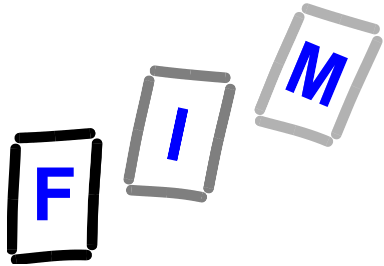
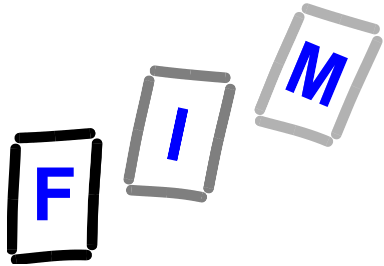


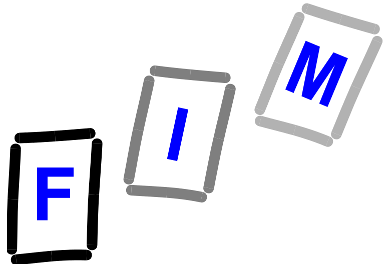
Betriebssysteme

Kap F: CPU-Steuerung
CPU-Scheduling



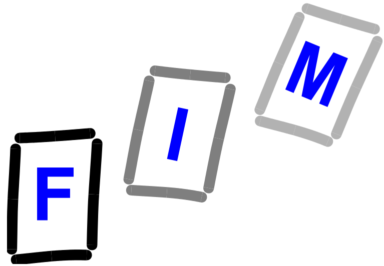
termini technici

- Der englische Fachausdruck **scheduler** wurde „eingedeutscht“:
  - ➡ Der Scheduler
- Für „scheduling“ ist im Deutschen auch zu verwenden:
  - ➡ Ablaufplanung
- Im Zusammenhang mit CPU
  - ➡ CPU Steuerung



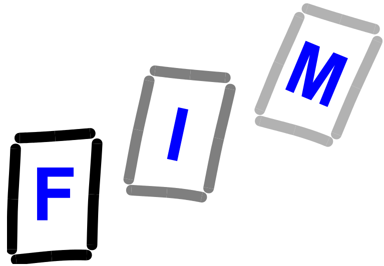
Steuerprogramm (CPU- Scheduler)

- Ein Steuerprogramm ist verantwortlich für
  - Auswahl der Prozesse zur Ausführung
    - Steuerung der Prozesse
  - Zuteilung von Betriebsmitteln an die Prozesse
    - CPU-Steuerung
    - ....
- Grundsätzlich zwei Ebenen:
  - Langzeitsteuerung (Long Term)
  - Kurzzeitsteuerung (Short Term)



Zum Vergleich
Zustandsdiagramm

Das Zustandsdiagramm beschreibt die Übergänge



Involvierte Warteschlangen

Interaktiver Benutzer

„Spooling“

Langfristiges Steuerungsprogramm (long term scheduler)

Kurzfristiges Steuerungsprogramm (short term scheduler)

Neue Jobs

Warteschlange für ankommende Tasks

„ready" Warteschlange

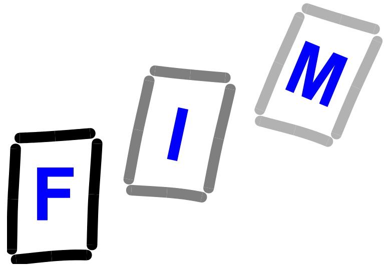
CPU

Ende

„waiting" Warteschlange

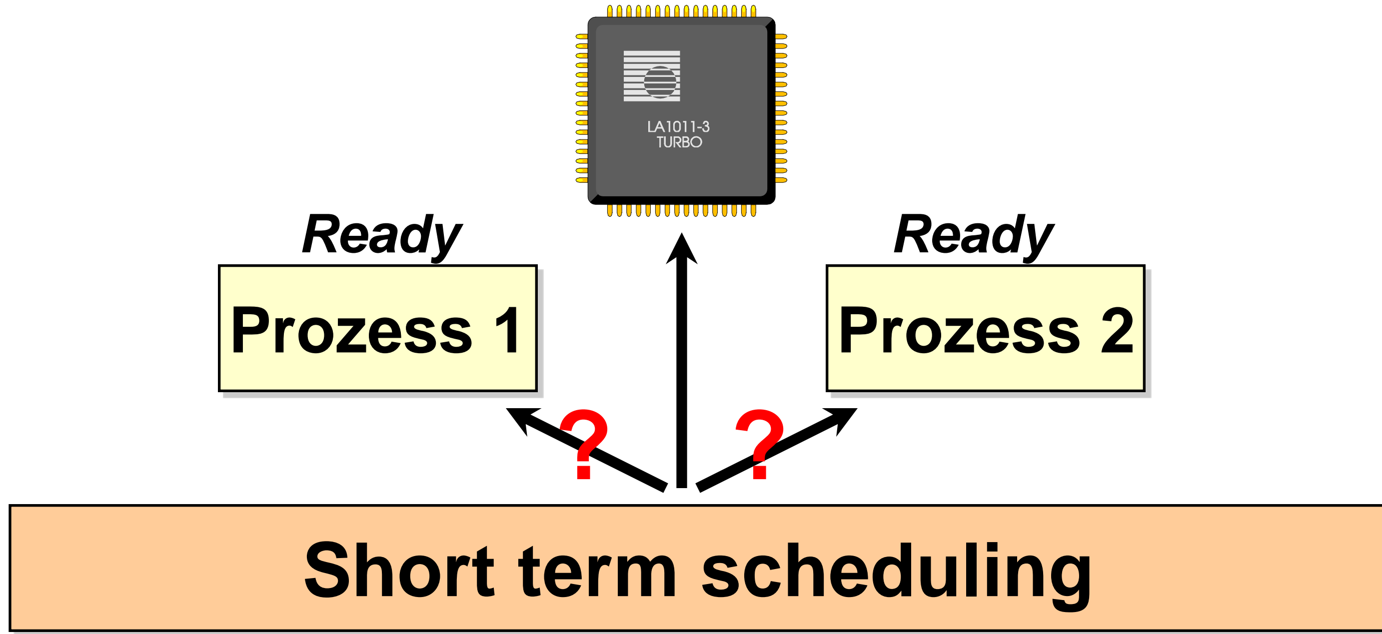
Jobs am Massenspeicher

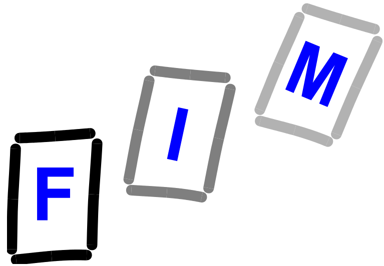
Prozesse im Hauptspeicher



Kurzfristige Ablaufsteuerung

Der Scheduler wählt einen der ausführbereiten (ready) Prozesse im Hauptspeicher aus und teilt ihm CPU zu





