Inhalt

Literatur

Andrew S. Tanenbaum,
Moderne Betriebssysteme,
2te Auflage, Pearson Studium, 2002.

Carsten Vogt,
Betriebssysteme,
Spektrum Akademischer Verlag, 2001.

Abraham Silberschatz, Peter B. Galvin,
Operating System Concepts (5th Edition),
John Wiley and Sons, 1999.

William Stallings,
Betriebssysteme - Prinzipien und Umsetzung (4te Auflage),
Prentice Hall, 2002.

Allgemein

Computersoftware gliedert sich in zwei Gruppen.

- **Systemprogramme** ermöglichen den Betrieb des Computers.
- **Anwenderprogramme** erfüllen die Anforderungen der Anwender.

Das **Betriebssystem** ist das wichtigste Systemprogramm.

Definition eines Betriebssystems nach DIN 44300

- Betriebssystem: Die Programme eines digitalen Rechnersystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.

Hardwaresicht

Man kann bei Rechnersystemen zwei Sichten einnehmen.

Hardwaresicht. Ein Rechnersystem besteht aus einer Menge kooperierender Hardwarekomponenten.

- Prozessor (CPU). Ausführung von Maschinenprogrammen
- Hauptspeicher. *Kurzfristige* Speicherung einer begrenzte Menge von Daten.
- Hintergrundspeicher (Festplatte, Diskette, CD, DVD). *Langfristige* Speicherung größerer Datenmengen.
- Eingabegeräte (Tastatur, Maus).
- Ausgabegeräte (Bildschirm, Drucker).
- Netzwerkkarte. Verbindung an ein Kommunikationsnetz.
- ...

Anwendersicht

Anwendersicht. Ein Rechnersystem stellt (benutzerfreundliche) Konzepte bereit, mit denen Daten und Informationen verarbeitet werden können.

- Dateisystem. Klar strukturiert mit Dienstprogrammen.
- Programmierumgebung. Schreiben und übersetzen von Programmen.
- Ein- und Ausgabedienste. Zugriff auf Daten, Internet Angebote, etc.
- Systemverwaltung.
- Multi-User-Fähigkeit. Unterstützung mehrere Benutzer.
- ...

Position des Betriebssystems

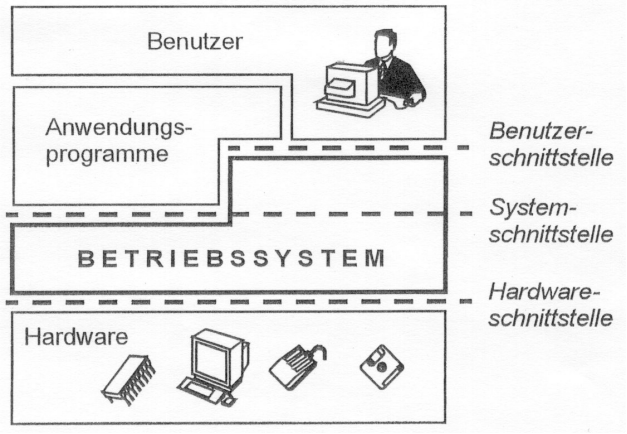


Abbildung: Position des Betriebssystems (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

Betriebssystem

Das Betriebssystem (engl. *operating system*) ist die Softwarekomponente, die die abstrakte Anwendersicht auf der Grundlage der realen Hardwaresicht umsetzt.

Das Betriebssystem

- macht die Hardware für den Anwender benutzbar.
- setzt auf der Hardware auf, steuert diese und bietet *nach oben* benutzer- und programmierfreundliche Dienste an.
- enthält interne Programme und Datenstrukturen.

Definitionen

Ressourcen (Betriebsmittel). Die Ressourcen (Betriebsmittel) eines Betriebssystems sind alle Hard- und Softwarekomponenten, die für die Programmausführung relevant sind. z.B. Prozessor, Hauptspeicher, I/O-Geräte, Hintergrundspeicher, etc.

Betriebssystem als Ressourcenverwalter. Ein Betriebssystem bezeichnet alle Programme eines Rechensystems, die die Ausführung der Benutzerprogramme, die Verteilung der Ressourcen auf die Benutzerprogramme und die Aufrechterhaltung der Betriebsart steuern und Überwachen.

Aufgaben eines Betriebssystems

Hauptaufgaben eines Betriebssystems

- Prozessverwaltung
- Speicherverwaltung
- Verwaltung des Dateisystems
- Geräteverwaltung

Prozessverwaltung

Prozessverwaltung (Ein Prozess oder auch Task ist ein in Ausführung befindliches Programm)

- Erzeugen und Löschen von Prozessen.
- Prozessorzuteilung (Scheduling).
- Prozesskommunikation.
- Synchronisation nebenläufiger Prozesse, die gemeinsame Daten benutzen.

Speicherverwaltung

Speicherverwaltung

- Zuteilung des verfügbaren physikalischen Speichers an Prozesse.
  - Segementierung (= Unterteilung des benutzten Speicheradressraums in einzelne Segmente).
  - ...
- Einbeziehen des Hintergrundspeichers (z.B. Festplatte).
  - Paging (= Bereitstellung von virtuellem Speicher).
  - Swapping (= Ein-/Auslagern von Prozessen).
  - ...

Verwaltung des Dateisystems

Verwaltung des Dateisystems

- Logische Sicht auf Speichereinheiten (Dateien).
  - Benutzer arbeitet mit Dateinamen. Wie und wo die Dateien gespeichert werden, ist ihm egal.
- Systemaufrufe für Dateioperationen.
  - Erzeugen, Löschen, Öffnen, Lesen, Schreiben, Kopieren, etc.
- Strukturierung mittels Verzeichnissen (engl. directories).
- Schutz von Dateien und Verzeichnissen vor unberechtigtem Zugriff

Geräteverwaltung

Geräteverwaltung

- Auswahl und Bereitstellung von I/O-Geräten.
- Anpassung an physikalische Eigenschaften der Geräte.
- Überwachung der Datenübertragung.

Weitere wichtige Konzepte

Fehlertoleranz.

- Graceful Degradation. Beim Ausfall einzelner Komponenten läuft das System mit vollem Funktionsumfang mit verminderter Leistung weiter.
- Fehlertoleranz wird durch Redundanz erkauft.

Realzeitbetrieb.

- Betriebssystem muss den Realzeit-kritischen Prozessen die Betriebsmittel so zuteilen, dass die angeforderten Zeitanforderungen eingehalten werden.
- Für zeitkritische Systeme. Messsysteme, Anlagensteuerungen, etc.

Benutzeroberflächen.

- Betriebssystem kann eine Benutzerschnittstelle für die eigene Bedienung enthalten.
- Betriebssystem kann Funktionen bereitstellen, mit denen aus Anwendungsprogrammen heraus auf die Benutzerschnittstelle zugegriffen werden kann.

Betriebsart

Die **Betriebsart** ist ein wichtiges Kennzeichen für die Leistungsfähigkeit und den Anwendungsbereich eines Betriebssystems.

Rechensysteme habe im Allgemeinem mehrere Aufträge gleichzeitig zu bearbeiten. Die Betriebsart bestimmt (im wesentlichen) den zeitlichen Ablauf der Ausführung dieser Aufträge.

