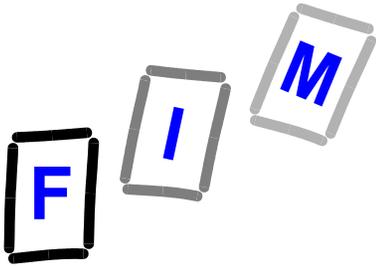


SS 2004

KV Betriebssysteme

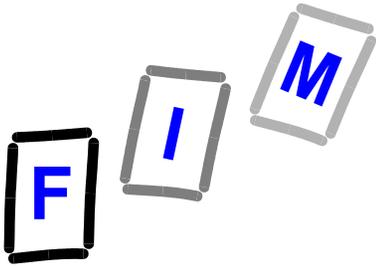
Multi-Processing & Scheduling

© A. Putzinger, FIM 2004



Wiederholung Multi Tasking

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



Non-preemptiv

Non-preemptives Scheduling:

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

Strategien:

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)

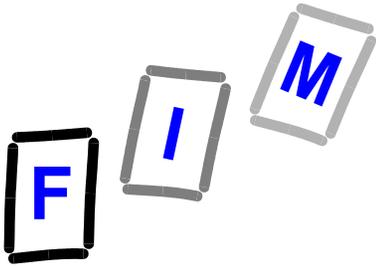
- Kürzere Jobs werden bevorzugt
- gesamte Ausführungszeit: $3+7+17$ ZE = 27 ZE
- mittl. Ausführungszeit $27/3$ ZE = 9 ZE

➤ HRN (Highest Response Ratio Next)

- Jobs mit kurzen Bedienzeiten bzw. langen Antwortzeiten werden bevorzugt (Antwortzeit/Bedienzeit). Kein Prozess verhungert, da die Antwortzeiten mit der Wartezeiten zunehmen.

➤ PS (Priority Scheduling)

- Jobs werden nach ihrer Priorität behandelt; Gefahr: Verhungern von Prozessen



Preemptiv

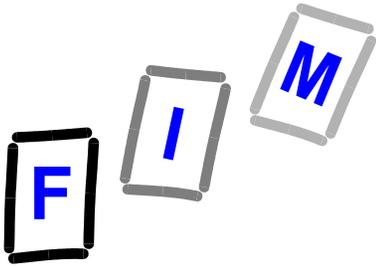
Preemptives Scheduling

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)**

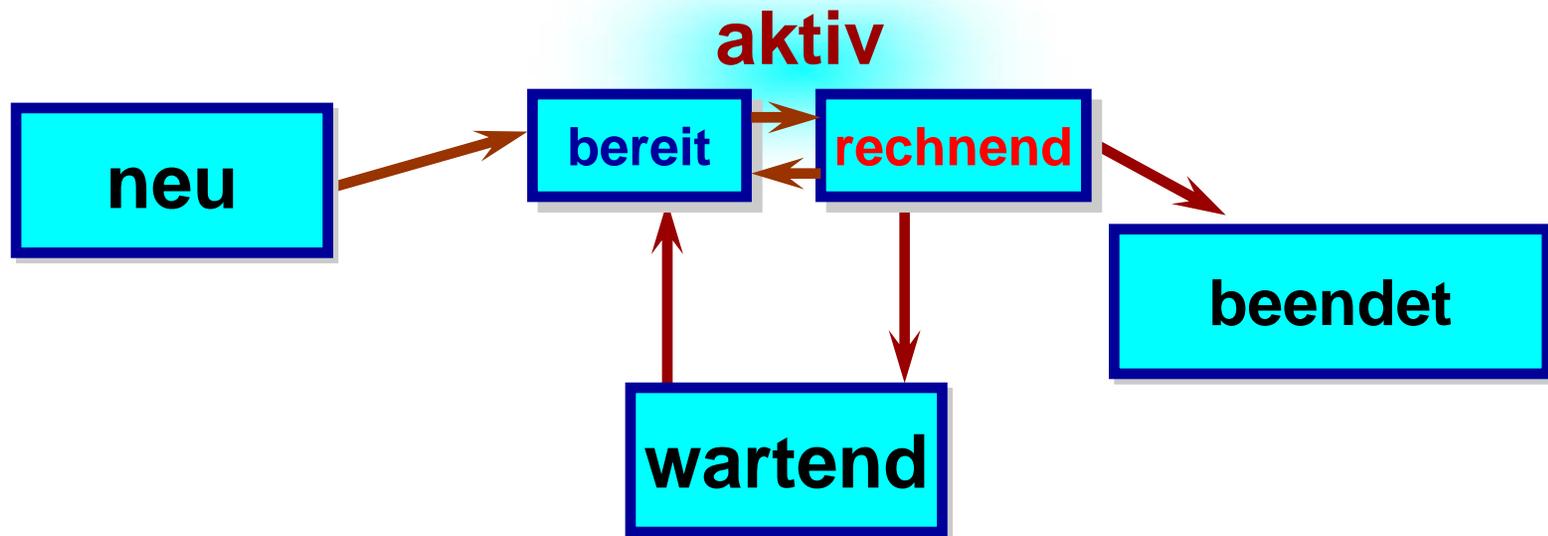


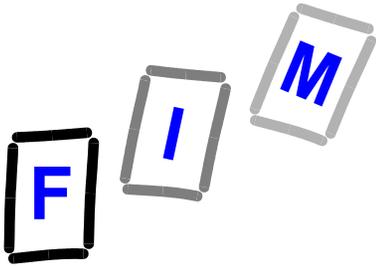
Prozesszustände Grobeinteilung

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

➤ Antwortzeit

Die Zeit zwischen Eingabe und Antwort des Rechners sollte minimal sein.

➤ + subjektive Ziele bei CPU Scheduling

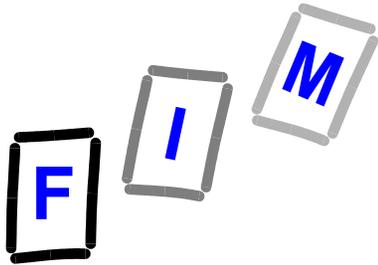
»Der Benutzer soll das Gefühl haben, dass das System „möglichst sofort“ auf seine Eingaben reagiert

»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.

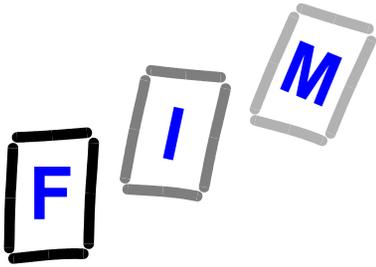
Es gibt keinen idealen Scheduling Algorithmus, der alle Scheduling Ziele erfüllt.



