1) **Betriebsarten**

Stapelverarbeitung (Lochkarten)

- Rechnerfamilien für wissenschaftliche und kommerzielle Berechnungen
- günstig (ICs statt Röhren)
- Software sollte auf diversen Rechnern laufen

Mehrprogrammbetrieb (Mutliprogramming, Multitasking)

- Gleichzeitiges Bereithalten mehrerer Jobs im Hauptspeicher (Partitionierung)
- Auf anderen Job umschalten statt auf E/A zu warten
- Timesharing
    - Jeder Benutzer hat Zugang zum System über sein Terminal

2) **Betriebssystemstrukturen**

Kernaufruf

- Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- BS Code bestimmt die Nummer des angeforderten Dienstes.
- BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- Kontrolle wird an das Anwendungsprogramm zurückgegeben.
- Wichtig: Kern selbst ist passiv (Menge von Datenstrukturen und Prozeduren)

Betriebsmodi

- meisten 2 Modi (privilegiert, nicht-privilegiert), bei x86 4 Modi.
- Hardwaresicht (Privilegierter Modus)
    - Sperren von Unterbrechungen, Zugriff auf Speicherverwaltung-Hardware
    - Exceptions (Interrupts,TRAPs und Faults (z.B. division by 0)) schalten in den privilegierten Modus
- Betriebssystemsicht (Benutzungsmodus = nicht-privilegiert)
    - Beschränkter Zugriff auf Betriebsmittel
    - Unberechtigter Zugriff auf Betriebsmittel lösen Faults aus
    - Unerlaubte Operationen lösen Faults aus
    - Systemcall = expliziter TRAP Befehl
- Betriebssystemsicht (privilegiert)
    - Uneingeschränkter Zugriff auf alle Betriebsmittel
    - Faults und Exceptions führen zum Absturz

Monolithische Systeme (Windows, Unix, ...)

- prozedurorientiert
    - Kern ist passiv und der Code besteht aus einer Menge von Prozeduren
    - Struktur: Hauptprogramme → Dienstprozeduren → Hilfsprozeduren
- Nachteil: Viel Code → viele Fehler, nicht alle Anwedungen benötigen alle Dienste , Art und Anzahl der Dienste vom Kern vorgegeben

Client/Server-Strukturen (Mikrokerne)

- Ansatz: Nur Dienste die im Kernmodus laufen müssen, dürfen in diesem laufen
- Dateisystem, Netzwerkprotokolle, Speicherverwaltung müssen nicht im Kern sein , „Server“-Prozesse (ohne besondere Privilegien) bieten diese Dienste an
- Kern bietet nur Dienste zur Kommunikation zwischen Klienten (Anwendungen) und Servern untereinander an
- Dienste werden durch Nachtrichten per IPC: Interprozesskommunikation von Servern angefordert (send & receive)
- Server liefern Dienste auch mit IPC-Nachrichten (reply & wait)

-Vorteile: Isolation der Systemteile, Erweiterbarkeit, Nachrichtenbasiert
- Policy & Mechanism
    - Beispiel Speicherverwaltung
        - Strategie (policy): Zuteilung von Speicher an Prozesse
        - Mechanismus (mechanism): Konfiguration der Hardware
- µKern SOLLTE klein und wenig komplex sein

-Single Server: Monolithisches BS in Server umwandeln
- Mehrere BS in einem Rechner, große Trusted Code Base, schlechte Performance

Virtualisierung
- Virtuelle Maschinen (Beispiel VM/370)
    - Trennen der Funktionen „Mehrprogrammbetrieb“ und „erweiterte Maschine“
    - Virtualisierung durch Hypervisor
    - virtuelle Maschinene als identische Kopien der Hardware
    -in jeder virtuellen Maschine: übliches Betriebssystem
- Virtualisierbarkeit (Anforderung: Identisches Verhalten der VM)
    - Emulation: Nachbild der HW ins SW (ineffizient!) [Bochs, JWVM]
    - Virtualisierung: die meisten Befehle werden von der realen Hardware ausgeführt,der Rest emuliert(schnell, Architekturabhängig)[QEMU,VMWare]
    - Paravirtualisierung (Falls nicht virtualisierbar): Priviligierte Befehle des Gast-BS durch „Hypercalls“ (= Aufrufe in den Hypervisor) ersetzen. Schnell oder schneller als Virtualisierung, aber Gast-BS muss angepasst werden. [Xen, KVM, Hyper-V]