Die Spanne möglicher Abläufe reicht von **streng sequentiell** (= hintereinander) bis **voll nebenläufig** (= gleichzeitig).

Betriebsarten

Einzelbenutzerbetrieb

> **Ein Benutzer** belegt das **gesamte Rechensystem** und erteilt Aufträge, die **streng sequentiell** abgearbeitet werden.

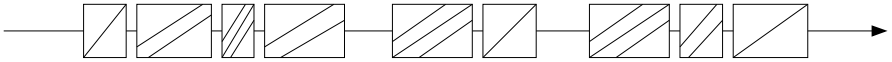
Batchbetrieb

> Ein Auftrag (engl. *job*) wird durch eine Reihe von **Steuerkommandos an das Betriebssystem**, formuliert in einer *job control language*, eingeleitet und abgeschlossen. Zwischen den Steuerkommandos befindet sich das eigentliche Programm.

Mehrprogrammbetrieb

> Mehrere Aufträge werden **nebenläufig** (engl. *concurrent*) bearbeit, wobei der Prozessor (im schnellen Wechsel) umgeschaltet wird.

Mehrprogrammbetrieb



 = Prozessor arbeitet

Mehrprogrammbetrieb

- Mehrer Aufträge werden **nebenläufig** (engl. *concurrent*) bearbeit, wobei der Prozessor (im schnellen Wechsel) umgeschaltet wird.
- Flexible Prozessorzuteilung. Der Prozessor kann auch während des Abarbeiten eines Auftrag zu einem anderen wechseln, weil
  - der gerade ausgeführte Auftrag wartet.
  - ein dringenderer Auftrag bearbeitet werden soll.
  - gleichberechtigte Aufträge gleichmäßig (fair) bearbeiten werden sollen.
- Ermöglicht **Timesharing**. Mehrere Benutzer können gleichzeitig mit dem Rechensystem arbeiten.

Inhalt

Prozesse

Auf modernen Rechensystemen können mehrere Programme nebenläufig bearbeitet werden.

- Benutzerprogramme, Lesen/Schreiben von/auf Platten, Drucken von Dateien, etc.
- Ermöglicht bessere Nutzung der Ressourcen.

Ein **Prozess** (engl. *process*, ***task***) ist die Abstraktion eines laufenden Programms.

Ein Prozess benötigt **Betriebsmittel** (Prozessorzeit, Speicher, Dateien, etc.) und ist selbst ein Betriebsmittel.

Ein Prozessor führt in jeder Zeiteinheit maximal einen Prozess aus. Laufen mehrere Prozesse, dann finden Prozesswechsel statt.

Kontext eines Prozesses

Prozesse werden vom Betriebssystem verwaltet.

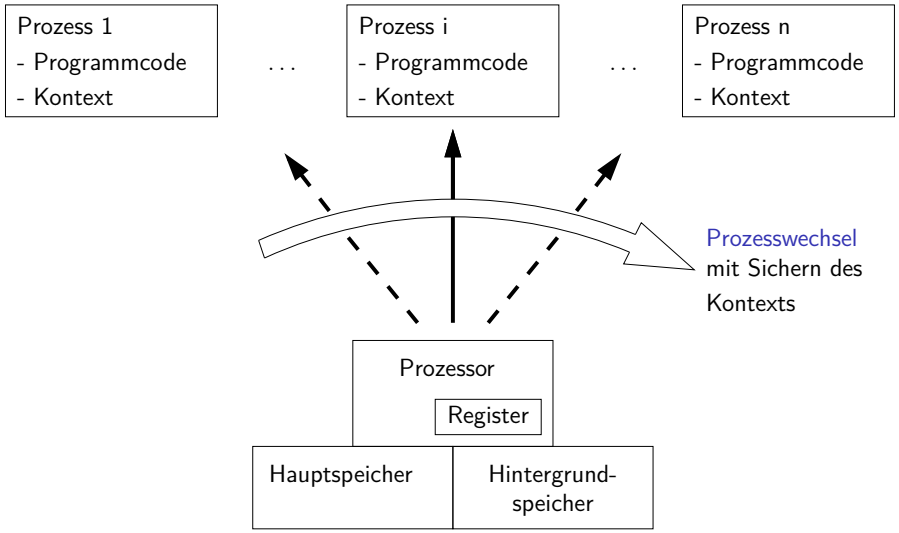
Eine Aufgabe des Betriebssystems besteht darin, den verschiedenen Prozessen Prozessorzeit zuzuteilen (engl. ***scheduling)***.

Um das Scheduling zu ermöglichen, besteht ein Prozess aus dem **auszuführenden Programmcode** (inkl. Daten) und einem **Kontext**.

Zum **Kontext** eines Prozesses gehören

- die **Registerinhalte des Prozessors**,
- dem Prozess zugeordnete **Bereiche des direkt zugreifbaren Speichers**,
- durch den Prozess **geöffnete Dateien**,
- dem Prozess **zugeordnete Peripheriegeräte**,
- **Verwaltungsinformationen** über den Prozess.

Prozesswechsel



Prozesszustände

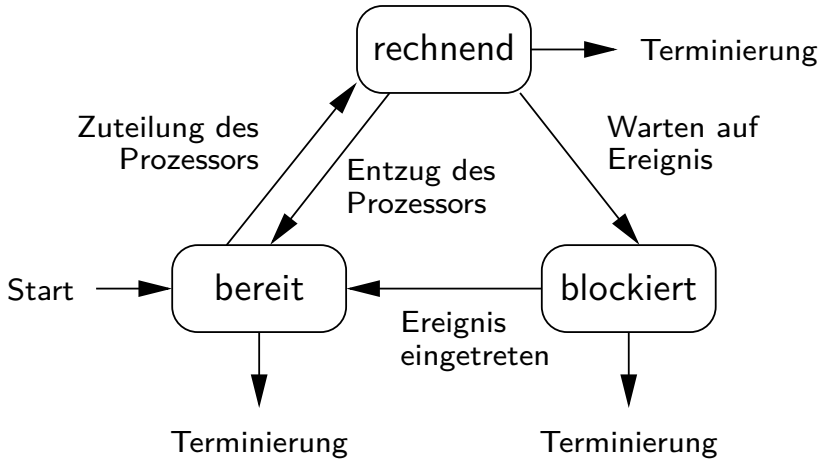
Ein Prozess befindet sich zu jedem Zeitpunkt in einem bestimmten **Zustand**. Es finden **dynamisch Zustandsübergänge**, also Veränderungen des Prozesszustands statt.

Prozesszustände

- **rechnend** (*running*). Prozess wird momentan ausgeführt.
- **bereit** (*ready*). Prozess ist ausführbar und wartet auf die Zuteilung des Prozessors.
- **blockiert** (*blocked*). Prozess kann momentan nicht ausgeführt werden und wartet auf das Eintreten eines Ereignisses (z.B. Nachricht von einem E/A-Prozess).

I.d.R. existieren noch weitere Zustände, z.B. *new* (Prozess wird gerade erzeugt) oder *exit* (Prozess wird gerade beendet) sowie evtl. weitere Verfeinerungen der obigen Zustände.