Microsoft
Windows NT
Server

Fallstudie: WinNT Scheduler (1)

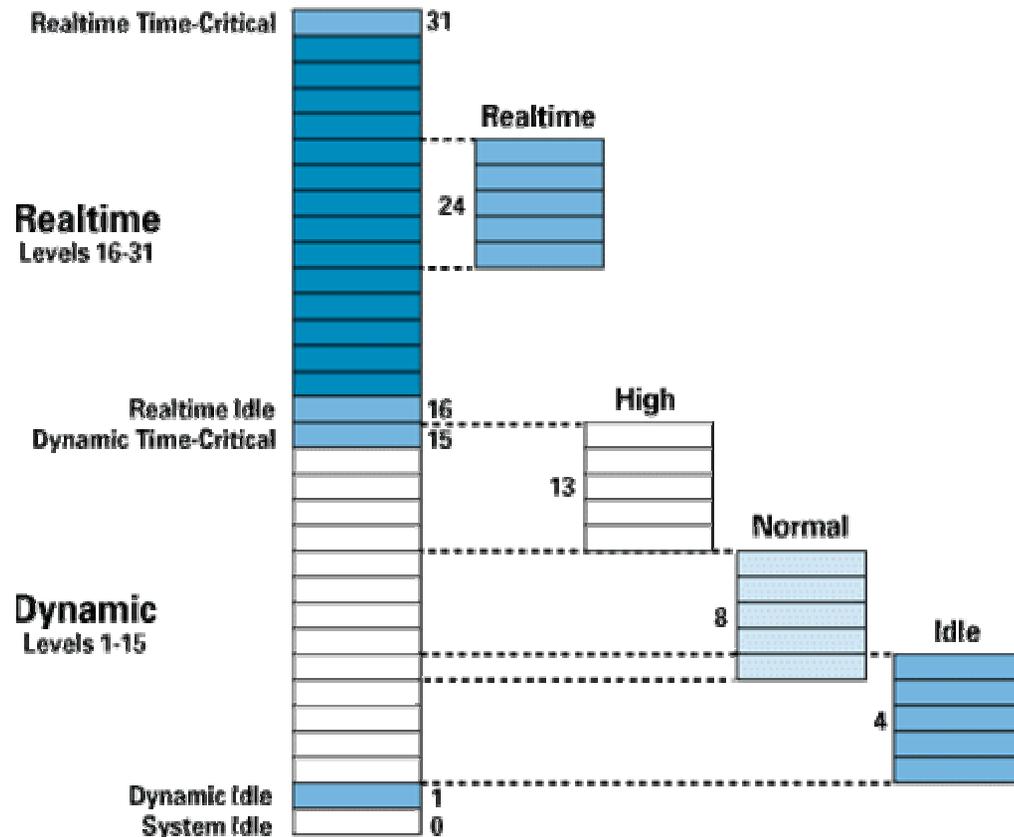
- **Windows NT führt Scheduling auf Basis von Threads durch**
- **NT verwaltet Prioritäten von 1 bis 31. Priorität 0 ist für den „System Idle“ Thread reserviert.**
- **16-31: Realtime Prioritäten. Administratorrechte erforderlich.**
- **Priorität 1-15: Dynamische Prioritäten für Std.-Applikationen**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

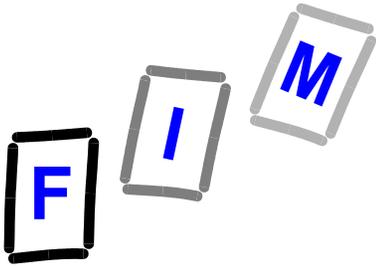


WinNT Prioritäten (1)

■ 4 Prioritätsklassen

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

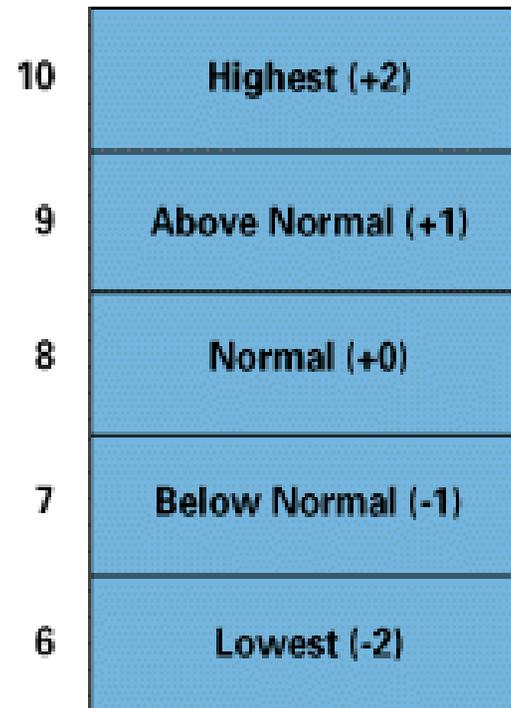


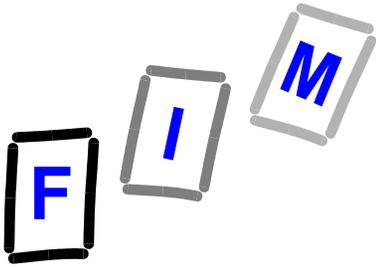


WinNT Prioritäten (2)

- **Relative Priorität**

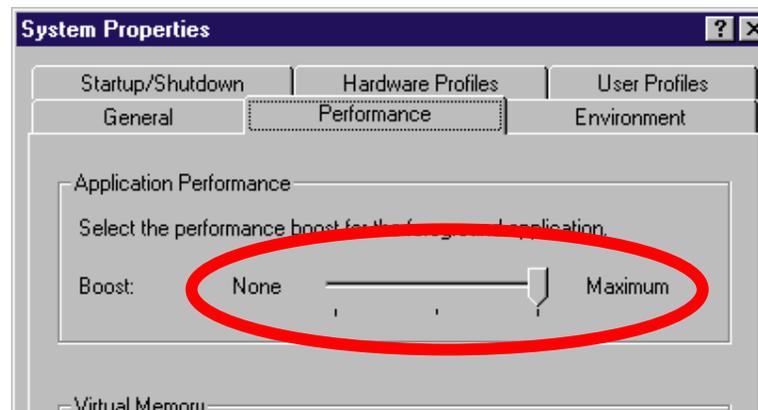
➤ **+/- 2**

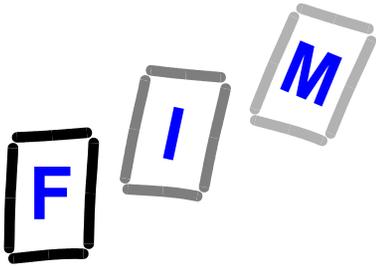




WinNT Scheduler (1)

