Betriebssysteme

Skrift zur Vorlesung im Wintersemester 2020/2021

Prof. Dr. Claudia Linnhoff-Popien
Unter Mitarbeit von:
Ulrich Bareth, Michael Dür, Tim Furche, Matthias Hampel, Thomas Höhler, Carmen Jaron, Thomas Schaaf und Diana Weiß

Das Copyright der auf dem Titelblatt verwendeten Logos liegt bei der Ubuntu Foundation (http://www.ubuntu.com), der Firma Red Hat (http://fedoraproject.org), Debian (http://www.debian.org) und der Suse Linux GmbH

© Prof. Dr. Claudia Linnhoff-Popien – alle Rechte vorbehalten
# Inhaltsverzeichnis

## I Einführung  
1 Das Betriebssystem  
   1.1 Einordnung der Maschinensprache  
   1.2 Aufgaben des Betriebssystems  
   1.3 Geschichte der Betriebssysteme  
      1.3.1 1. Generation: 1945 bis 1955  
      1.3.2 2. Generation: 1955 bis 1965  
      1.3.3 3. Generation: 1965 bis 1980  
      1.3.4 4. Generation: seit 1980  
      1.3.5 5. Generation: seit ca. 2000  
   1.4 Arten von Betriebssystemen  
2 Programme und Unterprogramme  
   2.1 Vom Programm zum Maschinenprogramm  
   2.2 Unterprogramme und Prozeduren  
      2.2.1 Die Befehle CALL und RET  
      2.2.2 Schema für Unterprogrammaufrufe  
      2.2.3 Module  
   2.3 Realisierung eines Unterprogrammaufrufs  
   2.4 Rekursive Prozeduraufüße  
3 Prozesse  
   3.1 Das Prozess-Konzept  
      3.1.1 Grundlagen von Prozessen  
      3.1.2 Erzeugung von Prozessen  
      3.1.3 Realisierung von Multiprogramming  
      3.1.4 Das 2-Zustands-Prozessmodell  
      3.1.5 Das 5-Zustands-Prozessmodell  
      3.1.6 Das 7-Zustands-Prozessmodell  

II Prozesse  

<table>
<thead>
<tr>
<th>Kapitel</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Programme und Unterprogramme</td>
</tr>
<tr>
<td>2.1</td>
<td>Vom Programm zum Maschinenprogramm</td>
</tr>
<tr>
<td>2.2</td>
<td>Unterprogramme und Prozeduren</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Die Befehle CALL und RET</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Schema für Unterprogrammaufrufe</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Module</td>
</tr>
<tr>
<td>2.3</td>
<td>Realisierung eines Unterprogrammaufrufs</td>
</tr>
<tr>
<td>2.4</td>
<td>Rekursive Prozeduraufüße</td>
</tr>
<tr>
<td>3</td>
<td>Prozesse</td>
</tr>
<tr>
<td>3.1</td>
<td>Das Prozess-Konzept</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Grundlagen von Prozessen</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Erzeugung von Prozessen</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Realisierung von Multiprogramming</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Das 2-Zustands-Prozessmodell</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Das 5-Zustands-Prozessmodell</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Das 7-Zustands-Prozessmodell</td>
</tr>
</tbody>
</table>
### INHALTSVERZEICHNIS

#### 3.2 Prozessbeschreibung
- 3.2.1 Kontrollstrukturen des Betriebssystems ........................................... 42
- 3.2.2 Prozesskontrollstrukturen ................................................................. 43
- 3.2.3 Zusammenfassung der Verwaltung und Beschreibung von Prozessen .......... 49

#### 3.3 Prozesskontrolle
- 3.3.1 Prozesswechsel (Kontext-Switch) ....................................................... 51
- 3.3.2 Unterbrechungen .................................................................................. 52
- 3.3.3 Moduswechsel ..................................................................................... 53
- 3.3.4 Konflikte bei Unterbrechungen ............................................................. 54
- 3.3.5 Ausführung des Betriebssystems ......................................................... 55

#### 4 Threads
- 4.1 Multithreading ....................................................................................... 56
- 4.2 Threadzustände ....................................................................................... 57
- 4.3 User-level-Threads (ULT) ....................................................................... 58
- 4.4 Kernel-level-Threads (KLT) .................................................................... 59
- 4.5 Kombinierte Konzepte ........................................................................... 60
- 4.6 Andere Formen paralleler Abläufe ........................................................ 61

#### 5 Scheduling
- 5.1 Das Prinzip des Schedulings ................................................................... 62
  - 5.1.1 Varianten des Schedulings ................................................................. 63
  - 5.1.2 Anforderungen an einen Scheduling-Algorithmus ......................... 64
  - 5.1.3 Scheduling vs. Dispatching .............................................................. 65
- 5.2 Scheduling-Algorithmen ......................................................................... 66
  - 5.2.1 Begriffe ............................................................................................ 67
  - 5.2.2 Nicht-preemptive Scheduling-Algorithmen ...................................... 68
  - 5.2.3 Preemptive Scheduling-Algorithmen ............................................... 69
  - 5.2.4 Priority Scheduling (PS) ................................................................. 70
  - 5.2.5 Multilevel Feedback Queueing ....................................................... 71
- 5.3 Prozesswechsel ....................................................................................... 72
- 5.4 Arten des Schedulings ............................................................................ 73

#### III Multiprocessing

#### 6 Deadlocks bei Prozessen
- 6.1 Motivation der Deadlocks anhand zweier Beispiele ............................... 74
- 6.2 Das Prinzip der Deadlocks .................................................................... 75
- 6.3 Deadlock Prevention ............................................................................. 76
  - 6.3.1 Deadlock Avoidance ....................................................................... 77
  - 6.3.2 Petri-Netze zur Prozeßmodellierung ................................................. 78
  - 6.3.3 Markierungen .................................................................................... 79
  - 6.3.4 Modellierung von nebenläufigen Prozessen ..................................... 80
  - 6.3.5 Deadlock Detection (Deadlockerkennung) ..................................... 81
## 7 Prozesskoordination

7.1 Nebenläufigkeit von Prozessen .................................................. 124
7.2 Kritische Bereiche ..................................................................... 125
  7.2.1 Erzeuger/Verbraucher-Problem ............................................ 125
7.3 Wechselseitiger Ausschluß ......................................................... 127
  7.3.1 Softwarelösungen für den wechselseitigen Ausschluß .......... 128
  7.3.2 Hardwarelösungen für den wechselseitigen Ausschluß ......... 134
7.4 Semaphore ................................................................................. 137
  7.4.1 Das Prinzip der Semaphore .................................................. 137
  7.4.2 Ablaufsteuerung mit Hilfe von Semaphoren ....................... 139
  7.4.3 Lösung des Erzeuger/Verbraucher-Problems mit Hilfe von Semaphoren 140
  7.4.4 Lösung für das Leser/Schreiber-Problem mit Hilfe von Semaphoren 141
  7.4.5 Das Philosophenproblem ..................................................... 142
7.5 Monitore ...................................................................................... 144
  7.5.1 Motivation der Monitore ..................................................... 144
  7.5.2 Prinzip der Monitore .......................................................... 146
7.6 Message Passing ........................................................................ 150
  7.6.1 Blockierung ........................................................................ 150
  7.6.2 Adressierung ...................................................................... 151

## IV Ressourcenverwaltung

8 Speicher ........................................................................................ 157
  8.1 Speicherverwaltung ................................................................. 158
  8.2 Speicherpartitionierung ............................................................ 159
    8.2.1 Feste Partitionierung ......................................................... 159
    8.2.2 Dynamische Partitionierung .............................................. 161
    8.2.3 Buddy-Systeme .............................................................. 162
  8.3 Virtueller Speicher .................................................................. 164
    8.3.1 Prinzip der Speicherverwaltung ....................................... 165
    8.3.2 Datentransport zwischen Hintergrund- und Arbeitsspeicher 166
    8.3.3 Abbildung virtueller auf reale Adressen ............................ 167
  8.4 Paging ....................................................................................... 169
    8.4.1 Paging-Strategien ............................................................ 169
    8.4.2 Seitenaustauschalgorithmen ............................................. 170
    8.4.3 Minimierung von Seitenfehlern ....................................... 175
    8.4.4 Working Set Strategie ...................................................... 178
  8.5 Segmentierungsstrategien ....................................................... 181

9 E/A Verwaltung ............................................................................. 185
  9.1 Klassifizierung von E/A-Geräten ........................................... 186
  9.2 E/A Techniken ......................................................................... 186
INHALTSVERZEICHNIS

V Interprozeßkommunikation 189

10 Lokale Interprozeßkommunikation 191
  10.1 Grundlagen des Nachrichtenaustauschs 192
  10.2 Pipes 195
  10.3 FIFOs 200
  10.4 Stream Pipes 200
  10.5 Sockets 201

11 Verteilte Systeme 203
  11.1 Einführung in Verteilte Systeme 204
    11.1.1 Historie Verteilter Systeme 204
    11.1.2 Vorteile Verteilter Systeme 205
    11.1.3 Klassifikation Verteilter Systeme 206
    11.1.4 Eigenschaften Verteilter Systeme 209
  11.2 Kommunikation in Verteilten Systemen 211
    11.2.1 Das Client/Server Modell 211
    11.2.2 Der Remote Procedure Call 218
    11.2.3 Kommunikation in Verteilten Systemen 229
1.1 Logische Hierarchie in einem Rechner ........................................... 4
1.2 Erste Stapelverarbeitungssysteme .................................................. 7
1.3 Erste Betriebssysteme: Kartenstapel zur Ausführung eines Jobs .......... 8

2.1 Sprünge bei rekursiven Unterprogrammaufrufen .............................. 25

3.1 Ressourcennutzung bei sequenzieller Ausführung von Jobs ................ 30
3.2 Pseudo-parallele Ausführung .......................................................... 31
3.3 Ressourcennutzung bei verzahnter Ausführung von Jobs .................. 32
3.4 FORK-Darstellung ........................................................................... 32
3.5 Beispiel einer Prozesshierarchie: A hat die beiden Kindprozesse B und C erzeugt, B wiederum die drei Kinder D, E und F .................................... 33
3.6 Speicherbelegung der drei Beispielprozesse ...................................... 35
3.7 2-Zustands-Prozessmodell ............................................................... 37
3.8 Warteschlangenmodell des 2-Zustand-Prozessmodells ....................... 37
3.9 5-Zustands-Prozessmodell ............................................................... 38
3.10 Warteschlangenmodell des 5-Zustand-Prozessmodells ..................... 39
3.11 Implementierung mit mehreren Warteschlangen ............................... 40
3.12 5-Zustands-Modell mit Suspend ..................................................... 41
3.13 7-Zustands-Prozessmodell ............................................................. 42
3.14 Verwaltung der Nutzung von Systemressourcen .............................. 43
3.15 Struktur der Prozesstabelle ............................................................ 46
3.16 Beispiel: Speicherbelegung bei CTSS ............................................. 47
3.17 Struktur des Pentium-EFLAGS-Registers ...................................... 48
3.18 Struktur eines Prozesses im Hintergrundspeicher ............................. 49
3.19 Implementierung des 5-Zustands-Prozessmodells ............................. 49
3.20 Blockierende und nicht-blockierende Realisierung von Signalen ......... 54
3.21 Prinzip eines Interrupts der CPU durch einen E/A-Kanal .................. 55
3.22 Prinzip einer Exception .................................................................. 56
3.23 Prozess- und Moduswechsel ........................................................... 56
3.24 Beispiel für einen Unterbrechungskonflikt ...................................... 57
3.25 Ausführung des Betriebssystem-Kerns außerhalb jeden Prozesses ...... 58
3.26 Ausführung des Betriebssystems durch die Prozesswechseloperationen und Betriebssystem-Routinen ................................................................. 58
3.27 Prozeß-Image bei Integration der BS-Funktionen ........................................ 59
3.28 Implementierung des Betriebssystems als eine Sammlung von Systemprozessen 59
3.29 9-Zustands-Modell (UNIX) ........................................................................... 60

4.1 Die 4 Fälle beim Zusammenspiel von Threads und Prozessen .............................. 65
4.2 Singlethreading-Prozessmodell vs. Multithreading-Prozessmodell ....................... 66
4.3 Thread-Zustandsübergangsdiagramm ................................................................. 67
4.4 User-level-Thread-Modell .................................................................................. 68
4.5 User-level-Thread-Modell 2 .............................................................................. 69
4.6 Kernel-level-Thread-Modell ................................................................................ 70
4.7 Thread-Modell für kombinierte Konzepte ........................................................... 71
4.8 Einordnung von “Parallele Prozessoren” ............................................................ 72
4.9 Parallele Prozessoren ......................................................................................... 73
4.10 Prinzip der SMP-Organisation .......................................................................... 73

5.1 Aufspaltung in mehrere Ready-Queues für Prioritäten ........................................... 78
5.2 Beispiel mittlere Verweildauer für FCFS .............................................................. 81
5.3 Beispiel mittlere Verweildauer für SJF .................................................................. 81
5.4 Programmbefehle .................................................................................................. 83
5.5 Beispiel mittlere Verweildauer für SRPT .............................................................. 84
5.6 Realisierung des Round Robin Verfahrens ............................................................ 85
5.7 Beispiel mittlere Verweildauer für RR ................................................................. 85
5.8 Belegung der Ready-Queue zu dem Beispiel aus Abbildung 5.7 ......................... 86
5.9 Beispiel mittlere Verweildauer für RR ................................................................. 87
5.10 Implementierung und Abarbeitung eines Auftrags .............................................. 89
5.11 Zuordnung der Prozesszustände zu den Arten des Scheduling ......................... 90
5.12 Visualisierung der hierarchischen Unterteilung der Scheduling-Arten ................ 91

6.1 Illustration eines Deadlock ................................................................................. 96
6.2 Beispiel Philosophenproblem .............................................................................. 97
6.3 Beispiel mit 4 Prozessen. Dicke Pfeile deuten an, dass ein Prozess ein Betriebs-
    mittel hält, dünne Pfeile, dass ein Prozess auf ein Betriebsmittel wartet ............... 98
6.4 Prozessfortschrittsdiagramm 1 ............................................................................ 99
6.5 Prozessfortschrittsdiagramm 2 .......................................................................... 100
6.6 Bsp. Zustand ohne Deadlock ............................................................................. 100
6.7 Circular Wait ....................................................................................................... 102
6.8 Beispiel eines sicheren Zustandes ..................................................................... 105
6.9 Beispiel eines unsicheren Zustandes .................................................................. 106
6.10 Beispiel für ein Petri-Netz ................................................................................ 107
6.11 Stellen-Transitionssysteme ................................................................................ 108
6.12 Philosophenproblem mit 5 Philosophen ........................................................... 109
6.13 Petri-Netz zum Philosophenproblem mit 5 Philosophen ................................. 110
6.14 Stellen-Transitionssystem mit Markierungen .................................................... 111
6.15 Petri-Netz mit Markierung und Folgemarkierung ............................................. 112
6.16 Petri-Netz mit Markierung und Gewichtung ..................................................... 113
6.17 Petri-Netz zur Modellierung des Erzeuger/Verbraucher-Problems .................... 114
6.18 R/W-Problem ................................................................................................... 114
ABBILDUNGVERZEICHNIS

6.19 Einfaches Petri-Netz zum R/W-Problem .......................................................... 115
6.20 Fertiges Petri-Netz zum R/W-Problem .............................................................. 116
6.21 Erreichbarkeitsgraph zum R/W-Problem ........................................................... 117
6.22 Vereinfachter Erreichbarkeitsgraph zum R/W-Problem ................................. 117
6.23 Beispiel eines Deadlocks .................................................................................. 118
6.24 Erreichbarkeitsgraph zu Abbildung 6.17 ............................................................ 119
6.25 Entstehen eines Partial Deadlocks ................................................................. 120
6.26 Entstehen eines Deadlocks .............................................................................. 121

7.1 Ungeschützter Zugriff auf kritische Daten ......................................................... 126
7.2 Darstellung eines einfachen Ringpuffers ............................................................ 126
7.3 Gemeinsam genutzte Variable, die anzeigt, welcher Prozess in den kritischen Bereich eintreten darf ............................................................. 128
7.4 Jeder Prozess erhält eine Variable "flag", die den Anspruch auf Eintritt in den kritischen Bereich zeigt .......................................................... 130
7.5 Die korrekte Lösung: Schiedsrichteriglo mit Variable turn zeigt an, welcher Prozess Vorrang hat ................................................................. 132
7.6 Ablaufsteuerung von zwei Prozessen mit Hilfe von Semaphoren .................... 139
7.7 Das Philosophenproblem .................................................................................. 143
7.8 Monitor ............................................................................................................... 147
7.9 Realisierung mit/ohne Blockierung und Pufferung ........................................ 151
7.10 Realisierung mit/ohne Blockierung ................................................................. 151
7.11 Indirekte Adressierung n:1 ............................................................................. 152
7.12 Indirekte Adressierung n:m ........................................................................... 153

8.1 Beispiel für feste Partitionierung .................................................................... 160
8.2 Technik der fixen Partitionierung .................................................................... 161
8.3 Dynamische Partitionierung ............................................................................. 162
8.4 Beispiel für Buddy-Systeme ............................................................................ 163
8.5 Adressierung in Buddy-Systemen ..................................................................... 163
8.6 Fragmentierung bei Buddy-Systemen ............................................................... 164
8.7 Veranschaulichung des Prinzips gewichteter Buddy Systeme ....................... 165
8.8 Abbildung zwischen frames und pages ............................................................ 167
8.9 MI-Maschinenadresse ...................................................................................... 167
8.10 MI-Programmadresse ...................................................................................... 168
8.11 Beispiel zur Clock-Strategie .......................................................................... 174
8.12 Lifetime-Function L(m) .................................................................................. 179
8.13 Beispiele für verschiedene Segmentierungstaktik ........................................... 182

10.1 Grundlegende Verbindungstypen ................................................................. 192
10.2 Send–Receive-Kommunikation (asynchron) .................................................... 194
10.3 Receive–Send-Kommunikation (asynchron) .................................................... 194
10.4 Synchrone Kommunikation ............................................................................ 195
10.5 Einrichtung einer Pipe im Kernel .................................................................... 197
10.6 Grundsätzlicher Aufbau einer Pipekonstellation ........................................... 197
10.7 Daten fließen zum Kindprozeß ..................................................................... 198
10.8 Daten fließen zum Elternprozeß .................................................................... 199
Literaturverzeichnis


Teil I

Einführung
Das Betriebssystem

- Einordnung der Maschinensprache
- Betriebssysteme und ihre Aufgaben
- Geschichte der Betriebssysteme
- Arten von Betriebssystemen

Inhaltsangabe

1.1 Einordnung der Maschinensprache ........................................... 4
1.2 Aufgaben des Betriebssystems .................................................. 5
1.3 Geschichte der Betriebssysteme ................................................. 6
  1.3.1 1. Generation: 1945 bis 1955 ........................................... 6
  1.3.2 2. Generation: 1955 bis 1965 ........................................... 6
  1.3.3 3. Generation: 1965 bis 1980 ........................................... 7
  1.3.4 4. Generation: seit 1980 ................................................. 9
  1.3.5 5. Generation: seit ca. 2000 ........................................... 9
1.4 Arten von Betriebssystemen .................................................... 10
1.1 Einordnung der Maschinensprache

In der Vorlesung “Rechnerarchitekturen” wurde die Arbeitsweise der Rechnerhardware betrachtet und darauf aufbauend dargestellt, wie sich Programme auf Maschinenenebene schreiben lassen. In dieser Vorlesung legen wir den Schwerpunkt auf das Betriebssystem: Wir betrachten, wie es Programme ausführt und welche Dienste es dem Nutzer zur Verfügung stellt. Dabei werden die eingeführten Konzepte wieder verwendet.

Logisch kann man die Hierarchie der Funktionen eines Rechners wie in Abbildung 1.1 darstellen.

<table>
<thead>
<tr>
<th>Anwendungsprogramme</th>
<th>Systemprogramme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Betriebssystem</td>
<td></td>
</tr>
<tr>
<td>Maschinensprache</td>
<td></td>
</tr>
<tr>
<td>Mikroprogrammierung</td>
<td></td>
</tr>
<tr>
<td>Physikalische Geräte</td>
<td></td>
</tr>
</tbody>
</table>

Abbildung 1.1: Logische Hierarchie in einem Rechner

Bislang wurde die Ebene der Maschinensprache betrachtet, die kein Bestandteil der Hardware, sondern abhängig von dem jeweils verwendeten Assembler ist. Darunter liegt eine einfache Softwareschicht (Mikroprogrammierung), die unmittelbar auf physische Komponenten der Hardware einwirkt und eine sehr unkomplizierte Schnittstelle zur Maschinensprache besitzt. Diese Software interpretiert einzelne Maschinenanweisungen wie ADD, MOVE und JUMP in einzelnen Schritten. Um dabei z.B. einen Befehl wie ADD auszuführen, benötigt das Mikroprogramm die Adressen der Speicherzellen, aus denen gelesen bzw. in die geschrieben werden soll. Die Menge aller Befehle, die das Mikroprogramm interpretiert, definiert den Befehlsvorrat der Maschinensprache. Um in dieser Schicht beispielsweise Ein- und Ausgabegeräte ansprechen zu können, müssen die Parameter in Geräteregister geladen werden.

**Beispiel 1.1** (Diskettenzugriff). Um eine Datei von Diskette zu laden müssen (unter anderem) folgende Schritte erfolgen:

1. Falls nötig Starten des Laufwerksmotors.
2. Positionieren des Lesekopfes über der Spur, die das Verzeichnis (Directory) enthält.
4. Suchen der Dateiinformationen im Verzeichnis (Dateianfang).
5. Positionieren des Lesekopfes über der entsprechenden Spur.

Für jeden Befehl an den Laufwerkscontroller sind hier (neben vielen anderen) die folgenden Werte in die jeweiligen Geräteregister zu laden:
Aufgaben des Betriebssystems 5

– Die Adresse der Spur,
– die Adresse der Hauptspeicherzelle,
– die Anzahl der zu übertragenden Bytes und
– der Befehlscode (Lesen oder Schreiben).

1.2 Aufgaben des Betriebssystems

Das Betriebssystem soll diese Komplexität nun mindern, es soll also dem Programmierer einen Befehlsatz bereitstellen, der einfach zu beherrschen ist und die internen Details vor dem Nutzer verbirgt.

**Aufgaben des Betriebssystems:** Das Betriebssystem hat im Wesentlichen die drei folgenden Aufgaben:

1. Das Betriebssystem ist eine **erweiterte Maschine**, die leichter zu programmieren ist als die darunter liegenden Schichten. Beispielsweise wird die Festplatte nicht mehr durch Sektoren repräsentiert, sondern als Sammlung von Dateien dargestellt, auf die der Benutzer als Einheit zugreifen kann.

2. Das Betriebssystem ist ein **Ressourcenmanager**. Es verwaltet die vorhandenen Systemressourcen wie z.B. Speicher, die CPU oder Ein- und Ausgabegeräte und vermittelt zwischen den sich gegenseitig beeinflussenden Anforderungen der verschiedenen Programme und Nutzer. Für das **Multiplexing**, also das Verteilen von Ressourcen auf zwei oder mehr Programme bzw. Nutzer, gibt es zwei Ansätze:
   – Beim **Time Multiplexing** wird die Ressource zeitlich verzahnt immer vollständig einem Programm zur Verfügung gestellt (z.B. Prozessor).
   – Beim **Space Multiplexing** hingegen teilen sich mehrere Programme die Ressource gleichzeitig (z.B. Hauptspeicher).

Die Aufgabe des Betriebssystems ist es, beschränkte Ressourcen so zuzuteilen, dass alle Programme ihren Auftrag korrekt ausführen können.

3. Das Betriebssystem dient als **Kontrollinstanz** für jegliche Zugriffe. Unerlaubte Zugriffe auf Hardware oder fremde Daten unterbindet es.

Das Betriebssystem hat die Aufgaben der Komplexitätsreduktion, Ressourcenverwaltung und Zugriffskontrolle und fungiert als Abstraktionsschicht zwischen Hardware und Anwendungsprogrammen.

Kapitel 1. Das Betriebssystem

Systemaufrufe: Die Schnittstelle zwischen Anwendungsprogrammen und Betriebssystem wird durch Systemaufrufe definiert. Ein Anwendungsprogramm, das also (da es im Nutzermodus läuft) nicht direkt auf eine Ressource zugreifen darf, tätigt dazu einen Systemaufruf mit den entsprechenden Parametern.

In ihren konkreten Architekturen unterscheiden sich verschiedene Betriebssysteme z.T. sehr grundlegend. Allerdings gibt es gemeinsame Basiskonzepte, die in nahezu jedem Betriebssystem zu finden sind. Der Schlüsselbegriff in allen modernen Betriebssystemen ist der Prozess.

1.3 Geschichte der Betriebssysteme


Basierend auf diesen frühen Erfindungen entstanden bis heute vier Generationen von Computern, die im Folgenden genauer betrachtet werden sollen.

1.3.1 1. Generation: 1945 bis 1955 (Röhren und Klinkenfelder)


− Programme wurden über Klinkenfelder gesteckt oder in Maschinencode eingegeben.

− Programmiersprachen, Assemblersprachen und Betriebssysteme waren unbekannt.

− Anfang der 50er Jahre wurde das Stecken der Klinkenfelder durch Lochkarten abgelöst.

1.3.2 2. Generation: 1955 bis 1965 (Transistoren und Stapelsysteme)

− Die Einführung von Transistoren in der Mitte der 50er Jahre erhöhte die Zuverlässigkeit der Systeme um ein Vielfaches, die Mainframe genannten Systeme waren aber extrem teuer (eine Anlage kostete mehrere Millionen Dollar).

− Programme wurden in Lochkarten gestanzt und durch Bedienpersonal eingelegt. Erst nach Abschluss der Berechnung (und Ausdruck des Ergebnisses) konnte vom Bedienpersonal das nächste Programm eingelegt werden, was zu großen Leerlaufzeiten führte.

− Dieser Mangel führte zur Entwicklung der Stapelverarbeitungssysteme (Batchsysteme) (siehe Abbildung 1.2):

Aufträge wurden gesammelt und auf einfachen Rechnern auf ein Magnetband über-
Abbildung 1.2: Erste Stapelverarbeitungssysteme

spielt. Der Mainframe las die einzelnen Jobs (mit Hilfe eines rudimentären Betriebssystems) von diesem Band und speicherte die Ergebnisse auf einem anderen. Das Ausdrucken der Ergebnisse erfolgte wieder an einfachen Rechnern.

– Aufbau eines Programmstapels:
  • Ein Auftrag begann mit einer $JOB-Karte, auf der die maximale Laufzeit in Minuten, der Abrechnungsnummer und der Name des Programmierers codiert waren.
  • Danach folgte eine $FORTRAN-Karte, die das Betriebssystem beauftragte, den FORTRAN-Compiler vom Band zu laden.
  • Darauf folgten das zu übersetzende Programm und eine $LOAD-Karte, die das Betriebssystem veranlasste, das – in übersetzter Form auf ein Hilfsband geschriebene – Programm zu laden.
  • Die folgende $RUN-Karte veranlasste das Betriebssystem, das Programm mit nachfolgenden Daten zu bearbeiten.
  • Den Abschluss bildete eine $END-Karte, die die Programmbearbeitung beendete.

– Erste Betriebssysteme:
  FMS (Fortran Monitor System)
  IBSYS (IBM Betriebssystem für die 7094)

– Das Hauptproblem dieser Betriebssystemgeneration lag in der Interaktion: Jede I/O-Operation unterbrach die CPU.

1.3.3 3. Generation: 1965 bis 1980 (Integrierte Bauelemente und Multiprogramming)

– Anfang der 60er Jahre hatten sich zwei Produktlinien entwickelt:
  • Großrechner für wissenschaftlich/technischen Bereich
  • Rechner für den kommerziellen Bereich

– Da die Inkompatibilität dieser beiden Linien sehr hohe Kosten verursachte, schuf IBM eine Serie von Rechnern, das System /360, das sowohl für den wissenschaftlichen als auch für den kommerziellen Bereich konzipiert wurde.
Abbildung 1.3: Erste Betriebssysteme: Kartenstapel zur Ausführung eines Jobs

- Der /360 war die erste größere Computerfamilie, die integrierte Bauelemente (integrated circuits, ICs) verwendete und so ein besseres Preis-/Leistungsverhältnis erreichte.

- Das Betriebssystem des /360 war sehr kompliziert (mehrere Millionen Assembler-Zeilen von tausenden Programmierern), was zu vielen Fehlern führte.

- Fortschritte gegenüber der zweiten Generation:
  
  1. **Spooling** (Simultaneous Peripheral Operation On Line):
     Jobs wurden auf Vorrat von Karten eingelesen und auf einer Festplatte gespeichert. Terminierte ein Auftrag, so konnte das Betriebssystem einen neuen Auftrag von der Platte in einen jetzt leeren Speicherbereich laden und ihn dort bearbeiten.
  
  2. **Mehrprogrammbetrieb** (Multiprogramming):
     - **Bislang:** Bei Zugriffen auf Bandgeräte oder E/A-Geräte blieb die CPU untätig, was bei vielen kleinen Jobs die Auslastung der CPU stark verschlechterte.
     - **Nun:** Hauptspeicheraufteilung mit je einem Auftrag in jedem Speicherabschnitt (Space Multiplexing). Wartete der aktive Auftrag auf eine I/O-Operation, so konnte ein anderer Auftrag ausgeführt werden.


- Die Entwickler von CTSS entwickelten 1965 **MULTICS** (Multiplexed Information and Computing Service), ein Timesharing-System, dessen Grundidee vergleichbar ist mit der
Stromversorgung: bei Bedarf steckt man einen Stecker in die Dose und nimmt sich die Energie/Kapazität, die man braucht. Trotz seiner großen wissenschaftlichen Bedeutung wurde MULTICS kein großer wirtschaftlicher Erfolg, nur in wenigen Firmen wurde es tatsächlich eingesetzt.

– Einer der Programmierer von MULTICS, Ken Thompson, entwickelte eine vereinfachte Einbenutzerversion namens UNICS (Uniplexed Information and Computing Service).

– Dennis Ritchie implementierte dieses Einbenutzerbetriebssystem in C neu. UNIX, das Ergebnis seiner Arbeit, wurde kostenlos an Universitäten lizenziert und kam so schnell zu großer Verbreitung auf einer Vielzahl von Computersystemen.

1.3.4 4. Generation: seit 1980 (PCs)

– Die Entwicklung von LSI (Large Scale Integration) Schaltkreisen (Bauelemente mit Tau-
senden von Transistoren auf 1 cm² Silizium) machte eine hohe Rechenleistung zu ver-
gleichweise geringen Kosten verfügbar und läutete damit das PC-Zeitalter ein. Als Be-
triebssystem für PCs setzten sich MS-DOS (Microsoft Disk Operating System) und (bei
den größeren Systemen) UNIX durch.

– Die Erfindung der GUI (Graphical User Interface) schließlich erschloss dem PC (allen
voran dem Apple Macintosh) eine völlig neue Käuferschicht: Anwender, die ein benut-
zerfreundliches System wünschen und von Hardware nichts verstehen (wollen). Bei-
spiele für aktuelle Betriebssysteme dieser “nächsten Generation” sind z.B. Microsoft
Windows XP, Mac OS X und Linux mit KDE.

– Mitte der 80er Jahre verbreiteten sich Netzwerke aus PCs immer weiter. Betriebssysteme
für diese Rechner lassen sich in zwei Kategorien unterteilen:

  •  Bei einem Netzwerk-Betriebssystem (Network Operating System) weiß der Be-
     nutzer um die Existenz mehrerer Rechner und kann sich gezielt auf fremden Rech-
     nern anmelden (etwa um Daten zu kopieren). Dabei läuft auf jedem Rechner ein
     eigenes Betriebssystem (meist mit eigener Benutzerverwaltung).

  •  Ein verteiltes Betriebssystem (Distributed Operating System) hingegen soll dem
     Benutzer wie ein traditionelles Einprozessorsystem erscheinen, obwohl es auf meh-
     reren (meist räumlich getrennten) Rechnern abläuft. In einem echten verteilten
     System wissen Nutzer nicht, wo Programme ablaufen und wo Daten liegen – dies
     wird vom Betriebssystem verwaltet und bleibt dem Nutzer verborgen.

1.3.5 5. Generation: seit ca. 2000 (mobile Betriebssysteme)

– Als erstes Smartphone wird der Simon Personal Computer von BellSouth und IBM be-
trachtet (ab 1994). Symbian bis 2006 Marktführer mit 76%. Alternativen waren Win-
dows Mobile, BlackBerry OS und Palm OS.

– Ab der Einführung des iPhones im Jahr 2007 erfolgte Umstieg auf Touch Screens. Es
entstanden Android, Palm webOS, und Windows Phone 7.

– Mobile Endgeräte besitzen begrenzte Energiereserven, die geschont werden müssen.
Energiesparfunktionen und effizienter Zugriff auf die Luftschmittstelle sind nötig.
Kapitel 1. Das Betriebssystem


- Herstellerbindung: Android, Windows Phone, Symbian und Firefox nicht an Gerätehersteller gebunden. iOS und Blackberry nur auf Apple und BlackBerry Geräten.

- Sicherheit: Bezug der Anwendungen i.d.R. über einen Distributionskanal des Betriebssystemherstellers. iOS und Blackberry setzen auf zusätzliche Überprüfung von Anwendungen (Apps), bevor diese darüber verteilt und installiert werden können.

1.4 Arten von Betriebssystemen

Im Laufe der Entwicklung der Betriebssysteme haben sich verschiedene Typen herausgebildet, die hier im Folgenden kurz vorgestellt werden sollen. Auf Besonderheiten einiger der Typen werden wir später noch eingehen.


Multiprozessor-Betriebssystem: Einige moderne Computersysteme beinhalten mehr als einen Prozessor und benötigen daher ein Multiprozessor-Betriebssystem, das die Verteilung der Aufgaben auf die Prozessoren vornimmt.


Embedded Betriebssysteme: Sie kommen in PDAs und vergleichbaren mobilen Endgeräten zum Einsatz, wobei die geringe Größe der Geräte und die damit verbundene oft geringere Rechenleistung und Speicherausstattung eine besondere Herausforderung darstellen. Effizientes Energiemanagement ist entscheidend, gleichzeitig aber muss eine Vielzahl von Kommunikationsschnittstellen unterstützt werden, wie z.B. der Aufbau oder das Einloggen in ein Ad-Hoc-Piconetz (Bluetooth) oder die Einwahl in ein Mobilfunknetz (GSM, UMTS).

Betriebssysteme für Chipkarten: Diese Mini-Betriebssysteme laufen auf Smart Cards, wie sie z.B. im Zusammenhang mit der RFID-Technologie eingesetzt werden.
Teil II

Prozesse
2

Programme und Unterprogramme

► Maschinenprogramme
► Unterprogramme und Prozeduren
► Programmschema für Unterprogrammaufrufe
► Nested Procedures und rekursive Prozedurauftrufe

Inhaltsangabe

2.1 Vom Programm zum Maschinenprogramm .......................... 14
2.2 Unterprogramme und Prozeduren ................................. 14
   2.2.1 Die Befehle CALL und RET ................................. 16
   2.2.2 Schema für Unterprogrammaufrufe .......................... 17
   2.2.3 Module ......................................................... 20
2.3 Realisierung eines Unterprogrammaufrufs ......................... 21
2.4 Rekursive Prozedurauftrufe ....................................... 25
Kapitel 2. Programme und Unterprogramme

2.1 Vom Programm zum Maschinenprogramm

Ausgangspunkt: Ein in Ausführung befindliches Programm ist eine Folge (Sequenz) von Befehlen. Die Befehle werden nacheinander vom Prozessor ausgeführt.

Wurde ein Programm in einer höheren Programmiersprache (z.B. Java, C, Pascal, SML, Prolog) formuliert, muss es natürlich zunächst in eine Befehlsfolge transformiert werden, die von der CPU auch “verstanden” wird. Dieser Vorgang wird Übersetzung genannt und vom Übersetzer (Compiler) ausgeführt. Das Hochsprachen-Programm dient also als Spezifikation für das gewünschte Maschinenprogramm.


Soll das so entstandene Maschinenprogramm nun ausgeführt werden, wird ihm vom Betriebssystem ein zusammenhängender Speicherbereich zugewiesen, der zunächst einmal aus genau so vielen Speicherzellen besteht wie das Programm Befehle enthält. Jeder Befehl wird in eine Speicherzelle kopiert, der Prozessor beginnt bei der ersten Speicherzelle mit der Ausführung des darin kodierten Befehls.

Programme in höheren Programmiersprachen werden in Maschinenprogramme übersetzt. Ein Maschinenprogramm ist eine Folge von Befehlen, die von einer bestimmten Maschine (CPU) direkt verarbeitet werden können.

2.2 Unterprogramme und Prozeduren

Problemstellung: Eine Folge von Befehlen wird mehrmals während der Ausführung eines Programms benötigt.


Geschlossenes Unterprogramm (Prozedur): Das Programmstück wird über seine Anfangsadresse (die Adresse der Speicherzelle, mit der es beginnt) angesprungen. Nach der Aus-
führung des Unterprogramms erfolgt ein Rück sprung zu der Adresse, die unmittelbar vor dem Unterprogrammaufruf als nächstes dran gewesen wäre (Rückkehradresse). Insbesondere werden Prozeduren und Funktionen höherer Programmiersprachen als geschlossene Unterprogramme realisiert. Wir unterscheiden:

- **Announcement**: Prozedur ohne Ergebnisparameter (in Java z.B. gekennzeichnet durch leeren Rückgabetyp `void`).
- **Invocation**: Prozedur mit Ergebnisparameter

Unabhängig von obigen Unterscheidungen heißt ein Unterprogramm **rekursiv**, wenn in dem Unterprogramm selbst ein Aufruf dieses Unterprogramms erfolgt (ggf. mit modifizierten Parametern). Beispiel: 

\[ n! = n \cdot (n-1)! \text{ für } n > 0; \quad n! = 1 \text{ für } n = 0. \]

Wir betrachten Prozeduren, also geschlossene Unterprogramme, näher:

**Informationen**: Welche Informationen werden zur Realisierung einer Prozedur genau benötigt?

- **Anfangsadresse des Unterprogramms**: Diese Adresse wird dem Maschinenbefehl `CALL` explizit übergeben. Beispiel: `CALL 0x1802C0D0` bewirkt einen Sprung zum Unterprogramm, das an der Adresse `0x1802C0D0` beginnt.
- **Rücksprungadresse zum Hauptprogramm**: Der Befehl `RET` bewirkt den Rück sprung zum Aufrufer; hier wird die Rückkehrradresse nicht explizit angegeben, sondern entweder aus dem Register `RA` oder aus dem Stack geholt.
- **Aufrufparameter**: Sie dienen dem Unterprogramm als Eingabe(n).
- **Rückgabewert(e)**: Hat das Unterprogramm ein oder mehrere Berechnungsergebnisse ermittelt, werden diese dem Aufrufer mitgeteilt.

**Kommunikation**: Wie tauschen Hauptprogramm (Aufrufer, Caller) und Unterprogramm (Callee) Informationen, Parameter und Ergebnisse aus?

- **Stack**: Eine prinzipiell beliebige Anzahl von Parametern (nur beschränkt durch die Größe des Speichers) lässt sich über den Stack (Kellerspeicher) übermitteln. Im Fall von Aufrufparametern werden diese in umgekehrter Reihenfolge vom Caller auf den Stack geladen (PUSH), sodass der Callee diese mit dem `POP`-Befehl in korrekter Reihenfolge erhält. Analog für Rückgabewerte.
- **Spezielle Register**: Eine schnellere Möglichkeit besteht darin, eine geringe Anzahl von Parametern (zum Beispiel die ersten vier) über bestimmte CPU-Register zu übergeben. Register stehen in der Speicherhierarchie (vgl. “Rechnerarchitekturen”) ganz oben und haben geringste Zugriffszeiten. Es ist daher sinnvoll, diese Möglichkeit auszu nutzen.

_Haupt- und Unterprogramm tauschen Adressen, Parameter und Rückgabewerte über einen gemeinsam genutzten Kellerspeicher oder über vorher verabredete Register aus._
2.2.1 Die Befehle CALL und RET


Die Definition des Befehls \textit{JMP} sieht also so aus:

\begin{verbatim}
COMMAND JMP addr
BEGIN
  PC := addr;
END
\end{verbatim}

Der Befehl CALL unterscheidet sich davon durch die zusätzliche Sicherung der Rücksprungadresse. Hier gibt es zwei Möglichkeiten:

1. Die Rückkehradresse wird in einem speziellen Register (RA) gesichert. Als Befehlsdefinition ergibt sich:

\begin{verbatim}
COMMAND CALL addr
BEGIN
  RA := PC + 1;
  PC := addr;
END
\end{verbatim}

2. Die Rückkehradresse wird auf dem Stack gespeichert. Zwar sind Stack-Zugriffe nicht so schnell wie Registerzugriffe (entsprechend länger dauert die Befehlausführung), dafür lassen sich rekursive Unterprogrammaufrufe und Nested Procedures (siehe später) leichter realisieren. Die Befehlsdefinition lautet in diesem Fall:

\begin{verbatim}
COMMAND CALL addr
BEGIN
  PUSH (PC + 1);
  PC := addr;
END
\end{verbatim}

Entsprechend ergeben sich zwei Varianten für die Definition des \textit{RET}-Befehls:

1. Die Rückkehradresse steht im Register RA:

\begin{verbatim}
COMMAND RET
BEGIN
  PC := RA;
END
\end{verbatim}

2. Die Rückkehradresse befindet sich auf dem Stack:

\begin{verbatim}
COMMAND RET
BEGIN
  PC := POP;
END
\end{verbatim}
2.2.2 Schema für Unterprogrammaufrufe

Wir wollen ein Beispiel betrachten, bei dem die Rücksprungadressen auf dem Stack abgelegt werden. Beim Rücksprung ist die aktuelle Rückkehradresse immer das oberste Kellerelement. Ferner betrachten wir hier auch den Fall “genesteter” Unterprogramme.

**Nested Procedure:** Ein Unterprogramm ruft selbst ein anderes Unterprogramm auf. (Dieses könnte ggf. wiederum ein anderes Unterprogramm aufrufen. So entsteht eine Kette von genesteten Unterprogrammaufrufen.) Unterschied zu rekursiven Aufrufen: Ein rekursives Unterprogramm ruft **sich selbst** auf.

**Beispiel 2.1.** *Im Beispiel sollen Parameter für den Aufruf des Unterprogramms und ggf. Ergebnisse in Registern – also nicht auf dem Stack – abgelegt werden. Die Sicherung der Registerinhalte wird nicht explizit betrachtet. Die Programmadressen sind im Folgenden für das Beispiel beliebig gewählt.*

**Hauptprogramm:**

```
4000  ...
   ...
4100  CALL 4500
4101  ...
   ...
```

**Prozedur 1:**

```
4500  ...
   ...
4600  CALL 4800
4601  ...
   ...
4650  CALL 4800
4651  ...
   ...
4700  RET
```

**Prozedur 2:**

```
4800  ...
   ...
4890  RET
```

Die Abarbeitung, beginnend mit dem Befehl in der Speicherzelle 4100, lässt sich nun schematisch wie folgt darstellen:
Kapitel 2. Programme und Unterprogramme

Der Kellerspeicher sieht zu den verschiedenen Zeitpunkten der Abarbeitung so aus:

- Unmittelbar vor der Ausführung des Befehls an der Adresse 4100 ist der Keller entweder leer oder enthält Daten, die für unsere Betrachtungen nicht relevant sind. Der Stack-Pointer (SP) zeigt stets auf die oberste belegte Kellerzelle.

```
| .... | <-- SP
|______|
```

- Unmittelbar nach der Ausführung des CALL-Befehls an der Adresse 4100 enthält der Keller die Rückkehradresse 4101, also die Adresse, an der das Hauptprogramm nach dem Unterprogrammaufruf fortgesetzt wird.

```
| 4101 | <-- SP
|______|
| .... |
|______|
```
Entsprechend ergibt sich nach der Ausführung des Befehls an der Adresse 4600:

- Nach dem ersten RET:

- Nach dem zweiten RET:

- Nach dem dritten RET:

Nach n CALL-Befehlen und n Rücksprüngen (RET) hat der Keller die gleiche Belegung wie zu Beginn.
2.2.3 Module

Module: Unter Modulen versteht man allgemeine Komponenten, die anderen Komponenten einen Dienst bereitstellen. Beispiele für Module sind:

- Unterprogramme (Prozeduren, Funktionen)
- Komponenten des Betriebssystems, sowie das Betriebssystem als Ganzes
- Benutzerprogramme
- Prozesse


Ein Zustand ist insb. charakterisiert durch

- den Rechnerkernzustand, d.h. die aktuellen CPU-Registerbelegungen,
- die Speicherabbildungstabellen und
- Programmcode und Daten.

