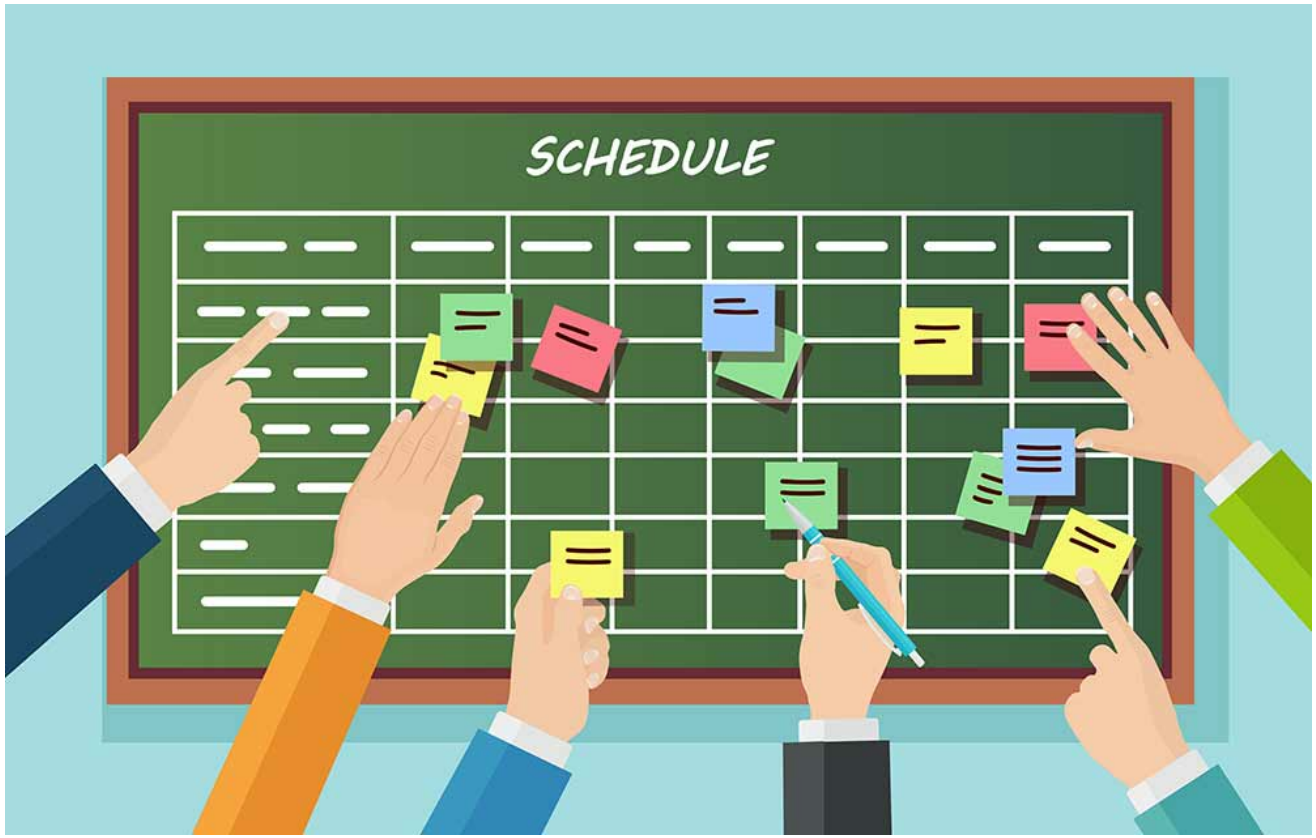
Betriebssysteme

Robert Kaiser

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2020/2021

4. Scheduling



https://advancesystems.ie/why-your-business-needs-to-switch-to-employee-scheduling-software/

Scheduling

Hochschule **RheinMain**

Einführung

Hochschule RheinMain

Grundlegende Begriffe:

- Wiederholung aus Kap. 3.1:
  - **Scheduler** oder **Dispatcher**: Umschalteinheit
  - **Scheduling-Algorithmus**: zugehöriger Algorithmus
  - **Prozesswechsel** oder **Kontextwechsel**: Umschaltvorgang, verursacht Kosten.
- **Non-preemptives Scheduling**-Verfahren[1] (*Run-to-Completion*, d.h. Prozess ist solange aktiv, bis er endet oder sich selbst blockiert).
  - Non-preemptive-Scheduling-Verfahren sind für General Purpose Systeme mit interaktiven Benutzern nicht geeignet.
- **Preemptives Scheduling**: rechnende Prozesse können suspendiert werden (Prozessorentzug)
  - *preemptive-resume*: Fortsetzung ohne Verlust
  - *preemptive-repeat*: Beginn von vorne

[1] auch: „kooperatives“ Scheduling-Verfahren

Grundlegende Begriffe (2)

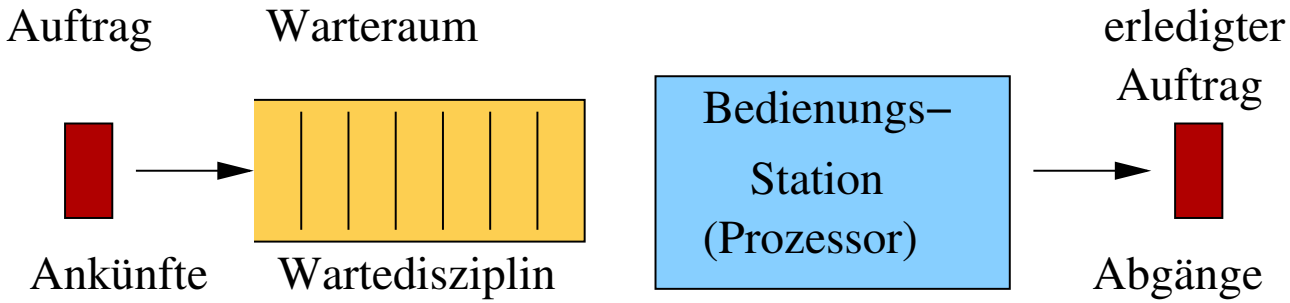
- **Prioritäten-basierte** Scheduling-Verfahren: ordnen Prozessen Prioritäten zu (relative Wichtigkeit).
- Prioritäten können **extern** vorgegeben sein, oder **intern** durch das Betriebssystem selbst bestimmt werden.
- Prioritäten können **statisch** sein, d.h. ändern sich während der Bearbeitung nicht, andernfalls: **dynamische** Prioritäten.
- Scheduling-Verfahren, die die Prozessorzuteilung auf der Basis gewählter Zeitspannen mittels Uhrunterbrechungen steuern, heißen **Zeitscheiben-basierte** Scheduling-Verfahren.
- Mischformen sind möglich

Grundlegende Begriffe (3)

- **Mehrstufiges Scheduling**[2]: Scheduler auf mehreren Ebenen
  Nur die auf einer höheren Ebene durch den entsprechenden Scheduler ausgewählten Prozesse sind dem Scheduler der nächst niederen Ebene für sein Scheduling sichtbar. Scheduling der niederen Ebene wird dann häufig als Dispatching bezeichnet.
- Beispiele:
  - Virtuelle Maschine (z.B. VirtualBox): Die virtuelle Maschine unterliegt als Anwendungsprozess dem Scheduler des Wirtssystems. In der VM arbeitet ein Gastsystem, dessen Prozesse dessen Scheduler unterliegen.
  - 2-stufiges Scheduling in UNIX; Der Scheduler der höheren Ebene transportiert Aufträge vom Hintergrundspeicher in den Arbeitsspeicher und zurück (*Swapping*); der Scheduler der niederen Ebene berücksichtigt ausschließlich Prozesse, die sich im Arbeitsspeicher befinden.