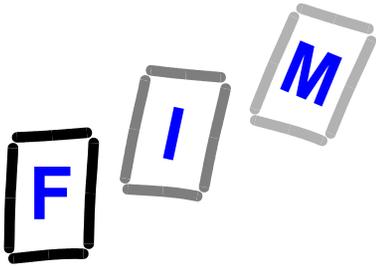
- Scheduler vergibt Rechenzeit an Threads (Quantum = gewisse Zeit an CPU-Nutzung)
- Ein Quantum ist sehr kurz
 - WinNT Server: 120 ms
 - WinNT Workstation: 20 – 60 ms, abhängig, ob es sich um eine Back- oder Foreground Applikation handelt





WinNT Scheduler (2)

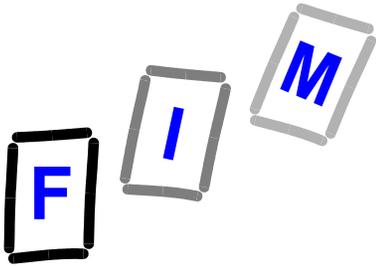
- **Der NT Scheduler muss zu folgenden Zeitpunkten eine Entscheidung treffen:**
 1. **Das zugewiesene Quantum eines Threads läuft ab**
 2. **Ein Thread wartet auf ein Ereignis**
 3. **Ein Thread wird bereit, ausgeführt zu werden**



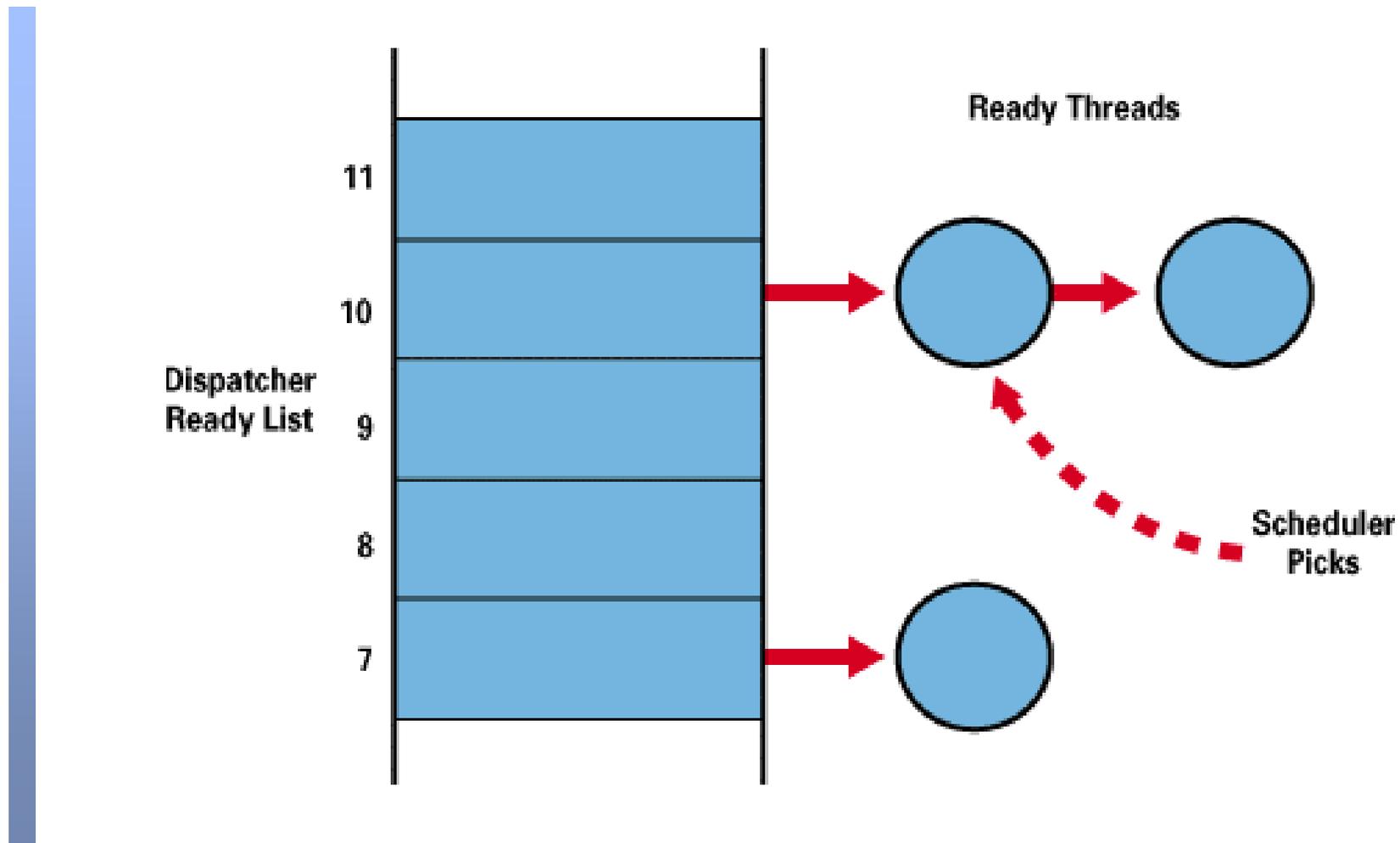
WinNT Scheduler FindReadyThread (1)

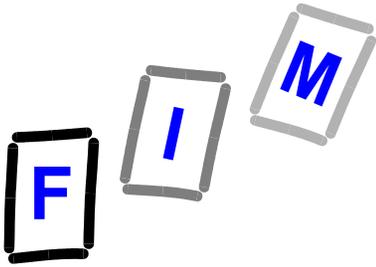
Fall 1: Zugewiesenes Quantum läuft ab

- 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.