Bezüglich Programm- und Speicherräumen müssen die einzelnen Module von einander isoliert werden (d.h. keine Teilräume gemeinsam nutzen). Eine Aktivierung eines Moduls kann beispielsweise erfolgen durch

- Unterprogramm-/Prozeduraufrufe (in diesem Kapitel betrachtet)
- Systemaufrufe
- Prozesswechsel


Dienst: Ein Dienst ist eine Funktion, die von einem Objekt (hier: einem Modul) an einer bestimmten Schnittstelle angeboten wird.

2.3 Realisierung eines Unterprogrammaufrufs


**Die Modell-Maschine MI:** MI verfügt über

- die Register PC (Programmzähler), SP (Stack-Pointer) und R0 bis R14, wobei R12 und R13 nicht frei genutzt werden dürfen, sondern spezielle Bedeutungen haben:
  - R12 enthält die Ablageadresse des ersten Aufrufparameters ($p_1$).
  - R13 enthält die Basisadresse des lokalen Datenraums des Unterprogramms.
- einen Kellerspeicher, von dem angenommen werden kann, dass er für das betrachtete Szenario über ausreichend Speicherplatz verfügt. Wichtig ist allerdings, dass die Adressen der Kellerzellen bei wachsendem Keller kleiner werden.


**Vorbemerkung:** Der CALL-Befehl übernimmt lediglich die Sicherung der Rücksprungadresse und die Durchführung des eigentlichen Sprungs. Bei einem allgemeinen Unterprogrammaufruf ist aber auch dafür zu sorgen, dass dem Unterprogramm ein **lokaler Datenraum** für Daten innerhalb des Prozedurablaufs zur Verfügung steht. Ferner wird Platz für die “Rettung” der Registerbelegungen der CPU vor dem Unterprogrammaufruf benötigt (da das Unterprogramm diese Register ggf. überschreibt), sowie für die Speicherung der Aufrufparameter ($p_1$ bis $p_n$), die an das Unterprogramm übergeben werden.

Bei den folgenden Betrachtungen gehen wir davon aus, dass

- die aktuellen Parameter $p_1$ bis $p_n$ des Unterprogrammaufrufs in umgekehrter Reihenfolge auf den Keller gelegt werden.
- bei einer Funktion der Rückgabewert als fiktiver Parameter $p_{n+1}$ ebenfalls auf dem Keller liegt bzw. Platz dafür reserviert ist.
- ein Unterprogramm die MI-Register nutzen kann und daher vor ihrer Benutzung sichern und vor dem Rücksprung zum Aufrufer wiederherstellen (mit den alten Werten belegen) muss.

*Die Register der Maschine MI sind Callee-saved, d.h. das aufgerufene Unterprogramm trägt die Verantwortung dafür, dass der Kontext des Aufrufers (Caller) nach dem Rücksprung unverändert dem Kontext vor dem Unterprogrammaufruf entspricht.*

Der Bereich der lokalen Variablen des Unterprogramms im Keller wird **lokaler Datenraum** des Unterprogramms genannt.
**Befehle von MI:** Zur Spezifikation einer (Modell-)Maschine gehört auch die Definition der Maschinensprache, die diese Maschine “versteht” – also die Angabe der Befehle. Zur Erinnerung: Die CPU (also der Maschinenkern) erhält die Befehle in Form von Bitcodes über den Befehlsbus. Wir setzen voraus, dass die Umsetzung eines Maschinenbefehls (eigentlich: Assembler-Befehl) wie z.B. `CALL 4100` in eine oder mehrere 32-Bitfolge(n) (z.B. 01101010 10010011 11110000 00000000) durch den Assembler kein Problem darstellt. Die MI-Befehle sind (ohne formale Definition):

- **PUSH val:** legt den Wert `val` auf den Keller (der Stack-Pointer wird implizit dekrementiert)
- **PUSH reg:** legt den Inhalt des Registers `reg` auf den Keller
- **PUSHR:** sichert den gesamten CPU-Register-Kontext, d.h. die Inhalte der Register R0 bis R14 werden nacheinander auf den Keller gelegt (der dadurch um 15 Plätze = 60 Bytes wächst)
- **POP reg:** legt den Inhalt der obersten Kellerzelle ins Register `reg` (der Stack-Pointer wird implizit inkrementiert)
- **POPR:** stellt den gesamten Register-Kontext wieder her, d.h. die Inhalte der obersten 15 Kellerzellen werden nacheinander in die Register R14 bis R0 (zurück)kopiert
- **MOVE addr, reg:** kopiert die Adresse `addr` ins Register `reg`
- **MOVE reg1, reg2:** kopiert den Inhalt (Wert) von Register `reg1` ins Register `reg2`
- **CALL addr:** Sprung zu `addr` und Sicherung der Rücksprungadresse auf dem Keller (siehe oben)
- **RET:** Rücksprung zum Aufrufer
- weitere grundlegende Befehle, wie (bedingte) Sprungbefehle, logische und arithmetische Operationen (z.B. JMP, ROL, ADD usw.)
Damit sieht die Belegung des Kellerspeichers und der Register R12 und R13 nach dem Aufruf eines Unterprogramms allgemein wie folgt aus:

\[
\begin{array}{c|c}
\hline
\text{SP} & \text{SP} \\
\hline
\text{var2} & \text{(lokale Variable des Unterprogramms)} \\
\hline
\text{var1} & \text{(lokale Variable des Unterprogramms)} \\
\hline
\hline
\text{R0} & \text{R13} \\
\hline
\text{...} & \text{...} \\
\hline
\text{R14} & \text{...} \\
\hline
\text{RET-Addr} & \text{(Rücksprungadresse)} \\
\hline
\text{p_1} & \text{R12} \\
\hline
\text{...} & \text{...} \\
\hline
\text{p_{n-1}} & \text{...} \\
\hline
\text{p_n} & \text{...} \\
\hline
\text{....} & \text{....} \\
\hline
\end{array}
\]

(1): lokaler Datenraum des Unterprogramms

Als erweitertes Unterprogramm-Schema ergibt sich:

Hauptprogramm:

```
4000  ...
...
40??  PUSH p_n bzw. p_{n+1}
...
4098  PUSH p_2
4099  PUSH p_1
4100  CALL 4500
4101  POP p_1
4102  POP p_2
...
```
Kapitel 2. Programme und Unterprogramme

41?? POP p_n bzw. p_(n+1)

Prozedur:

4500 PUSHR # (1)
4501 MOVE 64+SP, R12 # (2)
4502 MOVE SP, R13 # (3)

?? ?? MOVE R13, SP # (4)
POPR # (5)
RET # (6)

Kommentare dazu:

– (1) sichert die Register R0 bis R14 auf dem Stack, der dadurch um 15 Speicherzellen wächst (beachte: die Adressen fallen dabei).

– (2) addiert zum Wert des Stack-Pointers 64 Bytes; man erhält die Adresse der Speicherzelle, die 16 (= 64:4 wegen 4 Bytes Wortgröße) Zellen unterhalb der obersten Kellerzelle liegt. Die Zahl 16 setzt sich zusammen aus 15 Zellen für die Register und einer Zelle für die Zelle mit der Rücksprungadresse. Dies ist die **Ablageadresse**, die in R12 gespeichert wird.

– (3) sichert die **Basisadresse** des lokalen Datenraums in R13.

– (4) setzt den Stack-Pointer auf die “alte” Basisadresse zurück. Bei einer “sauberen” Programmierung wurde der Keller vollständig abgeräumt, und der Stack-Pointer enthält bereits vor Ausführung dieses Befehls die korrekte Adresse.

– (5) versetzt alle Register wieder in den Zustand, den sie zum Zeitpunkt der Ausführung des PUSHR-Befehls hatten.

– (6) realisiert den Rücksprung.

**Prolog und Epilog des Unterprogramms:** Die Schritte (1) bis (3) werden als Prolog, die Schritte (4) bis (6) als Epilog des Callees bezeichnet.

**Abschließende Bemerkungen:**

Das Unterprogramm muss jedoch auf diese Parameter zugreifen können. Zu diesem Zweck wird die Ablageadresse (Register R12) genutzt, mit deren Hilfe auf \( p_1 \) bis \( p_n \) zugegriffen werden kann.

**Arten der Parameterübergabe:** Die Parameterübergabe vom Hauptprogramm an das Unterprogramm und umgekehrt für das Ergebnis vom Unterprogramm ans Hauptprogramm kann auf zwei Arten erfolgen:

- Call by value (Wertübergabe): Übergabe eines konkreten Wertes (z.B. 7)
- Call by reference (Adressübergabe): Übergabe der Adresse der (ersten) Speicherzelle, an der sich der konkrete Parameter befindet (z.B. bei ASCII-Zeichenketten)

### 2.4 Rekursive Prozeduraufrufe

Wir wollen abschließend untersuchen, wie sich ein Unterprogramm selbst aufrufen kann.

**Beispiel 2.2 (Rekurseses Unterprogramm).** Als Beispiel betrachten wir die Faktabilität-Berechnung.

**PROCEDURE fak (k: INTEGER) RETURNS INTEGER**

**BEGIN**

IF \( k = 0 \) THEN RETURN 1
ELSE RETURN \( k \times fak(k-1) \)
**END**

Die Berechnung von \( 3! \) ergibt folgende Sprünge:

<table>
<thead>
<tr>
<th>HP</th>
<th>UP1</th>
<th>UP2</th>
<th>UP3</th>
<th>UP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>k:=3</td>
<td>2 =/ 0</td>
<td>CALL FAK(2)</td>
<td>1 =/ 0</td>
<td>CALL FAK(0)</td>
</tr>
<tr>
<td>CALL FAK(3)</td>
<td>3! = 3 * 2!</td>
<td>2! = 2 * 1!</td>
<td>1! = 1 * 0!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= 6</td>
<td>= 2</td>
<td>= 1</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

Abbildung 2.1: Sprünge bei rekursiven Unterprogrammaufrufen

Zu beachten: Im i-ten Unterprogramm wird das Ergebnis in den lokalen Datenraum des (i-1)-ten Unterprogramms geschrieben bzw. im ersten Unterprogramm in den lokalen Datenraum des Hauptprogramms. Daher ist es so wichtig, erst Speicherplatz für ein Ergebnis einzurichten und dann zu springen.
Prozesse

3.1 Das Prozess-Konzept
   3.1.1 Grundlagen von Prozessen
   3.1.2 Erzeugung von Prozessen
   3.1.3 Realisierung von Multiprogramming
   3.1.4 Das 2-Zustands-Prozessmodell
   3.1.5 Das 5-Zustands-Prozessmodell
   3.1.6 Das 7-Zustands-Prozessmodell

3.2 Prozessbeschreibung
   3.2.1 Kontrollstrukturen des Betriebssystems
   3.2.2 Prozesskontrollstrukturen
   3.2.3 Zusammenfassung der Verwaltung und Beschreibung von Prozessen:

3.3 Prozesskontrolle
   3.3.1 Prozesswechsel (Kontext-Switch)
   3.3.2 Unterbrechungen
   3.3.3 Moduswechsel
   3.3.4 Konflikte bei Unterbrechungen
   3.3.5 Ausführung des Betriebssystems

Inhaltsangabe

3.1 Das Prozess-Konzept ........................................ 28
   3.1.1 Grundlagen von Prozessen ................................ 28
   3.1.2 Erzeugung von Prozessen ................................ 31
   3.1.3 Realisierung von Multiprogramming ..................... 34
   3.1.4 Das 2-Zustands-Prozessmodell .......................... 36
   3.1.5 Das 5-Zustands-Prozessmodell .......................... 37
   3.1.6 Das 7-Zustands-Prozessmodell .......................... 39

3.2 Prozessbeschreibung ........................................ 42
   3.2.1 Kontrollstrukturen des Betriebssystems ............... 43
   3.2.2 Prozesskontrollstrukturen .............................. 44
   3.2.3 Zusammenfassung der Verwaltung und Beschreibung von Prozessen: 49

3.3 Prozesskontrolle ............................................ 51
   3.3.1 Prozesswechsel (Kontext-Switch) ....................... 52
   3.3.2 Unterbrechungen ....................................... 53
   3.3.3 Moduswechsel ......................................... 55
   3.3.4 Konflikte bei Unterbrechungen ....................... 56
   3.3.5 Ausführung des Betriebssystems ...................... 57
Kapitel 3. Prozesse

3.1 Das Prozess-Konzept

3.1.1 Grundlagen von Prozessen


**Prozess:** Ein Prozess ist ein in Ausführung befindliches (nicht zwingend aktives) Maschinenprogramm. Mit anderen Worten: Während ein Maschinenprogramm nur eine Folge von Befehlen ist, so wird dieses Programm zum Prozess, indem es in den Speicher geladen wird, einen eigenen Kontext erhält und gestartet wird (Programmzähler auf Anfangsadresse des Maschinenprogramms setzen). Insbesondere beinhaltet der Prozess:

- den aktuellen Wert des Programmzählers
- aktuelle Werte der Register und der Variablen

**(Prozess-)Kontext:** Zum Kontext eines Prozesses gehören alle Informationen, die den aktuellen Ausführungszustand eines Prozesses genau beschreiben. Dazu zählen insb. die CPU-Register-Belegungen und alle Prozess-Status-Informationen. Informell könnte man sagen: Zum Prozess-Kontext gehört alles, was man von einem unterbrochenen Prozess benötigt, um ihn auf demselben Computer später fortzusetzen.

**(Prozess-)Image:** Als Image eines Prozesses bezeichnet man die Gesamtheit der physischen Bestandteile eines Prozesses, also insbesondere seine Befehlsfolge (Trace), aber auch seinen Kontext, lokale und globale Variablen und der Ausführungs-Stack. Informell könnte man sagen: Zum Prozess-Image gehört alles, was man von einem unterbrochenen Prozess benötigen würde, um ihn auf einen anderen Computer zu transportieren und dort fortzusetzen.

**Uniprogramming:** Prozesse werden sequenziell nacheinander, vollständig und ohne Unterbrechung ausgeführt.

**Beispiel 3.1** (Ressourcennutzung im Einprogrammbetrieb). Wir betrachten das Modifizieren eines gespeicherten Datensatzes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Zeit in sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesen eines Datensatzes</td>
<td>0.0015 sec.</td>
</tr>
<tr>
<td>Ausführen von 100 Befehlen</td>
<td>0.0001 sec.</td>
</tr>
<tr>
<td>Schreiben des Datensatzes</td>
<td>0.0015 sec.</td>
</tr>
</tbody>
</table>

CPU-Auslastung: \( \frac{0.0001}{0.0031} = 0.3226 \approx 3.2\% \)

Problem: Mehr als 96% der Zeit verbringt die CPU in diesem Beispiel damit, auf E/A-Vorgänge (Lesen, Schreiben) zu warten.

**Multiprogramming:** Der Prozessor wird zwischen mehreren Prozessen hin- und hergeschaltet, wobei jeder Prozess für einige 10 bis 100 Millisekunden ausgeführt, dann unterbrochen und zu einem anderen Prozess gewechselt wird. Dabei führt der Prozessor zu jedem

Zu beachten: Bei Multiprogramming können a priori keine Annahmen über den zeitlichen Ablauf eines Prozesses gemacht werden. Denn durch das Hin- und Herschalten zwischen mehreren Prozessen kann sich ein ungleichmäßiger und nicht reproduzierbarer Ablauf ergeben.

**Multiprocessoring:** Stehen mehrere Prozessoren zur Verfügung, dann können Prozesse auch echt-parallel ausgeführt werden. Da es aber auch in einem Mehrprozessor-System in der Regel weit mehr Prozesse als Prozessoren gibt, wird auf den einzelnen Prozessoren wiederum Multiprogramming (und nicht Uniprogramming) realisiert.

*Indem Prozesse, die auf E/A-Operationen warten, unterbrochen werden können, blockieren sie nicht länger die CPU. Damit trägt Multiprogramming dazu bei, die CPU-Auslastung und damit den Durchsatz zu erhöhen.*

**Beispiel 3.2 (Ressourcennutzung im Mehrprogrammbetrieb).** Gegeben ist ein Rechner mit 256 KBytes Speicher, einer angeschlossenen Platte, einem Terminal (Hardware-Schnittstelle zum Benutzer) und einem Drucker. Betrachtet werden drei Prozesse (Jobs).

<table>
<thead>
<tr>
<th>Job</th>
<th>Charakteristik</th>
<th>Dauer</th>
<th>benötigter Speicher</th>
<th>Platte</th>
<th>Terminal</th>
<th>Drucker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU-intensiv</td>
<td>5'</td>
<td>50 KBytes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>E/A-intensiv</td>
<td>15'</td>
<td>100 KBytes</td>
<td>-</td>
<td>ja</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>E/A-intensiv</td>
<td>10'</td>
<td>80 KBytes</td>
<td>ja</td>
<td>-</td>
<td>ja</td>
</tr>
</tbody>
</table>

Dabei wird für Job 2 und 3 angenommen, dass diese nur minimal den Prozessor beanspruchen. Die CPU-Auslastung liegt für Job 1 bei 75%, für Job 2 und 3 jeweils bei 5%. Werden die Programme sequenziell im Einprogrammbetrieb abgearbeitet, so ergibt sich die in Abbildung 3.1 dargestellte Ressourcenausnutzung.

Alternativ dazu führt die Ausführung im Mehrprogrammbetrieb zur verzahnten Ausführung der drei Jobs, wie in Abbildung 3.2 dargestellt. Dabei verbessert sich die Ressourcennutzung deutlich (Vgl. Abbildung 3.3).

Eine detaillierte Berechnung und Gegenüberstellung der Ressourcennutzung zeigt die folgende Tabelle:

<table>
<thead>
<tr>
<th>Uniprogramming</th>
<th>Multiprogramming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prozessorauslastung</td>
<td>17%</td>
</tr>
<tr>
<td>Speichernutzung</td>
<td>30%</td>
</tr>
<tr>
<td>Plattenauslastung</td>
<td>33%</td>
</tr>
<tr>
<td>Druckerauslastung</td>
<td>33%</td>
</tr>
<tr>
<td>Gesamtausführungsduer</td>
<td>30'</td>
</tr>
<tr>
<td>Durchsatz</td>
<td>6 Jobs/Std.</td>
</tr>
<tr>
<td>Mittlere Antwortzeit</td>
<td>18 min</td>
</tr>
</tbody>
</table>

Die einzelnen Werte dieser Aufstellung ergeben sich wie folgt:

- Prozessorauslastung:
Kapitel 3. Prozesse

Abbildung 3.1: Ressourcennutzung bei sequenzieller Ausführung von Jobs

<table>
<thead>
<tr>
<th>Zeitraum</th>
<th>Auslastung</th>
<th>mittlere Auslastung</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Uniprogramming</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1–5) 5':</td>
<td>75%</td>
<td>(5' · 75% + 5' · 5%)/30' = (375% + 125%)/30 = 500%/30 ≈ 17%</td>
</tr>
<tr>
<td>(6–20) 15':</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>(21–30) 10':</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td><strong>Multiprogramming</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1–5) 5':</td>
<td>(75 + 5 + 5)% = 85%</td>
<td>(5' · 85% + 5' · 10%) + 5' · 5%)/15' = 500%/15 ≈ 33.5%</td>
</tr>
<tr>
<td>(6–10) 5':</td>
<td>(5 + 5)% = 10%</td>
<td></td>
</tr>
<tr>
<td>(11–15) 5':</td>
<td>5%</td>
<td></td>
</tr>
</tbody>
</table>

- **Speichernutzung:**
  - **Uniprogramming**
    
    \[(50kB · 5' + 100kB · 15' + 80kB · 10')/30'\]  
    \[= (250kB + 1500kB + 800kB)/30\]  
    \[= 2550kB/30 = 85k\]  
    Bei einer Gesamtgröße des Speichers von 256k ergibt sich eine Ausnutzung von 85/256 ≈ 33%
  - **Multiprogramming**
    
    \[(230kB · 5' + 180kB · 5' + 100kB · 5')/15'\]  
    \[= (1150kB + 900kB + 500kB)/15\]  
    \[= 2550kB/15 = 170kB\]  
    Hier liegt die Ausnutzung bei 170/256 ≈ 67%

- **Platten- bzw. Druckerauslastung:**
Job A  Run  wait  Run  wait  Run  
Job B  Run  wait  Run  wait  Run  
Job C  Run  wait  Run  wait  Run  

kombiniert: A CB A CB A CB

Abbildung 3.2: Pseudo-parallele Ausführung

Uniprogramming  $10'/30' = 1/3 \approx 33\%$
Multiprogramming  $10'/15' = 2/3 \approx 67\%$

– Terminalauslastung:
Uniprogramming  $15'/30' = 1/2 = 50\%$
Multiprogramming  $15'/15' = 1/1 = 100\%$

– Durchsatz:
Uniprogramming  3 Jobs / 1/2 Std. $\Rightarrow$ 6 Jobs/Std.
Multiprogramming  3 Jobs / 1/4 Std. $\Rightarrow$ 12 Jobs/Std.

– Mittlere Antwortzeit:
Man betrachtet für jeden Prozess die Zeit vom Beginn bis zur fertigen Abarbeitung (= Antwortzeit dieses Prozesses). Hiervon bildet man den Durchschnitt (als arithmetisches Mittel).

Uniprogramming  
Job 1  5'  5'
Job 2  $20' := 5' + 15'$  15'
Job 3  $30' := 20' + 10'$  10'

55'/3 = 18.3 min.  30'/3 = 10 min.

Obwohl der Prozessor zu jedem Zeitpunkt nur einen Prozeß ausführt, kann er innerhalb einer Sekunde an verschiedenen Prozessen arbeiten.
$\Rightarrow$ Der Nutzer erhält die Illusion von Parallelität.
Dieses Wechseln des Prozessors zwischen mehreren Prozessen wird als Pseudo-Parallelität bezeichnet. Dies unterscheidet sich aber von echter Hardware-Parallelität, bei der der Prozessor Berechnungen ausführt, während ein oder mehrere E/A-Geräte Aufträge bearbeiten.
Um mit Parallelität bei Betriebssystemen leichter umgehen zu können, wurden Modelle entwickelt, wie z. B. das Prozeßmodell. Beim Prozeßmodell wird angenommen, daß jeder Prozeß seinen eigenen virtuellen Prozessor besitzt.

3.1.2 Erzeugung von Prozessen

In den bisherigen Überlegungen wurde von einer bereits existierenden Menge an Prozessen ausgegangen. Betriebssysteme, die ein Prozess-Modell zur Verfügung stellen, müssen natürlich auch eine Möglichkeit bieten, die erforderlichen Prozesse zu erzeugen. Daraus resultiert
Kapitel 3. Prozesse

Abbildung 3.3: Ressourcennutzung bei verzahnter Ausführung von Jobs

das Konzept der Prozess-Hierarchie: Ausgehend von einem laufenden initialen Prozess können Prozesse neue Prozesse erzeugen, die dann Kindprozesse (des Elternprozesses) genannt werden.

**Prozesserzeugung unter Unix/Linux:** Mit dem Systemaufruf `fork` können Prozesse erzeugt werden. Genauer: Es wird eine identische Kopie des aufrufenden Prozesses erzeugt. Mit dem Kommando `execve` kann ein Programm in den Speicherbereich des aufrufenden Prozesses geladen werden (der also dann das neue Programm ausführt).

Abbildung 3.4: FORK-Darstellung

- Bei mehrfacher `fork`-Ausführung lassen sich verschiedene Kindprozesse parallel zum Vaterprozess starten.
- Die Kindprozesse können ihrerseits selbst wieder Kindprozesse starten.
- So entsteht ein beliebig tiefer Baum von Prozessen, d.h. eine beliebig tiefe Prozess-Hierarchie.
Abbildung 3.5: Beispiel einer Prozesshierarchie: A hat die beiden Kindprozesse B und C erzeugt, B wiederum die drei Kinder D, E und F.


**Prozesserzeugung unter Microsoft Windows XP:** Unter Windows gibt es keine echte Prozesshierarchie. Stattdessen erhält der Elternprozess bei der Erzeugung eines Kindprozesses ein Handle auf diesen. Dieses Handle kann jedoch beliebig weitergegeben werden.


**Ursachen für die Erzeugung eines Prozesses:** Grundsätzlich gibt es vier mögliche Ursachen, aufgrund derer das Betriebssystem einen neuen Prozess erzeugt:

2. Benutzeranmeldung: Meldet sich im interaktiven Betrieb ein Nutzer am System an, so wird für diesen Vorgang ein neuer Prozess erzeugt.

*In Unix sind alle Prozesse Nachkommen des Prozesses init, der beim Initialisieren des Systems gestartet wird.*

**Schritte bei der Prozess-Erzeugung:** Im Einzelnen müssen vom Betriebssystem bei der Prozess-Erzeugung die folgenden Schritte durchgeführt werden:
1. Der Prozess bekommt einen eindeutigen Identifikator zugewiesen und wird mit dieser PID in die Prozebstabelle eingetragen.


3.1.3 Realisierung von Multiprogramming


Sicht des Prozessors: Die CPU führt definierte Befehle aus, und zwar in der Reihenfolge, wie es ihr durch ihren Programmzähler (Program Counter) diktiert wird. Dass dieser Programmzähler dabei auf Code in unterschiedlichen Programmen verweist, die unter
Umständen Bestandteil verschiedenster Anwendungen sind, spielt für die CPU keine Rolle.

**Sicht eines individuellen Programms:** Seine Ausführung besteht aus einer Folge von Maschinenbefehlen, die auch Spur (Trace) des assoziierten Prozesses genannt wird. Hierbei ist nur von Bedeutung, dass die Reihenfolge der Abarbeitung seiner Instruktionen nicht verändert wird.

**Dispatcher:** Der Dispatcher ist ein Prozess, der einen Benutzer-Prozess, der sich in Bearbeitung befindet, unterbrechen und einen anderen Prozess dem Prozessor zuweisen kann.

**Beispiel 3.3.** Wir betrachten als einfaches Szenario drei Benutzer-Prozesse, die durch Programme repräsentiert werden, sowie einen (relativ kleinen) Dispatcher-Prozess. Alle Prozesse wurden vollständig in den Hauptspeicher geladen (vgl. Abbildung 3.6).

![Abbildung 3.6: Speicherbelegung der drei Beispielprozesse](image)

Die Spuren der drei Prozesse sollen wie folgt aussehen (Anfang der Traces):
Kapitel 3. Prozesse

<table>
<thead>
<tr>
<th>Trace des Prozesses A</th>
<th>Trace des Prozesses B</th>
<th>Trace des Prozesses C</th>
</tr>
</thead>
<tbody>
<tr>
<td>α + 0</td>
<td>β + 0</td>
<td>γ + 0</td>
</tr>
<tr>
<td>α + 1</td>
<td>β + 1</td>
<td>γ + 1</td>
</tr>
<tr>
<td>α + 2</td>
<td>β + 2</td>
<td>γ + 2</td>
</tr>
<tr>
<td>α + 3</td>
<td>β + 3</td>
<td>γ + 3</td>
</tr>
<tr>
<td>α + 4</td>
<td>(E/A-Op.)</td>
<td>γ + 4</td>
</tr>
<tr>
<td>α + 5</td>
<td></td>
<td>γ + 5</td>
</tr>
<tr>
<td>α + 6</td>
<td></td>
<td>γ + 6</td>
</tr>
<tr>
<td>α + 7</td>
<td></td>
<td>γ + 7</td>
</tr>
<tr>
<td>α + 8</td>
<td></td>
<td>γ + 8</td>
</tr>
<tr>
<td>α + 9</td>
<td></td>
<td>γ + 9</td>
</tr>
<tr>
<td>α + 10</td>
<td></td>
<td>γ + 10</td>
</tr>
<tr>
<td>α + 11</td>
<td></td>
<td>γ + 11</td>
</tr>
</tbody>
</table>

Dabei sind α, β und γ die Anfangsadressen der Prozesse A bzw. B und C. Bei Prozess B soll angenommen werden, dass der vierte Befehl (also β + 3) eine E/A-Operation ist, die ein Warten des Prozesses bedingt. Aus Sicht des Prozessors erhalten wir zur Ausführung dieser 12 + 4 + 12 = 28 Befehle folgende Sequenz abzuarbeitender Befehle:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01. α + 0</td>
<td>16. β + 3 E/A-Anforderung</td>
<td>31. δ + 2</td>
</tr>
<tr>
<td>02. α + 1</td>
<td>17. δ + 0</td>
<td>32. δ + 3</td>
</tr>
<tr>
<td>03. α + 2</td>
<td>18. δ + 1</td>
<td>33. δ + 4</td>
</tr>
<tr>
<td>04. α + 3</td>
<td>19. δ + 2</td>
<td>34. δ + 5</td>
</tr>
<tr>
<td>05. α + 4</td>
<td>20. δ + 3</td>
<td>35. α + 6</td>
</tr>
<tr>
<td>06. α + 5 Timeout</td>
<td>21. δ + 4</td>
<td>36. α + 7</td>
</tr>
<tr>
<td>07. δ + 0</td>
<td>22. δ + 5</td>
<td>37. α + 8</td>
</tr>
<tr>
<td>08. δ + 1</td>
<td>23. γ + 0</td>
<td>38. α + 9</td>
</tr>
<tr>
<td>09. δ + 2</td>
<td>24. γ + 1</td>
<td>39. α + 10</td>
</tr>
<tr>
<td>10. δ + 3</td>
<td>25. γ + 2</td>
<td>40. α + 11 Timeout</td>
</tr>
<tr>
<td>11. δ + 4</td>
<td>26. γ + 3</td>
<td>41. δ + 0</td>
</tr>
<tr>
<td>12. δ + 5</td>
<td>27. γ + 4</td>
<td>42. δ + 1</td>
</tr>
<tr>
<td>13. δ + 0</td>
<td>28. γ + 5 Timeout</td>
<td>43. δ + 2</td>
</tr>
<tr>
<td>14. β + 1</td>
<td>29. δ + 0</td>
<td>44. δ + 3</td>
</tr>
<tr>
<td>15. β + 2</td>
<td>30. δ + 1</td>
<td>45. δ + 4</td>
</tr>
</tbody>
</table>

Dem liegt das Prinzip zu Grunde, dass die Ausführung eines Prozesses nach maximal sechs Befehlszyklen durch ein Timeout unterbrochen wird. Dann übergibt der Dispatcher-Prozess die Kontrolle an den nächsten Benutzer-Prozess.

**Fragen:** Wie lassen sich dieses und andere Szenarien in Form von Kontrollstrukturen im Betriebssystem abbilden? Wie wird die Suspendierung eines Prozesses genau realisiert? Um diese Fragen beantworten zu können, benötigen wir ein Modell, das den Zustand eines Prozesses charakterisiert.

### 3.1.4 Das 2-Zustands-Prozessmodell

Prinzipiell kann sich ein Prozess in Ausführung befinden (d.h. Instruktionen aus seiner Trace werden von der CPU ausgeführt) oder aber er wird nicht ausgeführt. Die entsprechenden
Zustände heißen “Running” und “Not running”. Dieses einfachste Prozessmodell sieht wie folgt aus:

![Diagramm 2-Zustands-Prozessmodell](image)

Abbildung 3.7: 2-Zustands-Prozessmodell

Für das Betriebssystem können vorerst zwei Anforderungen an die Informationen über einen Prozess abgeleitet werden:

- der aktuelle Zustand des Prozesses muss beschreibbar sein
- die Speicherinformationen bzgl. des Prozesses müssen abrufbar sein (d.h. wo er gespeichert ist)

Aus Modellierungssicht müssen Prozesse im Zustand “Not running” in einer Art Warteschleife Zwischenspeichert werden, um auf ihre Ausführung zu warten. Dies ergibt das folgende Warteschlangenmodell:

![Diagramm Warteschlangenmodell des 2-Zustand-Prozessmodells](image)

Abbildung 3.8: Warteschlangenmodell des 2-Zustand-Prozessmodells


**3.1.5 Das 5-Zustands-Prozessmodell**

Bislang wurde davon ausgegangen, dass alle Prozesse stets zur Ausführung bereit sind. Das zugehörige 2-Zustands-Prozessmodell mit FIFO-Warteschlange würde dann effektiv arbeiten.
Aber: Prozesse können nicht rechenbereit sein, wenn sie z.B. auf die Beendigung einer E/A-Operation warten. In diesem Fall heißt ein Prozess blockiert. Nicht laufende (bisher: “Not running”) Prozesse werden in zwei Klassen eingeteilt:

- Prozesse, die rechenbereit sind: “Ready”
- Blockierte Prozesse: “Blocked”

Hinzu kommen die Zustände “New” (für neu generierte Prozesse) und “Exit” (für terminierte Prozesse). Dies ergibt folgendes Diagramm:

Abbildung 3.9: 5-Zustands-Prozessmodell

Zusammenfassung der Zustände: Zustände und ihre Bedeutung im 5-Zustands-Prozessmodell:

- Running: Der Prozess befindet sich in der Ausführung. Bei Einprozessor-Systemen kann sich zu jedem Zeitpunkt nur ein Prozess in diesem Zustand befinden.
- Ready: Der Prozess ist zur Ausführung bereit.
- Blocked: Der Prozess wartet auf ein Ereignis (z.B. Ende einer E/A-Operation, Benutzereingabe) und kann erst zur Ausführung kommen, nachdem das Ereignis eingetreten ist.
- New: Der Prozess wurde erzeugt, aber noch nicht durch das Betriebssystem zur Menge der ausführbaren Prozesse hinzugefügt.
- Exit: Der Prozess wurde durch das Betriebssystem aus der Menge der ausführbaren Prozesse entfernt.


Zulässige Zustandsübergänge: Beispiele für Ursachen aller zulässigen Zustandswechsel:

- Null→New: Ein Prozess wird erzeugt.
- New→Ready: Das Betriebssystem ist in der Lage, einen zusätzlichen Prozess aufzunehmen (genug Speicher vorhanden).
- Ready→Running: Ein rechenbereiter Prozess wird zur Ausführung ausgewählt.
- Running→Exit: Das Betriebssystem beendet den Prozess.
- **Running**→**Ready**: Ein anderer Prozess soll zur Ausführung ausgewählt werden (z.B. weil die maximale Zeit für einen Ausführungs-Slot abgelaufen oder ein höher-priorer Prozess ins System gekommen ist).
- **Running**→**Blocked**: Der Prozess muss auf ein Ereignis warten.
- **Blocked**→**Ready**: Das Ereignis, auf das gewartet wurde, ist eingetroffen.

**Beispiel 3.4.**

<table>
<thead>
<tr>
<th>Zeitintervall</th>
<th>Prozeß A</th>
<th>Prozeß B</th>
<th>Prozeß C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>Running</td>
<td>Ready</td>
<td>Ready</td>
</tr>
<tr>
<td>7-12</td>
<td>Ready</td>
<td>Ready</td>
<td>Ready</td>
</tr>
<tr>
<td>13-16</td>
<td>Ready</td>
<td>Running</td>
<td>Ready</td>
</tr>
<tr>
<td>17-22</td>
<td>Ready</td>
<td>Blocked</td>
<td>Ready</td>
</tr>
<tr>
<td>23-28</td>
<td>Ready</td>
<td>Blocked</td>
<td>Running</td>
</tr>
<tr>
<td>29-34</td>
<td>Ready</td>
<td>Blocked</td>
<td>Ready</td>
</tr>
<tr>
<td>35-40</td>
<td>Running</td>
<td>Blocked</td>
<td>Ready</td>
</tr>
<tr>
<td>41-46</td>
<td>Ready</td>
<td>Blocked</td>
<td>Ready</td>
</tr>
<tr>
<td>47-52</td>
<td>Ready</td>
<td>Blocked</td>
<td>Running</td>
</tr>
<tr>
<td>usw.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eine Implementierung kann auf zwei Warteschlangen basieren:

**Abbildung 3.10: Warteschlangenmodell des 5-Zustand-Prozessmodells**

*Zur Realisierung des 5-Zustands-Prozess-Modells werden mindestens zwei Warteschlangen benötigt.*

Alternativ können statt einer Blocked Queue auch mehrere Blocked Queues realisiert werden – jeweils eine pro Ereignisart:

**Splitting der Ready Queue**: Falls es für das Scheduling von Prozessen verschiedene Prioritätsklassen gibt, kann auch die Ready Queue noch einmal in verschiedene Queues unterteilt werden.

### 3.1.6 Das 7-Zustands-Prozessmodell

Bisher muss jeder Prozess, der vom Prozessor ausgeführt werden soll, komplett in den Hauptspeicher geladen werden. Ein häufiger Grund für das Warten auf ein Ereignis im blockierten
Zustand ist eine Ein- oder Ausgabe in Bearbeitung (E/A-Operation). Im Vergleich zu Berechnungen sind E/A-Operationen sehr langsam.

**Frage:** Was passiert, wenn alle Prozesse im Hauptspeicher auf das Ende einer E/A-Operation warten und kein Speicherplatz für weitere Prozesse mehr frei ist? Dann ist die CPU nicht mehr ausgelastet, potenziell vorhandene Rechenkapazität bleibt ungenutzt.

- Lösung 1: Vergrößerung des Hauptspeichers, sodass dieser mehr Prozesse aufnehmen kann (also Einbau weiterer Speichermodule).
- Lösung 2: Auslagerung (Swapping); ganze Prozesse oder auch nur Teile davon werden in den Hintergrundspeicher (in der Regel eine Platte) ausgelagert. Damit steht dem Betriebssystem praktisch mehr Speicher (virtuell) zur Verfügung als physisch in Form von Speichermodulen vorhanden ist.

**Auslagerung von Prozessen (Swapping):** Falls sich keiner der Prozesse im Hauptspeicher im Zustand “Ready” befindet, so lagert das Betriebssystem einen der blockierten Prozesse auf die Platte aus, indem es vor allem die speicherintensiven Bestandteile des Prozesses, wie die Instruktions-Spur (Trace) in einen reservierten Bereich des Datenträgers (oft: Swap-Partition) kopiert. Dadurch wird im Hauptspeicher Platz für neue, rechenbereite Prozesse geschaffen. Ausgelagerte Prozesse werden in einer Suspend Queue verwaltet (analog zu Ready Queue und Blocked Queue). Die Suspend Queue selbst und in der Regel auch die Verwaltungsinformationen zum Prozess (Prozesskontrollblock, siehe später) werden allerdings im Hauptspeicher gehalten.

**Zu beachten:** Swapping ist selbst eine E/A-Operation, da ja Daten zwischen zwei verschiedenen Ebenen der Speichерhierarchie transportiert werden müssen. Aber: Eine Lese- oder Schreiboperation auf einer Platte ist im Allgemeinen die schnellste aller E/A-Operationen.
Das Prozess-Konzept 41

(z.B. verglichen mit einem Druckauftrag oder dem Lesen von einem Wechseldatenträger wie Diskette, CD oder DVD).

**Virtueller Speicher:** Der virtuelle Speicher ist die Menge an Speicher (in Bytes), die dem Betriebssystems maximal zur Abbildung von Prozessen auf dem Hintergrundspeicher zur Verfügung steht. Dabei gilt allerdings:

- Der Hauptspeicher repräsentiert stets einen Ausschnitt des virtuellen Speichers. Anders formuliert enthält der virtuelle Speicher prinzipiell die gleichen Daten wie der Hauptspeicher, und zusätzlich Daten von suspendierten Prozessen. Dabei können bestimmte Daten im Hauptspeicher “aktueller” sein als die gleichen Daten im Hintergrundspeicher, z.B. wenn Variablen während der Ausführung neue Werte annehmen. Der Hintergrundspeicher kann also temporär “schmutzige” Daten enthalten, was eine Synchronisation von Hauptspeicher und virtuellem Speicher erforderlich macht (detaillierte Behandlung später).

  - Die Gesamt-Prozess-Kapazität des Betriebssystems ist damit gleich der Kapazität des virtuellen Speichers (und **nicht (!)** gleich der Summe der Kapazitäten von Hauptspeicher und virtuellem Speicher).

  - Demnach muss der virtuelle Speicher immer größer als der tatsächlich vorhandene Hauptspeicher sein (in der Praxis zwei- bis dreimal so groß).

*Durch virtuellen Speicher und Swapping wird die CPU-Auslastung zusätzlich erhöht, weil Prozesse, die auf das Ende von E/A-Operationen warten, nicht zwangsläufig den Hauptspeicher blockieren. Swapping verstärkt also die positiven Effekte von Multiprogramming.*

Das Swapping bedingt nun zunächst eine Erweiterung des 5-Zustands-Prozessmodells um einen Zustand “Suspend”:

Abbildung 3.12: 5-Zustands-Modell mit Suspend

**Problem:** Wenn durch das Auslagern eines Prozesses ein weiterer Prozess geladen werden kann, dann ist es oft günstiger, einen der ausgelagerten Prozesse, die inzwischen wieder rechenbereit sind, zu nehmen statt eines neuen. Solche Prozesse sollten sich natürlich nicht mehr im Zustand “blocked” befinden.

Aus diesem Grund macht es Sinn, sowohl bei den im Hauptspeicher enthaltenen als auch bei den ausgelagerten Prozessen zwischen blockierten und nicht blockierten zu unterscheiden:

<table>
<thead>
<tr>
<th></th>
<th>Ready</th>
<th>Blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hauptspeicher</td>
<td>Zustand: Ready</td>
<td>Zustand: Blocked</td>
</tr>
<tr>
<td>Hintergrundspeicher</td>
<td>Zustand: Ready, Suspend</td>
<td>Zustand: Blocked, Suspend</td>
</tr>
</tbody>
</table>
Kapitel 3. Prozesse

Dies ergibt noch einmal ein verfeinertes Zustandsübergangsdiagramm:

Abbildung 3.13: 7-Zustands-Prozessmodell

Ursachen für die Suspendierung von Prozessen: Für die Suspendierung von Prozessen kann es verschiedene Ursachen geben:


3.2 Prozessbeschreibung

Ein Betriebssystem kontrolliert Ereignisse innerhalb des Computersystems, realisiert Scheduling und Dispatching von Prozessen und ordnet Ressourcen (Betriebsmittel) den verschiedenen Prozessen zu.

Bemerkungen zu Abbildung 3.14:

- In einer Multiprogramming-Umgebung existieren n Prozesse P₁, ..., Pₙ.
- Jeder dieser Prozesse muss auf bestimmte Systemressourcen zugreifen.
Frage: Welche Informationen benötigt das Betriebssystem, um Prozesse zu kontrollieren und Ressourcen für diese Prozesse zu verwalten?

3.2.1 Kontrollstrukturen des Betriebssystems


Seiten (Pages): Analog dazu wird die Instruktionsfolge (Trace) eines Prozesses in mehrere Seiten aufgeteilt. Alle Seiten sind gleich groß (auch wenn dadurch einige Seiten nicht ganz gefüllt sind) und haben die gleiche Größe wie die Seitenrahmen (Frames) im Hauptspeicher. Jede Seite “passt” also in jeden beliebigen Seitenrahmen.

1. Speichertabellen (Seitentabellen)

- Speichertabellen dienen dazu, den Überblick sowohl über den Hauptspeicher als auch über den virtuellen Speicher zu behalten.
- Dabei ist stets ein Teil des Hauptspeichers für das Betriebssystem selbst reserviert, der Rest steht für (Benutzer-)Prozesse zur Verfügung.
Kapitel 3. Prozesse

- Speichertabellen beinhalten vor allem
  - die Zuordnung der Seiten eines Prozesses zu den Frames des Hauptspeichers,
  - die Zuteilung des virtuellen Speichers zu den Prozessen und
  - Schutzattribute von Seiten bzw. Frames, um den Zugriff auf gemeinsam genutzte Regionen zu kontrollieren.

2. E/A-Tabellen
- Sie dienen zur Verwaltung von E/A-Geräten sowie Kanälen des Computersystems.
- Ein E/A-Gerät ist entweder verfügbar, oder es ist einem bestimmten Prozess zugeordnet.
- Im zweiten Fall kennt das Betriebssystem den Status der E/A-Operation und den Ort im Hauptspeicher, der als Ziel bzw. Quelle des E/A-Transfers genutzt wird.

3. Dateitabellen
- Sie enthalten Informationen über die Existenz von Dateien, über ihren Ort im Hintergrundspeicher, ihren aktuellen Status und andere Attribute.
- Falls ein separates Dateisystem existiert (wie heute üblich), so kann ein Großteil dieser Informationen darin enthalten sein; das Betriebssystem besitzt dementsprechend weniger Informationen.

4. Prozesstabellen
- Sie enthalten Informationen zur Verwaltung aller Prozesse im System.


3.2.2 Prozesskontrollstrukturen
Um einen Prozess zu verwalten und zu kontrollieren, muss das Betriebssystem wissen:
I. Wo ist der Prozess gespeichert? (Prozesslokalisierung)
II. Welche Werte haben die für das Prozessmanagement relevanten Attribute? (Prozesskontrollblock)
I. Prozesslokalisierung: Die Lokalisierung eines Prozesses hängt von dem Paradigma der eingesetzten Speicherverwaltung (Segmentierung oder Paging) ab:
**Variante 1: Dynamische Partitionierung (→Segmentierung)**

– Der Prozess wird als zusammenhängender Block in aufeinanderfolgenden Speicherzellen abgelegt.
– Damit das Betriebssystems diesen Prozess verwalten kann, muss zumindest ein Teil der gespeicherten Daten in den Hauptspeicher geladen werden.
– Um den Prozess ausführen zu können, wird in der Regel der gesamte Prozess in den Hauptspeicher geladen.

