8. Foliensatz Betriebssysteme

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Fachbereich Informatik und Ingenieurwissenschaften
christianbaun@fb2.fra-uas.de

Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - den Unterschied zwischen **Interrupts** und **Exceptions**
  - welche Schritte der **Dispatcher** (Prozessumschalter) beim Prozesswechsel durchführt
  - was **Scheduling** ist
    - wie **präemptives Scheduling** und **nicht-präemptives Scheduling** funktioniert
    - die Arbeitsweise verschiedener **Scheduling-Verfahren**
    - warum **moderne Betriebssysteme** nicht nur ein einziges Scheduling-Verfahren verwenden
    - wie das **Scheduling moderner Betriebssysteme** im Detail funktioniert

Übungsblatt 8 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

Unterbrechungen

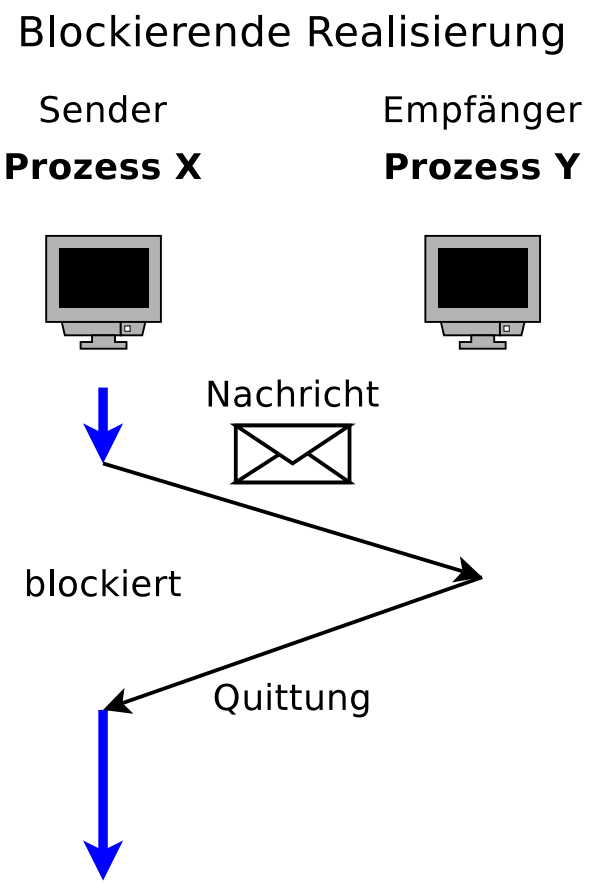
- Häufig treten unvorhersehbare Ereignisse ein, auf die ein Computer-System reagieren muss
- Unterbrechungen sind Ereignisse, deren Behandlung keinen Aufschub zulässt
- Häufige Unterbrechungen:
  - **Fehlersituation** (Fehler bei einer Rechenoperation)
    - Division durch Null, Gleitkommafehler, Adressfehler, usw.
  - **Software-Interrupt** bzw. **Exception** (Wird durch einen Prozess ausgelöst)
    - Beispiele sind die Exception 0x80 (siehe Foliensatz 7), um vom Benutzermodus in den Kernelmodus zu wechseln und der Einzelschrittbetrieb beim Programmtest (Debugging, Trace)
  - **Hardware-Interrupt**
    - Ein-/Ausgabe-Geräte liefern Rückmeldungen an einen Prozess

Beispiel für eine Unterbrechung

- Die Prozesse X und Y kommunizieren über ein Netzwerk
  - Beide Prozesse werden auf unterschiedlichen Rechnern ausgeführt
  - Antwortet ein Prozess nicht innerhalb eines festgelegten Zeitraums (Timeout) auf eine Nachricht, soll diese erneut geschickt werden
    - Grund: Es wird von einem Verlust der Nachricht ausgegangen
- Es gibt 2 Möglichkeiten eine solche Bedingung zu realisieren:
  - Auf **blockierende** Art
  - Auf **nicht-blockierende** Art

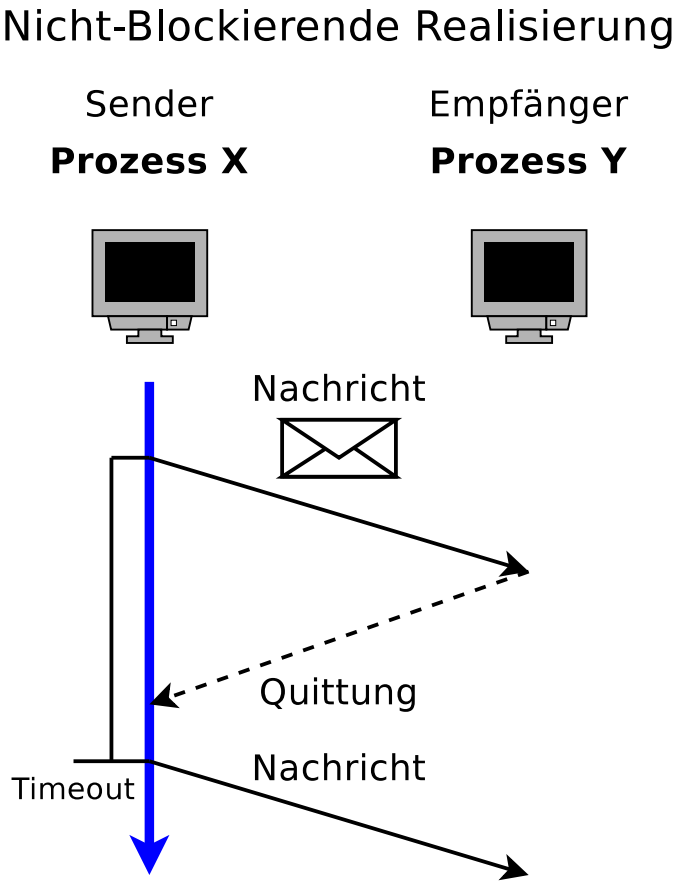
Blockierende Realisierung

- Prozess X wird so lange blockiert, bis die Nachricht quittiert ist oder es zum Timeout kommt
- Kommt die Quittung an, darf Prozess X weiterlaufen
  - Ansonsten muss Prozess X die Nachricht neu senden
- Nachteil: Es entstehen lange Leerlaufzeiten für Prozess X



Nicht-blockierende Realisierung

- Prozess X läuft nach dem Senden der Nachricht normal weiter
  - Kommt es wegen einer fehlenden Quittung zum Timeout, suspendiert das Betriebssystem den Prozess
- Der Kontext (siehe Foliensatz 7) des Prozesses wird gesichert und eine Prozedur zur Behandlung der Unterbrechung aufgerufen
  - Im Beispiel würde die Prozedur die Nachricht erneut senden
  - Ist die Prozedur zur Behandlung der Unterbrechung beendet, wird der Prozess reaktiviert



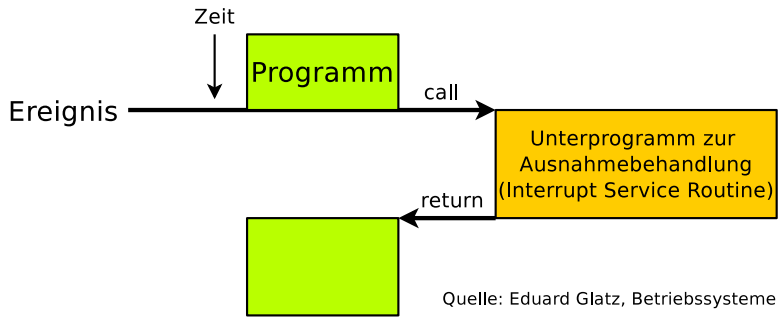
Nicht-blockierende Unterbrechungen können mit **Interrupts** und **Exceptions** realisiert werden

Unterprogramme ohne Rückgabewert heißen **Prozeduren**. Unterprogramme mit Rückgabewert heißen **Funktionen** oder **Methoden**

Unterbrechungsarten – Interrupts (1/2)

- Interrupts sind **externe Unterbrechungen**
  - Sie werden durch Ereignisse außerhalb des zu unterbrechenden Prozesses ausgelöst (z.B. ein Ein-/Ausgabe-Gerät meldet ein E/A-Ereignis)
- Das Konzept der Interrupts wird von der Hardware angeboten
  - Es gibt auch Softwareinterrupts (Exceptions)
    - Diese werden wie Hardwareinterrupts behandelt, aber von Software ausgelöst

Ein Interrupt signalisiert ein Ereignis und das Betriebssystem bietet zur **Behandlung** des Ereignisses einen „*Event Handler*“, die sogenannte **Interrupt Service Routine** an



- Die CPU wird unterbrochen und es wird zur Unterbrechungsbehandlung die **Interrupt Service Routine** des Kernels aufgerufen
  - Der Befehlszähler wird auf die Adresse der Unterbrechungsroutine gesetzt und es wird dort weitergearbeitet
  - Das Betriebssystem sichert den Prozesskontext (siehe Foliensatz 7) und stellt diesen nach Abschluss der Unterbrechungsroutine wieder her

Unterbrechungsarten – Interrupts (2/2)

- Das Betriebssystem verwaltet eine Liste mit den Adressen aller Unterbrechungsroutinen
  - Diese Liste heißt **Unterbrechungsvektor** (Interrupt Vector)
- Interrupts sind notwendig, um...
  - schnell auf Signale von Ein-/Ausgabe-Geräten (z.B. Maus, Tastatur, Festplatte, Netzwerk, usw.) reagieren zu können
  - auf zeitkritische Ereignisse reagieren zu können
- **Ohne Interrupts ist präemptives Multitasking nicht möglich**
  - Präemptives Multitasking = verdrängendes Multitasking
    - Bei diesem kann einem Prozess die CPU vor seiner Fertigstellung entzogen werden