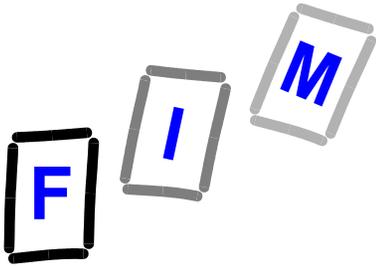
WinNT Scheduler FindReadyThread (2)





WinNT Scheduler (3)

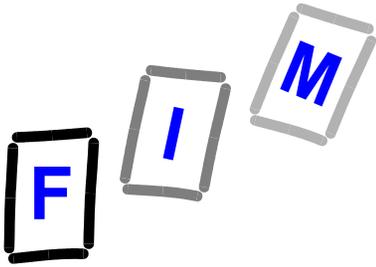
- **Der NT Scheduler muss zu folgenden Zeitpunkten eine Entscheidung treffen:**
 1. Das zugewiesene Quantum eines Threads läuft ab
 - 2. Ein Thread wartet auf ein Ereignis**
 3. Ein Thread wird bereit, ausgeführt zu werden



WinNT Scheduler

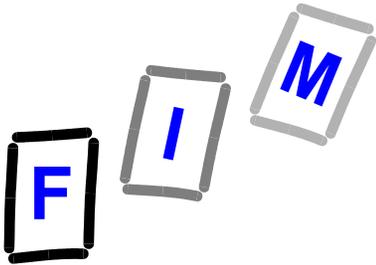
Warten auf Event

- **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**



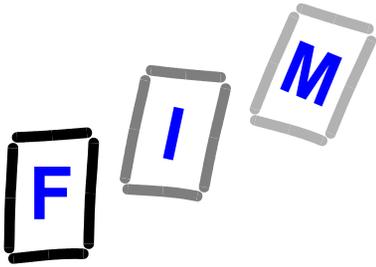
WinNT Scheduler (4)

- **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**



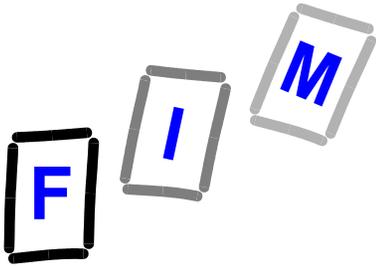
WinNT Scheduler Bereit für Ausführung

- 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.



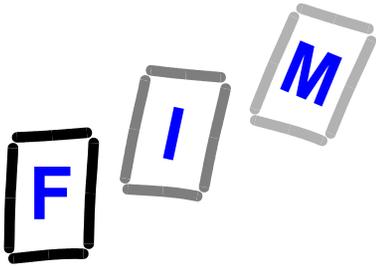
WinNT Scheduler Starvation

- 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



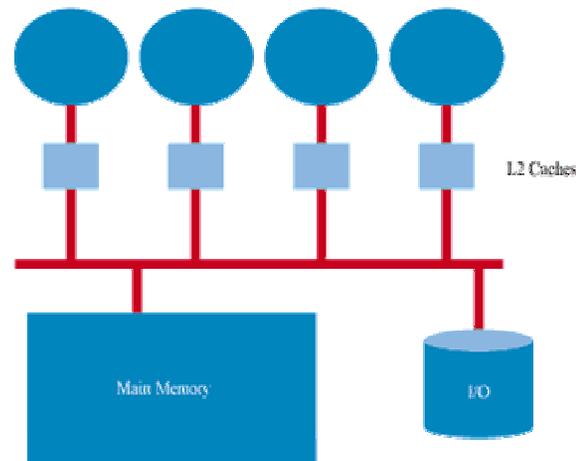
WinNT Scheduler Boosting

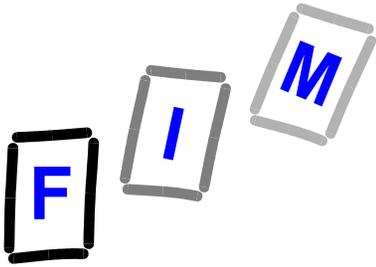
- Bisher wurde der Eindruck vermittelt, dass die **Priorität eines Threads über dessen Lebenszeit konstant bleibt, bis dass sie explizit vom Benutzer verändert wird**
- **In Wirklichkeit wird die Priorität vom System aber oft für kurze Zeit verändert. Beim Übergang von WAITING auf READY beispielsweise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



WinNT als Mehrprozessor OS

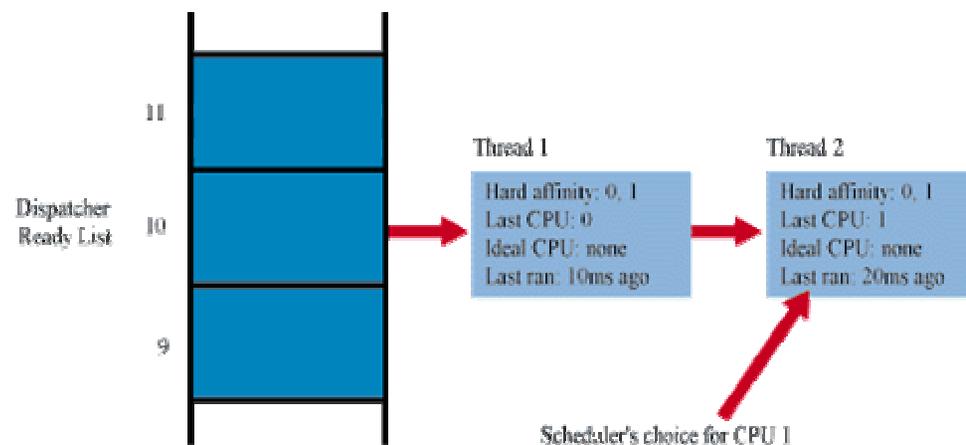
- **SMP – Symmetric Multi Processing**
 - Mehrere, identische CPUs mit privatem L1 und L2 Cache
 - Alle CPUs haben gleiche Zugriffsrechte auf Speicher und Peripherie
 - NT kann auf jeder CPU (oder auch auf mehreren) laufen

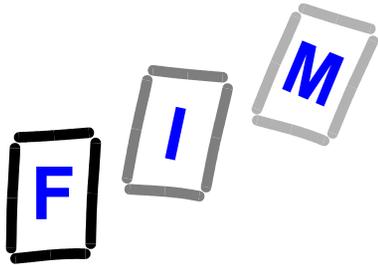




WinNT Mehrprozessor OS (2)

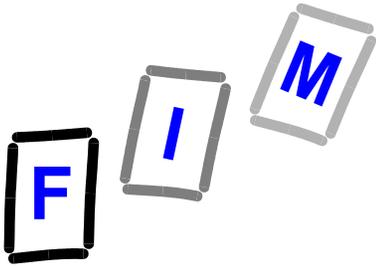
- Begriff der „Affinität“
 - **Soft-affinity:** Prozess soll – wenn möglich – auf einer bestimmten CPU ausgeführt werden (aus Cache Gründen)
 - **Hard-affinity:** Prozess muss auf einer bestimmten CPU ausgeführt werden