Zustandsübergänge



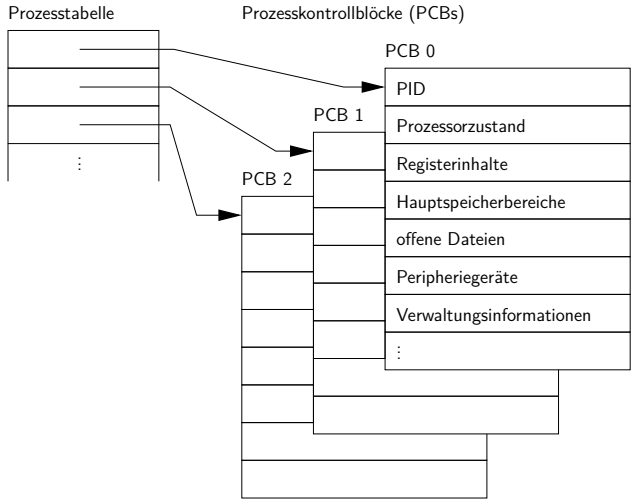
Prozesstabelle und Prozesskontrollblöcke (1/2)

Das Betriebssystem verwaltet Prozesse mit Hilfe einer **Prozesstabelle**, die **Prozesskontrollblöcke** (engl. *process control blocks*, **PCBs**), bzw. Verweise auf PCBs, für alle existierenden Prozesse enthält.

Ein PCB enthält

- den Kontext des Prozesses,
- für das Scheduling benötigte Informationen,
- weitere Verwaltungsinformationen.

Prozesstabelle und Prozesskontrollblöcke (2/2)



Prozesswechsel (Dispatching) (1/2)

Dispatcher (deutsch *Prozessumschalter*)

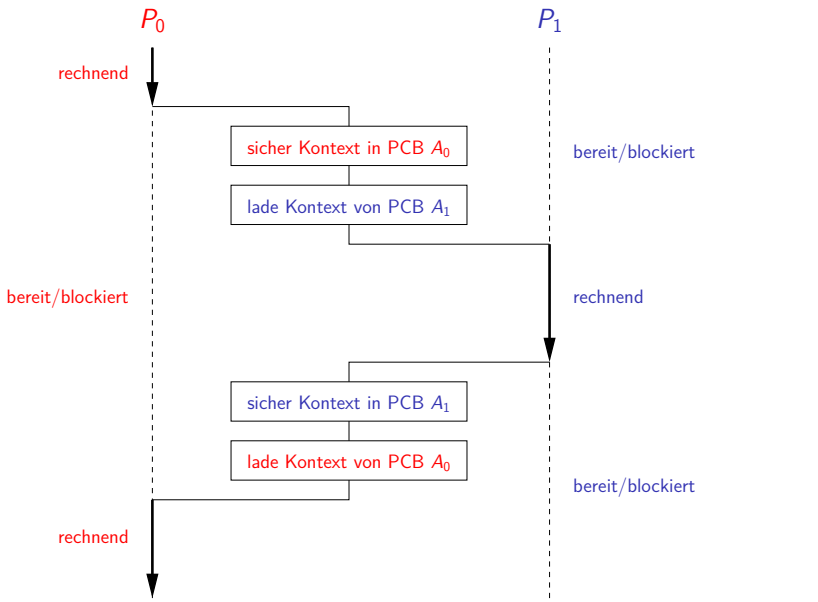
Prozesswechsel

- Der aktuelle Kontext eines Prozesses P_0 wird im einem PCB gesichert.
- Der Kontext eines anderen Prozesses P_1 wird aus einem PCB geladen.
- Der Zustand beider PCBs muss aktualisiert werden.
- **Prozesswechsel** = **Kontextwechsel**

Prozesswechsel sind relativ teuer (benötigen viel Zeit).

Prozesswechsel wird häufig von spezieller Hardware unterstützt.

Prozesswechsel (Dispatching) (2/2)



Inhalt

Scheduling

Scheduling ist die Zuteilung von Prozessorzeit an die Prozesse.

Komponenten des Scheduling.

- **Prozesswechselkosten.** Prozesswechsel sind relativ teuer, weil der Kontextes der Prozesse gesichert/gelanden werden muss.
- **Warteschlangenmodell.** Wartende Prozesse werden in internen Warteschlangen gehalten, die Auswahlstrategie der Warteschlangen haben wesentlichen Einfluss auf das Systemverhalten.
- **Scheduling-Verfahren.**

Fragen

- Wann erfolgt der Kontextwechsel?
- Nach welchen Kriterien wird der Prozess ausgewählt, der als nächstes bearbeitet wird?

Anforderungen (1/2)

Alle Systeme

- *Fairness*. Jeder Prozess bekommt Rechenzeit der CPU.
- *Policy Enforcement*. Durchsetzung der Verfahrensweisen, keine Ausnahmen.
- *Balance*. Alle Teile des Systems sind (gleichmäßig) ausgelastet.
- *Data Protection*. Keine Daten oder Prozesse gehen verloren.
- *Scalability*. Mittlere Leistung wird bei wachsender Last (Anzahl von Prozessen) beibehalten. D.h. es gibt keine Schwelle, ab der das Scheduling nur noch sehr langsam oder gar nicht mehr funktioniert.

Anforderungen (2/2)

Batch-Systeme (Stapelverarbeitungssysteme)

- *Throughput* (Durchsatz). Maximiere nach Prozessen pro Zeiteinheit.
- *Turnaround Time*. Minimiere die Zeit vom Start bis zur Beendigung eines Prozesses.
- *Processor Load*. Belege die CPU konstant mit Jobs.

Inhalt

First Come First Served (FCFS)

Prinzip

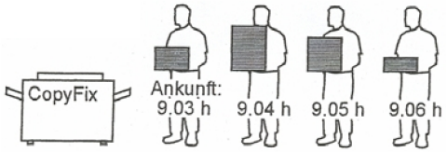
- Prozesse bekommen den Prozessor entsprechend ihrer Ankunftsreihenfolge zugeteilt.
- Keine Abhängigkeiten zwischen den Prozessen.
- Laufende Prozesse werden nicht unterbrochen.

Eigenschaften

- Fair (jeder Prozess kommt dran).
- Einfache Implementierung.

Bemerkungen

- Die mittlere Wartezeit kann unter Umständen sehr hoch werden.



Quelle: Carsten Vogt, Betriebssysteme, 2001

Inhalt

Shortest Job First (SJF)

Prinzip

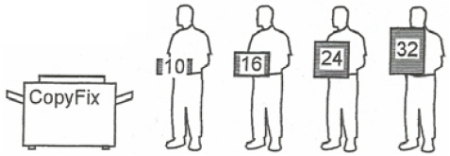
- Es wird jeweils der Prozess mit der kürzesten Rechenzeit als nächstes gerechnet.
- Keine Abhängigkeiten zwischen den Prozessen.
- Laufende Prozesse werden nicht unterbrochen.

Eigenschaften

- Nicht fair (kurze Prozesse können lange Prozesse überholen).

Problem

- Wie wird die Rechenzeit eines Prozesses ermittelt?



Quelle: Carsten Vogt, Betriebssysteme, 2001

Scheduling und Mehrprogrammbetrieb

Auf Sytemen im Mehrprogrammbetrieb sind normalerweise immer mehrere Prozesse zu einem Zeitpunkt rechenbereit.

