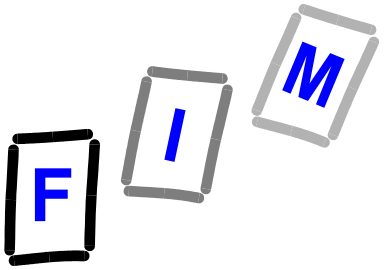
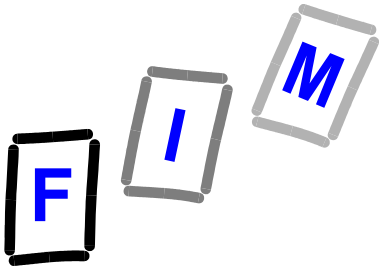


SS 2005

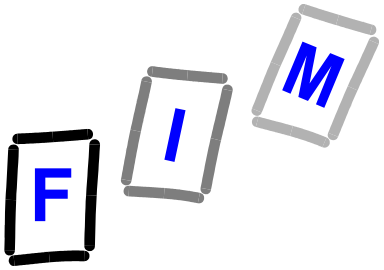
KV Betriebssysteme

Multi-Processing & Scheduling



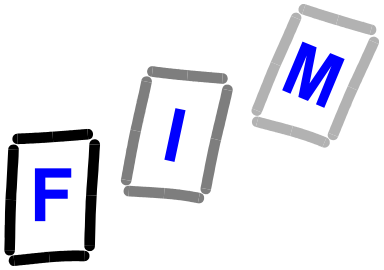
Übersicht

- **Wiederholung der wichtigsten Konzepte und Begriffe aus der VL**
- **Fallbeispiele**
  - **Win NT 4 Scheduler**
  - **Linux O(1) Scheduler**
- **Übungsvorbesprechung**
  - **Scheduler-Implementierung**



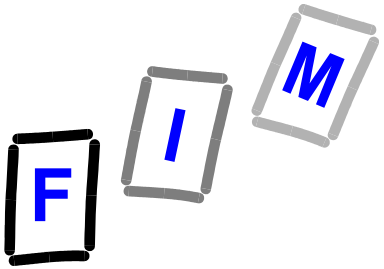
Steuerprogramm (Scheduler)

- **Ein Steuerprogramm ist verantwortlich für**
  - **Auswahl der Prozesse zur Ausführung**
    - » **Steuerung der Prozesse**
  - **Zuteilung von Betriebsmitteln an die Prozesse**
    - » **CPU-Steuerung**
    - » **Disk-Steuerung**
- **Grundsätzlich zwei Ebenen:**
  - **Langzeitsteuerung und**
  - **Kurzzeitsteuerung**



Wiederholung Multi Tasking

- **Kooperatives Multi Tasking Betriebsystem**
  - **mehrere Prozesse werden verwaltet**
  - **Prozesse müssen sich „kooperativ“ verhalten**
  - **Beispiel: Windows <= 3.11, Mac OS bis 9**
- **Präemptives Multi Tasking / Single User Betriebsystem**
  - **mehrere Prozesse werden verwaltet**
  - **ein Prozess kann gezwungen werden, die Kontrolle abzugeben**
  - **Beispiel: Windows NT**
- **Multi Tasking / Multi User Betriebsystem**
  - **Verwaltet mehrere Benutzer, die selbst wieder mehrere Tasks starten**
  - **Zugangskontrolle, und Speicherschutz**
  - **Beispiele: UNIX (z.B. HP-UX, AIX, D-UNIX, Linux, ...)**
- **„Echtes“ Multi Tasking**
- **Dispatcher**



Non-preemptiv

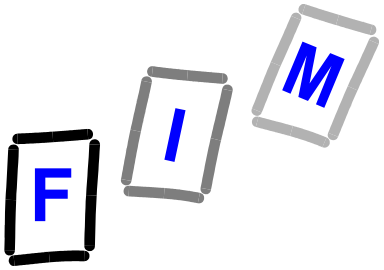
Non-preemptives Scheduling:

Prozesse werden nicht vorzeitig unterbrochen. Sie werden blockiert, geben freiwillig die CPU ab oder sie beenden sich selbst.

<u>Strategien</u>:

Beispiel:	**Job 1:**	**10 ZE**
	Job 2:	**4 ZE**
	Job 3:	**3 ZE**

- **FCFS (First-Come First-Served)**
  - **gesamte Ausführungszeit: 10+14+17 ZE = 41 ZE**
  - **mittl. Ausführungszeit: 41/3 ZE = 13,7 ZE**
- **SJF (Shortest Job First)**
  - **Kürzere Jobs werden bevorzugt**
  - **gesamte Ausführungszeit: 3+7+17 ZE = 27 ZE**
  - **mittl. Ausführungszeit 27/3 ZE = 9 ZE**
- **HRN (Highest Response Ratio Next)**
  - **Jobs mit kurzen Bedienzeiten bzw. langen Antwortzeiten werden bevorzugt (Antwortzeit/Bedienzeit). Kein Prozess verhungert, da die Antwortzeiten mit der Wartzeiten zunehmen.**
- **PS (Priority Scheduling)**
  - **Jobs werden nach ihrer Priorität behandelt; Gefahr: Verhungern von Prozessen**

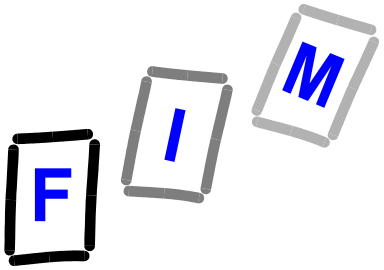


Starvation

- **Problem: Starvation**
  - **Prozess kann ewig auf Zuteilung der CPU warten, da ständig höher prioritäre Prozesse diese anfordern**

> *Es wird gesagt, dass bei der Abschaltung eines Rechners am MIT im Jahre 1973 ein Job in der CPU-Warteschlange gefunden worden ist, der dort sechs Jahre zuvor hineinkam, jedoch aufgrund seiner niedrigen Priorität nicht genügend oft die CPU zugeteilt bekam und dadurch seine Aufgabe noch nicht vollständig hat erfüllen können.*
>
> [http://www4.informatik.uni-erlangen.de/Lehre/WS04/V_BS/folien/09bs-A5.pdf]

- **Lösung: Aging**
  - **Priorität wird entsprechend der Wartezeit in der „Ready“-Queue erhöht**



Preemptiv

Preemptives Scheduling

Prozesse dürfen unterbrochen werden.

Die Zeit wird in Zeitscheiben eingeteilt (Time-Slice-Verfahren).

Strategien:

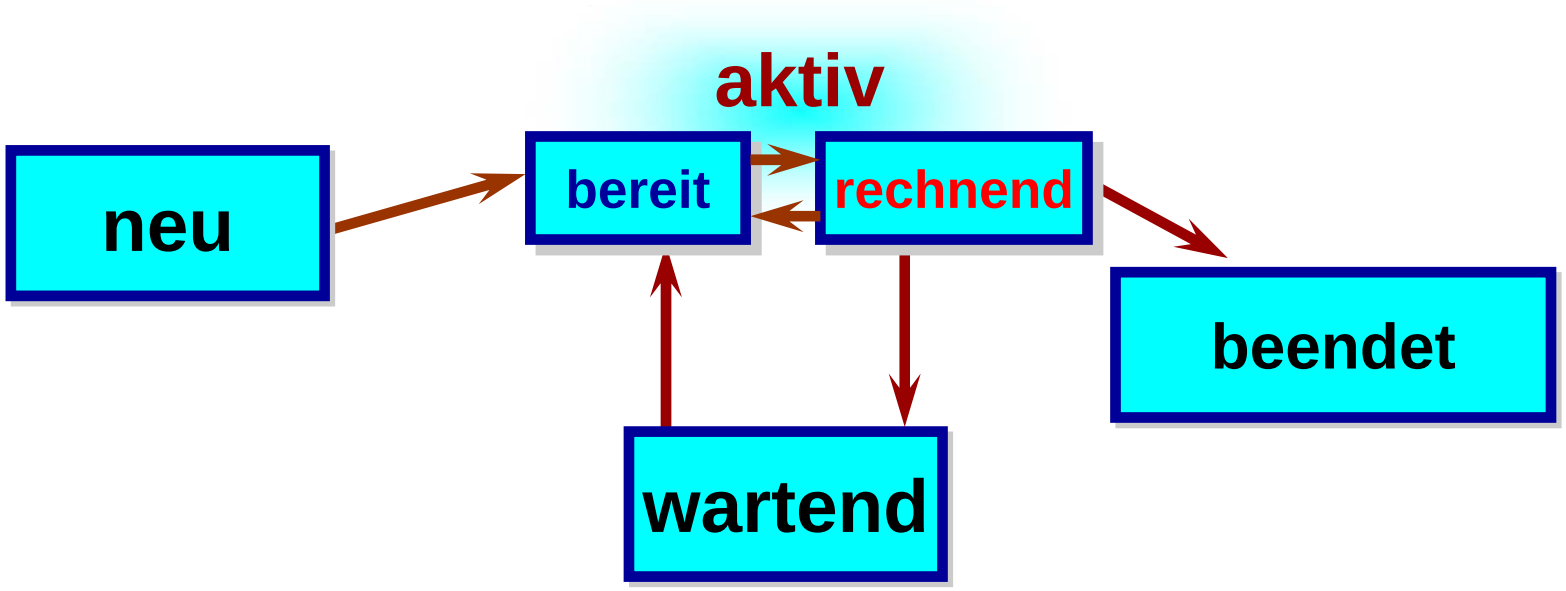
- **RR (Round Robin)**
  Kombination von FCFS mit Zeitscheiben
  Daumenregel: Die Länge einer Zeitscheibe ist ungefähr gleich der mittleren CPU-Zeit für zwei I/O-Aktivitäten
- **DPRR (Dynamic Priority RR)**
  dynamische Priorität
- **SRTF (Shortest Remaining Time First)**
  Jobs mit der kleinsten verbleibenden Bedienzeit werden bevorzugt
- **Stochastische Scheduling Modelle:**
  - **Warteschlangentheorie**
  - **Simulation (Modellierung der Betriebsmittelverteilung)**