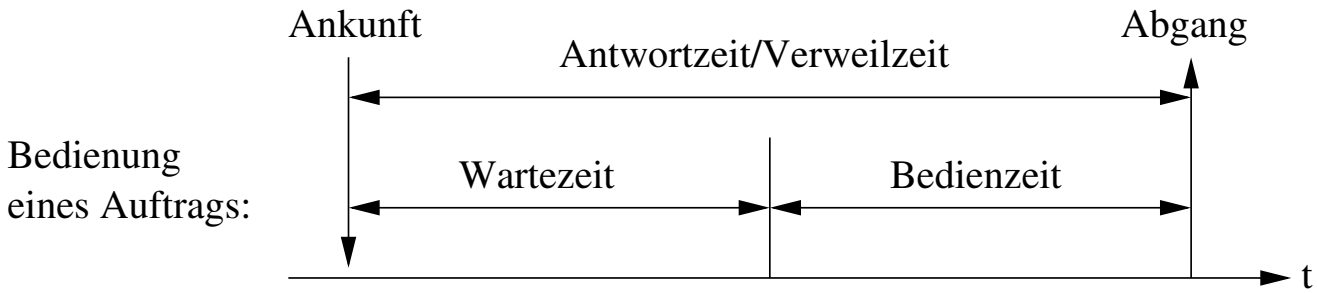
[2] auch: Hierarchisches Scheduling

Begriffe aus der Bedientheorie (1)



- **Auftrag:** Einheit zur Bearbeitung (z.B. Stapeljob, Dialogschritt)
- **Bedienzeit**:Zeitdauer für die reine Bearbeitung eines Auftrags durch die Bedienstation (hier: den Prozessor)
- **Wartedisziplin**: z.B. (einfache):
  - FCFS (*First-Come-First-Served*) oder FIFO (*First-In-First-Out*)
  - LIFO (*Last-In-First-Out*)
  - Random (zufällige Auswahl)
- Ankünfte und Bedienzeiten werden bei Bedienungsmodellen häufig durch stochastische Prozesse modelliert
- komplexere Bedienmodelle können mehrere Bedienstationen (Multiprozessorsystem), mehrere Warteräume, mehrphasige Bedienung sowie die Rückführung teilweise bearbeiteter Aufträge enthalten.

Begriffe aus der Bedientheorie (2)



- **Antwortzeit:** Zeitdauer vom Eintreffen eines Auftrags bis zur Fertigstellung
  Bei Dialogaufträgen Zeitdauer von der Eingabe eines Benutzers (z.B. Drücken der Return-Taste) bis zur Erzeugung einer zugehörigen Ausgabe (z.B. auf dem Bildschirm)
  bei Stapelaufträgen auch **Verweilzeit** genannt
- **Wartezeit**: Antwortzeit - Bedienzeit
- **Durchsatz**: Anzahl erledigter Aufträge pro Zeiteinheit
- **Auslastung**: Anteil der Zeit im Zustand „belegt”
- **Fairness**: „Gerechte” Behandlung aller Aufträge, z.B. alle rechenwilligen Prozesse haben gleichen Anteil an der zur Verfügung stehenden Rechenzeit

„Guter“ Scheduling-Algorithmus

Ziele:

- Nutzungsbezogen:
  - Kurze Antwortzeiten bei interaktiven Aufträgen.
  - Kurze Verweilzeiten für Stapelverarbeitungsaufträge.
- Betriebsbezogen:
  - Hoher Durchsatz.
  - Hohe Auslastung.
  - Fairness in der Behandlung aller Aufträge.
  - Geringer Aufwand für die Bearbeitung des Scheduling-Algorithmus selbst (Overhead!).

Probleme

Vorabwissen über die Bedienzeit:

- Einige Verfahren verlangen Kenntnis der Bedienzeit eines Auftrags bei dessen Ankunft (vgl. Verfahren zur Maschinenbelegung, Operations Research).
- Nur realistisch für wiederkehrende Stapeljobs, nicht realistisch bei interaktiven Aufträgen.
- Bei Echtzeit-Scheduling: Annahme / Abschätzung / Formale Bestimmung einer **Oberschranke der Bedienzeit** (*worst case execution time* - WCET) als Voraussetzung für die Echtzeitplanung

Moderne Anforderungen (1)

Trennung von Strategie und Mechanismus: (Ziel: höhere Flexibilität)

- Parametrisierbarer Scheduling-Mechanismus auf niederer Ebene (im Betriebssystemkern).
- Parameter können auf höherer Ebene (z.B. über Systemaufrufe in Benutzerprozessen) gesetzt werden, um applikationsbezogene Strategie zu implementieren.

→ Das Scheduling wird damit weiter auf niederer Ebene durchgeführt, aber von der Applikationsebene aus gesteuert.

- Beispiel: Ein Datenbank-Management-Prozess kann für seine Kindprozesse, die z.B. bestimmte ihm bekannte Anfragen bearbeiten oder interne Dienste durchführen, deren optimale Einplanung und Abfolge bewirken.