CPU-Scheduling Details (1 CPU)

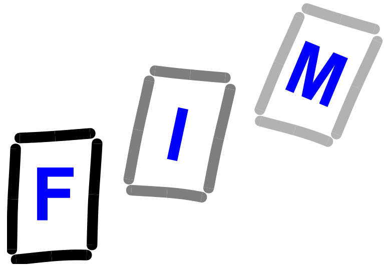
Entweder

- ➡ **sobald die CPU „untätig“ („idle“) ist,**
  - » **IDLE Prozess**

oder

- ➡ **aus übergeordneten Gründen (→ Scheduling-Srategie)**

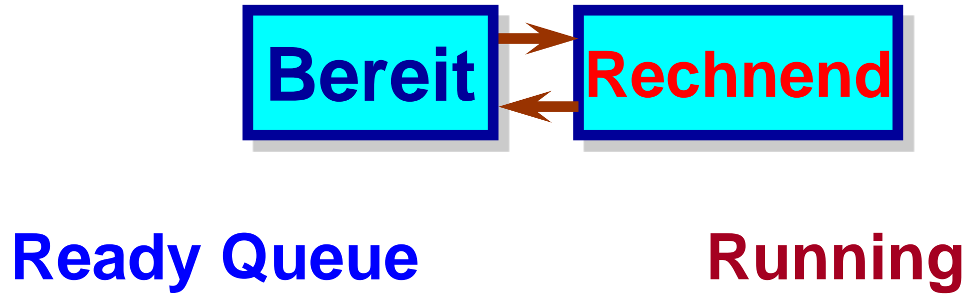
muss der Short-Term Scheduler einen der Prozesse aus der „ready queue“ auswählen und ausführen lassen

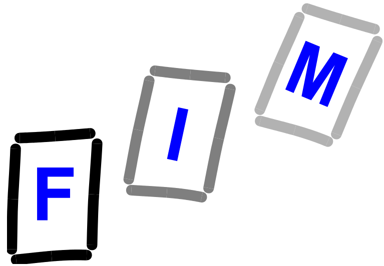


Prozessverdrängung $_1$?

Beachte:

Bei präemptiver Short-Term Strategie kann ein Prozess vom Zustand Running- wieder in die Ready Queue zurückgesetzt werden



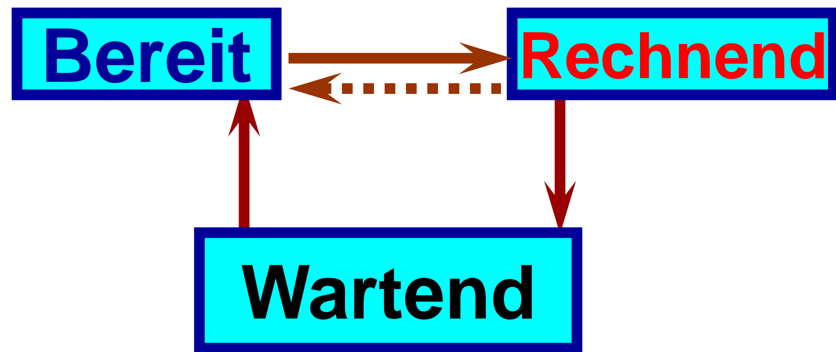


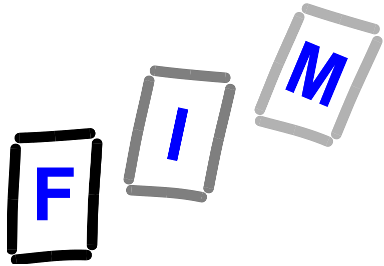
Prozessverdrängung $_2$?

Vereinfacht:

Bei **non preemtive Short-Term Strategie** bleibt ein Prozess **running**

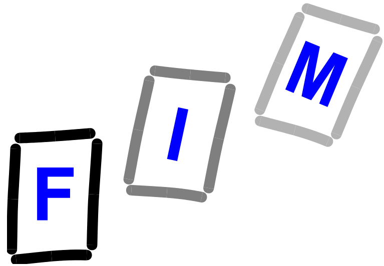
- Es sei denn, er wird „waiting“,
- oder er gibt die CPU freiwillig ab!
  - vgl.: cooperative multiprogramming





Ziele von Steuerungsalgorithmen

- **CPU-Auslastung** (CPU utilisation)
  Generelles Kriterium
- **Durchsatz** (throughput)
  Relevant bei Batch Systemen
- **Verweilzeit** (turnaround time)
  Relevant bei Batch Systemen
- **Antwortzeit** (response time)
  Relevant bei interaktiven Systemen
- **Wartezeit** (waiting time)



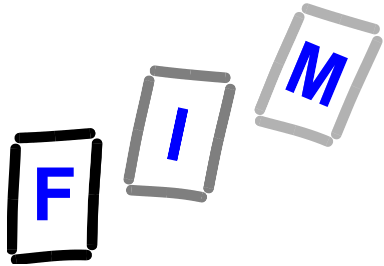
Durchsatz (throughput)

Der Durchsatz ist die Anzahl der fertig gerechneten Programme pro Zeiteinheit

- Trivial (unmittelbar einsichtig):

  Hängt stark vom durchschnittlichen Rechenbedarf der Prozesse ab

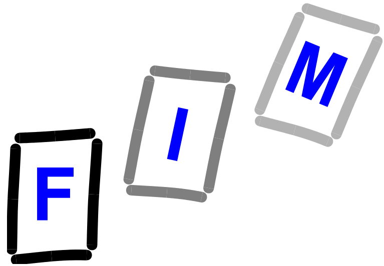
- Folgerung für unterschiedliche Scheduling Strategien
  - » CPU Auslastung besser, wenn Langläufer bevorzugt, aber Durchsatz sinkt
  - » CPU Auslastung sinkt, wenn Kurzläufer bevorzugt: Wegen Kontextswitch mit verbundener I/O, aber Durchsatz steigt



Verweilzeit
(turnaround time)

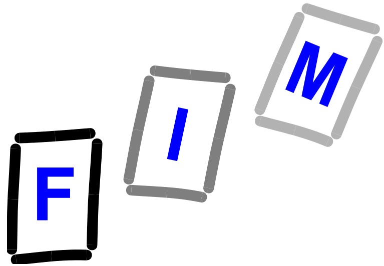
Verweilzeit :
Zeitspanne von der Übergabe eines Jobs an das System bis zu dessen Fertigstellung.