Unterbrechungsarten – Exceptions

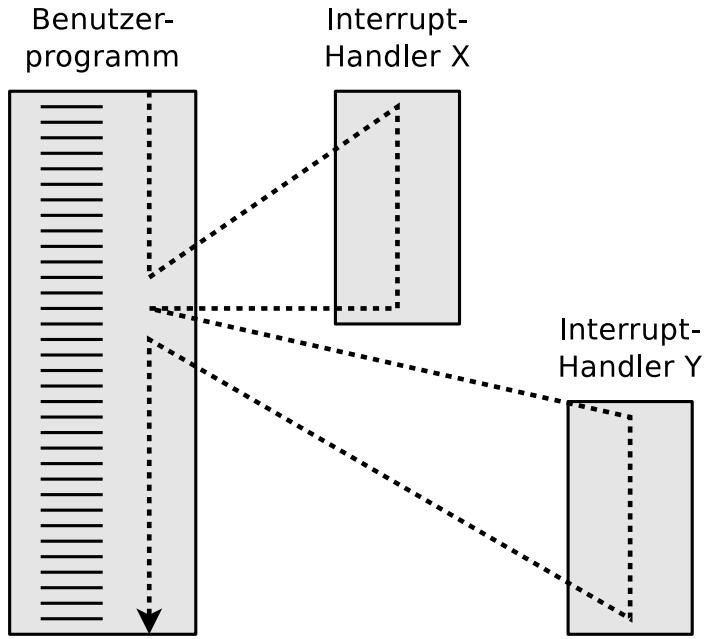
- Exceptions sind **interne Unterbrechungen** oder Ausnahmen
  - Werden vom Prozess selbst ausgelöst
  - Können an jeder Stelle im Programmcode ausgelöst werden
- Auch bei Exceptions kommt es zu einer Unterbrechung der CPU und es wird eine Unterbrechungsbehandlung (Unterbrechungsroutine) im Kernel aktiviert
- Exceptions werden in der Softwareentwicklung eingesetzt, um die Programme robuster zu machen gegen...
  - fehlerhafte Eingaben
  - Programmierfehler (Division durch 0, Addition verursacht Überlauf)
  - Gerätefehler (Gerät nicht erreichbar, Speichergerät voll)
- Weiterer Vorteil von Exceptions:
  - Trennung zwischen Algorithmus und Fehlerbehandlung

Konflikte bei Unterbrechungen (Interrupts)

- 2 mögliche Konflikte bei der Unterbrechungsbehandlung:
  - Während einer Unterbrechungsbehandlung treten weitere Interrupts auf
  - Es kommt zu mehreren Unterbrechungen gleichzeitig
- 2 mögliche Lösungen:
  - **Sequentielle Interrupt-Verarbeitung**
  - **Verschachtelte Interrupt-Verarbeitung**

Sequentielle Interrupt-Verarbeitung

- **Sequentielle Interrupt-Verarbeitung**
- Die Interrupts werden strikt nacheinander bearbeitet und nicht selbst unterbrochen
- Nachteil: Prioritäten und zeitkritische Reaktionen werden ignoriert

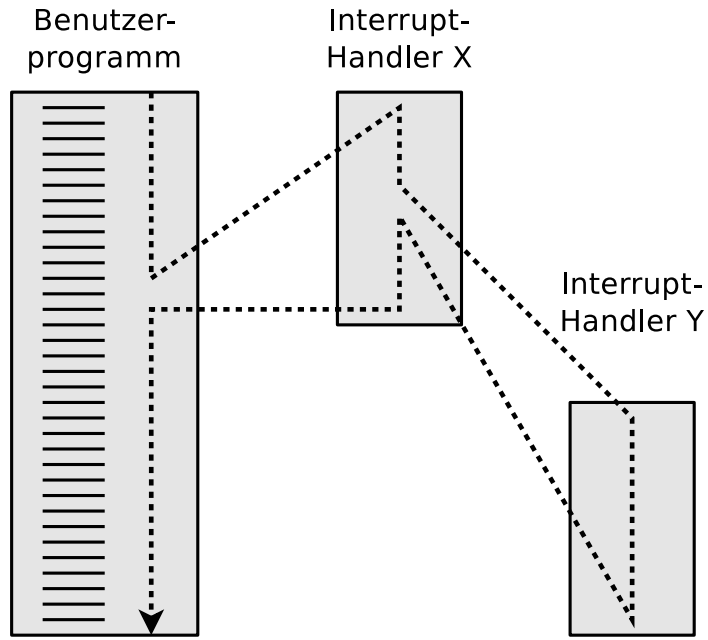


Quelle: William Stallings. Betriebssysteme. Pearson Studium. 2003

Verschachtelte Interrupt-Verarbeitung

- **Verschachtelte Interrupt-Verarbeitung** (engl. *Nested Interrupts*)
- Für die Interrupts werden Prioritäten festgelegt

- Unterbrechungsbehandlungen können unterbrochen werden, wenn eine Unterbrechung mit höherer Priorität auftritt
- Unterbrechungen mit niederer Priorität werden zurückgestellt, bis alle Unterbrechungen mit höherer Priorität abgearbeitet sind
- Nachteil: Unterbrechungen mit niedriger Priorität werden eventuell stark verzögert



Quelle: William Stallings. Betriebssysteme. Pearson Studium. 2003

Das Echtzeitbetriebssysteme QNX Neutrino und Windows CE 5.0 unterstützen beide verschachtelte Interrupts

`http://www.qnx.com/developers/docs/660/topic/com.qnx.doc.neutrino.sys_arch/topic/kernel_Nested_interrupts.html`
`http://msdn.microsoft.com/de-de/library/ms892539.aspx`

Prozesswechsel – Der Dispatcher (1/2)

- Aufgaben von Multitasking-Betriebssystemen sind u.a.:
  - **Dispatching**: Umschalten des Prozessors bei einem Prozesswechsel
  - **Scheduling**: Festlegen des Zeitpunkts des Prozesswechsels und der Ausführungsreihenfolge der Prozesse
- Der **Dispatcher** (Prozessumschalter) führt die Zustandsübergänge der Prozesse durch

Wir wissen bereits...

- Beim Prozesswechsel entzieht der Dispatcher dem rechnenden Prozess die CPU und teilt sie dem Prozess zu, der in der Warteschlange an erster Stelle steht
- Bei Übergängen zwischen den Zuständen `bereit` und `blockiert` werden vom Dispatcher die entsprechenden Prozesskontrollblöcke aus den Zustandslisten entfernt neu eingefügt
- Übergänge aus oder in den Zustand `rechnend` bedeuten immer einen Wechsel des aktuell rechnenden Prozesses auf der CPU

Beim Prozesswechsel in oder aus dem Zustand `rechnend`, muss der Dispatcher...

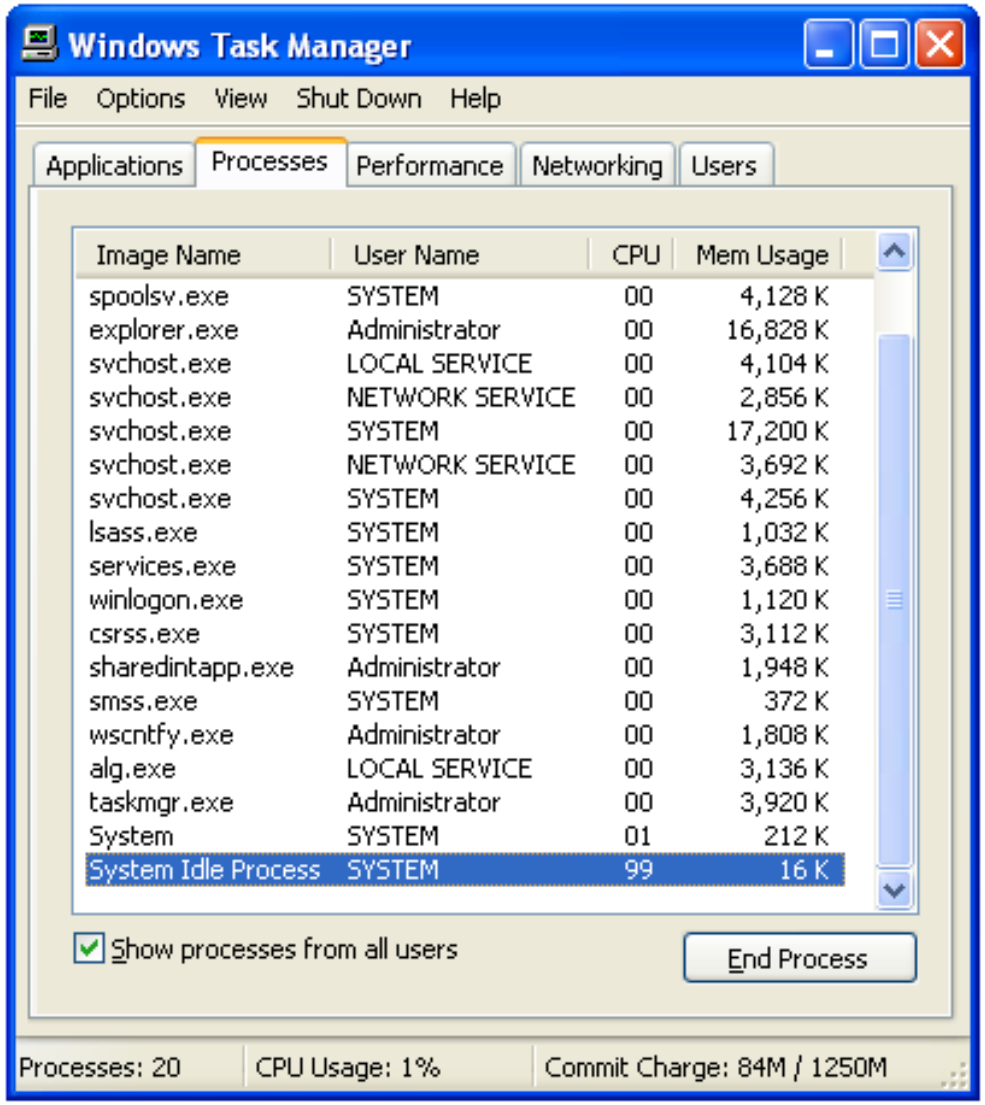
- den Kontext, also die Registerinhalte des aktuell ausgeführten Prozesses im Prozesskontrollblock speichern (retten)
- den Prozessor einem anderen Prozess zuteilen
- den Kontext (Registerinhalte) des jetzt auszuführenden Prozesses aus seinem Prozesskontrollblock wieder herstellen

