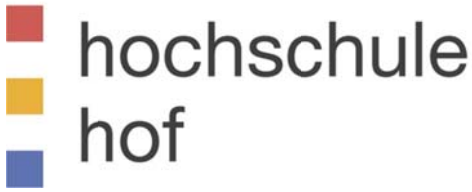


Fakultät Informatik

Vorlesung Informatik im 4. und 6. Semester

„Betriebssysteme II“

Prof. Dr. Horst Heineck
Alfons-Goppel-Platz 1
95028 Hof/Saale

Raum: **FB 134**
Tel.: **09281/409-444**
Fax: **09281/409-400**
Email: **Horst.Heineck@fh-hof.de**

Sprechstunde **Sommersemester 2010**
Montag: **16^{00} Uhr – 17^{00} Uhr**

Literaturverzeichnis

/Gates-95/ Gates, B.:
Der Weg nach vorn, William H. Gates III. und Hoffmann und Campe Verlag Hamburg, 1995

/Gulbins-95/ Gulbins, J., Obermayr, K.:
Unix System V.4. Begriffe, Konzepte, Kommandos, Schnittstellen, Springer-Verlag Berlin, Heidelberg, New York, London, Paris, Tokio, 1995

/Horn-95/ Horn, Ch., Kerner, I. O.:
Lehr- und Übungsbuch INFORMATIK, Band 1: Grundlagen und Überblick, Fachbuchverlag Leipzig GmbH, 1995

/Werner-95/ Werner, D.:
Taschenbuch der INFORMATIK, Fachbuchverlag Leipzig GmbH, 1995

/Born-98/ Born, G.:
Inside the Microsoft Windows 98 Registry, Microsoft Press, Redmond, 1998

/Hein-98/ Hein, J.:
Linux Systemadministration, Einrichten, Wartung, Software-Updates, Addision-Wesley, 1998

/Herold-99/ Herold, H.:
Linux Unix Shells, Addision-Wesley, 1999

/Hipson-00/ Hipson, P. D.:
Mastering Windows 2000 Registry, Sybex, Alameda, 2000

/Witzak-00/ Witzak, M.P.:
Echtzeitbetriebssysteme, Eine Einführung in Architektur und Programmierung, Franzis Verlag GmbH, 2000

/Hansen-01/ Hansen, H.R.:
Wirtschaftsinformatik I, 8.Auflage, Lucius&Lucius-Verlag Stuttgart, 2001

/Sterling-01/ Sterling, T.:
Beowulf Cluster Computing with Linux, MIT Press, Cambridge, Massachusetts, 2001

/Torvalds-01/ Torvalds, L., Diamond, D.:
Just for Fun, Wie ein Freak die Computerwelt revolutionierte, Hanser Verlag München Wien, 2001

/Mayer-01/ Mayer, A.:
Shellprogrammierung in Unix, Computer & Literatur Verlag GmbH, 2001

/Vogt-01/ Vogt. C.:
Betriebssysteme, Spektrum Akademischer Verlag Heidelberg, Berlin 2001

/Hansen-02/ Hansen, H.R.:
Arbeitsbuch Wirtschaftsinformatik I, 6.Auflage, Lucius&Lucius-Verlag Stuttgart, 2002

/Sterling-02/ Sterling, T.:
Beowulf Cluster Computing with Windows, MIT Press, Cambridge, Massachusetts, 2002

/Tanenbaum-02/ Tanenbaum, A.S.:
Moderne Betriebssysteme, 2. überarbeitete Auflage, Pearson Studium, 2002

/Weltner-02/ Weltner, T.:
Microsoft Windows XP Professional, Das Handbuch, Microsoft Press, 2002

/Berman-03/ Berman, F., Fox, G., Hey, T.:
Grid Computing, Making the global Infrastructure a Reality, John Wiley & Sons Ltd., 2003

/Gulbins-03/ Gulbins, J., Obermayr, K., Snoopy:
Linux, Springer-Verlag Berlin, Heidelberg, New York, London, Paris, Tokio, 2003