Das Betriebssystem muss dabei stets wissen, wo im Hintergrund- bzw. Hauptspeicher das Prozess-Image abgelegt ist.

**Variante 2: Feste Partitionierung (→Paging)**

– Der Speicher wird in feste (!) Blöcke partitioniert und ein Prozess-Image in aneinander grenzenden Blöcken gespeichert. Sind diese Blöcke alle gleich groß (das ist der Normalfall), so spricht man von Seiten (siehe oben).
– Das gesamte Prozess-Image befindet sich stets im Hintergrundspeicher.
– Wird ein Teil im Hauptspeicher benötigt, so wird dieser Teil in freie Frames kopiert.
– Falls der Teil im Hauptspeicher verändert wird, so ist die Kopie im Hintergrundspeicher so lange unaktuell, bis der Teil des Hauptspeichers zurückgeschrieben wird.

Die Prozesstabelle enthält bei dieser Struktur einen Eintrag für jeden Prozess. Dieser Eintrag enthält dann mindestens einen Zeiger auf das Prozess-Image.


**Weitere Variante: Compatible Time-Sharing System (CTSS)**


Im Folgenden soll nun das Time Sharing am Beispiel des Compatible Time-Sharing System (CTSS) betrachtet werden. Dieses Betriebssystem war extrem einfach und funktionierte für bis zu 32 Nutzer. Wichtig war nur, daß das Betriebssystem auch die Informationen verwaltete, welche Prozeßdaten im Hauptspeicher behalten wurden und welche Daten ausgelagert wurden.

**Beispiel: CTSS, MIT 1962, für IBM 709 und später IBM 7094**

Wir haben 36k Hauptspeicher (bei IBM 7094), von denen ein Monitorprozeß ständig 5k, und das Betriebssystemprozeß 4k benötigen. Bekommt ein interaktiver Nutzer die Kontrolle über den Prozessor, so werden die verbleibenden 27k des Hauptspeichers für sein Programm und die Nutzerdaten bereitgestellt.

Von einer Systemuhr werden alle 0.2 Sekunden Interrupts erzeugt, die es dem Betriebssystem erlaubt, den Prozessor einem anderen Nutzerprozeß zuzuordnen. Diese Art der Kontrolle wird auch als **Time Slicing** bezeichnet.

Um die Interaktion mit dem Hintergrundspeicher zu minimieren, werden Speicherbereiche des zu unterbrechenden Nutzerprozesses nur dann ausgelagert, wenn der Prozess, der als nächstes die Kontrolle über den Prozessor bekommen soll, diese überschreiben.
würde. Eine Besonderheit bei IBM 7094 war die Tatsache, dass ein Programm immer an der Adresse 5k (direkt nach dem Monitorjob) geladen wurde. Abhängig von dem Speicherplatzanforderungen des neuen Nutzerprozesses, würde daher nur ein Teil oder aber der gesamte gerade aktive Job ausgelagert.

Für das folgende Beispiel sollen folgende interaktive Jobs ausgeführt werden:

| Job 1: 15k | Job 2: 20k | Job 3: 5k | Job 4: 10k |

Die Speicherbelegung erfolgt wie in Abbildung 3.16 dargestellt.

Abbildung 3.16: Beispiel: Speicherbelegung bei CTSS


II. Prozessattribute im Prozesskontrollblock: Der Prozesskontrollblock (PCB) ist die bedeutendste Datenstruktur in der Prozessverwaltung des Betriebssystems. Er enthält die wesentlichen Informationen, die das Betriebssystem zur Kontrolle eines Prozess benötigt. In verschiedenen Betriebssystemen existieren unterschiedliche Realisierungsvarianten. Generell beinhaltet der PCB drei Kategorien von Informationen:

(a) Prozessidentifikation:
   - numerischer Identifikator (Prozess-ID, PID)
   - Identifikator des Prozesses, der diesen Prozess generiert hat (PID des Elternprozesses); dadurch wird die Prozess-Hierarchie abgebildet.
   - ID des Eigentümers/Nutzers des Prozesses
Kapitel 3. Prozesse

Diese Identifikatoren sind nützlich für Querverweise, z.B. in E/A- oder Speichertabellen. Sie können auch bei der Prozeßkommunikation gebraucht werden.

(b) Prozesszustandsinformationen:

- Inhalt der Prozessorregister (Befindet sich ein Prozeß in der Ausführung, so befinden sich diese Informationen in den Registern.)
- Wird ein Prozeß unterbrochen, so müssen alle Registerinformationen gespeichert werden, so daß dieser Prozeß beim erneuten Laden ordnungsgemäß wieder abgearbeitet werden kann.

**Wiederholung:** Anzahl und Art der Register hängt von der verwendeten Hardware ab.

In der Regel gibt es:

- Register, die vom Nutzer verwendet werden können,
- den Stackpointer
- Kontroll- und Statusregister (d.h. Programmzählern, Condition codes (carryflag, overflowflag, ...))

- Die Menge der Register, die Statusinformationen enthalten, sind in der Regel als sog. Programmstatuswort (PSW) gespeichert.

**Beispiel 3.5.** Innerhalb von Pentiumprozessoren wird das PSW als EFLAGS-Register bezeichnet (vgl. Abbildung 3.17). Diese Struktur wird von jedem Betriebssystem genutzt, das auf einem Pentiumprozessor läuft. (Z.B. UNIX oder Windows NT)

Abbildung 3.17: Struktur des Pentium-EFLAGS-Registers

(c) Prozesskontrollinformationen:

- Scheduling- und Zustandsinformationen, wie
  - Prozesszustand ("Ready", "Running", "Blocked", ...)
  - Priorität
  - Scheduling-Strategie-spezifische Zustandsinformationen, wie z.B. die Zeit, die der Prozeß bereits wartet
  - Ereignisse, auf die der Prozeß wartet
Datenstrukturen, z.B. eine Referenz auf den nächsten Prozess, falls alle wartenden Prozesse in einer Queue verwaltet werden
- Signale oder Nachrichten, die zwischen zwei unabhängigen Prozessen ausgetauscht werden (Interprozesskommunikation)
- zusätzliche Informationen über Privilegien des Prozesses, Speichermanagement, Eigentümerverhältnissen von Ressourcen u.a.

Damit können wir die Struktur eines Prozesses im Hintergrundspeicher wie folgt zusammenfassen:

Abbildung 3.18: Struktur eines Prozesses im Hintergrundspeicher

Mit dieser Prozessbeschreibung könnte das 5-Zustands-Prozessmodell wie folgt implementiert werden:

Abbildung 3.19: Implementierung des 5-Zustands-Prozessmodells

Bemerkung zu Abbildung [3.19]: Jeder Prozess besitzt eine eindeutige ID, die in einer Tabelle von Zeigern auf Prozesskontrollblöcken als Index genutzt werden kann, d.h. die Queues (Ready- und Blocked-Queue) können als verkettete Listen von PCBs implementiert werden.

3.2.3 Zusammenfassung der Verwaltung und Beschreibung von Prozessen:

Das Betriebssystem entscheidet periodisch, ob ein Prozess angehalten werden und ein anderer aktiviert werden soll (Scheduler). Man sagt, der Prozess wird suspendiert. Wenn ein Prozess
suspendiert wird, dann muß die Abarbeitung später genau in dem Zustand wieder aufgenommen werden, in dem er angehalten worden ist. Dazu muß man einen Prozess beschreiben können.

**Lösung:** Informationen über jeden Prozeß werden in der sogenannten Prozesstabelle gespeichert, die vom Betriebssystem verwaltet wird. Für jeden Prozess gibt es einen Eintrag, der alle Informationen enthält, die diesen Prozess charakterisieren. Der Eintrag wird als Prozessdescriptor festgehalten. Ein suspendierter Prozess wird dann durch folgende Informationen beschrieben:

1. Prozessadressraum (Programmdaten, Nutzerdaten, Nutzerkeller, . . .)
2. Eintrag in der Prozesstabelle (PCB)

**Realisierung von Prozessen:** Der Prozessdescriptor wird meist als Record dargestellt. Im Prozessdescriptor enthalten ist der Prozesskontrollblock (PCB), er enthält die Beschreibung des Prozeßzustands unmittelbar vor dessen Suspendierung. Dabei fällt auf: Um Prozesse in beliebiger Reihenfolge aktivieren und deaktivieren zu können, sollte die Verwaltung nicht kellerartig erfolgen.

**Bestandteile eines Prozessdescriptors am Beispiel der MI:**

- Ein eindeutiger Name des Prozesses (fortlaufende Nummer, PID in Unix).
- Der Name des Benutzers, dem der Prozess zugeordnet ist.
- zugeordneter Rechnerkern
- Spezifikation des Ereignisses, auf das der Rechnerkern wartet.
- Ablaufpriorität des Prozesses
- Prozesskontrollblock
- Gegebenenfalls die Zuordnung von Ressourcen

**Bestandteile des Prozesskontrollblocks:**

- Stackpointer
- Inhalte der Register z.B. für die Anfangsadresse und Länge von prozessspezifischen Speicherabbildungstabellen
- Programmstatuswort (PSW)

Achtung, je umfangreicher der Prozesskontrollblock ist, desto “teurer” ist ein Prozesswechsel (siehe unten).
3.3 Prozesskontrolle

**Ausgangspunkt:** Wir wissen jetzt, warum das Konzept der Prozesse wichtig ist, wie Prozesse erzeugt werden (Kapitel 3.1) und welche Kontrollstrukturen das Betriebssystem verwendet, um Prozesse zu beschreiben (Kapitel 3.2). Nun untersuchen wir, wie auf dieser Grundlage Prozesse kontrolliert und gesteuert werden können.

**Zwei Prozessormodi:** Bei der Arbeit des Prozessors werden zwei Prozessormodi unterschieden:

1. **Im Systemmodus (Kernel Mode)** ist der Prozessor dem Betriebssystem bzw. einer Funktion des Betriebssystems zugeordnet.
2. **Im Nutzermodus (User Mode)** ist der Prozessor einem Anwendungsprogramm zugeordnet.


**Frage:** Woher weiß der Prozessor, welcher Modus gerade ausgeführt wird? Dazu gibt es im Programm-Statuswort (PSW) ein Bit, das diesen Modus anzeigt (z.B. 0 für Nutzermodus, 1 für Systemmodus).

**Funktionen des Betriebssystem-Kerns:** Zusammenfassend lassen sich die Haupt-Funktionen des Betriebssystem-Kerns so klassifizieren:

- Support-Funktionen: Dienste, die von Nutzerprozessen in Anspruch genommen werden können (siehe oben).
- Selbstverwaltungs-Funktionen (FCAPS):
  - Fehlermanagement (Fault Management): Erkennung von Fehlerursachen und falls möglich Fehlerbehebung
    **Fehler:** dauernde oder vorübergehende Veränderung/Verfehlung operationaler Vorgaben
    Ziel ist die Erhöhung der Verfügbarkeit. Dazu müssen Fehlerursachen erkannt und Fehlerquellen behoben werden, wobei die Überwachung entweder permanent ablaufen kann oder alternativ Alarme gemeldet werden können.
    ⇒ Überwachung der Geräte/Betriebsmittel und Kontrolle der Einstellungen sind notwendig.
Abrechnungsmanagement (Accounting Management): Erfassung und Protokollierung von Nutzerzugriffen zu Abrechnungszwecken
Sicherheitsmanagement (Security Management): Verschlüsselung von Informationen (z.B. auf Datenträgern), Schutz vor Angriffen, Authentifizierung, Autorisierung

- E/A-Management: Verwaltung von Kanälen und Puffern
- Speichermanagement: Segmentierung, Paging (siehe später)
- Prozessmanagement: Prozesserzeugung und Prozesswechsel

3.3.1 Prozesswechsel (Kontext-Switch)

Ausgangspunkt: Zu gewissen Zeitpunkten wird ein laufender Prozess unterbrochen, und das Betriebssystem ordnet einem anderen Prozess den Zustand “Running” zu. Dafür gibt es im wesentlichen 3 Ursachen:

1. Externe Ereignisse führen die Unterbrechung des aktuell ausgeführten Prozesses herbei, z.B. clock interrupt, I/O interrupt, memory fault
2. Traps, die mit dem aktuell ausgeführten Befehl verbunden sind, rufen eine Unterbrechung hervor. (z.B. zur Behandlung eines Fehlers oder einer Ausnahmebedingung (exception)).
3. Ein Supervisor-Call ruft explizit eine Betriebssystem-Funktion auf, wodurch der aktuelle Befehl unterbrochen wird.

Eine gewöhnliche Unterbrechung übergibt die Kontrolle zunächst einer Unterbrechungsbearbeitung. Dann wird eine Betriebssystem-Routine aufgerufen, die speziell auf den Typ der aufgetretenen Unterbrechung zugeschnitten ist.

Kontext-Switch: Bei einem solchen vollständigen Prozesswechsel (Kontext-Switch) sind folgende Schritte erforderlich:

1. Aktualisierung und Sicherung des Prozesskontrollblocks (PCB) des bisherigen Prozesses, also insb. Sicherung der Zustandsinformationen (z.B. CPU-Register, PSW) und Aktualisierung und Sicherung der Kontrollinformationen (z.B. Scheduling-Informationen)
2. Einfügen des Prozesskontrollblocks des bisherigen Prozesses in die Warteschlange desSchedulers
3. Auswahl eines anderen Prozesses zur Ausführung
4. Wiederherstellung und Aktualisierung des Prozesskontrollblocks des neuen Prozesses (einschließlich der Änderung des Ausführungsstandes auf “Running”)

Beim Kontext-Switch wird der vollständige PCB des alten Prozesses (und damit sein gesamter Kontext) gesichert und der des neuen Prozesses wiederhergestellt. Der Aufwand ist umso größer, je mehr CPU-Register involviert sind.
3.3.2 Unterbrechungen

Motivation: Im Rechnerbetrieb treten häufig zeitlich nicht vorhersehbare Ereignisse ein, die in geeigneter Weise behandelt werden müssen.


1. Einerseits kann man den Prozeß A solange blockieren, bis die Quittung eingetroffen oder der timeout verstrichen ist. Trifft eine Quittung ein wird normal fortgefahren, andernfalls die Nachricht erneut gesendet. Dadurch entstehen offensichtlich relativ lange Leerlaufzeiten für Prozeß A, in denen nur auf die Quittung gewartet wird.


Aber: Für bestimmte Befehle (Programmteile) benötigt der Prozeß das Ereignis, auf das gewartet wird (z.B. Ergebnisse von Anfragen).

Unterbrechung: Eine Unterbrechung ist das Maschinenkonzept für die Behandlung nicht deterministischer, also unvorhersehbarer Abläufe. Ein gerade aktiver Prozeß wird dabei unterbrochen und eine Unterbrechungsbehandlung eingeschoben. Man unterscheidet externe und interne Unterbrechungen. Analog zu Prozeduraufrufen müssen Informationen über den gerade aktiven Prozeß gespeichert werden, um später an dieser Stelle fortfahren zu können. PC und Programmstatuswort PSW müssen, weitere Informationen können, bei der Unterbrechungsbehandlung gesichert werden. Generell ist es notwendig, daß eine Unterbrechung kontrollierbar erfolgt, d.h. der Prozeß muß vollständig in seinem Zustand
Abbildung 3.20: Blockierende und nicht-blockierende Realisierung von Signalen

beschrieben werden können, so daß eine weitere Abarbeitung später an dieser Stelle möglich ist. Je weniger Informationen zu einem bestimmten Zeitpunkt zur Beschreibung des Zustandes notwendig sind, desto “günstiger” ist eine Unterbrechung. In der Realität ist diese Situation meist nicht gegeben, da wenn z.B. ein katastrophaler Fehler auftritt (Division mit Null, Netz funktioniert nicht o.a.), der laufende Prozeß unterbrochen werden muß.


**Interne Unterbrechung (Exception):** Sie wird vom ausgeführten Prozess selbst ausgelöst und kann zur Ursache z.B. eine arithmetische Ausnahme haben. Sie bewirkt ebenfalls eine Unterbrechung der CPU und die Aktivierung der Unterbrechungsbehandlung im Systemkern wie in 3.22.

**Unterbrechungsroutine (Interrupt Handler):** Jede Unterbrechung muss vom Betriebssystem behandelt werden. Die Komponente, die die Unterbrechungsbehandlung durchführt, heißt Unterbrechungsroutine (Interrupt Handler) und gehört zu den Betriebssystem-Funktionen. Sie entscheidet z.B. darüber, ob aufgrund der Unterbrechung der zuletzt ausgeführte Prozeß fortgesetzt werden kann oder zu einem anderen Prozeß gewechselt werden muss.

**Problem:** Die Unterbrechungsroutine wird mit den entsprechenden Privilegien im Systemmodus ausgeführt. Muss Interrupt-bedingt ein Nutzerprozel unterbrochen werden, ist daher ein Wechsel vom Nutzer- in den Systemmodus erforderlich. Da die Unterbrechungsroutine
keinen vollständigen Prozesskontext benötigt, soll der Kontext des zuletzt ausgeführten Befehls noch so lange aufrecht erhalten bleiben, wie noch die Möglichkeit besteht, dass dieser Prozess seine Ausführung wiederaufnehmen kann.

### 3.3.3 Moduswechsel

**Moduswechsel:** Bei einem Moduswechsel sind folgende Schritte erforderlich:

1. Aktualisierung und Sicherung der Zustandsinformationen des Prozesskontrollblocks des gerade aktiven Prozesses (also insb. des Programmstatusworts)
2. Durchführung des eigentlichen Moduswechsels durch Freigabe aller Privilegien
3. Sprung zur Unterbrechungsroutine (Programmzähler wird auf dessen Anfangsadresse gesetzt)

Analog zu Prozeduraufträumen müssen beim Moduswechsel also Informationen über den Status des bisher aktiven Prozesses gespeichert werden, um später an der gleichen Stelle fortfahren zu können. Falls der Interrupt Handler CPU-Register verwendet, so muss er die enthaltenen Daten zuvor selbst auf dem Stack sichern (Callee-saved) und wiederherstellen, bevor er die Kontrolle wieder an den ursprünglichen Prozess zurück gibt (Moduswechsel zurück in den Nutzermodus). Auch wenn nach der Unterbrechungsbehandlung ein anderer als der bisherige Nutzerprozess ausgeführt werden soll, muss die Unterbrechungsroutine die von ihm überschriebenen Register wiederherstellen, da erst dann eine vollständige Kontextsicherung des Prozesses im Rahmen eines Prozesswechsels vorgenommen werden kann (Moduswechsel zurück in den Nutzermodus mit anschließendem Kontext-Switch).

*Ein Moduswechsel ist i.d.R. schneller als ein vollständiger Prozesswechsel, da die Zustandsinformationen nicht gesichert werden (vgl. Abbildung 3.23).*
Abbildung 3.22: Prinzip einer Exception

Abbildung 3.23: Prozess- und Moduswechsel

3.3.4 Konflikte bei Unterbrechungen

**Ursachen für Konflikte:** Bei der Abarbeitung von Unterbrechungen kann es zu Konflikten kommen. Dafür gibt es im Wesentlichen zwei Ursachen:

1. Während der Unterbrechungsbehandlung treten weitere Unterbrechungen auf.
2. Es treffen gleichzeitig (noch vor jeder Unterbrechungsbehandlung) mehrere Unterbrechungswünsche ein.

Lösung von Konflikten: Zur Lösung von Konflikten werden Prioritäten von Unterbrechungen eingeführt. Dabei gilt:

- Benutzerprozesse haben die niedrigste Unterbrechungs-Priorität (0).
- Bei internen Unterbrechungen erhält die zugehörige Unterbrechungsroutine die gleiche Priorität wie der Prozess, der unterbrochen wurde.
- Externen Unterbrechungen sind festgelegte Prioritäten zugeordnet, z.B. von 0 bis 31, wenn 5 Bits für das IPL (siehe unten) zur Verfügung stehen.
- Die Unterbrechung einer Unterbrechungsbehandlung ist möglich, wenn die Priorität der neuen Unterbrechung höher ist, als die der gerade aktiven Unterbrechungsbehandlung. Andernfalls wird der Unterbrechungswunsch zurückgestellt.

Interrupt Priority Level (IPL): Das IPL ist Teil des Programmstatuswortes (PSW) im Prozesskontrollblock und enthält die Unterbrechungs-Priorität zu einem Nutzer- oder Systemprozess bzw. zu einem Interrupt Handler.

3.3.5 Ausführung des Betriebssystems

Frage: Wie wird eigentlich das Betriebssystem selbst ausgeführt? Um diese Frage zu beantworten, wollen wir von den folgenden beiden Aspekten ausgehen:

1. Das Betriebssystem selbst funktioniert in der gleichen Art und Weise wie jede andere gewöhnliche Computersoftware, d.h. es ist ein Programm, das durch den Prozessor ausgeführt wird.

2. Das Betriebssystem muss oft die Kontrolle abgeben, und es hängt vom Prozessor ab, die Kontrolle wieder zuzuweisen.

**Konzept des separaten Kerns (Non-Process Kernel):**  

![Abbildung 3.25: Ausführung des Betriebssystem-Kerns außerhalb jeden Prozesses](image)

**Konzept der Integration in die Nutzerprozesse (Execution within User Process):**  

![Abbildung 3.26: Ausführung des Betriebssystems durch die Prozesswechselroutinen und Betriebssystem-Routinen](image)

**Konzept des prozessbasierten Betriebssystems (Process-based Operating System):**  
*Beispiel 3.8.* Das Prozessmanagement in UNIX (System V) verwendet das Konzept der Integration von Betriebssystem-Funktionen in die Nutzerprozesse, das hier vorgestellt wurde. Folglich existiert ein Nutzermodus und ein Systemmodus, und UNIX nutzt zwei Arten von Prozessen: Systemprozesse und Nutzerprozesse:


- Nutzerprozesse laufen im Nutzermodus ab, um Programme auszuführen, und im Systemmodus, um Befehle, die zum Kern gehören, auszuführen. Der Wechsel vom Nutzer- in den Systemmodus wird dabei durch einen Systemaufruf ausge löst. Ursache dafür ist eine interne oder externe Unterbrechung.

Bezüglich der Prozeßzustände teilen wir den Zustand “Running” noch einmal in 2 Zustände auf: “User Running” und “Kernel Running”.

Als Zustandsübergangsdiagramm ergibt sich ein Modell mit 9 Zuständen:
Kapitel 3. Prozesse

Abbildung 3.29: 9-Zustands-Modell (UNIX)

Dieses Modell ist eine Erweiterung des 7-Zustands-Modells (Abbildung 3.13) dahingehend, daß

1. der Zustand “Running” in “Kernel Running” und “User Running” aufgeteilt wird und

Ferner sind einige Begriffe variiert:
- Statt dem bekannten “Suspend” wird hier “Swapped” verwendet.
- Statt dem bekannten “Blocked” wird hier “Asleep” verwendet.

Die Prozeßbeschreibung wird durch das UNIX Process Image realisiert. Dieses besteht aus 3 Teilen:

1. User-Level Context mit Prozeßtext (ausführbare Maschinenbefehle des Programms), Prozeßdaten (Daten, die vom Programm in diesem Prozeß verwendet werden), dem User Stack (Nutzerkeller) und Shared Memory.
2. Register Context mit Programmcodierer (PC), Prozessorstatusregister, Stackpointer und Hardwareabhängigen Registern für allgemeine Zwecke.

Die Generierung von Prozessen wird in UNIX im Systemmodus mittels der Operation FORK ausgeführt. Ruft ein Prozeß eine FORK-Operation auf, so führt das Betriebssystem die folgenden Funktionen aus:

1. Es stellt in der Prozeßtabelle einen Bereich für den neuen Prozeß bereit.
2. Es ordnet dem Kindprozeß eine eindeutige ID zu.

3. Es macht eine Kopie des Process Images des Elternprozesses - ausgenommen sind dabei jegliche Teile des Shared Memory.

4. Das Betriebssystem aktualisiert (d.h. erhöht) die Zähler aller Dateien, die dem Elternprozeß gehören, um anzuzeigen, daß ein neuer Prozeß nun auch Zugriff hat.

5. Das Betriebssystem überführt den neuen Prozeß in den Zustand “Ready”.


Alle diese Schritte werden im Systemmodus des Elternprozesses ausgeführt. Danach kann - als Bestandteil einer Dispatcherroutine - eine der folgenden Alternativen ausgeführt werden:


2. Die Kontrolle wird an den Kindprozeß gegeben. Dieser Kindprozeß führt die Abarbeitung an der gleichen Stelle wie sein Elternprozeß fort, d.h. nach dem RETURN des FORK-Aufrufs.

3. Die Kontrolle wird einem ganz anderen Prozeß gegeben. Dabei bleiben sowohl der Eltern- als auch der Kindprozeß im Zustand “Ready”.

4

Threads

- Multithreading
- Threadzustände
- User-level-Threads
- Kernel-level-Threads
- Kombinierte Konzepte
- Andere Formen paralleler Abläufe

Inhaltsangabe

<table>
<thead>
<tr>
<th>Kapitel</th>
<th>Seitenzahl</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Multithreading</td>
<td>64</td>
</tr>
<tr>
<td>4.2 Threadzustände</td>
<td>67</td>
</tr>
<tr>
<td>4.3 User-level-Threads (ULT)</td>
<td>68</td>
</tr>
<tr>
<td>4.4 Kernel-level-Threads (KLT)</td>
<td>70</td>
</tr>
<tr>
<td>4.5 Kombinierte Konzepte</td>
<td>70</td>
</tr>
<tr>
<td>4.6 Andere Formen paralleler Abläufe</td>
<td>71</td>
</tr>
</tbody>
</table>
Motivation: Bislang wurden dem Konzept eines Prozesses zwei unterschiedliche Aufgaben zugeordnet:


2. Ein Prozess kann als Einheit, die eine gewisse Aufgabe erledigt (Ausführung eines Programms), betrachtet werden. D.h. ein Prozess ist ein Ausführungspfad (Trace) durch ein oder mehrere Programme. Diese Ausführung kann unterbrochen werden, weshalb einem Prozess Ausführungszustände ("Running", "Ready", ...) zugeordnet werden, sowie eine Priorität, welche die Dringlichkeit der Ausführung im Betriebssystem steuert.

Diese beiden Aufgaben sind vollkommen unabhängig voneinander! Deshalb können beide Aufgaben innerhalb eines Betriebssystems auch unabhängig voneinander erfüllt werden.

Zu diesem Zweck wird folgende Begriffswelt eingeführt:

- Die Einheit, die den Eigentümer von Ressourcen bezeichnet, wird weiter Prozess genannt. ABER:

- Eine Einheit, die eine gewisse Aufgabe erledigt, wird Thread (Faden) oder leichtgewichtiger Prozess genannt (lightweight process).

Bislang entsprach einem Prozess auch ein Thread. (Sonderfall Singlethread →1:1-Relation) Nun wollen wir einen Prozess durch die Ausführung mehrerer Threads realisieren. Es werden also einem Prozess mehrere Threads (Thread1, Thread2, ..., Threadn) in einer 1:n-Relation zugeordnet.
Dieser Sachverhalt wird durch einen neuen Begriff beschrieben, das Multithreading.

4.1 Multithreading

Beim Zusammenspiel von Threads und Prozessen können 4 Fälle unterschieden werden:

Wir wollen im Folgenden Multithread-Umgebungen betrachten.

Mit einem Prozess ist dabei assoziiert:
- ein virtueller Adressraum, in dem das Prozessimage abgelegt wird
- ein exklusiver Zugang zum Prozessor, zu anderen Prozessen (bei der Interprozeßkommunikation), zu Dateien und zu E/A-Geräten sowie E/A-Kanälen
- Zugriffsrechte

D.h. der Prozess ist die Einheit, die auf Ressourcen zugreift und Zugriffsrechte klärt.

Innerhalb eines Prozesses werden ein oder mehrere Threads ausgeführt, die dessen virtuellen Adressraum nutzen und im Bedarfsfall gemeinschaftlich auf diesen zugreifen können. Es ist im Gegensatz dazu wichtig zu beachten, dass mehreren Prozessen immer disjunkte virtuelle Adressräume zugeordnet werden.

Jeder der Threads innerhalb eines Prozesses bekommt folgende Bestandteile zugeordnet:
- einen Zustand der Threadausführung ("Running", "Ready", ...)
- falls sich der Thread nicht im Zustand "Running" befindet: Thread-Kontext (Speicherplatz nötig), insbesondere Programmzähler (PC) sichern
- zwei Stacks für die Ausführung (user stack; kernel stack)
- "etwas" Speicherplatz für lokale Variablen
- Zugriff auf Speicher und Ressourcen des zugehörigen Prozesses, die mit allen anderen Threads dieses Prozesses geteilt werden.

Abbildung 4.1: Die 4 Fälle beim Zusammenspiel von Threads und Prozessen
Im Folgenden soll der Unterschied zwischen einem Prozess und einem Thread aus der Sicht des Prozessmanagements betrachtet werden.

**Abbildung 4.2: Singlethreading-Prozessmodell vs. Multithreading-Prozessmodell**

**Singlethreading:** Zur Laufzeit werden die Prozessorregister vom Prozess kontrolliert. Be- findet sich der Prozess nicht im Zustand “Running”, so wird der Inhalt der Register abge- speichert.

**Multithreading:** Zuordnung von PCB und User Adress Space zum Prozess bleibt, aber jeder Thread bekommt zusätzlich einen Kontrollblock, den sog. TCB, der für jeden Thread die Register, den Zustand und andere thread-bezogene Informationen enthält.

Dieses Konzept führt dazu, daß alle Threads eines Prozesses den gleichen Adreßraum nutzen und auf die gleichen Daten zugreifen.

Z.B. wenn ein Thread das Ergebnis einer Rechnung im Adreßraum ablegt, so können alle anderen Threads dieses Prozesses auch darauf zugreifen. ⇒ Sicherheitsprobleme!

**Vorteil des Threadkonzepts:** verschiedene Leistungssteigerungen

- Zur Generierung eines neuen Threads in einem existierenden Prozess ist wesentlich weniger Zeit notwendig, als zur Generierung eines neuen Prozesses, teilweise Faktor 10!

Beispiel: File-Server eines LANs

- Für jede Dateianfrage genügt es, anstelle eines neuen Prozesses, einen neuen Thread zu generieren.
- In kurzen Zeitabständen werden viele Threads generiert und terminiert.

- Threads machen die Kommunikation zwischen verschiedenen ausgeführten Program- men effizienter, da sie untereinander kommunizieren können (z.B.: über abgelegte Da- ten), ohne den Betriebssystem-Kern zu involvieren.
– Kommt es zu Blockierungen, so entsteht weniger Wartezeit, wenn nur einzelne Threads blockiert sind und andere abgearbeitet werden können.

– Kontextwechsel unter Threads innerhalb eines Prozesses erfordern weniger Zeit als Kontextwechsel von Prozessen.

**Besonderheiten des Threadkonzepts:**

– Durch den gemeinsamen Adressraum sind Daten einzelner Threads eines Prozesses bezüglich anderer Threads nicht sicher, da von jedem anderen Thread innerhalb des Prozesses auf diese Daten zugegriffen werden kann.

– Erfolgt ein Swapping des übergeordneten Prozesses, so werden alle Threads gleichzeitig mit ausgelagert. Und analog: Terminiert der übergeordnete Prozess, so terminieren mit ihm alle Threads.

### 4.2 Threadzustände

Analog zu den Prozessen sind die grundlegenden Zustände eines Threads: “Running”, “Ready” und “Blocked”. Es macht keinen Sinn, Zustände zur Suspendierung den Threads zuzuordnen, da die Auslagerung ein Konzept von Prozessen ist. (Grund: gemeinsamer Adressraum aller Threads eines Prozesses)


Als Zustandsübergangsdiagramm ergibt sich für die Threads:

![Thread-Zustandsübergangsdiagramm](#)

Abbildung 4.3: Thread-Zustandsübergangsdiagramm

Annahme: Der Prozess befindet sich im Zustand “Running”. Bei Ein-Prozessor-Systemen kann zu einem Zeitpunkt nur einem Thread der Prozessor zugeteilt werden. Zu beachten ist dabei, ob die Blockierung eines Threads durch das Blockieren des zugehörigen Prozesses bedingt ist, oder ob alle anderen Threads weiter abgearbeitet werden können. Es können z.B. einzelne Threads auf ein Ereignis (E/A-Geräte,...) warten und blockiert sein, während gleichzeitig andere Threads nicht blockiert sind.

Insgesamt kann man sich also überlegen, Zustände nach den Kriterien global (für einen gesamten Prozess) und lokal (für die einzelnen Threads) zu unterscheiden.
** Threadarten  
Bei Threads unterscheidet man zwei Arten:

### 4.3 User-level-Threads (ULT)

- Bei User-level-Threads ist das Threadmanagement Aufgabe der Anwendung, und der Betriebssystem-Kern braucht von der Existenz solcher Threads gar nichts zu wissen.
- Jede Anwendung kann als multithreaded programmiert werden, indem sogenannte Threadbibliotheken genutzt werden.
- Diese Threadbibliotheken sind eine Sammlung von Routinen, die gerade zum Zwecke des User-level-Thread-Managements bereitgestellt werden. Insbesondere enthalten die Threadbibliotheken Code für:
  - die Generierung und Terminierung von Threads,
  - die Nachrichten- und Datenübermittlung zwischen Threads,
  - das Scheduling der Threadausführung, sowie
  - das Sichern und Löschen von Threadkontexten.

![Abbildung 4.4: User-level-Thread-Modell](image)

- Die Anwendung kann nun jederzeit einen neuen Thread generieren, der innerhalb desselben Prozesses läuft. Dies geschieht durch Zugriff auf die entsprechenden Routinen in der Threadbibliothek mittels eines Prozedurauftrufs. Die Threadbibliothek generiert eine Datenstruktur für den neuen Thread und übergibt die Kontrolle dann an einen Thread innerhalb des Prozesses, der sich im Zustand “Ready” befindet - die Auswahl dieses Threads erfolgt mittels Schedulingalgorithmen.

**Bemerkung:** Während bei dieser Ausführung die Kontrolle an die Threadbibliothek übergeben wird, muß der Kontext des aktuellen Threads natürlich gesichert sein und zur Weiterausführung dieses Threads wird der Kontext wieder geladen. (Kontext: Inhalt der Nutzerregister, Programmzähler, Stack-Pointer)
Vorteile solcher User-level-Threads liegen in folgendem begründet:

- Die Threadwechsel + Threaderzeugung/-terminierung benötigen keine Privilegien des Systemmodus, da das Threadmanagement innerhalb des Nutzeradreßraumes eines einzelnen Prozesses stattfindet. Damit wird Zeit für Modiwechsel (System \( \rightarrow \) Nutzermodus) eingespart.

- Das auf BS-Ebene implementierte Scheduling kann auf Thread-Ebene anwendungsspeziﬁsch realisiert werden.

- Dieser Mechanismus kann in jedem Betriebssystem ablaufen, ohne daß Betriebssystem-Kern-Erweiterungen vorgenommen werden müssen.

Im Prinzip wird durch User Level Threads nur ein Multithreading simuliert. Tatsächlich werden alle Konzepte von Anwendungsprogrammierern implementiert, wodurch Vorteile des Thread-Konzepts verlorengehen.

Nachteile bestehen in Folgendem:

1. Wird ein Thread blockiert, so blockiert gleichzeitig der gesamte Prozess, d.h. auch die Menge aller anderen Threads.


Abhilfe zu 1. Es wird die Technik des “Jacketings” genutzt: ein blockierender Systemaufruf wird dabei in einen nicht blockierenden Systemaufruf transformiert. Z.B. E/A-Gerät erst antesten, ob es “busy” ist - falls ja, dann zu anderem Thread wechseln und später neu probieren anstatt direkt zu blockieren.

Abhilfe zu 2. Man sollte also überlegen, ob anstelle der User-level-Threads nicht doch vielleicht separate Prozesse verwendet werden sollten, auch wenn dadurch wieder Prozesswechsel nötig werden.

Fazit: User Level Threads sind relativ einfach umsetzbar, realisieren aber wenige Vorteile.
4.4 Kernel-level-Threads (KLT)

- In einer reinen Kernel-level-Thread-Umgebung wird das Threadmanagement vom Betriebssystem-Kern durchgeführt,
- dabei kann jede Anwendung als multithreaded programmiert werden, das Threadmanagement wird vom Betriebssystem-Kern übernommen.

Vorteile der KLTs:
- Im Gegensatz zu echten Prozessen können KLTs deutlich schneller erzeugt, gewechselt und terminiert werden.
- Der Betriebssystem-Kern kann mehrere Threads eines Prozesses auf verschiedenen Prozessoren einer Multiprozessorumgebung ausführen.
- Falls ein Thread blockiert ist, so kann die Kontrolle einem anderen Thread desselben Prozesses übergeben werden, der sich im Zustand “ready” befindet.

Nachteile der KLTs:
- Wird die Kontrolle von einem Thread an einen anderen Thread desselben Prozesses übergeben, so ist jedesmal ein Moduswechsel erforderlich, d.h. diese Ausführung führt zu einer Verlangsamung.

4.5 Kombinierte Konzepte

- Einige Betriebssysteme, wie z.B. Solaris, stellen ein kombiniertes Konzept bereit.
- Die Generierung von Threads wird im Nutzerraum durchgeführt, die ULTs werden dann auf eine beliebige, auch kleinere oder größere Anzahl von KLTs abgebildet.
Andere Formen paralleler Abläufe

Motivation:

– Traditionell gesehen wird ein Computer immer als Maschine betrachtet, die Befehle sequentiell abarbeitet.

– In Wirklichkeit werden aber bei jedem Computer auch parallele Abläufe vorkommen, z.B. Signale gleichzeitig generiert oder beim Pipelining Abläufe gleichzeitig ausgeführt.

Abschließend sollen die verschiedenen Relationen zwischen Threads und Prozessen noch einmal zusammengefasst werden:

*Threads:* *Prozesse*

1:1 jedem Thread ist ein Prozess mit eigenem Adreßraum und entsprechenden Ressourcen zugeordnet (z.B. ältere UNIX-Versionen)

n:1 einem Prozess ist ein Adreßraum und eine gewisse Menge von Ressourcen zugeordnet; innerhalb dieses Prozesses können verschiedene Threads generiert und ausgeführt werden (z.B. Windows NT, Solaris, ...)

Neben diesen bereits betrachteten Ansätzen ist auch denkbar:

1:n ein Thread kann von einer Prozessumgebung zu einer anderen migrieren (z.B. Clouds Operating System, Emerald System, ...)

n:m kombiniert die n:1 und 1:n-Ansätze (z.B. Operating System TRIX)

4.6 Andere Formen paralleler Abläufe

Abbildung 4.7: Thread-Modell für kombinierte Konzepte

– Dabei können mehrere Threads innerhalb desselben Prozesses auf einem Mehrprozessorrechner auch parallel ausgeführt werden und

– ein blockierender Thread blockiert nicht den gesamten Prozess.

– Ziel ist es in jedem Fall, die Vorteile der ULT- und KLT-Ansätze zu kombinieren und die Nachteile zu minimieren.
Mit den sinkenden Kosten stellen Computer immer mehr Möglichkeiten für Parallelität bereit, wodurch ihre Leistung steigt.

Es gibt drei verbreitete Ansätze, die Parallelität durch das Replizieren von Prozessoren ermöglichen:
- Master/Slave Architektur
- Symmetrisches Multiprocessing (SMP)
- Cluster

**Einordnung**

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>SISD</td>
</tr>
<tr>
<td>MI</td>
<td>MISD</td>
</tr>
</tbody>
</table>

| Data entspricht processor, d.h. Verarbeitung der Daten |
| Instruction = Befehle, die sequentiell eintreffen |
| s...single  m...multiple |

Abbildung 4.8: Einordnung von “Parallele Prozessoren”

**SISD**: klassischer Rechner mit 1 Prozessor, bei dem über Adress- und Datenbus Befehle nacheinander bereitgestellt werden.

→ keine Parallelität in der Ausführung (Pipelining ausgenommen → TGI)

**MISD**: Situation: Daten/Befehle werden über mehrere Busse bereitgestellt, es ist jedoch nur ein Prozessor vorhanden, der diese Daten verarbeiten kann (Gegenteil des von Neumann’schen Flaschenhalses!) → Diese Architektur wurde nie implementiert. **SIMD, MIMD**: tatsächliche parallele Verarbeitung von Daten auf mehreren Prozessoren.

**Resümee**: Hier bei parallelen Prozessoren ist nur SIMD und MIMD interessant.

**Parallele Prozessoren**

Der SMP-Ansatz hat die Vorteile Verteilter Systeme (Fehlertransparenz, keinen Flaschenhals beim Master, ...) Aber er verkompliziert das Betriebssystem: Es muß gewährleistet werden, dass zwei Prozessoren nie denselben Prozess auswählen. Ferner müssen Ressourcen fair zugeordnet werden.

Die SIMD-Komponente (1 Bus, m Prozessoren) kann jedoch kaum Abhilfe bezüglich des von
Andere Formen paralleler Abläufe

Abbildung 4.9: Parallele Prozessoren

Neumann'schen Flaschenhalses schaffen. Steht für alle Prozessoren nur ein Bus bereit, kann die Problematik eher verschärft werden. Ist dagegen pro Prozessor ein Bus vorhanden, so liegt eine mit SISD vergleichbare Situation vor.

Prinzip der SMP-Organisation

Abbildung 4.10: Prinzip der SMP-Organisation

- Jeder Prozessor besitzt seine eigene Kontrolleinheit, seine eigene ALU und eigene Register.
Kapitel 4. Threads
Scheduling

- Scheduling und Scheduling-Verfahren als Basis für Multiprogramming
- Zuteilung des Prozessors an mehrere Prozesse

Inhaltsangabe

5.1 Das Prinzip des Schedulings .............................................. 76
  5.1.1 Varianten des Schedulings ........................................... 76
  5.1.2 Anforderungen an einen Scheduling-Algorithmus ................. 76
  5.1.3 Scheduling vs. Dispatching ......................................... 79
5.2 Scheduling-Algorithmen ................................................... 79
  5.2.1 Begriffe ..................................................................... 79
  5.2.2 Nicht-preemptive Scheduling-Algorithmen ......................... 80
  5.2.3 Preemptive Scheduling-Algorithmen ............................... 83
  5.2.4 Priority Scheduling (PS) ............................................. 88
  5.2.5 Multilevel Feedback Queueing ...................................... 88
5.3 Prozesswechsel ............................................................... 90
5.4 Arten des Schedulings ....................................................... 90
In den vorhergehenden Abschnitten ist oft die Situation aufgetreten, dass mehrere Prozesse (quasi) gleichzeitig ausgeführt werden können. In solchen Fällen muss das Betriebssystem entscheiden, welcher Prozess als erster ausgeführt werden soll. Denjenigen Teil des Betriebssystems, der diese Entscheidung übernimmt, bezeichnet man als Scheduler.

5.1 Das Prinzip des Schedulings

Scheduling wird notwendig, sobald
1. mindestens ein Prozess $A$ im Zustand \textit{running} ist und
2. ein oder mehrere andere Prozesse im Zustand \textit{ready} darauf warten, dass ihnen die CPU zugeteilt wird.

Tritt die Freigabe des Prozesses ein (oder wird sie erzwungen), so stellt sich die Frage, welcher der Prozesse im Zustand \textit{ready} nun dem Prozessor zugewiesen werden sollte. Dieser Entscheidung sollte offensichtlich eine Strategie (mittels eines Algorithmus) zugrunde liegen.

Die \textit{Aufgabe eines Schedulers} ist es, auf der Basis einer vorgegebenen Strategie die Entscheidung zu treffen, welcher Prozess als nächster dem Prozessor zugewiesen werden soll. Seine Aufgabe ist es jedoch nicht, einen Mechanismus für die eigentliche Zuweisung zur Verfügung zu stellen (dies wird außerhalb des Schedulers, aber auch im Betriebssystem, nämlich in einem \textit{Dispatcher}, realisiert, wie in Kapitel 5.3 erläutert).

5.1.1 Varianten des Schedulings

Beim Scheduling unterscheidet man zwei grundlegende Varianten:


2. \textbf{Preemptives Scheduling}: Bei der Verwendung von preemptivem Scheduling können Prozesse zu jedem Zeitpunkt unterbrochen (suspendiert) werden, so dass ein anderer Prozess zur Ausführung kommen kann. Dieses Scheduling kann jedoch zu kritischen Abläufen oder Bereichen führen, die eine aufwendige Behandlung (z.B. mittels Semaphore oder Monitoren) notwendig machen (alle heutigen Systeme).

5.1.2 Anforderungen an einen Scheduling-Algorithmus

Die Anforderungen an einen Scheduling-Algorithmus hängen sehr stark vom System ab, in dem Prozesse verwaltet werden sollen. Es gibt keinen optimalen Scheduling-Algorithmus für alle möglichen Systeme. Batch-Systeme erfordern einen hohen Durchsatz wohingegen Dialog-Systeme auf schnelle Antwortzeiten hin optimiert werden müssen. Im Wesentlichen unterscheidet man daher drei Klassen von Systemen:

– Batch-System (Stapelverarbeitungssystem)
Das Prinzip des Schedulings

Interaktives System und

Echtzeitsystem

Die folgenden Anforderungen sind allen Systemen gemein:

Fairness  Jeder Prozess sollte einen gerechten Anteil an der Prozessorzeit erhalten (insbesondere sollte ein Prozess nur endliche Zeit warten müssen).