Prozesswechsel – Der Dispatcher (2/2)

Bildquelle: Wikipedia

Der Leerlaufprozess (*System Idle Process*)

- Bei Windows-Betriebssystemen seit Windows NT erhält die CPU zu jedem Zeitpunkt einen Prozess
- Ist kein Prozess im Zustand `bereit`, kommt der **Leerlaufprozess** zum Zug
- Der Leerlaufprozess ist immer aktiv und hat die niedrigste Priorität
- Durch den Leerlaufprozesses muss der Scheduler nie den Fall berücksichtigen, dass kein aktiver Prozess existiert
- Seit Windows 2000 versetzt der Leerlaufprozess die CPU in einen stromsparenden Modus



Scheduling-Kriterien und Scheduling-Strategien

- Beim Scheduling legt des Betriebssystem die Ausführungsreihenfolge der Prozesse im Zustand `bereit` fest
- Keine Scheduling-Strategie...
  - ist für jedes System optimal geeignet
  - kann alle Scheduling-Kriterien optimal berücksichtigen
    - Scheduling-Kriterien sind u.a. CPU-Auslastung, Antwortzeit (Latenz), Durchlaufzeit (*Turnaround*), Durchsatz, Effizienz, Echtzeitverhalten (Termineinhaltung), Wartezeit, Overhead, Fairness, Berücksichtigen von Prioritäten, Gleichmäßige Ressourcenauslastung...
- Bei der Auswahl einer Scheduling-Strategie muss immer ein Kompromiss zwischen den Scheduling-Kriterien gefunden werden

Nicht-präemptives und präemptives Scheduling

- 2 Klassen von Schedulingverfahren existieren
  - **Nicht-präemptives Scheduling** bzw. **Kooperatives Scheduling** (nicht-verdrängendes Scheduling)
    - Ein Prozess, der vom Scheduler die CPU zugewiesen bekommen hat, behält die Kontrolle über diese bis zu seiner vollständigen Fertigstellung oder bis er die Kontrolle freiwillig wieder abgibt
    - Problematisch: Ein Prozess kann die CPU so lange belegen wie er will

Beispiele: Windows 3.x und MacOS 8/9

  - **Präemptives Scheduling** (verdrängendes Scheduling)
    - Einem Prozess kann die CPU vor seiner Fertigstellung entzogen werden
    - Wird einem Prozess die CPU entzogen, pausiert er so lange in seinem aktuellen Zustand, bis der Scheduler ihm erneut die CPU zuteilt
    - Nachteil: Höherer Overhead als nicht-präemptives Scheduling
    - Die Vorteile von präemptivem Scheduling, besonders die Beachtung von Prozessprioritäten, überwiegen die Nachteile

Beispiele: Linux, MacOS X, Windows 95 und neuere Versionen

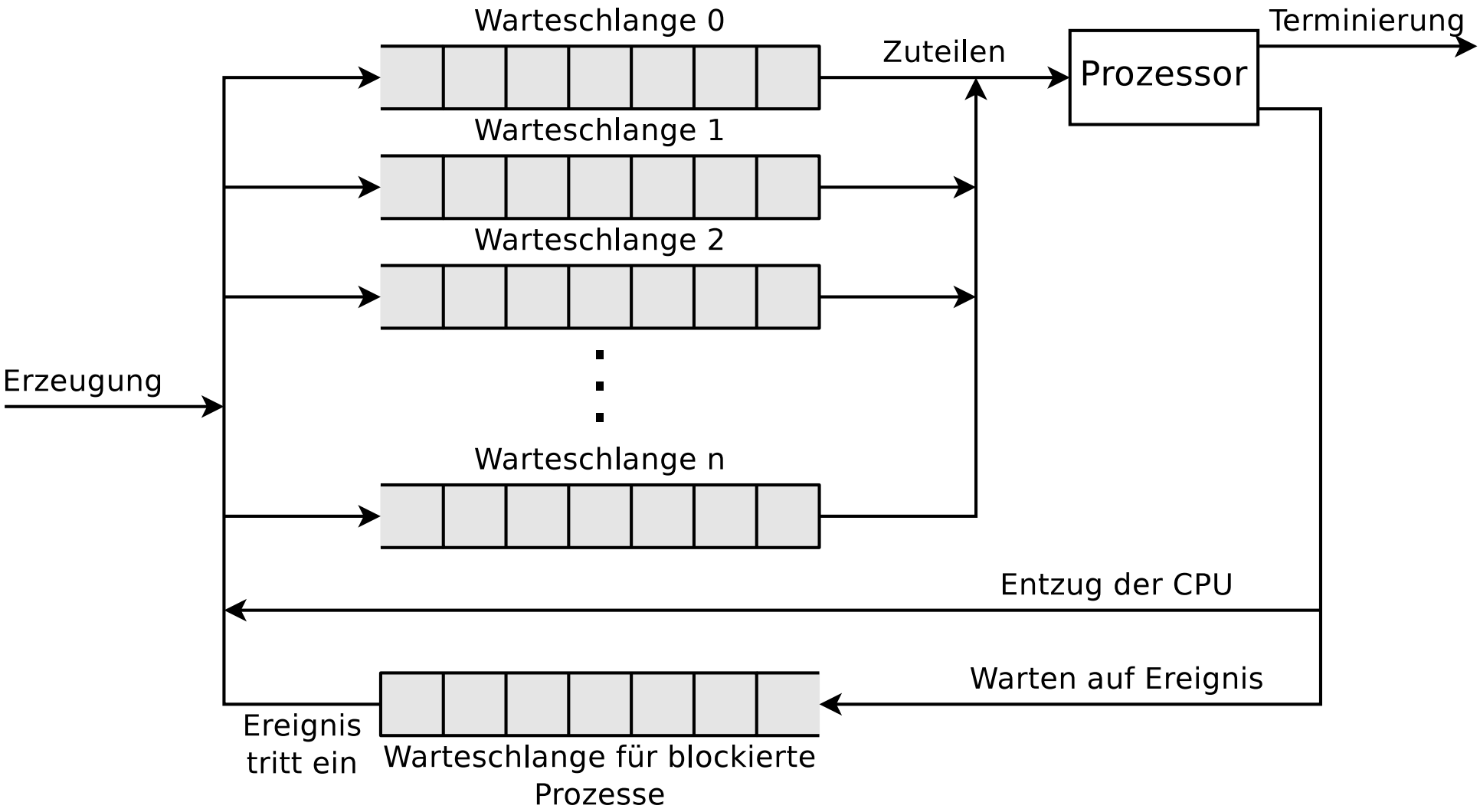
Scheduling-Verfahren

- Zahlreiche Scheduling-Verfahren (Algorithmen) existieren
- Jedes Scheduling-Verfahren versucht unterschiedlich stark, die bekannten Scheduling-Kriterien und -Grundsätze einzuhalten
- Bekannte Scheduling-Verfahren:
  - **Prioritätengesteuertes Scheduling**
  - **First Come First Served** (FCFS) bzw. **First In First Out** (FIFO)
  - **Last Come First Served** (LCFS)
  - **Round Robin** (RR) mit Zeitquantum
  - **Shortest Job First** (SJF) und **Longest Job First** (LJF)
  - **Shortest Remaining Time First** (SRTF)
  - **Longest Remaining Time First** (LRTF)
  - **Highest Response Ratio Next** (HRRN)
  - **Earliest Deadline First** (EDF)
  - **Fair-Share-Scheduling**
  - **Statisches Multilevel-Scheduling**
  - **Multilevel-Feedback-Scheduling**

Prioritätengesteuertes Scheduling

- Prozesse werden nach ihrer Priorität (= Wichtigkeit bzw. Dringlichkeit) abgearbeitet
- Es wird immer dem Prozess im Zustand `bereit` die CPU zugewiesen, der die höchste Priorität hat
  - Die Priorität kann von verschiedenen Kriterien abhängen, z.B. benötigte Ressourcen, Rang des Benutzers, geforderte Echtzeitkriterien, usw.
- Kann präemptiv (verdrängend) und nicht-präemptiv (nicht-verdrängend) sein
- Die Prioritätenvergabe kann statisch oder dynamisch sein
  - Statische Prioritäten ändern sich während der gesamten Lebensdauer eines Prozesses nicht und werden häufig in Echtzeitsystemen verwendet
  - Dynamische Prioritäten werden von Zeit zu Zeit angepasst
    $\Longrightarrow$ Multilevel-Feedback Scheduling (siehe Folie 44)
- Gefahr beim (statischen) prioritätengesteuertem Scheduling: Prozesse mit niedriger Priorität können verhungern ($\Longrightarrow$ **nicht fair**)
- Prioritätengesteuertes Scheduling eignet sich für interaktive Systeme

Prioritätengesteuertes Scheduling



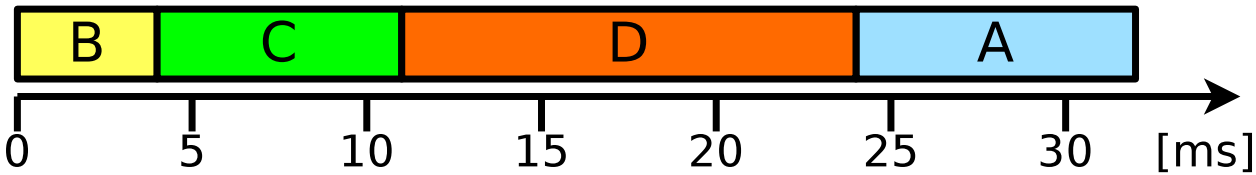
Quelle: William Stallings. Betriebssysteme. Pearson Studium. 2003

Beispiel zum Prioritätengesteuerten Scheduling

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand `bereit`

Prozess	CPU-Laufzeit	Priorität
A	8 ms	3
B	4 ms	15
C	7 ms	8
D	13 ms	4

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	32	4	11	24

$\frac{32+4+11+24}{4} = 17,75$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	24	0	4	11