/Harris-03/ Harris, J.A.:
Betriebssysteme, 330 praxisnahe Übungen mit Lösungen, mitp-Verlag Bonn, 2003

/Morrison-03/ Morrison, R.S.:
Cluster Computing, Architectures, Operating Systems, Parallel Processing & Programming Language, Richard S. Morrison, 2003

/Stallings-03/ Stallings, W.:
Betriebssysteme, Prinzipien und Umsetzung, 4. überarbeitete Auflage, Pearson-Studium, 2003

/Brause-04/ Brause, R.:
Betriebssysteme. Grundlagen und Konzepte, 3. überarbeitete Auflage, Springer Verlag Berlin, Heidelberg, New York, London, Paris, Tokio, 2004

/Burtch-04/ Burtch, K.O.:
Linux Shell Scripting witch Batch, Sams Publishing, Indianapolis, Indiana, 2004

/Zilm-04/ Zilm, T., Günther, K.:
Bash, ge-packt, mitp-Verlag Bonn, 2004

/Ehses-05/ Ehses, E., Köhler, L., Riemer, P., Stenzel, H., Victor, F.:
Betriebssysteme, Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/LINUX, Pearson-Studium, 2005

/Hammerschall-05/ Hammerschall, U.:
Verteilte Systeme und Anwendungen, Architekturkonzepte, Standards und Middleware-Technologien, Pearson-Studium, 2005

/Lucke-05/ Lucke, R.W.:
Building Clustered Linux Systems, Prentice Hall PTR, 2005

/Tanenbaum-05/ Tanenbaum, A., van Steen, M.:
Verteilte Systeme, Grundlagen und Paradigmen, Pearson-Studium, 2005

/Wörn-05/ Wörn, H., Brinkschulte, U.:
Echtzeitbetriebssysteme, Springer-Verlag, 2005

/Achilles-06/ Achilles, A.:
Betriebssysteme, Springer-Verlag Berlin, Heidelberg, 2006

/Adelstein-07/ Adelstein, T., Lubanovic, B.:
Linux, Schnellkurs für Administratoren, O'Reilly Verlag GmbH & Co. KG, 2007-12-04

/Herold-07/ Herold, H., Lurz, B., Wohlrab, J.:
Grundlagen der Informatik, Person-Studium, 2007

/Linux-07/ Alles über: High Availability, Ausfallsicherheit unter Linux, Linux Magazin, Linux New Media AG, 04/2007

/Tanenbaum-08/ Tanenbaum, A. S., van Stehen, M.:
Verteilte Systeme, Prinzipien und Paradigmen, 2, aktualisierte Ausgabe, Person-Studium, 2008

/Willemer-08/ UNIX, Das umfassende Handbuch, Galileo Press, Bonn, 2008

/Zimmer-10/ Zimmer, D., Wöhrmann, B., Schäfer, C., Wischer, S., Kügow, O.:
VMware vSphere 4 – Das umfassende Handbuch, Galileo Press, Bonn, 2010

Ein herzlicher Dank geht an den Verlag Pearson Studium, der freundlicherweise die Bilder aus, z.B. /Tanenbaum-02/ für Dozenten in elektronischer Form zur Verfügung stellt.

Inhaltsverzeichnis

1. Shell-Programmierung

Im Betriebssystem **Unix** ist die Verwendung von Shells als **Kommandointerpreter** Standard. In der Geschichte von Unix sind eine Reihe von solchen Kommandointerpretern entwickelt und vertrieben worden.

Der erste Kommandointerpreter von Unix und damit auf jedem Derivat verfügbar ist die **Bourne-Shell**, benannt nach ihrem Entwickler S. R. Bourne als Mitarbeiter der **Bell Laboratories von AT&T**.

Als Weiterentwicklung aus dem Berkeley-Unix-Derivaten BSD wurde an der University of Califonia in Berkeley die **C-Shell** entwickelt, die heute ebenfalls auf den meisten Unix-Derivaten verfügbar ist. Diese Shell orientiert sich sehr stark an der Syntax der Programmiersprache C und ist somit als Programmiersprache nicht kompatible mit der Bourne-Shell, verfügte aber über sinnvolle Erweiterungen, z.B. den Historymechanismus.