Policy Enforcement  Das gewählte Verfahren wird stets durchgesetzt, d.h. es gibt keine Ausnahmen.

Balance  Alle Teile des Systems sind (gleichmäßig) ausgelastet.

Datensicherheit  Es können keine Daten oder Prozesse verloren gehen.

Skalierbarkeit  Die mittlere Leistung wird bei wachsender Last (Anzahl von Prozessen) beibehalten. Es gibt keine Schwelle, ab der das Scheduling nur noch sehr langsam oder gar nicht mehr funktioniert.

Effizienz  Der Prozessor sollte möglichst immer vollständig ausgelastet sein.


- Batch-Systeme (Stapelverarbeitungssysteme)
  
  Durchsatz  Maximierung der Prozesse pro Zeiteinheit.
  
  Verweildauer  Minimierung der Zeit vom Start bis zur Beendigung eines Stapel-Prozesses.
  
  Prozessorauslastung  Konstante und maximale Auslastung der CPU.

- Interaktive Systeme

  Antwortzeit  Die Antwortzeit für die interaktiv arbeitenden Benutzer sollte minimal sein (→ möglichst schnelle Abarbeitung von Nutzerprozessen)
  
  Verhältnismäßigkeit  Der Algorithmus soll auf die Erwartungen des Benutzers eingehen (z.B. darf die Reaktion auf einfaches Klicken eines Icons nicht zu lange dauern).

  Interaktion  Die Anzahl interagierender Nutzer sollte maximal sein.

- Echtzeitsysteme

  Sollzeitpunkte  Zeitschranken müssen eingehalten werden, d.h. gewisse Berechnungen (Prozesse) müssen zu bestimmten Zeitpunkten beendet sein.

  Vorhersagbarkeit  Das Verhalten des Systems muss vorhersagbar sein.

Zu beachten ist, dass im praktischen Betrieb nicht alle Anforderungen gleichzeitig erfüllbar sind. So stehen etwa eine geringe Antwortzeit und maximaler Durchsatz/maximale Auslastung im Widerspruch zueinander. Für jeden Scheduling-Algorithmus muss also entschieden werden, auf welchem System er eingesetzt werden soll, d.h. auf welchen Kriterien der
Schwerpunkt liegt. Es kann sogar gezeigt werden, dass jeder Scheduling-Algorithmus, der eines dieser Kriterien betont, dafür ein anderes vernachlässigt. Schließlich ist die verfügbare Rechenzeit begrenzt.

Bei nutzerorientierten Systemen sind die systemorientierten Kriterien von geringerer Bedeutung.

Ein bedeutendes Hilfsmittel des Prozess-Management ist die Einführung von Prioritäten. Dabei wird jedem Prozess eine Priorität $Q_i$ zugeordnet, und der Scheduler gibt stets einem Prozess mit jeweils höchster Priorität den Vorrang, d.h. falls $Q_i > Q_j$, so wird $PQ_i$ zuerst abgearbeitet.

Zu diesem Zweck muss eine Erweiterung der Realisierung, d.h. Speicherung von Prozessen im Zustand “ready” vorgenommen werden. Die Ready-Queue wird nun in $n + 1$ Ready-Queues $RQ_0, ..., RQ_n$ aufgespaltet, falls es $n$ Prioritäten und die Priorität $0$ gibt (siehe Abbildung 5.1).

Abbildung 5.1: Aufspaltung in mehrere Ready-Queues für Prioritäten

Muss nun ein neuer Prozess dem Prozessor zugeordnet werden, so wird dieser aus $RQ_0$ gewählt - befinden sich keine Prozesse in dieser Queue, so wird einer aus $RQ_1$ genommen und so weiter.

**Problem:** Prozesse mit niedrigerer Priorität könnten verhungern, falls stets höherpriore Prozesse vorhanden sind.

**Lösung:** Die Prioritäten der Prozesse können sich mit der Zeit ihrer Abarbeitung ändern.

Unabhängig von den Prioritäten existiert eine sogenannte Auswahlfunktion (Selection Function), die bestimmt, welcher der Prozesse im Zustand “Ready” als nächstes ausgeführt wird. Den zeitlichen Aspekt charakterisiert dabei ein Entscheidungsmodus (Decision Mode). Dabei unterscheidet man zwei Kategorien:

**Nonpreemptive Modi (Nichtunterbrechende Modi):** Befindet sich ein Prozess im Zustand
“Running”, so wird seine Ausführung fortgesetzt, bis er terminiert oder durch ein Warten auf eine Betriebsmittelanforderung (z.B. E/A) blockiert wird.

**Preemptive Modi (Unterbrechende Modi):** Eine gerade in Ausführung befindlicher Prozess kann durch das Betriebssystem unterbrochen werden und damit wieder in den Zustand “ready” überführt werden. Gründe für eine solche Unterbrechung können z.B. in folgenden Dingen gesehen werden:

- wenn ein neuer Prozess ankommt
- wenn eine Unterbrechung auftritt, die einen blockierten Prozess mit höherer Priorität in den Zustand “ready” zurückbringen kann
- oder periodisch durch Uhrenunterbrechungen.

Die Preemptiven Modi verursachen dabei einen größeren Overhead als die Nichtpreemptiven - jedoch lässt sich damit eine bessere Prozessararbeitung aus Sicht des gesamten Systems erzielen.

### 5.1.3 Scheduling vs. Dispatching

Hier soll noch einmal kurz der Unterschied zwischen der Bedeutung der Begriffe Scheduling und Dispatching erläutert werden.

**Scheduling:** Unter Scheduling versteht man die Auswahl eines rechenbereiten Prozesses (Zustand “ready”) und die damit gleichzeitige Unterbrechung (bzw. Beendigung) des aktuell abgearbeiteten Prozesses.

**Dispatching:** Unter Dispatching versteht man die Realisierung des eigentlichen Kontextwechsels. Die Information, welcher Prozess als nächster abgearbeitet werden soll, wird dem Dispatcher vom Scheduler mitgeteilt.

### 5.2 Scheduling-Algorithmen

#### 5.2.1 Begriffe

Bevor im Detail auf die verschiedenen Scheduling-Algorithmen eingegangen wird, sollen im Folgenden noch einige Begrifflichkeiten geklärt werden.

**Ankunftszeit** Der Zeitpunkt ab dem ein Prozess existiert.

**Startzeit** Der Zeitpunkt, an dem der Prozess zum ersten mal dem Prozessor zugewiesen wird.

**Verweildauer** Die Zeitdifferenz der Ankunftszeit eines Prozesses im System und seiner Terminierung. Dieser Begriff wird vor allem im Hinblick auf Stapelverarbeitungssysteme verwendet.

**Antwortzeit** Dieser Begriff spielt vor allem in interaktiven Systemen eine große Rolle. Die Antwortzeit bezieht sich auf die Zeitspanne zwischen der Eingabe einer Anfrage und dem Beginn des Empfanges einer Antwort. Ein Prozess kann u.U. also schon mit der Ausgabe einer Antwort auf eine Benutzeranfrage beginnen, während die Anfrage sich aber noch
in Bearbeitung befindet.

**Achtung:** Der Begriff der Antwortzeit wird in der Literatur sehr oft auch anstelle der Verweilzeit verwendet. In diesem Fall versteht man unter der Antwortzeit ebenfalls die Zeitdifferenz der Ankunftszeit eines Prozesses im System und seiner Terminierung!

**Bedienzeit** Die Bedienzeit entspricht der Zeitspanne, in der der Prozessor einem Prozess zugeordnet wird, um den Prozess vollständig abzuarbeiten. Die Bedienzeit entspricht also der Zeitspanne, die ein Prozess bis zu seiner Terminierung im Zustand “running” verbringt (Gesamtbedienzeit). Betrachtet man die Bedienzeit eines schon existierenden Prozesses zu einem bestimmten Zeitpunkt, spricht man oft auch von der **Restbedienzeit** eines Prozesses.

**Wartezeit** Die Wartezeit entspricht der Zeitspanne, die ein Prozess im System verbringt, ohne dass der Prozess ausgeführt wird (nicht im Zustand “running”).

**Beendigungszeit** Der Zeitpunkt an dem der Prozess terminiert ist.

Es gilt

\[ \text{Verweildauer} = \text{Bedienzeit} + \text{Wartezeit} \]

Oft findet man im Zusammenhang mit Scheduling-Algorithmen Informationen über das Verhältnis von Verweildauer \( T_v \) zur Bedienzeit \( T_b \). Der Quotient \( \frac{T_v}{T_b} \) wird auch als **normalisierte Verweildauer** bezeichnet. Diese Kenngröße beinhaltet also die Information über die relative Verzögerung eines Prozesses. Für Prozesse mit langer Bedienzeit kann man in der Regel auch höhere Verzögerungen tolerieren. Natürlich sollen \( T_v \) und \( T_b \) für sich alleine so klein wie möglich ausfallen. Der Quotient sollte optimalweise einen Wert nahe 1 ergeben. Betrachtet man einen Scheduling-Algorithmus in Bezug auf seine normalisierte Verweildauer, so lässt sich festhalten, dass ein Anwachsen des Quotienten auf ein sinkendes Bedienlevel hindeutet.

### 5.2.2 Nicht-preemptive Scheduling-Algorithmen

**First Come First Served (FCFS)**

*First Come First Served (FCFS),* auch bekannt als *First In First Out (FIFO),* ist der einfachste nicht-preemptive Scheduling-Algorithmus. Ein rechenbereiter Prozess stellt sich in der Ready-Queue einfach hinten an, und bei frei werdendem Prozessor kann der jeweils älteste Prozess in den Zustand “running” überführt werden. Folgendes Beispiel veranschaulicht den Algorithmus:

<table>
<thead>
<tr>
<th>Prozess</th>
<th>Ankunftszeit</th>
<th>Verweildauer ( T_v )</th>
<th>Bedienzeit ( T_b )</th>
<th>Quotient ( \frac{T_v}{T_b} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>1.6</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>4</td>
<td>( \emptyset )</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Mittlere Verweildauer: 5.2
Mittlere Normalisierte Verweildauer \( \frac{T_v}{T_b} \): 3.12
Abbildung 5.2: Beispiel mittlere Verweildauer für FCFS

**Problem:** Blockierungen von E/A-Geräten werden bei dieser Prozessorzuteilung nur schlecht abgefangen. (Prozesse, die E/A-Zugriffe beanspruchen, warten in der Ready-Queue, während die E/A-Geräte nicht genutzt werden)

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

*Shortest Process Next (SPN)*, auch bekannt als *Shortest Job First (SJF)*, gehört zu den nonpreemptiven Modi. Dabei wird jeweils der Auftrag ausgewählt, bei dem die kürzeste Abarbeitungszeit erwartet wird.

Die Wirkungsweise von SJF kann an folgendem Beispiel nachvollzogen werden:

<table>
<thead>
<tr>
<th>Prozess</th>
<th>Ankunftszeit</th>
<th>Verweildauer $T_v$</th>
<th>Bedienzeit $T_b$</th>
<th>Quotient $\frac{T_v}{T_b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>1.8</td>
</tr>
<tr>
<td>$P_5$</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \emptyset \ 4.2 \ \emptyset \ 2.4 \ \emptyset \ 1.86 \]

Abbildung 5.3: Beispiel mittlere Verweildauer für SJF
Kapitel 5. Scheduling

Da das Scheduling (ebenso wie bei FCFS) nicht preemptiv ist, braucht erst nach der Abarbeitung des ersten Prozesses wieder eine Auswahl getroffen zu werden. Im Mittel werden die Prozesse 2,4 Sek. abgearbeitet, verbringen im Mittel aber nur 4,2 Sek. im System. Die Wartezeit eines Prozesses beträgt somit im Mittel 1,8 Sek. (Vgl.: FCFS: 5,2 Sek. mittlere Verweildauer bei 2,4 mittlerer Bearbeitungszeit, also 2,8 Sek. mittlere Wartezeit.)

SJF bringt jedoch einige Probleme mit sich:

**Problem 1:** Woher soll das Betriebssystem wissen, mit welcher Abarbeitungsdauer es bei dem Auftrag zu rechnen hat?

**Mögliche Lösung:** Bei gleichartigen Aufträgen den Mittelwert der bereits abgearbeiteten Aufträge betrachten.

**Problem 2:** Starvation (Verhungern) der längeren Prozesse. (Starvation kann natürlich nur auftreten, wenn nicht alle abzuarbeitenden Prozesse von Beginn an bekannt sind!)

**Vergleich der Strategien FCFS und SJF**

- **FCFS**
  - einfacher zu implementieren
  - Fair! (Wer am längsten wartet kommt als nächstes dran, kein Prozessverhungern)

- **SJF**
  - Günstiger bzgl. mittlerer Verweildauer, sogar optimal bei nicht-preemptiven Strategien
  - Nachteile: Vorteile von FCFS nicht vorhanden; der Implementierer von Scheduling-Algorithmen hat keinen Anhaltspunkt, wie lange die Ausführung eines Prozesses dauern wird.

**Optimalität von SJF** SJF ist optimal bei nicht-preemptiven Scheduling-Algorithmen bzgl. der mittleren Verweildauer.

**Beweis**

SJF berechnet zu einer Folge von $n$ Jobs mit Ausführungszeiten $t_1, t_2, \ldots, t_n$ eine Abarbeitungsreihenfolge mit minimaler Verweildauer (Antwortzeit). Die mittlere Verweildauer bei Abarbeitung der Jobs in der Reihenfolge $t_{i_1}, t_{i_2}, \ldots, t_{i_n}$ ($i_1, \ldots, i_n$ sei eine mögliche Permutation der Indizes $1, \ldots, n$) beträgt

$$
\frac{1}{n}(t_{i_1} + (t_{i_1} + t_{i_2}) + \cdots + (t_{i_1} + t_{i_2} + \cdots + t_{i_n})) = \frac{1}{n} \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j} = \frac{1}{n} \sum_{k=1}^{n} (n - k + 1)t_{i_k} \quad (5.1)
$$

Dieser Term soll minimal sein. Dies ist offensichtlich der Fall, wenn gilt

$$
nt_{i_1} + (n - 1)t_{i_2} + \cdots + 2t_{i_{n-1}} + t_{i_n} = \min
$$

Gleichung (5.2) ist offensichtlich minimal, wenn

$$
t_{i_1} < t_{i_2} < \cdots < t_{i_{n-1}} < t_{i_n}
$$

gewählt wird. \(\square\)
5.2.3 Preemptive Scheduling-Algorithmen


**ABER:**
Diese Algorithmen können zu Inkonsistenzen führen, z.B. “Geld abheben”:
Eine Banktransaktionsfolge in 3 Schritten:

1. lies Kontostand
2. k′ := k – AbzuhebendesGeld
3. Schreibe Kontostand k′


Abbildung 5.4: Programmbefehle

**Problem:**
Unterbrechungen bei der Ausführung der Jobs sind möglich. Dann sind 900 Euro auf dem Konto, obwohl (s.oben) 500 Euro abgehoben wurden.
⇒ Kapitel 7 (Verwendung preemptiver Scheduling-Algorithmen)

**Shortest Remaining Processing Time (SRPT)**

**NEU:**
Kommt während der Abarbeitung eines Prozesses ein neuer Prozess an (Zustand “ready”), so wird die Restbedienzeit des aktuellen Prozesses mit der Gesamtbetienzeit des neuen Prozesses verglichen. Ist also der neue Prozess kürzer, so erfolgt ein Kontextwechsel.

---

Ehefrau                       Scheduler                       Ehemann
1. lies k                      3. lies k
2. k′ = k - 100 Euro          4. k′ = k - 400 Euro
6. schreibe k′

Scheduler
5. schreibe k′
**Bemerkung:** Bei unseren Scheduling-Algorithmen wird nur die reine Bedienzeit der Nutzerprozesse (keine Prozessorzeit für Dispatcher o.ä.) betrachtet.

Die Wirkungsweise von SRPT kann an folgendem Beispiel nachvollzogen werden:

<table>
<thead>
<tr>
<th>Prozess</th>
<th>Ankunftszeit</th>
<th>Verweildauer $T_v$</th>
<th>Bedienzeit $T_b$</th>
<th>Quotient $\frac{T_v}{T_b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>1.33</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>1.8</td>
</tr>
<tr>
<td>P₅</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

|              | $∅$        | 4                   | $∅$              | $≈ 1.53$                   |

Abb. 5.5: Beispiel mittlere Verweildauer für SRPT


**Bemerkung:**
In dem vorangegangenen Beispiel hätte ein Scheduler zum Zeitpunkt $t = 2$ anstelle des Prozesses P₁ auch den Prozess P₃ dem Prozessor zuweisen können. Dies hätte keinen Einfluss auf die mittlere Verweilzeit, jedoch würde sich die mittlere normalisierte Verweilzeit auf $≈ 1,43$ verbessern. Die Entscheidung, in welcher Reihenfolge Prozesse mit gleicher Restbedienzeit ausgeführt werden, wird nicht von SRPT vorgegeben.

Wir wollen nun nach einem praktisch realisierbaren Algorithmus suchen, der möglichst fair ist und eine möglichst geringe mittlere Verweildauer aufweist.

**Round Robin**

*Round Robin (RR)* nutzt Uhren-basierte Unterbrechungen, d.h. in periodischen Intervallen werden Prozesse in ihrer Abarbeitung unterbrochen, bis sie wieder ein Zeitquantum Q für

Abbildung 5.6: Realisierung des Round Robin Verfahrens

Die Wirkungsweise von RR kann an folgendem Beispiel mit Q = 1 nachvollzogen werden (Abbildung 5.8 zeigt die Belegung der Ready-Queue zu den unterschiedlichen Ausführungszeitpunkten):

<table>
<thead>
<tr>
<th>Prozess</th>
<th>Ankunftszeit</th>
<th>Verweildauer $T_v$</th>
<th>Bedienzeit $T_b$</th>
<th>Quotient $\frac{T_v}{T_b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>1.33</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
<td>16</td>
<td>6</td>
<td>2.67</td>
</tr>
<tr>
<td>P₃</td>
<td>4</td>
<td>13</td>
<td>4</td>
<td>3.25</td>
</tr>
<tr>
<td>P₄</td>
<td>6</td>
<td>14</td>
<td>5</td>
<td>2.80</td>
</tr>
<tr>
<td>P₅</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10.8</td>
<td>4</td>
<td>2.71</td>
</tr>
</tbody>
</table>

Abbildung 5.7: Beispiel mittlere Verweildauer für RR
Abbildung 5.8: Belegung der Ready-Queue zu dem Beispiel aus Abbildung 5.7.
Mittlere Verweildauer: 10.8
Mittlere normalisierte Verweildauer $\frac{T_i}{t_i} : 2.71$

Für das schon früher verwendete Beispiel ergibt sich folgendes Bild:

<table>
<thead>
<tr>
<th>Prozess</th>
<th>Ankunftszeit</th>
<th>Verweildauer $T_v$</th>
<th>Bedienzeit $T_b$</th>
<th>Quotient $\frac{T_v}{T_b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>1.8</td>
</tr>
<tr>
<td>$P_5$</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

$\emptyset 5$  $\emptyset 2.4$  $\emptyset 2.16$

Abbildung 5.9: Beispiel mittlere Verweildauer für RR

Bemerkung:
In den vorangegangenen Beispielen wurde angenommen, dass wenn zu einem bestimmten Zeitpunkt $t$ ein neuer Prozess $P_n$ in die Ready-Queue geschrieben werden soll, der gerade aktive Prozess $P_a$ immer nach $P_n$ der Ready-Queue hinzugefügt wird! Diese Vorgehensweise wird jedoch nicht vom jeweiligen Scheduling-Algorithmus vorgeschrieben und muss vorher bekannt sein. Genauso gut hätte man sich auch für die Strategie entscheiden können, stets erst den gerade aktiven Prozess einzureihen. Eine weitere Möglichkeit wäre die Prozesse entsprechend ihrer lexikografischen Ordnung einzureihen.

Vergleich:
RR ist bezüglich der mittleren Verweildauer besser als FCFS, aber schlechter als SJF und als SRPT. Im Gegensatz zu SJF und SRPT ist RR jedoch

- fair in der Hinsicht, dass kein Verhungern vorkommt und
- praktisch implementierbar

Im Verhältnis zu FCFS profitiert RR davon, dass kürzere Jobs schneller abgearbeitet werden. Bei RR hängt die Dauer der Abarbeitung ab von

- der Länge der Jobs (→ linearer Zusammenhang: je kürzer ein Job, desto schneller ist er fertig) und
Kapitel 5. Scheduling

– der Anzahl vorhandener Jobs.

Man kann nun zwei alternative Fälle betrachten:

1. Das Dispatching wird bei der Zeitberechnung vernachlässigt.
2. Das Dispatching wird bei der Zeitberechnung nicht vernachlässigt.

In Fall 1. hat die Länge des Zeitquantums $Q$ keinen Einfluß auf die mittlere Verweildauer. In Fall 2. hingegen gilt, dass je kleiner $Q$ gewählt wird, desto mehr Overhead entsteht durch den Kontextwechsel, was wiederum zu einer Erhöhung der mittleren Verweildauer führt.

$\Rightarrow Q$ ist möglichst optimal zu wählen, so dass zum einen keine Konvergenz gegen FCFS auftritt und zum anderen das Dispatching im Verhältnis zur Prozessabarbeitung nicht zu aufwendig wird. In der Praxis werden z.B. Zeitquanten in der Größenordnung zwischen 10 und 100 ms verwendet. Prozesse, die weniger als 1 Zeitquantum brauchen, geben die CPU von sich aus frei, andernfalls initiiert der Scheduler eine Unterbrechung.


5.2.4 Priority Scheduling (PS)

Jedem Auftrag/Prozess wird bei dieser Strategie eine Priorität zugeordnet, die Abarbeitung erfolgt nach dem Prinzip Highest-Priority-First (Scheduler sucht aus). Bei einem Clustering in Prioritätsklassen würde der älteste aller Aufträge ausgewählt werden, der sich in der höchsten Prioritätsklasse befindet. Dabei kann das Problem des Verhungerns niederpriorer Aufträge auftreten. $\Rightarrow$ keine Fairness

Vorteile:
– Wichtige Prozesse können bevorzugt werden:
  Bislang haben wir bei “guten” Scheduling-Algorithmen ein Bevorzugen kurzer Prozesse beobachtet $\Rightarrow$ wichtige = kurze Prozesse sollen schnell abgearbeitet werden, d.h. eine hohe Priorität bekommen.
– praktisch implementierbar

Nachteile:
– Verhungern von Prozessen mit niedriger Priorität
– relativ schlechte mittlere Verweildauer

5.2.5 Multilevel Feedback Queueing

Arbeitet sowohl mit Prioritätsklassen (”Multilevel”) als auch mit Zeitscheiben. Das besondere dieses Algorithmus ist - im Gegensatz zu SPN und SRPT - dass man keine Kenntnisse über die voraussichtliche Abarbeitungszeit braucht und trotzdem kurze Aufträge bevorzugen kann. Das Scheduling wird nun unterbrechend ausgeführt; Prioritäten werden dabei dynamisch vergeben.
Eine Realisierung des Verfahrens kann durch eine Unterteilung der \( n \) Ready-Queues in \( n - 1 \) FIFO-Queues und einer Round-Robin-Queue erfolgen. Beim Beginn der Ausführung wird einem Prozess die Priorität \( Q_0 \) gegeben, d.h. ein neuer Prozess wird am Ende der obersten FIFO-Queue eingefügt. Der Scheduler teilt immer dem Prozess am Anfang der obersten nicht leeren Warteschlange den Prozessor zu. Erreicht der Prozess den Anfang der Queue, wird er dem Prozessor zugewiesen. Nun sind drei Fälle zu unterscheiden:

- Der Prozess wird vollständig abgearbeitet
  \( \Rightarrow \) Der Prozess verlässt das System.

- Der Prozess gibt den Prozessor freiwillig ab
  \( \Rightarrow \) Der Prozess verlässt die gegenwärtige Queue. Sobald der Prozess wieder in den Zustand “ready” gelangt, wird er wieder in \textbf{dieselbe} Queue eingereiht.

- Der Prozess nimmt die gesamte ihm zugewiesene Zeitscheibe in Anspruch
  \( \Rightarrow \) Der Prozess wird unterbrochen und an das Ende der \textbf{nächst niedrigeren} Queue gesetzt.


Implementierung und Abarbeitung eines Auftrags:

![Abbildung 5.10: Implementierung und Abarbeitung eines Auftrags](image)

Bei längeren Abarbeitungszeiten der Prozesse sollten entsprechend auch längere Zeitscheiben verwendet werden.
5.3 Prozesswechsel


5.4 Arten des Schedulings

Beim Scheduling werden verschiedene Arten des Scheduling unterschieden:

- Short Term Scheduling
- Medium Term Scheduling
- Long Term Scheduling

Vgl. auch Abbildungen 5.11 und 5.12.

Abbildung 5.11: Zuordnung der Prozesszustände zu den Arten des Scheduling

Genauer: Falls sich keiner der Prozesse im Hauptspeicher im Zustand “ready” befindet, so lagert das Betriebssystem einen der blockierten Prozesse aus und zwar in eine Suspended-Queue auf Platte.
Abbildung 5.12: Visualisierung der hierarchischen Unterteilung der Scheduling-Arten
Dadurch kann ein anderer Prozess dieser Suspended-Queue (der “ready” ist) oder ein ganz neuer Prozess in den Hauptspeicher geladen werden, der dann direkt vom Prozessor ausgeführt werden kann (höhere Priorität: suspendierter Prozess im Zustand ready).
Teil III

Multiprocessing
6

Deadlocks bei Prozessen

► Modellierung von nebenläufigen Prozessen
► Deadlocks und Partial Deadlocks
► Petri-Netze

Inhaltsangabe

6.1 Motivation der Deadlocks anhand zweier Beispiele . . . . . . . . . . . . . 96
6.2 Das Prinzip der Deadlocks . . . . . . . . . . . . . . . . . . . . . . . . . . 97
6.3 Deadlock Prevention . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
  6.3.1 Deadlock Avoidance . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
  6.3.2 Petri-Netze zur Prozeßmodellierung . . . . . . . . . . . . . . . . . . 106
  6.3.3 Markierungen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
  6.3.4 Modellierung von nebenläufigen Prozessen . . . . . . . . . . . . . . 113
  6.3.5 Deadlock Detection (Deadlockerkennung) . . . . . . . . . . . . . . . 118
In diesem Kapitel soll das Problem des Deadlocks bei kooperierenden Prozessen (vgl. auch Kapitel 7.4.5) diskutiert werden. Um Deadlocks beschreiben und erkennen zu können, wird es nötig sein, Prozesse und ihre Interaktion zu modellieren.

### 6.1 Motivation der Deadlocks anhand zweier Beispiele


![Abbildung 6.1: Illustration eines Deadlock](image)

**Philosophenproblem** Abbildung 6.2 veranschaulicht das 2 Philosophenproblem: 2 Philosophen können wahlweise denken oder essen. Damit wenigstens ein Philosoph essen kann (Reis), werden auf dem Tisch 2 Stäbchen zur Verfügung gestellt. Problem: Wollen beide Philosophen essen und nehmen dazu simultan jeweils das (von ihnen aus) rechte Stäbchen, so bekommt jeder nur eins und keiner kann den Reis essen. 

⇒ Deadlocksituation

**Definition 6.1. Deadlock** Ein System befindet sich im Zustand einer Verklemmung (Deadlock), wenn mindestens zwei Prozesse sich in einem wechselseitigen Wartezustand befinden und auch durch die Freigabe aller Betriebsmittel möglicher anderer Prozesse dieser Zustand nicht aufgehoben werden kann.
Das Prinzip der Deadlocks

**Aufgabe 1** Beispiel für einen Deadlock
Ein Computer habe sechs Bandlaufwerke und \( n \) Prozesse, von denen jeder zwei Bandlaufwerke gleichzeitig für seine Ausführung benötige. Für welche Werte von \( n \) ist dieses System frei von Verklemmungen (vgl. Abbildung 6.3)?

**Lösung zu Aufgabe 1:**
- \( n = 1 \): \( \Rightarrow \) kein Deadlock
- \( n = 2 \):
  - Fall 1: Prozesse benötigen ganz bestimmtes Laufwerk \( \Rightarrow \) Deadlocks sind möglich
  - Fall 2: Welche Laufwerke ein Prozess zugeteilt bekommt, ist egal, Hauptsache es sind zwei Stück \( \Rightarrow \) kein Deadlock
- \( n = 3 \): \( \Rightarrow \) kein Deadlock (wie Fall \( n = 2 \))
- \( n = 4 \): \( \Rightarrow \) kein Deadlock
  - \( P_1, P_3 \) haben 2 Laufwerke. Diese Prozesse werden abgearbeitet und geben dann ihre BM frei. Dann wird entweder \( P_2 \) oder \( P_4 \) ausgeführt, der wiederum nach der Abarbeitung seine BM freigibt, worauf der letzte Prozess abgearbeitet werden kann.
- \( n = 5 \): \( \Rightarrow \) kein Deadlock (ähnlich wie \( n = 4 \))
  - Bei 6 Laufwerken und 5 Prozessen gibt es mindestens 1 Prozess, der 2 Laufwerke zugeordnet bekommt und abgearbeitet werden kann.
- \( n = 6 \): \( \Rightarrow \) Deadlock kann auftreten! Wenn jeder Prozess ein Laufwerk zugeordnet bekommt und auf ein anderes wartet.
- \( n \geq 7 \): \( \Rightarrow \) Deadlocks können auftreten

### 6.2 Das Prinzip der Deadlocks

**Def.:** Ein Deadlock ist die dauerhafte Blockierung einer Menge \( M \) von Prozessen, die gemeinsame Systemressourcen \( S \) nutzen oder miteinander kommunizieren. \(|M|, |S| \geq 2\)

Leider gibt es für dieses Problem keine allgemeingültige Lösung.
Kapitel 6. Deadlocks bei Prozessen


Aufgabe 2 Anwendungsbeispiel für das Auftreten von Deadlocks
In einem elektronischen Banksystem gibt es hunderte von identischen Prozessen, die alle mittels der Information über die gutzuschreibende Geldmenge, das Zielkonto und das zu belastende Konto Transaktionen vornehmen. Diese Transaktionen werden vorgenommen, indem beide Konten gesperrt werden und erst wieder freigegeben werden, wenn der Transfer abgeschlossen ist.

Bei vielen parallel ablaufenden Prozessen ist die Wahrscheinlichkeit sehr hoch, daß ein Prozeß $A$ ein Konto $x$ sperrt und auf Konto $y$ wartet, während ein Prozeß $B$ das Konto $y$ sperrt und nun gerade auf $x$ wartet. Finden Sie eine Möglichkeit, Verklemmungen auszuschließen, wobei ein Konto erst dann freigegeben werden kann, wenn die Transaktion abgeschlossen ist, d.h. Lösungen, die ein Konto sperren und es sofort wieder freigeben, wenn das andere gesperrt ist, sind nicht gültig.

Zunächst wollen wir ein Beispiel betrachten, wie Deadlocks zustande kommen. Wir betrachten dazu zwei Prozesse $P$ und $Q$, die auf je 2 Betriebsmittel $A$ und $B$ zugreifen:

\[
\begin{array}{ll}
\text{Process P} & \text{Process Q} \\
\ldots & \ldots \\
\text{Get A} & \text{Get B} \\
\ldots & \ldots \\
\text{Get B} & \text{Get A} \\
\ldots & \ldots \\
\text{Release A} & \text{Release B} \\
\ldots & \ldots \\
\text{Release B} & \text{Release A} \\
\end{array}
\]

Auch durch geeignetes Scheduling findet man hier keine Möglichkeit, beide Prozesse vollständig abzuarbeiten. Dieser Sachverhalt soll nun in einem sogenannten Prozeßfortschrittdiagramm dargestellt werden (vgl. Abbildung 6.4):

Dabei verstehen wir unter Fortschritt die Reihenfolge oder Sequenz, in der einzelne Befehle abgearbeitet werden sowie insbesondere der Zugriff und die Freigabe von BM.

Zunächst betrachten wir nicht-preemptives Scheduling am Beispiel des FCFS (1. und 2.):

1. $Q$ wird zunächst vollständig abgearbeitet, dann $P$.
2. $P$ wird zunächst vollständig abgearbeitet, dann $Q$.

Nun das preemptive Scheduling:
Betrachten wir folgende Betriebsmittelzuordnung: \(\rightarrow a)\) $Q$ bekommt $B$, \(\rightarrow b)\) $P$ bekommt $A$
Es ergibt sich folgendes Szenario:
* $P$ fordert Betriebsmittel $B$ an, das jedoch von $Q$ gehalten wird und nicht frei ist. $P$ geht vom
Zustand *ready* in den Zustand *blocked* über, d.h. P wird unterbrochen, der Scheduler wählt den nächsten Prozess aus, der sich im Zustand *ready* befindet, nämlich Q. Nun braucht Q das Betriebsmittel A, A wird aber von P belegt ⇒ *Deadlock* an der Stelle . Der Deadlock ist bereits unvermeidbar, wenn der Graph in den unsicheren Bereich kommt, da dieser unweigerlich in den unmöglichen Bereich führt. Die Lösung des Problems ist relativ einfach, wenn die BM-Anforderungen vertauscht werden können, bzw. die Implementierung der Prozesse P und Q so zu gestalten ist, daß jeder der Prozesse die Betriebsmittel A und B nicht gleichzeitig benötigt:

<table>
<thead>
<tr>
<th>Process P'</th>
<th>Process Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Get A</td>
<td>Get B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Release A</td>
<td>Get A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Get B</td>
<td>Release B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Release B</td>
<td>Release A</td>
</tr>
</tbody>
</table>

Dadurch würde das Prozessfortschrittsdiagramm aus Abbildung 6.5 entstehen. Bei diesem Ablauf gibt es keine Kombination, bei der sowohl P als auch Q die Betriebsmittel A und B benötigen und demzufolge auch keinen Zustand, in welchem ein Deadlock unvermeidbar wird (vgl. Abbildung 6.6).

Eine weitere Alternative stellt folgende Lösung dar:
Kapitel 6. Deadlocks bei Prozessen

Abbildung 6.5: Prozeßfortschrittsdiagramm 2

Abbildung 6.6: Bsp. Zustand ohne Deadlock

<table>
<thead>
<tr>
<th>Process P''</th>
<th>Process Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Get B</td>
<td>Get B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Get A</td>
<td>Get A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Release A</td>
<td>Release B</td>
</tr>
<tr>
<td>...</td>
<td>Release A</td>
</tr>
<tr>
<td>Release B</td>
<td>...</td>
</tr>
</tbody>
</table>

Bei den Ressourcen unterscheiden wir 2 Arten:

- **Wiederverwendbare Ressourcen**
  sind Ressourcen, die von genau einem Prozeß genutzt werden können und durch diese
Ein Deadlock könnte hier (zusätzlich zu oben genanntem Beispiel) z.B. durch einen ungünstigen Speicherzugriff entstehen.
Wir nehmen als Beispiel 200 KiloByte Speicher an und folgenden Bedarf von 2 Prozessen:

<table>
<thead>
<tr>
<th>Process P₁</th>
<th>Process P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 KByte</td>
<td>70 KByte</td>
</tr>
<tr>
<td>60 KByte</td>
<td>90 KByte</td>
</tr>
</tbody>
</table>

Gelangen beide Prozesse zwischen die erste und zweite Speicheranforderung, so entsteht ein Deadlock, da nur noch 50 kByte verfügbar sind und 60 bzw. 90 kByte gefordert werden.
Lösung: Virtueller Speicher (→späteres Kapitel)

Verbrauchende Ressourcen
sind Ressourcen, die erzeugt und zerstört werden. Dabei kann ein produzierender Prozeß eine beliebige Anzahl solcher Ressourcen erzeugen. Z.B.: Unterbrechungen, Signale, Nachrichten, Informationen in E/A-Speichern. ⇒ kein Deadlock möglich

6.3 Deadlock Prevention

→Verhinderung/Verhütung von Deadlocks

Das Vorgehen zur Deadlockverhütung besteht darin, ein System zu konstruieren, in dem Deadlocks überhaupt nicht vorkommen können. Generell werden die Vorsichtsmaßnahmen, d.h., die Methoden zur Deadlockverhinderung (Deadlock Prevention) in zwei Klassen eingeteilt:

1. Indirekte Methoden

2. Direkte Methoden

Diese beiden Klassen wollen wir nacheinander näher betrachten.

→ Indirekte Methoden gehen davon aus, dass generell drei Bedingungen erfüllt sein müssen, damit überhaupt ein Deadlock eintreten kann. Diese 3 Bedingungen wollen wir im einzelnen betrachten:
1. **Bedingung:** Mutual Exclusion (wechselseitiger Ausschluss)
   Es gibt mindestens 2 Ressourcen, die nur von je einem Prozeß gleichzeitig genutzt werden können. Bei nur einem BM kann kein Deadlock entstehen, da ein wechselseitiger Zugriff nicht stattfinden kann.
   Deadlockvoraussetzung: mindestens 2 Prozesse und 2 BM.

2. **Bedingung:** Hold and Wait
   Ein Prozeß muß eine Ressource behalten, während er auf eine weitere Ressource wartet.

3. **Bedingung:** No Preemption (Keine Unterbrechung)
   Eine Ressource kann einem Prozeß, der sie behält, nicht wieder entzogen werden.

Ist mindestens eine dieser Bedingungen verletzt, so kann gar kein Deadlock auftreten.

– **Direkte Methoden** betrachten die genaue Situation, die zu einem Deadlock führen würde. Dieser Umstand wird auch als **Circular Wait** bezeichnet.

4. **Umstand:** Circular Wait
   Es existiert eine geschlossene Kette von Prozessen, so daß jeder Prozeß mindestens eine Ressource hält, die von einem anderen Prozeß der Kette gebraucht wird (vgl. Abbildung 6.7).

![Abbildung 6.7: Circular Wait](image)

### 6.3.1 Deadlock Avoidance

→Vermeidung von Deadlocks

Bei der Deadlockvermeidung lässt man die 3 Bedingungen außer Acht. Insbesondere werden Unterbrechungen und Nebenläufigkeit zugelassen.
Dafür wird jedoch dynamisch entschieden, ob Ressourcen genutzt werden können, oder ob dies zu einer potentiellen Deadlocksituation führt.
⇒Demzufolge erfordert die Deadlockvermeidung Wissen über die zukünftigen Ressourcenanforderungen der Prozesse.

Genau genommen gibt es zwei Verfahren für die Deadlockvermeidung:

1. Ein Prozeß darf nicht gestartet werden, falls seine Anforderungen zu einem Deadlock führen könnten und
2. Eine Ressourcenanfrage darf nicht befürwortet werden, falls sie zu einem Deadlock führen könnte.

Wir wollen nun eine Methode betrachten, die davon ausgeht, einen Prozess gar nicht erst zu starten, falls ein potentieller Deadlock droht:

I. Process Initiation Denial (Ablehnen einer Prozessinitierung)
Zunächst brauchen wir einige Vorbemerkungen:
Es werde ein System mit n Prozessen und m verschiedenen Arten von Ressourcen betrachtet. Durch Einführung von Vektoren und Matrizen werde Folgendes beschrieben:

Ressourcen = \((R_1, R_2, \ldots, R_m)\) bezeichne die Anzahl jedes Ressourcentyps im gesamten System (auch als Max bezeichnet).
Verfügbarkeit = \((V_1, V_2, \ldots, V_m)\) bezeichne die Anzahl verfügbarer Ressourcen, d.h. die Anzahl der Ressoucen pro Typ, die keinem Prozess zugeordnet sind.
Es wird nun davon ausgegangen, dass Prozess \(P_1\) bestimmte Ressoucen jedes Typs belegt, und zwar jeweils von der Ressource \(R_i\) \(c_{i1}\) Stück. Damit ergibt sich der Vektor:
\(\text{Claim}_{P_1} = (C_{11}, C_{12}, \ldots, C_{1m})\)

Durch Betrachten jedes Prozesses \(P_i\), \(i = 1, \ldots, n\) ergibt sich die Matrix der Anforderungen:

\[
\text{Claim} = \begin{pmatrix}
C_{11} & C_{12} & \cdots & C_{1m} \\
C_{21} & C_{22} & \cdots & C_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & \cdots & C_{nm}
\end{pmatrix}
\]

Die Informationen dieser Matrix müssen im Voraus durch einen Prozess bereitgestellt werden. Zu einem bestimmten Zeitpunkt belegt jeder Prozess \(i\) genau \(A_{ij}\) Ressourcen vom Typ \(R_j\), was dargestellt wird als

\[
\text{Allocation} = \begin{pmatrix}
A_{11} & \cdots & A_{1m} \\
\vdots & \ddots & \vdots \\
A_{n1} & \cdots & A_{nm}
\end{pmatrix}
\]

Damit ergeben sich folgende Zusammenhänge:

1. \(R_i = V_i + \sum_{k=1}^{n} A_{ki} \quad \forall \) Ressourcen \(i\),
d.h. jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.

2. \(C_{ki} \leq R_i \quad \forall \) Prozesse \(k\) und Ressourcen \(i\),
d.h. kein Prozess kann von irgendeiner Ressource mehr belegen als im gesamten System vorhanden ist.

3. \(A_{ki} \leq C_{ki} \quad \forall \) Prozesse \(k\) und Ressourcen \(i\),
d.h. kein Prozess kann von irgendeiner Ressource mehr belegen als er ursprünglich durch die Claim-Werte \(C_{ki}\) angefordert hat.
Mit diesen Größen kann eine Vorschrift zur Deadlockvermeidung (Deadlock Avoidance Policy) wie folgt definiert werden:
Starte keinen neuen Prozess, falls dessen Ressourcenanforderungen zu einem Deadlock führen könnten. D.h. starte einen neuen Prozess $P_{n+1}$ nur, falls $R_i \geq C_{(n+1)i} + \sum_{k=1}^{n} C_{ki} \forall$ Ressourcen i gilt.
Begründung: In diesem Fall wäre es möglich, dass alle gerade laufenden Prozesse quasi gleichzeitig maximale Anforderungen an die Ressource i stellen würden (Worst Case Abschätzung). Diese Strategie funktioniert zwar, ist aber weit von einem optimalen Systemverhalten entfernt, das die gleichzeitige Verwaltung möglichst vieler Prozesse ermöglicht.

Wir wollen nun einen alternativen Algorithmus betrachten.

**II. Ressource Allocation Denial** (Ablehnen einer Ressourcenbelegung oder Bankers Algorithm)

→Zustand des Systems:

- aktuelle Zuordnung von Ressourcen zu einem Prozess
- Zustand wird durch Vektoren und Matrizen beschrieben (Ressourcen, Verfügbarkeit, Claim, Allocation)

Ein sicherer Zustand (safe state) wird wie folgt definiert: $\exists$ mindestens eine Folge von Prozessabläufen, die nicht zu einem Deadlock führt, sondern bei der alle Prozesse erfolgreich bis zum Ende abgearbeitet werden können.
Ein unsicherer Zustand ist ein Zustand, der nicht sicher ist.

**Beispiel 6.1.** Es soll ein sicherer Zustand dargestellt werden (vgl. Abbildung 6.8). Dieser Zustand ist sicher, da die Abarbeitungsfolge $P_2, P_1, P_3$ ohne Deadlock möglich ist. $P_2$ wird nach Zuweisung einer Ressource vom Typ $R_3$ abgearbeitet. →run-to-completion und Programmzustand war sicher.
Deadlock Avoidance Strategie: Das System soll stets in einem sicheren Zustand gehalten werden, d.h., bei einer Ressourcenanfrage wird kontrolliert, ob nach einer evtl. Zuteilung noch ein sicherer Zustand vorliegt. Ist dies nicht der Fall, wird die Anfrage abgelehnt.
Beispiel 6.2. *Es soll ein unsicherer Zustand dargestellt werden (vgl. Abbildung [6.9]).* 
R₁ ist nicht mehr vorhanden, wird aber von jedem Prozess zur vollständigen Abarbeitung benötigt. Ein Deadlock ist an dieser Stelle nicht mehr ausgeschlossen: die Anforderung von R₁ und R₃ durch P₁ würde zu einem unsicheren Zustand führen und muss deshalb abgelehnt werden. Ob tatsächlich ein Deadlock auftritt, hängt jedoch von der konkreten Implementierung und den
spezifischen Anforderungen (in welcher Reihenfolge wird welche Ressource wieder freigegeben) der anderen Prozesse ab.

Abbildung 6.9: Beispiel eines unsicheren Zustandes

6.3.2 Petri-Netze zur Prozeßmodellierung


**Definition 6.2 (Netz).** Sei \( X = (S \cup T, F) \) ein endlicher Graph mit \( S \cap T = \emptyset \) (\( S \) und \( T \) disjunkt). Dann heißt \( X \) Netz genau dann, wenn

1. \( S \) eine endliche Menge \( S = \{s_1, \ldots, s_m\} \) von Stellen,
2. \( T \) eine endliche Menge \( T = \{t_1, \ldots, t_n\} \) von Transitionen und
3. \( F \subset (S \times T) \cup (T \times S) \) eine Menge von Kanten ist (\( X \) ist also ein gerichteter Graph).