3) **Prozesse und Threads**

Prozessmodell
- Prozess: ein in sich in Ausführung befindliches Programm inkl.Stack, Register,Pc
    - Menge von (virtuellen) Adressen, von Prozess zugreifbar
    - Programm und Daten in Addressraum sichtbar
- Verhältnis Prozessor - Prozessor
    - Prozess besitzt konzeptionel eigenen virtuellen Prozessor
    - Reale(r) Prozessor(en) werden zwischen virtuellen Prozessoren umgeschaltet (Mehrprogrammbetrieb)
    - Umschaltungseinheit heißt Scheduler oder Dispatcher
    - Umschaltvorgang heißt Prozesswechsel oder Kontextwechsel
- Prozesserzeugung
    1) feste Menge von Prozessen werden beim Systemstart erzeugt
    einfache, meist eingebettete System [Motorsteuerung, Videorekorder]
    einfache Verwaltung, deterministisches Zeitverhalten, unflexibel

    2) dynamisch (es können im Laufe der Zeit neue Prozesse erzeugt werden)
    impliziert die Bereitstellung geigneter Systemaufrufen durch BS
- Prozessende
    1) freiwillig: Prozess ist fertig (egal ob erfolgreich oder nicht)
    2) unfreiwillig: Prozess WIRD beendet (Bsp: Division 0, Segmentation Fault)
- Prozesshierarchie (Unix ja, Windows nein [Prozesse gleichwertig])

-Prozesszustände (aktiv, bereit, schlafend/blockiert) selten auch initiiert, terminiert

Implementierung
- PCB (Process Control Block)
  - Prozessverwaltung: Register, Id, Pc, StackPtr, Flags, Signal, Parent, Zustand
  - Speicherverwaltung: zeiger auf .text .data .bss, real und effektiv UID & GID
  - Dateisystem: effektive UID & GID, Flags, Wurzel- & aktuelles Verzeichnis
  - Zeiger zur Verkettung des PCB in (verschiedenen) Warteschlangen
- Scheduler-Aktivierung
  - kooperatives Multitasking: Problem MUSS Kontrolle an BS abgeben
  - preemptiv: Code wird unterbrochen bei z.B. Ablauf eines Timers, Scheduler wird aufgerufen
- Unterbrechungsbehandlung
  - Interrupt-Handler: Interrupt-Vektor-Tabelle (IVT) enthält Interrupts mit IDs
  - Ablauf: Pc (u.a.) wird durch HW auf dem Stack abgelegt
    HW lädt Pc-Inhalt aus Unterbrechungsvektor
    Assembly-Routine rettet Registerinhalte
    Assembly-Routine bereitet den neuen Stack vor
    C-Prozedur markiert den unterbrochenen Prozess als bereit
    Scheduler bestimmt den nächsten auszuführenden Prozess
    C-Prozedur gibt Kontrolle an die Assembly-Routine zurück
    Assembly-Routine startet den ausgewählten Prozess
- Interrupts aus Sicht des Prozesses
  - IRT: Interrupt Response Time
  - PDLT Process Dispatch Latency Time
  - SWT Process Switch Time