Daraufhin wurde von David G. Korn an den Bell Laboratories von AT&T die **Korn-Shell** als Auftragswerk entwickelt. Diese Shell vereinigt die Syntax der Bourne-Shell mit den Erweiterungen der C-Shell und verfügt darüber hinaus über weitere sinnvolle Erweiterungen.

Alle Unix-Derivate, die auf der Unix-Version **SVR4** basieren, verfügen über alle drei Shells, die der Anwender auswählen und verwenden kann.

Im Rahmen der Vorlesung sollen nur die Funktionen und Abläufe behandelt werden, die für die Bourne- und die Korn-Shell gelten.

1.1. Kommandosyntax

Eine Shell im Betriebssystem Unix umfasst jeweils zwei Funktionen:

- Kommandointerpreter
- Programmiersprache

Im Abschnitt 1.5. Verarbeitungsstrukturen soll die Shell als Programmiersprache behandelt werden. Zunächst soll jedoch der weiteren Betrachtung die Kommandosyntax des Betriebssystems, repräsentiert durch einen Kommandointerpreter, vorangestellt werden.

Die erste syntaktische Einheit in der Kommandozeile wird als Kommando, d.h. als ein ausführbares Programm, ein shellinternes Kommando oder ein Schlüsselwort der Programmiersprache interpretiert.

Beispiele für **kommandoname**:

- find ausführbares Programm (/usr/bin/find),
- ls ausführbares Programm (/bin/ls),
- cd shellinternes Kommando,
- pwd shellinternes Kommando,
- while Schlüsselwort der Programmiersprache und
- for Schlüsselwort der Programmiersprache.

In Abhängigkeit von **kommandoname** werden die **argumente** definiert. Dabei lässt sich allgemein angeben, dass es sich in der Regel bei dem ersten Argument um **Optionen** bzw. **Schalter** handelt, die im Allgemeinen mit **-** beginnen. Danach können weitere **Parameter** folgen.
Das Syntaxdiagramm ist in Abbildung 1. angegeben:

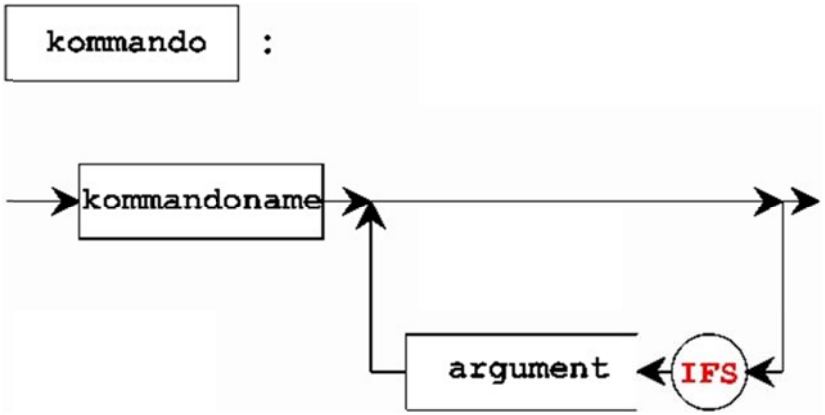


Abbildung 1: Syntaxdiagramm der Unix-Kommandos

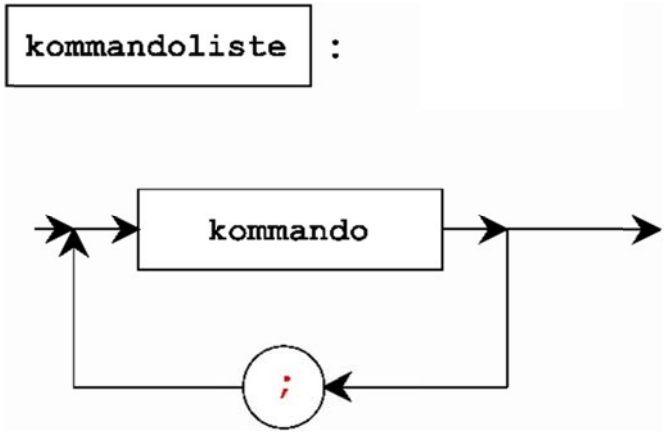


Abbildung 2: Syntaxdiagramm der Kommandoliste

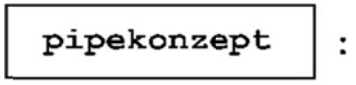
Wie in anderen Betriebssystemen auch, kann man in der Kommandozeile der Shell mehrere Kommandos angeben, die dann sequentiell abgearbeitet werden. Das Sonderzeichen, das als Trenner zwischen den Kommandos fungiert ist das **Semikolon**.

Das Syntaxdiagramm ist in Abbildung 2. angegeben.

Eine weitere Verknüpfung von Kommandos stellt das **Pipekonzept** von Unix dar. Dabei wird der **Standardausgabekanal** des Kommandos **i** mit dem Standardeingabekanal des Kommandos **i+1** verbunden.

Bezüglich der Betrachtung von Prozessen liegt somit eine einfache Form der

- **Prozesskommunikation und**
- **Prozesssynchronisation**



vor. Als Sonderzeichen zur Verbindung von Standardausgabekanal mit Standardeingabekanal fungiert das **Pipe-**Zeichen.

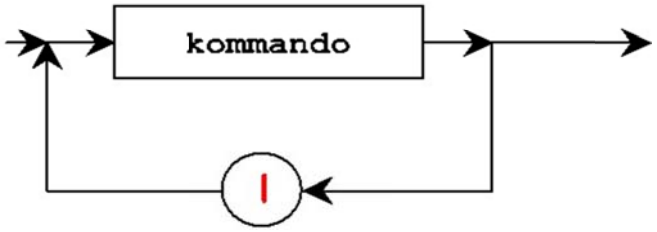
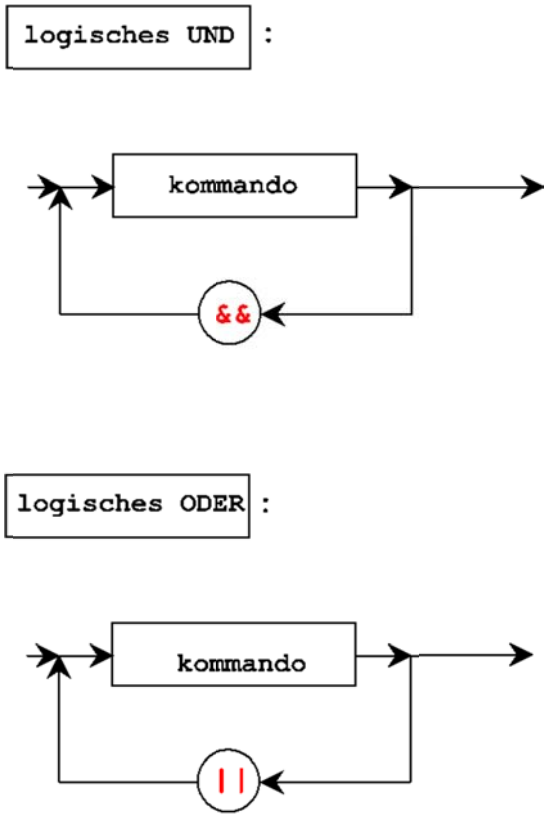


Abbildung 3: Syntaxdiagramm des Unix-Pipekonzeptes

Die Shell von Unix stellt weiterhin noch die Möglichkeit des logischen Verknüpfens von Kommandos zur Verfügung. Dabei sind die beiden Varianten **AND (UND)** und **OR (ODER)** vorhanden, die durch die Zeichenketten **&&** und **||** in der Kommandosyntax dargestellt werden.

Kommandofolgen, die mit dem logischen UND verknüpft sind, werden von links beginnend abgearbeitet, solange die Kommandos den **Exitstatus 0** liefern.