Moderne Anforderungen (2)

User-Level Scheduler

- Über die o.g. Forderung hinaus kann ein Applikationsprogramm dem Betriebssystem z.B. über einen Systemaufruf einen Scheduler übergeben, der für eine Teilmenge der Prozesse deren Scheduling durchführt (Höchstmaß an Flexibilität).
- Nur in wenigen Betriebssystemen vorhanden
- Mikrokern-Ansatz: Rechenzeit als Ressource, per IPC transferierbar
- Gegenstand aktueller Forschung[3]

[3] z.B.:

- Lyons et al: *Scheduling-Context Capabilities*
- Gadepalli et al: *Slite: OS Support for Near Zero-Cost, Configurable Scheduling*

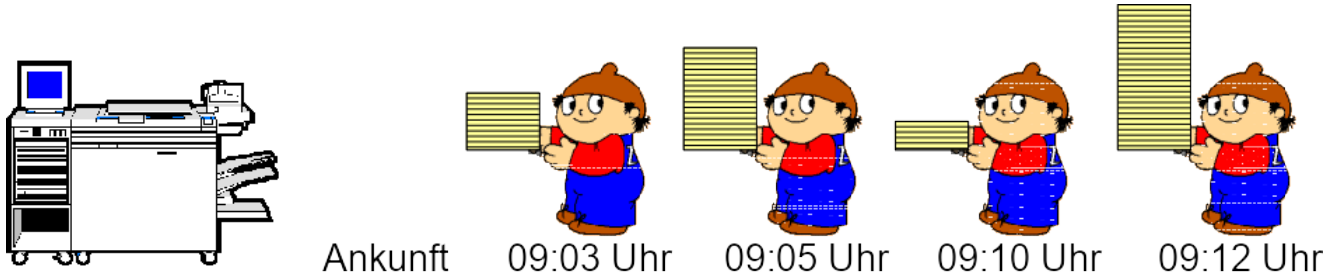
Non-Preemptive Scheduling

(Annahme: Bekannte Bedienzeiten)

1. First-Come-First-Served (FCFS)
2. Shortest-Job-First (SJF)
3. Prioritäts-Scheduling (Prio)

First-Come-First-Served (FCFS)

Hochschule RheinMain



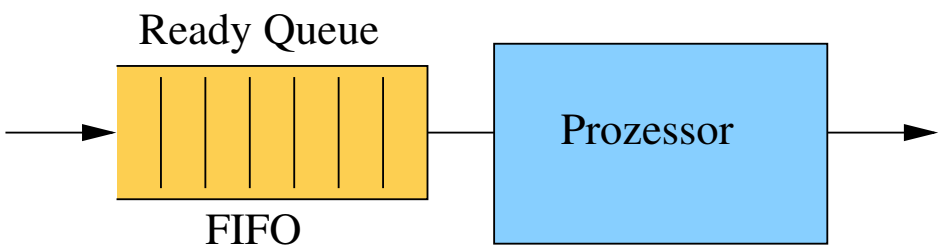
Algorithmus:

- Einfachst-Algorithmus: „Wer zuerst kommt, mahlt zuerst“.
- Vollständige Bearbeitung jedes Auftrags, bevor ein neuer begonnen wird.
- „Pech“, wenn ein Langläufer vor kurzem Prozess in der Schlange steht

Implementierung:

- Die Ready-Queue wird als FIFO- Liste verwaltet.

Bedienmodell:



Rechenbeispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

- Resultierender Schedule:

Prozess	Wartezeit	Antwortzeit
1	0	13
2	13	16
3	13+3=16	22

Durchschnittliche Wartezeit:
$(13 + 16)/3 = 29/3$

Im Falle der Ausführungsfolge 3, 2, 1 hätte sich ergeben:
Durchschnittliche Wartezeit: $(6 + 9)/3 = 5$

Shortest-Job-First (SJF)

Hochschule RheinMain

Algorithmus:

- Von allen rechenwilligen Prozessen wird derjenige mit der kleinsten Bedienzeitanforderung ausgewählt.
- Bei gleicher Bedienzeitanforderung wird nach FCFS gewählt.
- Der Algorithmus SJF ist in dem Sinne optimal in der Menge aller möglichen Algorithmen, dass er die kürzeste mittlere Wartezeit für alle Aufträge sichert.
- Notwendigkeit der Kenntnis der Bedienzeit!

Rechenbeispiel

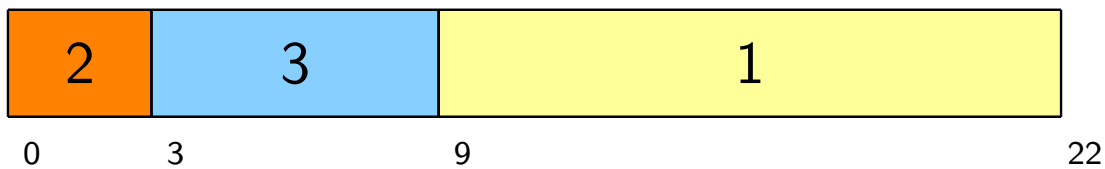
Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

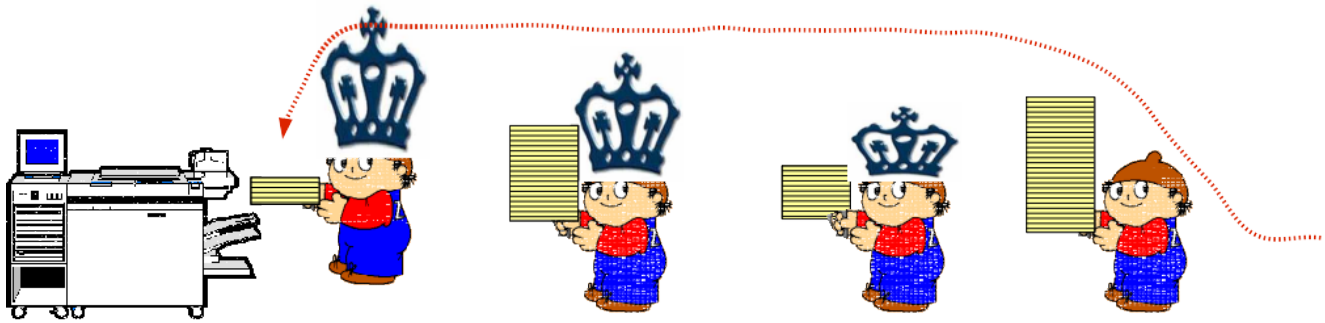
- Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	3+6=9	22
2	0	3
3	3	9

Durchschnittliche Wartezeit:
$(9 + 3)/3 = 4$

Prioritäts-Scheduling (Prio)



Algorithmus:

- Jeder Auftrag besitze eine statische Priorität.
- Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt.
- Bei gleicher Priorität wird nach FCFS ausgewählt.