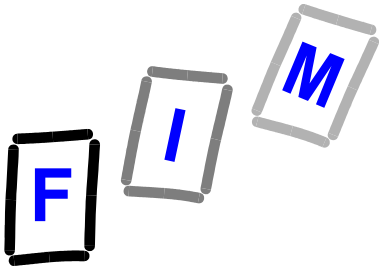
Prozesszustände Grobeinteilung

wenn ein Prozess P ausgeführt wird, ändert sich sein *Zustand*

die Zustände können sein:

neu, aktiv (bereit-rechnend), wartend, beendet





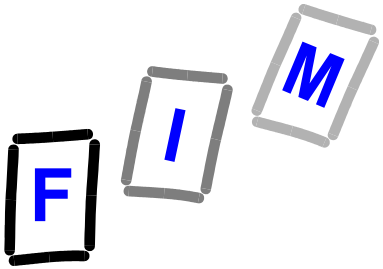
Ziele Scheduling

Scheduling Ziele:

- Effizienz
  - Die Auslastung der CPU soll möglichst maximal sein (100%)
- Durchsatz
  - Die Zahl der Jobs pro Zeiteinheit, die bearbeitet werden, soll maximal sein
- Ausführungszeit
  - Die Zeitspanne vom Job-Beginn bis Job-Ende sollte minimal sein
- Wartezeit
  - Die Zeit in der ein Prozess „bereit“ ist, sollte minimal sein
- Anwortzeit
  - Die Zeit zwischen Eingabe und Antwort des Rechners sollte minimal sein.
- + subjektive Ziele bei CPU Scheduling
  - »Der Benutzer soll das Gefühl haben, dass das System „möglichst sofort“ auf seine Eingaben reagiert
  - »Dem Benutzer soll das Gefühl vermittelt werden, dass die Prozesse quasi gleichzeitig ablaufen

Zielkonflikte:

Werden kurze Prozesse bevorzugt, so verkürzen sich die Antwortzeiten im Mittel; lange Prozesse werden benachteiligt, d. h. es wird die Fairness verletzt.
Es gibt keinen idealen Scheduling Algorithmus, der alle Scheduling Ziele erfüllt.

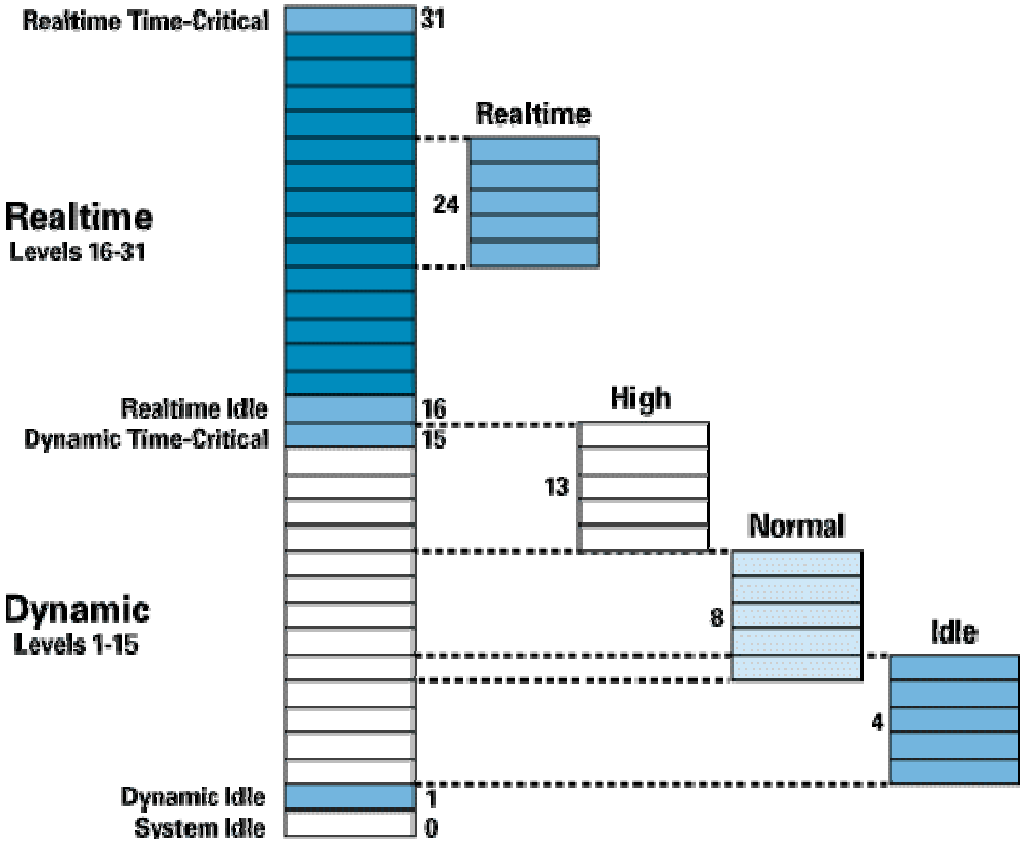


Fallstudie: WinNT Scheduler (1)

- **Windows NT führt Scheduling auf Basis von Threads durch.**
- **NT verwaltet Prioritäten von 1 bis 31. Priorität 0 ist für den „System Idle“ Thread reserviert.**
  - **16-31: Realtime Prioritäten. Administratorrechte erforderlich.**
  - **Priorität 1-15: Dynamische Prioritäten für Std.-Applikationen**
- **Win32 API sieht das Setzen der Priorität in zwei Schritten vor:**
  1. **Setzen der Prioritätsklasse**
  2. **Setzen der relativen Priorität**

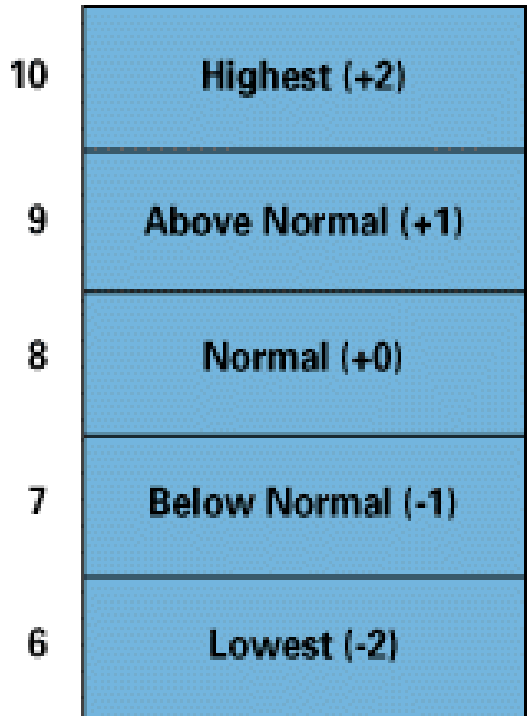
WinNT Prioritäten (1)

- **4 Prioritätsklassen**
  - **Realtime (24)**
  - **High (13)**
  - **Normal (8)**
  - **Idle (4)**



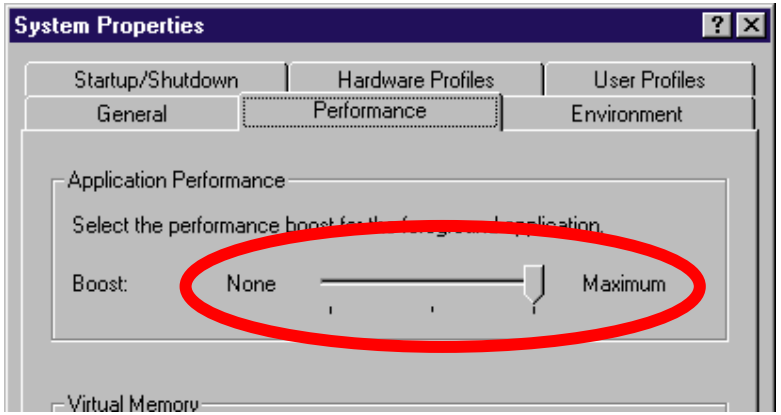
WinNT Prioritäten (2)