**Bemerkung 6.1.** Damit dürfen zwischen je zwei Stellen und zwei Transitionen keine Kanten auftreten. Eine Kante besteht also zwischen genau einer Stelle \( s \) und einer Transition \( t \), entweder von \( s \) nach \( t \) oder \( t \) nach \( s \).
Graphisch werden Stellen als Kreise, Transitionen als (schmale) Rechtecke und Kanten als Pfeile dargestellt, wie in Abbildung 6.10.

Beispiel für ein Petri-Netz mit $S = \{S_1, S_2, S_3\}$, $T = \{T_1, T_2, T_3\}$ und $F = \{(S_1, T_1), (S_1, T_2), (S_2, T_1), (S_2, T_2), (T_1, S_3), (T_2, S_3), (S_3, T_1), (T_3, S_1), (T_3, S_2)\}$. Durch die Angabe von $S$, $T$ und $F$ ist das Netz vollständig beschrieben.

Abbildung 6.10: Beispiel für ein Petri-Netz

Ein sehr gebräuchliches Konzept eines Petri-Netzes ist das sogenannte Stellen/Transitions-System:

**Definition 6.3** (Stellen/Transitions-System). Ein Stellen/Transitions-System ist ein 6-Tupel $Y = (S, T, F, K, W, M_0)$ bestehend aus einem Netz $(S \cup T, F)$ sowie den Abbildungen

1. $K : S \rightarrow \mathbb{N} \cup \{\infty\}$, durch die jeder Stelle eine Kapazität zugeordnet wird,

2. $W : F \rightarrow \mathbb{N}$, durch die jeder Kante ein Kantengewicht zugeordnet wird und

3. $M : S \rightarrow \mathbb{N}$ (Markierung), durch die jeder Stelle eine Anzahl von Marken (Token) zugeordnet wird.

Dabei kann eine Stelle nie mehr Marken aufnehmen als ihre Kapazität erlaubt, d.h. für alle $s \in S$ gilt $M(s) \leq K(s)$.

$M_0$ heißt Anfangsmarkierung und stellt die Markierung dar, wie sie zu Beginn des durch das Petri-Netz modellierten Prozesses vorliegt.

**Bemerkung 6.2.** Im einfachsten Fall beträgt das Kantengewicht 1 und wird nicht weiter angegeben. Analog nimmt man als Kapazität der Stellen $\infty$ an, wenn keine Angabe erfolgt.

**Bemerkung 6.3.** Marken werden graphisch durch schwarze Punkte innerhalb der kreisförmigen Stellen symbolisiert (vgl. Abbildung 6.11).
Kapitel 6. Deadlocks bei Prozessen

1. Bsp

Ausgangssituation

Nach dem Feuern der Transition

Falls kein Kantengewicht angegeben ist, so ist \( W = 1 \)

2. Bsp zur Kapazität

nun ist kein weiteres Feuern möglich, da die Kapazität der rechten Stelle nur 3 Token zulässt.

3. Bsp

vorher

nachher

Abbildung 6.11: Stellen-Transitionssysteme
Beziehungen zu Prozessen

Die **Stellen eines Petri-Netzes** repräsentieren z.B. Signale, Daten, Speicher eines Objekts, d.h. statische Objekte. Bei Speichern kann die Anzahl der Marken beispielsweise die Anzahl der im Speicher vorhandenen Objekte repräsentieren.

Die **Transitionen eines Petri-Netzes** repräsentieren z.B. Ereignisse, Vorgänge, Berechnungsschritte oder Jobs, d.h. Aktionen.

Die modellierten Systeme sollen dynamisch sein, es stellt sich also die Frage, wie Ereignisse ausgeführt werden. Dies wird durch Schalten der zugehörigen Transition erreicht, d.h. es werden zur Beschreibung der Dynamik eines Systems zusätzlich sogenannte Schaltregeln benötigt.

**Definition 6.4.** Sei \( x \in S \cup T \). Dann definiert man

1. \( \cdot x := \{ y \in S \cup T : (y, x) \in F \} \) heißt der Vorbereich einer Stelle \( x \in S \) bzw. einer Transition \( t \in T \).
2. \( x : = \{ y \in S \cup T : (x, y) \in F \} \) heißt der Nachbereich einer Stelle \( x \in S \) bzw. einer Transition \( t \in T \).
3. \( -x := \{ (y, x) \in (S \times T) \cup (T \times S) : (y, x) \in F \} \) heißt die Menge aller **Eingangskanten** von \( x \) und
4. \( x - := \{ (x, y) \in (S \times T) \cup (T \times S) : (x, y) \in F \} \) heißt die Menge aller **Ausgangskanten** von \( x \).

**Bemerkung 6.4.** Der Vorbereich einer Stelle besteht nur aus Transitionen, der Vorbereich einer Transition nur aus Stellen.

**Beispiel 6.3.** Wir wollen uns anhand des **Fünf-Philosophen-Problems** ein Petri-Netz modellieren (vgl. Abbildung 6.12).

Kapitel 6. Deadlocks bei Prozessen

Hier: Stäbchen (=BM) kann frei verfügbar sein (d.h. liegt auf dem Tisch) → Dafür wird eine Stelle benötigt (Denken).
Oder: Stäbchen wird von einem Philosophen genutzt (Essen)

Abbildung 6.13: Petri-Netz zum Philosophenproblem mit 5 Philosophen

6.3.3 Markierungen

Definition 6.5. Sei $M$ ein Markierung. Dann kann $M$ eine Menge $T_{akt}(M)$ zugeordnet werden mit

$$T_{akt}(M) = \{ t \in T : (\forall s \in t : M(s) \geq W(s, t)) \land (\forall s \in t : M(s) + W(t, s) \leq K(s)) \}.$$


Bemerkung 6.5. $(\forall s \in t : M(s) + W(t, s) \leq K(s))$ : durch neu hinzukommende Marken wird die Kapazität der Stelle $n$ im Nachbereich der Transition nicht überschritten.


Bemerkung 6.7. Mehrere Transitionen feuern sequentiell nacheinander (Quasiparallelität)

Bemerkung 6.8. Eine Transition $t \in T$ ist also genau dann aktiviert, wenn mindestens soviel “Input” zur Verfügung steht wie das Gewicht (die “Bandbreite”) der eingehenden Kanten und genügend Platz für den “Output” bereit steht.
Beispiel 6.4 (Markierte Petri-Netze). Es soll das Petri-Netz in Abbildung 6.15 betrachtet werden, wobei alle Kanten mit 1 gewichtet seien und die Kapazität der Stellen $\infty$ betrage. Als Markierung $M$ gegeben: $\{(S_1, 1), (S_2, 2), (S_3, 0)\}$, d.h. $S_1$ besitzt eine, $S_2$ zwei und $S_3$ keine Marke.

Gemäß Definition sind nun die Transitionen $T_1$ und $T_2$ aktiviert, $T_3$ ist nicht aktiviert, da z.B. für $T_1$ gilt: Für alle Stellen im Vorbereich von $T_1$, d.h. für $S_1$ und $S_2$, gilt $M(s) \geq W(s, t) = 1$, da $M(S_1) = 1$ und $M(S_2) = 2$. Außerdem ist für alle Stellen $s \in T_1$, also für $S_3$, $M(s) + W(T_1, s) \leq K(s)$, da die Kapazität aller Stellen $\infty$ ist.

Feuert nun die Transition $T_1$, so ergibt sich die in der 6.15 gezeigte Folgemarkierung. Analog hätte auch $T_2$ feuern können (die Eingangs- und Ausgangskanten von $T_2$ und $T_1$ sind gleich) und zu der gleichen Folgemarkierung geführt.

Nun ist $T_3$ aktiv (kann feuern), nicht aber $T_1$ und $T_2$. Feuert $T_3$, so ist die Belegung der Marken wie vor dem Feuern von $T_1$.

Durch Schalten oder Feuern einer aktiven Transition ergibt sich aus der Markierung $M$ die unmittelbare Folgemarkierung $M'$ wie folgt:

$$M'(s) = \begin{cases} 
  M(s) - W(s, t) & \text{für } s \in (\cdot \setminus \cdot t) (1.) \\
  M(s) + W(t, s) & \text{für } s \in (\cdot \setminus \cdot t) (2.) \\
  M(s) - W(s, t) + W(t, s) & \text{für } s \in (\cdot \cap \cdot t) (3.) \\
  M(s) & \text{für } s \not\in (\cdot \cup \cdot t) (4.) 
\end{cases}$$

Dabei bedeuten:

1. alle Stellen des Vorbereichs der feuernenden Transition
2. alle Stellen des Nachbereichs einer feuernenden Transition
3. Stellen des Vor- und Nachbereichs der feuernenden Transition
4. alle Stellen, die keine Kante zur feuernenden Transition aufweisen
Abbildung 6.15: Petri-Netz mit Markierung und Folgemarkierung
Abbildung 6.16: Petri-Netz mit Markierung und Gewichtung

**Beispiel 6.5** (Gewichtete markierte Petri-Netze). *Sei wieder die Kapazität der Stellen $\infty$, jedoch die Kanten einzeln gewichtet. Dann ergibt sich ein Bild wie in Abbildung 6.16.*

### 6.3.4 Modellierung von nebenläufigen Prozessen

Petri-Netze können z.B. zur Modellierung von nebenläufigen Prozessen genutzt werden, z.B. im Falle des klassischen Erzeuger/Verbraucher-Problems. Es handelt sich bei diesem Problem um zwei Prozesse (E=Erzeuger, V=Verbraucher), von denen E Resultate produziert, die V auf dem Drucker ausgeben soll. Zur Synchronisation wird ein endlicher Speicher S der Kapazität $\text{MAX}$ eingeführt, der von E gefüllt und von V geleert wird. Es treten bei der Synchronisation 3 Probleme auf:

1. V darf nur auf produzierte Daten von S zugreifen
2. E darf nur bei freien Plätzen ablegen
3. Speicher S darf nicht gleichzeitig von E und V verändert werden


- $S_1$: Prozess E hat ein erzeugtes Element
- $S_2$: Prozess E greift auf den Speicher S zu
- $S_3$: Prozess E hat den Speicher S wieder freigegeben und noch kein Element erzeugt
- $S_4$: Es gibt $x$ freie Plätze
- $S_5$: Der exklusive Zugriff durch Prozesse E oder V auf Speicher S ist möglich
- $S_6$: Es gibt $x$ belegte Plätze
- $S_7$: Prozess V möchte ein Element verbrauchen
- $S_8$: Prozess V greift auf den Speicher S zu
- $S_9$: Prozess V verbraucht ein Element

**Bemerkung 6.9.** $S_5$ ist eine Synchronisationssemaphore (vgl. binäre Semaphore in Kapitel 7).
Kapitel 6. Deadlocks bei Prozessen

Abbildung 6.17: Petri-Netz zur Modellierung des Erzeuger/Verbraucher-Problems

Beispiel 6.6. Leser/Schreiber-Problem (Reader/Writer-Problem)
Das Leser/Schreiber-Problem wird auf verschiedene Arten realisiert:

1. \( R/W \)-Problem \( \rightarrow \) Leser muss (müssen) nur dann warten, wenn ein Schreiber aktiv ist (Schreiber kann verhungern!!)

2. \( R/W \)-Problem \( \rightarrow \) Wartet der Schreiber, so darf kein neuer Leser mit dem Lesen beginnen.

Abbildung 6.18: R/W-Problem
jede Marke $M$ entspricht einem Prozess

Abbildung 6.19: Einfaches Petri-Netz zum R/W-Problem
Abbildung 6.20: Fertiges Petri-Netz zum R/W-Problem
z. B. (mögliche Systemverhalten):

\[
\begin{align*}
M_0 &= (3, 0, 0, 0, 0, 3) \\
M_1 &= (2, 1, 0, 0, 0, 3) \\
M_2 &= (2, 0, 0, 1, 0, 3) \\
M_3 &= (1, 2, 0, 0, 0, 3) \\
M_4 &= (2, 0, 1, 0, 0, 2) \\
M_5 &= (1, 1, 0, 1, 0, 3) \\
M_6 & \\
M_7 &
\end{align*}
\]

Abbildung 6.21: Erreichbarkeitsgraph zum R/W-Problem

Abbildung 6.22: Vereinfachter Erreichbarkeitsgraph zum R/W-Problem
6.3.5 Deadlock Detection (Deadlockerkennung)

Diese Methode betrachtet nicht die zuvor benutzten 3 Bedingungen, sondern prüft in periodischen Zeitabständen, ob ein Circular Wait vorliegt. Ist dies der Fall, werden Recovering-Strategien angewandt, die einen Abbruch der verklemmten Prozesse realisieren. Im Folgenden sollen Methoden zur Deadlockerkennung vorgestellt werden.

Teilweise (partial deadlocks) und vollständige Verklemmungen (deadlocks) kann man durch eine Analyse von Petri-Netzen erkennen und somit vermeiden. Dazu wird das Hilfsmittel des Erreichbarkeitsgraphen genutzt:
**Definition 6.6** (Erreichbarkeitsgraph). Sei Y ein gegebenes Petri-Netz, $T = \{t_1, t_2, \ldots\}$ die Menge der einfachen Transitionen. Dann heißt eine Markierung $M'$ von M aus erreichbar, falls $t_1 t_2 t_3 \ldots t_n \in T^*$ und ein $n \in \mathbb{N}_0$ existieren, so daß

$$M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \ldots \xrightarrow{t_n} M'.$$

Hierfür schreibt man auch $M \xrightarrow{t_1 t_2 \ldots t_n} M'$. $M'$ wird hierbei als Folgemarkierung von $M$ bezeichnet.

Die Erreichbarkeitsmenge $E_Y$ zu dem Petri-Netz $Y$ enthält die Anfangsmarkierung $M_0$ und alle Markierungen, die von dieser aus erreichbar sind, d.h.

$$E_Y = E_Y(M_0) := \{M' : \exists n \in \mathbb{N}_0, t_1, \ldots, t_n \in T^* : M_0 \xrightarrow{t_1 \ldots t_n} M'\}.$$  

Der Erreichbarkeitsgraph verbindet über eine Kante alle Markierungen $M, M' \in E_Y$, für die ein $t_i \in T$ existiert, so daß $M \xrightarrow{t_i} M'$. Zwischen $M$ und $M'$ existiert also genau dann eine Kante im Erreichbarkeitsgraphen, wenn $M'$ unmittelbare Folgemarkierung von $M$ ist. $t_i$ wird zur Kantenbeschriftung benutzt. Abbildung 6.24 zeigt den Erreichbarkeitsgraphen zu Abbildung 6.17.

**Abbildung 6.24: Erreichbarkeitsgraph zu Abbildung 6.17**

In der Regel entsteht als Erreichbarkeitsgraph ein zyklischer, stark zusammenhängender Graph, bei dem jeder Zustand, d.h. jede Markierung von jeder beliebigen anderen aus erreichbar ist. Ein solches System heißt dann *verklemmungsfrei*. Es kann beliebig lange arbeiten, ohne in einen Zustand der Verklemmung zu gelangen.


**Definition 6.8** (Teilweise Verklemmung). Sei wieder $Y$ ein Petri-Netz und $E_Y$ die zugehörige Erreichbarkeitsmenge. Existiere weiterhin eine Markierung $M \in E_Y$, die nicht (durch

Befindet sich ein System im Zustand der Verklemmung, so befindet es sich auch im Zustand der teilweisen Verklemmung. Es kann jedoch teilweise Verklemmungen geben, die keine (echten) Verklemmungen sind.


Abbildung 6.26: Entstehen eines Deadlocks

$$M_0 = (1,0,0,1,0)$$
$$M_1 = (0,1,0,0,0)$$
$$M_2 = (0,0,1,0,1)$$
$$M_3 = (1,0,0,0,1)$$

"Bestand vorhanden"
Prozesskoordination

- Parallele Prozesse
- Koordinierung von parallelen Prozessen
- Wechselseitiger Ausschluß
- Semaphore und Monitore

Inhaltsangabe

7.1 Nebenläufigkeit von Prozessen .................................................. 124
7.2 Kritische Bereiche ................................................................. 125
  7.2.1 Erzeuger/Verbraucher-Problem ........................................ 125
7.3 Wechselseitiger Ausschluß ...................................................... 127
  7.3.1 Softwarelösungen für den wechselseitigen Ausschluß ............... 128
  7.3.2 Hardwarelösungen für den wechselseitigen Ausschluß ............... 134
7.4 Semaphore ................................................................. 137
  7.4.1 Das Prinzip der Semaphore .............................................. 137
  7.4.2 Ablaufsteuerung mit Hilfe von Semaphoren ....................... 139
  7.4.3 Lösung des Erzeuger/Verbraucher-Problems mit Hilfe von Semaphoren 140
  7.4.4 Lösung für das Leser/Schreiber-Problem mit Hilfe von Semaphoren 141
  7.4.5 Das Philosophenproblem .............................................. 142
7.5 Monitore ................................................................. 144
  7.5.1 Motivation der Monitore .............................................. 144
  7.5.2 Prinzip der Monitore .............................................. 146
7.6 Message Passing ............................................................. 150
  7.6.1 Blockierung ......................................................... 150
  7.6.2 Adressierung ......................................................... 151
7.1 Nebenläufigkeit von Prozessen

Eine Aufgabe eines Betriebssystems ist das Vermitteln zwischen sich gegenseitig beeinflussen- den Anforderungen der verschiedenen Programme bzw. Prozesse.

**Definition 7.1** (Abhängige und unabhängige Abläufe). Dabei unterscheidet man zwei Arten von Abläufen:

**Unabhängige Abläufe:** Gegebenenfalls parallel auszuführende Benutzeranträge, die sich gegenseitig nicht beeinflussen, d.h. die für ihre Ausführung unterschiedliche Ressourcen des Systems benötigen (Speicher, Peripheriegeräte, etc.).

**Abhängige Abläufe:** Kooperierende Prozesse, die voneinander abhängig sind, da sie z.B. über gemeinsame Speicherbereiche Daten austauschen oder Betriebsmittel gemeinsam nutzen.

Probleme, die durch parallele Prozesse oder Abläufe entstehen, können gleichermaßen in Verteilten Systemen (z.B. Netzwerken, Parallelrechnern) entstehen, die über mehrere CPUs verfügen, als auch in einem zentralen System mit nur einer CPU.

**Definition 7.2** (Parallelität). Man unterscheidet allerdings je nach System die folgenden Formen der Parallelität von Prozessen:

**Quasi-Parallelität:** Parallele Abläufe sind quasi-parallel, wenn mehrere Prozesse von *einem* Prozessor stückweise bearbeitet werden (preemptive Schedulingstrategien).

**Parallelität:** Parallele Abläufe sind (echt) parallel, wenn *mehrere* Prozessoren ein oder mehrere Prozesse bearbeiten (Mehrprozessorsysteme).

**Definition 7.3** (Nebenläufigkeit). Mehrere Abläufe, die parallel oder quasi-parallel ausgeführt werden, heißen **nebenläufig**, wenn sie inhaltlich zusammenwirken, also voneinander abhängig sind und sich damit gegenseitig beeinflussen können, d.h.

\[
\text{Nebenläufigkeit} = \text{Parallelität} + \text{Abhängigkeit}.
\]

Die Verwaltung paralleler Prozesse wird erst dann interessant, wenn die einzelnen Prozesse voneinander abhängig sind, d.h. wenn es sich um nebenläufige Prozesse handelt. Solche Prozesse müssen dann koordiniert werden.

**Definition 7.4** (Synchrones und asynchrones Ablaufverhalten). Für die Koordination von Prozessen gibt es zwei grundsätzliche Vorgehensweisen:

**Synchrones Ablaufverhalten:** Alle Abläufe sind streng getaktet, so dass jeder der beteiligten Prozesse zu einem bestimmten Zeitpunkt an einer bestimmten Stelle seiner Ausführung angekommen ist.

**Asynchrones Ablaufverhalten:** Alle Abläufe erfolgen ungeregelt – eine Koordination wird durch das Eintreffen bestimmter Ereignisse erzielt (z.B. wenn ein bestimmtes Signal kommt, dann wird ein bestimmter Ablauf initiiert).
7.2 Kritische Bereiche

7.2.1 Erzeuger/Verbraucher-Problem

**Beispiel 7.1.** Wir betrachten zwei Prozesse E und V. E berechnet Resultate, V soll diese auf einem Drucker ausgeben.
E erzeugt also Dinge (meist Daten), die V benötigt („verbraucht“).

Um E und V zu synchronisieren, wird ein endlicher Speicherplatz der Kapazität MAX eingeführt, der von E gefüllt und von V geleert werden kann. Das Synchronisationsproblem besteht darin, dass

1. der Verbraucher nicht auf Daten zugreifen darf, die noch gar nicht produziert worden sind,
2. der Erzeuger keine Daten mehr ablegen darf, falls der Speicherplatz bereits gefüllt ist,
3. der Speicher nicht gleichzeitig von E und V verändert werden darf.

**Einfacher Lösungsansatz**

Vorläufig sei angenommen, dass die Bedingung 3 erfüllt sei. Das Erzeuger/Verbraucher-Problem soll mittels zwei Routinen Erzeuger und Verbraucher gelöst werden, die beide auf eine gemeinsame Variable s, den aktuellen “Lagerbestand”, zugreifen (vgl. Programm 1).

**Programm 1 Routinen für Erzeuger und Verbraucher – einfacher Lösungsansatz**

<table>
<thead>
<tr>
<th>Erzeuger:</th>
<th>Verbraucher:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 REPEAT [ (* unkrit. Ber. *) ]</td>
<td>1 REPEAT [ (* unkrit. Ber. *) ]</td>
</tr>
<tr>
<td>3 &lt;erzeuge Element&gt;</td>
<td>[ (* Beg. krit. Ber. *) ]</td>
</tr>
<tr>
<td>WHILE (s = MAX) DO skip;</td>
<td>[ (* Beg. krit. Ber. *) ]</td>
</tr>
<tr>
<td>5 [ (* Beg. krit. Ber. *) ]</td>
<td>&lt;entnimm Element aus Speicher&gt;;</td>
</tr>
<tr>
<td>7 s := s + 1; [ (* Ende krit. Ber. *) ]</td>
<td>5 s := s - 1; [ (* Ende krit. Ber. *) ]</td>
</tr>
<tr>
<td>9 UNTIL FALSE;</td>
<td>7 [ (* unkrit. Ber. *) ]</td>
</tr>
<tr>
<td></td>
<td>[ &lt;verbrauche Element&gt; ]</td>
</tr>
<tr>
<td></td>
<td>9 UNTIL FALSE;</td>
</tr>
</tbody>
</table>

Beide Routinen sollen unabhängig voneinander ablaufen. Daher sind bei unkontrolliertem Zugriff auf den Lagerbestand s Inkonsistenzen möglich.

**Beispiel 7.2.** Sei z.B. angenommen s = 17, MAX = 20 und ein Scheduling, das alle 3 Zeittakte die Prozessorzuteilung zwischen Erzeuger und Verbraucher umschaltet. Bei der Ausführung entsteht dann u.U. ein Problem wie in 7.1 dargestellt: Obwohl sich die Anzahl der Elemente nicht geändert hat, wurde s inkrementiert.

Offensichtlich ist also der Zugriff auf gemeinsame Ressourcen (hier die gemeinsam genutzte Variable s) der kritische Teil in der vorliegenden Realisierung.

**Lösung:**
Kapitel 7. Prozesskoordination

Abbildung 7.1: Ungeschützter Zugriff auf kritische Daten

1. Wir vermeiden Parallelität, d.h. nur nicht-preemptive Schedulingalgorithmen und damit sequentielle Ausführung der Prozesse wird zugelassen.

2. Vermeidung von Abhängigkeit → Ringpuffer

Lösungsansatz mit Ringpuffer

Eine Lösungsmöglichkeit für dieses Problem besteht im Einsatz eines **Ringpuffers**, wie er in 7.2 definiert ist.

Der Ringpuffer wird in Form eines eindimensionalen Arrays mittels einer Modulo-Funktion \( \text{MOD MAX} \) realisiert. Damit kann vorläufig nur ein Erzeuger und ein Verbraucher realisiert werden.

Der Erzeuger legt sein Element auf den ersten freien Speicherplatz ab, der Verbraucher entnimmt ein Element aus dem am längsten belegten Speicherplatz. Zur einfachen Erkennung

Abbildung 7.2: Darstellung eines einfachen Ringpuffers
der Zustände leer und voll dürfen nur höchstens $\text{MAX} - 1$ Komponenten belegt werden. Dann ist der Speicher leer bei $\text{in} = \text{out}$ und voll bei $(\text{in} + 1) \mod \text{MAX} = \text{out}$ (vgl. Programm 2).

**Programm 2** Routinen für Erzeuger und Verbraucher – Lösungsansatz mit Ringpuffer

<table>
<thead>
<tr>
<th>Erzeuger:</th>
<th>Verbraucher:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>REPEAT</strong></td>
<td>1. <strong>REPEAT</strong></td>
</tr>
<tr>
<td>&lt;erzeuge Element&gt; ;</td>
<td>WHILE (in = out) DO skip ;</td>
</tr>
<tr>
<td>3. <strong>WHILE</strong> $(\text{in} + 1) \mod \text{MAX} = \text{out}$ DO skip ;</td>
<td>&lt;entnimm Element aus Speicher[in]&gt; ;</td>
</tr>
<tr>
<td>5. Speicher[in] := &lt;Element&gt; ;</td>
<td>out := out + 1 $\mod \text{MAX}$ ;</td>
</tr>
<tr>
<td>in := in + 1 $\mod \text{MAX}$ ;</td>
<td>&lt;verbrauche Element&gt; ;</td>
</tr>
<tr>
<td>7. UNTIL FALSE ;</td>
<td>7. UNTIL FALSE ;</td>
</tr>
</tbody>
</table>

Wird das Erzeuger/Verbraucher-Problem auf mehrere Erzeuger oder Verbraucher erweitert, so können beim Schreiben des $\text{in}$- oder des $\text{out}$-Parameters Inkonsistenzen wie beim ursprünglichen Erzeuger/Verbraucher-Problem auftreten.

### 7.3 Wechselseitiger Ausschluß

Solange sich ein Prozess in einem kritischen Bereich befindet (z.B. ein Programm-Modul auf eine globale Variable zugreift), solange kann kein anderer Prozess in diesen kritischen Bereich eintreten.

Es sollen die $n$ Prozesse $P_0, ..., P_{n-1}$ betrachtet werden. Der Programmcode jedes dieser Prozesse läßt sich in kritische und unkritische Bereiche aufteilen.

**Definition 7.5** (Kritischer Bereich). Unter einem **kritischen Bereich** versteht man eine Phase, in der dieser Prozess gemeinsam benutzte (globale) Daten oder Betriebsmittel nutzt.

Alle anderen Phasen werden als **unkritische Bereiche** des Prozesses bezeichnet.

Könnten sich zwei oder mehrere Prozesse gleichzeitig im kritischen Bereich befinden, so wäre das Ergebnis ihrer Operation nicht mehr determiniert, sondern von zufälligen Umständen wie z.B. der Prozessorzuteilung für die Prozesse (Scheduling-Strategie) abhängig.

Deshalb darf zu einem Zeitpunkt nur einem Prozess der Zugang zum kritischen Bereich gestattet werden.

**Definition 7.6.** Das Problem des wechselseitigen Ausschlusses besteht darin, Algorithmen zu finden, die – sobald sich ein Prozess im kritischen Bereich befindet – es keinem anderen Prozess mehr gestatten, den kritischen Bereich zu betreten.

Für eine korrekte Lösung des wechselseitigen Ausschlusses müssen die folgenden drei **Bedingungen** erfüllt sein:

1. **Mutual exclusion:** Zu jedem Zeitpunkt darf sich höchstens ein Prozess im kritischen Bereich befinden.
2. **Progress:** Befindet sich kein Prozess im kritischen Bereich, aber es gibt einen Kandidaten für diesen Bereich, so hängt die Entscheidung, welcher Prozess ihn betreten darf, nur von diesem Kandidaten ab und fällt in endlicher Zeit.
Insbesondere darf es keinen Einfluß haben, wenn ein Prozess in einem unkritischen Bereich terminiert ("stirbt").


### 7.3.1 Softwarelösungen für den wechselseitigen Ausschluß

Im folgenden sollen hierfür zwei unterschiedliche Algorithmen hergeleitet werden.

#### Der Algorithmus von Decker

**Erster Ansatz:** Hierbei wird eine *geschützte*, aber gemeinsam genutzte globale Variable verwendet, die anzeigt, welcher Prozess in den kritischen Bereich eintreten darf. Man kann sich eine Art Schalter für 2 Prozesse vorstellen, der hin- und herschaltet und damit angezeigt, welcher Prozess den kritischen Bereich betreten darf. Im folgenden soll dies nun anhand des Modells des „Protokolliglos“, verdeutlicht werden.

Abbildung 7.3: Gemeinsam genutzte Variable, die anzeigt, welcher Prozess in den kritischen Bereich eintreten darf.

- Durch den Iglo-Eingang und in den Iglo selbst kann zu einem Zeitpunkt nur ein Eskimo.
- Im Iglo befindet sich eine Tafel, auf die ein Wert geschrieben werden kann.
- Möchte nun ein Eskimo (Prozess $P_0$ oder $P_1$) in den kritischen Bereich, so muß er zunächst im Iglo nachschauen, ob seine Nummer auf der Tafel steht, siehe Abbildung 7.3 falls ja, darf er den kritischen Bereich betreten, falls nicht, muß er weiter warten, und von Zeit zu Zeit wieder nachsehen.
  → Dieses Verfahren wird als „busy waiting“, bezeichnet, denn das Nachschauen verbraucht Prozessorzeit.
Kehrt ein Prozess aus dem kritischen Bereich zurück, so muß er in den Iglo gehen und seine eigene Nummer durch eine Nummer eines wartenden Prozesses ersetzen.

Die gemeinsam genutzte Variable (Schalter) soll hier durch \( \text{turn} \) bezeichnet werden.

\[
\text{VAR} \quad \text{turn} : 0 \ldots n-1;
\]

Der Algorithmus soll für \( n = 2 \) Prozesse, d.h. \( P_0 \) und \( P_1 \) betrachtet werden.

\[
\begin{align*}
\text{Prozess } P_0 & \quad \text{Prozess } P_1 \\
\text{1. WHILE} & \quad \text{1. WHILE} \\
\text{2. turn} \not= 0 \text{ DO \{nothing\}}; & \quad \text{2. turn} \not= 1 \text{ DO \{nothing\}}; \\
\text{3. turn} := 1; & \quad \text{3. turn} := 0;
\end{align*}
\]

Hierbei können folgende Probleme auftreten:

1. Die Prozesse können den kritischen Bereich nur abwechselnd betreten. (→ Dies ist absolut unzulässig)

2. Terminiert ein Prozess im unkritischen Bereich, so kann auch der andere höchstens noch einmal den kritischen Bereich betreten und ist dann blockiert. (Verstoß gegen die "Progress"-Eigenschaft)

3. Turn selbst ist eine kritische Variable.

Prüft man den Algorithmus auf die im vorigen Kapitel beschriebenen 3 Bedingungen, so erhält man folgende Bewertung:

1. Mutual Exclusion ⇒ erfüllt
2. Progress ⇒ nicht erfüllt
3. Bounded Waiting ⇒ erfüllt (kein Prozess kann verhungern)

D.h., der Algorithmus ist nicht geeignet.

Zweiter Ansatz:

1. Nun soll nicht mehr die Nummer des Folgeprozesses gespeichert werden, sondern jedem Prozess wird ein Zustand mit Hilfe einer Variablen (flag) zugewiesen.
2. "true" steht für Anspruch auf den kritischen Bereich, und "false" für keinen Anspruch, wie in Abbildung 7.4 zu sehen.
3. Bei eigenem Bedarf für den kritischen Bereich muß zunächst jede Variable der anderen Prozesse kontrolliert werden, sind diese alle false, darf der entsprechende Prozess seine eigene Variable true setzen.
Abbildung 7.4: Jeder Prozess erhält eine Variable “flag”, die den Anspruch auf Eintritt in den kritischen Bereich zeigt.

```
VAR flag: array [0..1] of boolean;

“flag” wird mit false initialisiert.
```

```
Prozess P0
1. WHILE flag[1] DO {nothing};
   flag[0]:= true;
   <kritischer Bereich>;
   flag[0]:= false;

Prozess P1
2. WHILE flag[0] DO {nothing};
   flag[1]:= true;
   <kritischer Bereich>;
   flag[1]:= false;
```

Fällt nun ein Prozess außerhalb des kritischen Bereichs aus, so wird der andere Prozess trotzdem nicht blockiert. Zusätzlich kann ein Prozess mehrfach hintereinander den kritischen Bereich betreten.

Aber: Fällt ein Prozess zwischen dem “Setzen eines Flags auf true” und dem “Betreten des kritischen Bereichs” aus, so funktioniert dieser Algorithmus nicht mehr. Und zwar auch dann nicht, wenn ein Prozessorwechsel nach der WHILE-Überprüfung stattfindet:

1. P0 untersucht in der WHILE Anweisung flag[1], dieses ist false
2. P1 untersucht in der WHILE Anweisung flag[0], dieses ist false
3. P0 setzt flag[0] auf true und betritt den kritischen Bereich
4. P1 setzt flag[1] auf true und betritt den kritischen Bereich

Dies führt zu einem Problem!

D.h. der zweite Ansatz ist mindestens so schlecht wie der erste Ansatz, da er die drei Bedingungen für den wechselseitigen Ausschluß nicht garantiert.

Prüft man den Algorithmus wieder auf unsere 3 Bedingungen, so ergibt sich:
1. Mutual Exclusion $\Rightarrow$ nicht erfüllt.

2. Progress $\Rightarrow$ erfüllt.

3. Bounded Waiting $\Rightarrow$ erfüllt

Lösung: Vertauschen der ersten beiden Zeilen?

→ Durch ungünstiges Scheduling dürfte in diesem Fall evtl. keiner der Prozesse mehr in den kritischen Bereich eintreten $\Rightarrow$ Deadlocksituation. Gegenbeispiel zum Problem des Mutual Exclusion:

\[
\text{flag}[1] = \text{flag}[0] = \text{false} \quad (* \text{kein Prozess will in den kritischen Bereich *})
\]

2. WHILE NOT flag[1] DO {nothing};
4. WHILE NOT flag[0] DO {nothing};
6. flag[1] := true;
8. <kritischer Bereich>;

Dritter Ansatz: Es wird eine größere Vorsicht dadurch erreicht, dass die eigene flag-Variable erst auf true gesetzt wird und dann die andere getestet wird.

Prozess P0

flag[0] := true;
2. WHILE flag[1] DO {nothing};
4. flag[0] := false;

Prozess P1

flag[1] := true;
2. WHILE flag[0] DO {nothing};
4. flag[1] := false;

Nun tritt das Problem auf, dass beide Prozesse ihre flags auf true setzen können und keiner kann den kritischen Bereich betreten.

Eine korrekte Lösung: Es müssen beide Prozesszustände betrachtet und eine Ausführungsreihenfolge eingebracht werden. Letzteres kann mit der Variable turn aus dem ersten Ansatz erfolgen, so dass wir nun eine Kombination der bisherigen Ideen erhalten:

Einführung eines Schiedsrichteriglos, wie in Abbildung 7.5 mit einer Tafel/Variable turn, die anzeigt, welcher Prozess Vorrang hat.

Algorithmus:
Abbildung 7.5: Die korrekte Lösung: Schiedsrichteriglo mit Variable turn zeigt an, welcher Prozess Vorrang hat.

```pascal
VAR flag: ARRAY [0..1] of boolean;
    turn: 0..1;
BEGIN
    flag[0]:= false;
    flag[1]:= false;
    turn :=1;
    PARBEGIN (* Dieses PARBEGIN/PAREND-Konstrukt bedeutet *)
    Prozess 0; (* folgendes: Suspendiere das Hauptprogramm und *)
    Prozess 1; (* führe Prozess 0 und 1 parallel aus. Wenn alle *)
    PAREND (* Prozesse terminiert sind, dann gehe zurück ins *)
    (* Hauptprogramm. *)

Prozess P0:
PROCEDURE P0;
BEGIN
    REPEAT
        flag[0]:= true;
        WHILE flag[1] DO IF turn=1 THEN
            BEGIN
                flag[0]:= false;
                WHILE turn=1 DO {nothing};
                flag[0]:= true
            END;
        END;
        <kritischer Bereich>;
        turn:=1
        flag[0]:= false;
        <unkritischer Bereich>;
        FOREVER
    END.
```
**Prozess P1:**

```pascal
PROCEDURE P1;
BEGIN
  REPEAT
    flag[1] := true;
    WHILE flag[0] DO IF turn=0 THEN
      BEGIN
        flag[1] := false;
        WHILE turn=0 DO {nothing};
        flag[1] := true
      END;
      <kritischer Bereich>;
    turn := 0
    flag[1] := false;
    <unkritischer Bereich>;
  FOREVER
END.
```

Dieser Algorithmus ist nicht intuitiv nachvollziehbar und sehr schwierig zu beweisen. Daher soll im folgenden ein etwas einfacherer Algorithmus betrachtet werden.

**Der Algorithmus von Peterson**

Um das Prinzip der Implementierung jedes Prozesses zu verdeutlichen, die einen kritischen Bereich betreten wollen, nimmt man nochmals das Modell mit den zwei Eskimos und dem Iglo zu Hilfe.

Schritt 1: Ich zeige an, dass *ich will*.

Schritt 2: Ich räume dem Anderen Vortritt ein, d.h. *du darfst*.


**Bemerkung 7.1.** *Es darf zu einem Zeitpunkt nur einer in den kritischen Bereich und das letzte Wort gilt (z.B. wenn beide nacheinander sagen: du darfst)*

**Algorithmus:**

```pascal
VAR flag : ARRAY [0..1] OF boolean;
   turn : 0..1 ;
PROCEDURE P0;
BEGIN
  REPEAT
    flag[0] := true; (* ich will *)
    turn := 1;
    WHILE flag[1] AND turn=1 DO {nothing}; (* warte, falls du willst *)
    (* und du darfst *)
    <kritischer Bereich>;
    flag[0] := false;
    <unkritischer Bereich>;
  FOREVER
END;
```

PROCEDURE P1;
BEGIN
  REPEAT
    flag[1]:= true;
    turn:=0;
    WHILE flag[0] AND turn=0 DO {nothing};
  <kritischer Bereich>;
  flag[1]:= false;
  <unkritischer Bereich>;
  FOREVER
END;
BEGIN
  flag[0]:= false;
  flag[1]:= false;
  turn:=1;
PARBEGIN
  P0;
  P1;
PAREND
END;

Dieser Algorithmus ist auch recht einfach von zwei auf n Prozesse erweiterbar. Die Prüfung unserer 3 Bedingungen ergibt:

1. Mutual Exclusion ⇒ erfüllt.
2. Progress ⇒ erfüllt (abhängig vom Scheduling kann bei 2 Anfragen ein Prozess den kritischen Bereich betreten).

7.3.2 Hardwarelösungen für den wechselseitigen Ausschluß

Im folgenden soll ein ganz anderer Ansatz verfolgt werden. Der wechselseitige Ausschluß soll nicht durch eine geeignete Programmierung erreicht werden, sondern mittels Unterstützung durch die Hardware.

Unterbrechungsvermeidung (Interrupt Disabling): In einem Einprozessrechner können sich Prozesse nicht überschneiden, d.h. wirklich parallel ausgeführt werden, sondern sie können nur unterbrochen und wechselseitig ausgeführt werden. Wechselseitiger Ausschluß kann dadurch erreicht werden, dass man Unterbrechungen unmöglich macht und Prozesse sequentiell abgearbeitet werden, d.h. das preemptive Scheduling wird unterbunden. Insbesondere dürfen Prozesse, die sich im kritischen Bereich befinden, nicht unterbrochen werden. Diese Eigenschaft des Unterbrechungsausschlusses kann durch Primitive auf der Systemebene bereitgestellt werden. Ein Prozess würde dan wie folgt auf einen kritischen Bereich zugreifen (Interrupt-Steuerung):

  REPEAT
  <verbiete Unterbrechungen> (* Disable Interrupt *)
  <kritischer Bereich>;
Bei diesem Ansatz sind aber folgende Nachteile zu beachten:

– Er ist gefährlich und aus Anwendersicht nicht realisierbar. Tritt im kritischen Bereich ein Problem auf, kann durch den Scheduler keine Problembehandlungsroutine eingeschoben werden.

– Er funktioniert in einer Multiprozessorumgebung nicht, da hier echte Parallelität nicht vermieden werden kann.

**Test and Set - Spezielle Maschinenbefehle:** Nun soll ein Mehrprozessorsystem gemäß SMP-Ansatz betrachtet werden. Hier müssen die Unterbrechungsvermeidungen nicht auf den Prozessor bezogen werden, sondern auf das eigentliche Betriebsmittel, z.B. den Speicher. Ein solcher Mechanismus kann z.B. in einem Test-and-set-Befehl implementiert werden, der atomar ausgeführt wird.

```pascal
FUNCTION testset (VAR i: integer): boolean;
BEGIN
  IF i=0 THEN
    BEGIN
      i := 1;
      testset := true
    END;
  ELSE testset := false
END.
```

Dieser Befehl testet den Wert eines Argumentes i und leitet einen logischen Ausdruck zurück.

– i=0 bewirkt i:=1 und true wird zurückgegeben

– i=1 bewirkt i:=1 und false wird zurückgegeben

Ein wechselseitiger Ausschluß wird nun dadurch erreicht, dass ein Programm den test-and-set-Befehl in seine Ausführung integriert und im Fall i=0 den kritischen Bereich betritt. Nach Verlassen des kritischen Bereichs wird i:=1 gesetzt. Das Gesamtprogramm würde wie folgt aussehen:

**Algorithmus:**

```pascal
PROGRAM mutual exclusion;
CONST  n = ...; (* Anzahl der Prozesse *)
VAR i: integer;
PROCEDURE P(k: integer);
BEGIN
  BEGIN
    REPEAT
      REPEAT {nothing} UNTIL testset(i); (* weiter nur bei i=0 (true) *)
    END;
    <kritischer Bereich>;
    i := 0;
    <unkritischer Bereich>;
  END;
END.
```

Eine andere Version dieses Algorithmus sieht fernermassen aus:

```pascal
PROCEDURE test_and_set (VAR lock: Boolean) : Boolean
BEGIN
  test_and_set := lock;
  lock := true;
END
```

Diese Prozedur lädt den Wert von lock und belegt ihn mit dem Wert true (initialisiert mit false). Ist ein Betriebsmittel nicht genutzt, so ist lock auf false gesetzt. Wird es benutzt, oder soll es benutzt werden, so wird lock auf true gesetzt.

Es muss sichergestellt werden, dass diese Prozedur atomar benutzt wird, dass heißt, kein Prozess darf auf sie zugreifen, solange sie nicht durch einen anderen Prozess beendet wurde. (Realisierung: Sperren eines Speicherbusses.)

Nutzung:

```pascal
WHILE test_and_set(lock) DO {nothing};
  (* Ist das Ergebnis true -> Busy Waiting, BM wird benutzt. *)
  (* Ist das Ergebnis false -> setze test_and_set auf true *)
  (* und trete in kritischen Bereich ein. *)
  <kritischer Bereich>;
  lock := FALSE;
  <unkritischer Bereich>;
UNTIL FALSE;
```

Bewertung nach unseren 3 Bedingungen:


2. Progress \(\Rightarrow\) erfüllt.


Lösung: Um dieses unbeschränkte Warten zu vermeiden, müsste ein recht komplizierter Algorithmus ergänzt werden, der die “willigen” Prozesse in zyklischer Reihenfolge abarbeitet.

Eine Variante des vorgestellten Prinzips wird mittels eines sog. Austauschbefehls realisiert. Dabei ist \(i\) mit 0 initialisiert, jeder Prozess hat eine Nummer oder ID, die er vor dem Eintritt
in den kritischen Bereich mit \( i \) austauscht, sofern \( i = 0 \) gilt. Vorteil ist, dass man sehen kann, welcher Prozess sich im kritischen Bereich befindet. Diese Ansätze sind einfach und auf eine beliebige Anzahl von Prozessen anwendbar.

Aber: Da ein Prozess im Zustand “busy waiting” auf den Eintritt in den kritischen Bereich wartet, konsumiert er Prozessorzeit.

Und: Bezogen auf unsere drei Kriterien für den wechselseitigen Ausschluß sind Starvation und Deadlock möglich.

### 7.4 Semaphore


**Beispiel 7.3.** Ein Parkhaus nutzt einen Zähler zum Modellieren der Anzahl freier Parkplätze. Dieser wird am Morgen eines jeden Tages initialisiert, beim Einfahren eines Autos wird der Zähler um eins verringert, beim Ausfahren um eins erhöht. Steht der Zähler auf 0, so muß ein Auto solange in einer Warteschlange (fair), oder um den Block fahrend (unfair) warten, bis ein anderes Auto das Parkhaus verläßt und dem Zähler signalisiert, dass jetzt wieder ein freier Platz zur Verfügung steht. Die Operationen Initialisieren, Inkrementieren und Dekrementieren sind als atomare Operationen realisiert.