Bei Kommandofolgen mit dem logischen ODER werden die Kommandos solange von links beginnend bearbeitet, bis ein Kommando den **Exitstatus 0** liefert.

Die Syntaxdiagramme für beide Formen der logischen Verknüpfung von Unix-Kommandos sind im Abbildung 4. dargestellt:

Abbildung 4: Syntaxdiagramme für UND und ODER

Beispiel für eine UND-Verknüpfung von Unix-Kommandos:

```
$ who | grep -s "^paul" && write paul <nachricht
```

Für die Bewertung der Abarbeitung wird folgende Wahrheitstabelle verwendet:

Exitstatus Kommando 1	**Exitstatus Kommando 2**	**UND-Verknüpfung**
Exitstatus = 0 → ok	Exitstatus = 0 → ok	Exitstatus = 0 → ok
Exitstatus = 0 → ok	Exitstatus ≠ 0 → nicht ok	Exitstatus ≠ 0 → nicht ok
Exitstatus ≠ 0 → nicht ok	**wird nicht gestartet!**	Exitstatus ≠ 0 → nicht ok
Exitstatus ≠ 0 → nicht ok	**wird nicht gestartet!**	Exitstatus ≠ 0 → nicht ok

Tabelle 1: Wahrheitstabelle für UND-Verknüpfung

Beachtet werden muss, dass bei fehlerhafter Ausführung des Kommandos 1 das Kommando 2 nicht mehr gestartet wird. Die Abarbeitung wird abgebrochen.

Beispiel für eine ODER-Verknüpfung von Unix-Kommandos:

```
$ (who | grep -s "^paul" && write paul <nachricht ) || > mail paul < nachricht
```

Auch hier muss die Bewertung der Abarbeitung nach folgender Wahrheitstabelle erfolgen:

Exitstatus Kommando 1	**Exitstatus Kommando 2**	**ODER-Verknüpfung**
Exitstatus $=0$ → ok	**wird nicht gestartet!**	Exitstatus $=0$ → ok
Exitstatus $=0$ → ok	**wird nicht gestartet!**	Exitstatus $=0$ → ok
Exitstatus $\neq 0$ → nicht ok	Exitstatus $=0$ → ok	Exitstatus $=0$ → ok
Exitstatus $\neq 0$ → nicht ok	Exitstatus $\neq 0$ → nicht ok	Exitstatus $\neq 0$ → nicht ok

Tabelle 2: Wahrheitstabelle für ODER-Verknüpfung

1.2. Variable und Parameter

Die Shells von Unix kennen nur einen Datentyp ihrer Variablen und das ist der Datentype **string.**

In der Korn-Shell wurden darüber hinausgehend noch die Datentypen **integer** und **Feld (array)** realisiert. Damit lassen sich bestimmte Aufgaben in Shell-Skripten einfacher, übersichtlicher und im Ablauf schneller realisieren. Dennoch sind auch die einfachen Mechanismen, aufwärtskompatible zur Bourne-Shell vorhanden.

1.2.1. Umgang mit Shell-Variablen

Im Umgang mit Variablen der Shell sind doch einige Besonderheiten gegenüber anderen Programmiersprachen zu beachten.

Im Umgang mit Variablen lassen sich grundlegend drei Formen unterscheiden:

- **Variablendeklaration,**
- **Wertzuweisung und**
- **Wertreferenzierung (Zugriff auf den Wert einer Variablen)**

Zur Verdeutlichung des Sachverhaltes sollen wenige Beispiele dienen:

Form	Beispiel	Bildausgabe
Deklaration	$ paul=	
Wertzuweisung	$ paul=otto	
Wertreferenzierung	$ echo $paul	otto

Tabelle 3: Umgang mit Shell-Variablen

Im Allgemeinen werden Variablen in der Shell nicht deklariert. In der Wertzuweisung ist die Variablendeklaration implizit mit enthalten.

Wird eine Variable dennoch ohne Wertzuweisung deklariert, so wird bei der Wertreferenzierung die leere Zeichenkette zurückgegeben. Es gibt aber in der Shell Möglichkeiten, diesen Fall auszutesten und gegebenenfalls bestimmte Aktionen daraufhin auszuführen (siehe 1.3. Ersetzungsmechanismen.).