Threads (Leichtgewichtsprozesse für billige Nebenläufigkeit im Prozessadressraum)
- Idee einer „parallel ausgeführten Programmfunktion“
- eigener Prozessor-Context (Registerinhalte usw.)
- eigener Stack (i.d.R. 2, getrennt für unser und kernel mode)
- eigener kleiner privater Datenbereich (Thread Local Storage)
- Threads nutzen alles Betriebsmittel, Programm- & Adressraum des Prozesses
- WICHTIG: Bein 1-Prozessorsystemen kein Performancegewinn
- Kooperationsformen: Verteiler-/Arbeitermodell, Teammodell, Fließbandmodell
- Implementierung
  - Thread-Bibliothek (User level threads)
    - Threadfunktionen/Kontextwechsel auf Applikationsebene
    - einfache Implementierung, keine Nutzung von MehrprozessorArch
  - Im BS-Kern (Kernel level threads)
    - Threads als Einheiten denen Prozessoren zugeordnet sind
    - Nutzung von Mehrprozessor Architekturen,Kernelunterstüzung nötig

4) **Scheduling (Priorität- oder Zeitscheiben-basiert)**

Begriffe
-Bedienzeit: Zeitdauer für reine Bearbeitung des Auftrags
-Antwortzeit: Zeitdauer vom Eintreffen bis zur Fertigstellung des Auftrags
Bei Dialogaufträgen Zeitdauer von Benutzereingabe bis Ausgabe
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

Moderne Anforderungen

-Scheduling wird von Applikationsebene gesteuert

Non-Preemptive Scheduling (Annahme: Bekannte Bedienzeiten)

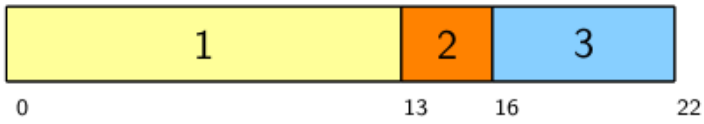
-FCFS (first come first served): Ready-Queue als FIFO Liste

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: $(6 + 9)/3 = 5$

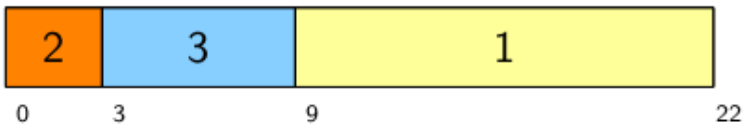
-SJF (Shortest Job First)

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

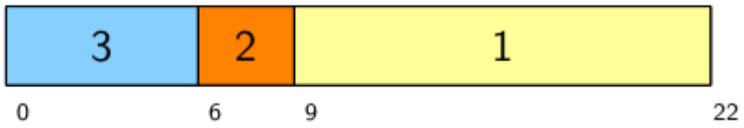
-Prioritäts-Scheduling: Jeder Auftrag hat statische Priorität
höchste Priorität hat Vorrang
Bei gleicher Priorität FCFS

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

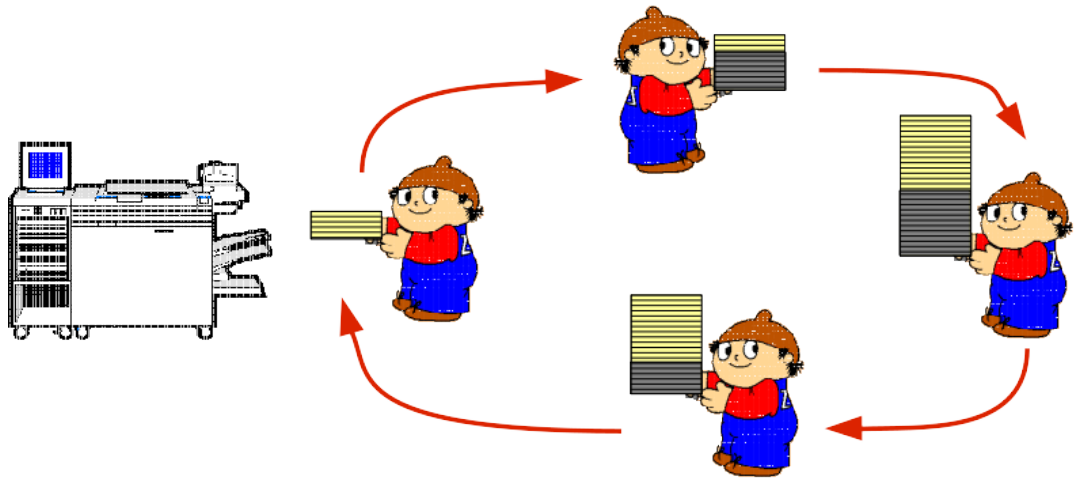
Durchschnittliche Wartezeit[4]:
$(9+6)/3 = 5$

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

Preemptive Scheduling

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.

