# Betriebssysteme und Systemsoftware

Zusammenfassung verschiedener Strategien

Merlin Denker

Version 2

## Vorwort

Dieses Dokument soll einen Überblick über verschiedene Strategien aus der an der RWTH Aachen gehaltenen Vorlesung „Betriebssysteme und Systemsoftware“ bieten. Die vorliegende Version dieses Dokuments basiert auf den Inhalten der Vorlesung aus dem Sommersemester 2015. Es erhebt keinen Anspruch auf Vollständigkeit und/oder Richtigkeit der Informationen und ist nur dafür gedacht meinen Kommilitonen beim Lernen zu helfen.

Da ich im Sommersemester 2016 als Tutor in diesem Fach tätig bin werde ich dieses Dokument während des Semesters laufend anpassen, sollten die Inhalte von denen aus 2015 abweichen oder neue hinzugekommen seien. Ich empfehle daher, falls dieses Dokument zum Lernen genutzt wird und während des Semesters heruntergeladen wurde, am Ende des Semesters nochmal zu überprüfen ob ich eine aktualisierte Version herausgegeben habe. Diese wird gegebenenfalls auf Studydrive [1] veröffentlicht.

In den Vorlesungsfolien die vom Lehrstuhl bereitgestellt werden findet ihr zu den meisten Strategien noch ausführlichere Informationen und Beispiele. Einige Dinge finden sich nicht in den Folien, jedoch auf den Übungsblättern.

All diese Strategien sollte man für die Klausur beherrschen. Es empfiehlt sich zu lernen, wofür die Abkürzungen stehen, da daraus meist schon die komplette Funktionsweise abgeleitet werden kann. Es reicht natürlich NICHT nur diese Strategien zu beherrschen um die Klausur zu bestehen. Zum Lernen empfehle ich Altklausuren, welche man sich in der Fachschaft ausdrucken kann. Man findet diese außerdem zusammen mit viel anderem Lernmaterial auf Studydrive zum Download.

Eventuelle Fehler können gerne an mich gemeldet werden, damit sie korrigiert werden:
merlin.denker@rwth-aachen.de

[1] https://www.studydrive.net/course_65217_Betriebssysteme-und-Systemsoftware.html

# Inhaltsverzeichnis

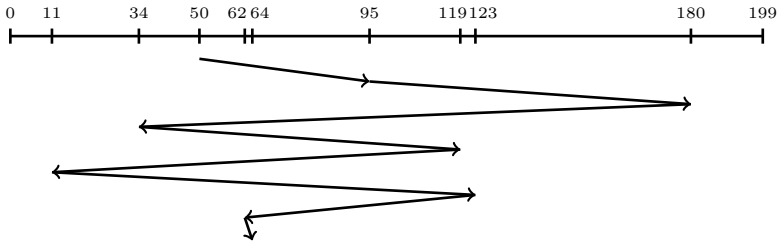
# 1 Disk-Scheduling

Disk-Scheduling-Strategien legen fest, in welcher Reihenfolge ankommende Speicheranfragen bearbeitet werden. Das Ziel dabei ist den Lese- und Schreibkopf so wenig wie möglich zu bewegen. Um die verschiedenen Strategien zu demonstrieren verwenden wir für alle Strategien dasselbe Beispiel:

- Unser Speicher hat 200 Zylinder (0 bis 199)
- Der Lese- und Schreibkopf steht anfangs bei Zylinder 50
- Die folgenden Zylinder werden in dieser Reihenfolge angefragt:
  95, 180, 34, 119, 11, 123, 62, 64

## 1.1 FCFS (first come, first serve)

Anfragen werden in der Reihenfolge bearbeitet, in der sie ankommen (FIFO-Prinzip).

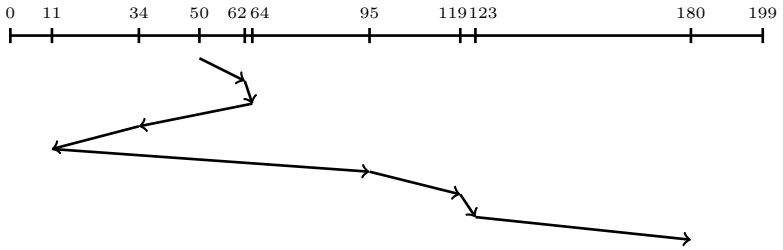



zurückgelegte Strecke:
$|50-95|+|95-180|+|180-34|+|34-119|+|119-11|+|11-123|+|123-62|+|62-64|$
$=45+85+146+85+108+112+61+2$
$=640$

In diesem Beispiel hat sich der Kopf insgesamt über 640 Zylinder bewegt. Bei der Berechnung solcher Strecken in Hausaufgaben oder Klausuren sollte man unbedingt darauf achten nicht nur die Strecken zwischen den angefragten Speicherstellen zu summieren, sondern auch noch die Bewegung von der aktuellen Position (50) zur ersten angefragten Speicherstelle (95) zu addieren.