Rechenbeispiel

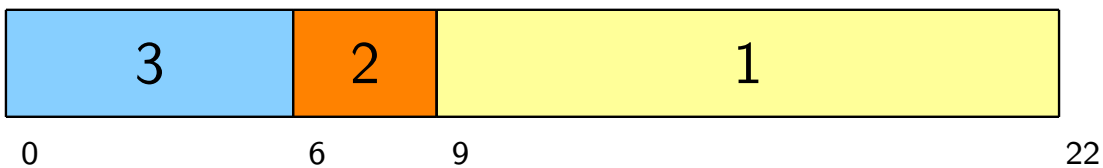
Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit	Priorität
1	13	2
2	3	3
3	6	4

Alle Aufträge seien zur Zeit Null bekannt

- Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	6+3=9	22
2	6	9
3	0	6

Durchschnittliche Wartezeit[4]:
$(9 + 6)/3 = 5$

[4] In diesem Beispiel – abhängig von Prioritätsvergabe sind auch alle anderen Ergebnisse möglich

Preemptive Scheduling

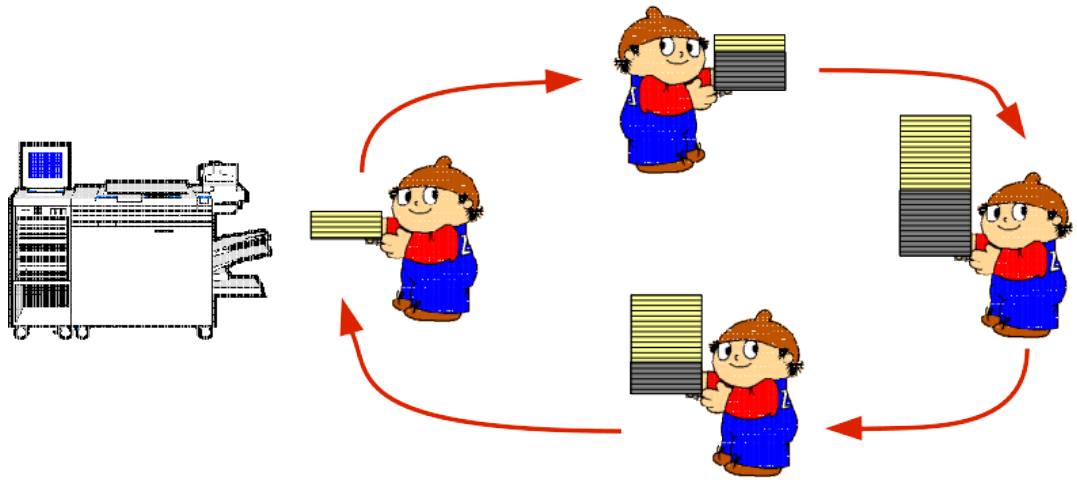
Hochschule RheinMain

(realistisch für heutige Rechensysteme)

1. Round-Robin-Scheduling (RR)
2. Unterbrechendes Prioritäts-Scheduling
3. Mehrschlangen-Scheduling
4. Mehrschlangen-Feedback-Scheduling

Round-Robin-Scheduling (RR)

Hochschule RheinMain



Algorithmus:

- Menge der rechenwilligen Prozesse linear geordnet.
- Jeder rechenwillige Prozess erhält den Prozessor für eine feste Zeitdauer q, die **Zeitscheibe** (*time slice*) oder **Quantum** genannt wird.
- Nach Ablauf des Quantums wird der Prozessor entzogen und dem nächsten zugeordnet (preemptive-resume).
- Tritt vor Ende des Quantums Blockierung oder Prozessende ein, erfolgt der Prozesswechsel sofort.
- Dynamisch eintreffende Aufträge werden z.B. am Ende der Warteschlange eingefügt.

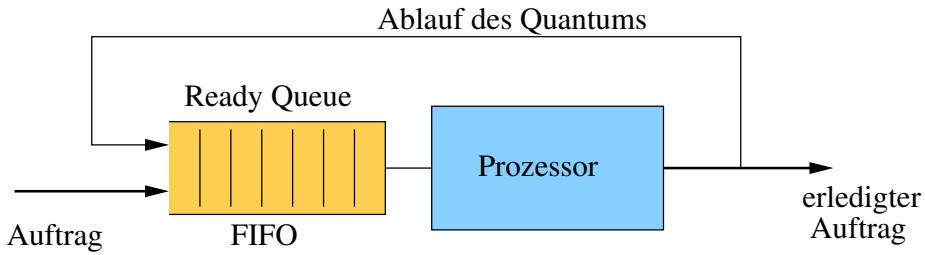
Implementierung:

- Die Zeitscheibe wird durch einen Uhr-Interrupt realisiert.
- Die Ready Queue wird als lineare Liste verwaltet, bei Ende eines Quantums wird der Prozess am Ende der Ready Queue eingefügt.

Round-Robin-Scheduling (RR)

Hochschule RheinMain

Bedienmodell:



Bewertung:

- Round-Robin ist einfach und weit verbreitet.
- Alle Prozesse werden als gleich wichtig angenommen und fair bedient.
- Langläufer benötigen ggf. mehrere „Runden“
- Keine Benachteiligung von Kurzläufern
  (ohne Bedienzeit vorab zu kennen)
- Einziger kritischer Punkt: Wahl der Dauer des Quantums.
  - Quantum zu klein → häufige Prozesswechsel, sinnvolle Prozessornutzung sinkt
  - Quantum zu groß → schlechte Antwortzeiten bei kurzen interaktiven Aufträgen.

Rechenbeispiel

Hochschule RheinMain

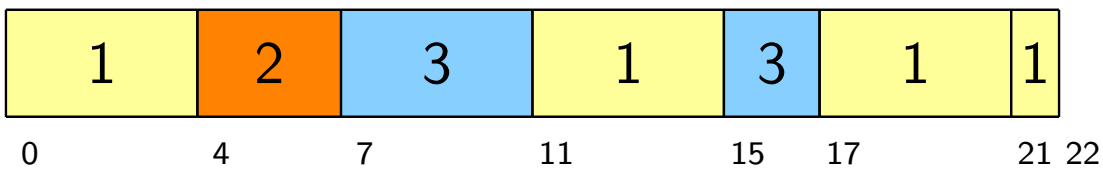
Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

Quantum sei $q = 4$

- Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	3+4+2=9	22
2	4	7
3	4+3+4=11	17

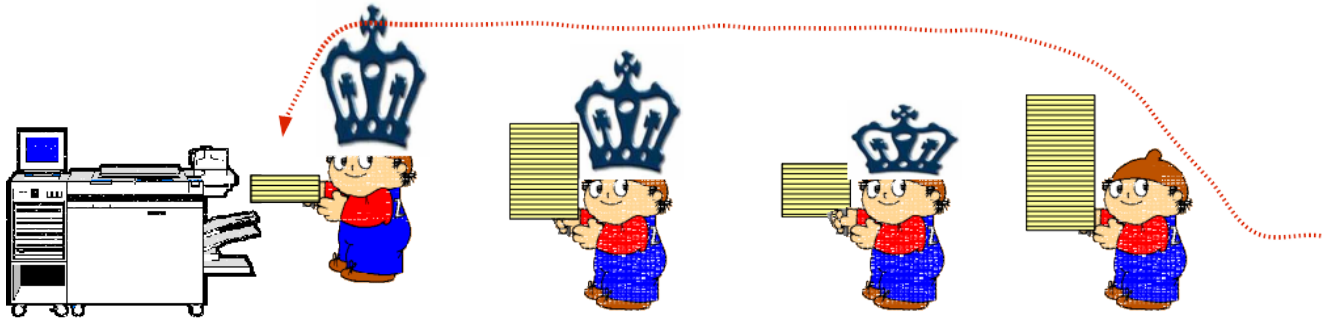
Durchschnittliche Wartezeit:
$(9 + 4 + 11)/3 = 8$

Grenzwertbetrachtung

Grenzwertbetrachtung für Quantum q:

- $q \to \infty$:
  Round-Robin verhält sich wie FCFS.
- $q \to 0$:
  Round-Robin führt zu sogenanntem **processor sharing**:
  jeder der n rechenwilligen Prozesse erfährt $\frac{1}{n}$ der Prozessorleistung.
  (Kontextwechselzeiten als Null angenommen).