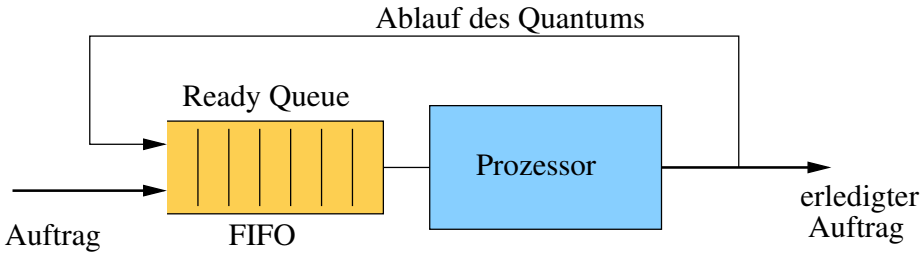
Implementierung:

- 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

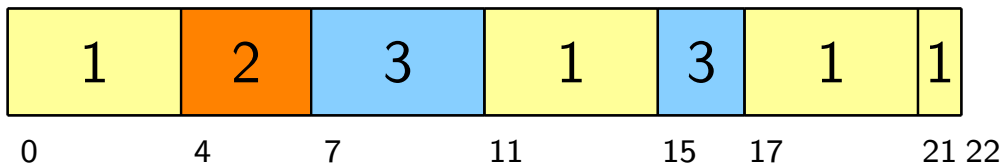
Gegeben: Prozessmenge mit 3 Prozessen

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

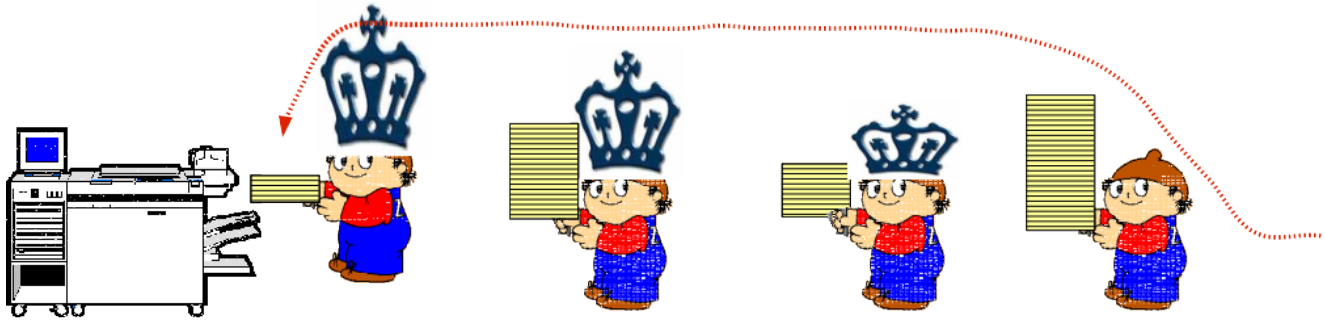
Durchschnittliche Wartezeit:
$(9 + 4 + 11)/3 = 8$

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.