## 1.2 SSTF (shortest seek time first)

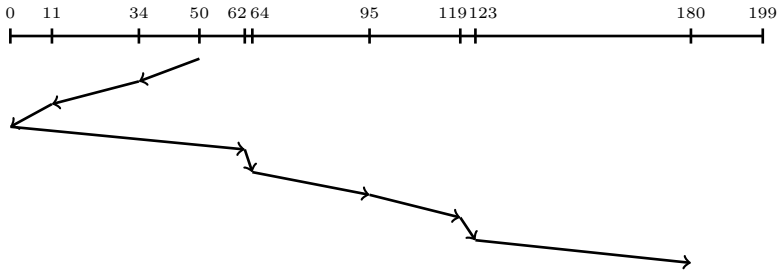
Suche den Zylinder der dem aktuellen am nächsten ist und gehe dorthin.




zurückgelegte Strecke:
$|50-62|+|62-64|+|64-34|+|34-11|+|11-95|+|95-119|+|119-123|+|123-180|$
$=12+2+30+23+84+24+4+57$
$=236$

## 1.3 SCAN

Der Kopf bewegt sich in eine Richtung bis zum Ende und dreht dann um. Unterwegs werden alle Anfragen bedient, die passiert werden. Für unser Beispiel nehmen wir an, dass sich der Kopf zu Beginn in einer Bewegung von Zylinder 199 zu Zylinder 0 befindet.

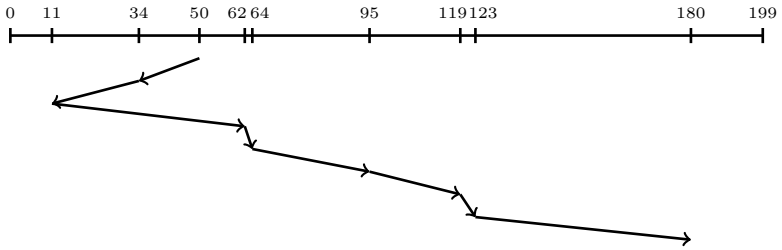



zurückgelegte Strecke:
$|50-34|+|34-11|+|11-0|+|0-62|+|62-64|+|95-119|+|119-123|+|123-180|$
$=16+23+11+62+2+31+24+4+57$
$=230$

Hier bei ist insbesondere darauf zu achten, dass der Kopf sich nach der Anfrage auf Zylinder 11 nicht direkt zu Zylinder 62 bewegt, sondern einen Umweg bis Zylinder 0 und wieder zurück läuft.

## 1.4 LOOK

Diese Strategie arbeitet im Prinzip wie SCAN, wechselt aber die Richtung wenn in die aktuelle Richtung keine weiteren Anfragen mehr vorliegen.



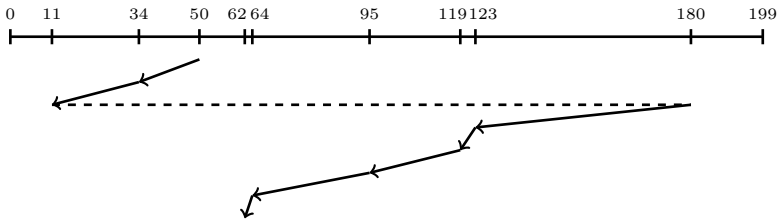

zurückgelegte Strecke:
$|50-34|+|34-11|+|11-62|+|62-64|+|95-119|+|119-123|+|123-180|$
$=16+23+51+2+31+24+4+57$
$=208$

Diese Strategie bringt also die Vorteile von SCAN, ohne jedoch einen unnötigen Umweg zu machen.

## 1.5 C-SCAN / C-LOOK

Das C steht jeweils für Circular. Diese Strategien arbeiten genau wie SCAN und LOOK, jedoch wird die Richtung nicht geändert. Das bedeutet für C-SCAN, dass der Kopf wieder an den Anfang springt sobald er das Ende erreicht. C-LOOK springt entsprechend zur ersten angefragten Adresse (vom Anfang der Bewegungsrichtung aus gesehen), sobald keine weiteren Zylinder in der aktuellen Bewegungsrichtung mehr erreicht werden können.

Für unser Beispiel nehmen wir C-LOOK und legen fest dass die Zylinder in absteigender Reihenfolge abgefahren werden (geht natürlich auch andersrum).




zurückgelegte Strecke:
$|50-34|+|34-11|+|11-180|+|180-123|+|123-119|+|119-95|+|95-64|+|64-62|$
$=16+23+169+57+4+24+31+2$
$=326$

Wichtig ist dabei, dass man den Weg des Sprungs mit aufsummiert. Verschiedene Quellen zählen diesen Weg nicht mit, was zu einer Strecke von nur 157 Zylindern führen würde. Da sich der Kopf aber tatsächlich bewegen muss macht es nicht viel Sinn diesen Summanden wegzulassen.

# 2 Paging

Paging-Strategien beschäftigen sich mit der Frage welche Daten im Hauptspeicher bleiben dürfen und welche auf die Festplatte ausgelagert werden, wenn dieser voll ist und neue Daten angefragt werden. Die Daten liegen dabei in sogenannte Seiten (Pages) vor. Das Ziel hierbei ist es, die Anzahl der Seitenfehler (Page Faults) zu minimieren. Ein Seitenfehler liegt vor, wenn eine Seite angefordert wird, die derzeit nicht im Hauptspeicher liegt.