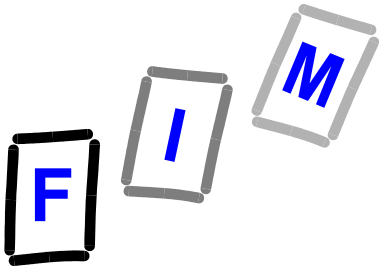
- **Relative Priorität**
  - **+/- 2**



WinNT Scheduler (1)

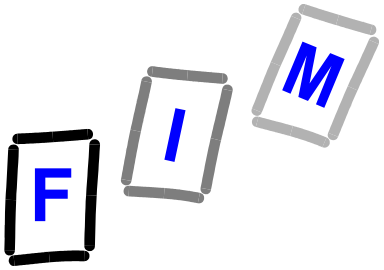
- **Scheduler vergibt Rechenzeit an Threads (Quantum = gewisse Zeit an CPU-Nutzung)**
- **Ein Quantum ist sehr kurz**
  - **WinNT Server: 120 ms**
  - **WinNT Workstation: 20 – 60 ms, abhängig, ob es sich um eine Back- oder Foreground Applikation handelt**





WinNT Scheduler (2)

- **Der NT Scheduler muss zu folgenden Zeitpunkten eine Entscheidung treffen:**
  1. **Das zugewiesene Quantum eines Threads läuft ab**
  2. **Ein Thread wartet auf ein Ereignis**
  3. **Ein Thread wird bereit, ausgeführt zu werden**

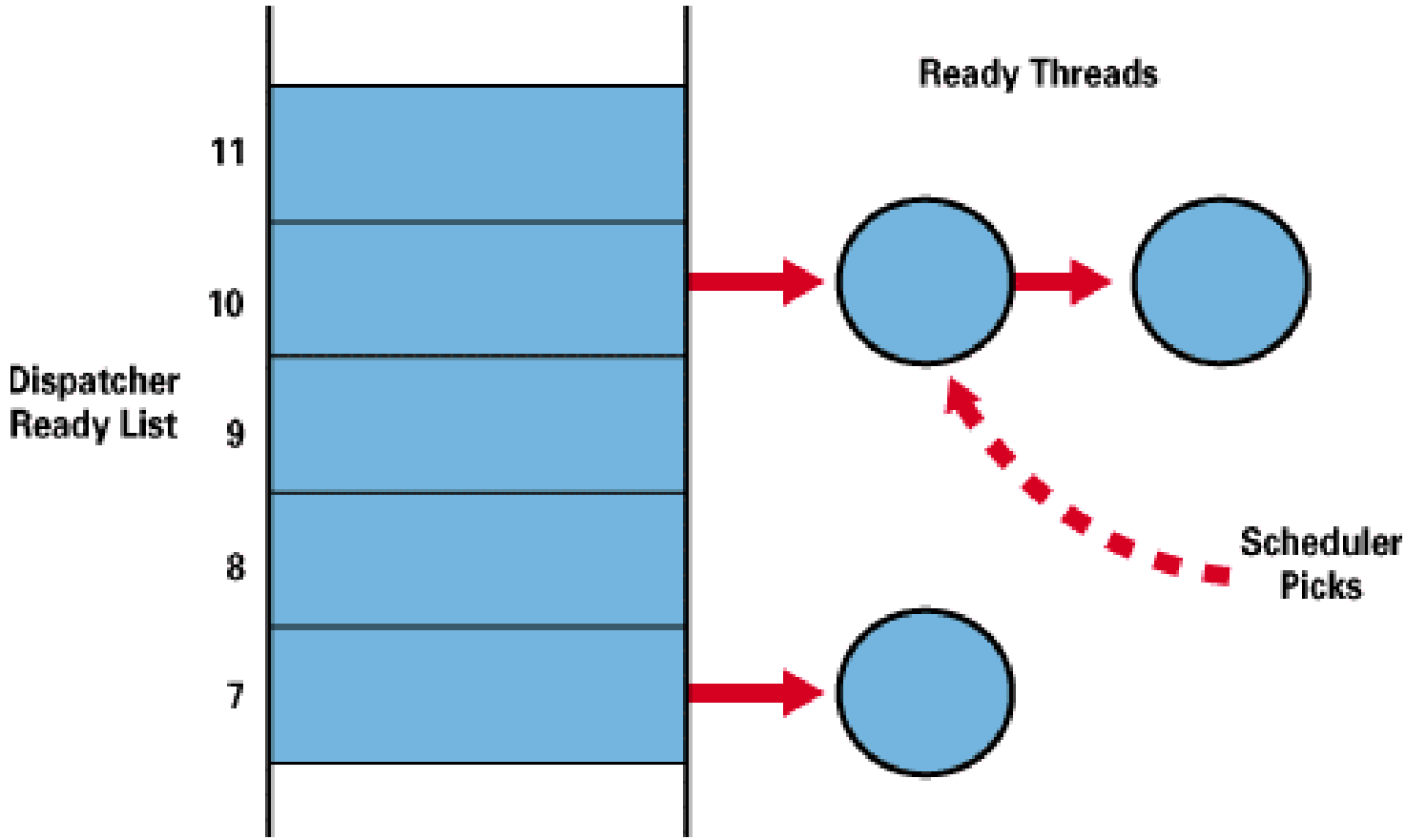


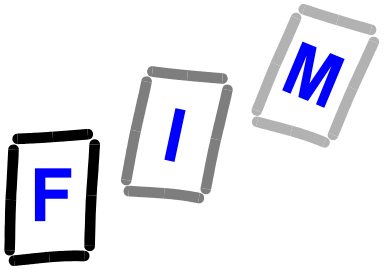
WinNT Scheduler FindReadyThread (1)

Fall 1: Zugewiesenes Quantum läuft ab

- Der NT Scheduler führt die **FindReadyThread** Methode aus, um zu entscheiden, welcher Thread als nächstes ausgeführt werden soll.
- Die Methode FindReadyThread sucht den Thread mit höchster Priorität (bei gleicher FCFS). Alle Threads, die sich im „READY“ Zustand befinden, werden in der **Dispatcher Ready List** verwaltet. Diese enthält 31 Einträge (für jede Priorität einen). Bei jedem Eintrag hängt eine (eventuell leere) Liste mit den Threads der jeweiligen Priorität.
- FindReadyThread geht die Liste von 31 bis 1 durch, bis eine nicht leere Liste vorgefunden wird. Dem ersten Element in dieser Liste wird die CPU zugeteilt.

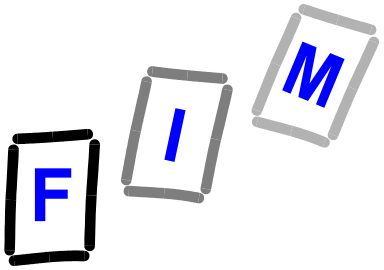
WinNT Scheduler FindReadyThread (2)





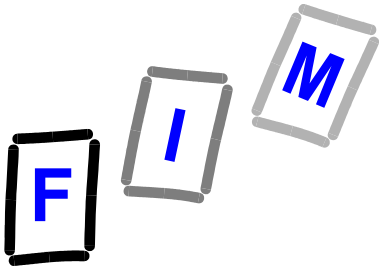
WinNT Scheduler (3)

- **Der NT Scheduler muss zu folgenden Zeitpunkten eine Entscheidung treffen:**
  1. Das zugewiesene Quantum eines Threads läuft ab
  2. **Ein Thread wartet auf ein Ereignis**
  3. Ein Thread wird bereit, ausgeführt zu werden



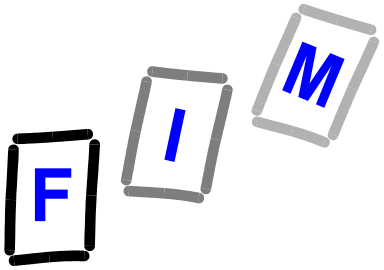
WinNT Scheduler
Warten auf Event

- **Typischer Fall bei Server-Anwendungen oder bei Interaktiven („Klick“) Anwendungen**
- **Die Applikation wartet auf ein Ereignis (Verbinden eines Clients, Klicken auf Button, etc.)**
- **Geschieht keine Interaktion, wird das Quantum vorzeitig abgegeben**



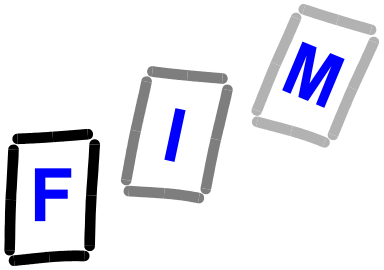
WinNT Scheduler (4)

- **Der NT Scheduler muss zu folgenden Zeitpunkten eine Entscheidung treffen:**
  1. Das zugewiesene Quantum eines Threads läuft ab
  2. Ein Thread wartet auf ein Ereignis
  3. **Ein Thread wird bereit, ausgeführt zu werden**