Unterbrechendes Prioritäts-Scheduling



Algorithmus:

- Jeder Auftrag besitze eine statische Priorität.
- Prozesse werden gemäß ihrer Priorität in eine Warteschlange eingereiht.
- Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt und bedient.
- Wird ein Prozess höherer Priorität rechenwillig (z.B. nach Beendigung einer Blockierung), so wird der laufende Prozess unterbrochen (*preemption*) und in die Ready Queue eingefügt.

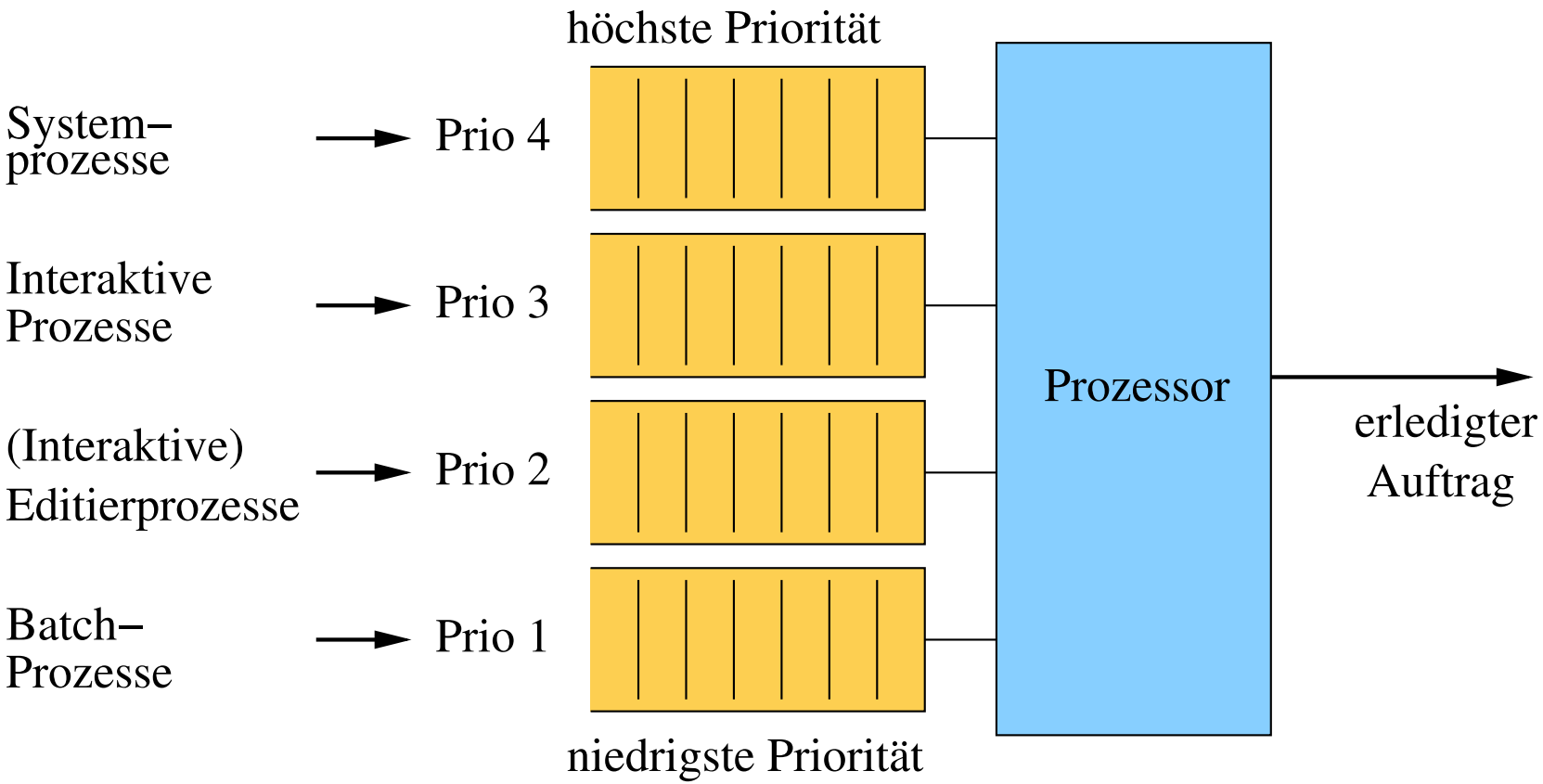
Mehrschlangen-Scheduling

Algorithmus:

- Prozesse werden statisch klassifiziert als einer bestimmten Gruppe zugehörig (z.B. interaktiv, batch).
- Alle rechenwilligen Prozesse einer bestimmten Klasse werden in einer eigenen Ready Queue verwaltet.
- Jede Ready Queue kann ihr eigenes Scheduling-Verfahren haben (z.B. Round-Robin für interaktive Prozesse, FCFS für batch-Prozesse).
- Zwischen den Ready Queues wird i.d.R. unterbrechendes Prioritäts-Scheduling angewendet, d.h.: jede Ready Queue besitzt eine feste Priorität im Verhältnis zu den anderen; wird ein Prozess höherer Priorität rechenwillig, wird der laufende Prozess unterbrochen (preemption).

Mehrschlangen-Scheduling (2)

Bedienmodell (Beispiel):



Mehrschlangen-Feedback-Scheduling

Prinzip:

- Erweiterung des Mehrschlangen-Scheduling.
- Rechenwillige Prozesse können im Verlauf in verschiedene Warteschlangen eingeordnet werden (dynamische Prioritäten).
- Algorithmen zur **Neubestimmung der Priorität** wesentlich

Bsp. 1: Wenn ein Prozess blockiert, wird die Priorität nach Ende der Blockierung um so größer, je weniger er von seinem Quantum verbraucht hat (Bevorzugung von I/O-intensiven Prozessen).