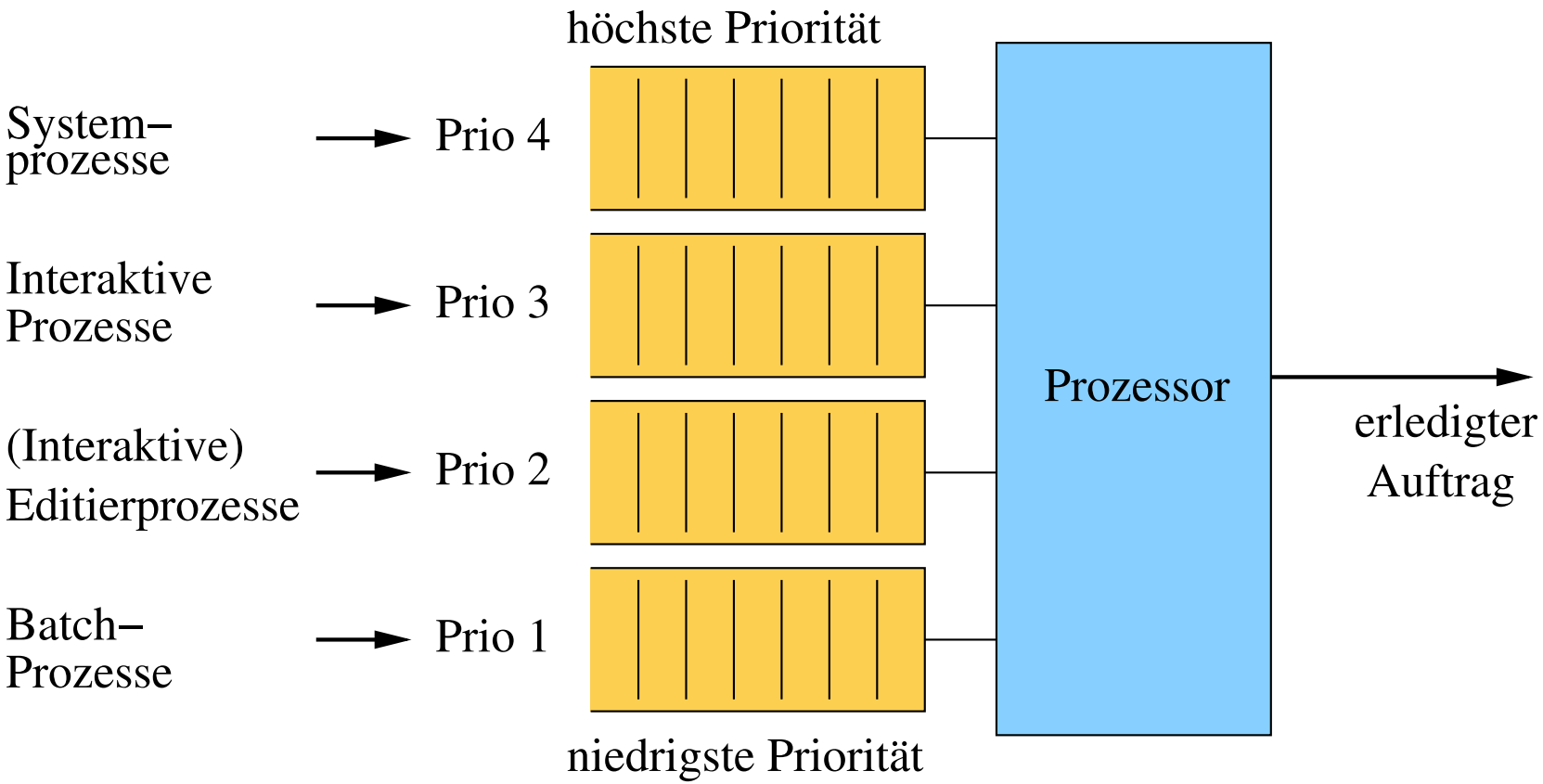
Mehrschlangen-Scheduling

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).