Verfahren

- Man zerlegt die Rechenzeit in Zeitscheiben (gleicher oder variabler Länge) und ordnet diese nach bestimmten Kriterien (z.B. Fairness, Prioritäten, Rechenzeit, etc.) den rechenbereiten Prozessen zu.
- Hat ein Prozess seine Zeitscheibe verbraucht wird er unterbrochen und muss auf eine neue Zuteilung warten.

Inhalt

Round-Robin-Scheduling

Prinzip

- Die Rechenzeit wird in gleichlange Zeitscheiben/-schlitze (*time slices*) aufgeteilt.
- Prozesse werden in einer Warteschlange eingereiht und in FIFO-Ordnung (*first in, first out*) ausgewählt.
- Ein rechnender Prozess wird nach Ablauf einer Zeitscheibe unterbrochen und wieder hinten in die Warteschlange eingestellt (Rundlauf, *round robin*).
  Bemerkung
  Wird ein Prozess blockiert oder beendet er sich bevor dessen Zeitscheibe komplett aufgebraucht ist, wird sofort der nächste Prozess ausgewählt und kann eine Zeitscheiben lang rechnen.

Eigenschaften

- Die Prozessorzeit wird nahezu gleichmässig auf die vorhandenen Prozesse aufgeteilt.

Quelle: Carsten Vogt, Betriebssysteme, 2001

Ankunfts-/Rechenzeit

Die **Ankunftszeit** eines Prozesses ist der Zeitpunkt ab dem der Prozess vom Scheduling berücksichtigt wird. Der Prozess ist rechenbereit und wenn zu diesem Zeitpunkt der Prozessor nicht belegt ist, bekommt der Prozess sofort Rechenzeit zugeteilt.

Die **Rechenzeit** eines Prozesses ist die Anzahl an Zeiteinheiten, für die der Prozess Rechenzeit zugeteilt bekommt muss, um vollständig abzulaufen, d.h. sich zu beenden.

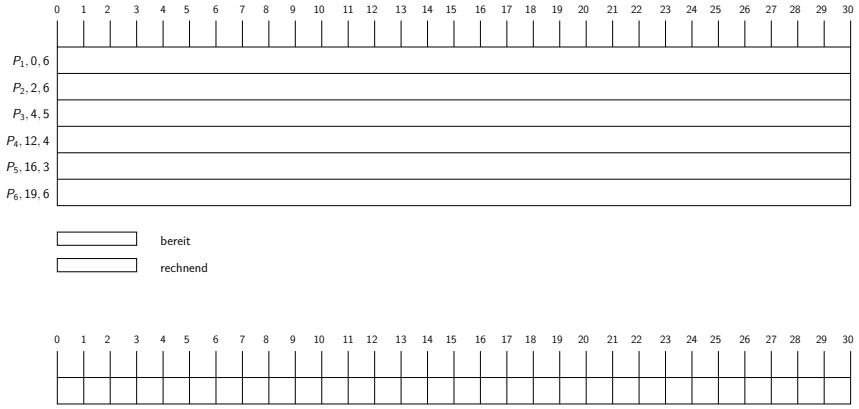
Beispiel

Gegeben seien die Prozesse P_1 bis P_6 mit folgenden *Ankunftszeiten* a_i und *Rechenzeiten* t_i.

Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

Beispiel, FCFS

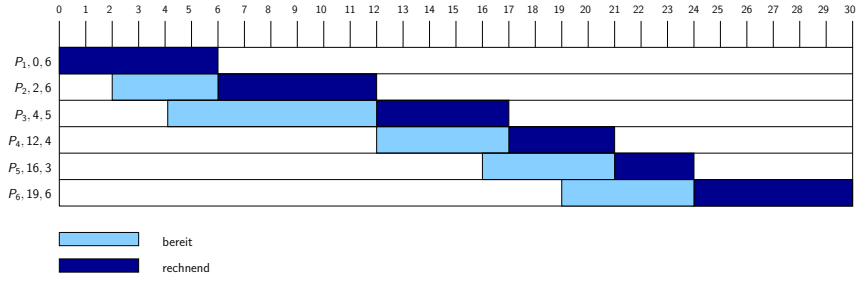
Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

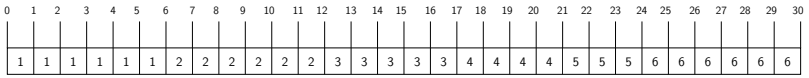


Beispiel, FCFS

Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

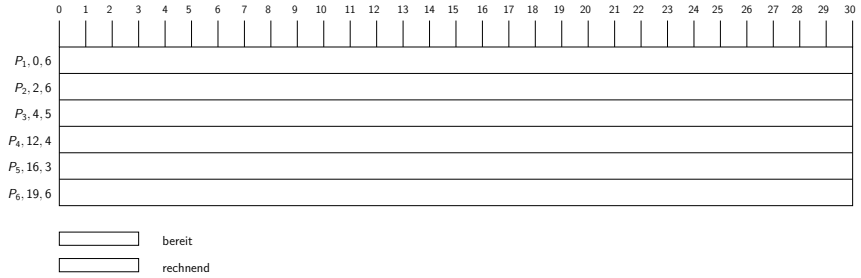
Darstellung des Schedules als *Gantt Chart* (nach Henry L. Gantt 1861-1919) oder *Balkenplan*.

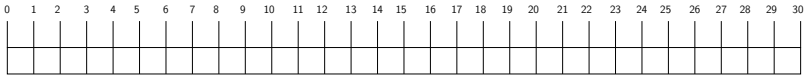




Beispiel, SJF

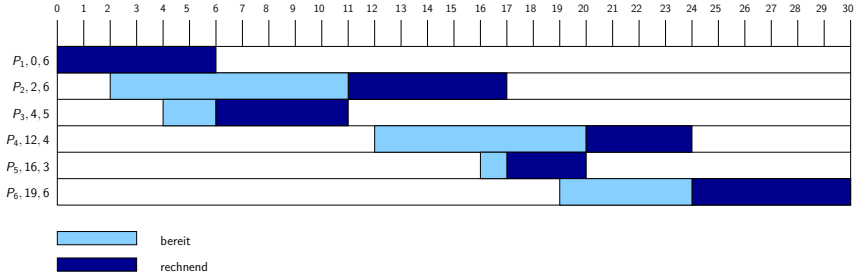
Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

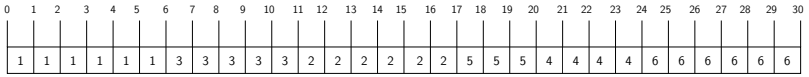




Beispiel, SJF

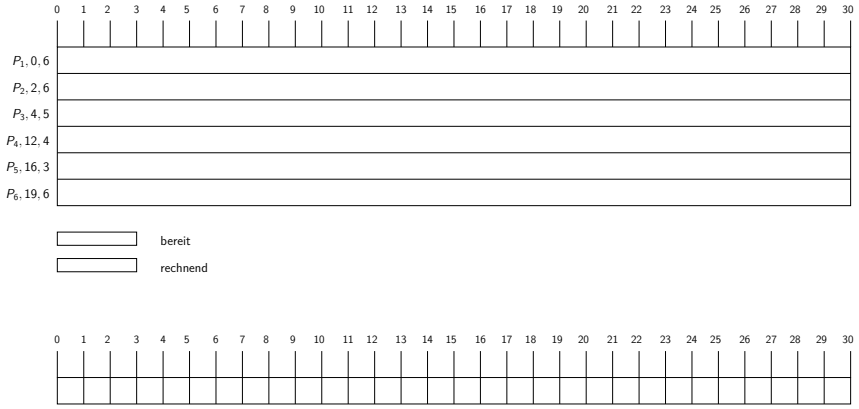
Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6