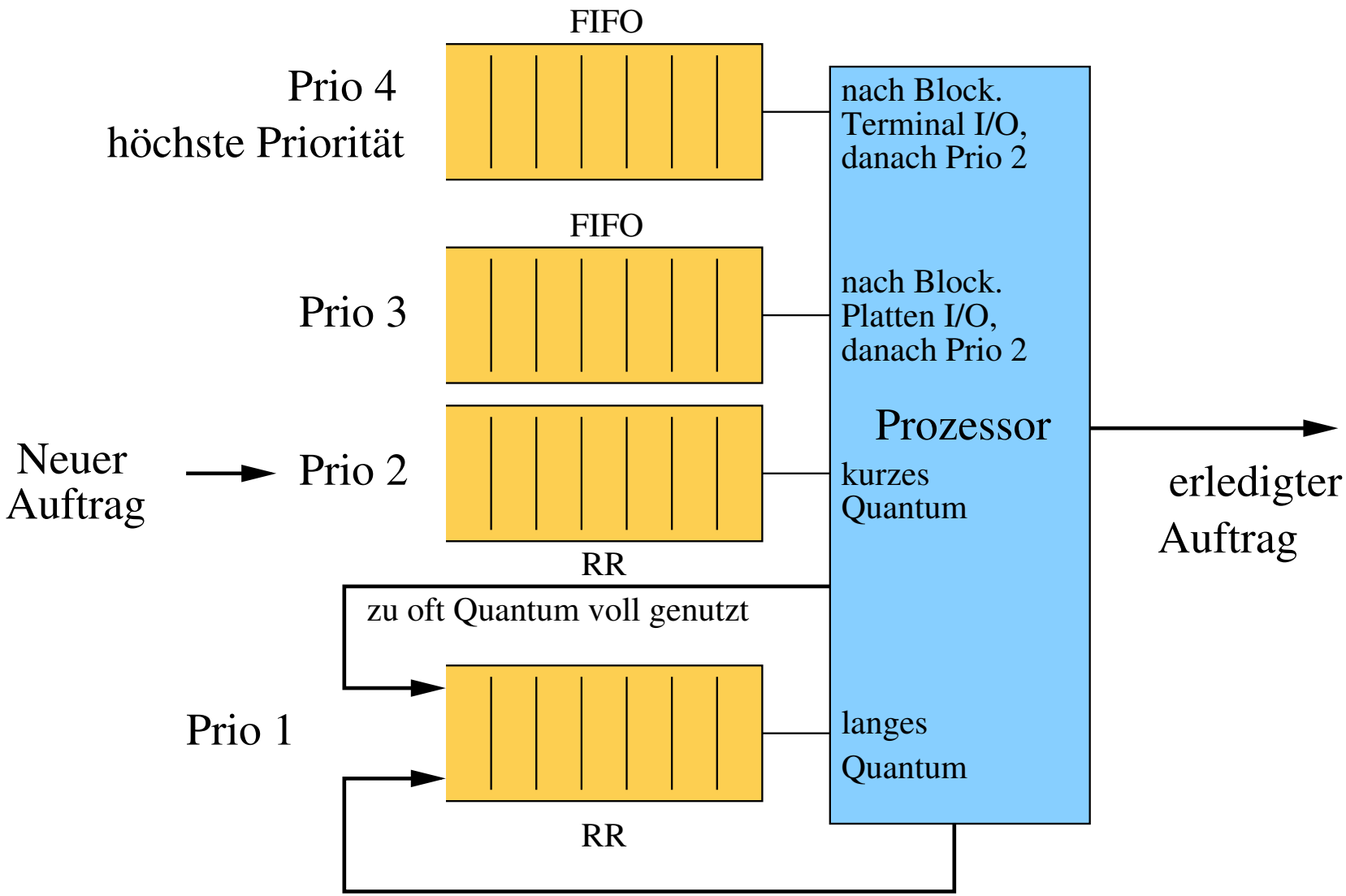
Bsp. 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von Langläufern).

Bsp. 3: Wenn ein Prozess lange nicht bedient worden ist, wird seine Priorität verbessert (Altern, Vermeidung einer „ewigen“ Bestrafung).

Mehrschlangen-Feedback-Scheduling (2)

Hochschule RheinMain

Bedienmodell:



Bewertung

- Mit wachsender Bedienzeit sinkt die Priorität, d.h. Kurzläufer werden bevorzugt, Langläufer werden zurückgesetzt.
- Wachsende Länge des Quantums mit fallender Priorität verringert die Anzahl der notwendigen Prozesswechsel (Einsparen von Overhead).
- Verbesserung der Priorität nach Beendigung einer Blockierung berücksichtigt I/O-Verhalten (Bevorzugung von I/O-intensiven Prozessen). Durch Unterscheidung von Terminal I/O und sonstigem I/O können interaktive Prozesse weiter bevorzugt werden.
- sehr flexibel.
- Die Scheduler in Windows und Linux arbeiten nach diesem Prinzip

Scheduling in Linux (1)

- Linux 1.2
  - Zyklische Liste, Round-Robin
- Linux 2.2
  - Scheduling-Klassen (Echtzeit, Non-Preemptive, Nicht-Echtzeit)
  - Unterstützung für Multiprozessoren
- Linux 2.4
  - O(n)-Komplexität (jeder Task-Kontrollblock muss angefasst werden)
  - Round-Robin
  - Teilweiser Ausgleich bei nicht verbrauchter Zeitscheibe
  - Insgesamt relativ schwacher Algorithmus

Scheduling in Linux (2)

- Linux 2.6
  - O(1)-Komplexität (konstanter Aufwand für Auswahl unabhängig von Anzahl Tasks)
  - Run Queue je Priorität
  - Zahlreiche Heuristiken für Entscheidung I/O-intensiv oder rechenintensiv
  - Sehr viel Code
- ab Linux Kernel 2.6.23: „Completely Fair Scheduler“ (CFS)
  - Sehr gute Approximation von Processor Sharing
  - Task mit geringster *Virtual Runtime* (größter Rückstand) bekommt Prozessor
  - Zeit-geordnete spezielle Baumstruktur für Taskverwaltung ($\rightarrow$ O(log n)-Komplexität)
  - Kein periodischer Timer-Interrupt sondern One-Shot-Timer („tickless Kernel“)