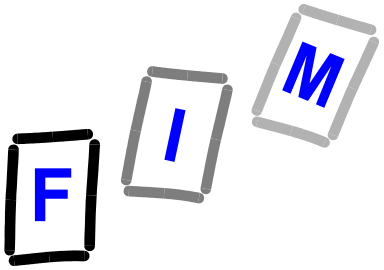
WinNT Scheduler
Bereit für Ausführung

- **Ein neuer Thread oder ein blockierter Thread wird bereit, (wieder) ausgeführt zu werden (Client verbindet sich, *Klick*, etc.)**
- **Die Methode ReadyThread wird ausgeführt. Diese Methode untersucht, ob die Priorität des Threads höher ist, als der momentan ausgeführte Thread.**
  - **Wenn Ja → Momentaner Prozess wird unterbrochen (Eingereiht auf Wartelistenplatz 1 in die jeweilige Prioritätsliste).**
  - **Wenn Nein → Momentaner Prozess läuft weiter; neuer „Ready-Prozess“ wird in Warteliste eingereiht.**



WinNT Scheduler Starvation

- **Starvation wäre ein Problem, …**
- **… gäbe es nicht den NT Balance Set Manager**
  - **Die Methode ScanReadyQueues sucht jede Sekunde in der Dispatcher Ready List nach Prozessen, die seit mehr als 3 Sekunden nicht mehr ausgeführt worden sind. Diese werden dann mit doppeltem Quantum 1x ausgeführt**

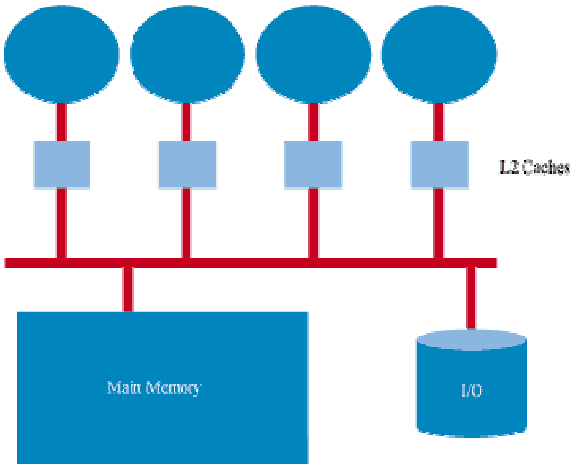


WinNT Scheduler Boosting

- **Bisher wurde der Eindruck vermittelt, dass die Priorität eines Threads über dessen Lebenszeit konstant bleibt, biss dass sie explizit vom Benutzer verändert wird**
- **In Wirklichkeit wird die Priorität vom System aber oft für kurze Zeit verändert. Beim Übergang von WAITING auf READY beispielsweise wird die Priorität um bis zu 6 Stufen erhöht (boosting).**

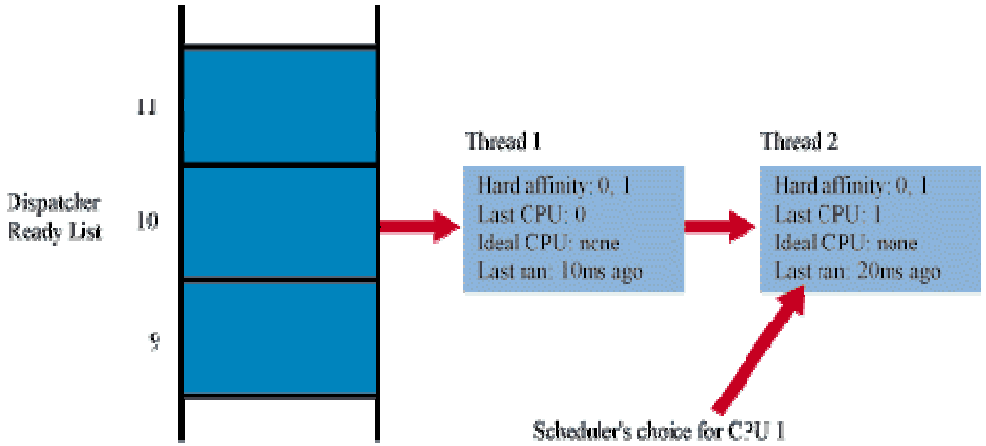
WinNT als Mehrprozessor OS

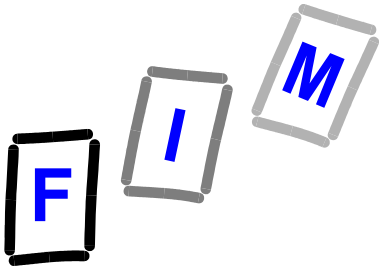
- **SMP – Symmetric Multi Processing**
  - **Mehrere, identische CPUs mit privatem L1 und L2 Cache**
  - **Alle CPUs haben gleiche Zugriffsrechte auf Speicher und Peripherie**
  - **NT kann auf jeder CPU (oder auch auf mehreren) laufen**



WinNT Mehrprozessor OS (2)

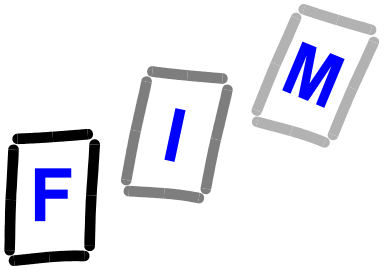
- **Begriff der „Affinität“**
  - **Soft-affinity: Prozess soll – wenn möglich – auf einer bestimmten CPU ausgeführt werden (aus Cache Gründen)**
  - **Hard-affinity: Prozess muss auf einer bestimmten CPU ausgeführt werden**





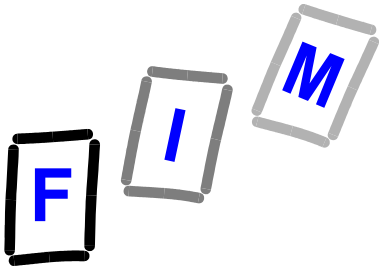
Fallstudie: Linux 2.6 O(1) Scheduler

- **Mit der Kernel-Version 2.6 (freigegeben im Dez. 2003) hat ein neuer Scheduler Einzug in den Kernel gehalten.**
- **O(1) deutet auf die asymptotische Laufzeitkomplexität des Schedulers hin:**
  - **O(n) – Laufzeit des Schedulers erhöht sich linear mit der der Anzahl der Prozesse in der READY-Queue**
  - **O(1) – Laufzeit ist UNABHÄNGIG von der Anzahl der Prozesse in der READY-Queue**
- **D. h., egal wie viele Prozesse verwaltet gerade „bereit“ sind, Scheduling dauert konstante Zeit.**



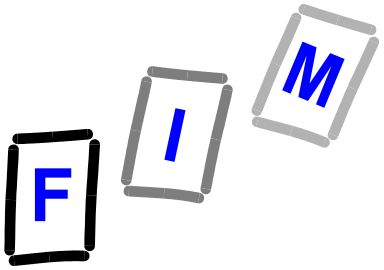
Probleme bei altem Scheduler (1)

- **SMP**
  - **1 globale Runqueue mit Single-Lock**
  - **Scheduling wird von 1 CPU für alle CPUs durchgeführt**
  - **Tasks werden anschließend verteilt**
  - **Epochensystem: Zuteilung der Tasks für eine Epoche für jede CPU. Bis zur nächsten Zuteilung muss gewartet werden, bis dass jede einzelne CPU die Tasks abgearbeitet hat.**
  - **Wenig CPU Affinität → „bouncing tasks“**



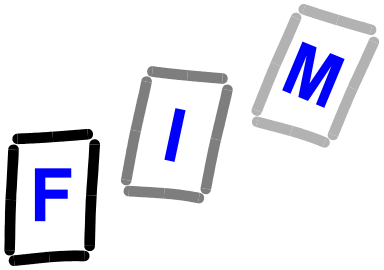
Probleme bei altem Scheduler (2)

- **Performanz**
  - **Bei Taskwechsel wird Run-Queue des gesamten Systems nach dem best-geeignetsten Task durchsucht → O(n)**
  - **Am Epochenende werden nochmals ALLE Tasks in der Ready-Queue durchsucht**
  - **Komplexe Berechnungen für nächsten Task → „Verschmutzung“ des L1-Caches**



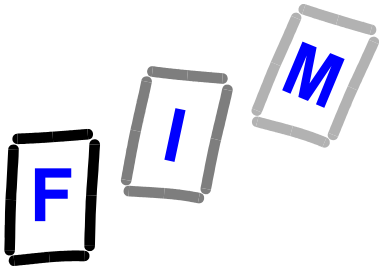
Ziele bei O(1) Scheduler

- Konstante Zeit für Task-Zuteilung → O(1)
- Fairness (kein Verhungern von Tasks)
- Gute interaktive Performanz
- Sehr gute SMP Unterstützung
- Realtime-Scheduling ermöglichen/verbessern
- Erwähnte, allgemeine Ziele bei Scheduling verbessern (Auslastung maximieren, etc.)