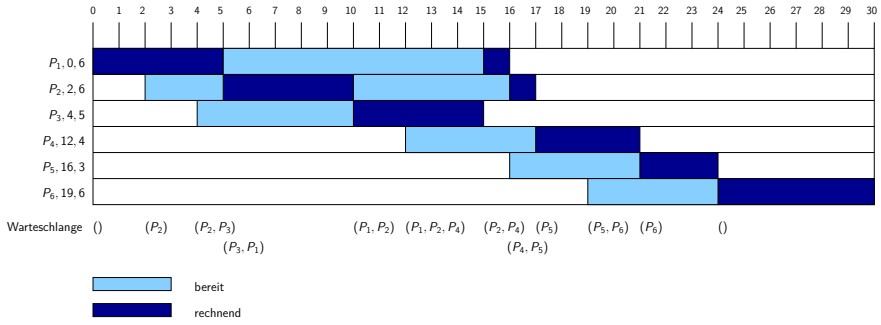
Beispiel, Round-Robin (Zeitscheibe 5)

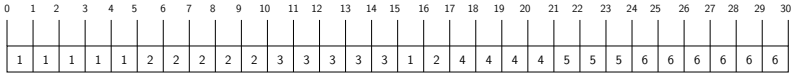
Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6



Beispiel, Round-Robin (Zeitscheibe 5)

Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6





Inhalt

Inhalt

Synonyme für bestehende Typen

Ein Synonym (*type synonym*) ist ein neuer Name für einen existierenden Typ, man definiert ein Synonym mit **`type`**.

Ausdrücke verschiedener Synonyme desselben Typs sind kompatibel.

Beispiel

Strings bilden in Haskell keinen eigenen Datentypen, sondern sind ein Synonym für Listen von Zeichen.

```
type String = [Char]
```

Durch die Verwendung von Synonymen kann man die Lesbarkeit von Programmen verbessern, z.B. können komplizierte Typen abgekürzt oder sprechende Namen verwendet werden.

Beispiel

Synonyme **Nibble** und **Byte**.

```
type Nibble = (Bool, Bool, Bool, Bool)
type Byte = (Nibble,Nibble)

lastBitNibble :: Nibble -> Bool
lastBitNibble (a, b, c, d) = d

lastBitByte :: Byte -> Bool
lastBitByte (a, b) = lastBitNibble b
```

Test in **ghci**.

```
> lastBitNibble (True, True, True, False)
False
> lastBitByte ((True, True, True, False), (False, True, True, Tru
True
```

Neue Typen

Neue Typen können mit **`data`** definiert werden.

Beispiel

```
data Signal = X | O
```

Damit wird der Typ **`Signal`** definiert, der genau die zwei Werte **`X`** und **`O`** hat.

`X` und **`O`** sind (parameterlose) Konstruktoren, die jeweiligen Ausdrücke erzeugen einen Wert des Typs **`Signal`**.

Beispiel

```
> let z = X
> :t z
z :: Signal
```

Informationen

Hinweis

Im laufenden GHCi kann man mit den Kommandos

```
:info [name]
:i [name]
```

die verfügbaren Informationen über den vergebenen Namen *name* bekommen, insbesondere auch, ob der Name *name* bereits vergeben ist.

Eigenschaften von Typen

Selbstdefinierten Typen fehlen einige Eigenschaften, die die in **`Prelude`** definierten Typen mitbringen.

Z.B. fehlt die Funktion **`show`**, die für jeden Wert der Typs eine Zeichenkettenrepräsentation (**`String`**) zurückliefert oder der Test auf Gleichheit.

Beispiel

```
> let z = X
> :t z
z :: Signal
> show z
No instance for (Show Signal) arising from a use of 'show'
...
> X == 0
No instance for (Eq Signal) arising from a use of '=='
...
```

Man kann (selbstdefinierten) Typen Eigenschaften verleihen, indem man den Typ einer oder mehrerer Typklasse (*typeclasses*) zuordnet.

Inhalt

Typklassen

Die Deklaration einer Typklasse beginnt mit dem Schlüsselwort **`class`**, gefolgt vom Namen der Typklasse und einem Platzhalter für einen Typ (in den folgenden Beispielen **`a`**), der Instanz der Typklasse werden soll.

Dem Schlüsselwort **`where`** folgen die Deklarationen der Funktionen, die Teil der Typklasse sind. In den Deklarationen kann der Platzhalter für den Typ verwendet werden.

Weiterhin können Definitionen der deklarierten Funktionen enthalten sein.

Beispiel

Vereinfachte Versionen der Typklassen **`Show`** und **`Eq`** (*equality*) aus **`Prelude`**.

```
class Show a where
    show :: a -> String
```

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

Instanz einer Typklasse

Ein neuer Typ wird Mitglied eine Typklasse, indem eine Instanz der Typklasse über diesem Typ gebildet wird.

Dazu dient das Schlüsselwort **`instance`** gefolgt von der Typklasse und dem Typ.

Dem Schlüsselwort **`where`** folgen Definitionen von Funktion, die in der Typklasse deklariert wurden.

Zur Bildung einer Instanz müssen alle in der Typklasse deklarierten Funktionen gültig Definitionen haben. Welche Funktionen dazu definiert werden müssen, hängt von den in der Typklasse bereits angegebenen Definitionen ab.

Die Bildung einer Instanz führt dazu, dass die Funktionen der Typklasse, die mit dem Typ-Platzhalter deklariert wurden, jetzt für den Typ, über dem die Instanz gebildet wurde, verfügbar sind.

Polymorphismus

In Haskell führt die Bildung einer Instanz zum Überladen der Funktionen (*overloading*), die in der Typklasse deklarierten sind. D.h. von einer Funktion stehen verschiedene Definition zur Verfügung und beim Aufruf der Funktion wird anhand der Typen der Argumente entschieden, welche Definition der Funktion Anwendung findet.

Polymorphismus (Vielgestaltigkeit) nennt man das Konzept, das durch den Kontext bestimmt wird, welche Definition eines Sprachkonstrukts, z.B. einer Funktion, verwendet wird.

Typklasse Show

Um Mitglied der Typklasse **Show** zu werden, muss eine Instanz von **Show** gebildet werden. Dabei ist die Definiton der Funktion **show** ausreichend.

Beispiel

```
instance Show Signal where
  show X = "X"
  show O = "O"
```

Jetzt gibt es eine Funktion **show :: Signal -> String**, die auf Argumente vom Typ **Signal** angewendet wird.

```
> let z = X
> :t z
z :: Signal
> show z
"X"
> print z
X
```

Typklasse Eq (1/2)

Zum Bilden einer Instanz der Typklasse **Eq** ist die Definition des Operators **(==)** ausreichend, denn die Definition von **(/=)**, basierend auf **(==)**, ist bereits in **Eq** enthalten.