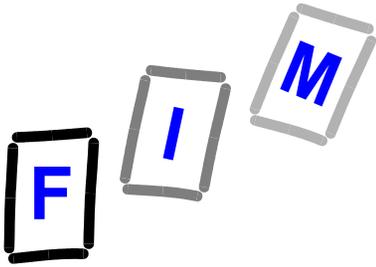
Fallstudie: Linux 2.6 O(1) Scheduler

- Mit der Kernel-Version 2.6 (freigegeben im Dez. 2003) hat ein neuer Scheduler Einzug in den Kernel gehalten.
- O(1) deutet auf die Laufzeitkomplexität des Schedulers hin.
 - O(N) – Laufzeit des Schedulers ist abhängig von der Anzahl der Prozesse in der READY-Queue
 - O(1) – Laufzeit ist unabhängig von der Anzahl der Prozesse in der READY-Queue
- Prozessinformationen werden in einer doppelt verketteten Liste verwaltet.



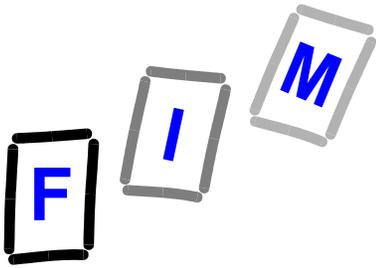
O(1) Scheduler (2)

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



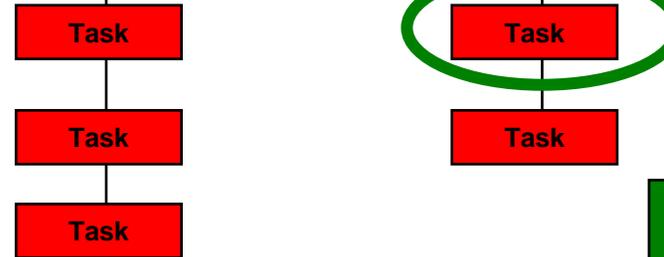
O(1) Scheduler (3)

- 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.
- Ist die Zeitscheibe eines Prozesses abgelaufen, wird die neue Priorität und die nächste Time-Slice berechnet, und der Prozess wird von active nach expired verschoben (am Ende der Liste, durch Doppelverkettung (head→prev) trotzdem O(1)).
- Ist active leer, werden die Zeiger active und expired ausgetauscht.



Run Queue (1)

Active

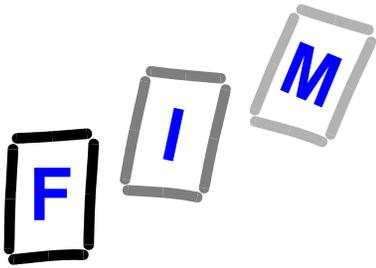


CPU für eine best. Time-Slice

Berechnen der Priorität

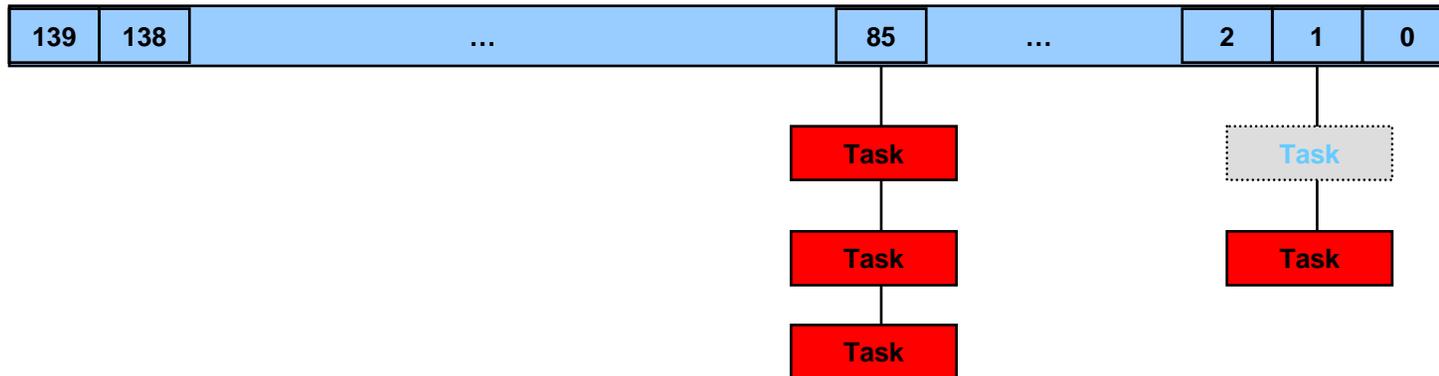
Expired



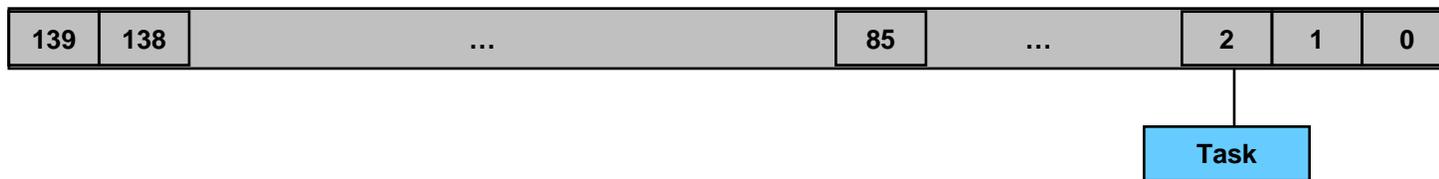


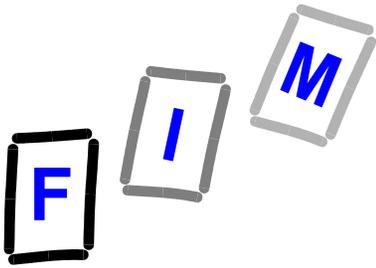
Run Queue (2)