Als Beispiel nehmen wir ein System welches 4 Rahmen besitzt. Ein Rahmen stellt eine Speicherstelle im Hauptspeicher dar, in welcher genau eine Seite Platz hat. Die Reihenfolge der angeforderten Seiten wird häufig als Referenzstring $\omega$ angegeben. In unserem Beispiel sei $\omega$ wie folgt definiert:
$\omega = 0\ 1\ 3\ 2\ 4\ 4\ 2\ 1\ 3\ 2\ 4\ 0\ 1\ 3\ 2\ 1\ 4\ 3\ 3\ 0$

## 2.1 FIFO (first in, first out)

Bei dieser Strategie wird ganz einfach die älteste Seite ausgelagert. Das Alter der Seiten wird in einer Prioritätsliste festgehalten, wobei Priorität 1 die höchste Priorität (jüngste Seite) darstellt. An den Prioritäten ändert sich also nur etwas, wenn ein Seitenfehler auftritt. In diesem Fall sinken alle Seiten um 1 in der Priorität, die Seite mit der niedrigsten Priorität wird ausgelagert und die neue Seite erhält die höchste Priorität.

| Referenz $\omega$ | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 2 | 2 | 2 | 2 | 2 | 2 |
| Rahmen 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 |
| Rahmen 3 | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Rahmen 4 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Priorität 1 | 0 | 1 | 3 | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 0 | 1 | 3 | 2 | 2 | 4 | 4 | 4 | 0 |
| Priorität 2 | | 0 | 1 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 4 | 0 | 1 | 3 | 3 | 2 | 2 | 2 | 4 |
| Priorität 3 | | | 0 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 4 | 0 | 1 | 1 | 3 | 3 | 3 | 2 |
| Priorität 4 | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 4 | 0 | 0 | 1 | 1 | 1 | 3 |
| Seitenfehler | X | X | X | X | X | | | | | | | X | X | X | X | | X | | | X |

**Anzahl Seitenfehler: 11**

## 2.2 LRU (least recently used)

Die Seite, auf die am längsten kein Zugriff erfolgt ist, wird ausgelagert. Das bedeutet: Wenn auf eine Seite zugegriffen wird rutscht sie auf Priorität 1. Die Seiten die zuvor eine höhere Priorität als diese Seite hatten sinken in der Prioritätliste um 1. Im Falle eines Seitenfehlers wird die Seite mit der niedrigsten Priorität ausgelagert und die neue Seite erhält die höchste Priorität.

| Referenz $\omega$ | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 2 | 2 | 2 | 2 | 2 | 0 |
| Rahmen 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 |
| Rahmen 3 | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Rahmen 4 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Priorität 1 | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
| Priorität 2 | | 0 | 1 | 3 | 2 | 2 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 4 | 3 |
| Priorität 3 | | | 0 | 1 | 3 | 3 | 3 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 1 | 4 |
| Priorität 4 | | | | 0 | 1 | 1 | 1 | 3 | 4 | 4 | 1 | 3 | 2 | 4 | 0 | 0 | 3 | 2 | 2 | 1 |
| Seitenfehler | X | X | X | X | X | | | | | | | X | X | X | X | | X | | | X |

**Anzahl Seitenfehler: 11**

## 2.3 SC (second chance)

Diese Strategie arbeitet ähnlich wie LRU. Für jede Seite wird ein Accessed-Bit gespeichert, welches gesetzt wird wenn erneut auf die Seite zugegriffen wird. Sobald eine Seite ersetzt werden muss wird die älteste (geringste Priorität) geprüft:

- Accessed-Bit nicht gesetzt: Seite wird ausgelagert
- Accessed-Bit gesetzt: Bit zurücksetzen und Seite zur jüngsten Seite machen (auf höchste Priorität setzen). Danach neue älteste Seite prüfen. Solange wiederholen bis eine Seite ausgelagert wird.

Sind alle Accessed-Bits gesetzt werden sofort alle wieder zurückgesetzt.

Im Beispiel markiert ein ’ ein gesetztes Accessed-Bit.

| Referenz $\omega$ | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Rahmen 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Rahmen 3 | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Rahmen 4 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Priorität 1 | 0 | 1 | 3 | 2 | 4 | 4’ | 4’ | 4’ | 4 | 4 | 4’ | 0 | 1 | 3 | 3 | 3 | 3 | 3 | 3’ | 0 |
| Priorität 2 | | 0 | 1 | 3 | 2 | 2 | 2’ | 2’ | 2 | 2’ | 2’ | 4’ | 0 | 4 | 4 | 4 | 4’ | 4 | 4 | 3’ |
| Priorität 3 | | | 0 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2’ | 4’ | 2 | 2’ | 2’ | 2’ | 2 | 2 | 4 |
| Priorität 4 | | | | 0 | 1 | 1 | 1 | 1’ | 1 | 1 | 1 | 3 | 2’ | 1 | 1 | 1’ | 1’ | 1 | 1 | 2 |
| Seitenfehler | X | X | X | X | X | | | | | | | X | X | X | | | | | | X |

**Anzahl Seitenfehler: 9**

## 2.4 LFU (least frequently used)

Die Seite mit der geringsten Nutzungshäufigkeit (in einem bestimmten Zeitraum) wird ausgetauscht. In unseren Aufgaben ist dieser Zeitraum meist vom Zeitpunkt des Ladens der Seite bis zum aktuellen Zeitpunkt zu betrachten. Dies steht meist auch nochmal explizit in der Aufgabenstellung drin.