Die Abstraktion dieses Mechanismus führt zum Konzept des Semaphors.

#### 7.4.1 Das Prinzip der Semaphore

Ein Semaphore \( S \) ist eine Integervariable, die nur durch 3 atomare Operationen verändert werden kann. Diese heißen:

1. init(S, Anfangswert) setzt \( S \) auf den Anfangswert. (Initialisierung)
2. wait(S) oder auch P(S) (nach niederländisch “,proberen”, = probieren) versucht \( S \) zu verringern. (Dekrementierung)
3. signal(S) oder auch V(S) (nach niederländisch “,verhogen”, = erhöhen) erhöht \( S \). (Inkrementierung)

In den meisten Fällen ist einem Semaphore eine Warteschlange zugeordnet, um auszuschließen, dass Prozesse über eine längere Zeit in einer Wartschleife verweilen und damit unnötig Ressourcen belegen. Solche assoziierten Warteschlangen verhindern ein busy waiting, indem sie Prozessnummern in der Reihenfolge der Anfragen festhalten. Daraufhin können wartende Prozesse suspendiert werden und beanspruchen so keine CPU-Zeit.
Wird der kritische Bereich frei, so wird der in der Warteschlange folgende Prozess rechnend gesetzt und kann den kritischen Bereich betreten.

**Bemerkung 7.2.** Eine Implementierung ohne Warteschlange würde dazu führen, dass wenn bereits \( S := 0 \) gilt, \( \text{wait}(S) \) nicht ausgeführt werden kann, da keine Ressourcen mehr vorhanden sind. \( S \) bliebe dann so lange auf dem Wert 0, bis ein anderer Prozess ein \( \text{signal}(S) \) ausgeführt hat. Abhängig vom Scheduling könnten sich hieraus unfaire Abläufe ergeben. Angenommen es existieren drei Prozesse \( P_1, P_2 \) und \( P_3 \), die nacheinander folgende Aufrufe tätigen

\[
P_1: \quad \text{wait}(S) \\
P_3: \quad \text{wait}(S) \\
P_2: \quad \text{signal}(S)
\]

dann könnte abhängig vom Scheduling auch erst der Prozess \( P_3 \) den Zugriff auf \( S \) erhalten. Zudem impliziert die Implementierung ohne Semaphoren die Anwendung von Busy Waiting, was zu unötigem Verbrauch an Prozessorzeit führt.

Bei der Implementierung mit einer Warteschlange wird \( S \) stets um 1 verringert. Dies ist auch dann der Fall, wenn ein Prozess \( P \) \( \text{wait}(S) \) aufruft obwohl bereits \( S \leq 0 \) gilt. Der aufrufende Prozess wird dann an der \(-S\)-ter Stelle in der Warteschlange aller wartenden Prozesse einreihet.

**Beispiel 7.4.** Der Wert von \( S \) ist gerade \(-1\). Ein Prozess \( P_1 \) ruft \( \text{wait}(S) \) auf, d.h. die Semaphore wird um 1 dekrementiert (\( S := S - 1 = -2 \)). Somit reiht sich \( P_1 \) an der Stelle \(-S := (-(−2)) = 2\) ein.

Die drei Operationen werden wie folgt realisiert:

1. \( \text{init}(S, \text{Anfangswert}) \): bewirkt die Zuweisung
   \[
   S := \text{Anfangswert};
   \]
   Sinnvoll ist es, als Anfangswert die Anzahl der Prozesse zu setzen, die sich gleichzeitig im kritischen Bereich aufhalten dürfen.
   Wechselseitiger Ausschluß wird durch \( S := 1 \) erreicht, man spricht dann auch von **binären Semaphoren** im Gegensatz zu sogenannten Zählsemaphoren.

2. \( \text{wait}(S) \) bewirkt

   \[
   \text{WHILE } (S <= 0) \text{ DO } \text{skip} ; \]
   \[
   S := S - 1 ;
   \]
   oder bei Vorhandensein einer Warteschlange:
   \[
   S := S - 1 ;
   \]
   \[
   \text{IF } (S < 0) \text{ THEN }
   \]
   \[
   \text{ordne Prozess in Warteschlange an Position } -S \text{ ein} >;
   \]
   Beide Realisierungen müssen als atomare Operationen erfolgen.

3. \( \text{signal}(S) \) bewirkt die Zuweisung
Semaphore

1. S := S + 1;

oder bei Verwendung einer Warteschlange

1. S := S + 1;
   IF (S <= 0) THEN
      BEGIN
         <setze den am längsten wartenden Prozess rechnend>;
      END;
   END;

   IF (S <= 0) THEN
      BEGIN
         <setze den am längsten wartenden Prozess rechnend>;
      END;
   END;

Beide Realisierung müssen wieder als atomare Operationen erfolgen.

Mit diesen drei Operationen kann man das generelle Problem des wechselseitigen Ausschlusses wie folgt lösen:

1. Globale Initialisierung für Prozess \( P_i \): \texttt{init}(S,1);

2. Für jeden Prozess \( P_i \) mit \( i \in \{0, \ldots, n-1\} \):
   \begin{enumerate}
   \item \texttt{wait}(S);
   \item \texttt{critical Bereich};
   \item \texttt{signal}(S);
   \item \texttt{unkritischer Bereich};
   \end{enumerate}
   \texttt{UNTIL FALSE};

Dieser Algorithmus soll an vier verschiedenen Beispielen verdeutlicht werden.

7.4.2 Ablaufsteuerung mit Hilfe von Semaphoren

Seien zwei Prozesse \( X \) und \( Y \) gegeben, wobei \( Y \) vor \( X \) ausgeführt werden soll.
Dazu benötigen wir einen Semaphor \( a \) mit \texttt{init}(a,0). Dann ergibt sich ein wechselseitiger Ablauf von \( X \) und \( Y \) (vgl. Abbildung 7.6).
Diese “umgekehrte”, Semaphoren­nuttung bewirkt, dass \( X \) solange warten muß, bis \( Y \) fertig ist.

Abbildung 7.6: Ablaufsteuerung von zwei Prozessen mit Hilfe von Semaphoren
7.4.3 Lösung des Erzeuger/Verbraucher-Problems mit Hilfe von Semaphoren

Das aus Abschnitt 7.2 auf Seite 125 bekannte Erzeuger/Verbraucher-Problem soll nun mittels einer Semaphore $S$ realisiert werden. Es soll also exklusiver Zugriff auf das Lager, welches $\text{MAX}$ Speicherplätze aufnehmen kann, garantiert werden, d.h. wir benötigen eine binäre Semaphore zur Realisierung dieses kritischen Bereiches, also

$$\text{init}(S, 1);$$

Somit erhalten wir für den Zugriff auf den kritischen Bereich sowohl im Erzeuger als auch im Verbraucher folgende Anweisungen:

1. $\text{wait}(S);$  
   $<$Element auf Speicher ablegen oder vom Speicher holen$>$;  
2. $\text{signal}(S);$  

Um allerdings die Nebenbedingung, dass maximal $\text{MAX}$ Speicherplätze zur Verfügung stehen, ebenfalls zu garantieren, können zwei Zählsemaphoren verwendet werden, und zwar derartig, dass die erste Semaphore $b$ die Anzahl der belegten Speicherplätze (den Bestand), die zweite Semaphore $p$ die Anzahl der freien Speicherplätze (den verfügbaren Platz) repräsentiert. Ist also $b = 0$, so ist das Lager leer, ist hingegen $p = 0$, so ist das Lager voll.

Die Initialisierung für $b$ und $p$ geschieht wie folgt:

1. $\text{init}(b, 0);$  
2. $\text{init}(p, \text{MAX});$

Während der gesamten Laufzeit des Algorithmus soll also $b + p = \text{MAX}$ gelten, d.h. belegte und freie Speicherplätze ergeben zusammen alle Speicherplätze. 

Somit ergibt sich der in Programm 3 auf der nächsten Seite gezeigte Algorithmus für Erzeuger und Verbraucher.

Ein Vertauschen der ersten beiden $\text{wait}$-Operationen (Zeilen 4 und 5) kann zu Konflikten führen, da in diesem Fall der kritische Bereich durch einen Erzeuger blockiert ist, ohne dass Speicherplatz frei ist. Hätten wir:

1. $(* \text{ Erzeuger *})$  
   $s := 1; \quad b := 0; \quad p := \text{Max};$  
2. $<\text{erzeuge}>;$  
3. $\text{wait}(s);$  
4. $\text{wait}(p);$  
   $<\text{lege Element ab}>;$  
5. $\text{signal}(s);$  
6. $\text{signal}(b);$  
7. $(* \text{ Deadlock möglich *})$

$(* \text{ Verbraucher *})$
8. $\text{wait}(s);$  
9. $s := 0; \quad b := 0; \quad p := \text{Max};$  
10. $\text{wait}(b);$  
11. $<\text{entnimm Element}>;$  
12. $\text{signal}(s);$  
13. $\text{signal}(p);$  
14. $<\text{verbrauche Element}>;$

dann kann in Zeile 5 ein Deadlock entstehen, d.h. $\text{wait}(s)$ und $\text{wait}(p/b)$ dürfen nicht vertauscht werden.

Die Reihenfolge der $\text{signal}$-Befehle ist jedoch beliebig.
Programm 3 Algorithmen für Erzeuger und Verbraucher mit Semaphoren

(* Erzeuger: *)
REPEAT
<erzeuge Element>;
wait(p);
wait(s);
<lege Element im Speicher ab>;
signal(s);
signal(b);
UNTIL FALSE;
(* Verbraucher: *)
REPEAT
wait(b);
wait(s);
<entnimm Element aus Speicher>;
signal(s);
signal(p);
<verbrauche Element>;
UNTIL FALSE;

7.4.4 Lösung für das Leser/Schreiber-Problem mit Hilfe von Semaphoren

Das Leser/Schreiber- oder R/W- (Reader/Writer-) Problem ist ein klassisches Problem, insbesondere bei der Verwaltung von Datenbanken. Es seien zwei Arten von Prozessen gegeben:

1. Schreiber, die Daten modifizieren können, und
2. Leser, die Anfragen stellen, d.h. Daten lesen können.

Diese Prozesse sollen Zugriff auf ein gemeinsames Datenobjekt (z.B. eine Datei) besitzen. Dabei sollen Schreibzugriffe exklusiv erfolgen, d.h. wenn Daten geschrieben werden, so darf nur genau ein Prozess aktiv sein, nämlich der Schreiber (weitere Schreiber und Leser müssen ausgeschlossen sein).

Es gibt drei Ansätze, Fairneß für Leser und Schreiber zu garantieren:

1. Ein Leser muß nur dann auf Eintrittserlaubnis in den kritischen Bereich warten, wenn ein Schreiber gerade aktiv ist.

   Dabei könnten jedoch Leser einen sogenannten “Verschwörerkreis” bilden (d.h. ein Leser ist immer aktiv), der ewig den Schreiber blockiert. Diese Lösung ist also “Leserfreundlich”.

2. Wartet ein Schreiber, so darf ein Leser nicht in den kritischen Bereich eintreten.

   Analog jedoch können in diesem Fall die Schreiber die Leser blockieren (“Schreiberfreundlich”).
3. Daher sollen nun Lese- und Schreibphasen alternieren:
   Ist also gerade Lesephase (ein Leser aktiv) und ein Schreiber meldet sich an, so werden
   keine neuen Leser mehr zugelassen und der Schreiber wird aktiv, sobald alle gerade akti-
   ven Leser beendet wurden. Analog werden alle nachfolgenden Anfragen von Schreibern
   in der Schreibphase abgewiesen, sobald sich ein Leser angemeldet hat.

   Es soll für den einfachen ersten Fall ein Algorithmus angeben werden (vgl. Programm \[4\]),
   wobei

   1. ein binärer Semaphor \(S\) für Schreibphase mit \(\text{init}(S, 1)\),
   2. ein binärer Semaphor \(L\) für Lesephase mit \(\text{init}(L, 1)\) und
   3. eine Zählvariable \(n\), die die Anzahl aktiver Leser enthält (zu Anfang 0), benötigt werden.

   **Programm 4** Lösung des R/W-Problems mit Semaphoren, Leser-freundlich

   \begin{verbatim}
   (* Writer: *)
   REPEAT
   wait(S);
   <Schreibvorgang>;
   signal(S);
   UNTIL FALSE;
   (* Reader: *)
   REPEAT
   wait(L); (* \(l := l - 1\), exklusiver Zugriff auf \(n\) *)
   n := n+1;
   IF (n = 1) THEN wait(s);
   IF (n = 0) THEN signal(s);
   (* Freigabe der Schreibphase, wenn keine Leser mehr aktiv *)
   signal(l);
   
   (* Falls kein Leser vorhanden, blockiere Schreibphase *)
   (* ist Schreiber aktiv, so warte, bis Schreiber fertig, *)
   (* dann blockiere; sonst blockiere sofort. *)
   signal(l);
   
   <Lesevorgang>; (* nicht wenn \(n=0\) und Schreiber aktiv *)
   wait(l); (* exklusiver Zugriff auf \(n\) *)
   n := n - 1;
   IF (n = 0) THEN signal(s);
   (* Freigabe der Schreibphase, wenn keine Leser mehr aktiv *)
   signal(l);
   \end{verbatim}

3.4.5 Das Philosophenproblem

Auch dieses ist ein klassisches Problem, das die Problematik des exklusiven Zugriffes ver-

deutlicht, jedoch aus Sicht eines Betriebssystems in der Praxis von eher geringer Relevanz

Philosophen verbringen ihr Leben ausschließlich mit Essen und Denken. Seien fünf Philoso-

phen gegeben, die an einem runden Tisch sitzen (vgl. Abbildung \[7.7\]). Vor jedem Philosophen
besonders sich ein Teller voller Reis und zwischen je zwei Tellern befindet sich ein Eßstäbchen. Ein Philosoph, der hungrig ist, benötigt sowohl das linke als auch das rechte Stäbchen, um essen zu können. Hat er zwei Stäbchen, so ißt er, bis er satt ist, erst dann legt er die Stäbchen an die ursprünglichen Plätze zurück, so dass seine Nachbarn davon Gebrauch machen können.

Aus Betriebssystemsicht gibt es also fünf Prozesse \( P_0, \ldots, P_4 \) und fünf Betriebsmittel \( BM_0, \ldots, BM_4 \). Ein Prozess benötigt zwei Betriebsmittel, um aktiv zu werden, und zwar der Prozess \( P_i \) die Betriebsmittel \( BM_i \) (linkes Stäbchen) und \( BM_{(i+1) \mod 5} \) (rechtes Stäbchen).

Damit ergibt sich für den Prozess \( P_i \) der Ablauf:

```plaintext
REPEAT
  Denken; (* unkritisicher Bereich *)
  * essen *
UNTIL FALSE;
```

**Einfacher Lösungsansatz für das Philosophenproblem**

Jedes Betriebsmittel (Stäbchen) wird durch eine Semaphore \( BM_0 \) bis \( BM_4 \) repräsentiert, die mit \( \text{init}(BM_i, 1) \) initialisiert wird.

Damit ergibt sich für \( P_i \) folgender Algorithmus:

```plaintext
REPEAT
  Denken; (* unkritisicher Bereich *)
  * essen *
UNTIL FALSE;
```

```plaintext
wait (BM_i);
wait (BM_{(i+1) \mod 5});
Essen; (* kritisicher Bereich *)
signal (BM_i);
signal (BM_{((i+1) \mod 5)});
UNTIL FALSE
```
Bei dieser Realisierung ergibt sich jedoch das Problem, dass in dem Fall, dass alle Philosophen (quasi) gleichzeitig zu essen beginnen, \texttt{wait(BM}_i\texttt{)} fünfmal quasi gleichzeitig ausgeführt wird. Jeder Prozess hält also ein Betriebsmittel (das linke Stäbchen), wartet jedoch nun vergeblich auf das zweite Betriebsmittel (das rechte Stäbchen). Es entsteht ein \textbf{Deadlock} (Verklemmung).

Es existieren verschiedene Lösungen dieses Problems, z.B. kann ein Philosoph zum “Rechts-händer” werden, d.h. er wartet zuerst auf das rechte, dann auf das linke Stäbchen, bei diesem Philosophen werden also die Zeilen 3 und 4 in obigem Algorithmus vertauscht.

\section*{Lösung des Philosophenproblems unter Vermeidung von Deadlocks}

Es wird vorausgesetzt, dass maximal vier Philosophen gleichzeitig zu ihrem ersten Stäbchen Zugriff bekommen. Zu diesem Zweck wird ein weiterer Semaphor \texttt{Erlaubte_Esser} eingeführt, der durch \texttt{init(Erlaubte_Esser, 4)} auf 4 beschränkt ist.

Damit können höchstens vier Philosophen ihre linken Stäbchen nehmen und immer mindestens ein Philosoph auch das rechte Stäbchen erhalten, mindestens ein Philosoph kann also zu jeder Zeit essen.

Für \(P_i\) ergibt sich nun der folgende Algorithmus:

\begin{verbatim}
1 REPEAT
2   Denken;
3   wait(Erlaubte_Esser);  (* dekrementiere Erlaubte_Esser *)
4   wait(BM_i);
5   wait(BM_{((i+1) \mod 5)});
6   Essen;
7   signal(BM_i);
8   signal(BM_{((i+1) \mod 5)});
9   signal(Erlaubte_Esser);  (* inkrementiere Erlaubte_Esser *)
10 UNTIL FALSE;
\end{verbatim}

\section*{7.5 Monitore}

\subsection*{7.5.1 Motivation der Monitore}

Wir wollen ein - scheinbar korrektes - Beispiel betrachten, das das Erzeuger/Verbraucher-Problem für einen unendlich großen Speicher realisieren soll. Dabei gehen wir von 2 binären Semaphoren \(S\) und \texttt{NOT_EMPTY} aus. Kritisch ist - neben dem exklusiven Zugriff - nur, dass der Verbraucher nicht aus einem leeren Speicher entnimmt. Weiterhin sei \(n\) eine Zähllvariable, die mit \(n := 0\) initialisiert ist.

\begin{verbatim}
1 init(S,1);  (* Zugriff *)
2 init(NOT_EMPTY,0);  (* Ausgangssituation: empty, d.h. not_empty=false *)
\end{verbatim}

Es ergibt sich der Quellcode aus Programm[5] Durch den Scheduler wird nun eine Sequenz von Teilen der Erzeuger- und Verbraucherprozesse dem Prozessor zugewiesen, die folgende Variablenbelegung bedingt:
**Programm 5** Routinen für Erzeuger und Verbraucher mit 2 binären Semaphoren und 1 Zählvariable n

Erzeuger E:

```plaintext
REPEAT
2 <erzeuge Element>;
  wait (S); (*/ Beg. krit. Ber. */)
4 <lege Element im Speicher ab>;
  n := n + 1
6 IF n = 1 THEN signal (NOT_EMPTY);
  signal (S); (*/ Ende krit. Ber. */)
8 UNTIL FALSE;
```

Verbraucher V:

```plaintext
wait (NOT_EMPTY);
2 REPEAT
4 <entnimm Element>;
4 n := n - 1;
6 signal (S); (*/ Ende krit. Ber. */)
8 IF n=0 THEN wait (NOT_EMPTY);
UNTIL FALSE;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
<th></th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>initial</td>
<td>n := 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>E krit. Bereich</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>V wait (NOT_EMPTY)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>V krit. Bereich</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>E krit. Bereich</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>V IF n = 0 m = 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Then wait(NOTEMPTY)</td>
<td>f</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>V krit. Bereich</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>V IF n = 0 m = 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Then wait(NOTEMPTY)</td>
<td>f</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>V krit. Bereich</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*= wartet auf m = 1

Obwohl die Programmmodule für den Erzeuger und den Verbraucher offensichtlich korrekt programmiert sind, ist im Ablauf ein Fehler aufgetreten. Um diesen zu beseitigen, führen wir eine Hilfsvariable m ein, die innerhalb des kritischen Bereichs beim Verbraucher einen Wert zugewiesen bekommt, der zu einem späteren Zeitpunkt ausgewertet wird. Eine korrekte Lösung ergibt sich damit in folgender Form, wobei der Erzeuger vollständig übernommen wird:

```plaintext
n := 0;
init (S,1);
init (NOT_EMPTY, 0);
```

Damit ergibt sich der Quellcode aus Programm 6. Die neue Hilfsvariable m kann durch Erzeugerbefehle nicht manipuliert werden! Betrachten wir den obigen Abarbeitungspfad, so sieht man (vgl. auch Tabelle), dass diese Lösung nicht zu dem beschriebenen Konflikt führt.

**FAZIT** Obwohl Semaphore ein mächtiges und flexibles Tool für die Realisierung des wechselseitigen Ausschlusses sind, ist es in gewissen Situationen doch schwierig, mit ihnen ein korrektes Programm zu erstellen. Diese Schwierigkeit liegt darin begründet, dass Wait- und Signal-Operationen oft so über das Programm verstreut sind, dass es nicht einfach ist, ihre globale Wirkung zu erkennen.
Kapitel 7. Prozesskoordination

Programm 6 Routinen für Erzeuger und Verbraucher mit Hilfsvariable m

Erzeuger E:

1. REPEAT
2. <erzeuge Element >;
3. wait (S ); (* Beg. krit. Ber. *)
4. <lege Element im Speicher ab >;
5. n : = n + 1
6. IF n = 1 THEN signal (NOT_EMPTY);
7. signal (S ); (* Ende krit. Ber. *)
8. UNTIL FALSE;

Verbraucher V:

1. VAR m: INTEGER (* Hilfsvariablen *)
2. wait (NOT_EMPTY);
3. REPEAT
4. wait (S ); (* Beg. krit. Ber. *)
5. <entnimm Element >;
6. n : = n − 1;
7. m : = n ; (* NEU! *)
8. signal (S ); (* Ende krit. Ber. *)
9. <verbrauche Element >;
10. IF m = 0 THEN wait (NOT_EMPTY);
11. until FALSE;

Aus diesem Grund wollen wir die Monitore als ein weiteres höherprogrammiersprachliches Konstrukt einführen, das eine äquivalente Funktionalität bezüglich des wechselseitigen Auschlusses bereitstellt, jedoch einfacher zu kontrollieren ist.

7.5.2 Das Prinzip der Monitore


Definition 7.7 (Monitor). Ein Monitor ist ein Objekt, das sich im wesentlichen aus einer Menge von Prozeduren auf gegebenenfalls gemeinsam genutzten Daten zusammensetzt. Entscheidend ist, dass der Monitor – zu jedem Zeitpunkt – stets nur von (höchstens) einem Prozess genutzt werden darf. Fordert demzufolge ein Prozess eine Monitorprozedur an und erhält er die Erlaubnis, sie abzuarbeiten, so läuft diese Prozedur atomar ab.

Dieses Konzept wurde dann auch in Programmiersprachen wie Pascal-Plus und Modula 2+3 implementiert.

Einen Monitor wollen wir nun als ein Softwaremodul betrachten, das aus folgenden Bestandteilen besteht:

1. einer oder mehreren Prozeduren
2. lokalen Daten
3. einer Warteschlange für ankommende Prozesse

Der Grundgedanke besteht nun darin, dass auf die lokalen Variablen nur durch Zugang zu den Monitorprozeduren zugegriffen werden darf. Insbesondere ist es ausgeschlossen, dass eine externe Prozedur auf diese lokalen Daten zugreift!

Dies bedeutet insbesondere: ein Prozess betritt den Monitor, indem er auf eine der Monitorprozeduren zugreift.
Ferner darf zu einem Zeitpunkt immer nur ein Prozess im Monitor ausgeführt werden. Alle anderen Prozesse, die Zugriff auf den Monitor erfordern

– haben ihn bereits wieder verlassen oder
– warten in der Warteschlange für ankommende Prozesse auf ihren Eintritt oder
– sind so lange suspendiert, bis der Monitor für sie wieder zur Verfügung steht, also z.B. bis der zur Zeit im Monitor auf eine Prozedur zugreifende Prozess mit der Abarbeitung fertig ist und den Monitor verlässt.

Diese Grundgedanken wollen wir nun in einem Modell (vgl. Abbildung 7.8) zusammenfassen, das wie folgt aussehen kann:


Beispiel 7.5. Nehmen wir an, bei vollem Speicher würde ein Erzeuger den Monitor betreten. Dann müsste der Erzeugerprozess suspendiert werden, bis die Bedingung erfüllt ist, dass der Speicher nicht mehr voll ist. Während der Suspendierung - die in einem Wartebereich außerhalb des eigentlichen Monitors erfolgt - kann ein anderer Prozess, vielleicht ein Verbraucher, den Monitor betreten und auf eine Prozedur zugreifen, die den kritischen Bereich über die lokalen Variablen anspricht (vgl. auch Programm [7]).

Bemerkung: Statt eingüßen(x) bzw. entnehmen(x) sind manchmal auch die Schreibweisen Endlicher_Speicher.einfügen(x) bzw. Endlicher_Speicher.entnehmen(x) gebräuchlich.

Ein Hauptprogramm zur parallelen Ausführung von Erzeuger und Verbraucher könnte man durch folgende Zeilen ergänzen:

```
1 BEGIN (* Hauptprogramm *)
2   PAR BEGIN (* Schlüsselwort für parallele Ausführung *)
3   Erzeuger; Verbraucher;
4   PAR END;
5 END;
```

Die korrekte Darstellung z.B. für den Erzeuger und den Verbraucher hätte dann die Syntax aus Programm[8].

Realisierung der Synchronisation in einem Monitor Wir wollen noch einmal den Fall betrachten, dass ein Prozess den Monitor durch Aufruf einer entsprechenden Monitorprozedur (z.B. eingüßen(x)) betritt, dieser Prozess jedoch noch nicht ausgeführt werden kann, solange eine bestimmte Bedingung (z.B. not_full) nicht erfüllt ist. Wir benötigen folglich ein Konzept, bei dem ein Prozess, der auf ein Ereignis wartet, in suspendiertem Zustand den Monitor freigibt, bis dieses Ereignis eingetreten ist. Während dieser Freigabe kann der Monitor von anderen Prozessen betreten werden. Bei Ereigniseintritt kann der suspendierte Prozess den Monitor wieder betreten und seine Abarbeitung an der Stelle fortsetzen, an der er zuvor unterbrochen wurde. Der Monitor unterstützt diese Synchronisation durch die Nutzung von Bedingungsvariablen (condition variables), die nur innerhalb des Monitors verfügbar sind, d.h. aufgerufen werden können (vgl Programm[7]). Es gibt 2 Funktionen, die auf diese Bedingungsvariablen ausgeführt werden können:

- `cwait(c)`: suspendiert die Ausführung des Prozesses, der diese Operation aufruft, bis die Bedingung c eingetreten ist. Logisch gesehen reiht sich dieser Prozess in die Warteschlange ein. Der Monitor steht nun anderen Prozessen zur Verfügung.

- `csignal(c)`: Nimmt die Ausführung eines suspendierten Prozesses, der auf die Bedingung c wartet, wieder auf. Falls es mehrere solche wartende Prozesse gibt, wird einer von ihnen ausgewählt (z.B. der am längsten wartende im Fall von FIFO). Falls es keinen Prozess gibt, der auf c wartet, so wird nichts gemacht.
**Programm 7** Monitor für den Zugriff auf den endlichen Speicher

```pascal
MONITOR Endlicher_Speicher;

buffer: ARRAY[0...N] OF char;
nextin, nextout: INTEGER;
count: INTEGER;
not_full, not_empty: CONDITION;

PROCEDURE einfügen (x: char);
BEGIN
  IF count = N THEN cwait(not_full);
  buffer[nextin]:= x;
  nextin := nextin + 1 MOD N;
  count := count + 1;
  csignal(not_empty);
END;

PROCEDURE entnehmen(x: char);
BEGIN
  IF count = 0 THEN cwait(not_empty);
  x := buffer[nextout];
  nextout := nextout + 1 MOD N;
  count := count - 1;
  csignal(not_full);
END;

BEGIN
  nextin := 0; nextout := 0; count := 0;
END;

(* Erzeuger: *)
REPEAT
  <erzeuge Element x>;
  einfügen(x);
UNTIL FALSE;