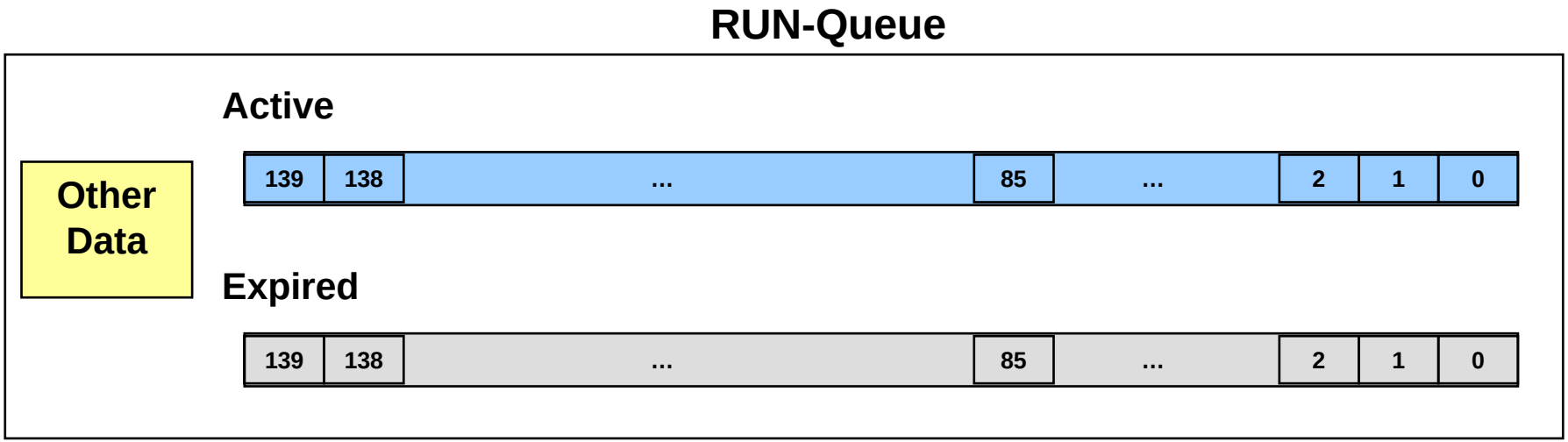
Prioritäten

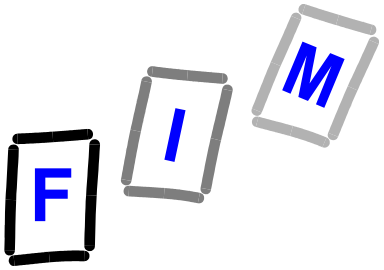
- **Es werden verschiedene Prozessprioritäten unterschieden:**
  - **Statische – abhängig vom „Nice“ Wert (wird beim Starten des Prozesses das erste Mal zugewiesen)**
  - **Dynamische Prozesspriorität auf Grund der Interaktivität des Prozesses (Kernel beobachtet Aktivitäten, CPU oder I/O lastig?)**
    - » **Die Dynamik des Prozesses beeinflusst die Priorität um maximal +/- 5 Stufen.**
    - » **-5 → Belohnung für interaktive Prozesse**
    - » **+5 → Bestrafung für CPU-lastige Prozesse**
- **Prioritäten von 0 – 139**
  - **0-99: Realtime**
  - **100 – 139: Nice-Wert -20 bis +19; „normale“ Prozesse**



Run-Queues

- Es existiert pro CPU eine eigene Run-Queue
- In der Run-Queue werden 2 Prioritätsarrays (active und expired) sowie Verweise auf die laufenden Tasks gespeichert.



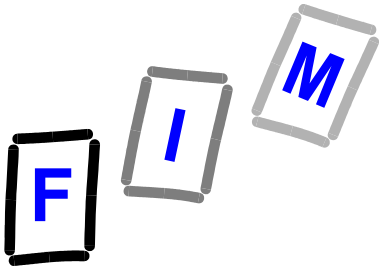


O(1) Scheduling (1)

- **Im Active-Array sind immer jene Prozesse referenziert, welche in dieser „Epoche“ noch die CPU zugeteilt bekommen, d. h., ihre Time-Slice noch nicht aufgebraucht haben.**
- **Länge der zugeteilten CPU-Zeit: Je höher die Priorität, desto länger die zugeteilte Zeiteinheit.**
- **Task zur Ausführung bestimmen:**
  - **Liste bestimmen, welche nicht leer ist und am meisten Priorität hat (Zeitdauer: t1)**
  - **Der Kopf der Liste enthält die Info für den nächsten auszuführenden Task. (Zeitdauer für das Bestimmen des Bestimmen und Auswerten des „Kopfes“ der Liste: t2)**
  - **Gesamtzeitdauer für Scheduling-Vorgang:**

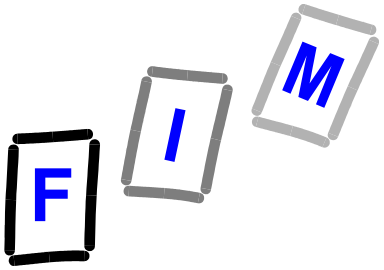
$$t = t1 + t2$$

t1, t2: O(1) → t: O(1)



O(1) Scheduling (2)

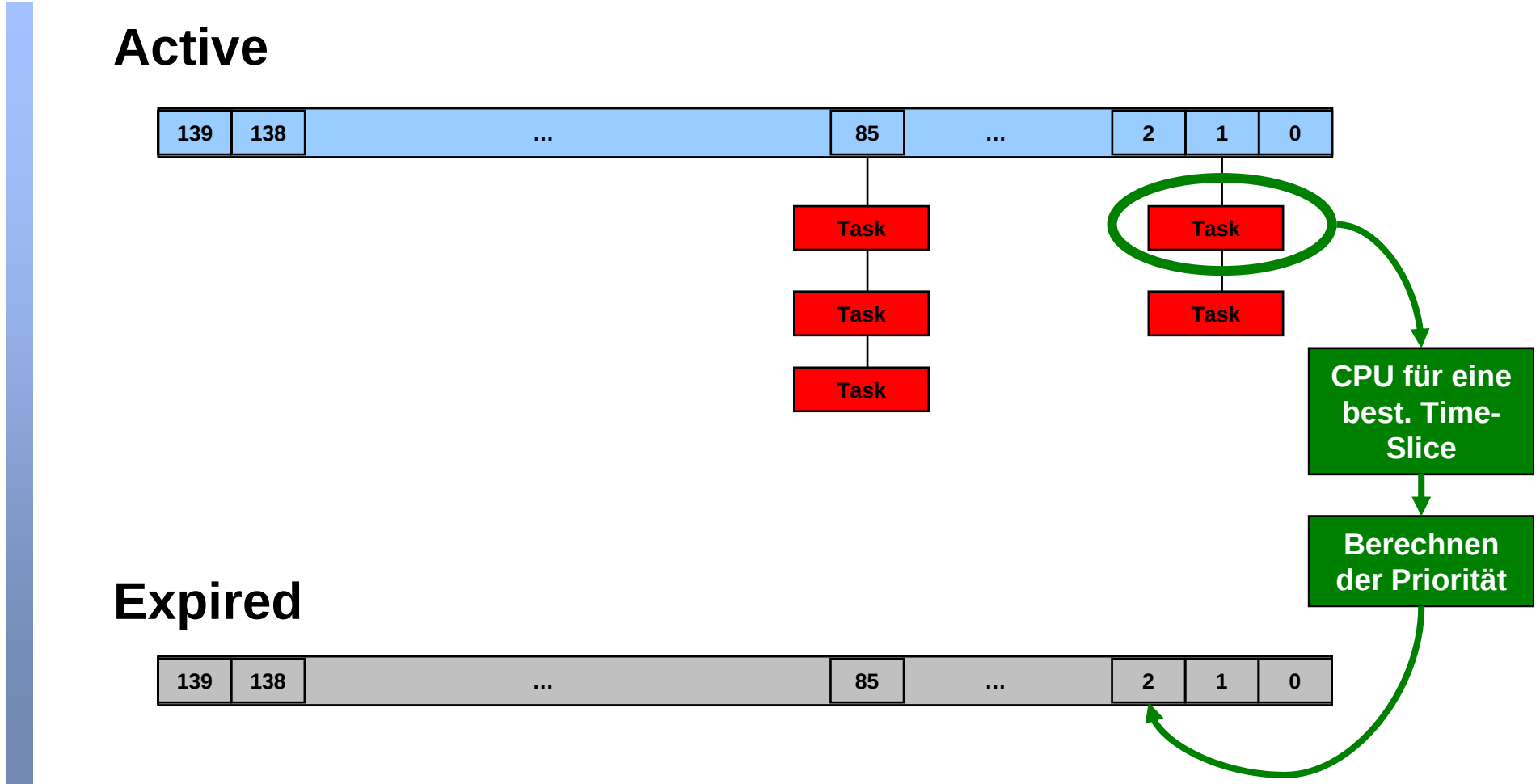
- **Länge der Zeitscheibe: Vielfaches von 20ms**
- **Ist die Zeitscheibe eines Prozesses abgelaufen, wird die Priorität für die nächste CPU-Zuteilung bestimmt und in die entsprechende Stelle im Expired-Array verschoben (vom Kopf einer Liste im active-Array an das Ende einer Liste im expired Array).**
- **Es ist auch möglich, dass ein gerade abgearbeiteter Prozess sofort wieder in das Active-Array eingehängt wird (bei hochprioritären, sehr interaktiver Prozess)**
- **Wie wird Ende der Liste in O(1) gefunden? → Doppelverkettung → head.prev == tail**
- **Ist active leer, werden die Zeiger in der Run-Queue auf active und expired ausgetauscht.**