Beispiel

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

```
instance Eq Signal where
  O == O    = True
  O == X    = False
  X == O    = False
  X == X    = True
```

Typklasse Eq (2/2)

Durch das Bilden einer Instanz von **Eq** über **Signal** sind die Operatoren **(==)** und **(/=)** für Werte vom Typ **Signal** definiert.

```
> X == O
False
> O /= O
False
```

Inhalt

Eingeschränkte Typ-Parameter

Bei der Verwendung von Typ-Parametern, z.B. bei der Definition von Funktionen oder Typklassen, kann festgelegt werden, dass nur Typen, die Mitglieder einer oder mehrerer Typklassen sind, als Agumente für diese Parameter benutzt werden können, dann spricht man von eingeschränkte Typ-Parametern.

Bei der Vereinbarung des eingeschränkte Parameters wird diesem die Typklasse, auf die er eigeschränkt wird, vorangestellt (kann in runde Klammern eingeschlossen werden). Wird ein Parameter mehrfach eingeschränkt oder werden mehrere eingeschränkt Parameter vereinbart, werden diese durch Kommata getrennt und in runde Klammern eingeschlossen.

Der Vereinbarung von eingeschränkten Parametern folgt **=>** und die Verwendung der Typ-Parameter.

Beispiele

```
class (Eq a) => X a where
  foo :: a -> a -> Bool

f :: (Eq a, Show a) => a -> String

g :: (Eq a, Show b) => a -> b -> String
```

Eingeschränkte Typ-Parameter und Funktionen (1/2)

Beispiel

Die Funktion **contains** ermittelt, ob ein Wert vom Typ **a** in einer Liste vom gleichen Typ **[a]** enthalten ist, mit der Einschränkung, dass der Typ **a** Mitglied der Typklasse **Eq** sein muss.

```
contains :: (Eq a) => a -> [a] -> Bool
contains _ [] = False
contains z (x:xs)
  | z == x    = True
  | otherwise = contains z xs
```

Hinweis. Vergleiche Funktion **elem** aus **Prelude**.

Eingeschränkte Typ-Parameter und Funktionen (2/2)

```
contains :: (Eq a) => a -> [a] -> Bool
contains _ [] = False
contains z (x:xs)
  | z == x    = True
  | otherwise = contains z xs
```

In der Definition der Funktion **`contains`** sind *pattern matching* und *guarded equations* kombiniert.

- `_` ist eine *wildcard*, zu diesem Pattern passt jeder Wert, des zugehörigen Definitionsbereichs. Dieses Pattern wird verwendet, wenn der konkrete Wert nicht benötigt wird.
- `[]` ist die leere Liste.
- **`z`** ist ein Pattern, zu dem ebenfalls jeder Wert des zugehörigen Definitionsbereichs passt, der konkrete Wert wird an **`z`** gebunden und kann nachfolgend verwendet werden.
- **`(x:xs)`** passt zu jeder nicht leeren Liste. An **`x`** wird das erste Element (*head*) und an **`xs`** die Liste der nachfolgenden Elemente (*tail*) gebunden.

Fall Through

Bei *pattern matching* und *guarded equations* gilt *fall through*, d.h. die *pattern*/die *guards* werden von oben nach unten abgearbeitet, bis ein passendes/passender gefunden ist. Passt das aktuelle *pattern*/der aktuelle *guard* nicht, wird zum nächsten weitergegangen.

Bei *guarded equations* geht *fall through* noch weiter. Passt keiner der *guards*, wird das zu den *guarded equations* gehörige *pattern* als nicht passend behandelt, d.h. es wird mit dem nächsten *pattern* fortgefahren (falls vorhanden).

Beispiel

```
contains :: (Eq a) => a -> [a] -> Bool
contains _ [] = False
contains z (x:xs)
  | z == x    = True
contains z (x:xs) = contains z xs
```

Eingeschränkte Typ-Parameter und Typklassen (1/2)

Beispiel

Typklasse `Ord`

```
class (Eq a) => Ord a where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min             :: a -> a -> a

  -- Minimal complete definition (<=) or compare
  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT

  x <= y = compare x y /= GT
  x <  y = compare x y == LT
  x >= y = compare x y /= LT
  x >  y = compare x y == GT

  max x y
    | x >= y    = x
    | otherwise = y
  min x y
    | x <  y    = x
    | otherwise = y
```

Eingeschränkte Typ-Parameter und Typklassen (2/2)

Damit ein Typ einen Instanz der Typklasse `Ord` werden kann, muss dieser Typ bereits Instanz der Typklasse `Eq` sein.

Dann reicht die Definition des Operators `(<=)`, um die Funktion `compare` vollständig zu definieren, denn eine Definition des Operator `(==)` ist bereits vorhanden.

Mit der Definition von `compare` sind alle anderen Operatoren und Funktion der Typklasse `Ord` definiert.

Hinweis

Alternativ können alle anderen Operatoren und Funktionen auch durch die Angabe einer Definition für `compare` vollständig definiert werden.

Inhalt

Record Syntax (1/2)

Wenn der Konstruktor eines Datentyps mehrere Argumente hat, ist es meistens ohne Kommentare nicht ersichtlich, was die einzelnen Komponenten darstellen.

Beispiel

```
data ProzessA = ProzessA
                  String  -- pid
                  Int     -- arrival
                  Int     -- computing
                  deriving (Show)
```

Weiterhin müssen neue Funktionen definiert werden, um Zugriff auf die einzelnen Komponenten zu bekommen.

Bemerkung

Durch **`deriving (Show)`** erbt ein Datentyp die Standarddarstellung, die für einen Wert des Datentyps eine **`String`**-Darstellung des Ausdrucks zurückliefert, mit dem dieser Wert erzeugt werden kann.

```
> show (ProzessA "P1" 5 10)
"ProzessA \"P1\" 5 10"
```

Record Syntax (2/2)

Record Syntax erlaubt es, die Argumente eines Konstruktors zu benennen. Benannte Argumente werden als Felder bezeichnet.

Wenn Record Syntax benutzt wird, werden Funktionen zum Zugriff auf die Felder (*accessor functions*) automatisch erzeugt.

Beispiel

```
data Prozess = Prozess { pid        :: String
                       ,arrival    :: Int
                       ,computing :: Int } deriving (Show)
```

```
> let p = Prozess { pid = "P1", arrival = 5, computing = 10 }
> :t pid
pid :: Prozess -> String
> pid p
"P1"
> arrival p
5
> computing p
10
> print p
Prozess {pid = "P1", arrival = 5, computing = 10}
```

Record Syntax und Pattern Matching

Pattern Matching ist mit Record Syntax ebenfalls möglich, dabei ist die Reihenfolge der Felder beliebig und es müssen nicht alle Felder berücksichtigt werden.

Beispiel

```
arrivedBefore :: Int -> Prozess -> Bool
arrivedBefore t Prozess { arrival = a } = a < t
```

```
> let p = Prozess { pid = "P1", arrival = 5, computing = 10 }
> arrivedBefore 2 p
False
> arrivedBefore 6 p
True
```

Inhalt

Gemeinsame Ressourcen

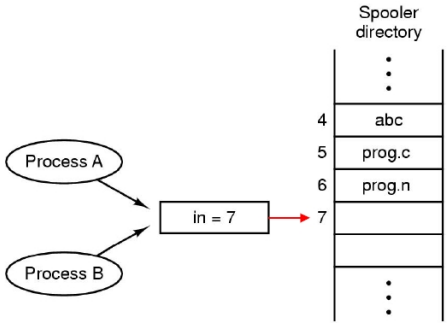
Nebenläufig ablaufende Prozesse (Mehrprogrammbetrieb) haben häufig **gemeinsame Ressourcen**.

