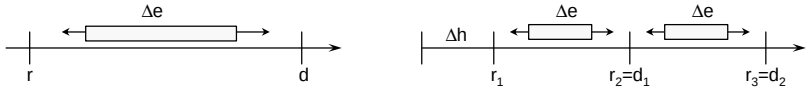
Echtzeitscheduling

- Einhaltung anwendungsspezifischer Zeitvorgaben
  - Zeitvorgaben und Programmcharakteristika ermitteln (schwierig)
  - Schedulerentscheidung
- Harte Echtzeitsysteme
  - Verletzung einer Zeitvorgabe kann katastrophale Folgen haben
  - Beispiele:
    - Steuerung kritischer chemischer Reaktoren
    - Antiblockiersystem
    - Digitalsteuerung in Airbus-Flugzeugen
- Weiche Echtzeitsysteme
  - Verletzung von Zeitvorgaben störend Beispiele:
    - Digitale Telefonvermittlungen
    - Multimedia



22

Beschreibung von Echtzeitaktivitäten



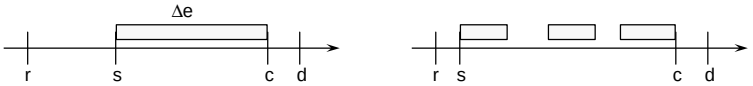
- Modellierung der Zeitvorgaben für Aktivitäten:
  - Bereitzeit (r, Ready Time): Frühester Ausführungszeitpunkt
  - Frist (d, Deadline): Spätester Zeitpunkt für das Aktivitätsende
  - Ausführungszeit (Δe, Execution Time)
- Problem: Ausführungszeit bestimmen
  - Worst-Case-Abschätzung
  - Schlechte Auslastung
- Periodische Aktivitäten:
  - Periode (Δp): Ausführungshäufigkeit
  - Phase (Δh): Zeitlicher Versatz zum Anfang

$$r_k = \Delta h + (k-1)\Delta p$$
$$d_k = \Delta h + k\Delta p = r_{k+1}$$

23

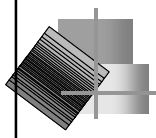
Erfolgreiche Bearbeitung

- Jede Aktivität der Anwendung muß ausgeführt werden
  - Startzeit (s): Tatsächlicher Beginn der Ausführung
  - Abschlußzeit (c, Completion Time): Reales Ausführungsende
- ... und zwar innerhalb der Zeitvorgaben:
  - nicht zu früh: $\forall k: r_k \leq s_k$
  - nicht zu spät: $\forall k: c_k \leq d_k$



ohne Preemption mit Preemption

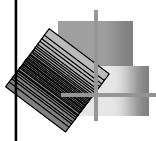
24



Klassifikationskriterien

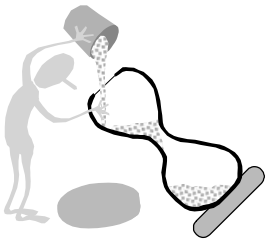
- Statisches / Dynamisches Scheduling
  - Statische Planung
    - Alle Anwendungsvorgaben sind vorab bekannt
  - Dynamische Planung
    - Informationen werden erst im Verlauf der Planung bekannt
- Explizites / Implizites Scheduling
  - Explizit
    - Zur Laufzeit steht ein vollständiger Plan zur Verfügung
    - Offline-Scheduling
    - Scheduling zur Laufzeit = Indizierter Tabellenzugriff
  - Implizit
    - Planungsregeln, z.B. Prioritäten