Wenn zwei Seiten gleich oft genutzt wurden wird die ältere ausgelagert. Die Priorität ergibt sich also aus der Kombination der Zugriffsanzahl und dem Alter.

| Referenz $\omega$ | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Rahmen 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Rahmen 3 | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Rahmen 4 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Priorität 1 | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 4 | 4 | 3 | 2 | 2 | 4 | 3 | 3 | 3 |
| Priorität 2 | | 0 | 1 | 3 | 2 | 2 | 4 | 2 | 1 | 3 | 2 | 2 | 2 | 4 | 3 | 3 | 2 | 4 | 4 | 4 |
| Priorität 3 | | | 0 | 1 | 3 | 3 | 3 | 4 | 2 | 1 | 3 | 3 | 3 | 2 | 4 | 4 | 3 | 2 | 2 | 2 |
| Priorität 4 | | | | 0 | 1 | 1 | 1 | 3 | 4 | 4 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Zugriffe Seite 0 | 1 | 1 | 1 | 1 | | | | | | | | 1 | | | | | | | | 1 |
| Zugriffe Seite 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | | 1 | 1 | 1 | 2 | 2 | 2 | 2 | |
| Zugriffe Seite 2 | | | | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| Zugriffe Seite 3 | | | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 5 | 5 |
| Zugriffe Seite 4 | | | | | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 |
| jüngste Seite | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
| | | 0 | 1 | 3 | 2 | 2 | 4 | 2 | 1 | 3 | 2 | 4 | 4 | 1 | 3 | 2 | 1 | 4 | 4 | 3 |
| | | | 0 | 1 | 3 | 3 | 3 | 4 | 2 | 1 | 3 | 2 | 2 | 4 | 1 | 3 | 2 | 1 | 1 | 4 |
| älteste Seite | | | | 0 | 1 | 1 | 1 | 3 | 4 | 4 | 1 | 3 | 3 | 2 | 4 | 4 | 3 | 2 | 2 | 2 |
| Seitenfehler | X | X | X | X | X | | | | | | | X | X | | | | | | | X |

**Anzahl Seitenfehler: 8**

## 2.5 CLIMB

Bei dieser Strategie werden neu ankommende Seiten mit der niedrigsten Priorität eingereiht. Werden sie erneut aufgerufen steigen sie um einen Platz in der Prioritätsliste auf. Bei einem Seitenfehler wird immer die Seite mit der niedrigsten Priorität verdrängt.

| Referenz $\omega$ | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Rahmen 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Rahmen 3 | | | 3 | 3 | 3 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 3 | 2 | 2 | 2 | 3 | 3 | 3 |
| Rahmen 4 | | | | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Priorität 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Priorität 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 |
| Priorität 3 | | | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 4 | 4 | 0 | 0 | 3 | 0 |
| Priorität 4 | | | | 2 | 4 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 3 | 2 | 2 | 2 | 3 | 0 | 3 |
| Seitenfehler | X | X | X | X | X | | X | | X | X | X | | | X | | | | X | | |

**Anzahl Seitenfehler: 11**

## 2.6 OPT (Optimalstrategie)

Bei dieser Strategie wird die Seite, welche am längsten nicht mehr benötigt wird, ausgelagert. In der Realität kann man das natürlich nicht wissen, aber es gibt Wege dies zumindest abzuschätzen. Wenn solche Aufgaben auf Papier gestellt werden ist die Reihenfolge der Speicheranforderungen natürlich bekannt und kann somit genutzt werden.

| Referenz $\omega$ | 0 | 1 | 3 | 2 | 4 | 4 | 2 | 1 | 3 | 2 | 4 | 0 | 1 | 3 | 2 | 1 | 4 | 3 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Rahmen 2 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 |
| Rahmen 3 | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Rahmen 4 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Priorität 1 | 0 | 1 | 1 | 2 | 4 | 2 | 1 | 3 | 2 | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 3 | 3 | 0 | 2 |
| Priorität 2 | | 0 | 3 | 1 | 2 | 1 | 3 | 2 | 4 | 1 | 3 | 3 | 2 | 1 | 3 | 0 | 0 | 0 | 2 | 4 |
| Priorität 3 | | | 0 | 3 | 1 | 3 | 2 | 4 | 1 | 3 | 2 | 2 | 1 | 3 | 0 | 2 | 2 | 2 | 4 | 3 |
| Priorität 4 | | | | 0 | 3 | 4 | 4 | 1 | 3 | 2 | 4 | 0 | 0 | 0 | 2 | 1 | 4 | 4 | 3 | 0 |
| Seitenfehler | X | X | X | X | X | | | | | | | X | | | | | X | | | |