$\frac{24+0+4+11}{4} = 9,75$ ms

First Come First Served (FCFS)

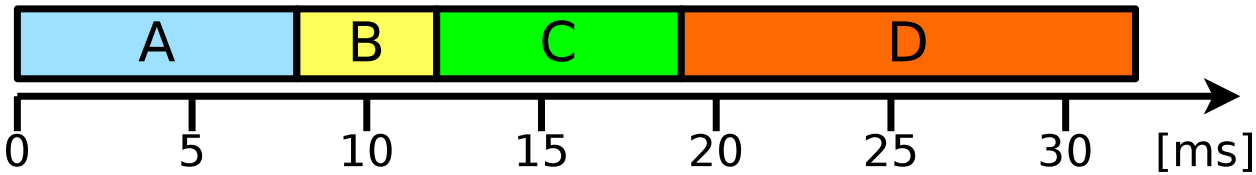
- Funktioniert nach dem Prinzip **First In First Out** (FIFO)
- Die Prozesse bekommen die CPU entsprechend ihrer Ankunftsreihenfolge zugewiesen
- Dieses Scheduling-Verfahren ist vergleichbar mit einer Warteschlange von Kunden in einem Geschäft
- Laufende Prozesse werden nicht unterbrochen
  - Es handelt sich um **nicht-präemptives** (nicht-verdrängendes) Scheduling
- FCFS ist **fair**
  - Alle Prozesse werden berücksichtigt
- Die **mittlere Wartezeit kann unter Umständen sehr hoch sein**
  - Prozesse mit kurzer Abarbeitungszeit müssen eventuell lange warten, wenn vor ihren Prozesse mit langer Abarbeitungszeit eingetroffen sind
- FCFS/FIFO eignet sich für Stapelverarbeitung ($\Longrightarrow$ Foliensatz 1)

Beispiel zu First Come First Served

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden

Prozess	CPU-Laufzeit	Ankunftszeit
A	8 ms	0 ms
B	4 ms	1 ms
C	7 ms	3 ms
D	13 ms	5 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	8	11	16	27

$\frac{8+11+16+27}{4} = 15,5$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	0	7	9	14

$\frac{0+7+9+14}{4} = 7,5$ ms

Last Come First Served (LCFS)

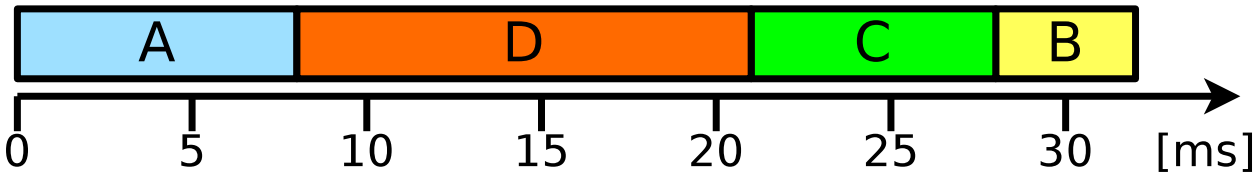
- Funktioniert nach dem Prinzip **Last In First Out** (FIFO)
- Die Prozesse werden in der umgekehrten Ankunftsreihenfolge bedient
  - Das Konzept entspricht einem Stack bzw. Kellerspeicher
- Laufende Prozesse werden **nicht unterbrochen**
  - Die Prozesse behalten den Zugriff auf die CPU bis zu Beendigung oder freiwilligen Abgabe
- LCFS ist **nicht fair**
  - Beim kontinuierlichen Eintreffen neuer Prozesse werden die alten Prozesse nicht berücksichtigt und können dadurch **verhungern**
- LCFS/LIFO eignet sich für Stapelverarbeitung ($\Longrightarrow$ Foliensatz 1)
  - Wird in der reinen Form selten verwendet

Beispiel zu Last Come First Served

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden

Prozess	CPU-Laufzeit	Ankunftszeit
A	8 ms	0 ms
B	4 ms	1 ms
C	7 ms	3 ms
D	13 ms	5 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	8	31	25	16

$\frac{8+31+25+16}{4} = 20$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	0	27	18	3

$\frac{0+27+18+3}{4} = 12$ ms

Last Come First Served – Präemptive Variante (LCFS-PR)

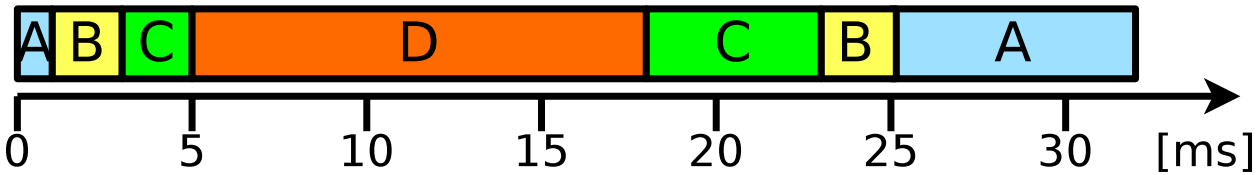
- Ein neuer Prozess in der Bereitliste verdrängt den aktuell laufenden Prozesse von der CPU
  - Verdrängte Prozesse werden an das Ende der Warteschlange eingereiht
  - Werden keine neuen Prozesse gestartet, findet auch keine Verdrängung statt
- **Bevorzugt Prozesse mit kurzer Ausführungszeit**
  - Ein kurzer Prozess hat die Chance, noch vor dem Eintreffen eines neuen Prozesse fertig zu sein
  - Lange Prozesse werden u.U. mehrfach verdrängt und dadurch stark verzögert
- LCFS-PR ist **nicht fair**
  - Es besteht die Gefahr, dass lange Prozesse nie Zugriff auf die CPU erhalten und **verhungern**
- Wird in der reinen Form selten verwendet

Beispiel zu Last Come First Served – Präemptive Variante

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden

Prozess	CPU-Laufzeit	Ankunftszeit
A	8 ms	0 ms
B	4 ms	1 ms
C	7 ms	3 ms
D	13 ms	5 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	32	24	20	13

$\frac{32+24+20+13}{4} = 22,25$ ms

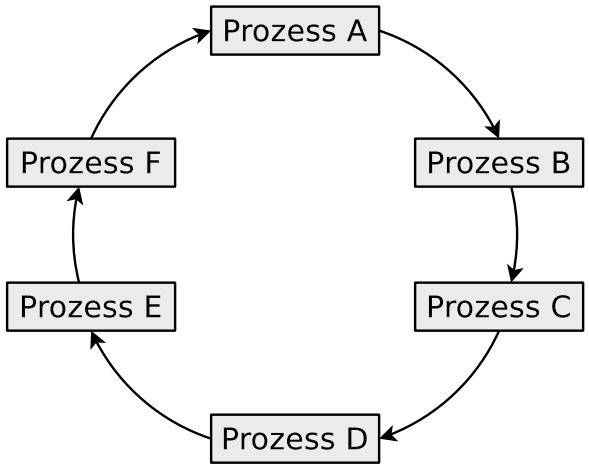
- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	24	20	13	0

$\frac{24+20+13+0}{4} = 14,25$ ms

Round Robin (RR) – Zeitscheibenverfahren (1/2)

- Es werden Zeitscheiben (*Time Slices*) mit einer festen Dauer festgelegt
- Die Prozesse werden in einer zyklischen Warteschlange nach dem FIFO-Prinzip eingereiht
  - Der erste Prozess der Warteschlange erhält für die Dauer einer Zeitscheibe Zugriff auf die CPU
  - Nach dem Ablauf der Zeitscheibe wird diesem der Zugriff auf die CPU wieder entzogen und er wird am Ende der Warteschlange eingereiht
  - Wird ein Prozess erfolgreich beendet, wird er aus der Warteschlange entfernt
    - Neue Prozesse werden am Ende der Warteschlange eingereiht



- Die Zugriffszeit auf die CPU wird **fair** auf die Prozesse aufgeteilt
- RR mit Zeitscheibengröße ∞ verhält sich wie FCFS

Round Robin (RR) – Zeitscheibenverfahren (2/2)

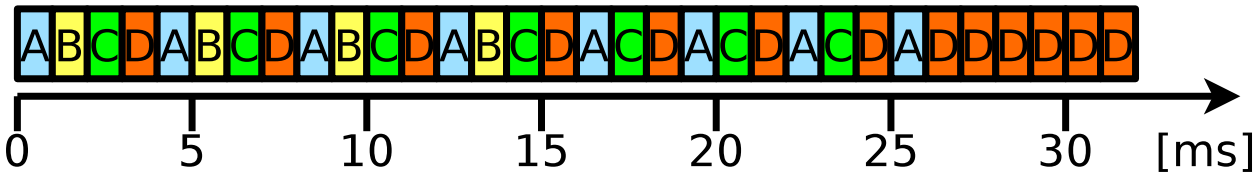
- Je länger die Bearbeitungsdauer eines Prozesses ist, desto mehr Runden sind für seine vollständige Ausführung nötig
- Die Größe der Zeitschlitze ist wichtig für die Systemgeschwindigkeit
  - Je kürzer sie sind, desto mehr Prozesswechsel müssen stattfinden
    $\Longrightarrow$ Hoher Overhead
  - Je länger sie sind, desto mehr geht die Gleichzeitigkeit verloren
    $\Longrightarrow$ Das System hängt/*ruckelt*
- Die Größe der Zeitschlitze liegt üblicherweise im ein- oder zweistelligen Millisekundenbereich
- **Bevorzugt Prozesse, die eine kurze Abarbeitungszeit haben**
- **Präemptives (verdrängendes) Scheduling-Verfahren**
- Round Robin Scheduling eignet sich für interaktive Systeme

Beispiel zu Round Robin

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand `bereit`
- Zeitquantum $q = 1$ ms

Prozess	CPU-Laufzeit
A	8 ms
B	4 ms
C	7 ms
D	13 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	26	14	24	32

$\frac{26+14+24+32}{4} = 24$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	18	10	17	19

$\frac{18+10+17+19}{4} = 16$ ms

Shortest Job First (SJF) / Shortest Process Next (SPN)