25



Best Effort

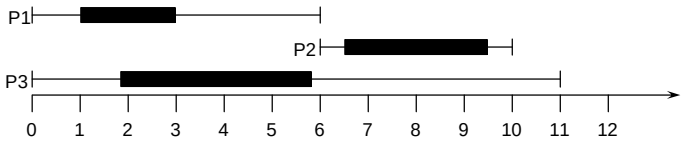
- Keine Berücksichtigung von Echtzeitanforderungen
  - Normale Mono- und Multiprozessor-Schedulingstrategien
  - Keine Angabe von Zeitvorgaben
- Hard- und Software verkraften Maximalanforderung
- Einhaltung von Zeitvorgaben nicht garantiert
- Anwendbar:
  - Vielen weichen Echtzeitbereichen
  - Erschöpfend getesteten harten Echtzeitsystemen (selten)
- Verbreitungsgrad: 95% geschätzt



26

Earliest Deadline First (EDF)

- CPU wird dem Prozeß mit der nächsten Frist zugeordnet
- Preemptive und nicht-preemptive Variante
- EDF ist nicht optimal, d.h. es wird kein brauchbarer Plan gefunden obwohl einer existiert:



  - EDF plant P1 und dann P2 ein; bei P3 wird Frist verletzt?
  - Ein brauchbarer Plan wäre: (P3,s=0), (P1,s=4), (P2,s=6)

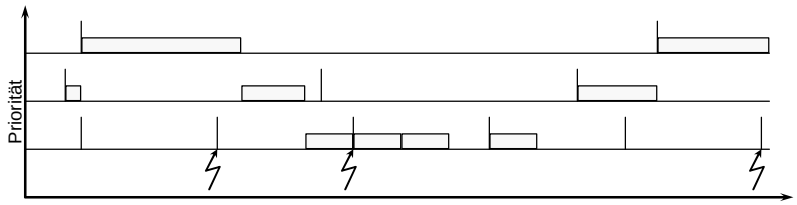
27

Rate Monotonic Scheduling (RMS)

- Implizites Verfahren mit festen Prioritäten
- Theorie von Liu und Layland (1973)
  - Periodische Prozesse
  - Periode bestimmt Priorität: Hohe Frequenz = Große Priorität
  - “Wichtigkeit” und Ausführungszeit (Δe) von Prozessen fließen nicht ein
- Annahmen
  - Bereitzeit beginnt mit der Periode
  - Frist endet mit der Periode
  - Prozesse suspendieren sich nicht selbst
  - Prozesse können unterbrochen werden (Preemption)
  - Overhead für Kontextwechsel und Scheduling ist vernachlässigbar
  - Prozesse sind unabhängig voneinander
- Erweiterungen
  - Aufhebung einzelner Annahmen

28

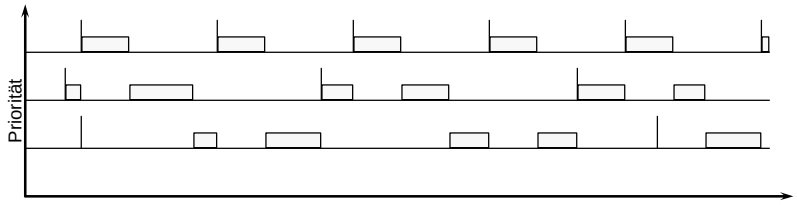
Warum nicht: Wichtiger Prozeß = Hohe Priorität?



- ... aber niedrige Frequenz
- Wachsende Wahrscheinlichkeit, daß hochfrequente Prozesse mit niedriger Priorität ihre Fristen verletzen
- Für sehr wichtige Dinge u.U. einsetzbar
  - z.B. wichtige Systemprozesse
  - Vermeidung von größeren Schäden