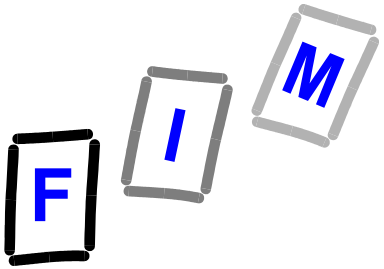


Load-Balancing bei SMP

- **Jede CPU hält zum Lastausgleich einen dedizierten „Migration-Thread“.**
- **Dieser greift dann ein, wenn die Run-Queues zwischen den CPUs sehr unausgeglichen sind (>25%). In unregelmäßigen Abständen wird überprüft, ob Lastausgleich (Verschieben von Prozessen auf eine andere CPU) durchgeführt werden soll.**
- **Beim Verschieben von Prozessen müssen (mind.) 2 Run-Queues gesperrt werden. Beim „normalen“ Scheduling Vorgang muss nur die Run-Queue der jeweiligen CPU gesperrt werden.**

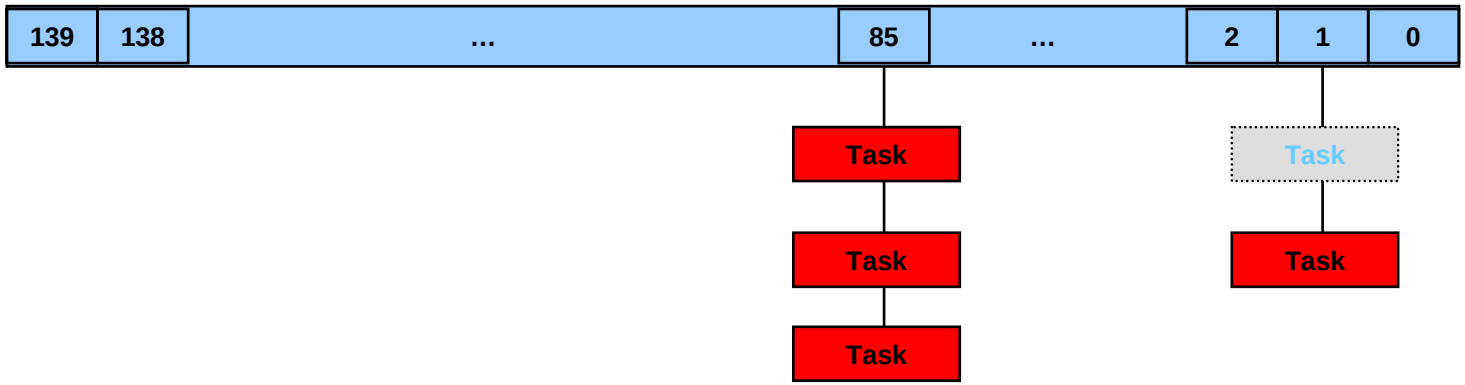
Run Queue (1)


Active
139
138
...
85
...
2
1
0
Task
Task
Task
Task
Task
CPU für eine best. Time-Slice
Berechnen der Priorität
Expired
139
138
...
85
...
2
1
0

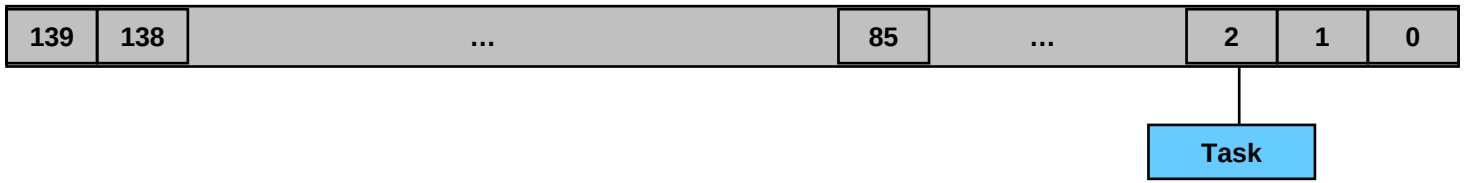


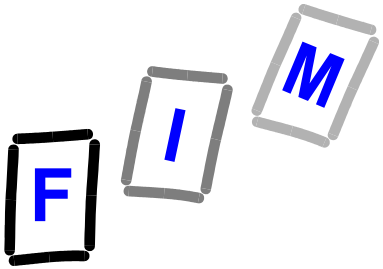
Run Queue (2)

Active

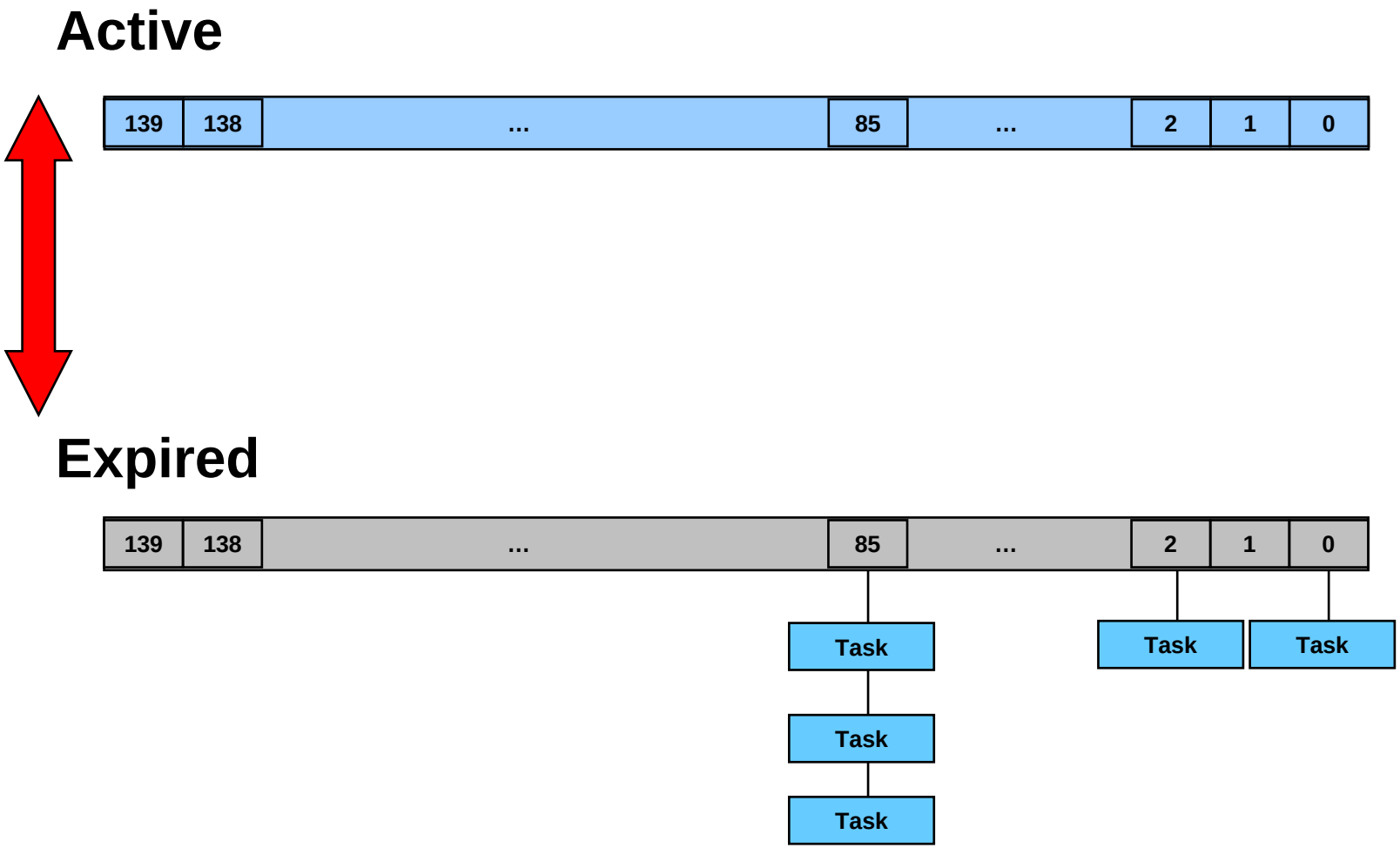


Expired





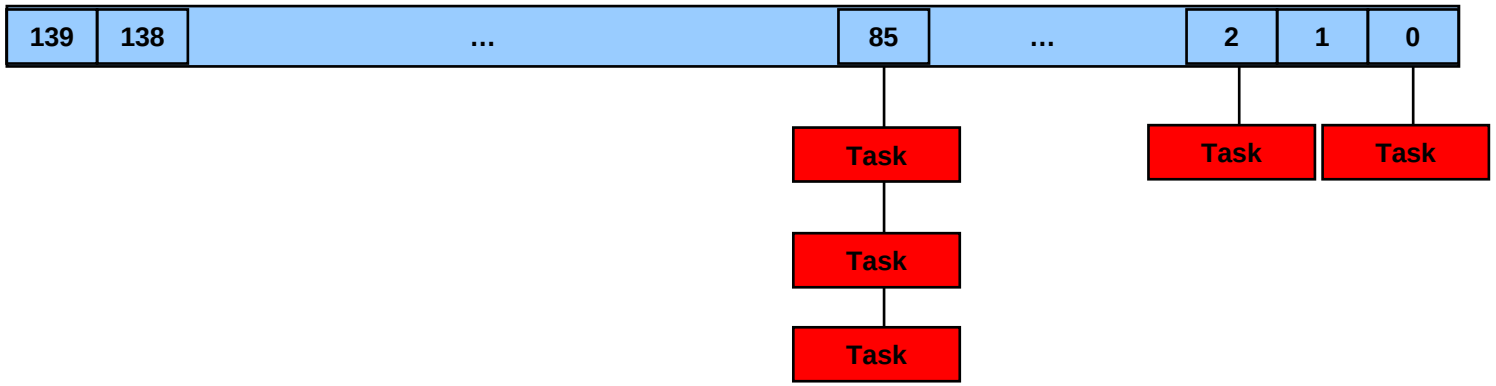
Run Queue (3)