Active

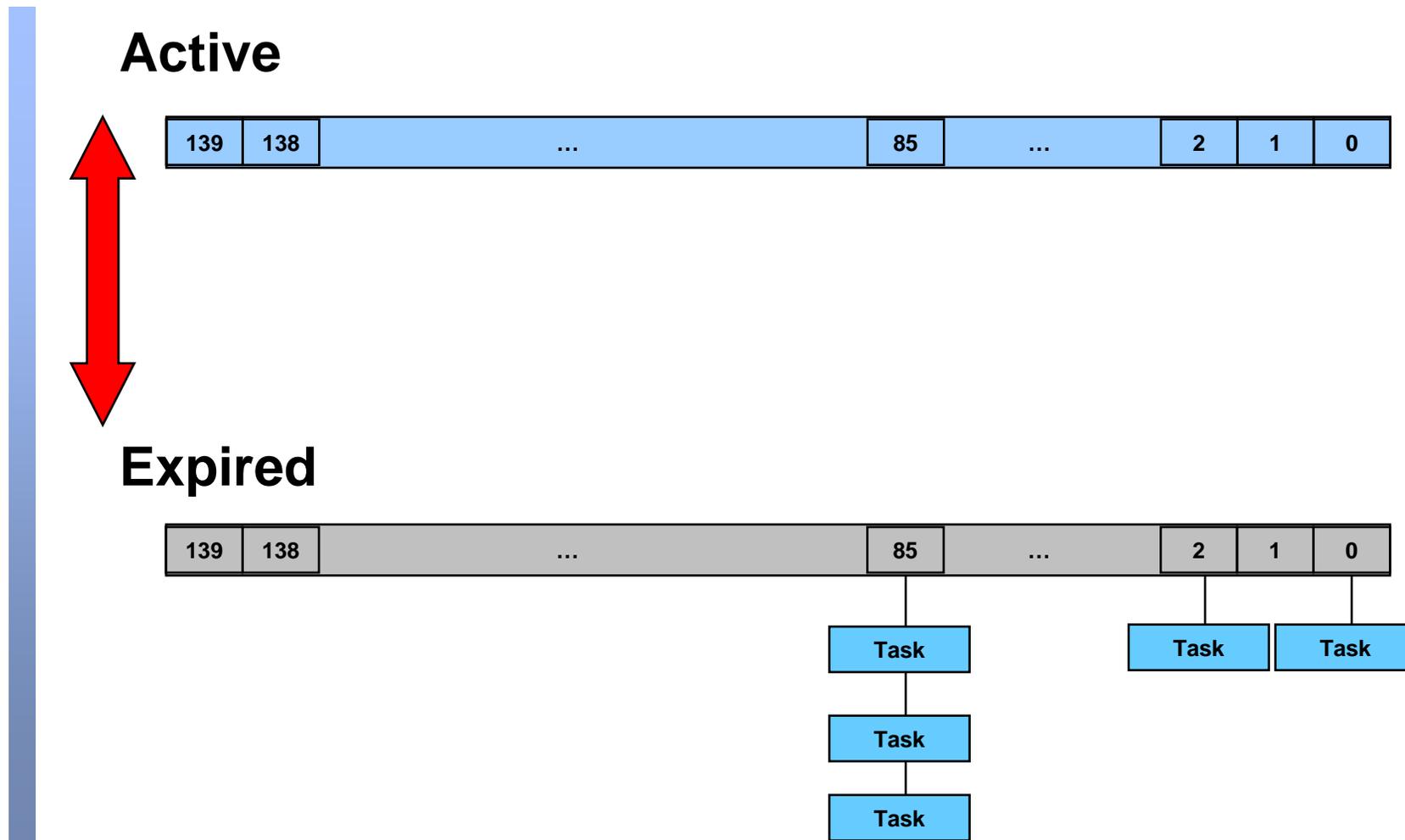


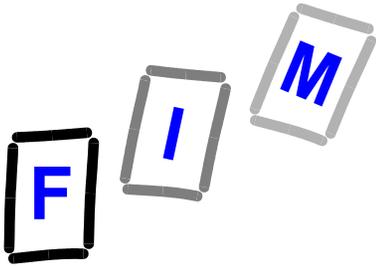
Expired





Run Queue (3)



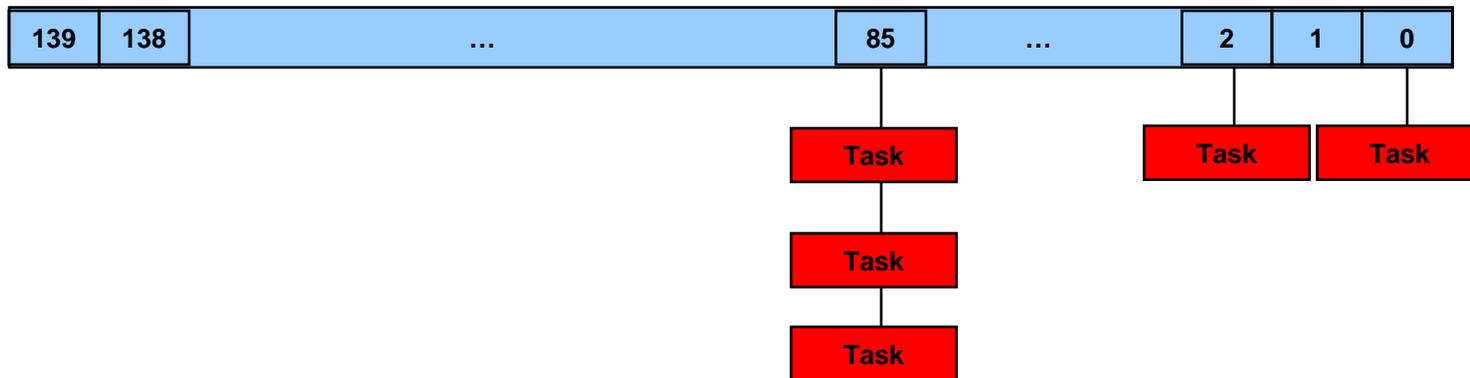


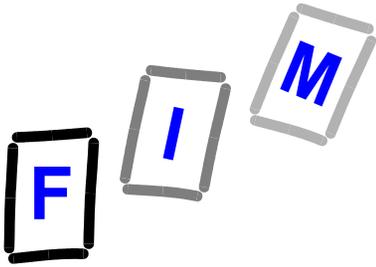
Run Queue (4)