29

Hohe Frequenz = Hohe Priorität

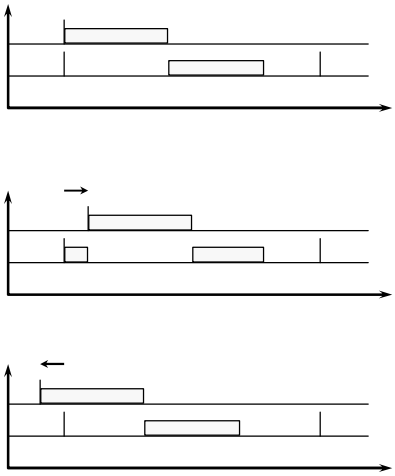


- Hohe Frequenz = Kurze Ausführungszeit ($\Delta e < \Delta p$)
- Hochfrequente Prozesse minimal verzögert
  - Zeiten für Kontextwechsel (werden vernachlässigt)
  - Ausführungszeiten von Prozessen mit noch höheren Frequenzen
- Beobachtung
  - Stärkere Zerstückelung niederfrequenter Prozesse
  - Mehr Kontextwechsel

30

Kritische Instanz

- Wichtig für Einplanbarkeitstest
- Beispiel
  - P_L niederpriorer Prozeß
  - Kann P_L Frist einhalten?
  - P_H hochpriorer Prozeß
- Bereitzeit von P_H und P_L gleich
  - **Kritische Instanz**
- Bereitzeit von P_L vor P_H
  - reine Verschiebung
  - Für die Frist von P_L keine Änderung
- Bereitzeit von P_H vor P_L
  - geringerer Restanteil von P_H ab Bereitzeit von P_L



31

Antwortzeiten

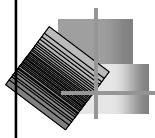
- Antwortzeit eines Prozesses = Abschlußzeit - Bereitzeit
- Längste Antwortzeit
  - Startzeit wird durch die Ausführung und Perioden höher-priorer Prozesse bestimmt
- Keine Verzögerung für den höchst-prioren Prozeß:

$$c_0 - r_0 = \Delta e_0$$

- Für alle anderen Prozesse:

$$c_p - r_p < \sum_{0 \le k < p} \left(\left\lfloor \frac{\Delta p_p}{\Delta p_k} \right\rfloor \cdot \Delta e_k \right) + \Delta e_p$$

32



RMS: Optimalitätskriterium

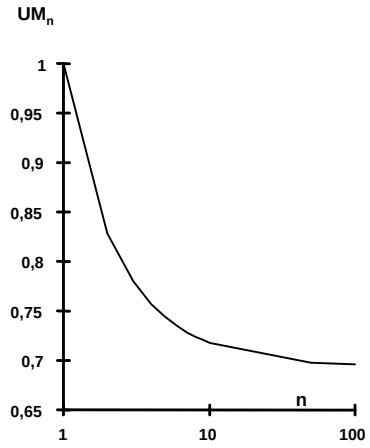
- Für eine periodische Prozeßmenge P_1 bis P_n sei

$$U_n = \sum_{k=1}^{n} \frac{\Delta e_k}{\Delta p_k}$$

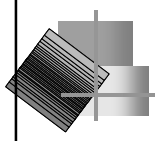
die Auslastung

- RMS liefert einen brauchbaren Plan, wenn

$$U_n < UM_n := n(2^{\frac{1}{n}} - 1)$$



33



Beispiel: QNX

- Echtzeitbetriebssystem
  - Verteilt
  - unterstützt POSIX.4 (Echtzeiterweiterungen)
- Eigenschaften
  - Prioritäten 0 (tief) bis 31 (hoch)
    - **`getprio (pid_t pid)`** und **`setprio (pid_t pid, int new_prio)`**
  - 3 Schedulerklassen (Policy):
    - **`SCHED_FIFO`** (FCFS-Scheduling)
    - **`SCHED_RR`** (Round-Robin)
    - **`SCHED_OTHER`** (Feedback bzgl. aufbrauchen der Zeitscheibe)
    - **`sched_setscheduler (pid_t pid, int policy, int prio)`**
  - Preemptiv
    - FCFS: höher-priorer Prozeß wird rechenbereit
    - RR: Zeitscheibe einstellbar (500 µsec bis 55 msec)
  - Freiwillige Abgabe (meist bei FCFS): **`sched_yield ()`**

34