**Anzahl Seitenfehler: 7**

# 3 Nicht-Präemptives Prozess-Scheduling

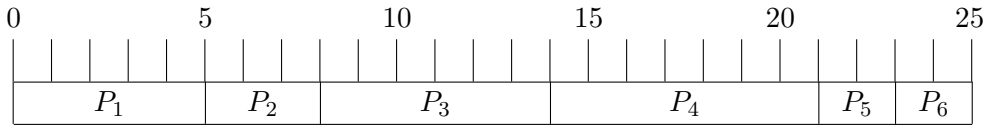
Beim nicht-präemptiven Prozess-Management müssen die Prozesse ihre Rechenzeit eigenständig an den Prozessor zurückgeben. Der Prozessor hat keine Möglichkeit einen laufenden Prozess zu unterbrechen. Ein Ziel ist es hierbei die mittlere Wartezeit von Prozessen, also die Zeit vom Eintreffen des Prozesses bis zur Erteilung von Rechenzeit, so gering wie möglich zu halten.

Als Beispiel nehmen wir eine CPU mit 6 Prozessen $P_1, ..., P_6$, welche zu den folgenden Zeitpunkten im System ankommen und die angegebene Bedienzeit benötigen:

| Prozess | Ankunftszeit | Bedienzeit |
|---|---|---|
| $P_1$ | 0 | 5 |
| $P_2$ | 1 | 3 |
| $P_3$ | 2 | 6 |
| $P_4$ | 4 | 7 |
| $P_5$ | 6 | 2 |
| $P_6$ | 21 | 2 |

## 3.1 FIFO / FCFS (first in, first out / first come, first serve)

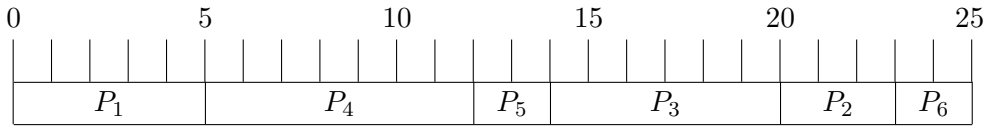
Bei diesem Verfahren gibt es eine Warteschlange für eintreffende Prozesse. Die Warteschlange wird nach dem FIFO-Prinzip abgearbeitet, also nach der Reihenfolge des Eintreffens. Jeder Prozess darf dabei seine gesamte Bedienzeit nutzen.




Mittlere Wartezeit: $\frac{0+4+6+10+15+2}{6} = 6,17$

## 3.2 LIFO (last in, first out)

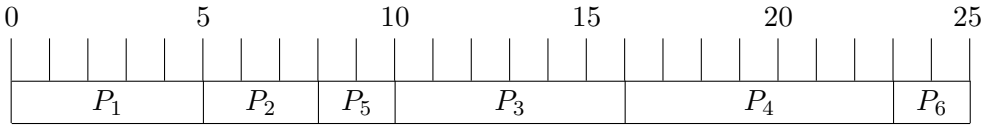
FIFO arbeitet im Prinzip wie FIFO, nur das statt einer Warteschlange ein Stack genutzt wird. Das bedeutet dass immer der zuletzt eingetroffene Prozess als nächstes bearbeitet wird.




Mittlere Wartezeit: $\frac{0+19+12+1+6+2}{6} = 6,67$

### 3.3 SPT / SJF (shortest processing time / shortest job first)

Für diese Strategie muss die Bedienzeit eines Prozesses vorhersehbar sein, denn es wird immer der wartende Prozess mit der geringsten Bedienzeit als nächstes ausgeführt. Diese Strategie minimiert unter den nicht-präemptiven Strategien die mittlere Wartezeit.



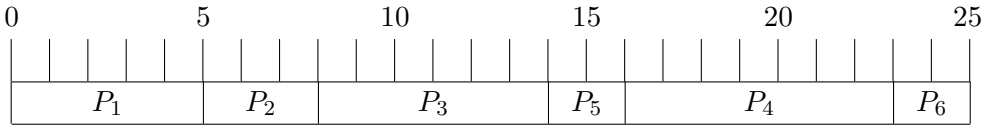

Mittlere Wartezeit: $\frac{0+4+2+8+12+2}{6} = 4,67$

### 3.4 HRN (highest response ratio next)

Um zu ermitteln welcher Prozess als nächstes ausgeführt wird wird für jeden Prozess die response ratio bestimmt:

$$r = \frac{Wartezeit+Bedienzeit}{Bedienzeit}$$

Der Prozess mit der größten response ratio wird ausgeführt. Die response ratio muss jedes Mal wenn ein Prozess endet für alle Prozesse neu bestimmt werden!