- **Geräte.** Drucker, Platten, usw.
- **Daten.** Dateien, Shared Memory, usw.

Zugriffe auf gemeinsame Ressourcen müssen geordnet erfolgen. Um zu vermeiden, dass die Ergebnisse abhängig sind von der **Reihenfolge** der Abarbeitungsschritte der einzelnen Prozesse (**Race Condition**).

Beispiel. Race Condition (1/3)

Zwei Prozesse A und B schreiben Druckaufträge in einen Druckerspooler.
Die Prozesse verwenden hierzu die Kontrollvariable `in` (nächster freier Slot im Spoolerdirectory) des Druckerspoolers.



Beispiel. Race Condition (1/3)

```
1  // main process
2
3  start_concurrent(A, B);
```

```
1  // A
2
3  in = spooler->in;
4  spool(spooler, in, jobA);
5  in = in + 1;
6  spooler.in = in;
```

```
1  // B
2
3  in = spooler->in;
4  spool(spooler, in, jobB);
5  in = in + 1;
6  spooler->in = in;
```

Beispiel. Race Condition (2/2)

Prozess A

- Liest die Variable `in`.
- Schreibt den Auftrag `jobA` in den durch `in` angegebenen Slot (= 7) des Spoolers.
- Berechnet den Wert (= 8), mit dem `in` aktualisiert werden soll.
- Wird vom Scheduler unterbrochen und von Prozess B aus dem Prozessor verdrängt.

Prozess B

- Liest Variable `in`.
- Schreibt den Auftrag `jobB` den durch `in` angegebenen Slot (= 7) des Spoolers überschriebt damit den Auftrag `jobA`.
- Aktualisiert Variable `in` (neuer Wert = 8) und terminiert.

Prozess A

- Nimmt die Bearbeitung wieder auf.
- Aktualisiert Variable `in` (neuer Wert = 8). In der (falschen) Annahme das niemand zwischenzeitlich auf den Spooler zugegriffen hat und terminiert.

Der Auftrag von Prozess A wird nie bearbeitet.

Problem

Ein Prozess befindet sich in seinem **kritischen Abschnitt**, wenn er auf gemeinsame Ressourcen zugreift.

Vermeidung von Race Conditions.

- **Wechselseitiger Ausschluss** (*mutual exclusion*). Keine zwei Prozesse dürfen sich gleichzeitig in ihren kritischen Abschnitten befinden.
- Kein Prozess, der außerhalb seines kritischen Abschnitts läuft, darf andere Prozesse blockieren.
- Es dürfen keine Annahmen über Hardware (z.B. Geschwindigkeit und Anzahl der CPUs) und Betriebssystem (z.B. Scheduling-Algorithmus) gemacht werden.
- Kein Prozess sollte ewig darauf warten müssen, in seinen kritischen Abschnitt einzutreten.

Inhalt

Mutex (1/2)

Ein **Mutex** (von *mutual exclusion*) kann zur Synchronisation von nebenläufigen Prozessen benutzt werden.

Ein Mutex **ist** eine neue (geschützte) Variablenart auf der nur **unteilbaren (atomaren) Operationen** ausgeführt werden können.

- Datenstruktur

```
mutex {
  boolean          free;
  prozess_queue    queue;
}
```

- Deklaration und Initialisierung

```
1  mutex m;
2  mutex m = true;
3  mutex m = false;
```

Der Mutex `m` wird deklariert und wie folgt initialisiert

- 1: `m.free=true` 2: `m.free=true` 3: `m.free=false`
- `m.queue` ist eine (leere) Datenstruktur, die Prozesse (Verweise auf Prozesskontrollblöcke) aufnehmen kann.

Mutex (2/2)

Operationen auf einem Mutex.

- **down** (wait, P)

```
down(m) {
  if (m.free)
    m.free = false;
  else {
    block(this_process);
    insert(m.queue, this_process);
  }
}
```

- **up** (signal, V)

```
up(m) {
  if (is_empty(m.queue))
    m.free = true;
  else {
    process = remove(m.queue);
    wake_up(process);
  }
}
```

Beispiel, ohne Race Condition

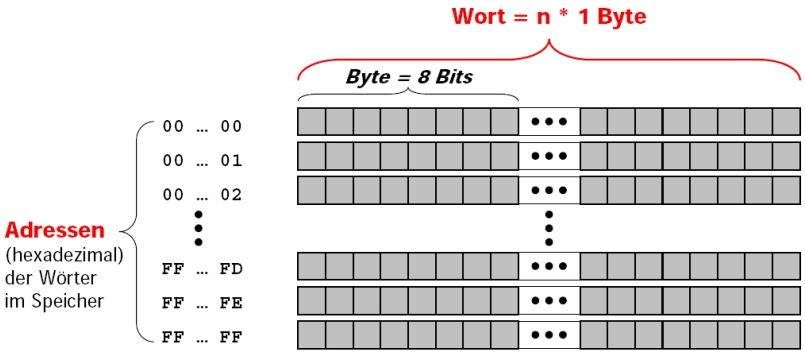
```
1 // main process
2 global mutex m;
3 start_concurrent(A, B);
```

```
1 // A
2 down(m);
3 in = spooler->in;
4 spool(spooler, in, jobA);
5 in = in + 1;
6 spooler->in = in;
7 up(m);
```

```
1 // B
2 down(m);
3 in = spooler->in;
4 spool(spooler, in, jobB);
5 in = in + 1;
6 spooler->in = in;
7 up(m);
```

Inhalt

Speicherorganisation



Speicheraufteilung

Die einfachste Speicherverwaltungsstrategie ist den Speicher zwischen Prozesses und Betriebssystem aufzuteilen.

Dann wird entweder einem Prozess der gesamten, für Prozesse reservierten, Speicher zur Verfügung gestellt oder der Speicher wird unter mehreren Prozessen aufgeteilt.

Dabei wird jeder Prozess immer **vollständig** im Hauptspeicher gehalten.

Adressraum

Wird der Speicher zwischen Betriebssystem und einem oder mehreren Prozessen aufgeteilt, ergibt sich ein wesentliches Problem.

Prozesse sprechen grundsätzlich direkt physikalische Adressen an.

- **Relokation.** Ein Programm enthält absolute Adressen, diese müssen relativ zur Lage des Prozesses im Speicher umgesetzt werden.
- **Schutz.** Jeder Prozess soll nur die Adressen des Speicherbereichs ansprechen, der im zugeteilt ist.

Diese Anforderungen werden durch die Einführung des **Adressraums** (*adress space*), einer naheliegenden Speicherabstraktion, erfüllt.

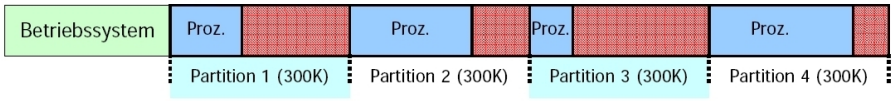
Ein Adressraum ist eine Menge von Adressen, die ein Prozess zur Adressierung des Speichers nutzen kann.