Weiterhin interessant ist die Gültigkeitsdauer von Shell-Variablen. Dazu soll eine kurze Kommandosequenz zur Verdeutlichung helfen:

$ lage=oben	# Variable *lage* wird deklariert mit dem Wert oben
$ echo $lage	# der Wert der Variablen wird referenziert
oben	# Wert der Variablen auf dem Bildschirm
$ ksh	# Aufruf einer Sub-Shell (Unterprogramm)
$ echo $lage	# erneut wird der Wert der Variablen referenziert die Variable *lage* ist in der Sub- # Shell nicht bekannt
$ lage=unten	# der Variablen *lage* wird ein neuer Wert zugewiesen
$ echo $lage	# der Wert der Variablen wird wieder referenziert
unten	# neuer Wert der Variablen auf dem Bildschirm
$ ^D	# Abmelden von der Sub-Shell, damit ist die alte Shell wieder aktiv
$ echo $lage	# erneut wird der Wert der Variablen referenziert
oben	# der erste Wert der Variablen ist erhalten geblieben

Tabelle 4: Kommandosequenz ohne export

$ lage=oben	#	Variable *lage* wird deklariert mit dem Wert oben
$ echo $lage	#	der Wert der Variablen wird referenziert
oben	#	Wert der Variablen auf dem Bildschirm
$ export lage	**#**	**die Variable *lage* wird exportiert**
$ ksh	#	Aufruf einer Sub-Shell (Unterprogramm)
$ echo $lage	#	erneut wird der Wert der Variablen referenziert
oben	**#**	**durch den Export der Variablen wird sie an die Sub-Shell übergeben**
$ lage=unten	#	der Variablen *lage* wird ein neuer Wert zugewiesen
$ echo $lage	#	der Wert der Variablen wird wieder referenziert
unten	#	neuer Wert der Variablen auf dem Bildschirm
$ ^D	#	Abmelden von der Sub-Shell, damit ist die alte Shell wieder aktiv
$ echo $lage	#	erneut wird der Wert der Variablen referenziert
oben	#	der erste Wert der Variablen ist erhalten geblieben

Tabelle 5: Kommandosequenz mit export

Grundsätzlich ist zu verzeichnen, dass Shell-Variable nur lokal in einer Shell, und damit in einem Shell-Skript Gültigkeit haben. Sie können durch den Befehl **export** global für weitere Ebenen (Unterprogramme) gemacht werden.

Es ist jedoch nicht möglich, den Wert einer Variablen aus einem Unterprogramm in die aufrufende Ebene zu übergeben.

1.2.2. Spezielle Variable

Die Arbeitsweise der Shell lässt sich sehr stark an individuelle Belange anpassen. Dazu werden Variable, so genannte Umgebungsvariable verwendet. Diese Variablen werden beim Starten der **Login-Shell** mit entsprechenden Werten deklariert.

Die Anpassung dieser Umgebungsvariablen kann durch die Abarbeitung der Shell-Skripte

- **/etc/profile**
- **$HOME/.profile**

durch den Systemadministrator und den Benutzer selbst vorgenommen werden.

Beispiele für Umgebungsvariable:

Variable	**Bedeutung**	**typischerer Wert**
HOME	Home-Directory des Benutzers	/home/otto
PATH	Anzahl von Directories zur Kommandosuche	/bin:/usr/bin:/etc
PS1	1. Promptzeichen **$** oder **#**	"$\PWD> "
PS2	2. Promptzeichen **>**	"weiter> "
MAIL	Pfad zum elektronischen Briefkasten	/usr/spool/mail/otto
IFS	Separatorzeichen für Kommandozeile	TAB, NL, Leerzeichen
TZ	Zeitzone	MEZ-1MESZ,3.5.0,9.5.0

Tabelle 6: Beispiele für Umgebungsvariable

Über die Umgebungsvariablen hinausgehend verwendet die Shell weitere spezielle Variablen, deren Werte zwar in einem Shell-Skript referenziert werden können. Das Verändern der Werte ist jedoch nicht möglich.