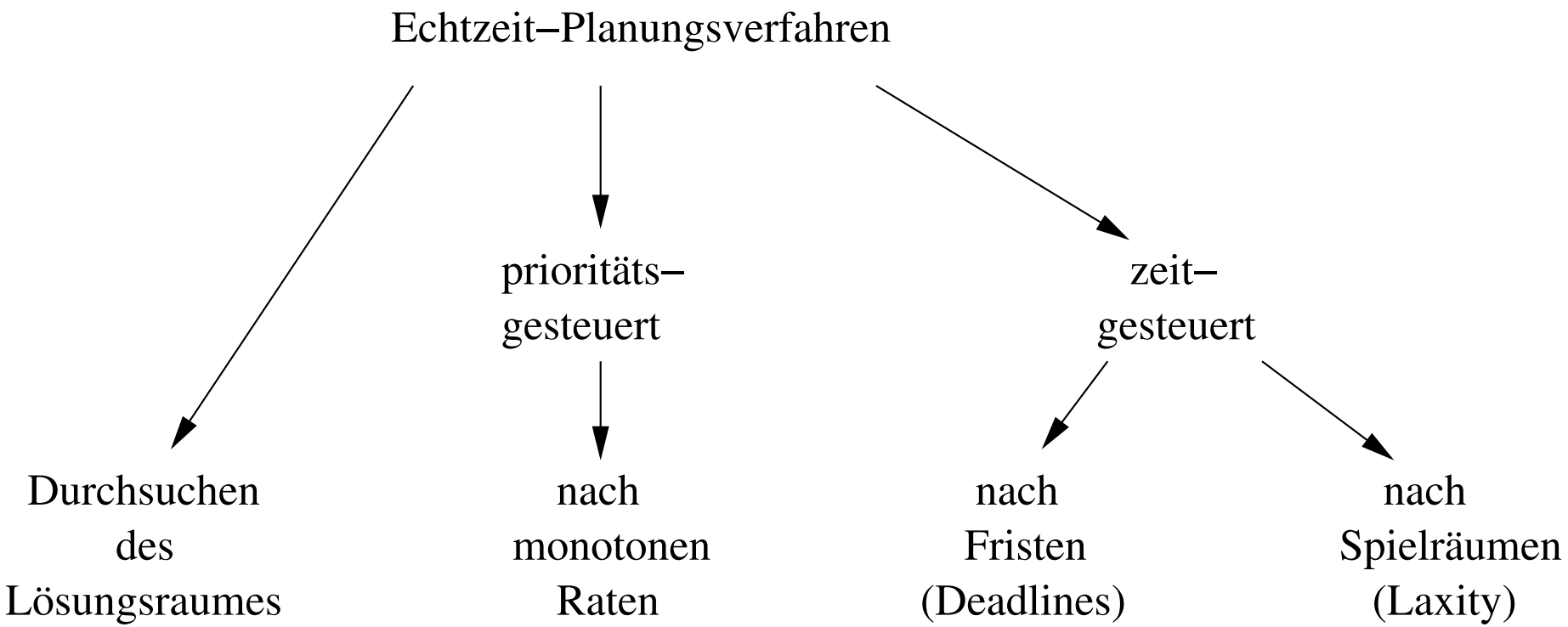
Echtzeit-Scheduling

- Scheduling in Realzeit-Systemen beinhaltet zahlreiche neue Aspekte. Hier nur erster kleiner Einblick.[5]
- Varianten in der Vorgehensweise
  - *Statisches Scheduling:*
    Alle Daten für die Planung sind vorab bekannt, die Planung erfolgt durch eine Offline-Analyse.
  - *Dynamisches Scheduling:*
    Daten für die Planung fallen zur Laufzeit an und müssen zur Laufzeit verarbeitet werden.
  - *Explizite Planung:*
    Dem Rechensystem wird ein vollständiger Ausführungsplan (Schedule) übergeben und zur Laufzeit befolgt (Umfang kann extrem groß werden).
  - *Implizite Planung:*
    Dem Rechensystem werden nur die Planungsregeln übergeben.

[5] Mehr dazu im Listenfach „Echtzeitverarbeitung“ im nächsten SoSe

Klassifizierung

Hochschule RheinMain



Periodische Prozesse

- Gewisse Prozesse müssen häufig zyklisch, bzw perodisch ausgeführt werden.
- **Hard-Realtime**-Prozesse müssen unter allen Umständen ausgeführt werden (ansonsten sind z.B. Menschenleben bedroht).
- Zeitliche **Fristen (*Deadlines*)** vorgegeben, zu denen der Auftrag erledigt sein muss.
- Scheduler muss die Erledigung aller Hard-Realtime-Prozesse innerhalb der Fristen **garantieren**.
- Scheduling geschieht in manchen Anwendungssystemen statisch vor Beginn der Laufzeit (z.B. Automotive). Dazu muss die Bedienzeit-Anforderung (z.B. worst case) bekannt sein.
- Im Falle von dynamischem Scheduling sind das **Rate-Monotonic** (RMS) und das **Earliest-Deadline-First** (EDF) Scheduling-Verfahren verbreitet.

Rate Monotonic Scheduling (RMS) (1)

- Ausgangspunkt: Periodisches Prozessmodell
  - Planungsproblem gegeben als Menge unterbrechbarer, periodischer Prozesse P_i mit Periodendauern Δp_i und Bedienzeiten Δe_i.
  - Perioden zugleich Fristen.
- RMS ordnet Prozessen **feste Prioritäten** proportional zur **Rate**[6] zu:

$$prio(i) < prio(j) \iff \frac{1}{\Delta p_i} < \frac{1}{\Delta p_j}$$

- Daher auch *fixed priority scheduling*
- Die meisten Echtzeit-Betriebssysteme unterstützen prioritätsbasiertes, unterbrechendes Scheduling

→ Voraussetzungen für die Anwendung sind unmittelbar gegeben

- Zur Festlegung der Prioritäten genügt allein die Kenntnis der Periodendauern Δp_i

[6] = Kehrwert der Periodendauer

Rate Monotonic Scheduling (RMS) (2)

- RMS Zulassungskriterium (*admission test*):
  Wenn für n periodische Prozesse gilt . . . :

$$\sum_{i=0}^{n} \frac{\Delta e_i}{\Delta p_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

- . . . dann ist bei Prioritätsvergabe nach RMS **garantiert**, dass alle Fristen eingehalten werden.
- Hinreichendes (nicht: notwendiges) Kriterium
- Einfach zu überprüfen, mathematisch beweisbare Garantie
- Erfordert Kenntnis der *worst case* Bedienzeiten (WCET)

Beispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 2 Prozessen

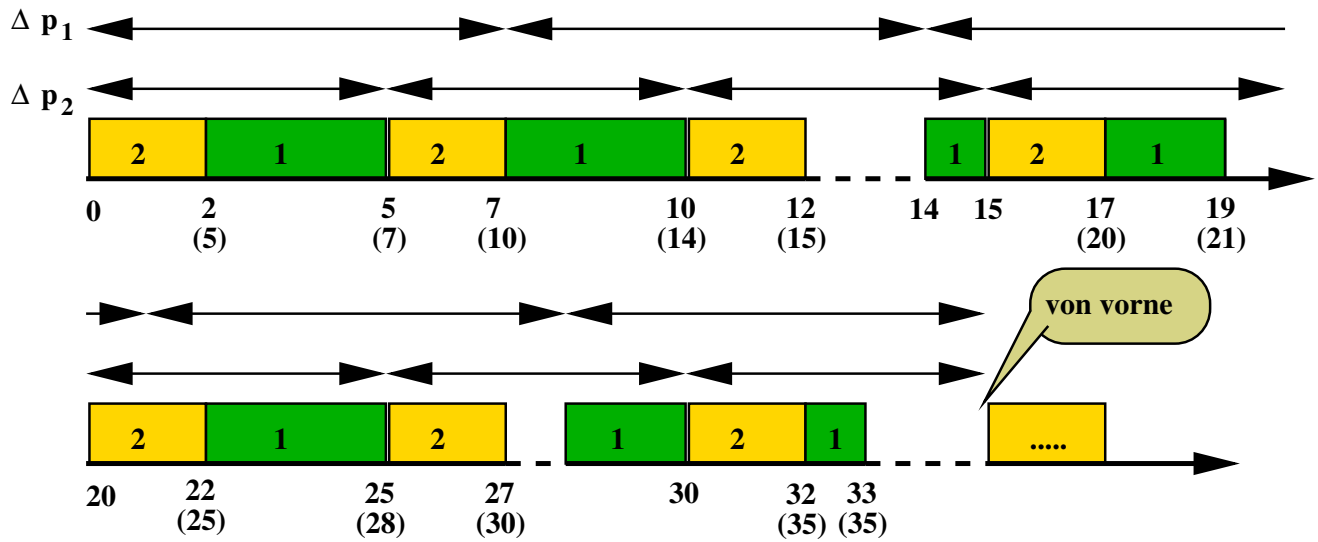
Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	3	7
2	2	5

$$\frac{3}{7} + \frac{2}{5} \approx 0,8286$$

$$2 \cdot \left(2^{\frac{1}{2}} - 1\right) \approx 0,8284$$

→ Kriterium knapp nicht erfüllt, trotzdem wurde ein Plan gefunden

- Aus RMS resultierender Schedule [7]:



[7] (wg. $\frac{1}{7} < \frac{1}{5}$ bekommt P_2 höhere Priorität)

Earliest Deadline First Scheduling (EDF) (1)

- Strategie: *Earliest Deadline First* (EDF)
  - Der Prozessor wird demjenigen Prozess P_i zugeteilt, dessen Frist d_i den kleinsten Wert hat (am nächsten ist)
  - Wenn es keinen rechenbereiten Prozess gibt, bleibt der Prozessor untätig (d.h. „idle“)
- Falls EDF keinen brauchbaren Plan liefert, gibt es keinen (!)
- Zur Planung nach EDF genügt allein die Kenntnis der Fristen, bzw. der Periodendauern Δp_i
- Die Umsetzung eines EDF-Planes mithilfe des prioritätsbasierten, unterbrechenden Scheduling erfordert die dynamische Änderung von Prozessprioritäten zur Laufzeit.
- Daher auch *dynamic priority scheduling*
- Nicht alle Echtzeitbetriebssysteme unterstützen dynamische Prioritäten.

Earliest Deadline First Scheduling (EDF) (2)

- EDF Zulassungskriterium (*admission test*):
  Wenn für n periodische Prozesse gilt . . . :

$$\sum_{i=0}^{n} \frac{\Delta e_i}{\Delta p_i} \leq 1$$

- . . . dann ist bei Planung nach EDF **garantiert**, dass alle Fristen eingehalten werden.
- Notwendiges und hinreichendes Kriterium
- Einfach zu überprüfen, mathematisch beweisbare Garantie
- Erfordert Kenntnis der *worst case* Bedienzeiten (WCET)

Beispiel

Hochschule RheinMain

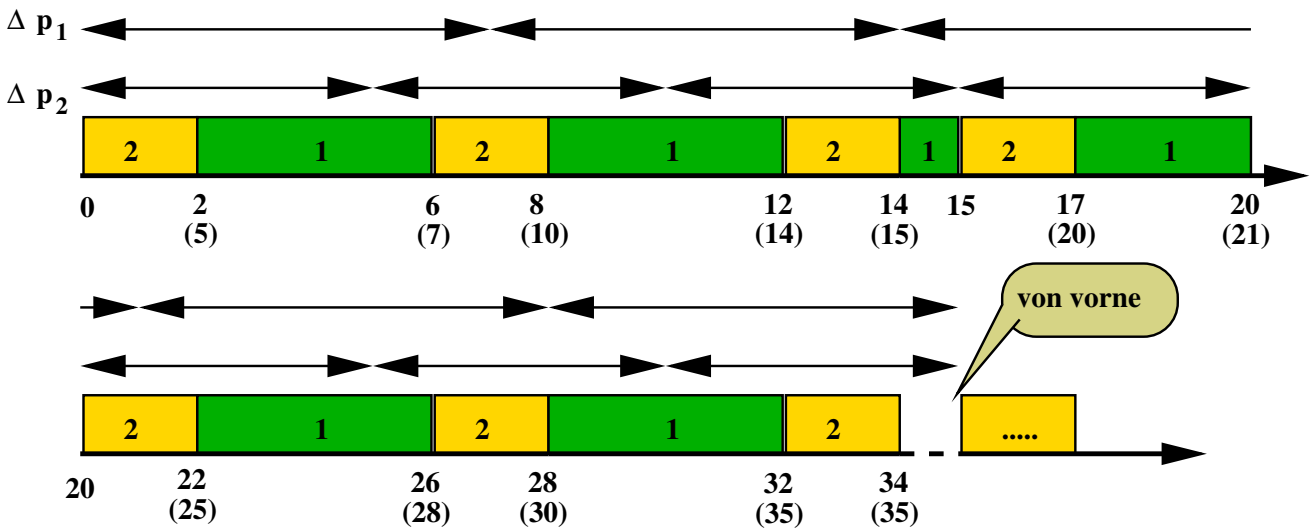
Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	4	7
2	2	5

$$\frac{4}{7} + \frac{2}{5} \approx 0,97143 < 1$$

$\rightarrow$ Kriterium erfüllt, Plan existiert

- Aus EDF resultierender Schedule



Gegenüberstellung RMS ↔ EDF

- RMS
  - \- - Keine 100% Auslastung möglich
  - ++ Auf gängigen Echtzeit-BS direkt einsetzbar
  - ++ Bei Überlastsituationen werden zunächst niedrig priorisierte Prozesse nicht mehr bedient
- EDF
  - ++ 100% Auslastung möglich
  - \- - Erfordert dynamische Prioritäten - nicht auf allen Echtzeit-BS möglich
  - \- - Bei Überlastsituationen erratisches Verhalten

Zusammenfassung

Hochschule RheinMain

Was haben wir in Kap. 4 gemacht?

- Scheduling-Verfahren
  - Kenngrößen wie Antwortzeit, Auslastung und Fairness relevant
  - einfache Verfahren: FCFS, SJF, PRIO
  - unterbrechende Verfahren Round-Robin, statische Prioritäten, Mehrschlangen-Verfahren
  - flexible Feedback-Algorithmen, nach denen Prioritäten dynamisch neu berechnet werden
  - Beispiel Linux
  - Grundzüge des Echtzeit-Scheduling: RMS und EDF