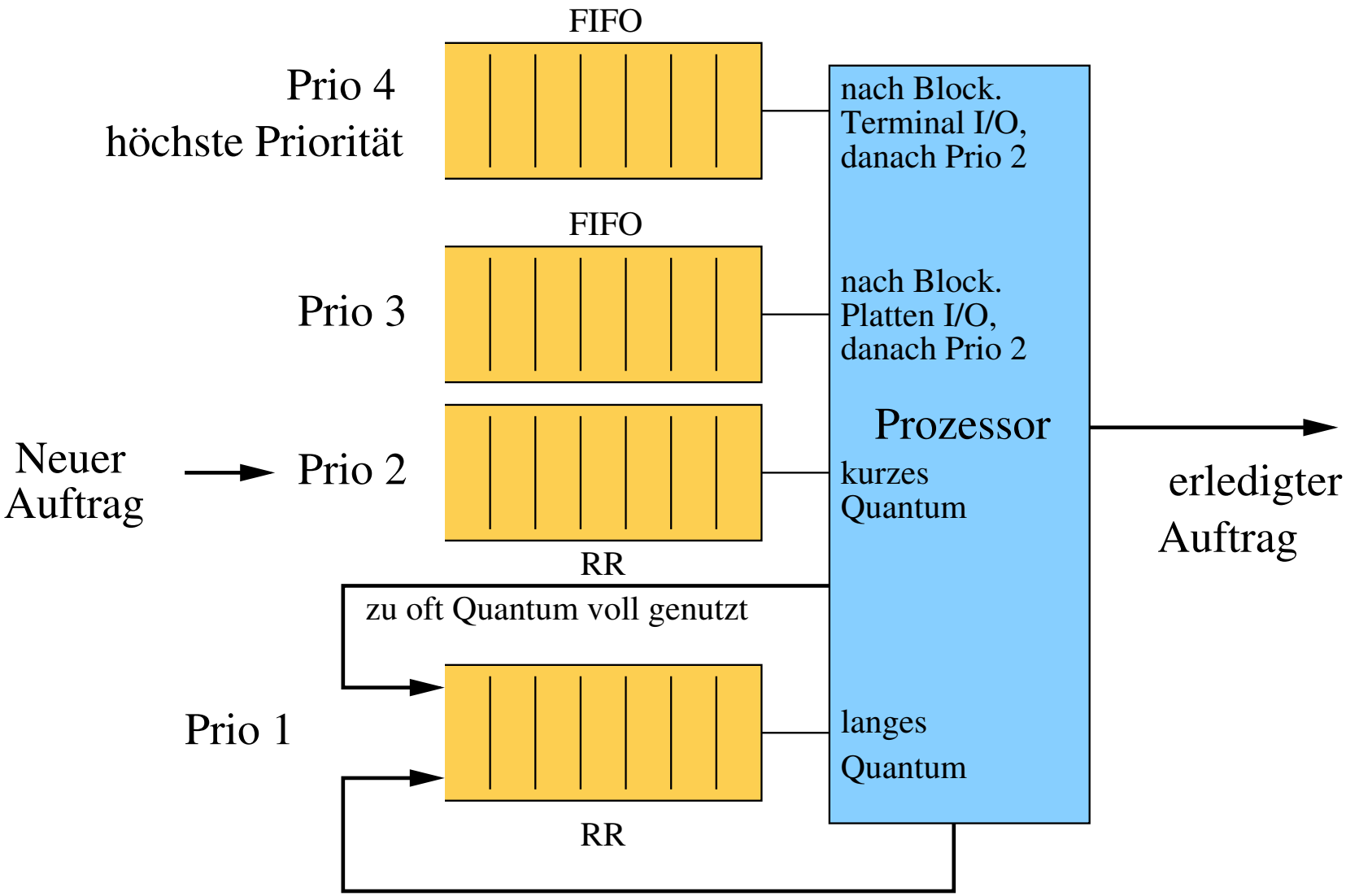
Bsp. 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von Langläufern).