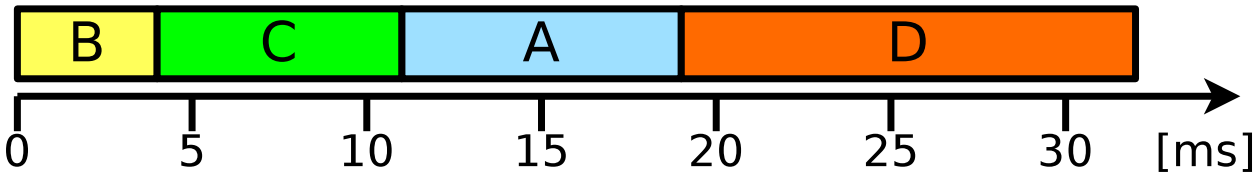
- Der Prozess mit der kürzesten Abarbeitungszeit erhält als erster Zugriff auf die CPU
- **Nicht-präemptives (nicht-verdrängendes) Scheduling**
- Hauptproblem:
  - Für jeden Prozess muss bekannt sein, wie lange er bis zu seiner Terminierung braucht, also wie lange seine Abarbeitungszeit ist
    - Ist in der Realität praktisch nie der Fall ($\Longrightarrow$ **unrealistisch**)
- Lösung:
  - Die Abarbeitungszeit der Prozesse wird abgeschätzt, indem die Abarbeitungszeit vorheriger Prozesse erfasst und analysiert wird
- SJF ist **nicht fair**
  - **Prozesse mit kurzer Abarbeitungszeit werden bevorzugt**
  - Prozesse mit langer Abarbeitungszeit erhalten eventuell erst nach sehr langer Wartezeit oder **verhungern**
- Wenn die Abarbeitungszeit der Prozesse abgeschätzt werden kann, eignet sich SJF für Stapelverarbeitung ($\Longrightarrow$ Foliensatz 1)

Beispiel zu Shortest Job First

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand `bereit`

Prozess	**CPU-Laufzeit**
A	8 ms
B	4 ms
C	7 ms
D	13 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	**A**	**B**	**C**	**D**
Laufzeit	19	4	11	32

$\frac{19+4+11+32}{4} = 16,5$ ms

- Wartezeit der Prozesse

Prozess	**A**	**B**	**C**	**D**
Wartezeit	11	0	4	19

$\frac{11+0+4+19}{4} = 8,5$ ms

Shortest Remaining Time First (SRTF)

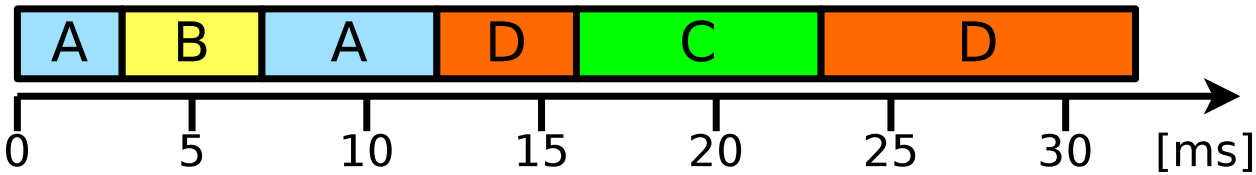
- **Präemptives** SJF heißt **Shortest Remaining Time First** (SRTF)
- Trifft ein neuer Prozess ein, wird die Restlaufzeit des aktuell rechnenden Prozesses mit jedem Prozess in der Liste der wartenden Prozesse verglichen
  - Hat der aktuell rechnende Prozesses die kürzeste Restlaufzeit, darf er weiter rechnen
  - Haben ein oder mehr Prozesse in der Liste der wartenden Prozesse eine kürzere Abarbeitungszeit bzw. Restlaufzeit, erhält der Prozess mit der kürzesten Restlaufzeit Zugriff auf die CPU
- Hauptproblem: Die Restlaufzeit muss bekannt sein ($\Longrightarrow$ **unrealistisch**)
- Solange kein neuer Prozess eintrifft, wird kein rechnender Prozess unterbrochen
  - Die Prozesse in der Liste der wartenden Prozesse werden nur dann mit dem aktuell rechnenden Prozess verglichen, wenn ein neuer Prozess eintrifft!
- Prozesse mit langer Laufzeit können **verhungern** ($\Longrightarrow$ **nicht fair**)

Beispiel zu Shortest Remaining Time First

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden

Prozess	CPU-Laufzeit	Ankunftszeit
A	8 ms	0 ms
B	4 ms	3 ms
C	7 ms	16 ms
D	13 ms	11 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	12	4	7	21

$\frac{12+4+7+21}{4} = 11$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	4	0	0	8

$\frac{4+0+0+8}{4} = 3$ ms

Longest Job First (LJF)

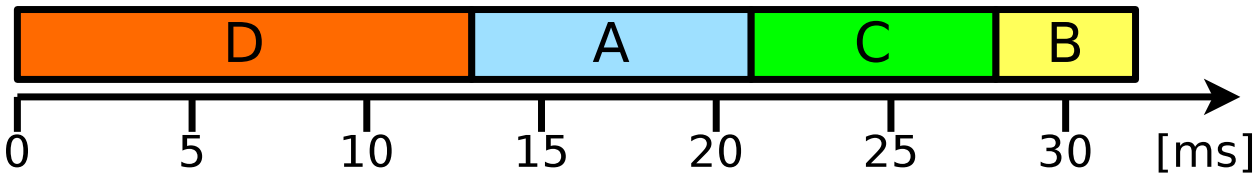
- Der Prozess mit der längsten Abarbeitungszeit erhält als erster Zugriff auf die CPU
- **Nicht-präemptives (nicht-verdrängendes) Scheduling**
- Hauptproblem: Genau wie bei SJF muss für jedem Prozess bekannt sein, wie lange seine Abarbeitungszeit ist
  - Das ist in der Realität nur selten der Fall ($\Longrightarrow$ **unrealistisch**)
- LJF ist **nicht fair**
  - **Prozesse mit langer Abarbeitungszeit werden bevorzugt**
  - Prozesse mit kurzer Abarbeitungszeit erhalten eventuell erst nach sehr langer Wartezeit Zugriff auf die CPU oder **verhungern**
- Wenn die Abarbeitungszeit der Prozesse abgeschätzt werden kann, eignet sich LJF für Stapelverarbeitung ($\Longrightarrow$ Foliensatz 1)

Beispiel zu Longest Job First

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand `bereit`

Prozess	**CPU-Laufzeit**
A	8 ms
B	4 ms
C	7 ms
D	13 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	**A**	**B**	**C**	**D**
Laufzeit	21	32	28	13

$\frac{21+32+28+13}{4} = 23,5$ ms

- Wartezeit der Prozesse

Prozess	**A**	**B**	**C**	**D**
Wartezeit	13	28	21	0

$\frac{13+28+21+0}{4} = 15,5$ ms

Longest Remaining Time First (LRTF)

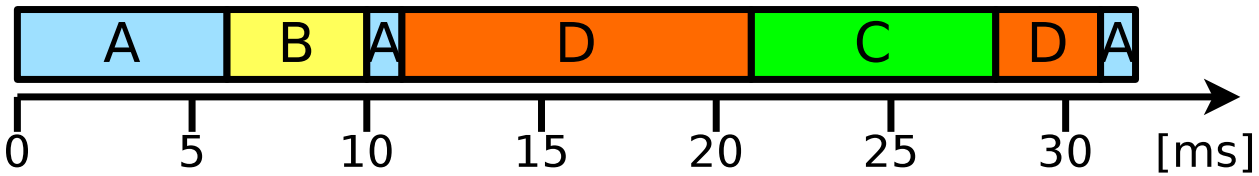
- **Präemptives** LJF heißt **Longest Remaining Time First** (LRTF)
- Trifft ein neuer Prozess ein, wird die Restlaufzeit des aktuell rechnenden Prozesses mit jedem Prozess in der Liste der wartenden Prozesse verglichen
  - Hat der aktuell rechnende Prozesses die längste Restlaufzeit, darf er weiter rechnen
  - Haben ein oder mehr Prozesse in der Liste der wartenden Prozesse eine längere Abarbeitungszeit bzw. Restlaufzeit, erhält der Prozess mit der längsten Restlaufzeit Zugriff auf die CPU
- Hauptproblem: Die Restlaufzeit muss bekannt sein ($\Longrightarrow$ **unrealistisch**)
- Solange kein neuer Prozess eintrifft, wird kein rechnender Prozess unterbrochen
  - Die Prozesse in der Liste der wartenden Prozesse werden nur dann mit dem aktuell rechnenden Prozess verglichen, wenn ein neuer Prozess eintrifft!
- Prozesse mit kurzer Laufzeit können verhungern ($\Longrightarrow$ **nicht fair**)

Beispiel zu Longest Remaining Time First

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden

Prozess	CPU-Laufzeit	Ankunftszeit
A	8 ms	0 ms
B	4 ms	6 ms
C	7 ms	21 ms
D	13 ms	11 ms

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	32	4	7	20

$\frac{32+4+7+20}{4} = 15,75$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	24	0	0	7

$\frac{24+0+0+7}{4} = 7,75$ ms

Highest Response Ratio Next (HRRN)

- Faire Variante von SJF/SRTF/LJF/LRTF
  - Berücksichtigt das Alter der Prozesse um **Verhungern zu vermeiden**
- **Antwortquotient** (Response Ratio) wird für jeden Prozess berechnet

$$\text{Antwortquotient} = \frac{\text{geschätzte Rechenzeit} + \text{Wartezeit}}{\text{geschätzte Rechenzeit}}$$

- Wert des Antwortquotienten bei der Erzeugung eines Prozesses: 1.0
  - Der Wert steigt bei kurzen Prozessen schnell an
  - Ziel: Der Antwortquotient soll für alle Prozesse möglichst gering sein
    - Dann arbeitet das Scheduling effizient
- Nach Beendigung oder bei Blockade eines Prozesses, bekommt der Prozess mit dem höchsten Antwortquotient die CPU zugewiesen
- Wie bei SJF/SRTF/LJF/LRTF müssen die Laufzeiten der Prozesse durch statistische Erfassung aus der Vergangenheit abgeschätzt werden
- Es ist unmöglich, dass Prozesse verhungern $\Longrightarrow$ HRRN ist **fair**