Beachte:
Verweilzeit enthält die Zeit für die Ausgabe der Ergebnisse, daher auch die Leistungsfähigkeit der Ausgabegeräte (Drucker, Harddisk !) relevant.



Antwortzeit
(response time)

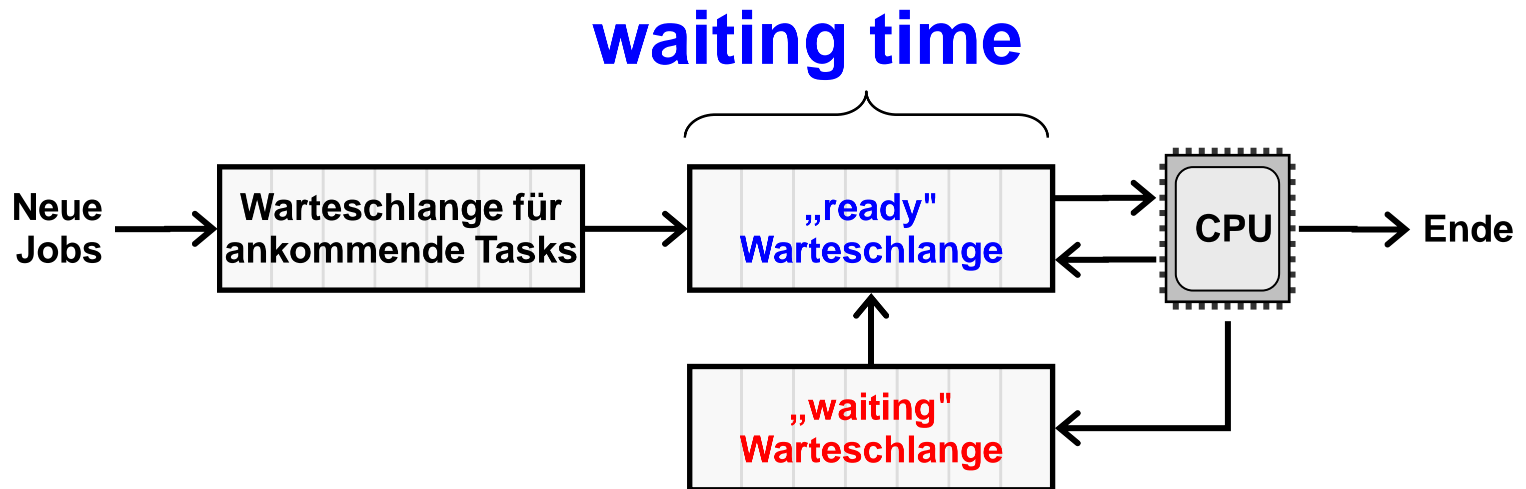
- Bezieht sich auf interaktive Systeme
- Zeitspanne vom Abschluss einer (Benutzer-) Eingabe bis zum Beginn der nachfolgenden Ausgabe
  - Mausklick/Tastendruck → Reaktion darauf beginnt
- Die Antwortzeit ist bei interaktiven Systemen ein besseres Kriterium als die Verweilzeit:
  - Subjektives Gefühl der Benutzer
    - » Wichtig: „Subjektiv“ → Schnell beginnen ist oft wichtiger als schnell fertig sein
- Besonderes Ziel bei Realtime



Wartezeit

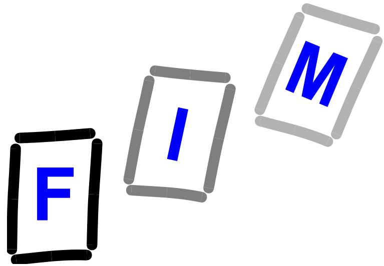
Die Wartezeit ist die Summe der Zeiten, die ein Prozess in der ready queue (!) verbringt

m.a.W.: = Summe Wartezeit auf CPU-Zuteilung



Warten ≠ Warten ???

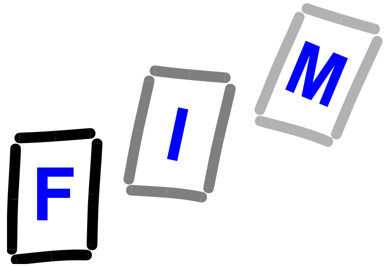
Ich hätt‘ ja wollen, aber man hat mich nicht lassen!



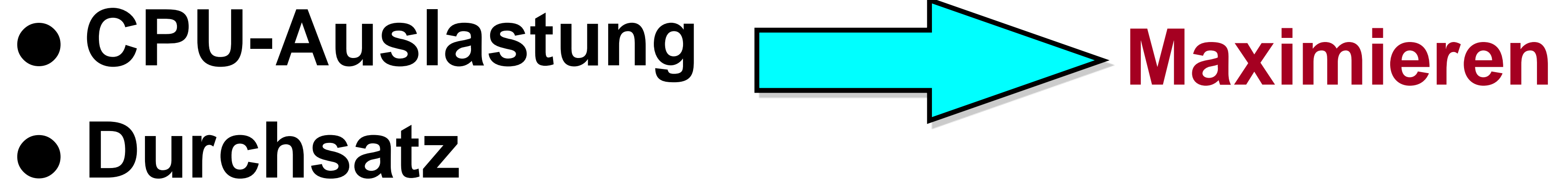
Warten in Ready Queue versus Warten in Waiting-Queue

Achtung:

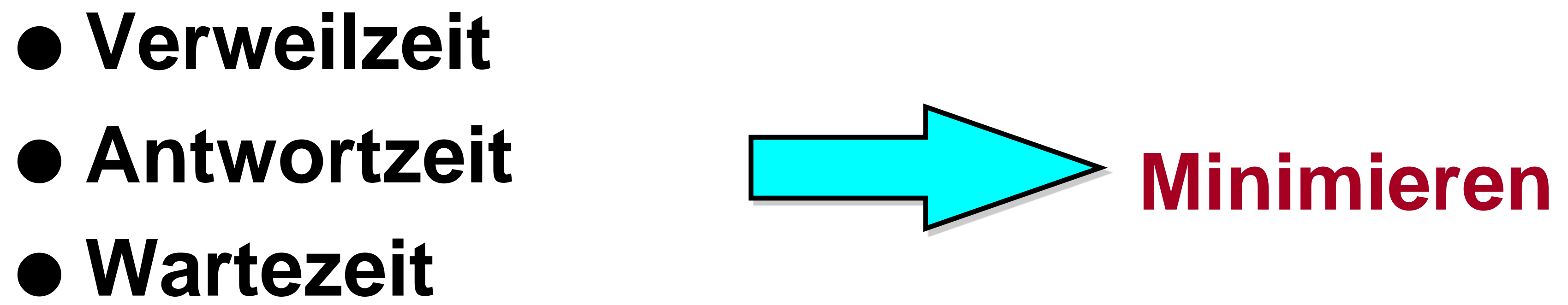
- Scheduler wählt Prozesse aus, die *bereit (ready)* sind
- Ein „not-ready“ Prozess wartet auf irgendein Ereignis in der waiting Queue und kann solange nicht ausgewählt werden, bis das Ereignis stattgefunden hat
- Daher:
  Zeit, die in der „waiting“-Schlange verbracht wird, wird <u>nicht</u> gezählt



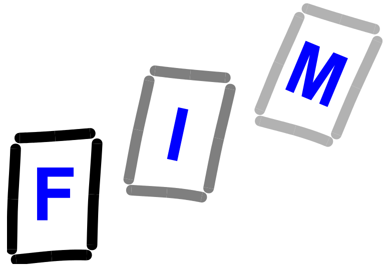
Anforderungen



- CPU-Auslastung
- Durchsatz



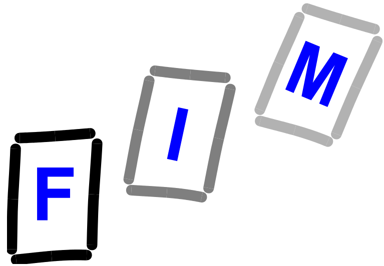
- Verweilzeit
- Antwortzeit
- Wartezeit



Nicht-unterbrechende Steuerungsalgorithmen

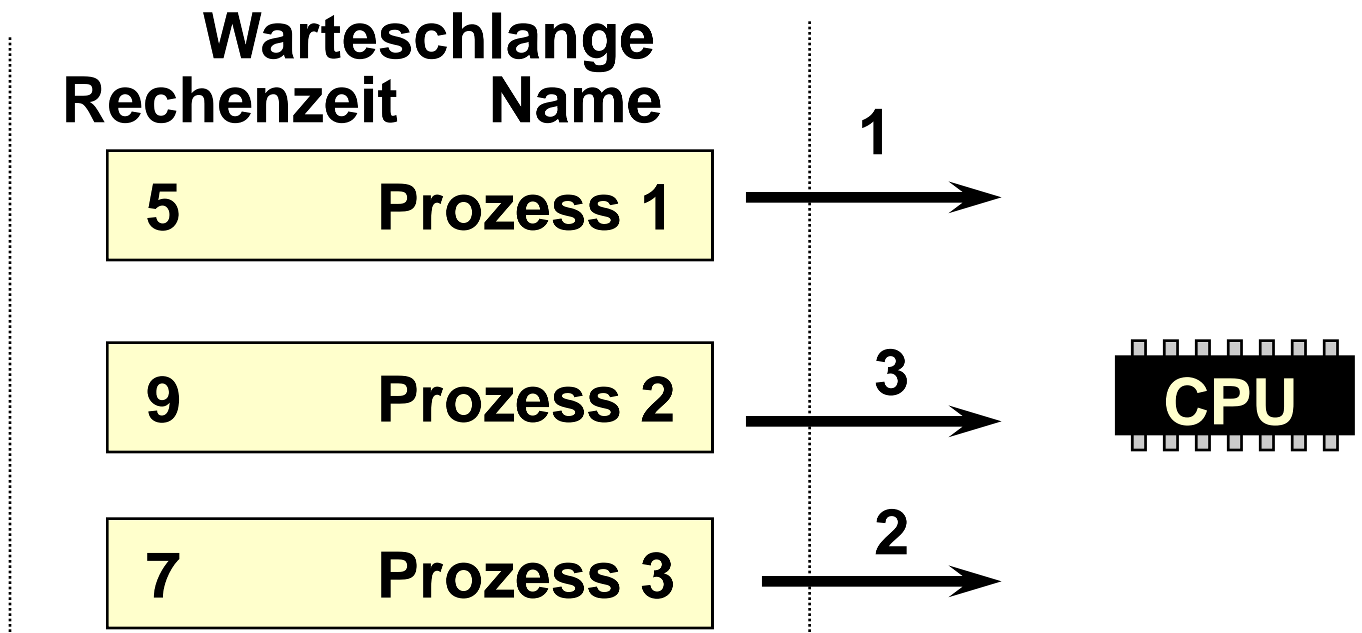
- Primär für Batch-orientierte Systeme
  - First Come First Served (FCFS)
    (Wer zuerst kommt, mahlt zuerst)
  - Shortest Job First (SJF)
  - Prioritätssteuerung
  - Prioritätssteuerung mit Altern (Aging)

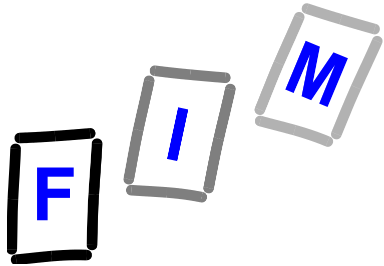
Bemerkung: Für den Algorithmus Prioritätssteuerung gibt es auch *unterbrechende Varianten.*