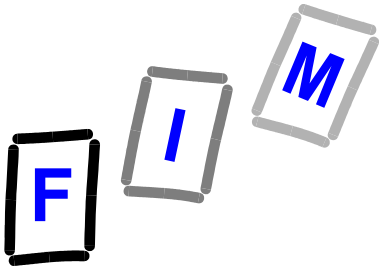

Active
139
138
...
85
...
2
1
0
Expired
139
138
...
85
...
2
1
0
Task
Task
Task
Task
Task

Run Queue (4)

Expired

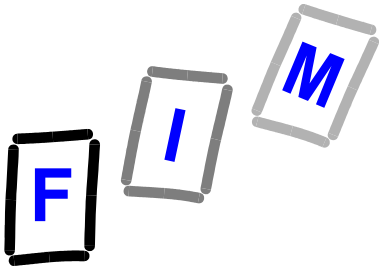
Active





Übungsvorbesprechung

Übung 4

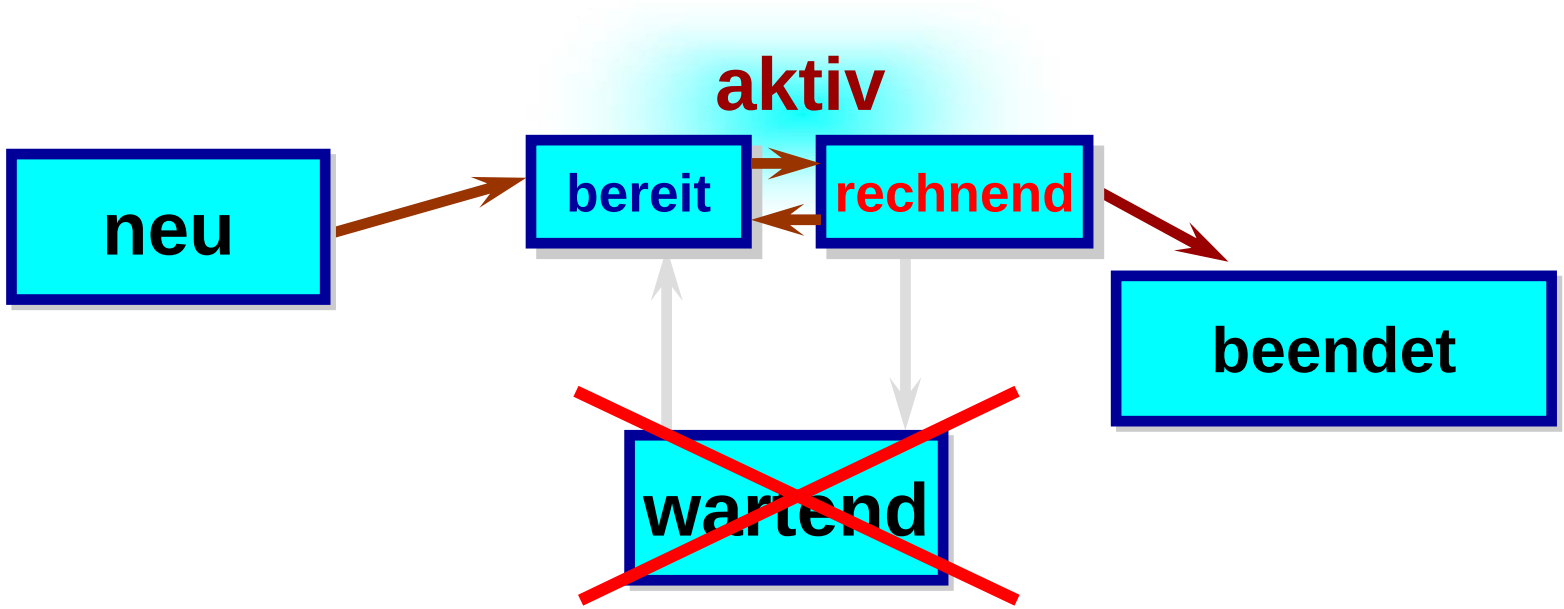


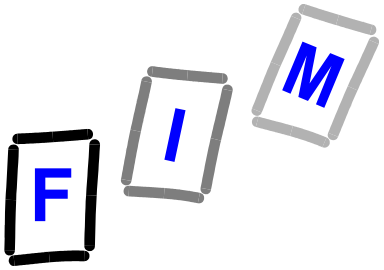
Allgemeines zur Übung (1)

- **Sie bekommen ein „fertiges“ Framework zur Simulation von CPU Scheduler Algorithmen zur Verfügung gestellt.**
- **Auf dieses aufbauend führen Sie die Übungsaufgaben durch.**
- **Download von der LVA Homepage**
- **Simulation:**
  - **Scheduler für zwei parallel arbeitende CPUs**
  - **Nur Prozesse (und keine untergeordneten Threads) werden simuliert.**
  - **Die einzigen Ressourcen, auf welche Prozesse warten müssen, sind die CPUs**

Allgemeines zur Übung (2)

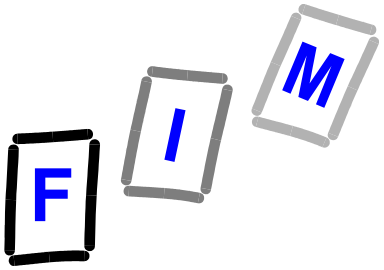
- **Der Zustand WAITING entfällt quasi. Prozesse sind immer bereit, ausgeführt zu werden → Vereinfachung!**





Übungshinweise

- **Zum Framework selbst:**
  - **Obwohl Sie Source-Code des Frameworks erhalten, darf dieser nicht geändert werden (denken Sie z. B. an kommerzielle Frameworks, wo Sie nur binaries erhalten). Die vorgesehenen Schnittstellen müssen also ausreichen.**
  - **Entnehmen Sie die Schnittstellen beispielsweise aus der JavaDoc!**
  - **Allgemein: Bei Verwendung von Frameworks muss eine gewisse Zeit für den „Kennenlernvorgang“ miteinberechnet werden.**

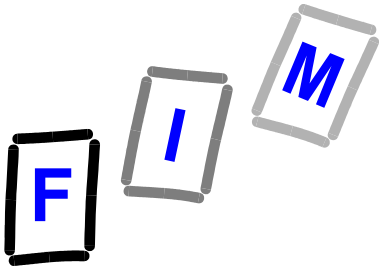


Java Reflection (1)

- **Wie erstellt man eine konkrete Instanz einer Klasse?**
  - **Zur Compilezeit ist bereits bekannt sein, von welcher Klasse ich eine Instanz erzeugen will.**

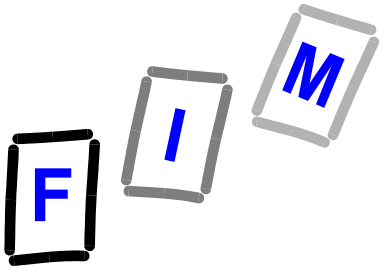
    Vector v = new java.util.Vector();
- **Problemstellung:**
  - **Ich erfahre erst zur Laufzeit, welche Klasse ich instanziieren möchte.**
  - **Klassenname liegt vollständig qualifiziert (also inklusive Paketname) als String vor**

    String className = „java.util.Vector“;



Java Reflection (2)

- **Wie kann man eine Instanz von einer Klasse erzeugen, von der man nur den Namen kennt?**
- ***Lösung: JAVA Reflection API***
- **Mit Hilfe der Reflection Klassen können ab JDK 1.1 zur Laufzeit Metainformationen über Klassen abgerufen werden.**
- **Reflection ermöglicht:**
  - **Bestimmen der Eigenschaften einer Klasse**
  - **Erzeugen eines Objekts**
  - **Verändern von Werten**
  - **Aufrufen von Methoden**
  - **Erzeugen und Verändern eines Arrays**