Physikalisch ist der Adressraum, im einfachsten Fall, ein zusammenhängender Speicherbereich (Partition) fester Größe.

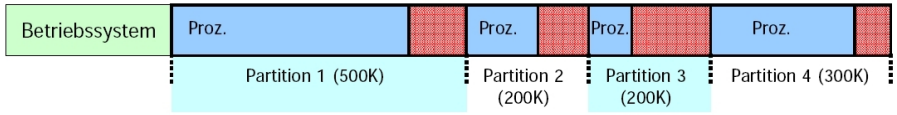
Aus Sicht des Prozesses ist der Adressraum der ganze zur Verfügung stehende Speicher, der in der Regel mit der Adresse 0 beginnt. Die Endadresse ist abhängig von der Größe des zugeteilten Speicherbereichs.

Partition

Feste Partition/Adressräume gleicher Größe.



Feste Partition/Adressräume unterschiedlicher Größen.



Dynamische Relokation

Der Adressraum jedes Prozeses wird auf einen feste Partition des Speichers abgebildet.

Der Prozessor hat zwei zusätzliche Register, das **Basisregister** (`base`) und das **Limitregister** (`limit`). Diese Register gehören zum Kontext des Prozesses.

Beim Erzeugen eines Prozesses wird die physikalische Anfangsadresse der dem Prozess zugeteilten Partition in das Basisregister geladen und die Größe der Partition wird in das Limitregister geladen.

Wenn ein Prozess Speicher referenziert, um eine Befehl zu holen, einen Datenwort zu lesen oder zu schreiben, etc. benutzt der Prozessor automatisch Limit- und Basisregister.

- **Schutz**. Für jede Adresse prüft der Prozessor ob der Wert kleiner oder gleich dem Wert im Limitregister ist, wenn nicht wird der Zugriff verweigert.
- **Relokation**. Zu jeder Adresse addiert der Prozessor automatisch den Wert im Basisregister und schreibt die berechnete Adresse auf den Adressbus.

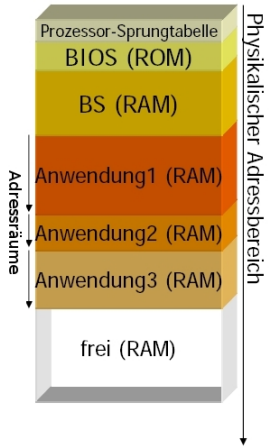
Bemerkung
Dynamische Relokation wurde noch im Intel 8088 eingesetzt.

Beispiel UNIX mit dynamischer Relokation

Mehrere Prozesse, jeder hat **eigenen Adressraum**.

Bei jedem Kontextwechsel werden Basis- und Limitregister mitgeführt, sodass jeweils der Adressraum des aktuellen Prozesses gestellt wird.

- **Relokation**. Alle Programme werden für den gleichen Adressraum kompiliert.
- **Schutz**. Fremder Speicher ist gar nicht sichtbar.



Inhalt

Moderne Speicherverwaltung

Die Anforderungen an die Verwaltung des Speichers ergeben sich aus den Design modernen Rechensysteme.

Es laufen soviele Prozesse, dass der physikalische Speicher nicht groß genug ist um alle gleichzeitig aufzunehmen.

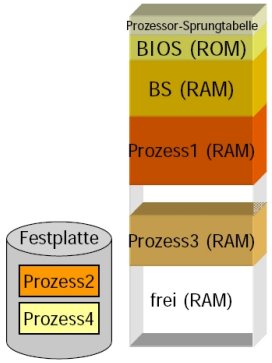
Swapping ist ein grundlegender Ansatz, wie man der überlastung des Speichers begegnen kann, wobei der Adressraum eines Prozesses entweder vollständig im Speicher gehalten wird oder vollständig auf den Hintergrundspeicher (z.B. Festplatte) ausgelagert wird.

Swapping, Prinzip

Swapping beschreibt das Ein-/Auslagern des **vollständigen** Adressraumes der Prozesses.

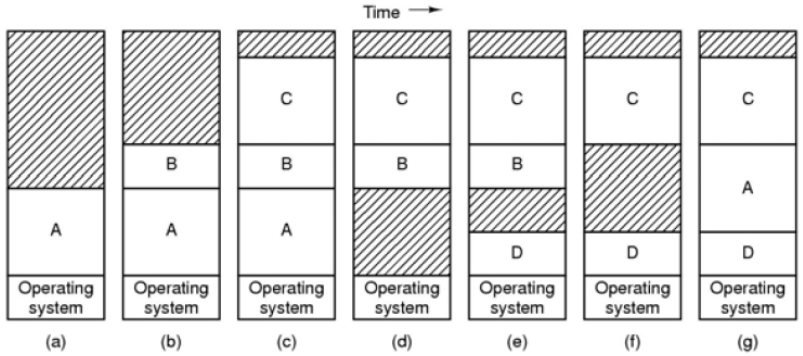
- Auslagern von (z.B. blockierten) Prozessen auf die Festplatte.
- Einlagern von (z.B. bereiten) Prozesses in den Speicher.

Es ist keine spezielle Hardware notwendig. Der Scheduler hat überblick über den rechnenden, die bereiten und blockierte Prozesse und kann die Ein-/Auslagerung veranlassen.



Swapping, Beispiel

Speicherzuteilung für einzelne Prozesse ändert sich bei der Ein- und Auslagerung (freier Speicher = schraffierte Bereiche).

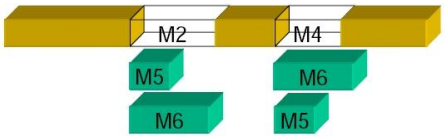


Beim Einlagern kann sich der physiklaische Adressraum (die Partition) des Prozesses ändern. Das heißt bei dynamischer Relokation müssen beim Einlagern eines Prozesses u.U. die Werte im Basis- und Limitregister angepasst werden.

Swapping, Probleme

Positionierung

M2, M4 werden freigegeben, wo M5, M6 platzieren?



Fragmentierung

Anforderung hat nur selten genau die Größe eines freien Speicherbereichs. Nach einiger Zeit entstehen viele kleine *Löcher* im Speicher.

Fazit.

Verschiedene **Speicherbelegungsstrategien** sind möglich.

Speicherbelegungsstrategien (1/2)

First Fit
Der erste ausreichend große freie Speicherbereich wird belegt. Die Suche beginnt am Anfang des Speichers.

Next Fit
Wie First Fit, aber die Suche beginnt an der Stelle, wo zuletzt ein passendes Speicherbereich gefunden wurde. D.h. wenn beim letzten Mal das Loch nicht vollständig geschlossen, sondern lediglich verkleinert wurde, beginnt die Suche bei diesem Loch.

Speicherbelegungsstrategien (2/2)

Best Fit
Es wird der kleinste freie Speicherbereich belegt, der die Anforderung noch erfüllen kann. Werden mehrere passende Speicherbereiche gefunden, wird einer davon ausgewählt.

Worst Fit
Es wird der größte freie Speicherbereich belegt. Werden mehrere passende Speicherbereiche gefunden, wird einer davon ausgewählt.