Expired



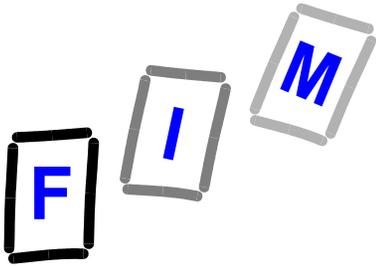
Active





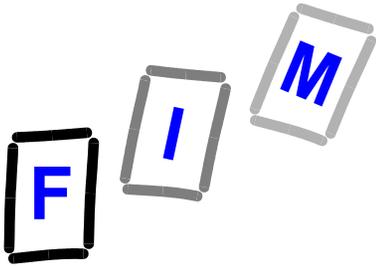
Übungsvorbesprechung

Übung 4



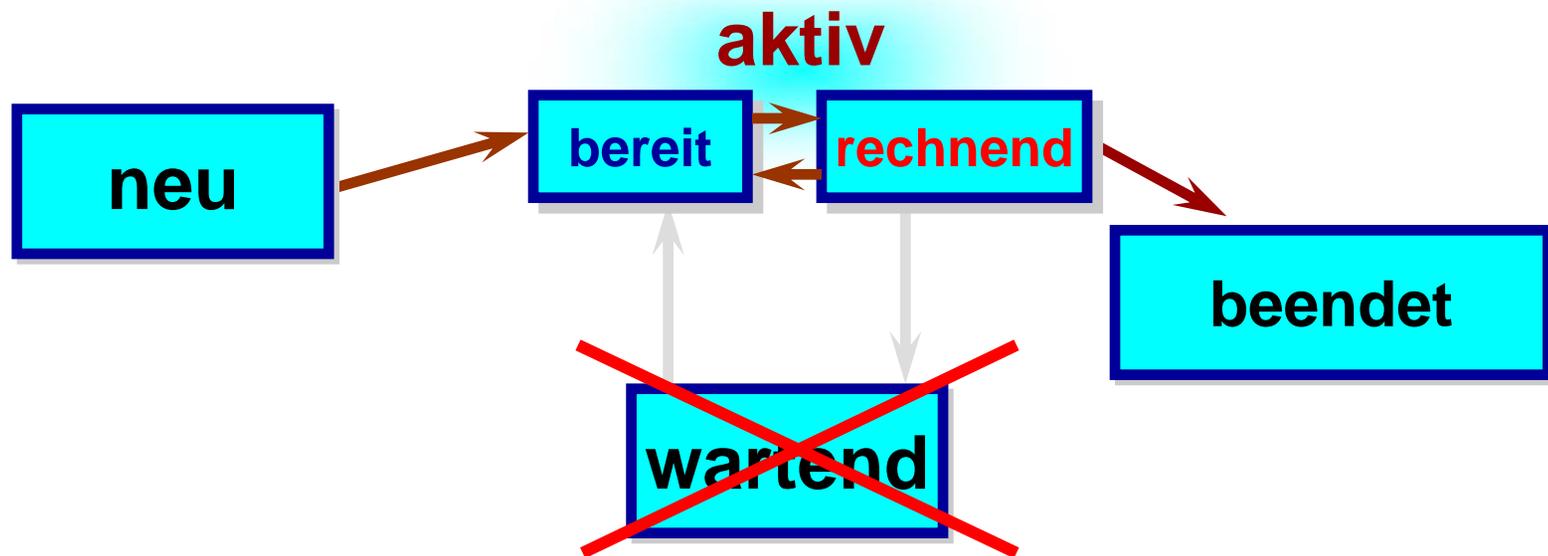
Allgemeines zur Übung (1)

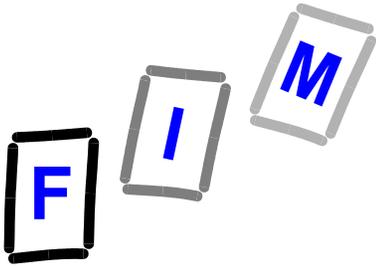
- Sie bekommen ein „fertiges“ Framework zur Simulation von CPU Scheduler Algorithmen zur Verfügung gestellt.
- Auf dieses aufbauend führen Sie die Übungsaufgaben durch.
- Download von der LVA Homepage
- Simulation:
 - Scheduler für zwei parallel arbeitende CPUs
 - Nur Prozesse (und keine untergeordneten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



Allgemeines zur Übung (2)

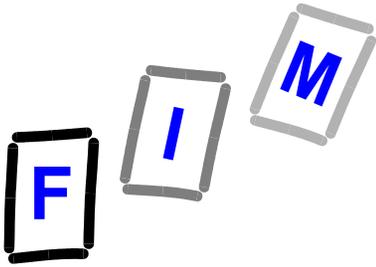
- Der Zustand **WAITING** entfällt quasi. Prozesse sind immer bereit, ausgeführt zu werden → Starke Vereinfachung





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.**
- **TIPP: Beginnen Sie rechtzeitig mit der Übung.**



Java Reflection (1)

- **Wie erstellt man eine konkrete Instanz einer Klasse?**

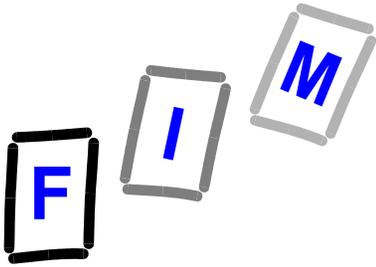
- **Zur Compilezeit ist bereits bekannt sein, von welcher Klasse ich eine Instanz erzeugen will.**

`Vector v = new java.util.Vector();`

- **Problemstellung:**

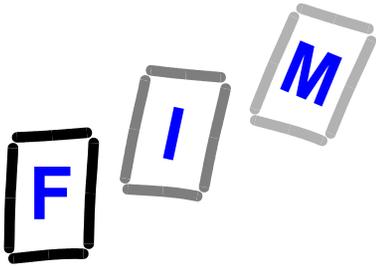
- **Ich erfahre erst zur Laufzeit, welche Klasse ich instanziiieren möchte.**
- **Klassenname liegt vollständig qualifiziert (also inklusive Paketname) als String vor**

`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

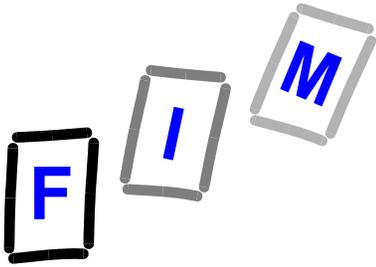


Java Reflection (3)

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

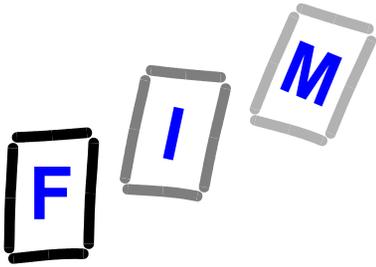
```
Vector v = new Vector();  
Class classOfV = v.getClass();
```
- **Klassenobjekt durch Klassennamen bestimmen**