Earliest Deadline First (EDF)

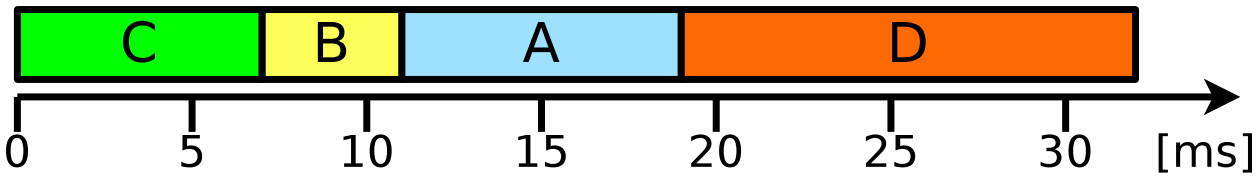
- Ziel: Prozesse sollen nach Möglichkeit ihre Termine zur Fertigstellung (*Deadlines*) einhalten
- Prozesse im Zustand `bereit` werden aufsteigend **nach ihrer Deadline geordnet**
  - Der Prozess, dessen Deadline am nächsten ist, bekommt die CPU zugewiesen
- Eine Überprüfung und gegebenenfalls Neuorganisation der Warteschlange findet statt, wenn...
  - ein neuer Prozess in den Zustand `bereit` wechselt
  - oder ein aktiver Prozess terminiert
- Kann als **präemptives und nicht-präemptives Scheduling** realisiert werden
  - Präemptives EDF eignet sich für Echtzeitbetriebssysteme
  - Nicht-präemptives EDF eignet sich für Stapelverarbeitung

Beispiel zu Earliest Deadline First

- Auf einem Einprozessorrechner sollen vier Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand `bereit`

Prozess	CPU-Laufzeit	Deadline
A	8 ms	25
B	4 ms	18
C	7 ms	9
D	13 ms	34

- Ausführungsreihenfolge der Prozesse als Gantt-Diagramm (Zeitleiste)



- Laufzeit der Prozesse

Prozess	A	B	C	D
Laufzeit	19	11	7	32

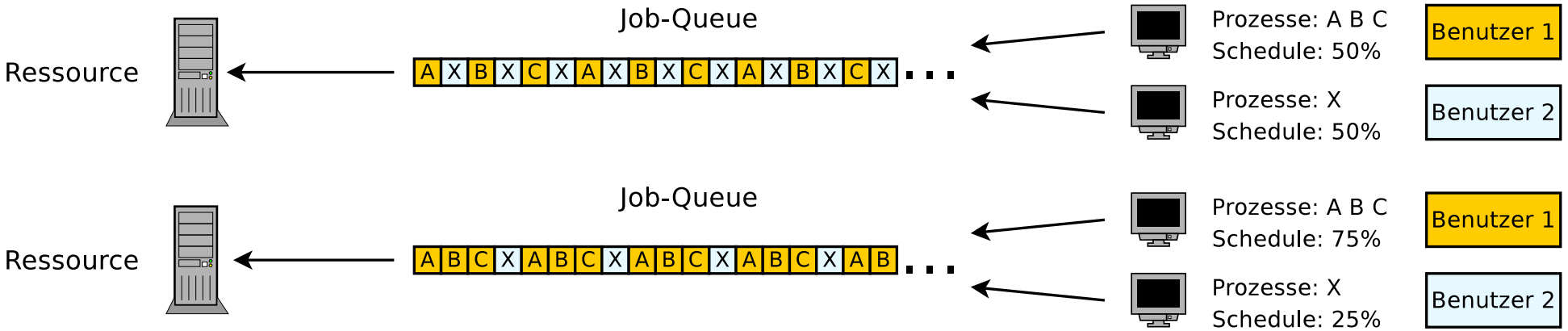
$\frac{19+11+7+32}{4} = 17,25$ ms

- Wartezeit der Prozesse

Prozess	A	B	C	D
Wartezeit	11	7	0	19

$\frac{11+7+0+19}{4} = 9,25$ ms

Fair-Share



- Bei **Fair-Share** werden Ressourcen zwischen Gruppen von Prozessen in einer fairen Art und Weise aufgeteilt
- Besonderheit:
  - Die Rechenzeit wird den Benutzern und nicht den Prozessen zugeteilt
  - Die Rechenzeit, die ein Benutzer erhält, ist unabhängig von der Anzahl seiner Prozesse
- Die Ressourcenanteile, die die Benutzer erhalten, heißen *Shares*

Fair-Share wird häufig in Cluster- und Grid-Systemen eingesetzt

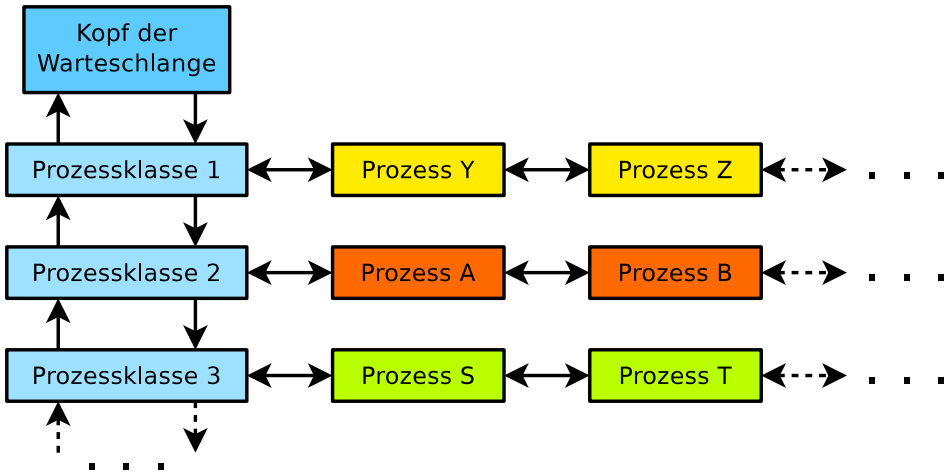
Fair-Share wird in Job-Schedulern und Meta-Schedulern (z.B. Oracle Grid Engine) zur Verteilung der Aufträge auf Ressourcen in Grid-Standorten und zwischen den Standorten in Grids eingesetzt

Multilevel-Scheduling / Mehrebenen-Scheduling

- Bei jeder Scheduling-Strategie müssen Kompromisse bzgl. der unterschiedlichen Scheduling-Kriterien gemacht werden
  - Vorgehen in der Praxis: Mehrere Scheduling-Strategien kombinieren
    $\Longrightarrow$ **Statisches oder dynamisches Multilevel-Scheduling**

Statisches Multilevel-Scheduling

- Die `bereit`-Liste wird in mehrere Teillisten unterteilt
  - Für jede Teilliste kann eine andere Scheduling-Strategie verwendet werden
- Die Teillisten haben unterschiedliche Prioritäten oder Zeitmultiplexe (z.B. 80%:20% oder 60%:30%:10%)
  - Geeignet, um zeitkritische von zeitunkritischen Prozessen zu trennen



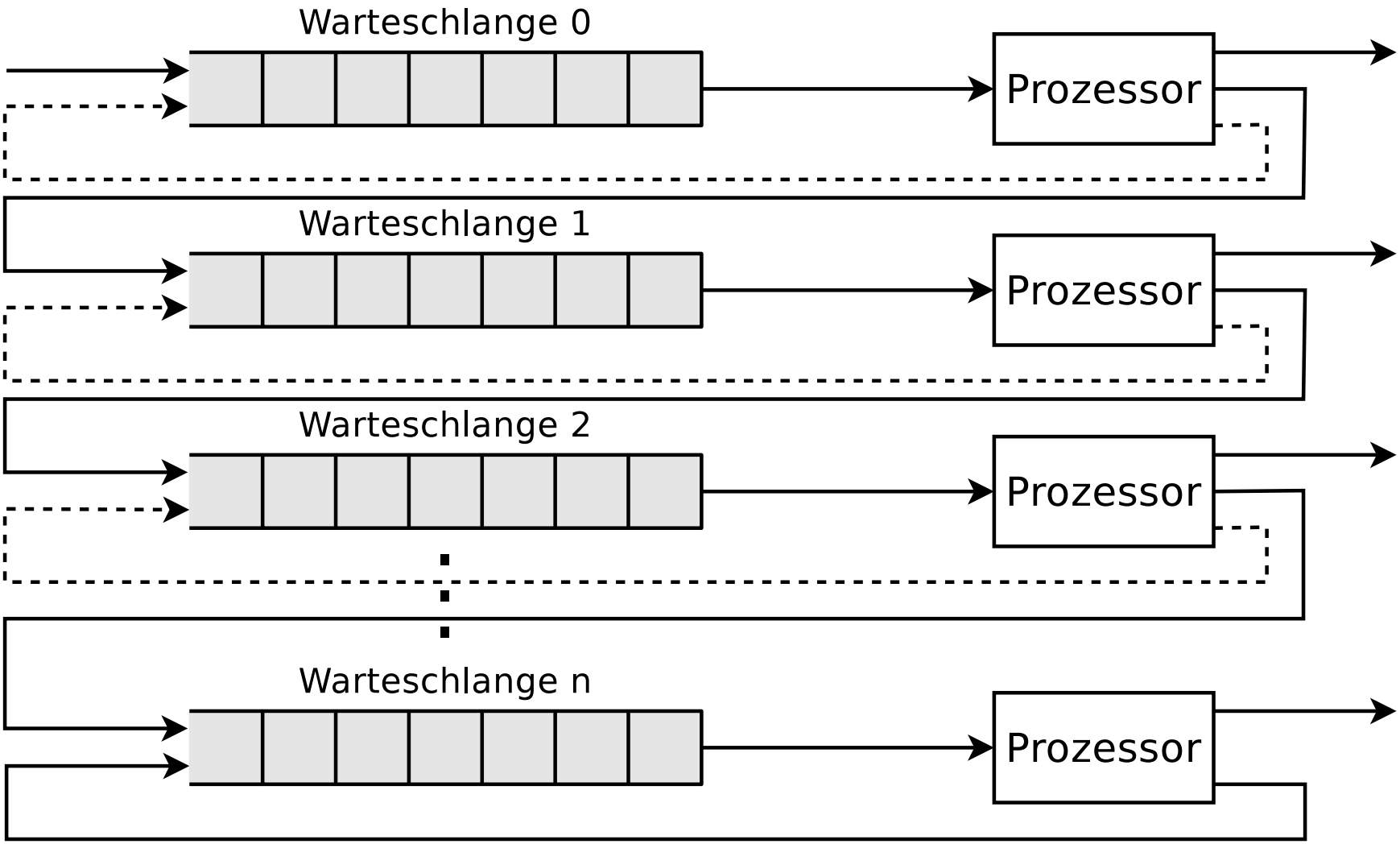
- Beispiel für eine Unterteilung der Prozesse in verschiedene Prozessklassen (Teillisten) mit verschiedenen Scheduling-Strategien:

Priorität	Prozessklasse	Scheduling-Strategie
3	Echtzeitprozesse (zeitkritisch)	Prioritätengesteuert
2	Interaktive Prozesse	Round Robin
1	Rechenintensive Stapelprozesse	First Come First Served

Multilevel-Feedback-Scheduling (1/2)

- Es ist **unmöglich, die Rechenzeit verlässlich im voraus zu kalkulieren**
  - Lösung: Prozesse, die schon länger aktiv sind, werden **bestraft**
- **Multilevel-Feedback-Scheduling** arbeitet wie Multilevel-Scheduling mit mehreren Warteschlangen
  - Jede Warteschlange hat eine andere Priorität oder Zeitmultiplex
- Jeder neue Prozess kommt in die oberste Warteschlange
  - Damit hat er die höchste Priorität
- Innerhalb jeder Warteschlange wird Round Robin eingesetzt
  - Gibt ein Prozess die CPU freiwillig wieder ab, wird er wieder in die selbe Warteschlange eingereiht
  - Hat ein Prozess seine volle Zeitscheibe genutzt, kommt er in die nächst tiefere Warteschlange mit einer niedrigeren Priorität
    - Die Prioritäten werden bei diesem Verfahren also **dynamisch** vergeben
- Multilevel-Feedback-Scheduling ist **unterbrechendes Scheduling**

Warteschlangen beim Multilevel-Feedback-Scheduling



Quelle: William Stallings. Betriebssysteme. Pearson Studium. 2003

Multilevel-Feedback-Scheduling (2/2)

- Vorteil:
  - Es sind **keine komplizierten Abschätzungen** nötig!
    - Neue Prozesse werden schnell in eine Prioritätsklasse eingeordnet
- **Bevorzugt neue Prozesse gegenüber älteren** (länger laufenden) Prozessen
- Prozesse mit vielen Ein-/Ausgabeoperationen werden bevorzugt, weil sie nach einer freiwilligen Abgabe der CPU wieder in die ursprüngliche Warteliste eingeordnet werden
  - Dadurch behalten Sie ihre Priorität
- Ältere, länger laufende Prozesse werden verzögert

Klassische und moderne Scheduling-Verfahren

	Scheduling NP	Scheduling P	Fair	CPU-Laufzeit muss bekannt sein	Berücksichtigt Prioritäten
Prioritätengesteuertes Scheduling	X	X	nein	nein	ja
First Come First Served	X		ja	nein	nein
Last Come First Served	X	X	nein	nein	nein
Round Robin		X	ja	nein	nein
Shortest Job First	X		nein	ja	nein
Longest Job First	X		nein	ja	nein
Shortest Remaining Time First		X	nein	ja	nein
Longest Remaining Time First		X	nein	ja	nein
Highest Response Ratio Next	X		ja	ja	nein
Earliest Deadline First	X	X	ja	nein	nein
Fair-Share		X	ja	nein	nein
Statisches Multilevel-Scheduling		X	nein	nein	ja (statisch)
Multilevel-Feedback-Scheduling		X	ja	nein	ja (dynamisch)

- NP = Nicht-präemptives Scheduling, P = Präemptives Scheduling
- Ein Schedulingverfahren ist „fair“, wenn jeder Prozess irgendwann Zugriff auf die CPU erhält
- Es ist unmöglich, die Rechenzeit verlässlich im voraus zu kalkulieren

Einfaches Beispiel zum Scheduling

Prozess	CPU-Laufzeit	Priorität
A	5 ms	15
B	10 ms	5
C	3 ms	4
D	6 ms	12
E	8 ms	7

- Auf einem Einprozessorrechner sollen 5 Prozesse verarbeitet werden
- Alle Prozesse sind zum Zeitpunkt 0 im Zustand `bereit`
- Hohe Prioritäten sind durch hohe Zahlen gekennzeichnet

- Skizzieren Sie die Ausführungsreihenfolge der Prozesse mit einem Gantt-Diagramm (Zeitleiste) für **Round Robin** (Zeitquantum $q = 1\,\text{ms}$), **FCFS**, **SJF**, **LJF** und **Prioritätengesteuertes Scheduling**
- Berechnen Sie die mittleren Laufzeiten und Wartezeiten der Prozesse
  - Laufzeit = Zeit von der Ankunft bis zur Terminierung
  - Wartezeit = Laufzeit - Rechenzeit

Im Zweifelsfall immer FIFO anwenden

Das heißt im Detail: Wenn das Entscheidungskriterium des verwendeten Scheduling-Verfahrens auf mehrere Prozesse zutrifft, dann nehmen Sie den ältesten Prozess $\Longrightarrow$ FIFO

Einfaches Beispiel zum Scheduling

Prozess	CPU-Laufzeit	Priorität
A	5 ms	15
B	10 ms	5
C	3 ms	4
D	6 ms	12
E	8 ms	7

Laufzeit	A	B	C	D	E
RR					
FCFS					
SJF					
LJF					
PS*					

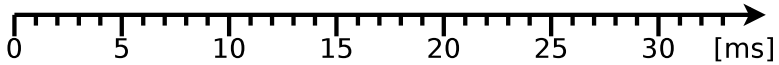
* Prioritätengesteuertes Scheduling

Round Robin (Zeitquantum = 1)

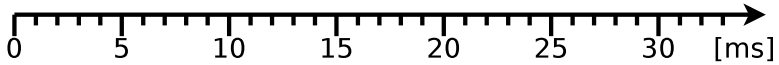
First Come First Served

Shortest Job First

Longest Job First



Prioritätengesteuertes Scheduling

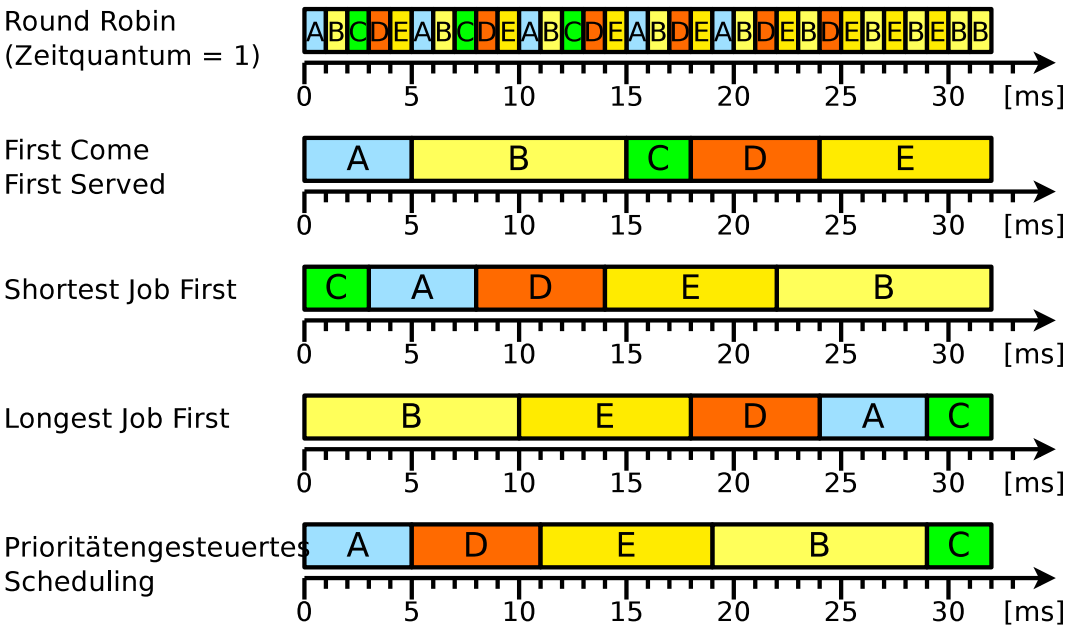


Wartezeit	A	B	C	D	E
RR					
FCFS					
SJF					
LJF					
PS*					

* Prioritätengesteuertes Scheduling

- Die Wartezeit ist die Zeit in der `bereit`-Liste

Lösung – Gantt-Diagramm + Laufzeit (Turnaround Time)



Laufzeit	A	B	C	D	E
RR	20	32	13	25	30
FCFS	5	15	18	24	32
SJF	8	32	3	14	22
LJF	29	10	32	24	18
PS*	5	29	32	11	19