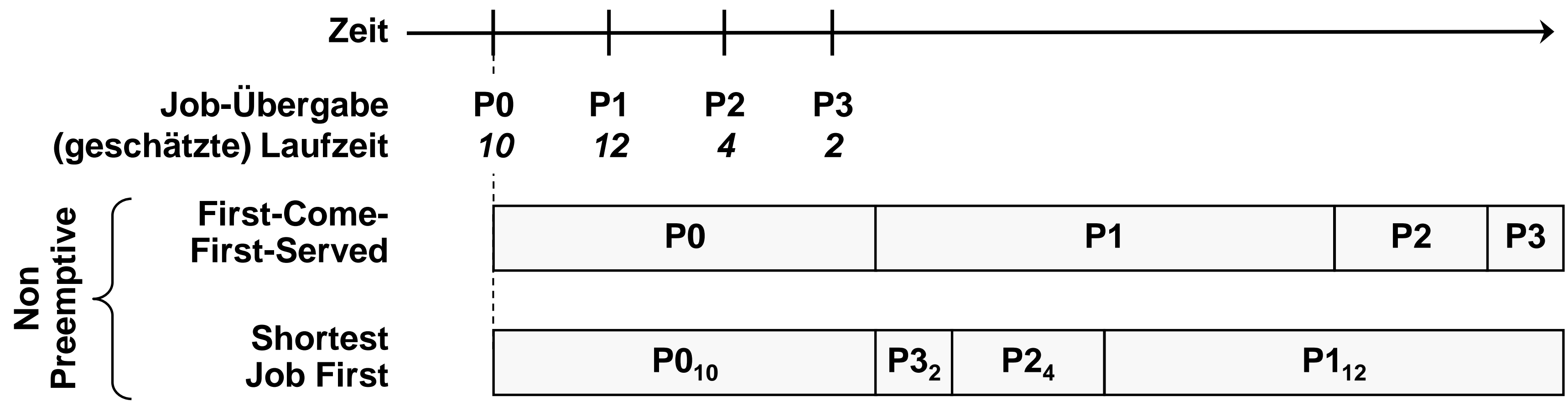
SJF (1)
(nonpreemptive)
Shortest Job First

SJF assoziiert mit jedem Prozess die (geschätzte) Dauer der Rechenzeit.
Ist die CPU verfügbar, wird sie dem Prozess mit der kürzesten Rechenzeit zugeordnet:

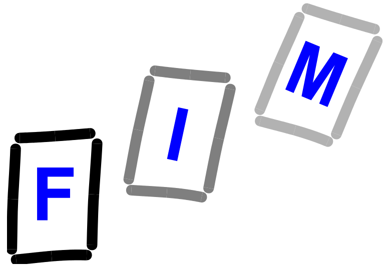




Vergleich
(Beginn mit P0)

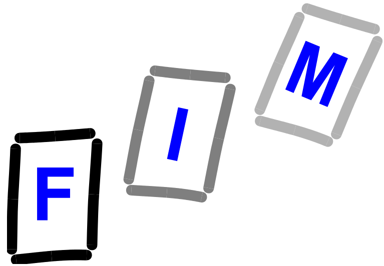


- **In Servicezentren werden regelmäßig Jobs ausgeführt, deren Verarbeitungszeit bekannt ist.**



SJF (2) Shortest Job First

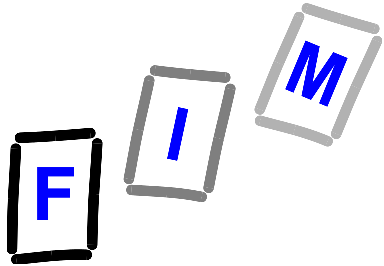
- **Ist die Rechenzeit von zwei Prozessen gleich, wird nach FCFS entschieden.**



SJF (3) Shortest Job First

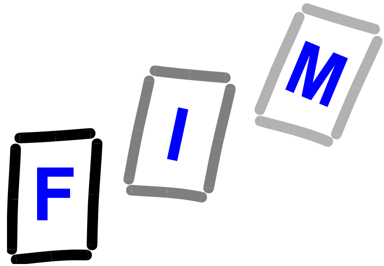
- SJF minimiert die Gesamt-Verweilzeit W einer gegebenen Menge von n Jobs „i“ mit der Verweilzeit w_i

  - ➡ $W = w_1+(w_1+ w_2)+..(w_1+ w_2+...+ w_n)$
    $= nw_1+(n-1)\ w_2+...+ w_n$
  - ➡ nw_1 trägt am meisten bei, also sollte w_1 minimal sein



SJF (4)
Shortest Job First

- **Länge der Rechenzeit ist allerdings nicht (genau) vorhersehbar. Man behilft sich daher mit einer Schätzung.**
  - ➡ **Beachte: Interaktivität, externe Inputs, …**
- **Werden Jobs periodisch ausgeführt, kann man ihre Laufzeit mitprotokollieren**
  - ➡ **T_n gemessene CPU-Zeit bei n-ter Jobausführung**
- **Auch auf „normalen“ Computern?**
  - ➡ **Für Server-Dienste wären dies eine Option!**



SJF (4)
Schätzung für Laufzeit eines Jobs nach n-ter Ausführung

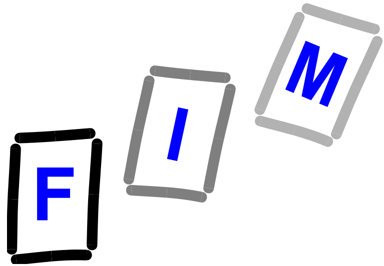
- **Sei**
  - S_1 anfangs geschätzter/bekannt gegebener Wert
  - S_n berechneter Schätzwert, n = 2,3....
  - T_n gemessene CPU-Zeit bei n-ter Jobausführung