(* Verbraucher: *)
REPEAT
  entnehmen(x);
  <verbrauche Element x>;
UNTIL FALSE;
```
Kapitel 7. Prozesskoordination

Programm 8 Erzeuger-Verbraucher mit endlichem Speicher

Erzeuger E:
PROCEDURE Erzeuger;
VAR x: char;
BEGIN
REPEAT
<erzeuge Element x>;
einfügen (x);
UNTIL FALSE;
END;

Verbraucher V:
PROCEDURE Verbraucher;
VAR x: char;
BEGIN
REPEAT
entnehmen (x);
<verbrauche Element x>;
UNTIL FALSE;
END;

Beachte, dass diese cwait/csignal-Operationen eine völlig andere Arbeitsweise haben als die signal/wait-Operationen der Semaphore. Sie sind weder Zähllvariablen noch Operationen auf solche. So kann ein csignal-Signal im Falle nicht wartender Prozesse z.B. ohne Bedeutung sein, ein signal-Signal hat jedoch immer eine Wirkung.

7.6 Message Passing

Prozesskoordination erfordert Synchronisation und Kommunikation. Diese soll hier mittels Message Passing, d.h. Nachrichtenaustausch realisiert werden. Für den Nachrichtenaustausch werden zwei Primitive genutzt:
– send(destination, message)
– receive(source, message)

Für die Ausführung dieser Primitive muß zwischen den Prozessen eine Synchronisation stattfinden. Hierbei sind verschiedene Aspekte zu beachten.

7.6.1 Blockierung

Dabei unterscheiden wir 3 Fälle:

1. Blocking Send, Blocking Receive
   Der sendende Prozess wird blockiert, bis die Nachricht vollständig abgeschickt und beim Empfänger angekommen ist.
   Beim Empfänger wird nach einem receive-Aufruf auch eine Blockierung vorgenommen, bis eine Nachricht eingetroffen ist.
   → Diese Kombination wird auch als Rendezvous bezeichnet. Sie ermöglicht eine feste Synchronisation zwischen zwei Prozessen.
   Bei den folgenden Fällen gehen wir von einem nichtblockierenden Senden aus.

2. Nonblocking Send, Blocking Receive
   Nach dem Abschicken einer Nachricht kann der Sender mit seiner Ausführung fortfahren. Der Empfänger ist bis zum Erhalt einer Nachricht blockiert.
   → Dies ist offensichtlich die sinnvollste Kombination, denn eine oder mehrere Nachrichten können effizient, d.h. unmittelbar nacheinander an verschiedene Empfänger gesendet werden.
3. **Nonblocking-Receive-Ansätze**

<table>
<thead>
<tr>
<th></th>
<th>mit Pufferung</th>
<th>ohne Pufferung</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit Blockierung</td>
<td>Nachrichtenaustausch mit Blockierung und Pufferung</td>
<td>analog NoPB</td>
</tr>
<tr>
<td></td>
<td>NPB Problem: Redundanz!</td>
<td>sinnvolle Realisierung</td>
</tr>
<tr>
<td>ohne Blockierung</td>
<td>analog NoBP</td>
<td>analog NoPoB</td>
</tr>
<tr>
<td></td>
<td>sinnvolle Realisierung</td>
<td>Nachrichten gehen verloren!</td>
</tr>
</tbody>
</table>

Abbildung 7.9: Realsisierung mit/ohne Blockierung und Pufferung

<table>
<thead>
<tr>
<th></th>
<th>Blocking Send</th>
<th>Non Blocking Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Receive</td>
<td>1. Rendez - Vous</td>
<td>2. Post - Modell</td>
</tr>
<tr>
<td>Non Blocking Receive</td>
<td></td>
<td>3.</td>
</tr>
</tbody>
</table>

Abbildung 7.10: Realsisierung mit/ohne Blockierung

**7.6.2 Adressierung**

**Problem:** Wie wird in dem Send-Primitiv angegeben, welcher Prozess die Nachricht erhalten soll, d.h. wie wird die Adressierung vorgenommen?
Und analog: Wie kann ein empfangender Prozess Nachrichten entsprechend deren Absender einschränken?

Dabei werden zwei Ansätze unterschieden:

1. **Direkte Adressierung:**
   Das Send-Primitiv enthält einen speziellen Identifikator des Zielprozesses
   Beim Zielprozeß kann
   (a) genau angegeben werden, von welchem Prozess eine Nachricht erwartet wird
      (macht z.B. Sinn bei nebeneinlaufenden Prozessen) oder
   (b) jede Nachricht akzeptiert werden. (ist z.B. für einen Drucker geeignet)
   Der Fall b) ließe sich auch sehr gut mit dem folgenden Ansatz realisieren.

2. **Indirekte Adressierung:**
   Nachrichten werden bei diesem Ansatz nicht direkt vom Sender zum Empfänger geschickt, sondern zunächst an eine gemeinsam genutzte Datenstruktur, die im wesentlichen aus einer Warteschlange besteht, die temporär Nachrichten speichern kann.
   Solche Warteschlangen (Queues) werden auch als Mailboxen bezeichnet.
   Dabei bestehen folgende Möglichkeiten:

   - **1:1** (1 Sender → 1 Empfänger)
   - **n:1** - Die Mailbox wird dann auf einen Port zurückgeführt
   - **1:n** - Broadcasting
   - **n:m** Mailbox

   Abbildung 7.11: Indirekte Adressierung n:1

   Die Zuordnung von Prozessen zu Mailboxen kann dabei statisch oder auch dynamisch sein.
   Ports sind in der Regel statisch den Prozessen zugeordnet.
   Für dynamische Zuordnungen können 2 Primitive verwendet werden: connect und disconnect.

Ablauf

PROGRAM mutualexclusion;
CONST n = ...; (* Anzahl der Prozesse *)
PROCEDURE P(i:INTEGER);
VAR msg: message;
BEGIN
  REPEAT
    receive (mutex , msg); (* Erhalt einer Nachricht ist Bedingung für *)
    (* Eintritt in kritisches Bereich *)
    <critical section >;
    send (mutex , msg); (* nun ist die Mailbox leer *)
    (* Rückgabe der Nachricht an die Mailbox *)
    <remainder> (* Damit ist der kritische Bereich *)
    (* wieder freigegeben *)
  FOREVER
  END;
BEGIN
  createmailbox (mutex );
  send (mutex , NULL);
  PARBEGIN
    P(1);
    P(2);
    ...
    P(n);
  PAREND;
END.

Bei diesem Verfahren muß folgendes beachtet werden:

- Rufen mehrere Prozesse quasi gleichzeitig die receive-Operation auf, so wird die Nachricht an nur einen Prozess ausgeliefert, alle anderen Prozesse werden blockiert, bis sie

Abbildung 7.12: Indirekte Adressierung n:m
eine Nachricht erhalten können.

– Ist die Mailbox leer, so werden alle Prozesse blockiert.

– Wird eine Nachricht in der Mailbox verfügbar, so wird nur ein Prozess aktiviert und ihm die Nachricht übertragen.

Abschließend sollen einige Betriebssysteme und deren Möglichkeiten zur Prozesskoordinati-
on und -kommunikation betrachtet werden: (Unterstrichen: Vorwiegend genutzte Möglich-
heiten)

– UNIX

  • Pipes (circular buffer to communicate on the E/A-problem basis, d.h. FIFO-Queue, in die ein Prozess schreibt und aus der ein anderer liest.)
  • Messages
  • Shared Memory
  • Semaphore
  • Signale

– Solaris Threads

  • Mutex (Mutual exclusion locks)
  • Semaphore
  • Multiple reader, single writer locks
  • Condition variables (warten, bis Bedingung wahr ist)

– Windows NT

  • File change notification
  • Mutex
  • Semaphore
  • Events
  • Waitable Timer
Teil IV

Ressourcenverwaltung
8

Speicher

- Verwaltung des Speichers
- Aufteilung des Speichers
- Paging und Segmentierung
- Virtueller Speicher

Inhaltsangabe

8.1 Speicherverwaltung .................................................. 158
8.2 Speicherpartitionierung ............................................... 159
  8.2.1 Feste Partitionierung ............................................ 159
  8.2.2 Dynamische Partitionierung ................................. 161
  8.2.3 Buddy-Systeme .................................................. 162
8.3 Virtueller Speicher ................................................... 164
  8.3.1 Prinzip der Speicherverwaltung ................................ 165
  8.3.2 Datentransport zwischen Hintergrund- und Arbeitsspeicher 166
  8.3.3 Abbildung virtueller auf reale Adressen .................... 167
8.4 Paging ................................................................. 169
  8.4.1 Paging-Strategien ............................................... 169
  8.4.2 Seitenaustauschalgorithmen .................................. 170
  8.4.3 Minimierung von Seitenfehlern ............................. 175
  8.4.4 Working Set Strategie ........................................ 178
8.5 Segmentierungsstrategien ......................................... 181

157
Kapitel 8. Speicher


Nun: Befehle, die vom Prozessor verarbeitet werden, müssen im Hauptspeicher vorliegen. Da dieser jedoch relativ klein ist, können Daten/Programme dort nur temporär vorhanden sein, und Daten müssen zwischen Haupt- und Hintergrundspeicher verschoben werden.


8.1 Speicherverwaltung

Generell werden fünf Anforderungen an die Speicherverwaltung gestellt:

1. **Relocation:** (Wiederfinden)
   - Der Hauptspeicher eines Multiprogrammingsystems wird von den verschiedenen Prozessen gemeinsam genutzt. Hierfür ist eine Verwaltungsstruktur notwendig, z.B. in Form von Listen.
   - Programme werden zum Teil ausgelagert, und nur das Prozess Image der aktiven Prozesse muß im Hauptspeicher verfügbar sein.

   → die Prozessorhardware und die Betriebssystem-Software müssen in der Lage sein, Referenzen im Programmcode in aktuelle physische Adressen umzuwandeln.

2. **Protection:** (Schutz)
   - jeder Prozess muß gegen ungewollte Einmischungen oder Störungen anderer Prozesse geschützt werden, egal ob diese Eingriffe beabsichtigt oder unbeabsichtigt sind.

   → fremde Prozesse sollten nicht in der Lage sein, die gespeicherten Informationen eines anderen Prozesses zu lesen oder zu modifizieren, wenn sie dafür keine Erlaubnis haben.

3. **Sharing:** (Teilen/Aufteilen)
   - Die Speicherverwaltung muss verschiedenen Prozessen den Zugriff auf einen gemeinsam genutzten Speicherbereich ermöglichen. Falls mehrere Prozesse das gleiche Programm ausführen, so ist es z.B. vorteilhaft, dass jeder Prozess auf dieselbe Version des Programms zugreift anstatt eine eigene Kopie zu nutzen.
4. **Logical Organization:** (Logische Organisation)
   - Der Hauptspeicher (und in der Regel auch der Sekundärspeicher) ist immer als linearer (1-dimensionaler) Adressraum organisiert, der aus einer Folge von Bits (Bytes oder Wörtern) besteht.
   - Programme werden jedoch modular geschrieben, d.h. diese Darstellung im Speicher entspricht nicht der Art und Weise, in der Programme geschrieben werden.

   → Eine modulare Organisation ist besser zu verwalten und sollte von der Speicherverwaltung unterstützt werden.

5. **Physical Organization:** (Physische Organisation)
   - Umfasst die Einteilung in Hauptspeicher (schneller Zugriff, hohe Kosten) und Hintergrundspeicher (langsamer und billiger).
     Nur der Hintergrundspeicher ist in der Lage, Daten permanent zu speichern. Die Daten werden jedoch vom Prozessor im Hauptspeicher benötigt.

   → Die eigentliche Aufgabe der Speicherverwaltung ist der Transport von Daten zwischen Haupt- und dem Hintergrundspeicher.

### 8.2 Speicherpartitionierung

Die grundlegende Aufgabe der Speicherverwaltung besteht also darin, Programme zwecks Ausführung durch den Prozessor in den Hauptspeicher zu bringen.
In quasi allen modernen Multiprogrammingsystemen wird dabei das Prinzip des virtuellen Speichers verwendet.
Zunächst sollen jedoch Speichertechniken ohne virtuellen Speicher betrachtet werden.

#### 8.2.1 Feste Partitionierung

- Dabei unterscheidet man gleichgroße und unterschiedlich große Partitionen des Hauptspeichers, wie in Abbildung 8.1 zu sehen.
Abbildung 8.1: Beispiel für feste Partitionierung

– In die jeweilige Partition kann jeweils ein Prozess geladen werden, dessen Größe kleiner oder gleich der Partitionsgröße ist.

– Sind alle Partitionen voll und kein Prozess ist im State Running oder Ready, so kann das Betriebssystem Prozesse auslagern, dadurch Partitionen freibekommen und dann andere Prozesse nachladen.

Mögliche Probleme:

– Ein Programm ist zu groß, um in eine Partition hineinzupassen. Dann ist es die Aufgabe des Programmierers, das Programm in Teilprogramme zu zerlegen (oder sich einen anderen Hauptspeicher zuzulegen: größer oder anders partitioniert).

– Die Partitionierung ist extrem ineffizient, d.h. viele kleine Module belegen die Partitionen, so dass viel Platz verschwendet wird.
  → Diesen Sachverhalt bezeichnet man als interne Fragmentierung.

Beide Probleme können mittels unterschiedlich großen Partitionen des Hauptspeichers verringert, jedoch nicht beseitigt werden.

Allerdings ist bei verschiedenen großen Partitionen zu beachten, dass immer die jeweils kleinste, freie und hinreichend große Partition verwendet wird.
8.2.2 Dynamische Partitionierung

Bei der dynamischen Partitionierung ist die Anzahl sowie die Größe der Partitionen variabel. Alle Partitionen werden sequentiell beschrieben.


Abbildung 8.3 veranschaulicht das Prinzip für 1 MByte Hauptspeicher.

Geht man von einem freien Speicher aus, so funktioniert dieser Algorithmus recht gut - der Speicher wird jedoch mit der Zeit zunehmend fragmentiert.

Mögliche Probleme:

- Neue Speicheranforderungen entsprechen in der Regel in ihrer Größe nicht den bestehenden Lücken, so dass Freiräume zwischen den Segmenten entstehen. → Dieses Phänomen bezeichnet man als externe Fragmentierung.

Dagegen hilft die Komprimierung (Compaction): Von Zeit zu Zeit werden die Prozesse zu einem Block zusammengeschoben.

Um innerhalb einer Speicherstruktur, in der einzelne Blöcke abgespeichert sind, ein optimales Füllen der freien Speicherbereiche zu erhalten, gibt es verschiedene Strategien:

**Best-Fit:** suche die kleinste Lücke, in die der Prozess paßt.

**First-Fit:** nimm vorne die erste passende Lücke.

**Next-Fit:** nimm ab letzter Belegung die nächste passende Lücke.
### 8.2.3 Buddy-Systeme

Eine feste Partitionierung hat den Nachteil, dass die Anzahl aktiver Prozesse begrenzt ist und der Speicherplatz ineffizient genutzt wird. Eine dynamische Partitionierung ist dagegen komplex zu verwalten und hinterläßt unnutzbare Freiräume.

Ein interessanter Kompromiß sind die sogenannten Buddy-Systeme.

Ein Speicher der Größe \(2^N\) wird in Blöcke der Größe \(2^K\) eingeteilt. Die kleinste Speichergröße wird auf \(2^L\), die größte auf \(2^U\) festgelegt. Für unterschiedliche \(K\) muss in jedem Fall gelten: \(L \leq K \leq U\).

Im allgemeinen gilt: \(U = N\).

(\(\rightarrow L/U = \text{Lower/Upper Bound (Grenze).} L\) ist geeignet zu wählen, so dass hinreichend kleine Buddies möglich sind, deren Verwaltung noch sinnvoll ist.)

**Anfangszustand:** Der verfügbare Speicherplatz wird als einzelner Block der Größe \(2^U\) betrachtet.

**Algorithmus:** Kommt nun eine Speicheranforderung \(S\), so wird die Bedingung \(2^{U-1} < S \leq 2^U\) geprüft und in diesem Fall der gesamte Speicher der Anforderung zugewiesen.

Ist \(S > 2^U\), so kann die Anforderung nicht erfüllt werden.

In der Regel wird jedoch \(S \leq 2^U\) gelten.

\(\Rightarrow\) Der Speicherblock wird in zwei Buddies der Größe \(2^{U-1}\) aufgespalten und für den ersten Block die Bedingung \(2^{U-2} < S \leq 2^{U-1}\) geprüft. Und so weiter.

Das Buddy-System verwaltet pro Zweierpotenz eine Liste der freien Speicherplätze, d.h. der Lücken der Größen \(2^i\).
Durch das Splitten wird eine Lücke von der entsprechenden Liste entfernt. Von den beiden (halb so großen) entstehenden Gliedern wird entweder eines besetzt oder weitergesplittet u.s.w.

Abbildung 8.4: Beispiel für Buddy-Systeme

Umgekehrt: Ist eine gewünschte Liste leer, so wird die nächsthöhere freie Liste entsprechend gesplittet.
Die Adressierung geschieht dabei auf folgende Art und Weise:
Wir gehen von einem Speicherbereich der Größe $2^U$ aus, geben ihm die Adresse
\[
\begin{array}{c}
\times 0 0 0 \ldots 0 \\
\uparrow \text{U Nullen}
\end{array}
\]

wobei x eine beliebige Dualzahl darstellt (z.B. $x = 0$). Die Blöcke der Größe $2^{U-1}$ erhalten die Adressen
\[
\begin{array}{c}
x 0 0 0 \ldots 0 \\
\uparrow \text{U–1 Nullen}
\end{array}
\]

und
\[
\begin{array}{c}
x 1 0 0 0 \ldots 0 \\
\uparrow \text{U–1 Nullen}
\end{array}
\]

dann ergibt sich für die Blöcke der Größen $2^{U-2}$ die Adressierung analog (Abbildung 8.5):

Abbildung 8.5: Adressierung in Buddy-Systemen
In der Praxis könnte für einen Speicher von 1 Megabyte zunächst eine Anforderung von 100k kommen, dann 250k, dann 50k:

**Abbildung 8.6: Fragmentierung bei Buddy-Systemen**

**Merk**: Immer Listen aktualisieren! Bei Freigabe von Buddies muss geprüft werden, ob der benachbarte Buddy auch frei ist. Falls ja, werden beide freien Buddies zu einem nächst grösseren zusammengefasst.

→Eintrag in Liste, erneute Zusammenlegung prüfen etc.

**Nachteil**: →interne und externe Fragmentierung

**Anwendung**: Eine Variante dieser Speicherform wird im UNIX Kernel verwendet.

Eine Alternative zu den vorgestellten *Buddy-Systemen* sind die *gewichteten Buddy-Systeme*.

**Gewichtete Buddy-Systeme**

Um bei Speicherplatzanforderungen ungleich $2^n$ weniger Platz zu verschenken, d.h. um die interne Fragmentierung möglichst gering zu halten, werden *gewichtete Buddy-Systeme* verwandt.

**Prinzip**: Ein Block der Größe $2^{n+2}$ wird im Verhältnis 1:3 geteilt. Das ergibt Blöcke der Größe $2'^i$ und $3 \cdot 2'^i$. Der *Buddy* der Größe $3 \cdot 2'^i$ wird im Verhältnis 2:1 geteilt, d.h. in Blöcke der Größe $2^{r+1}$ und $2'^i$ wie man auch in Abbildung 8.7 sehen kann.

**Vorteil**: Man erhält flexible Buddygrößen und damit weniger Verschwendung von Platz.

**Aber**: Der Verwaltungsaufwand für die einzelnen Buddies, insbesondere zur Adressberechnung wird deutlich größer.

**8.3 Virtueller Speicher**

Adressräume stehen in sehr engem Zusammenhang mit der Speicherverwaltung. Wir wollen nun eine statische Partitionierung betrachten und teilen dazu den Hintergrundspeicher und

8.3.1 Prinzip der Speicherverwaltung

Definition 8.1 (Physischer Adressraum). Die Menge der im Arbeitsspeicher physisch vorhandenen Speicherplätze bildet den realen oder physischen Adressraum \( P \).

Im physischen Adressraum werden die Speicherzellen, die eine einheitliche Größe besitzen (in der MI 8 Bit), linear angeordnet, d.h. jeder Zelle ist eindeutig eine Adresse zugeordnet, die mit 0 beginnen.

Definition 8.2 (Logischer Adressraum). Dem physischen oder realen Adressraum steht ein virtueller oder logischer Adressraum \( L \) gegenüber.

Der logische Adressraum wird benötigt, falls die Kapazität des physischen Arbeitsspeichers nicht ausreicht, um alle Daten zu speichern. Der so entstehende Kapazitätsengpaß soll durch die zusätzliche Nutzung des Hintergrundspeichers (in der Regel der Festplatten) beseitigt werden.

Aufgabe der Speicherverwaltung  Die Aufgabe der Speicherverwaltung besteht in der Zuordnung von logischen zu physischen Adressen, d.h. im Auffinden einer geeigneten Abbildung \( L \mapsto P \) ("logischer Adressraum" wird abgebildet auf "physischen Adressraum"). Für die Konstruktion einer solchen Abbildung sind die beiden folgenden Fälle zu unterscheiden:

1. Ist \( |L| \leq |P| \), also der logische Adressraum kleiner als der physische, so kann der logische Adressraum problemlos in den physischen aufgenommen werden. Bei modernen Rechnern ist diese Situation in der Regel gegeben, wenn nur ein einziger Benutzer mit einem einzelnen Programm betrachtet wird.
Kapitel 8. Speicher

2. Der Fall $|L| > |P|$ ist jedoch der klassische Fall. Dies resultiert u.a. aus der weit verbreiteten Wortlänge von 32 Bit, mit der sich $2^{32}$ Adressen ansprechen lassen. Nimmt man an, dass jede Adresse ein Speicherwort von 4 Byte adressiert, so lassen sich $2^{32} \cdot 4 = 2^{34}$ Byte = 16 Gigabyte in $L$ ansprechen.

Weiterhin fordern folgende Entwicklungen die Verwendung von virtuellem Speicher: An Rechnern, die Multiprogramming-fähig sind, können mehrere Nutzer gleichzeitig arbeiten und Teile des Speichers durch Systemsoftware belegt werden.

Offensichtlich wird also der Arbeitsspeicher im allgemeinen nicht ausreichen. Gleichzeitig ist ausreichend Hintergrundspeicher vorhanden, um große Datenmengen aufzunehmen.

Aus diesen beiden Beobachtungen heraus wurde die Verwendung von virtuellem Speicher notwendig.

**Bemerkung 8.1.** In der MI ist $|L| > |P|$. Mit einer der $2^{32}$ Adressen läßt sich nur je 1 Byte ansprechen, also hat L eine Größe von 4 Gigabyte. Die MI besitzt allerdings nur einen “realen” Arbeitsspeicher $P$ von 1 Gigabyte, es muß also eine Methode gefunden werden, den logischen auf den realen Adressraum abzubilden.

8.3.2 Datentransport zwischen Hintergrund- und Arbeitsspeicher

Im allgemeinen ist der virtuelle Speicher $L$ (deutlich) größer als der reale Speicher $P$. Daher ist es nötig, zu definieren, wie der Datentransport zwischen Hintergrund- und Arbeitsspeicher vonstatten geht und welche Teile des virtuellen Adressraumes auf Hintergrundspeicher auszulagern sind.

**Definition 8.3** (Seite). Zum Datentransport werden Transporteinheiten fester Länge definiert, die als pages oder Seiten bezeichnet werden. Der logische Speicher wird also in Seiten aufgeteilt, die einzeln zwischen Hintergrund- und Arbeitsspeicher transferiert werden können.

In der Regel werden für eine Seite Größen zwischen 2 und 8 Kilobyte verwendet, in der MI werden jedoch nur 512 Byte verwendet.

**Definition 8.4.** Als Gegenstück zu den pages des logischen Adressraumes wird der reale Adressraum (d.h. der Hauptspeicher) in Frames (Seitenrahmen oder Kacheln) eingeteilt. Frames haben immer die gleiche Größe wie die pages (vgl. Abbildung [8.8]).
Die Größe einer Seite sollte weder “zu klein” (zuviel Transport- und Verwaltungsaufwand) noch “zu groß” (zuviel Speicherverschwendung, interne Fragmentierung) sein.

8.3.3 Abbildung virtueller auf reale Adressen

MI-Maschinenadressen

Die Aufteilung einer MI-Maschinenadresse ergibt sich aus Abbildung 8.9.

Eine Maschinenadresse muß folgende Informationen liefern:

1. die Nummer des Seitenrahmens und
2. innerhalb einer Seite muß eine Adresse relativ zum Anfang des Seitenrahmens angegeben werden (Offset).

Beispiel 8.1. Bei der MI hat eine Seite $2^9 = 512$ Byte, also benötigen wir gerade 9 Bit für die Adressierung innerhalb einer Seite. Für die Adressierung der Seitenrahmen hingegen benötigen wir 21 Bit, da es $1\ \text{GB} = 2^{30} ; 2^9 = 2^{21}$ (Größe des Arbeitsspeichers durch Größe einer Seite) Seitenrahmen gibt.
Beispiel 8.2. Der HGS hat viermal so viele Speicheradressen wie der HS. Daher sind die virtuellen Adressen um 2 Bit länger als die realen Adressen.  
⇒ Mit k zusätzlichen Bits können \(2^k\) mal so viele Speicherzellen adressiert werden.

Virtuelle Adressen (MI-Programmadressen)

Wiederum sind die Adressen von 32 Bit Länge. Die Aufteilung ist ähnlich zu der von MI-Maschinenadressen, jedoch wird folgende Codierung für \(b_0\) und \(b_1\) ergänzt:

\[
\begin{array}{c|c|c}
 b_0 & b_1 & \text{Bereich} \\
 0 & 0 & P0 \\
 0 & 1 & P1 \\
 1 & 0 & P2 \\
 1 & 1 & P3 \\
\end{array}
\]

Damit ergibt sich die Aufteilung in 8.10.

Abbildung 8.10: MI-Programmadresse


- sind die Seiten im Hintergrundspeicher durchnumeriert und
- zu jeder Seite wird angegeben, ob sie im Hauptspeicher verfügbar ist und falls ja, in welchem Seitenrahmen und mit welcher Adresse.


Wird eine virtuelle Adresse angefordert, so werden die Bits für die Seitennummer (MI: \(b_2\) bis \(b_{22}\)) in der Tabelle gesucht. Falls die Seite sich im Hauptspeicher befindet, so werden diese Bits durch die “realen Bits” ersetzt, d.h. die Seitennummer der virtuellen Adresse wird durch die Seitenrahmennummer der realen Adresse ersetzt.
Tabelle 8.1: Speichertabelle mit angenommener Hauptspeicher-Adresse 0A32 für das Frame Nr. 3; alle anderen Hauptspeicheradressen lassen sich automatisch berechnen. Hierbei stehe HGS für Hintergrundspeicher und HS für Hauptspeicher.

### 8.4 Paging


#### 8.4.1 Paging-Strategien

Unter verschiedenen Kriterien sollen drei mögliche Strategien des Pagings betrachtet werden:

1. **Demand Paging:** Fehlt eine Seite im Hauptspeicher (d.h. liegt ein Seitenfehler vor), so wird diese on demand aus dem HGS nachgeladen.

2. **Demand Prepaging:** Fehlt eine Seite im Hauptspeicher, werden gleichzeitig *mehrere* Seiten geladen und verdrängt. Dadurch wird also die Zahl der Zugriffe auf den Hintergrundspeicher geringer gehalten als beim ersten Verfahren.

3. **Look-Ahead-Paging:** Nicht nur bei Seitenfehlern, sondern auch nach anderen (näher zu definierenden) Kriterien können Nachladeoperationen stattfinden.

8.4.2 Seitenaustauschalgorithmen

Dabei gibt es verschiedene Vorschriften (Policies), die das Betriebssystem für den Virtuellen Speicher macht. Im Folgenden sollen einige Beispiele solcher Policies betrachtet werden.

**Resident Set Management Policy**

Macht Vorschriften, ob sich eine feste oder eine variable Anzahl von Seiten eines bestimmten Prozesses während der Abarbeitung ständig im Hauptspeicher befinden sollte. (Zur Wiederholung: Einige Informationen müssen als residente Menge des Prozesses ständig im Hauptspeicher verfügbar sein.) Weiterhin kann mit dieser Policy die Menge der für einen Austausch infrage kommenden Seiten eingeschränkt werden, diesen Sachverhalt nennt man Replacement Scope.

**Fetch Policy**

Bestimmt im wesentlichen, ob eine neue Seite nur dann nachgeladen wird, wenn ein Seitenfehler vorliegt (Demand Paging) oder ob Seiten auch dann nachgeladen werden dürfen, wenn kein Seitenfehler vorliegt (Prepaging).

Der letztergenannte Sachverhalt könnte dann sinnvoll sein, wenn - z.B. im Falle eines Seitenfehlers - auch mehrere Seiten auf einmal als Block gut transportiert werden könnten. Prepaging wird von den meisten Betriebssystemen umgesetzt.

**Placement Policy**


**Replacement Policy**

Situation: Die einem Prozess zugeordneten Frames sind alle belegt, und es wird eine weitere Page im HS benötigt. Bestimmt die Seite bzw. den Block, der ersetzt werden soll, wenn eine neue Seite geladen werden muß, d.h. wenn ein Seitenfehler auftritt. In den vergangenen 20 Jahren wurde im Bereich der Betriebssysteme dieses Thema wahrscheinlich am meisten betrachtet.

Es existieren nun verschiedene, grundlegende Ersetzungsstrategien.

1. **OPT (Optimalstrategie):** Lagert jeweils diejenige Seite mit dem größten Vorwärtsabstand aus, also diejenige Seite, die am längsten nicht mehr gebraucht wird. Dieses Verfahren verursacht theoretisch die wenigsten Seitenfehler, man sagt: Die geringsten Kosten. Praktisch ist es jedoch nur in Ausnahmefällen realisierbar. Als Approximation kann man den Vorwärtsabstand aufgrund bisheriger Beobachtungen schätzen.

Wenn man dieses Verfahren im Nachhinein auf eine Abarbeitung von Prozessen anwendet, erhält man ein Maß, um die Güte anderer Verfahren bewerten zu können.
Z.B. bei einer Anforderung der Seiten:

\[2 3 2 1^* 5^* 2^* 4^* 5 3^* 2 5 2\]

wäre bei einer Kapazität von 3 Seiten im Speicher folgende Belegung denkbar:

<table>
<thead>
<tr>
<th></th>
<th>1^*</th>
<th>5^*</th>
<th>2^*</th>
<th>4^*</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>FR2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2^*</td>
<td>4</td>
</tr>
<tr>
<td>FR3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

2 ist die Seite mit dem grössten Vorwärtsabstand, d.h. diese Seite wird am längsten nicht mehr gebraucht → ersetzt. (r steht für raus!)

* bezeichnet dabei das Auftreten eines Seitenfehlers, d.h. bei einer optimalen Strategie treten nur 3 Seitenfehler auf. Streng genommen rechnet man zusätzliche 3 Seitenfehler für die Erstbelegungen und erhält somit 6 Seitenfehler.

**FAZIT:**

Diese Strategie ist wie ihr Name optimal und sinnvoll v.a. als Benchmarkingstrategie. Aber: In der Praxis kann der Referencestring erst im Nachhinein aufgestellt werden, da zu einem bestimmten Zeitpunkt immer nur die als nächstes zu bearbeitende Seite sichtbar ist.

2. **First In First Out (FIFO-Strategie):** Ordnet Seiten nach dem Alter, im Bedarfsfall wird die älteste Seite, d.h. die Seite, die sich bereits am längsten im Hauptspeicher befindet, ausgelagert.

**Beispiel:**

\[2 3 2 1^* 5^* 2^* 4^* 5 3^* 2 5 2\]

<table>
<thead>
<tr>
<th></th>
<th>1^*</th>
<th>5^*</th>
<th>2^*</th>
<th>4^*</th>
<th>3^*</th>
<th>5^*</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>FR2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FR3</td>
<td>1^*</td>
<td>5</td>
<td>3</td>
<td>5^*</td>
<td>2</td>
<td>2^*</td>
<td>4</td>
</tr>
</tbody>
</table>

FR1 |     | 1   | 5   | 2   | 4   | 3   | 3   | 5   | 2 |
| FR2 | 3   | 3   | 3   | 1   | 5   | 2   | 2   | 4   | 3   | 5 |
| FR3 | 2   | 2   | 2   | 2^* | 3^* | 1^* | 5   | 5^* | 2   | 2^* | 4^* | 3 |
D.h. es treten mit den Erstbelegungen 9 \((3+6)\) Seitenfehler auf. Im Vergleich zur OPT-Strategie ist FIFO also eine relativ schlechte Strategie. Realisiert werden kann dieser Ansatz mit einem zirkulierenden Zeiger. Von der Implementierung her ist dies der einfachste - aber leider nicht der beste - Algorithmus.

3. **Least Recently Used (LRU-Strategie):** Verdrängt Seiten nach dem Alter, d.h. die am längsten nicht mehr benutzte Seite wird aus dem Hauptspeicher ausgelagert. Die zugrundeliegende Idee besteht in der Lokalität, d.h. die Seite, die gerade gebraucht wurde, wird wahrscheinlich wieder gebraucht.
Diese Strategie kommt der Optimalstrategie relativ nahe.

**Beispiel:**

\[
\begin{array}{cccccc}
2 & 3 & 2 & 1* & 5 & 2* \\
1* & 5* & 4* & 3* & 2 & 2 \\
3 & 1 & 5 & 4 & 3 & 5 \\
2 & 2 & 2 & 5 & 5 & \\
\end{array}
\]

oder:

\[
\begin{array}{cccccc}
3 & 2 & 1* & 5 & 2* \\
2 & 1 & 5 & 2 & 4 & 5 \\
3 & 3 & 2 & 1 & 5 & 2 \\
4 & 5 & 3 & 2 & 5 & 3 \\
\end{array}
\]

D.h. in diesem Fall treten 7 \((3+4)\) Seitenfehler auf.

4. **LFU (Least Frequently Used):** Bei diesem Verfahren soll diejenige Seite mit niedrigster Nutzungshäufigkeit ausgetauscht werden, so dass die am häufigsten genutzten Seiten im HS bleiben. Dazu muß jedoch für jeden Frame die Anzahl an Zugriffen verwaltet werden. Dieses Verfahren wird im wesentlichen in drei Varianten verwendet, die sich nach der Dauer der Zählung der Seitenzugriffe unterscheiden:

Die Seitenzugriffe können gezählt werden
(a) seit Laden der Seite
(b) innerhalb der letzten \(h\) Zugriffe oder
(c) seit dem letzten Seitenfehler.
Diese Variante ist nur bei starker Lokalität geeignet, d.h., wenn zwischen 2 Seitenfehlern ein hinreichend großer Abstand liegt. Dies ist etwa der Fall, wenn mehr Seiten genutzt werden, als Frames in den Hauptspeicher passen.

\[1\text{In der Literatur wird auch häufig der Begriff Last Frequently Used verwendet}\]
Bei diesem Verfahren ist jedoch wie auch bei LRU, bei jedem Speicherzugriff ein zusätzlicher Verwaltungsaufwand notwendig. Der Aufwand ist in der Praxis so groß, dass das Verfahren kaum genutzt wird.

5. **Climb (Aufstieg bei Bewährung):** Hierbei steigt eine Seite bei jedem Aufruf eine Position höher, wenn sie bereits im Speicher vorhanden ist, d.h. sie tauscht ihre Position mit der vor ihr stehenden Seite. **Beispiel:**

\[
\begin{array}{c|c|c|c|c|c|c|c}
2 & 3 & 2 & 1^* & 5^* & 2^* & 4^* & 5 & 3 & 2 & 5 & 2 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
2 & 3 & 3 & 1 & 5^* & 5 & 4^* & 5^* & 5 & 5 & 3 & 3 \\
\end{array}
\]

Ist eine neue Seite nicht im Speicher vorhanden, so wird die unterste Seite ausgelagert und die neue Seite dorthin geladen.
Es treten wie bei der OPT-Strategie 6 (3+3) Seitenfehler auf.

6. **Clock-Strategie:** Im Folgenden soll die Verwaltung von Seiten in einer zyklischen Liste betrachtet werden, die die Form einer Uhr darstellt. Der Zeiger der Uhr zeigt dabei auf die älteste Seite.
Ein zusätzliches Bit pro Seite wird auch als Use Bit bezeichnet. Wird eine Seite das erste mal in ein Frame des Hauptspeichers geladen, so wird dieses Use Bit auf 1 gesetzt. Bei einer nachfolgenden Referenzierung erfolgt ebenfalls ein Setzen auf 1.
Ist es nötig, eine Seite zu ersetzen, so wird der Speicher durchsucht, um ein Frame mit einem Use Bit = 0 zu finden. Trifft er dabei auf ein Frame mit einem Use Bit = 1, so wird dieses auf den Wert 0 zurückgesetzt. Haben alle Frames Use Bits = 0, so wird das erste Frame, auf das der Zeiger zeigt, für die Ersetzung gewählt. Haben andererseits alle Frames ein Use Bit = 1, so durchläuft der Zeiger einen kompletten Zyklus durch den Speicher, setzt alle Use Bits auf 0 und stoppt dann an der ursprünglichen Stelle. Das Frame wird genutzt, um eine Seite zu ersetzen, woraufhin der Zeiger um eins erhöht wird.
Dieser Algorithmus ähnelt FIFO, allerdings mit der Ausnahme, dass Frames mit einem Use Bit = 1 durch den Algorithmus übersprungen werden.
Kapitel 8. Speicher

RS: 2 3 2 1 5 2 4 5 3 2 5 2

2:   U-Bit
     1
     2 1

3:    3 1
      2 1

2:   3 1
     2 1

1: 1 1
   3 1

Position des Zeigers ändert sich gdw. eine Seite ersetzt werden muss

5:

Zeiger kreist
3 0
2 0

1 0
3 0
5 1

2: 1 0
3 0
5 1

4: 2 1
2 1
5 1

Beim Überstreichen einer Seite wird das Use-Bit von 1 auf 0 gesetzt. Der Zeiger kreist, bis er wieder an der ersten 0 angekommen ist.

2: 4 0
2 0
5 0

3: 4 0
2 0
5 0

4: 2 1
2 1
5 1

Zeiger auf älteste Seite

5:

keine Seite ersetzt heißt keine Änderung der Zeigerposition

2:

4 0
2 0
3 1

5:

5 1
2 0
3 1

nur U-Bit verändert
Zeiger bleibt, weil keine Ersetzung nötig (nur Referenzierung)

Abbildung 8.11: Beispiel zur Clock-Strategie
Beispiel:

\[
\begin{array}{ccccccccccc}
2 & 3 & 2 & 1^* & 5^* & 2^* & 4^* & 5^* & 3^* & 2^* & 5^*\\
3^* & 3^* & 3^* & 3^* & 2^* & 2^* & 2^* & 2^* & 2^* & 2^* & 2^*\\
2^* & 2^* & 2^* & 5^* & 5^* & 5^* & 5^* & 3^* & 3^* & 3^* & 3^*
\end{array}
\]

Diese Strategie führt zu 5 Seitenfehlern (Seitenfehler bis alle Frames belegt sind, werden hier nicht berücksichtigt).

8.4.3 Minimierung von Seitenfehlern

Auf den ersten Blick könnte man annehmen, dass die Anzahl der Seitenfehler sinkt, wenn man die Anzahl der Rahmen im Hauptspeicher bei gleicher Rahmengröße (also den vorhandenen physischen Speicher) erhöht.

Im Jahr 1969 wurde jedoch ein Gegenbeispiel gefunden, das bei der Verwendung von FIFO als Seitenwechselalgorithmus bei vier Rahmen mehr Seitenfehler verursacht als bei drei (Belady's Anomalie).


<table>
<thead>
<tr>
<th>Reference String</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jüngste Seite</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Alteste Seite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Page Fault</td>
<td>(P)</td>
<td>(P)</td>
<td>(P)</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference String</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jüngste Seite</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Alteste Seite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Fault</td>
<td>(P)</td>
<td>(P)</td>
<td>(P)</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tabelle 8.2: Belady's Anomalie


Definition 8.5 (Reference String). Jeder Prozess erzeugt eine eindeutige Folge von Speicherzugriffen während seiner Ausführung, die in eine eindeutige Folge von Zugriffen auf
bestimmte Speicherseiten umgesetzt werden kann. Diese Folge wird als **Reference String** bezeichnet.

Ein Paging-System kann eindeutig durch die folgenden drei Eigenschaften beschrieben werden:

1. den Reference String des gerade ausgeführten Prozesses,
2. den Seitenaustauschalgorithmus und
3. die Anzahl der Seitenrahmen \( m \), die im Speicher gehalten werden.

Mit Hilfe dieser Definitionen kann man eine abstrakte Maschine wie folgt definieren: Sei \( M \) ein Feld (eine Menge), das den Status des Speichers beschreibe, dann ist die Anzahl \( n \) der Elemente von \( M \) genauso groß wie die Anzahl der virtuellen Seiten des zugeordneten Prozesses. Das Feld \( M \) teilt sich in zwei Bereiche, der obere enthält \( m \) Einträge, nämlich die Seiten, die sich gerade im Speicher befinden, der untere enthält \( n - m \) Seiten, nämlich diejenigen, die bereits einmal angesprochen wurden, jedoch wieder ausgelagert wurden und sich so nicht im Speicher befinden.

Zu Beginn ist \( M \) also leer, da noch keine Seiten angesprochen wurden und sich auch keine im Speicher befinden.

**Beispiel 8.4** (Operationsweise). **Angenommen als Seitenaustauschalgorithmus wird LRU verwendet und der virtuelle Adressraum hat 8 Seiten, während nur vier Seitenrahmen im physischen Speicher zur Verfügung stehen. Der zugehörige Prozess greift auf die Seiten in der folgenden Reihenfolge zu (Reference String):**

\[
021354637473355311172341
\]

Dann ergibt sich für die Ausführung der Maschine das Bild in Tabelle 8.3.

Wenn also auf eine Seite zugegriffen wird, wird sie an die Spitze der Einträge von \( M \) verschoben. War sie bereits in \( M \) (also im Speicher), so werden alle Seiten, die bisher höher standen, um eins nach unten geschoben, war sie bisher nicht im Speicher, so wird die tiefste Seite aus \( M \) nach unten bewegt (ausgelagert).

Dieses Beispiel ist dank der Verwendung von LRU als Seitenaustauschalgorithmus zu einer Klasse von besonders interessanten Algorithmen gehörig:

Für alle Seitenaustauschalgorithmen dieser Klasse gilt

\[
M(m, r) \subset M(m + 1, r), \forall r \geq m,
\]

| RS | 0 | 2 | 1 | 3 | 5 | 6 | 4 | 7 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 1 | 7 | 2 | 4 | 3 | 4 |
|    | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 4 | 7 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 1 | 7 | 2 | 4 | 3 |
|    | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 4 | 7 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 1 | 7 | 2 | 4 | 3 |
|    | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 4 | 7 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 1 | 7 | 2 | 4 | 3 |

| PF | P | P | P | P | P | P | P | P | P |
| DS | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) |

Tabelle 8.3: Beispiel für Reference String
wobei M die eben erklärte abstrakte Maschine, m die Anzahl der Seitenrahmen und r die Indexposition im Reference String bezeichne, d.h. alle Seiten, die sich bei einem Speicher mit m Seitenrahmen nach r Speicherzugriffen im Speicher befinden, befinden sich auch bei einem Speicher mit m + 1 Seitenrahmen nach r Speicherzugriffen im Speicher. Dadurch ist gewährleistet, dass bei einer Vergrößerung des physischen Speichers (Erhöhung der Anzahl der Seitenrahmen), die Anzahl der Seitenfehler niemals zunehmen kann. Algorithmen in dieser Klasse werden als Stack-Algorithmen bezeichnet. FIFO wäre zum Beispiel kein Stack-Algorithmus.

**Beispiel 8.5.** Zum Zeitpunkt $t'$ habe unsere Speicherbelegung folgendes Aussehen:

```
4 | 5 | 3 | 1 | 2 | 0 | |
```

Diese Belegung entspricht Spalte 7 in $8.3$ Wir definieren $M := 4, 5, 3, 1$. Sei $n = 8$, da der Prozess die Seiten 0, 1, ..., 7 benutzt. Dabei bezeichnet der String $[4 | 5 | 3 | 1]$ die Frames des gerade belegten Hauptspeichers HS' mit $m = 4$ und der String $[2 | 0 | ]$ den belegten Hintergrundspeicher HGS' mit $n - m$ Seiten. Wir zählen die Seitenaufrufe durch und nutzen dabei den Index r. Wir erhalten:

- $r = 0$, $M = \emptyset$
- $r = 1$, $M = 0$
- $r = 2$, $M = 0, 2$
- ...
- $r = 6$, $M = 0, 2, 1, 3, 5, 4$

$Es gilt hier beispielsweise auch M(4, r) = M(5, r) und insbesondere M(4, r) \subseteq M(5, r)$.

$M$ hängt also von $r, m$, dem RS und der verwendeten Strategie (z.B. LRU) ab. Eine Erhöhung der Framezahl im HS bedingt, dass zu einem bestimmten Zeitpunkt die gleichen Seiten im HS geladen sind wie bei weniger Frames, und zusätzlich noch weitere Frames.

**Bemerkung 8.2.** Für $r \leq m$ gilt: $M(m, r) \subseteq M(m + 1, r)$ und insbesondere $M(m, r) = M(m + 1, r)$. Dieser Fall wird jedoch hier nicht betrachtet, da der Speicher nicht voll wäre und keine Seiten ausgetauscht werden müssten.

Im Folgenden sollen nur noch Stack-Algorithmen betrachtet werden.

**Distance String**

Die Distanz einer Seite bei einem gewissen Aufruf gibt an, wieviele Frames der HS für diesen Prozess bereitstellen muss, damit kein Seitenfehler auftritt. Bei Stack-Algorithmen ist es möglich und sinnvoll, den Reference String auf eine abstraktere Weise zu repräsentieren als durch die einfachen Seitennummern. Eine Seitenreferenz soll beschrieben werden durch die Entfernung der bisherigen Position der Seite im Stack von dessen Beginn, wobei für Seiten, auf die bisher noch nicht zugegriffen wurde, eine Distanz von $\infty$ angenommen wird.

Offensichtlich hängt der **Distance String** nicht nur vom Reference String, sondern auch vom Seitenau斯塔uschalgorithmus ab.
Vorhersage der Anzahl von Seitenfehlern

Die voranstehenden Definitionen sind notwendig, um eine genaue Analyse der zu erwartenden Seitenfehler für einen bestimmten Prozess bei Verwendung eines bestimmten Seitenau斯塔uschalgorithmus zu ermöglichen.

Das Ziel des Algorithmus ist es, bei einem Lauf über den Distance String die notwendigen Informationen zu sammeln, um vorhersagen zu können, wie viele Seitenfehler der Prozess bei \( n \) Seitenrahmen verursachen würde.

Man erhält – wie man leicht sieht – als Anzahl der Seitenfehler \( F_m \) bei einem vorgegebenen Distance String und \( m \) Seitenrahmen:

\[
F_m = \sum_{k=m+1}^{n} C_k + C_\infty
\]

\( C_k \) ist die Anzahl der Vorkommen von \( k \) im Distance String und \( C_\infty \) die Anzahl der Vorkommen von \( \infty \).

Für den Reference String aus Beispiel 8.4 auf Seite 176 ergibt sich dann die Werte aus Tabelle 8.4 bei Verwendung von LRU.

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & C_7 & C_\infty \\
F_0 & F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 \\
24 & 20 & 18 & 17 & 13 & 11 & 9 & 8 & 0 \\
\end{array}
\]

Tabelle 8.4: Vorhersage von Seitenfehlern aus 8.3

Berechnung:
\[
F_0 = C_1 + C_2 + \ldots + C_n + C_\infty \quad F_0 = 4 + 2 + 1 + 4 + 2 + 2 + 1 + 8 = 24
\]
\[
F_1 = C_2 + \ldots + C_n + C_\infty \quad F_1 = 2 + 1 + 4 + 2 + 2 + 1 + 8 = 20
\]
Oder:
\[
F_1 = F_0 - C_1 = 24 - 4 = 20
\]

Das bedeutet, dass man bei einem Frame und vorgegebenen Distance String 20 Seitenfehler erhält.

\[
F_2 = C_3 + \ldots + C_n + C_\infty \quad F_2 = 1 + 4 + 2 + 2 + 1 + 8 = 18
\]
Oder:
\[
F_2 = F_1 - C_2 = F_0 - C_1 - C_2 = 24 - 4 - 2 = 18
\]

Das bedeutet man erhält bei zwei Frames und vorgegebenen Distance String 18 Seitenfehler.

8.4.4 Working Set Strategie

Bisher haben wir Situationen betrachtet, in denen jedem Prozess eine konstante Anzahl von Frames zugeordnet wurde. Jetzt soll die Anzahl von Frames, die ein Prozess über seine Lebenszeit belegen kann, variieren können. Die Working Set Strategie ist im Vergleich zu den anderen Strategien dynamisch und variant - etwa im Gegensatz zur sehr stabilen LRU. Wir betrachten zunächst die sogenannte Lifetime-Funktion \( L(m) \). Diese gibt die mittlere Zeit zwischen aufeinanderfolgenden Seitenfehlern in Abhängigkeit von der zugeordneten Rahmenzahl \( m \) an.
Gewöhnlich steigt L mit wachsendem m monoton an. Und zwar in folgendem Sinne: Je mehr Frames einem Prozess im Hauptspeicher zur Verfügung stehen, desto seltener werden Seitenfehler, und desto mehr Zeit vergeht daher im Mittel zwischen 2 Seitenfehlern. 

L(M) ist natürlich von den jeweiligen Prozessen abhängig, in der Regel hat L(m) aber folgende Form:

![Diagramm der Lifetime-Function L(m)](Abbildung 8.12: Lifetime-Function L(m))

**Beispiel 8.6.** Bezeichne im Folgenden $\bar{t}$ den mittleren Zeitabstand zwischen 2 Seitenfehlern. Es sollen im Sekundentakt Seiten gebraucht werden. Man erhält folgenden Zusammenhang:

\[
\begin{align*}
F_0 &= 24 & t_0 &= 1\text{Sek.} \\
F_1 &= 20 & t_1 &= 24\text{Sek.}/20\text{Pagefaults} = 1,25\text{Sek.} \\
F_2 &= 18 & t_2 &= 24\text{Sek.}/18\text{Pagefaults} \approx 1,2551\text{Sek.} \\
& & t_3 &\approx 1,5\text{Sek.} \\
& & t_4 &\approx 1,8\text{Sek.} \\
& & t_5 &\approx 2,2\text{Sek.} \\
& & t_6 &\approx 2,6\text{Sek.} \\
& & t_7 &\approx 3,0\text{Sek.} \\
& & t_8 &\approx 3,0\text{Sek.} \\
& & t_9 &\approx t_{10} \approx \ldots \approx 3,0\text{Sek.}
\end{align*}
\]

D.h. bei wenigen zur Verfügung stehenden Frames kommt es quasi ständig zu Seitenfehlern, dann bringt jeder zusätzliche Frame eine deutliche Verbesserung, d.h. eine Verlängerung der Lifetime mit sich, bis irgendwann ein Sättigungsbereich auftritt.


Dem Ansatz liegen folgende Annahmen zugrunde:

- Die jüngste Vergangenheit ist eng mit der unmittelbaren Zukunft korreliert.
- Ein Prozess, der bislang viele aktive Seiten hatte, braucht diese mit einer gewissen Wahr- scheinlichkeit auch in Zukunft.
Über weitere Bereiche verhalten sich Prozesse “lokal”, d.h. Seiten, die erst vor kurzem referenziert wurden, sind in Zukunft wahrscheinlicher als solche, deren letzter Zugriff schon länger zurückliegt.

Diese Annahmen sind allerdings nicht allgemeingültig - z.B. wenn ein Prozess in einen völlig neuen Abschnitt eintritt, dann sind Ausnahmen durchaus denkbar.

**Definition 8.6.** Sei $r_1 r_2 ... r_T$ der Referenzstring eines Prozesses, d.h. die zugehörige Folge von Seitenzugriffen.
Dann ist der Working Set $W(t, h)$ dieses Prozesses zur Zeit $t$ unter einem Rückwärtsfenster der Größe $h$ definiert als:

$$W(t, h) := \bigcup_{i=t-h+1}^{t} r_i$$

D.h. $W(t, h)$ ist die Menge der Seiten, die bei den letzten $h$ Zugriffen mindestens einmal referenziert wurden.

 Folgendes Beispiel veranschaulicht das Working Set eines Reference String zum Zeitpunkt $x$ bzw $t$ mit einem Rückwärtsfenster $h = 4$

<table>
<thead>
<tr>
<th>Fenster zur Zeit x</th>
<th>Fenster zur Zeit t</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdots \ r_1 \ r_2 \ r_3 \ r_4 \ r_5 \ r_6 \ r_7 \ r_8 \ r_9 \ r_{10} \ r_{11} \ r_{12} \ r_{13} \ r_{14} \ r_{15} \ r_{16} \ r_{17} \ r_{18} \ r_{19} \ r_{20}$</td>
<td></td>
</tr>
</tbody>
</table>

Sei $w(t, h)$ die Mächtigkeit von $W(t, h)$. Dann steigt (trivialerweise) $w(t, h)$ mit wachsendem $h$.
Außerdem haben “lokale” Prozesse einen relativ kleinen Working Set, was bei nicht-lokalen Prozessen nicht der Fall ist.
Entscheidend ist immer das $h$ - ist es zu klein gewählt, dann sind nicht alle aktiven Seiten im Working Set, ist es zu groß, dann befinden sich viele inaktive Seiten darin.

**Beispiel 8.7.** Gegeben sei folgender Referenzstring:

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
<th>$r_7$</th>
<th>$r_8$</th>
<th>$r_9$</th>
<th>$r_{10}$</th>
<th>$r_{11}$</th>
<th>$r_{12}$</th>
<th>$r_{13}$</th>
<th>$r_{14}$</th>
<th>$r_{15}$</th>
<th>$r_{16}$</th>
<th>$r_{17}$</th>
<th>$r_{18}$</th>
<th>$r_{19}$</th>
<th>$r_{20}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

$W(t, h)$ habe $w(t, h)$ Elemente. Die Strategie basiert auf einem fest vorgegebenen $h$, z.B. $h = 8$.
Dann kann zu jedem $r_i$ ein Working Set berechnet werden:
$W(1, 8) = 0$ (negative Indizes sind nicht definiert! Geht man von der Stelle $r_1$ 8 Stellen nach links, so erhält man 0 und 7 mal die leere Menge $\emptyset$ - also eigentlich: $W(1, 8) = \emptyset, \emptyset, \emptyset, \emptyset, 0, 0, 0, 0$.)
Außerdem ist $w(1, 8) = 1$.
Wir erhalten weiter:
$W(2, 8) = 0, 2, w(2, 8) = 2$
$W(3, 8) = 0, 1, 2, w(3, 8) = 3$
$W(4, 8) = 0, 1, 2, 3, w(4, 8) = 4$
$W(2, 8) = 0, 2$, $w(2, 8) = 2$

... Die Mächtigkeit von $w$ wächst zunächst und stagniert dann, wenn alle Pages einmal schon erfasst worden sind.

Es ergibt sich für $r = 9$ folgender Working Set, indem man von der Stelle $r = 9$ Stellen rückwärts zählt und alle vorkommenden Werte nur einmal berücksichtigt:

$W(9, 8) = 1, 2, 3, 4, 5, 6, 7$ und somit $w(9, 8) = 7$.

Für $r = 19$ ergibt sich dagegen:

$W(19, 8) = 1, 3, 5$ und somit nur $w(19, 8) = 3$.

**Bemerkung 8.3.** Die Working Set Strategie wirft folgende Probleme auf:

1. Die Anzahl der Frames variiert pro Prozess und damit auch bezogen auf alle Prozesse insgesamt.⇒ Es kann passieren, dass mehrere Frames gebraucht werden als vorhanden sind, d.h. das System ist überlastet und es müssen Prozesse suspendiert werden.

2. Es stellt sich die Frage, wie der Parameter $h$ zu wählen ist. Eine Möglichkeit ist die Betrachtung der Lifetime-Funktion $L(m)$ und die Ermittlung/Wahl des Wertes $h$ in Relation zu $m$.

Die Working Set Strategie soll nun wie folgt zusammengefaßt werden:


2. Jedem aktiven Prozess so viele Frames zuteilen, wie seinem aktuellen Working Set $W(t, h)$ entsprechen. Wird zusätzlicher Speicherplatz frei, so kann gegebenenfalls ein neuer Prozess aktiviert werden.

Umgekehrt muß ein Prozess stillgelegt werden, wenn die Summe der Größen aller Working Sets zu einem bestimmten Zeitpunkt zu hoch ist.


(Ein Prozess stiehlt sich ein - nicht benutztes - Frame eines anderen Prozesses)

4. Ist keine ersetzbare Seite vorhanden, d.h. jede Seite also im Working Set eines bestimmten Prozesses, so wäre eine Seitenersetzung schädlich, da dadurch weitere Seitenfehler hervorgerufen werden würden. Man kann davon ausgehen, dass das System momentan überlastet ist - der anfordernde Prozess sollte deshalb stillgelegt werden.

Eine Reaktivierung ist erst sinnvoll, wenn sich die Verhältnisse gebessert haben.

5. Die Working Set Strategie eignet sich vor allem bei starker Lokalität.

**8.5 Segmentierungsstrategien**

Es gibt auch Seitenaustauschstrategien, die nicht auf einer Einteilung der Speicher in Frames und Pages basieren, sondern eine dynamische Speicherpartitionierung vornehmen. Bei
Kapitel 8. Speicher

aufeinanderfolgenden Speicherallokationen und -deallokationen können “Löcher” oder Frag-
mente freien Speichers entstehen, wodurch die Größe, des maximal an einem Stück zu allo-
zierenden Blockes immer mehr abnimmt. Um dies zu vermeiden, wird eine sogenannte Spei-
cherverdichtung durchgeführt, wozu allerdings ein relativ großer Zeitaufwand notwendig
ist. Daher benötigt man Strategien, wie man in einer vorliegenden, fragmentierten Speicher-
struktur mit gewissen Lücken neue Speichersegmente allokiert kann, sog. Segmentierungs-
strategien.

1. **First Fit (FF):** Plaziere das Segment in die erste passende Lücke.

2. **Best Fit (BF):** Plaziere das Segment in die kleinste passende Lücke.

3. **Rotating First Fit (RFF):** Plazierung wie bei First Fit, jedoch wird von der Position
der vorherigen Plazierung, d.h. dem Ende der benutzten Lücke ausgehend, die nächste
Lücke gesucht. Dieses Verfahren kann sowohl von vorne als auch von hinten begonnen
werden, wie man auch in Abbildung 8.13 sehen kann.

<table>
<thead>
<tr>
<th>Voraussetzungen:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>700 frei</td>
</tr>
<tr>
<td>900 frei</td>
</tr>
<tr>
<td>Eintreffende Anforderungen:</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>First Fit:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>Best Fit:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>Rotating First Fit:</td>
</tr>
<tr>
<td>von vorne begonnen</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>von hinten begonnen</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Abbildung 8.13: Beispiele für verschiedene Segmentierungstrategien

Während bei First Fit die Anzahl der Lücken schnell sehr groß wird, so dass ein hoher Spei-
cherverlust entsteht, und bei Best Fit der Fall auftreten kann, dass der verbleibende Rest der
Lücken extrem klein wird und später praktisch nicht mehr verwendet werden kann, wurde simulativ gezeigt, dass Rotating First Fit in der Regel am besten abschneidet, wobei der Anfangspunkt unerheblich ist.
E/A Verwaltung

- E/A Geräte
- E/A Techniken

Inhaltsangabe

| 9.1 Klassifizierung von E/A-Geräten | 186 |
| 9.2 E/A Techniken                 | 186 |
Wegen der Vielfalt der E/A Geräte ist es sehr schwierig eine allgemein gültige und effektive Ein/Ausgabe zu konzipieren.

9.1 Klassifizierung von E/A-Geräten

Externe Geräte für die Ein-/Ausgabe können in prinzipiell 3 Kategorien eingeordnet werden:

**Menschen - lesbare Geräte**  (human-computer interface)

Diese Geräte sind geeignet, mit dem Nutzer zu kommunizieren. In der Forschung beschäftigt man sich mit der Mensch-Maschine-Kommunikation (Oberflächendesign, Bedie-nungselemente etc.).

(z.B. Monitor, Tastatur, Maus, Drucker, ...)

**Maschinen - lesbare Geräte**

Sind geeignet, mit elektronischer Ausrüstung zu kommunizieren.

(z.B. Band und Plattenspeicher, Sensoren, ...)

**Kommunikationsgeräte**

Sind geeignet, mit entfernten Geräten zu kommunizieren.

(z.B. Modems, Netzwerkkarten, ...)

Die einzelnen Geräte können sich dabei noch einmal in verschiedenen Merkmalen unterschei-den:

- Datenrate
- Anwendung
- Komplexität der Kontrolle
- Arten des Datentransfer und Darstellung
- Fehlerbehandlung

9.2 E/A Techniken

Prinzipiell gibt es 3 verschiedene E/A Techniken:

**Programmierte E/A**

Durch einen Prozess wird im Prozessor ein E/A-Befehl ausgeführt.

Zum Beispiel wartet der Prozess dann im Busy-Waiting-Modus darauf, daß die Operation abgeschlossen ist, bevor er selber seine Abarbeitung weiter ausführen kann.

Dieses veraltete Konzept wurde später verbessert.

**Unterbrechungsgesteuerte E/A**

Direct Memory Access (DMA)


Zusammenfassung:

<table>
<thead>
<tr>
<th>E/A Speichertransfer durch den Prozessor</th>
<th>keine Unterbrechung</th>
<th>mit Unterbrechung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmierte E/A</td>
<td></td>
<td>Unterbrechungs-gesteuerte E/A</td>
</tr>
<tr>
<td>Direkter E/A Speichertransfer</td>
<td></td>
<td>DMA</td>
</tr>
</tbody>
</table>

Historische Entwicklung

1. direkte Kontrolle des Prozesses über die E/A.
4. Übergabe der Kontrolle an ein DMA-Modul, welches schließlich ein seperater Prozessor wird.
5. Integration von lokalem Speicher in das DMA-Modul.

D.h. der Prozessor ist im Laufe der Entwicklung immer weniger mit der E/A beschäftigt.

Begriffe in Zusammenhang mit E/A Modulen

E/A Kanal für ein fortgeschrittenes E/A Modul (ab Schritt 4.), welches bereits die Funktionalität eines eigenständigen Prozessors besitzt.

E/A Prozessor ist ein E/A Modul mit zusätzlichem lokalen Speicher

Realisierung des DMA

Zur Realisierung gibt es verschiedene Möglichkeiten:

1. Single Bus Detached DMA

⇒ Dieses Prinzip ist einfach und billig, aber nicht besonders effizient, denn jeder Transport benötigt 2 Buszyklen und es besteht die Gefahr eines von Neumannsch’en Flaschenhalses.

2. Single Bus Integrate DMA


Die Last auf dem Systembus wird gegenüber Single Bus Detached-DMA reduziert.

3. E/A Bus


Die Last auf dem Systembus wird wiederum reduziert.
Teil V

Interprozeßkommunikation
Lokale Interprozeßkommunikation

10.1 Grundlagen des Nachrichtenaustauschs .......................... 192
10.2 Pipes .............................................................................. 195
10.3 FIFOs .............................................................................. 200
10.4 Stream Pipes ................................................................. 200
10.5 Sockets ........................................................................... 201
**Motivation:** Um in einem Betriebssystem die Aktionen mehrerer Prozesse zu koordinieren und ein gemeinsames Arbeitsziel zu erreichen, ist es wichtig, daß Prozesse miteinander kommunizieren. Erfolgt diese Kommunikation über Prozessgrenzen hinweg, d.h. zwischen zwei oder mehreren Prozessen, so spricht man von Interprozesskommunikation (interprocess communication – IPC).


**10.1 Grundlagen des Nachrichtenaustauschs**

Beim Nachrichtenaustausch unterscheidet man drei verschiedene Arten der Verbindung:

- **Unicast**
- **Multicast**
- **Broadcast**

Unicast wird in der Literatur auch als Punkt-zu-Punkt Verbindung bezeichnet, Multicast als Punkt-zu-Mehrpunkt und Broadcast als Rundsendung. Diese Verbindungen können wie in Abbildung 10.1 veranschaulicht werden.

![Abbildung 10.1: Grundlegende Verbindungstypen](image-url)

Dabei lassen sich einzelne Verbindungsarten mittels anderer implementieren, z.B. kann eine Punkt-zu-Mehrpunkt Verbindung auf mehrere Punkt-zu-Punkt Verbindungen zurückgeführt werden.
werden, oder eine Broadcastverbindung mit nur einem Empfänger als Unicastverbindung realisiert werden.

Bei der Implementierung unterscheidet man:

– verbindungsorientierte Kommunikation
– verbindungslose Kommunikation

Prinzipiell kann man sich unter einer verbindungslosen Kommunikation vorstellen, daß ein Prozess Daten an einen anderen Prozess sendet, ohne mit diesem eine Vereinbarung hinsichtlich des Kommunikationsablaufs zu treffen. Dieser Fall ist z.B. beim IP-Protokoll des Internets vorhanden.

Bei der verbindungsorientierten Kommunikation wird erst eine Vereinbarung mit dem Partner getroffen, d.h. festgestellt, ob der Empfänger existiert und bereit ist, die Verbindung einzugehen. Nach dem Aufbau der Verbindung kann der Nachrichtenaustausch erfolgen. Danach wird die Verbindung wieder abgebaut. Dieser Fall ist z.B. beim Telefonieren oder beim TCP gegeben.

Da der Aufwand für den Aufbau und den Unterhalt einer derartigen Verbindung ziemlich groß ist, wird der moderne Nachrichtenaustausch – bei dem es in der Regel auch nicht so sehr auf Qualität ankommt, sondern der oft einfach nur billig sein soll – über verbindungslose Kommunikation realisiert.


– send (Kanal, Quelldaten),
– receive (Kanal, Zieldatenbereich)


Bei der asynchronen Kommunikation kann es z.B. vorkommen, daß der sendende Prozess Daten abschickt und der Zieldatenbereich beim empfangenden Prozß noch gar nicht bekannt ist. Dann werden die Daten oder der Name des Quelldatenbereichs im Kanal zwischengespeichert, bis eine receive-Operation aufgerufen wurde, wie man auch in Abbildung 10.2 sehen kann.

Erfolgt umgekehrt ein receive ohne vorheriges send wie in Abbildung 10.3, so muß im Kanal der Name des Zieldatenbereichs für den späteren Transport zwischengespeichert werden.

Die Alternative dazu besteht in der synchronen Kommunikation. Dabei weiß der Sender in der Regel, ob Daten beim Empfänger erwartet werden, und der Empfänger weiß, daß Daten
Kapitel 10. Lokale Interprozeßkommunikation

Abbildung 10.2: Send–Receive-Kommunikation (asynchron)

Abbildung 10.3: Receive–Send-Kommunikation (asynchron)
schon für ihn vorliegen, siehe auch Abbildung[10.4]. Diese *synchron*e Kommunikation geht also von einer zeitlichen Abstimmung zwischen dem Sender und Empfänger aus. Zum Teil wird diese Kommunikationsbeziehung auch als Rendevouz bezeichnet.

Abbildung 10.4: Synchron Kommunikation


<table>
<thead>
<tr>
<th></th>
<th>POSIX.1</th>
<th>XPG3</th>
<th>SVR4</th>
<th>BSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes (halbduplex)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FIFOs (named Pipes)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Stream Pipes</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Named Stream Pipes</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Message Queues</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semaphore</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Memory</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sockets</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>streams</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**10.2 Pipes**


Beispiel 10.1.
Kapitel 10. Lokale Interprozeßkommunikation

**cat:** sei ein Programm, das mehrere Dateien aneinanderhängt,

**pr:** sei ein Programm, das einen Text formatiert,

**lpr:** sei ein Programm, das einen Text ausdruckt.

Durch `cat Text1 Text2 | pr | lpr` werden die Texte `Text1` und `Text2` aneinandergehängt, formatiert und ausgedruckt. Der Interpreter, dem der Befehl übergeben wird, startet dazu die drei Programme als drei eigene Prozesse, wobei das Zeichen "|" ein Umlenken der Ausgabe des einen Programms in die Eingabe des anderen veranlaßt – und somit die Pipe realisiert. Im Mehrprozessorsystem können die drei Prozesse echt parallel ablaufen, auf einem Einprozessorsystem durch das Scheduling bedingt, quasi parallel.

Der Übergabemechanismus aus dem Beispiel, der Daten von einem Programm zum nächsten übermittelt, wird jeweils durch eine Pipe realisiert. Im folgenden soll betrachtet werden, wie ein Kommunikationskanal funktioniert. Klassische Pipes besitzen grundsätzlich zwei Eigenschaften:


- Pipes sind nur halbduplex, d.h. Daten können nur in eine Richtung fließen. Das Prinzip besteht darin, daß ein Prozeß, der eine Pipe zum Schreiben eingerichtet hat, in diese nur schreiben darf, während der Prozeß auf der anderen Pipe-Seite nur aus der Pipe lesen kann. Soll die Kommunikation auch in umgekehrter Form stattfinden, so muß eine zweite Pipe in umgekehrter Richtung etabliert werden.

Es gibt sogenannte **Stream Pipes**, die diese beiden Einschränkungen nicht besitzen. Im Rahmen dieser Vorlesung sollen jedoch nur die grundlegenden Mechanismen der Pipes betrachtet werden und daher diese beiden Eigenschaften zugrundegelegt werden.

**Prinzip einer Pipe:** In UNIX wird eine Pipe mit dem gleichnamigen Systemaufruf `pipe` eingerichtet, d.h. der Prozeß richtet eine Pipe im Kern ein. Ist dieser Aufruf erfolgreich so ist:

- `fd[0]`: ein geöffneter Filedescriptor zum Lesen aus der Pipe.


Abbildung 10.5: Einrichtung einer Pipe im Kernel

Abbildung 10.6: Grundsätzlicher Aufbau einer Pipekonstellation
Kapitel 10. Lokale Interprozeßkommunikation


Beim Schreiben in eine Pipe legt die Konstante PIPE_BUF die vom Kern verwendete Buffergröße für die Pipe fest. Werden mit einem oder mehreren write-Aufrufen, ohne zwischenzeitliche, den Buffer angemessen leerende, read-Aufrufe, mehr als PIPE_BUF Bytes hintereinander geschrieben, so können nicht alle Daten in den Buffer geschrieben werden und gehen verloren. Es ist auch möglich, daß mehrere Prozesse gleichzeitig Daten in dieselbe Pipe schreiben, diese werden dann sequentiell abgelegt.

Nun soll noch die Kommunikation zweier Kindprozesse über eine Pipe betrachtet werden. Dies passiert z.B. in nachstehender Reihenfolge:

- der Elternprozess kreiert das Schreib-Kind,
- der Elternprozess schließt die Schreibseite seiner Pipe,
- das Schreibkind schließt die Leseseite seiner Pipe,
- der Elternprozess kreiert das Lese-Kind,
- der Elternprozess schließt die Leseseite seiner Pipe,
- das Lese-Kind schließt die Schreibseite seiner Pipe,

Damit ergibt sich ein Kommunikationskanal wie in Abbildung 10.9.

Abbildung 10.7: Daten fließen zum Kindprozeß
Abbildung 10.8: Daten fließen zum Elternprozeß

Abbildung 10.9: Kommunikation mit zwei Kindprozessen
10.3 FIFOs

Während die bislang betrachteten Pipes nur zwischen Prozessen verwendet werden können, bei denen ein gemeinsamer Vorfahre die Pipe kreiert hat, werden oft auch Kommunikationsmechanismen benötigt, die zwischen beliebigen Prozessen einen Austausch von Daten ermöglichen. In den gängigen UNIX-Varianten stehen dafür sogenannte named pipes (benannte Pipes) zur Verfügung die in der Literatur auch als FIFOs bezeichnet werden. Sie sind eine spezielle Dateiart. Solche FIFOs können auch für die Realisierung einer Client/Server-Kommunikation wie in Abbildung 10.10 genutzt werden, wenn also Prozesse auf ganz unterschiedlichen Rechnern miteinander kommunizieren sollen. Bei Prozeduraufgaben unter-

![Abbildung 10.10: FIFO-Prinzip für drei Clients und einen Server](image)


10.4 Stream Pipes

10.5 Sockets


Beispiel 10.2.

_Einer TELNET-Verbindung wird in Rechner 1 die Portnummer 1080 und in Rechner 2 die Portnummer 23 zugewiesen. Die IP-Adresse von Rechner 1 sei x und die von Rechner 2 sei y. Dann lautet die Socket-Angabe in Recher 1 (Socket:[1080,x]) und die in Rechner 2 (Socket: [23,y]). Die Socket-Nummer bleibt während der ganzen Verbindung unverändert._

In diesem Sinne ist ein Socket eine Verallgemeinerung eines UNIX Zugangsmechanismus, der einen Kommunikationsendpunkt darstellt. Anwendungsprogramme können dabei das Betriebssystem beauftragen, bei Bedarf, einen Socket zu generieren.
Abbildung 10.12: Das Socket Prinzip
Verteilte Systeme

Inhaltsangabe

11.1 Einführung in Verteilte Systeme
  11.1.1 Historie Verteilter Systeme
  11.1.2 Vorteile Verteilter Systeme
  11.1.3 Klassifikation Verteilter Systeme
  11.1.4 Eigenschaften Verteilter Systeme

11.2 Kommunikation in Verteilten Systemen
  11.2.1 Das Client/Server Modell
  11.2.2 Der Remote Procedure Call
  11.2.3 Kommunikation in Verteilten Systemen
11.1 Einführung in Verteilte Systeme

Für den Begriff des Verteilten Systems kann man in der Literatur eine ganze Reihe unterschiedlicher Interpretationen finden. Das Verständnis davon, was man sich unter einem Verteilten System vorzustellen hat, ist historisch gewachsen und unterliegt noch heute einem beständigen Wandel. In \[7\] findet sich auf Seite 364 der Hinweis, man habe es immer dann mit einem Verteilten System zu tun, falls “multiple interconnected CPUs work together”. Demgegenüber ist auf Seite 382 die Rede von einem Verteilten System als “collection of machines that do not have shared memory”.


11.1.1 Historie Verteilter Systeme


Ein genereller Trend zur Abkehr von dem zentralen Ansatz ist seit Mitte der achtziger Jahre zu erkennen. Ausgelöst wurde er im wesentlichen durch vier Entwicklungstendenzen:


Diese vier Aspekte ermöglichen nicht nur die Entwicklung Verteilter Systeme, sie provozieren sie geradezu.

### 11.1.2 Vorteile Verteilter Systeme

Warum nutzt man die oben erwähnten Entwicklungstendenzen, um zentrale Systeme durch verteilte zu ersetzen? Es gibt eine ganze Reihe von Gründen, welche für Verteilte Systeme sprechen:

Kapitel 11. Verteilte Systeme

- Bestehende Lösungen sind integrierbar. Existierende Systeme können von neu hinzukommenden Systemkomponenten genutzt werden, ohne daß ein System gleicher Funktionalität neu entwickelt werden muß.

- Eine sukzessive Systemerweiterung minimiert das Risiko der Überlastung einzelner Systemkomponenten, indem stets auf die gleichmäßige Auslastung sowohl bestehender als auch neu hinzugekommener Module geachtet wird.

- Die überschaubare, organisatorische Verwaltung der Kapazität eines Verteilten Systems bedingt kosteneffektive Realisierungen. Das System ist flexibel und anpassbar.

- Der Eigentümer einer Ressource hat die Möglichkeit, das Management dieser Komponente selbst zu übernehmen. In jedem Fall steht es ihm frei, bei Bedarf einzugreifen, um seine eigenen Interessen wahrzunehmen.

- Die einzelnen Bestandteile eines Verteilten Systems sind weitestgehend autonom. Im Falle eines Fehlers oder sogar Ausfalls einer Systemkomponente können die übrigen Einheiten im Idealfall unbeeinflußt weiterarbeiten und ggf. den Störfall überbrücken.

Neben diesen (und vielen weiteren) Vorteilen eines Verteilten Systems kann man sich auch über Nachteile streiten, die u.a. in folgenden Punkten zum Ausdruck kommen:


- Die hinzugekommenen Netzwerkkomponenten können vollkommen neuartige Fehler verursachen.

- Aus der Sicht des Datenschutzes sind Verteilte Systeme bedenklich. Vernetzte Daten ermöglichen generell einfacher den Zugriff, als dies bei separater Datenhaltung der Fall ist.

11.1.3 Klassifikation Verteilter Systeme


Kapitel 11. Verteilte Systeme

Abbildung 11.2: Bus- und switchbasierte Systeme mit und ohne gemeinsamen Speicher

Tabelle 11.1: Kategorien Verteilter Systeme

11.1.4 Eigenschaften Verteilter Systeme

Nachdem im vorangegangenen Abschnitt geklärt worden ist, auf welche Weise sich verteilte Systeme unterscheiden, soll im folgenden auf die Gemeinsamkeiten der unterschiedlichen Ausprägungen verteilter Systeme eingegangen werden, dabei gibt es folgende Charakteristiken:


3. Es ist nicht praktikabel, ausschließlich globale Systemzustände zu betrachten. Lokale Zustandsbetrachtungen müssen zusätzlich durchgeführt werden.


Um diese Charakteristiken zu erreichen, müssen bestimmte Anforderungen erfüllt sein, um Verteilte Systeme geeignet modellieren und implementieren zu können:

– Offenheit d.h. Portabilität eines Systems,
– Integrierbarkeit, zur Behandlung der Heterogenität,
– Flexibilität, um die Evolution des Systems zu unterstützen
– Modularität, welche eine wichtige Grundvoraussetzung für die Flexibilität darstellt,
– Föderation, um den Zusammenschluß autonomer Einheiten zu ermöglichen,
– Verwaltbarkeit,
– Sicherstellung von Dienstqualitäten.
– Sicherheit und
– Transparenz.


– Die Zugriffstransparenz verbirgt die speziellen Zugriffsmechanismen für einen lokalen oder entfernten Dienst- oder Ressourcenauftruf.
– Die Ortstransparenz verbirgt die Systemtopologie als solche.
– Dem gegenüber hält die Migrationstransparenz Bewegung und Auslagerung von Funktion und Anwendungen transparent.
– Abarbeitungstransparenz verbirgt, ob die Abarbeitung eines Aufrufs parallel oder sequentiell erfolgt.
– Wird bei Ausfall einer angesprochenen Netzkomponente ohne Zutun des Nutzers ein Ersatz gefunden und angesprochen so spricht man von **Ausfalltransparenz**.

– Bleibt auch die Zuordnung von Ressourcen zu Anwendungsprozessen verkapselt, so liegt **Ressourcentransparenz** vor.

– Die **Verbundtransparenz** versteckt Grenzen zwischen administrativen und technischen Bereichen.

– **Gruppentransparenz** verbirgt das Benutzen von Gruppen.


### 11.2 Kommunikation in Verteilten Systemen

#### 11.2.1 Das Client/Server Modell

Am Anfang unserer Überlegungen soll die These stehen, daß das **OSI-MODELL** nicht zur Modellierung Verteilter Systeme geeignet ist. Warum ist dies der Fall? Der Verwaltungsaufwand der sieben Schichten ist zu hoch, denn bei der Übertragung einer Nachricht wird sieben mal ein Datenkopf angehängt bzw. wieder entfernt. Dies kostet Zeit, bei LANs in Relation zur Übertragung sogar sehr viel Zeit. Die Folge dieser Erkenntnis ist, daß Verteilte Systeme auf ein eigenes Grundmodell zurückgreifen : das **Client/Server-Modell(C/S-Modell)**.
Kapitel 11. Verteilte Systeme

Abbildung 11.4: Das Client/Server-Modell


Abbildung 11.5: Der Protokollstapel des Client/Server-Modells

Kommunikation in Verteilten Systemen


Abbildung 11.6: header.h

Exemplarisch ist in Abbildung 11.8 eine Client-Prozedur angegeben, die mit Hilfe des Servers eine Datei kopiert. Der Funktion copy werde dabei Zeiger src und dst auf die Dateinamen übergeben. Die Bestandteile der Datei werden paketweise vom Server eingelesen und anschließend wieder an diesen zurückgesendet.


Adressierung: In diesem Beispiel war der Client die Adresse des Servers dadurch bekannt, daß sie als Konstante in der Datei header.h enthalten war. Von einer solchen Voraussetzung kann nicht immer ausgegangen werden. Es ergibt sich ein Adressierungsproblem. Im Sinne...
#include <header.h>

void main(void) /* Es handelt sich um einen Prozedurauftrag. Es werden weder Parameter übergeben noch erwartet. */
void initialize(); /* wird später fuer dynam. Binden gebraucht */

while (true) {
    receive (FILE_SERVER,&ml); /* fuer FILE_SERVER ankomende Nachricht wird in Puffer ml kopiert. Blockierung, bis Nachricht angekommen und kopiert ist. */
    switch (ml.opcode) { /* Fallunterscheidung für Operation von ml */
        case CREATE: r=do_create(&ml, &m2); break;
        case READ: r=do_read(&ml, &m2); break;
        case WRITE: r=do_write(&ml, &m2); break;
        case DELETE: r=do_delete(&ml, &m2); break;
        default: r=E_BAD_OPCODE;
    }
    m2.result=r; /* Ergebniscode wird der ausgehenden Nachricht zugewiesen */
    send(ml.source, &m2); /* sende Antwort; Blockierung des Servers, bis die Nachricht verschickt ist */
}

Abbildung 11.7: server.c

#include <header.h> /* gleiche Definition wie beim Server */

int copy(char *src, char *dst) {
    struct message ml; /* Nachrichtenpuffer */
    long position=0; /* aktuelle Dateiposition */
    long client=110; /* Adresse des Client */
    initialize (); /* bereite Ausführung vor */
    do {
        /* lies einen Datenblock aus der Queldatei */
        ml.opcode=READ; /* als Operation wird Lesen definiert */
        ml.offset=position; /* setzen der aktuellen Dateiposition */
        ml.count=BUF_SIZE; /* def. wieviele Bytes auf einmal gelesen werden sollen */
        strcpy(ml.name, src); /* kopiere Dateinamen in Nachricht ml */
        send(FILE_SERVER, &ml); /* sende Nachricht an Dateiserver */
        receive(client, &ml); /* warte blockiert auf die Antwort */
        /* schreibe die empfangenen Daten in die Zieldatei */
        ml.opcode=WRITE; /* Operation ist Lesen */
        ml.offset=position; /* aktuelle Dateiposition */
        ml.count=BUF_SIZE; /* wieviele Bytes sollen gelesen werden */
        strcpy(ml.name, src); /* kopiere Dateinamen in Nachricht */
        send(FILE_SERVER, &ml); /* sende Nachricht an Dateiserver */
        receive(client, &ml); /* position=ml.result /* ml.result ist Anzahl geschriebener Bytes */
    } while (ml.result>0); /* wiederhole, bis alle Pakete übertragen, oder Fehler aufgetreten sind */
    return (ml.result); /* gebe OK oder Fehlercode zurück */
}

Abbildung 11.8: client.c
Abbildung 11.9: Blockierende Kommunikationsprimitive

Kommunikation in Verteilten Systemen

1. Der Anfrage an den Server und
2. dessen Antwort an den Client.


Ein anderer Ansatz sieht ein spezielles Lokalisierungspaket vor, das der Sender an alle anderen Rechner schickt. Das Lokalisierungspaket enthält die Adresse des Zielprozesses oder eine Bezeichnung des gesuchten Prozesses (Dienstes). Insbesondere für LANs, welche echtes Broadcast bereitstellen, wäre ein "Lokalisierung-durch-Broadcast"-Verfahren denkbar. Dieses Vorgehen sichert zwar die Ortstransparenz, verursacht durch den Broadcastaufruf jedoch zusätzliche Netzlast. Der Ablauf wird auf 4 Schritte ausgedehnt:

1. Broadcasten des Lokalisierungspakets,
2. "Ich bin hier"-Antwort des Servers,
3. Anfrage an den Server und

Eine Weiterentwicklung auf diesem Gebiet ist die Einführung eines Name-Servers oder Traders. Diese Begriffe unterscheiden sich leicht, da ein Trader i.d.R. etwas komfortabler ist. Das Prinzip sieht in beiden Fällen folgenden Ablauf vor:
Kapitel 11. Verteilte Systeme

Abbildung 11.10: Varianten der Adressfindung


2. Der Name-Server (Trader) übermittelt die Adresse an den Client.

3. Mit Hilfe dieser Adresse kann der Client nun die ursprüngliche gewünschte Anfrage stellen.

4. Der adressierte Server antwortet auf die Anfrage.

Bei Vorhandensein eines Traders kann dem Client auch bei unvollständigen Angaben ein Dienst vermittelt werden. Die Dienstvermittlung erfordert jedoch zahlreiche neue Mechanismen, Siehe [SPA94].

**Blockierung:** Bei Kommunikationsprimitiven unterscheidet man zwischen blockierenden (synchronen) und nichtblockierenden (asynchronen) Primitiven. Der Systementwickler wählt zwischen diesen beiden Arten. Bei blockierenden Primitiven wird der Prozeß während der Sendung einer Nachricht blockiert, d.h. Anweisungen werden erst dann weiter abgearbeitet, wenn die Nachricht vollständig abgesendet ist. Analog endet die Blockierung beim Empfangen erst, nachdem die Nachricht angekommen und kopiert ist. Die in Abbildung 11.6 bis Abbildung 11.8 vorgestellte Implementierung basierte auf der Verwendung von blockierenden Primitiven. Bei nichtblockierenden Primitiven wird die zu sendende Nachricht zunächst nur in einen Puffer der Betriebssystems kopiert. Direkt im Anschluß daran, also noch vor
Kommunikation in Verteilten Systemen


![Abbildung 11.11: ungepufferte und gepufferte Primitive](image-url)

Eine Alternative Implementierung, welche die genannten Probleme zu umgehen versucht, sind die gepufferten Primitive. Ein empfangender Kern speichert die eintreffenden Nachrichten für einen bestimmten Zeitraum zwischen. Wird keine passendes `receive` aufgerufen, so werden Nachrichten nach einem Timeout gelöscht, es müssen Puffer vom Kern bereitgestellt und verwaltet werden. Eine konzeptionell einfache Lösung sieht die Definition einer Daten-
strukturmailbox vor. Ein Prozeß, welcher Nachrichten empfangen will, fordert den Kern auf, eine Mailbox für ihn zu Erzeugen und gibt die Adresse an, mit der in eintreffenden Netzwerkpakten nachgesehen werden soll. Bei einem `receive` wird eine Nachricht aus der Mailbox geholt. Ist die Mailbox leer, so wird der Prozeß blockiert. Probleme treten nur bei volle Mailbox auf. In diesem Fall werden Aufrufe wie im ungepufferten Fall verworfen.

**Zuverlässigkeit:** Für eine Weiterklassifizierung der Primitive nach dem Grad ihrer Zuverlässigkeit bietet sich eine Einteilung in drei grobe Klassen an.

2. Primitive, bei denen der empfangende Kern eine individuelle Bestätigung an den Sender zurückschicken muß.

Eine Kombination der beiden letztgenannten Möglichkeiten sieht vor, daß vom piggy-backing auf individuelle Bestätigung übergegangen wird, falls der gesteckte Zeitrahmen überschritten wird.  

![Abbildung 11.12: Bestätigung mittels piggy-backing](image)


**11.2.2 Der Remote Procedure Call**

Das Client/Server-Modell bietet einen brauchbaren Weg, ein verteiltes Betriebssystem zu strukturieren. Trotzdem hat es eine extreme Schwachstelle: Das Basisparadigma auf dem alle Kom-


Abbildung 11.13: Stack bei lokalem Prozeduraufruf


Zusammenstellung der einzelnen Schritte:

1. Der Client ruft ein Unterprogramm im Client-Stub auf.
2. Der Client-Stub erzeugt eine Nachricht und übergibt sie an den Kern.
3. Der Kern sendet die Nachricht an den entfernten Kern.
4. Der entfernte Kern übergibt die empfangene Nachricht dem Server-Stub.
5. Der Server-Stub packt die Parameter aus und ruft ein Unterprogramm im Server auf.
6. Der Server führt das Unterprogramm aus und übergibt der Server-Stub die Ergebnisse.
9. Der Client-Kern übergibt die Nachricht an den Client-Stub.
10. Der Client-Stub packt die Ergebnisse aus und übergibt sie dem Client.

Hierbei wird der entfernte Prozedurauftrag auf die lokalen Aufrufe 1 und 5 zurückgeführt.

Abbildung 11.14: Ablauf eines RPCs

**Erweiterungen des entfernten Prozedurauftrags:** Der Client besitzt die Aufgabe, Parameter entgegenzunehmen, sie in eine Nachricht zu verpacken und diese an den Server-Stub zu senden. Hierbei tritt eine ganze Reihe von Problemen auf, welche im folgenden vorgestellt werden. Für die Parameterübergabe ist zunächst das Verpacken der Nachricht, das sogenannte (parameter)marshalling notwendig. Ein Aufruf der Art \( n = \text{sum}(4, 7) \) wird dabei in eine Nachricht der Form:

\[
\begin{array}{c}
\text{sum} \\
4 \\
7
\end{array}
\]

Kapitel 11. Verteilte Systeme

In dem Buch “Gullivers Reisen” ist von zwei Politikern die Rede, welche einen Krieg darüber begannen, an welchem Ende ein Ei aufzuschlagen sei. In Anlehnung an die dort verwendeten Bezeichnungen nennt man das INTEL-Format, bei dem die Bytes von rechts nach links nummeriert werden auch little endian. Das SPARC-Format, bei dem umgekehrt verfahren wird, kann demnach mit big endian bezeichnet werden.


Dynamisches Binden

In einem Verteilten System ist es möglich, dass sich aufgrund bestimmter Ereignisse die Adresse eines Servers bzw. seiner Schnittstelle ändert. Dies hätte zur Folge, daß zahlreiche Programme neu geschrieben und übersetzt werden müssten. Statt dessen benutzt man den Mechanismus des dynamischen Bindens. Ausgangspunkt hierfür ist eine formale Spezifikation des Servers, z.B. in folgender Form:

Programm 9 Formale Spezifikation eines Servers

<table>
<thead>
<tr>
<th>Zeile</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#include&lt;header.h&gt;</td>
</tr>
<tr>
<td>2</td>
<td>specification of file server, version 3.1;</td>
</tr>
<tr>
<td>3</td>
<td>long read (in char name[MAX_PATH], out char buf[BUF Size],</td>
</tr>
<tr>
<td>4</td>
<td>in long Bytes, in Long Position);</td>
</tr>
<tr>
<td>5</td>
<td>long write (in char name...);</td>
</tr>
<tr>
<td>6</td>
<td>int create(...);</td>
</tr>
<tr>
<td>7</td>
<td>int delete(...);</td>
</tr>
<tr>
<td>8</td>
<td>end;</td>
</tr>
</tbody>
</table>

Diese Spezifikation enthält:

<table>
<thead>
<tr>
<th>little endian</th>
<th>big endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adresse</td>
<td>10</td>
</tr>
<tr>
<td>Inhalt</td>
<td>T</td>
</tr>
</tbody>
</table>

Tabelle 11.2: Der String CLIENT und der Integerwert 5 in INTEL- und SPARC-Format
– den Namen des Servers (im Programm 6 ein file-server)
– die Versionsnummer (3.1)
– eine Liste von Unterprogrammen (read, write,...), die der Server anbietet.

Für alle Unterprogramm werden die Typen der Parameter angegeben. Diese Typen beziehen sich auf den Server, d.h.


– Ein out-Parameter wird vom Server an den Client gesendet, z.B. verweist buf auf die Adresse, an die der Server die Daten ablegt, die der Client angefordert hat.

– Außerdem gibt es in/out Parameter, die vom Client an den Server gesendet werden, dort modifiziert und schließlich an den Client zurückgesendet werden.


– seinen Namen,
– die Versionsnummer sowie ggfs. einen eindeutigen Bezeichner und
– ein sog. Handle, das zur Lokalisierung des Servers dient, z.B. Ethernet oder IP-Adresse

Möchte der Server keinen Dienst mehr anbieten, so läßt er sich “entregistrieren”. Mit diesen Voraussetzungen ist Dynamisches Binden möglich, das folgendermaßen geschieht:

– der Client ruft zu ersten mal ein entferntes UP, z.B. read, auf.
– der Client-Stub erkennt daraufhin, dass der Client an noch keinen Server gebunden ist.
– der Client-Stub sendet eine Nachricht an den Binder, um z.B. Version 3.1 der Schnittstelle file-server zu importieren.

Der Binder überprüft, ob (ein oder mehrere) Server eine Schnittstelle mit diesem Namen und der entsprechenden Versionsnummer exportieren. Ist das nicht der Fall, so schlägt die Operation fehl. Falls andererseits ein passender Server existiert, so liefert der Binder das
Handle und den eindeutigen Bezeichner an den Client-Stub zurück. Der Client-Stub benutzt das Handle als Adresse, an die er die Anfragenachricht sendet. Die Nachricht enthält die Parameter und den eindeutigen Bezeichner, mit dem der richtige Server ausgewählt wird, falls mehrere existieren. Der Vorteil *dynamischen Bindens* ist die hohe Flexibilität. Der Binder kann:

- den Server in regelmäßigen Abständen überprüfen und ggfs. entregistrieren,
- bei identischen Servern und Clientanfragen die Auslastung steuern (*load balancing*)
- Authentifikation unterstützen.

Demgegenüber hat das dynamische Binden aber auch Nachteile:

- Es bedeutet zusätzlichen Aufwand für das Im- und Exportieren von Schnittstellen.
- In großen Systemen kann der Binder zum Engpaß werden, so daß mehrere Binder benötigt werden. Dies führt dazu, daß beim Registrieren und Entregistrieren viele Nachrichten verschickt werden müssen.

Solange sowohl Client als auch Server fehlerfrei funktionieren, erfüllt der RPC seine Aufgabe sehr gut. Treten jedoch Fehler auf, so gibt es Unterschiede zwischen lokalem und entfernten Aufruf. Man Unterscheidet fünf Klassen von Fehlerquellen in RPC-Systemen:

1. **Client kann Server nicht lokalisieren.** Z.B. weil kein passender Server ausgeführt wird (Versionsproblem o.ä.). Lösungsmöglichkeiten:
   - Fehlertyp anzeigen lassen und reagieren oder
   - Ausnahmebehandlung auslösen.

2. **Anfragenachricht vom Client an Server geht verloren.** Ist einfach zu lösen: Kern wie- derholt Anfrage nach einem Timeout.

3. **Der Server fällt nach Erhalt einer Anfrage aus.** Hierbei müssen wie man in Abbildung 11.15 sehen kann, drei Fehler unterschieden werden:

   ![Abbildung 11.15: Serverausfall nach Erhalt einer Anfrage](image)

   - (a) ist Normalfall
   - (b) ist gleichbedeutend damit, daß die Anfrage nie angekommen ist (→ Wiederholung),
(c) bedingt, dass der Fehler dem Client gemeldet werden muß.

ABER: Wie soll der Client (b) und (c) unterscheiden?


5. **Der Client fällt aus, nachdem eine Anfrage gestellt wird.** Dann wird die Berechnung ausgeführt, obwohl niemand auf eine Antwort wartet (“verwaiste Berechnung”). Dies verschwendet Rechenzeit und kann zu Konfusionen führen, wenn der Client danach eine neue Anfrage startet und darauf das alte Ergebnis eintrifft. Für dieses Problem gibt es vier Lösungen von Nelson:

   – **Ausrottung:** Eine externe Protokollierungsdatei terminiert Waisen nach Neustart.
   – **Reinkarnation:** Jeder Neustart bestimmt eine neue Epoche (linear aufsteigend).
   – **sanfte Reinkarnation:** In einer neuen Epoche wird versucht, im verteilten System entfernte Besitzer zu finden.
   – **Verfallszeitpunkte:** Feste Zeitdauern T werden vereinbart, in den eine Antwort da sein muß oder erneut angefordert werden muß

**Implementierungsaspekte**

Zugrundeliegende Konzepte und die Art ihrer Implementierung bestimmen die Leistungsfähigkeit eines Verteilten Systems und dessen Erfolg.


– Soll man die Protokolle RPC-Spezifisch entwickeln oder allgemeine Standards nutzen? In der Regel benutzen Verteilte Systeme IP (oder UDP, das auf IP aufbaut) als Basisprotokoll. Dabei macht man sich den Vorteil zunutze, dass dieses Protokoll bereits existiert und somit keine Entwicklungsarbeit mehr investiert werden muß. Im übrigen können die Pakete von fast allen UNIX Systemen gesendet, empfangen und über viele Netze übertragen werden.

– Wie soll Bestätigung geregelt werden? Hierfür gibt es verschiedene Möglichkeiten:
Bei Stop-and-wait braucht das fehlerhafte Protokoll einfach nur wiederholt zu werden, beim Blast-Protokoll muß eine Entscheidung getroffen werden, ob komplett neu übertragen wird, oder ob man zwischenspeichert und das fehlerhafte Paket neu anfordert (selective repeat). Selective repeat ist nur mit großem Aufwand zu implementieren, senkt aber die Netzbelastung.


**Kritische Pfade**

Ein kritische Pfad (critical path) ist die Folge von Instruktionen, die bei jedem RPC ausgeführt werden. Dieser beginnt mit dem Aufruf des Client-Stubs durch den Client, geht dann über zum Kern, beschreibt die Nachrichtenübertragung, die Server-Seite, die Ausführungen des Unterprogramms und die Rückantwort. Genauer:

**Client-Seite:**

1. Im Client: Rufe Stub-Routine auf → Ab sofort Aktionen im Client-Stub.
2. Bereite Nachrichtenpuffer vor, d.h. erzeuge Puffer, um Anfragenachricht zusammenzustellen.
4. Fülle den Nachrichtenkopf aus.
5. Berechne die Prüfsumme.
   – Der Kern rettet den Inhalt des Prozessorregisters und der Adresstabelle, die er
     benutzt.
   – Die Nachricht wird in den Kern kopiert, da sie sich zur Verarbeitung in seinem
     Adressraum befinden muß.
   – Die Zieladresse wird bestimmt.
   – Die Adresse wird in den Nachrichtenkopf eingetragen.
   – Die Netzwerkschnittstelle wird initialisiert.

7. Füge das Paket zur Übertragung in die Warteschlange ein.

8. Übertrage das Paket über den Bus zu Steuerwerk.

9. Übertrage das Pakt über Ethernet.

→ Ende der Client-Seite.

**Server-Seite:**

1. Im Server-Kern: Hole Paket beim Steuerwerk ab.

2. Unterbrechungsbehandlungsroutine

3. Überprüfe, ob das Paket korrekt übertragen wurde, d.h. berechne die Prüfsumme
   – Entscheide, welcher Stub die Nachricht erhält.
   – Überprüfe, ob Stub wartet. Ansonsten wird die Nachricht abgelehnt oder zwischen-
     gespeichert.
   – Kopiere die Nachricht in den Server-Stub.


5. Packe die Parameter aus, lege die Parameter auf dem Stack ab und rufe dann den Server
   auf.
   – Der Server führt den Auftrag aus.

Die Frage, die sich bei der Implementierung des RPC´s stellt, ist nun, welcher Teil des kriti-
ischen Pfades der aufwendigste ist, also am meisten Rechenzeit in Anspruch nimmt(siehe auch
Abbildung [11.17]. Die Schwachstellen müssen analysiert und dann die Ausführung optimiert
werden. 1990 haben Schroeder und Burrows den kritischen Pfad einer Multiprozessorwork-
station analysiert und zwar:

1. für einen “leeren RPC”, d.h. den aufruf eines entfernten Unterprogramms ohne Daten-
   übertragung,

2. für einen RPC mit einem 1440 Byte-Feld als Parameter. Dieses Feld muß also verpackt
   und als Nachricht übertragen werden.
Interpretation: Beim leeren RPC entstehen die meisten Kosten durch:

- den Kontextwechsel in den Server-Stub, d.h. Retten des Inhaltes von Prozessorregister und Adresstabelle und Laden der benötigten Adresstabelle (Server-Seite 4.),
- die Unterbrechungsbehandlungsroutine im Server-Kern (Server-Seite 2.) und
- das Übergeben des Paketes an die Netzwerkschnittstelle im Client-Kern (8.).

Beim 1440 “Byte-RPC” gibt es andere Engpässe:

- Die Paketübertragung über Ethernet (9.),
- das Übergeben des Paketes an die Netzwerkschnittstelle im Client-Kern (wie oben, 8.),
- Entnehmen des Paketes von der Netzwerkschnittstelle im Server-Kern.

Zu beachten ist allerdings, daß man diese Messergebnisse so nicht ohne weiteres verallgemeinern kann, da sie sich nur auf das zugrundeliegende Multiprozessorsystem beziehen. Die Messwerte basieren auf einer bereits angepassten Implementierung, die aber hinsichtlich gewisser Aspekte noch weiter verbessert werden kann:

- Das Kopieren von Daten beeinflusst die Ausführungszeit eines RPC’s ganz wesentlich. Man wird sich also bemühen, einerseits so wenige Kopiervorgänge wie möglich zu benötigen und diese dann andererseits so effizient wie möglich zu gestalten. Im Idealfall wird die Nachricht aus dem Adressraum des Client-Stubs direkt auf das Netzwerk ausgegeben. (Kopiervorgang 1) und in Echtzeit in den Speicher des Server-Kerns abgelegt bzw. umgekehrt (Kopiervorgang 2). Im worst-case benötigt man 8 Kopiervorgänge. Durchschnittlich benötigt das Kopieren eines 32-Bit-Wortes etwa 500 Nanosekunden, d.h. man benötigt etwa 4 Mikrosekunden bei 8 Kopiervorgängen (unabhängig von der Netzschwelligkeit). Bei der Implementierung muß also ein Kompromiß zwischen komplizierten Mechanismen und zeitaufwendigen Kopiervorgängen gefunden werden.
Ein anderer Aspekt ist die Verwaltung von Stoppuhren, wie in Abbildung 11.18. Hierfür muß eine Datenstruktur erzeugt werden, die angibt, wann die Stoppuhr ablaufen soll und was in diesem Fall zu unternehmen ist. Man kann hierfür eine verkettete Liste verwenden, die alle laufenden Uhren enthält und ein Sortieren nach Ablaufzeiten (timeouts) regelt:

Abbildung 11.18: Verwaltung von Stoppuhren

Trifft eine Antwort oder Bestätigung vor dem Ablauf einer Stoppuhr ein, so muß der zugehörige Eintrag aus der Liste entfernt werden. In der Praxis werden nur wenige Stoppuhren ablaufen, somit macht das Eintragen/Entfernen die meiste Arbeit.

11.2.3 Kommunikation in Verteilten Systemen

Grundbegriffe

Def.: Eine **Gruppe** ist eine Menge von Prozessen, die miteinander kooperieren, d.h. auf eine vom Benutzer oder System festgelegte Art und Weise zusammenarbeiten.

Eine Gruppe besitzt folgende Eigenschaften:

- Wird eine Nachricht an eine Gruppe gesendet, so wird die Nachricht von allen Mitgliedern der Gruppe empfangen:

Abbildung 11.19: Punkt-zu-Punkt vs. Punkt-zu-Mehrpunkt-Kommunikation

- Gruppen sind dynamisch, d.h. neue Gruppen können erzeugt und existierende Gruppen aufgelöst werden; Gruppenmitgliedschaften brauchen nicht disjunkt zu sein.


Die Gruppenkommunikation kann zu Beispiel bei replizierten Dateiservern angewendet werden. Sie wird auch insbesondere bei der sog. **Groupware** verwendet, die die Computerunterstützung von Arbeitsgruppen oder Projektteams bezeichnet. Der Schwerpunkt liegt hier auf der Zusammenarbeit der einzelnen Mitarbeiter. Ein verwandter Ansatz findet sich beim **Computer Supported Cooperative Work (CSCW)**. Groupware lässt sich wie folgt kategorisieren: Beim Entwurf von Gruppenkommunikationsdiensten müssen die gleichen Entwurfseinscheidungen getroffen werden, wie beim 2-Parteien-RPC hinsichtlich Adressierung, Blockierung,

<table>
<thead>
<tr>
<th>gleichzeitig</th>
<th>zeitlich versetzt</th>
</tr>
</thead>
<tbody>
<tr>
<td>zentral</td>
<td>Group Decision Support System</td>
</tr>
<tr>
<td>verteilt</td>
<td>Video Konferenz, Server Sharing</td>
</tr>
</tbody>
</table>

Tabelle 11.3: Klassifikation von Gruppenkommunikationsssoftware
Pufferung und Zuverlässigkeit. Die neue Komplexität der Architektur bedingt jedoch prinzipiell neun weitere Aspekte (von denen auf die wichtigsten im folgenden eingegangen werden soll), die bei Punkt-zu-Punkt-Kommunikation keine Rolle spielen. Die Adressierung an die Gruppe erfolgt entweder durch:

- Einrichtung einer eigenen Multicastadresse für die Gruppe,
- oder durch Broadcasting und Entscheidung des Kerns, ob er zur Gruppe gehört,
- oder durch Prädikatadressierung, bei der die Nachricht ein Prädikat enthält, das ausgewertet wird und damit die Annahme der Nachricht bestimmt.

Desweiteren müssen die Sende- und Empfangsprimitive für die Gruppe ab `group send` und `group receive` modifiziert und angepasst werden. Damit die Gruppenkommunikation einfach benutzbar ist, braucht man zwei weitere Eigenschaften:

- *Atomarität*: Ein Multi- oder Broadcast wird immer entweder an alle oder an kein Mitglied der Adressatengruppe geschickt.
- *Nachrichtenreihenfolgeeinhaltung*
Abbildung 11.21: Aspekte der Gruppenkommunikation

Abgeschlossenheit der Gruppe:
- geschlossene Gruppe: Außenstehende Mitglieder können Nachrichten nur an einzelne Gruppenmitglieder und nicht an die gesamte Gruppe senden
- offene Gruppe: Außenstehende Mitglieder können Nachrichten nur an einzelne Gruppenmitglieder und nicht an die gesamte Gruppe senden

Strukturierung der Gruppe:
- demokratische Gruppe: Symmetrische Struktur; bei Ausfall einer Komponente wird die Gruppe kleiner, arbeitet aber weiter
- hierarchische Gruppe: Ausfall des Koordinators bringt Gruppe zum Stillstand, aber Koordinator kann ggfs. Entscheidungen treffen, z.B. Ein-/Austritt eines Mitglieds