* Prioritätengesteuertes Scheduling

$$\text{RR} \quad \frac{20+32+13+25+30}{5} = 24 \text{ ms}$$

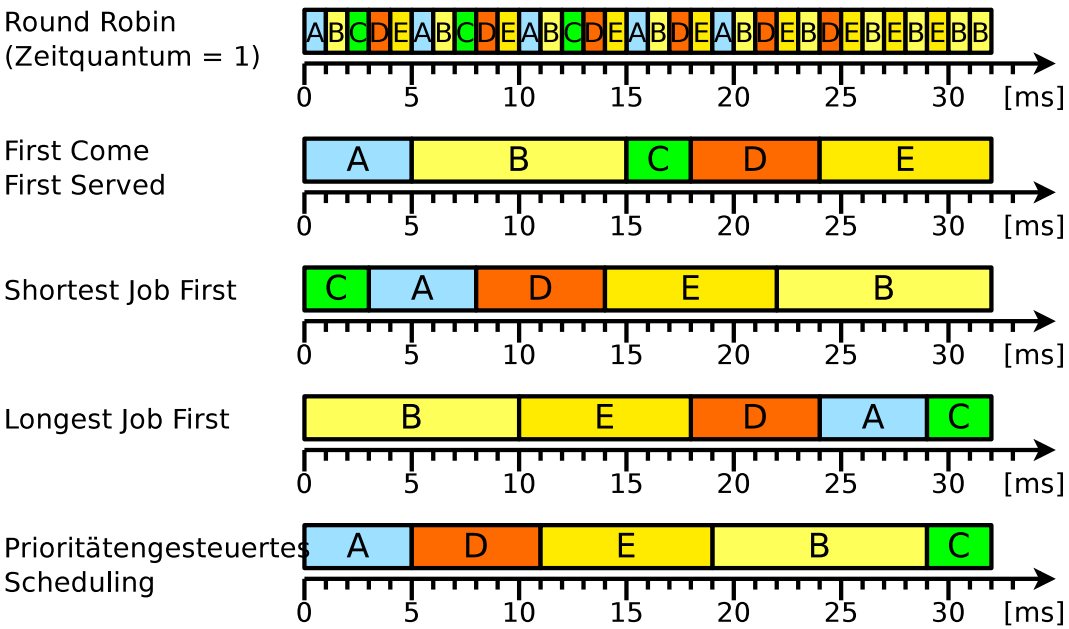
$$\text{FCFS} \quad \frac{5+15+18+24+32}{5} = 18,8 \text{ ms}$$

$$\text{SJF} \quad \frac{8+32+3+14+22}{5} = 15,8 \text{ ms}$$

$$\text{LJF} \quad \frac{29+10+32+24+18}{5} = 22,6 \text{ ms}$$

$$\text{PS} \quad \frac{5+29+32+11+19}{5} = 19,2 \text{ ms}$$

Lösung – Gantt-Diagramm + Wartezeit



Wartezeit	A	B	C	D	E
RR	15	22	10	19	22
FCFS	0	5	15	18	24
SJF	3	22	0	8	14
LJF	24	0	29	18	10
PS*	0	19	29	5	11

* Prioritätengesteuertes Scheduling

$$\text{RR} \quad \frac{15+22+10+19+22}{5} = 17,6 \text{ ms}$$

$$\text{FCFS} \quad \frac{0+5+15+18+24}{5} = 12,4 \text{ ms}$$

$$\text{SJF} \quad \frac{3+22+0+8+14}{5} = 9,4 \text{ ms}$$

$$\text{LJF} \quad \frac{24+0+29+18+10}{5} = 16,2 \text{ ms}$$

$$\text{PS} \quad \frac{0+19+29+5+11}{5} = 12,8 \text{ ms}$$

Fazit (1/2)

- Von den untersuchten Scheduling-Verfahren hat/haben...
  - **SJF** die beste mittlere Laufzeit und kürzeste mittlere Wartezeit
  - **RR** und **LJF** die schlechteste mittlere Laufzeit und mittlere Wartezeit

Prozess	**CPU-Laufzeit**
A	24 ms
B	2 ms

- Grund:
  - Läuft ein Prozess mit kurzer Laufzeit vor einem Prozess mit langer Laufzeit, **verschlechtern** sich Laufzeit und Wartezeit des langen Prozesses **wenig**
  - Läuft ein Prozess mit langer Laufzeit vor einem Prozess mit kurzer Laufzeit, **verschlechtern** sich Laufzeit und Wartezeit des kurzen Prozesses **stark**

Reihenfolge	**Laufzeit**		**Durchschnittliche Laufzeit**	**Wartezeit**		**Durchschnittliche Wartezeit**
	A	**B**		**A**	**B**	
P_A, P_B	24 ms	26 ms	$\frac{24+26}{2} = 25$ ms	0 ms	24 ms	$\frac{0+24}{2} = 12$ ms
P_B, P_A	26 ms	2 ms	$\frac{2+26}{2} = 14$ ms	2 ms	0 ms	$\frac{0+2}{2} = 1$ ms

Fazit (2/2)

- **RR** verursacht häufige Prozesswechsel
  - Der dadurch entstehende **Overhead** wirkt sich zusätzlich negativ auf die Systemleistung aus
- Die Größe des Overhead hängt von der Größe der Zeitscheiben ab
  - **Kurze Zeitscheiben** $\Longrightarrow$ hoher Overhead
  - **Lange Zeitscheiben** $\Longrightarrow$ Antwortzeiten sind eventuell zu lang für interaktive Prozesse

Ein weiteres Scheduling-Beispiel

(Klausuraufgabe SS2009)

Prozess	CPU-Laufzeit	Ankunftszeit
A	3 ms	0 ms
B	2 ms	3 ms
C	5 ms	4 ms
D	3 ms	5 ms
E	2 ms	9 ms
F	5 ms	10 ms

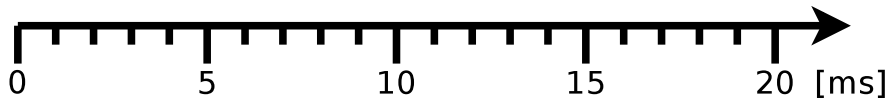
- Auf einem Einprozessorrechner sollen folgende Prozesse mit unterschiedlichen Ankunftszeiten verarbeitet werden
- Skizzieren Sie die Ausführungsreihenfolge der Prozesse mit einem Gantt-Diagramm (Zeitleiste) für **Round Robin** (Zeitquantum $q = 1\,\text{ms}$), **Longest Remaining Time First** (LRTF) und **Shortest Remaining Time First** (SRTF)
- **ACHTUNG!!!** Für Round Robin ist bei allen Prozessen die Ankunftszeit 0 ms. Diese Ausnahme gibt nur für Round Robin! Bei den anderen Scheduling-Verfahren sind die in der Tabelle angegebenen Ankunftszeiten zu berücksichtigen
- Berechnen Sie die mittleren Laufzeiten und Wartezeiten der Prozesse

Scheduling-Beispiel

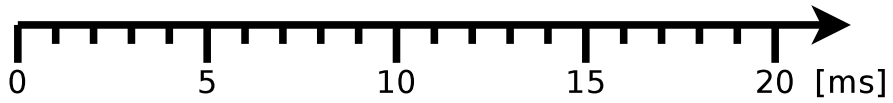
(Klausuraufgabe SS2009)

Prozess	CPU-Laufzeit	Ankunftszeit
A	3 ms	0 ms
B	2 ms	3 ms
C	5 ms	4 ms
D	3 ms	5 ms
E	2 ms	9 ms
F	5 ms	10 ms

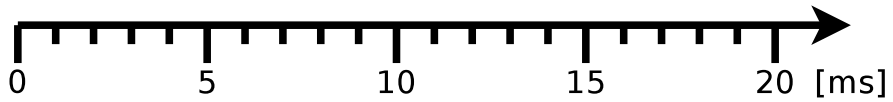
RR



LRTF



SRTF

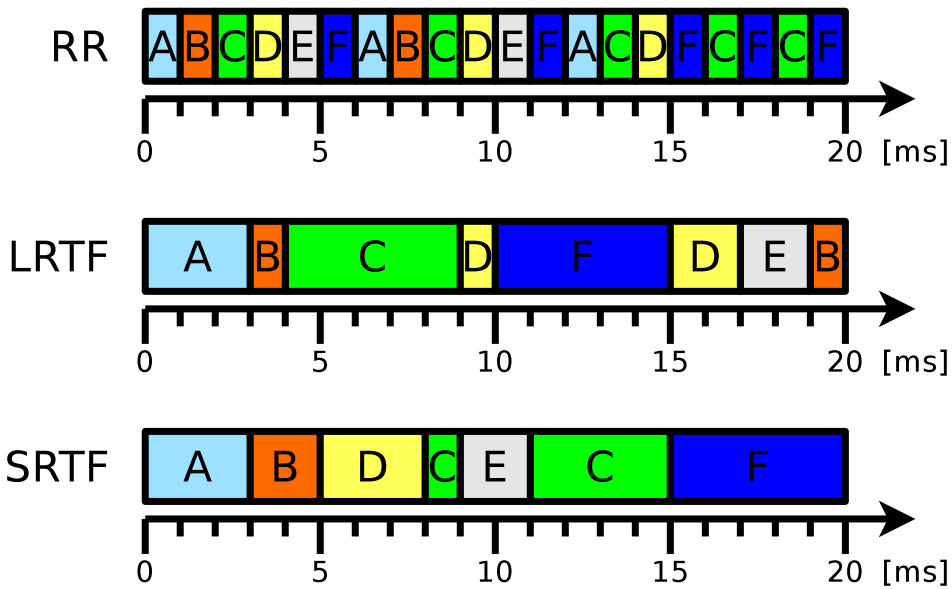


Laufzeit	A	B	C	D	E	F
RR						
SRTF						
LRTF						

Wartezeit	A	B	C	D	E	F
RR						
SRTF						
LRTF						

Scheduling-Beispiel (Lösung)

(Klausuraufgabe SS2009)



Laufzeit	A	B	C	D	E	F
RR	13	8	19	15	11	20
LRTF	3	17	5	12	10	5
SRTF	3	2	11	3	2	10

RR $\frac{13+8+19+15+11+20}{6} = 14,\overline{3}$ ms

LRTF $\frac{3+17+5+12+10+5}{6} = 8,\overline{6}$ ms

SRTF $\frac{3+2+11+3+2+10}{6} = 5,1\overline{6}$ ms

Wartezeit	A	B	C	D	E	F
RR	10	6	14	12	9	15
LRTF	0	15	0	9	8	0
SRTF	0	0	6	0	0	5

RR $\frac{10+6+14+12+9+15}{6} = 11$ ms

LRTF $\frac{0+15+0+9+8+0}{6} = 5,\overline{3}$ ms

SRTF $\frac{0+0+6+0+0+5}{6} = 1,8\overline{3}$ ms