  - 0<a<1 Gewichtungsfaktor

$$S_{n+1} = aT_n + (1-a)S_n$$

$$S_{n+1} = aT_n + (1-a)aT_{n-1} + \ldots$$

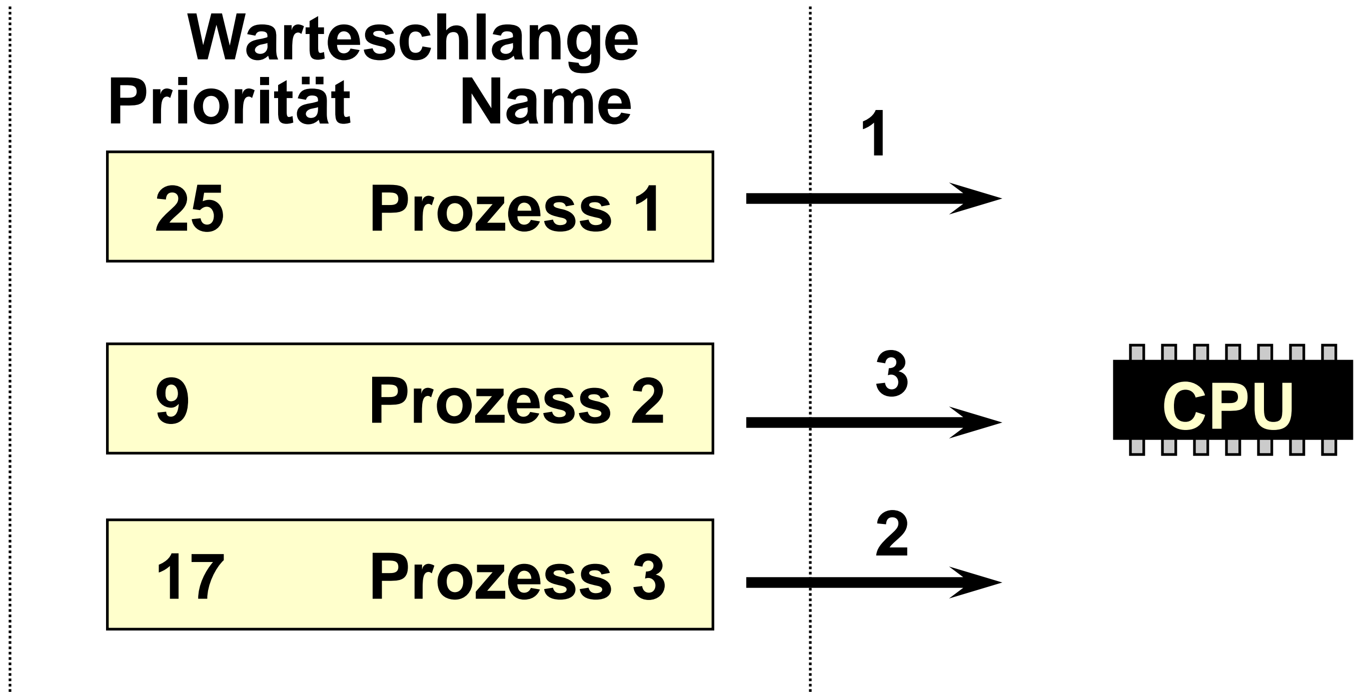
$$+ (1-a)^i aT_{n-i} + \ldots + (1-a)^{n-1} aT_1 + (1-a)^n S_1$$

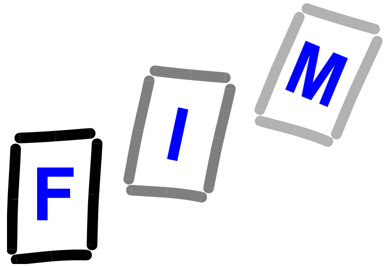


Prioritätssteuerung

Highest Priority First (HPF):

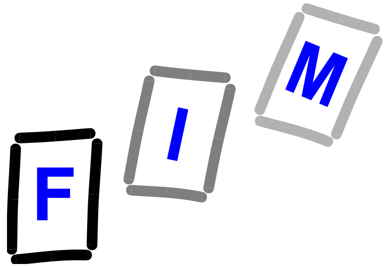
Jener Prozess erhält die CPU zugeteilt, der die höchste Priorität hat.





Prioritätssteuerung (2)

- Bei Prozessen mit gleicher Priorität wird wieder nach FCFS entschieden.
- Basis für die Vergabe von Prioritäten können sein
  - Kürzeste Rechenzeit
    - » SJF als preemptive Version: *Shortest Remaining Time Next*
  - Benötigte Ressourcen
    - » Dateien, Speicher
  - Arten von Prozessen
    - » Anwender (User mode)
    - » System (Kernel mode, Systemprozess)
  - Organisatorische Aspekte
    - » Endtermine, Fristen, ...



Verhungern
(Starvation)

Bei der Prioritätssteuerung kann es vorkommen, dass Prozesse „ewig“ auf die CPU warten, da Prozesse mit höherer Priorität laufend vorgereiht werden:

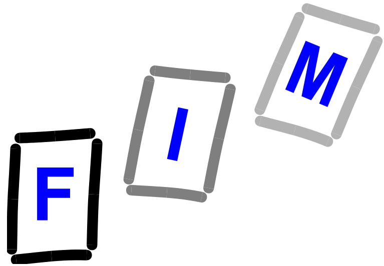
Warteschlange

Priorität	Name
25	Prozess 1
9	Prozess 2
17	Prozess 3

16 Prozess 4

CPU

1: Prozess 1 → CPU
2: Prozess 3 → CPU
3: Prozess 4 → CPU



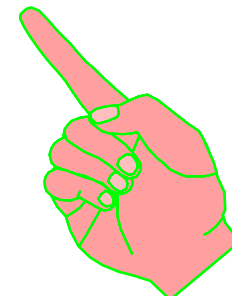
Altern
(Aging)

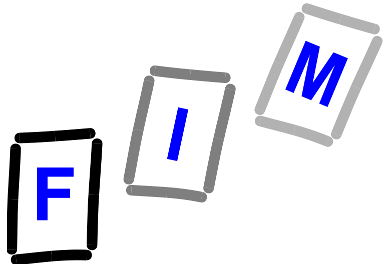
- **Lösung des vorher genannten Problems:**

  Altern (Aging)

- **Die Priorität eines Prozesses wird entsprechend seiner Wartezeit erhöht**

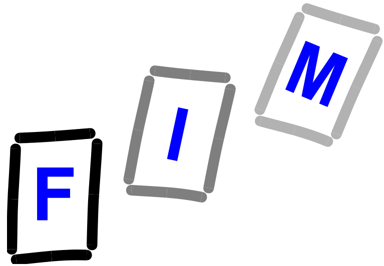
  ➡ **BS prüft periodisch die Prozesse auf ihre bisherige Wartezeit und erhöht ihre Priorität z.B. um „1“**





Preemptive Scheduling

- Klingt einsichtig und einfach
  - ➡ Unterbrechen, falls zweckmäßig
- Viele Probleme
  - ➡ Wechsel des PCB erforderlich
    - » Retten des Prozesszustandes (Register, ...)
    - » Allgemein: Kontextswitch kostet Zeit
  - ➡ Speicherverwaltung ?
  - ➡ Kann man immer zu beliebigen Zeitpunkten unterbrechen?
  - ➡ . . .

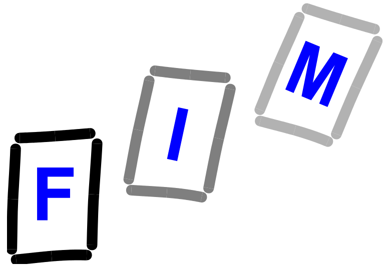


Unterbrechendes (preemptives) Scheduling

- **Preemptive Shortest Job First**
  bzw. Shortest Remaining Time Next
- **Prioritätssteuerung:**

Diese Verfahren arbeiten wie ihre nicht-unterbrechenden Varianten, aber:

Fordert ein Prozess mit kürzerer (verbleibender) Rechenzeit bzw. eine höherer Priorität die CPU an, so wird dem laufenden Prozess die CPU entzogen !