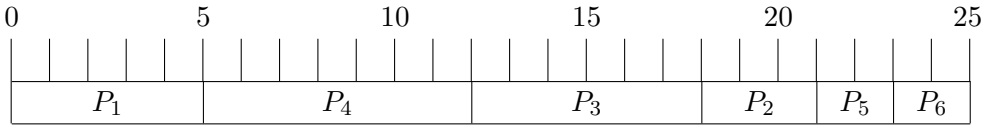

Mittlere Wartezeit: $\frac{0+4+6+8+12+2}{6} = 5,3$

Ein Rechenbeispiel zum Zeitpunkt t=5, nachdem $P_1$ seine Rechenzeit abgibt:

Während der laufzeit von $P_1$ sind weitere Prozesse eingetroffen.
Es warten jetzt auf Ausführung: $P_2, P_3$ und $P_4$.
$r_2 = \frac{4+5}{5} = \frac{9}{5} = 1,8$
$r_3 = \frac{3+6}{6} = \frac{9}{6} = 1,5$
$r_4 = \frac{1+7}{7} = \frac{8}{7} \approx 1,14$
Daraus folgt $r_2 > r_3 > r_4$, also wird $P_2$ als nächstes ausgeführt.

### 3.5 LPT (longest processing time)

Diese Strategie ist sozusagen das Gegenteil von SPT und bearbeitet zuerst Prozesse mit großer Bedienzeit. Dadurch wird die mittlere Wartezeit maximiert.




Mittlere Wartezeit: $\frac{0+17+10+1+15+2}{6} = 7,5$

# 4 Präemptives Prozess-Scheduling

Im Gegensatz zum nicht-präemptiven Scheduling kann hier das Betriebssystem die Kontrolle über den Prozessor eigenständig zurückerlangen, ohne dass der Prozess seine Rechenzeit selbsständig abgibt. Der Vorgang, wenn das Betriebssystem einem Prozess die Prozessorkontrolle entzieht und einem anderen erteilt, nennt sich Kontextwechsel. Solche Kontextwechsel kosten einiges an Rechenzeit, deshalb stellen häufige Kontextwechsel eine Verschwendung von Rechenzeit dar. Finden zu wenige Kontextwechsel statt ist jedoch die Antwortzeit der Prozesse schlecht und das Gefühl des Benutzers, das mehrere Prozesse zeitgleich ausgeführt werden, geht verloren. Hier muss die Strategie eine Balance finden.

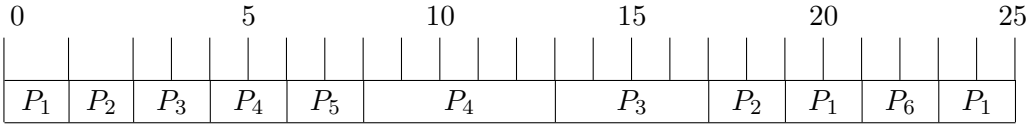
Wir nutzen zur Demonstration der Strategien das Beispiel aus Abschnitt 3. Außerdem vernachlässigen wir die für Kontextwechsel benötigte Zeit.

Für die Strategien EDF (4.4) und LLF (4.5) führen wir außerdem noch sogenannte Deadlines ein. Ein Prozess sollte möglichst vor seiner Deadline enden. Die Anzahl der Deadline-Verletzungen sollte also minimal sein.

| Prozess | Ankunftszeit | Bedienzeit | Deadline |
|---|---|---|---|
| $P_1$ | 0 | 5 | 7 |
| $P_2$ | 1 | 3 | 6 |
| $P_3$ | 2 | 6 | 11 |
| $P_4$ | 4 | 7 | 15 |
| $P_5$ | 6 | 2 | 10 |
| $P_6$ | 21 | 2 | 25 |

## 4.1 LIFO-PR (LIFO preemptive resume)

LIFO-PR funktioniert wie das nicht-präemptive LIFO aus Abschnitt 3.2, wenn jedoch ein neuer Job eintrifft wird der Aktuelle pausiert und der neue sofort ausgeführt

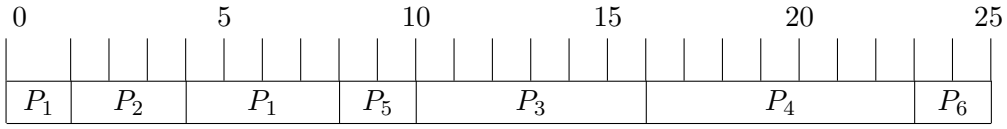



Mittlere Wartezeit: $\frac{20+15+9+2+0+0}{6} = 7,66$

Kontextwechsel: 10

## 4.2 SRPT (shortest remaining processing time)

SRPT ist die präemptive Variante von SPT aus Abschnitt 3.3. Es wird wie bei SPT der Prozess mit der kürzesten Bedienzeit bearbeitet. Trifft jedoch ein neuer Prozess ein, dessen Bedienzeit kürzer ist als die Restlaufzeit des aktuell Laufenden, so wird der laufende Prozess pausiert und der neue sofort ausgeführt.