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“)

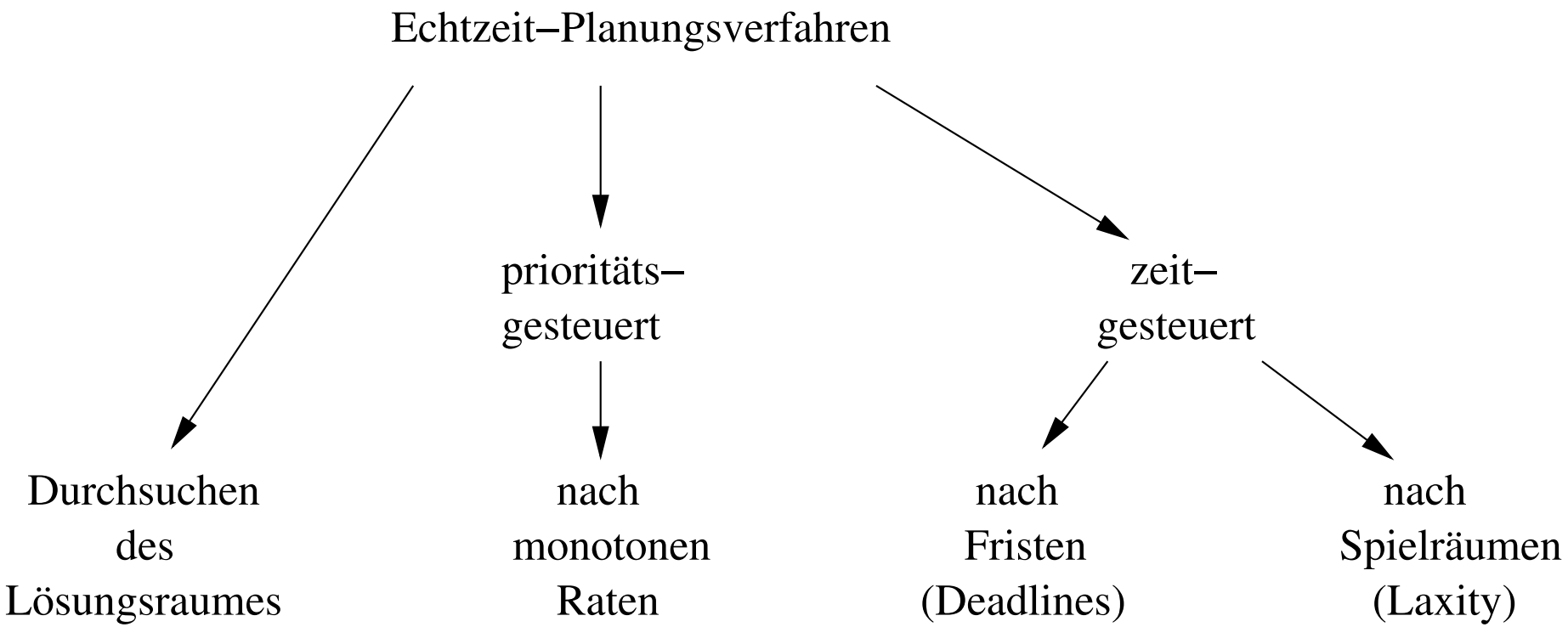
Echtzeit-Scheduling

- 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

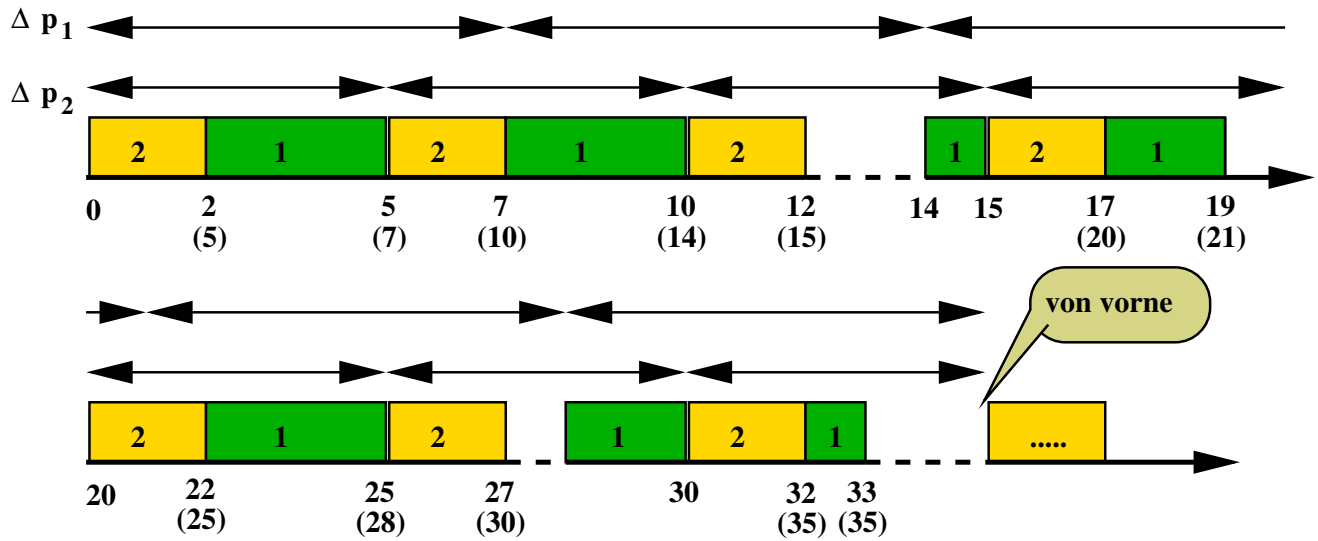
Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	3	7
2	2	5