Unterbrechende Prioritätssteuerung

Prozess mit höchster Priorität unterbricht einen mit niedrigerer Priorität

Warteschlange

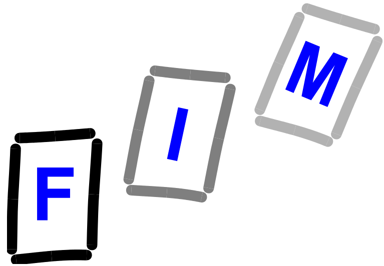
CPU

1

9	Prozess 1
Priorität	Name

2

16 Prozess 2

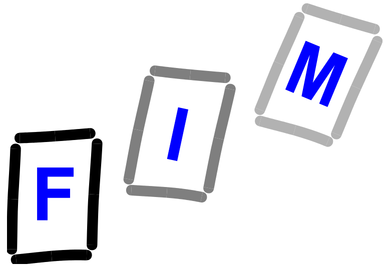


Dispatcher

Im Zusammenhang mit Prozessumschaltung benötigt man eine spezielle OS-Komponente: *Dispatcher*

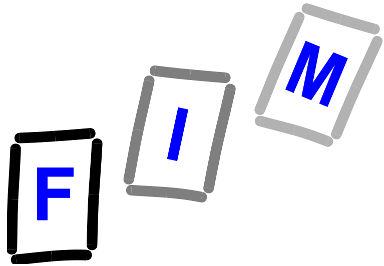
Der Dispatcher ist das Modul, das dem vom Scheduler ausgewählten Prozess die Kontrolle über die CPU gibt:

- **Wechseln des Kontextes**
  - » **Mit allem was dazugehört!**
- **Wechsel von *System-Modus* zu *Benutzer-Modus***
  - » ***Kernel → User mode***
- **(Neu)Start des in Frage kommenden Prozesses**



Zeitscheibenverfahren (Round Robin)

- **Eine kleine Zeiteinheit, *Zeitquantum*, (auch: Zeitscheibe) wird definiert.**
  - ➡ **Zeitquantum: typisch 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**
  - ➡ **Dann wird unterbrochen und der nächste Prozess kommt dran**
  - ➡ **Wenn ein Prozess auf ein Ereignis wartet (Running → Waiting), wird sofort weitergeschaltet**



Zeitquantum

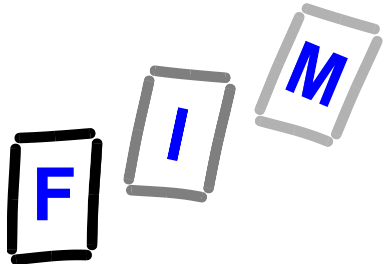
- **Dynamisches Verfahren**
  - » **Festlegung des Quantums wenn Prozess gestartet wird,**
  - » **dann dynamische Anpassung je nach Prozesstyp**

    Rechenintensive Prozesse → Kürzeres Quantum

    I/O- intensive Prozesse → Längeres Quantum
  - » **In Kombination mit Multilevel-Feedback Scheduling**

    Dynamische Veränderung möglich

    (In der Praxis sehr komplexe Strategien)

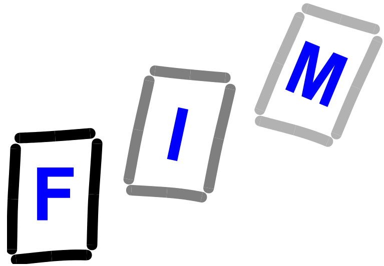


Round Robin-Verfahren RR

- Moderne OS verwenden RR auch ohne eine TS-Funktion anzustreben:
  - ➡ Unterbrechende Prioritätssteuerung
  - ➡ Innerhalb einer Klasse “gleichrangiger” Prozesse aber RR
  - ➡ Verwendung von variablen Zeit-Quanten

Beispiele:

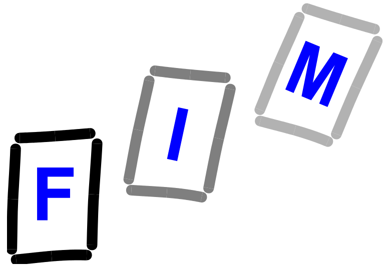
- ➡ Windows >=XP, Linux, MacOS...



Multilevel Feedback-Queue

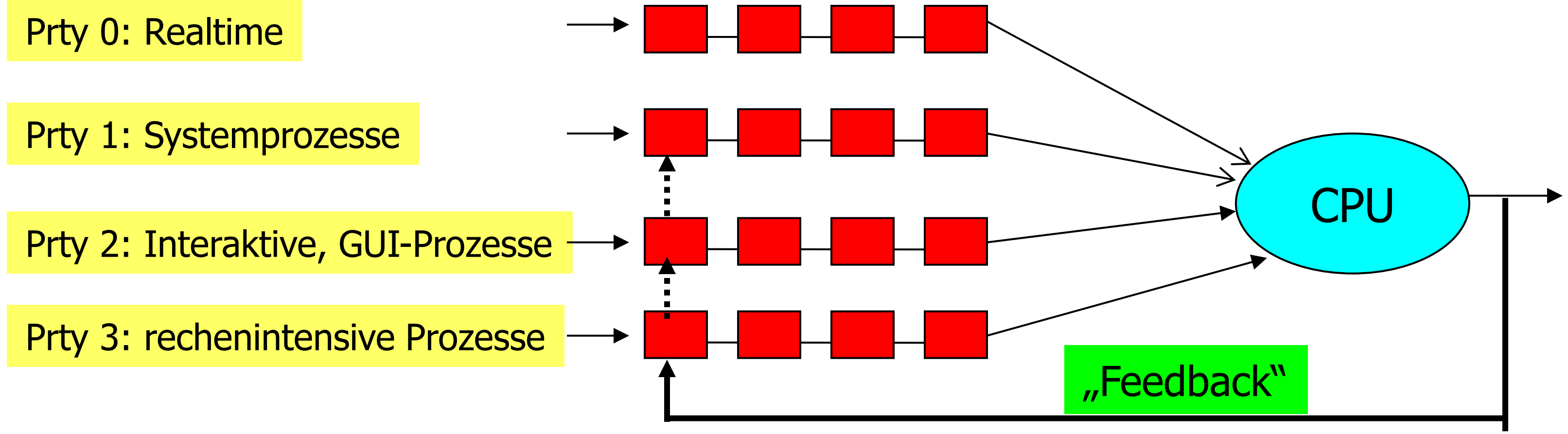
- **Basis:**
  - **Preemptive**
  - **Warteschlangen Q_1, Q_2, ... Q_n**
    - » **Zeitquanten T_i für Q_i mit $T_1 < T_2 ..... < T_n$**
  - **Q_1 hat höchste, Q_n niederste Priorität**
  - **P wird anfangs in Q_1 eingereiht**
    - » **Läuft T_1 ab, wird P nach Q_2 verschoben usw.**
    - » **Bevorzugt Kurzläufer**
    - » **Starvation-Problem**