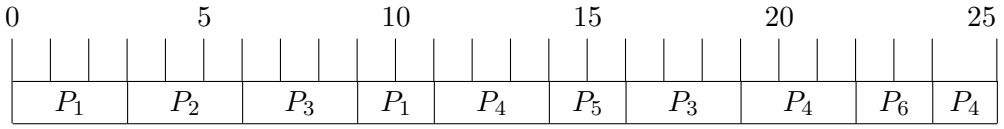

Mittlere Wartezeit: $\frac{3+0+8+12+2+2}{6} = 4,5$

Kontextwechsel: 6

## 4.3 Round Robin

Round Robin nutzt genau wie das nicht-präemptive FIFO aus Abschnitt 3.1 eine Warteschlange um zu entscheiden welcher Prozess als nächstes ausgeführt wird. Anders als bei FIFO dürfen die Prozesse jedoch nicht solange laufen wie sie möchten, sondern nur für eine bestimmte Zeit. Diese nennt man Quantum oder Zeitscheibe. Läuft die Zeitscheibe ab so wird dem Prozess die Kontrolle über den Prozessor entzogen und er wird hinten in die Warteschlange eingereiht. Das Quantum kann für alle Prozesse identisch sein, oder von Dingen wie zum Beispiel der Prozess-Priorität abhängen. Bei der Wahl des Quantums ist zu beachten: Wird es zu groß gewählt führt dies zu schlechten Antwortzeiten der Prozesse. Wird es zu klein gewählt geht viel Rechenzeit durch häufige Kontextwechsel verloren.

Für unser Beispiel sei das Quantum für alle Prozesse 3.

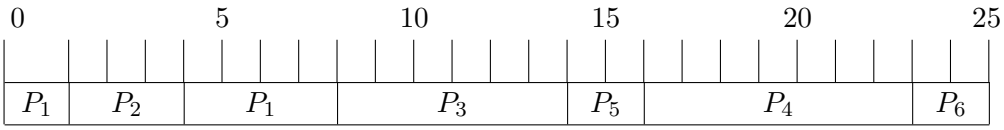



Mittlere Wartezeit: $\frac{6+0+11+14+8+1}{6} = 6,67$

Kontextwechsel: 9

## 4.4 EDF (earliest deadline first)

Diese Strategie versucht Deadline-Verletzungen möglichst zu vermeiden, indem immer der Prozess mit der frühsten Deadline ausgeführt wird. Der Prozess läuft dann solange bis er terminiert, oder ein anderer Prozess mit einer früheren Deadline eintrifft.




Mittlere Wartezeit: $\frac{3+0+6+12+8+2}{6} = 5,17$

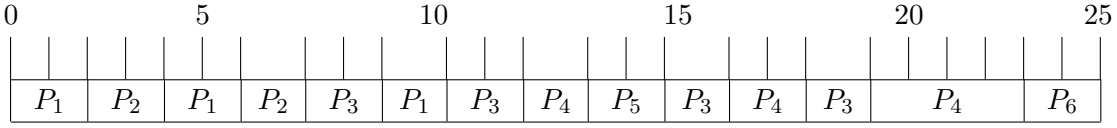
Kontextwechsel: 6

Deadline-Verletzungen: 4 ($P_1, P_3, P_4, P_5$)

## 4.5 LLF (least laxity first)

Die Idee bei LLF ist jeden Prozess so spät wie möglich auszuführen. „least laxity first“ lässt sich übersetzen als „der am wenigsten entspannte zuerst“. Eine hohe Priorität hat ein Prozess also, wenn die Differenz zwischen seiner Restlaufzeit und der verbleibenden Zeit bis zu seiner Deadline gering ist. Dieses Verfahren minimiert Deadline-Verletzungen, bringt jedoch einen hohen Rechenaufwand und viele Kontextwechsel mit sich, da die Werte laufend neu berechnet werden.

Für
d = Deadline des Prozesses
t = aktueller Zeitpunkt
r = Restlaufzeit des Prozesses
gilt: laxity = d - t - r
Dieser Wert muss möglichst niedrig sein, um in der Priorität aufzusteigen.




Mittlere Wartezeit: $\frac{5+3+11+12+7+2}{6} = 6,67$

Kontextwechsel: 13

Deadline-Verletzungen: 5 ($P_1, P_2, P_3, P_4, P_5$)

Dazu sei gesagt dass die Deadlines für dieses Beispiel wohl nicht gut gewählt waren, was zu dieser hohen Rate an Deadline-Verletzungen führt. Schön zu sehen sind jedoch die vielen Kontextwechsel.

# 5 Segementierung / Speicherallokation

Allokations-Strategien beschäftigen sich damit für eine Speicheranforderung einen möglichst gut passenden Bereich im Speicher zu finden, der noch frei ist. Ein Ziel ist es hierbei den Speicher möglichst nicht zu Fragmentieren, da dies dazu führen kann das zukünftige Speicher-Anforderungen nicht mehr bedient werden können, obwohl eigentlich noch genug freier Speicher vorhanden ist.