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



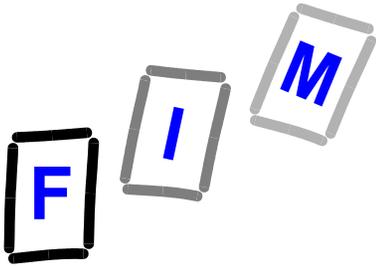
Aufgabe 1

- 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)**

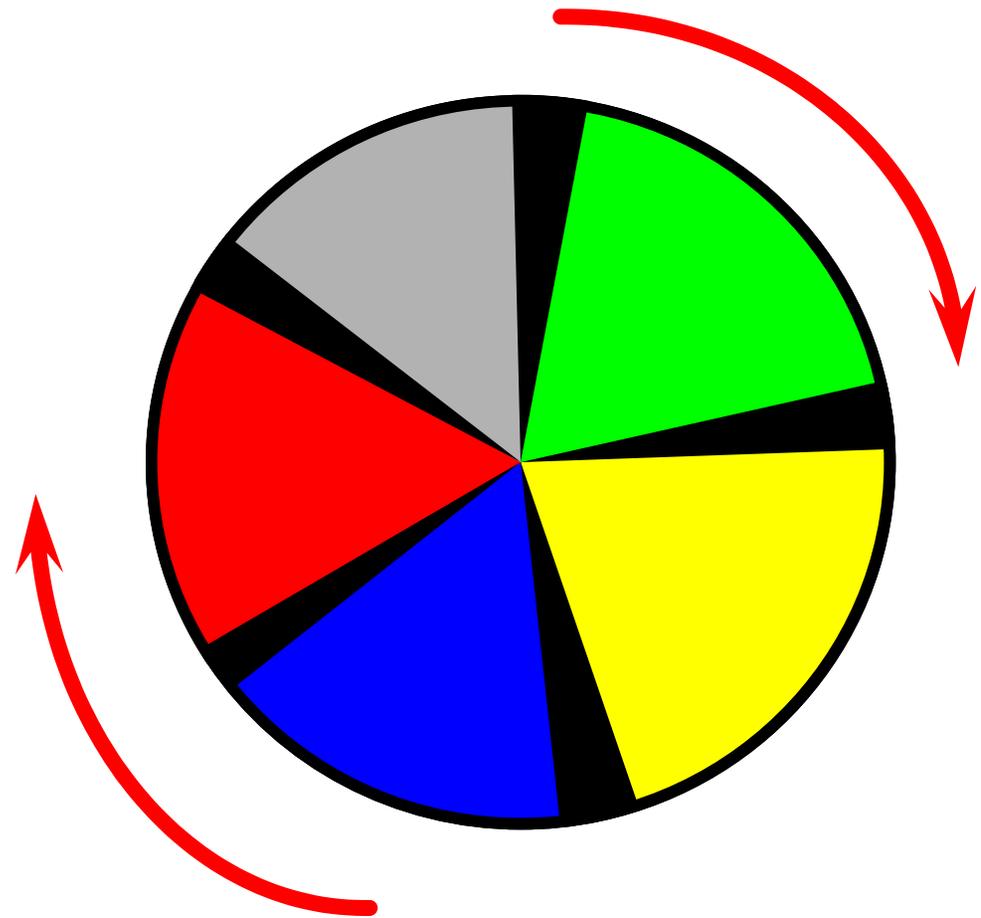
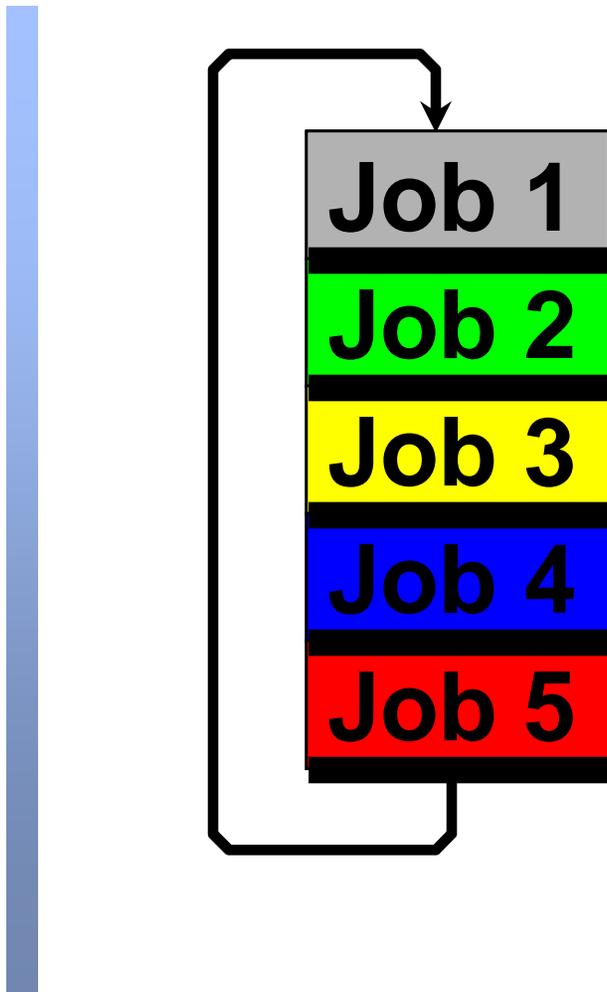


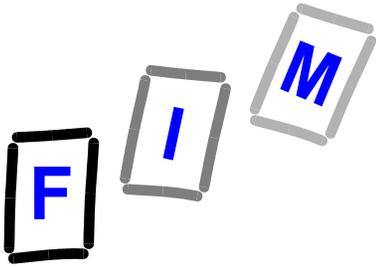
Zeitscheibenverfahren (Round Robin)

- 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: ($n \geq 1$ Zeitscheiben)
- dann wird unterbrochen, und der nächste Prozess kommt dran

F I M

Round Robin (1)



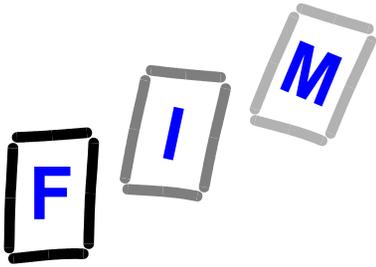


Round Robin (2)

- **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**

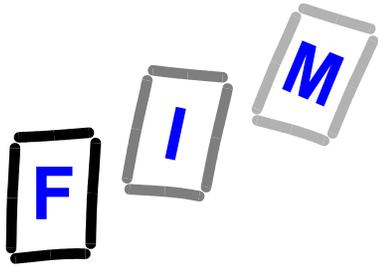
Beispiele:

- **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**