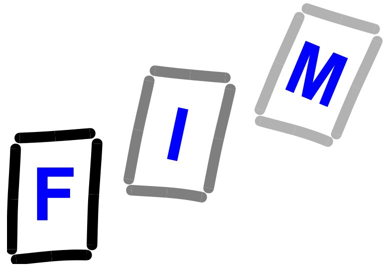
      Aging: Nach Ablauf von t (Systemparameter) wird ein P aus Q_i in eine darüber liegende Q_{i-1} (z.B.) verschoben



Multilevel Feedback-Queue

- Prozesstypen mit Queues assoziieren
  - Soft-Realtime-Prozesse
  - Systemprozesse
  - I/O intensive oder interaktive
    - (Kleine CPU Bursts, Warten auf I/O complete)
  - Rechenintensive





MultilevelFeedbackQueue Details

neuer Job

CPU

hohe Priorität

Multilevel Feedback Queue Scheduling

niedere Priorität

Aging

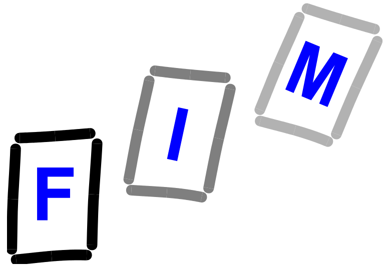
Q_1 → T_1

Q_2 → T_2

Q_3 → T_3

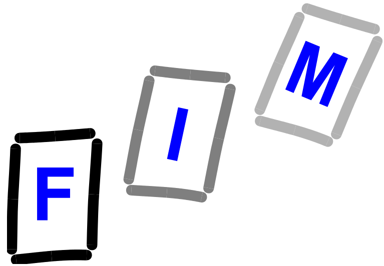
Q_n → T_n

Job beendet



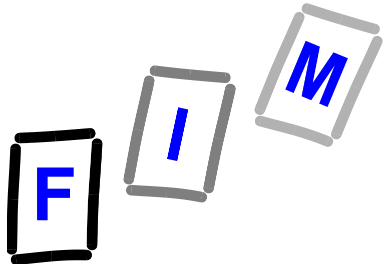
MultilevelFeedbackQueue Details$_2$

- Verschiedene Varianten
  - In welche Q_{min} kommt ein Prozess P?
    - » Zu Beginn immer nach Q_1?
  - Wie weit kann P zu Q_{max} absinken?
    - » Mit $1 \leq min \leq max \leq n$
    - » Soft RealTime: min=max=1
  - Aging: Wie hoch hinauf kommt P?
    - » Von Q_i nach Q_{i-1} oder höher?



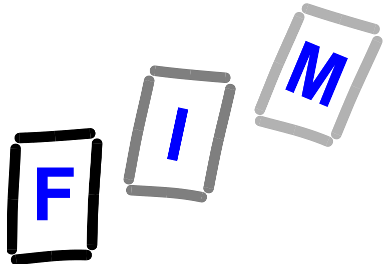
Fair Share Scheduling

- **Fairness:**
  - ➡ **Jedem Prozess wird garantiert**
    - » **Innerhalb endlicher Zeit die CPU zu erhalten**
    - » **Fertig gerechnet zu werden (kein Verhungern)**
  - ➡ **Wenn Prozess ein Bündel von Threads**
    - » **Würde er bevorzugt werden**
    - » **Scheduling erfolgt i.A. auf Thread-Ebene**
- **Fair Share Scheduling**
  - ➡ **Bilde Gruppen von Prozessen (Threads)**
  - ➡ **Weise den Gruppen eine Priorität zu**
    - » **CFS (Completely Fair Scheduler) von Linux >2.6.23**



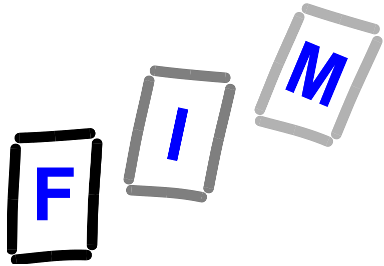
Realtime Scheduling

- Besondere Probleme:
  - Power Management (CPU, ganzer PC, …)
  - Hochauflösende Timer
  - Echtes Preemption immer möglich?
    - Critical sections, Interrupt handler, lange-dauernde Befehle (zB MMX/SSE) etc!
  - Auslagerungsdatei: Keine? Speicher-Sperren?



Realtime Scheduling Priority inversion

- **Grundlagen:**
  - Prozesse A (hohe Priority), B (mittlere P.) und C (niedrige P.)
  - Ressource R
- **Einfache Version:**
  - R wird von C gehalten → A hat höhere Priorität, muss aber dennoch warten bis C die Ressource freigibt
- **Komplexe Version:**
  - R wird von C gehalten → B unterbricht C (benötigt R nicht)
  - A muss solange warten, bis B (=niedrigere P.!) beendet ist, damit wir wieder bei der einfachen Version landen



Realtime Scheduling Priority inversion

- Lösungen:
  - Priority Aging: C erhöht seine Priorität, sodass er B die CPU wegnimmt und R freigeben kann
  - Alle Interrupts in Critical Sections abschalten
    - » Auf Multiprozessor-Systemen für alle CPUs!
    - » Critical Sections müssen kurz sein; wenn anwendbar sehr hilfreich
  - Priority Ceiling: Reservieren einer Ressource ist nur möglich, wenn seine Priorität über der System-Schranke liegt oder er die Schranken-bestimmende Ressource besitzt
    - » Schranke: Max(Priorität) der Prozesse, die auf eine Ressource zugreifen könnten
    - » System-Schranke: Max(Schranke aller tatsächlich benutzten Ressourcen)
  - Priority Inheritance
    - » C bekommt eine höhere Priorität (die von A), wenn R reserviert ist, das A haben möchte
  - Random Boosting: Zufälliger Ready-Task bekommt die CPU
  - Verzicht auf blockierende Synchronisation