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

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

→ 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

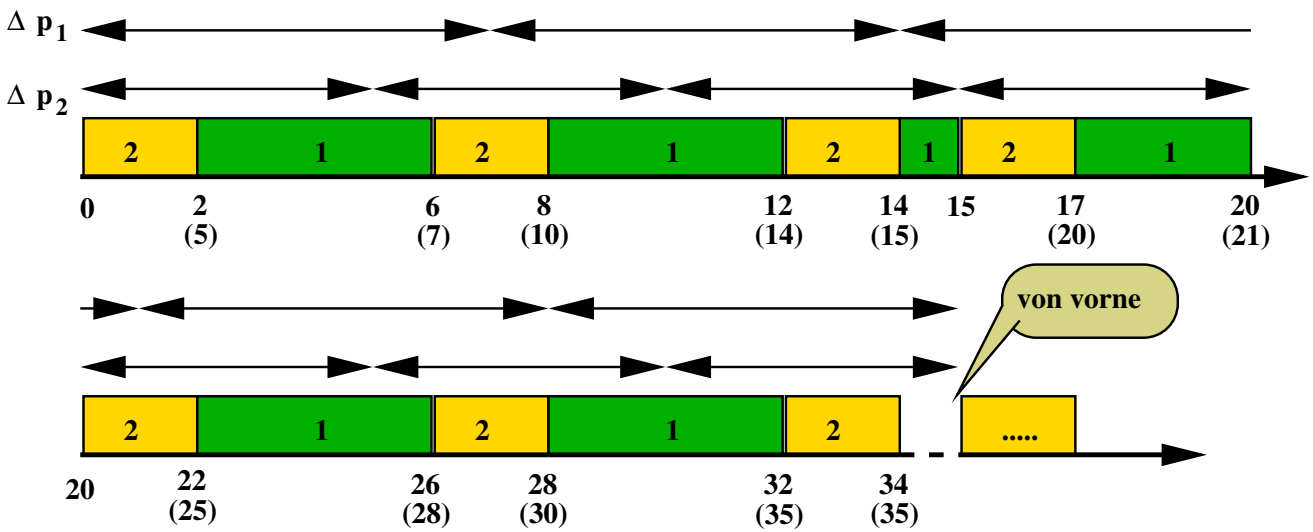
Gegeben: Prozessmenge mit 2 Prozessen

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