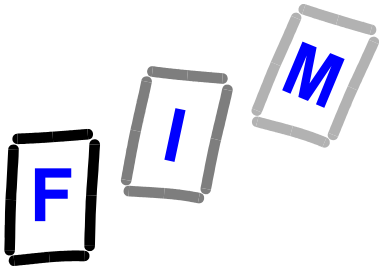
Java Reflection (3)

- **Schlüssel für Reflection**
  - Klasse java.lang.Class
  - Package java.lang.reflect
- **Klassenobjekt für eine Variable bestimmen**

  Vector v = new Vector();

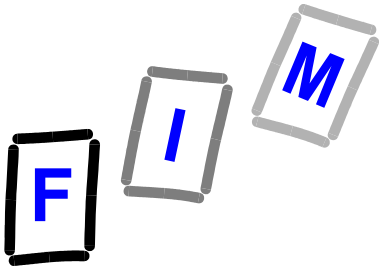
  Class classOfV = v.getClass();
- **Klassenobjekt durch Klassennamen bestimmen**

  Class vectorClass = Class.forName(„java.util.Vector“);
- **Lesen Sie die JavaDoc für die Klasse java.lang.Class**



Aufgabe 1

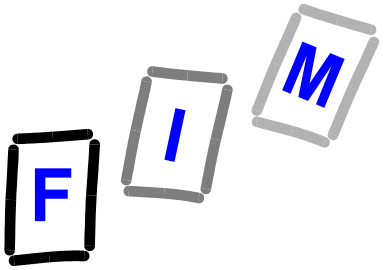
- **Ihre Aufgabe ist es, dynamisch Objekte von Klassen zu erzeugen. Den Klassennamen bekommen Sie als String.**
- **Weiters müssen Sie überprüfen, ob eine bestimmte Klasse überhaupt für den Anwendungsfall gültig ist, oder nicht (Ist die Klasse ein Subtyp einer anderen, bekannten Klasse)**



Aufgabe 2

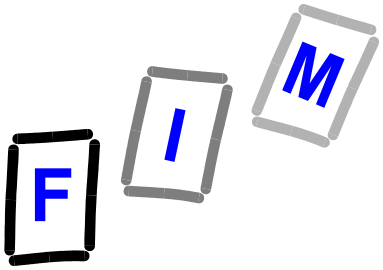
- **Implementieren Sie einen RoundRobin Scheduler für das vorgegebene Framework (ohne Prioritäten-Berücksichtigung)**

- **TIPP: Sehen Sie sich den Scheduler an, der bereits inkludiert ist (FifoBatchScheduler)**

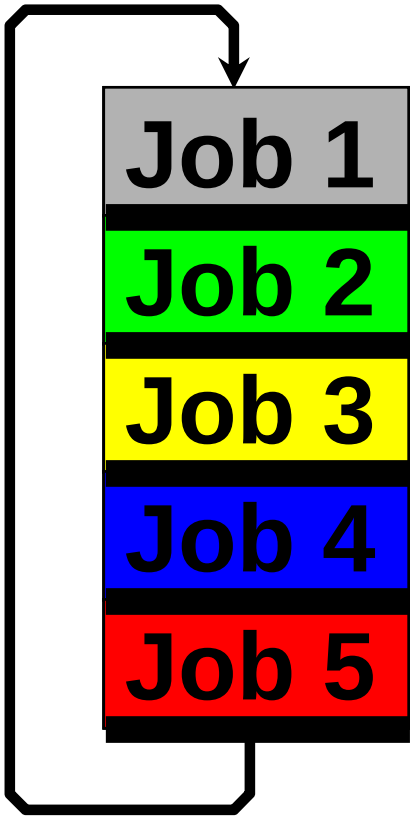


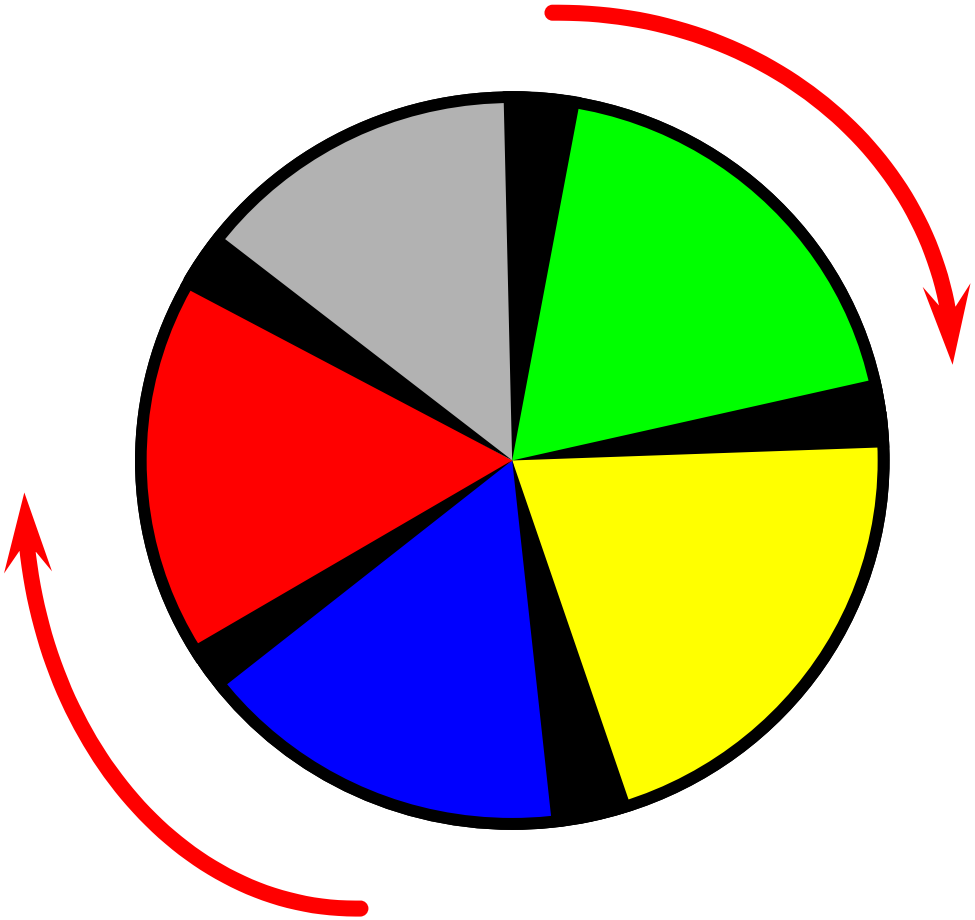
Zeitscheibenverfahren (Round Robin)

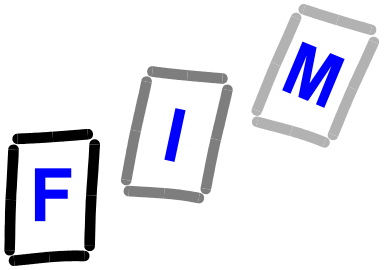
- **eine kleine Zeiteinheit, *Zeitquantum*, (auch: Zeitscheibe) wird definiert.**
- **das Zeitquantum ist normalerweise $10 < t < 100$ msec**
- **„ready“-Schlange: Zyklische Warteschlange**
- **Scheduler arbeitet die „ready“-Schlange reihum, zyklisch ab**
- **weist die CPU jedem Prozess für eine bestimmte Zeit zu: (n>=1 Zeitscheiben)**
- **dann wird unterbrochen, und der nächste Prozess kommt dran**



Round Robin (1)





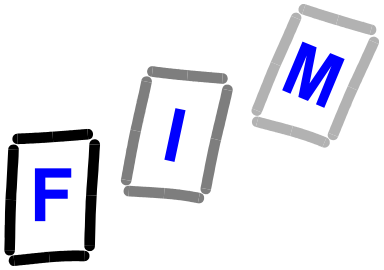


Round Robin (2)

- **Das Rundlauf-Verfahren (round robin, RR, Zeitscheiben-Verfahren)**

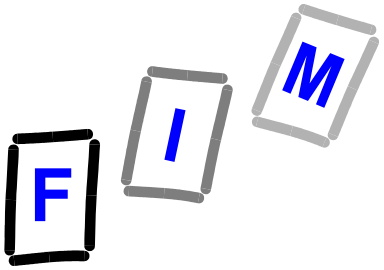
  eignet sich vor allem für Timesharing-Systeme
- **Moderne OS verwenden RR auch ohne eine TS-Funktion anzustreben:**
  - **unterbrechende Prioritätssteuerung**
  - **innerhalb einer Klasse von "gleichrangigen" Prozessen aber RR**
  - **Verwendung von variablen Zeit-Quanten**

  Beispiele:
  - **Windows 2000, XP, Linux, ...**



Aufgabe 3

- **Überlegen Sie sich eine eigene Strategie für Scheduling mit Prioritäten, von der Sie glauben, dass sie optimal arbeitet.**
- **Bedenken Sie, dass Ihr Algorithmus einerseits für eine optimale Nutzung der Ressourcen sorgen soll, andererseits aber auch möglichst effizient und schnell arbeiten muss.**
- **Der Benutzer soll zur Laufzeit die Möglichkeit haben, den Scheduler dynamisch zu beeinflussen**



Aufgabe 4

- **Implementierung von diversen Statistikberechnungen**