Als Beispiel nehmen wir die folgende Belegung eines 4096 Byte großen Speichers:



Für unsere Beispiele nehmen wir an, dass an den Speicher nacheinander Anforderungen der folgenden Größe gestellt werden: 124 Byte, 388 Byte, 640 Byte, 1024 Byte

Anmerkung: Die Größenverhältnisse sind für Speicherbereiche welche kleiner als 384 Byte groß sind nicht mehr maßstabsgetreu dargestellt.

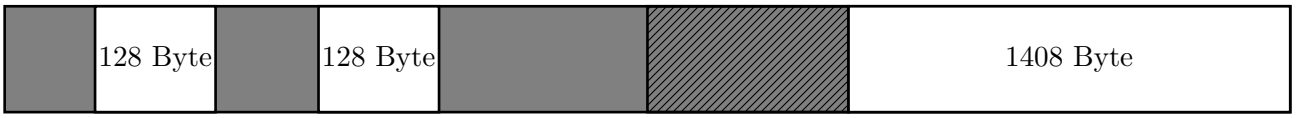
## 5.1 First-Fit

Bei dieser Strategie wird einfach der erste freie Bereich belegt, welcher groß genug ist.

Anforderung 1: 124 Byte


Anforderung 2: 388 Byte


Anforderung 3: 640 Byte




Anforderung 4: 1024 Byte

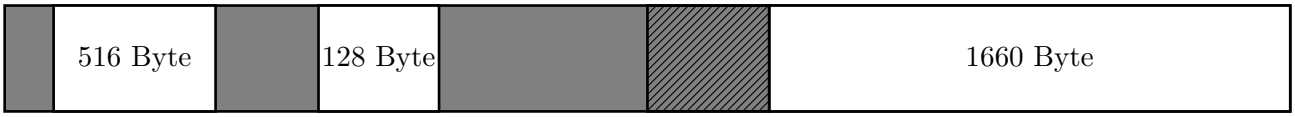
## 5.2 Rotating-First-Fit

Diese Strategie funktioniert im Prinzip genauso wie First-Fit, nur dass man nicht immer wieder von vorne anfängt zu suchen. Hat man in einem Durchlauf einen freien Speicherbereich gefunden und dort Platz alloziert, so beginnt die Suche beim nächsten Mal hinter dem freien Bereich von dem man ein Teil belegt hat. Dieses Prinzip wird bei Anforderung 2 im Beispiel nochmal besonders verdeutlicht.

Kommt man bei der Suche am Ende des Speichers an, so beginnt man wieder Anfang und sucht von da aus nochmal ob es einen passenden Speicherbereich gibt.

Anforderung 1: 124 Byte


Anforderung 2: 388 Byte




Anforderung 3: 640 Byte


Anforderung 4: 1024 Byte
Diese Anforderung kann nicht erfüllt werden.

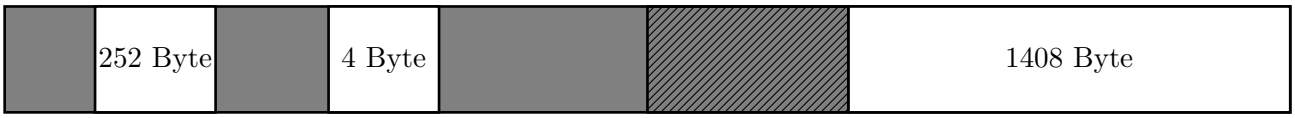
## 5.3 Best-Fit

Bei dieser Strategie wird der gesamte Speicher durchsucht und der kleinste Bereich ausgewählt der die Speicheranforderung nich erfüllt.

Anforderung 1: 124 Byte


Anforderung 2: 388 Byte


Anforderung 3: 640 Byte

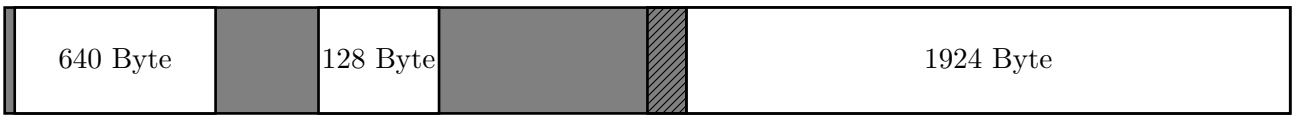



Anforderung 4: 1024 Byte

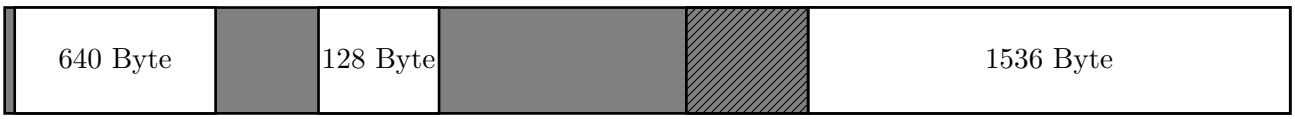

## 5.4 Worst-Fit

Dies ist das Gegenteil von Best-Fit: Es wird der größte Bereich belegt, wenn er ausreicht.

Anforderung 1: 124 Byte




Anforderung 2: 388 Byte




Anforderung 3: 640 Byte


Anforderung 4: 1024 Byte

Diese Anforderung kann nicht erfüllt werden.