Diese speziellen Variablen werden durch die Shell selbst gesetzt und beinhalten Informationen über den Zustand der Shell.

Folgende spezielle Variablen sind definiert:

Variable	Bedeutung	Kommando
-	gesetzte Shell-Optionen	set -xv
$	PID (Prozessnummer) der Shell	kill -9 $$
!	PID des letzten Hintergrundprozesses	kill -9 $!
?	Exitstatus des letzten Kommandos	cat lalala ; echo $?

Tabelle 7: spezielle Variable

1.2.3. Positionsparameter

Sehr häufig wird man Shell-Skripte in der gleichen Art und Weise verwenden wollen, wie die "normalen" Unix-Kommandos. D.h., dem ablaufenden Shell-Skript müssen beim Aufruf eine Reihe von **argumenten** übergeben werden. Zu diesem Zweck gibt es die Positionsparameter. Dieser Begriff ist frei gewählt.

Die Shell analysiert die Kommandozeile und übergibt diese Eingabe in aufbereiteter Form an das Shell-Skript. Interessant dabei ist, dass dieser Mechanismus auch verschachtelt funktioniert. Somit kann man die

Positionsparameter **lokal** für einen Programmaufruf betrachten. Für die Parameterübergabe in die nächste Ebene werden dann dieselben Positionsparameter verwendet, allerdings mit anderen Werten.

Folgende Positionsparameter gibt es in der Umgebung des aufgerufenen Shell-Skriptes:

Positionsparameter	**Bedeutung**
#	Anzahl der Argumente **ohne kommandoname**
0	Name des Kommandos (***kommandoname***)
1	1. Argument **nach dem kommandoname**
:	:
9	9. Argument **nach dem kommandoname**
@	alle Argumente **ohne kommandoname**
*	alle Argumente **ohne kommandoname**

Tabelle 8: Positionsparameter

Zur Verdeutlichung soll ein kleines Beispiel-Shell-Skript dienen:

Shell_Proz:

```
# !/bin/ksh
# copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter übergeben"
echo "1. Parameter = $1"
echo "2. Parameter = $2"
```

```
echo "3. Parameter = $3"
echo "Alle Parameter zusammen:
$*"
echo "Meine Prozessnummer PID = $$"
```

Nachdem dieses Shell-Skript mit einem Editor erstellt wurde, muss es noch ausführbar gemacht werden. Dazu wird folgender Befehl eingegeben:

```
$ chmod u+x Shell_Proz
```

Daran anschließend wird das Shell-Skript wie folgt gestartet und erzeugt die entsprechenden Ausgaben auf dem Bildschirm:

```
$ Shell_Proz emil erna paul fred otto
Mein Name ist ./Shell_ProzMir wurden 5 Parameter übergeben
1. Parameter = emil
2. Parameter = erna
3. Parameter = paul
Alle Parameter zusammen:
emil erna paul fred otto
Meine Prozessnummer PID = 4711
$
```

Für ein sinnvolles Handeln der Positionsparameter 0 .. 9 bietet die Shell die Möglichkeit der **Linksverschiebung** mit dem Kommando **shift <zahl>.**

Dieses kleine Shell-Skript kann nun noch erweitert werden. Um das Prozesssystem andeutungsweise beobachten zu können, werden die speziellen Variablen verwendet. Dazu wird das Shellskript wie folgt ergänzt:

Shell_Proz:

```
# !/bin/ksh
# copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter übergeben"
echo "1. Parameter = $1"
echo "2. Parameter = $2"
echo "3. Parameter = $3"
echo "Alle Parameter zusammen:
$*"
echo "Meine Prozessnummer PID = $$"
# Ergänzungen für das Prozesssystem
UNTERPROGRAMM="Shell_Up"
$UNTERPROGRAMM &
echo "Das Unterprogramm $UNTERPROGRAMM mit der PID = $! wurde gestartet"
date
echo "Das Unterprogramm wird jetzt terminiert"
kill -9 $!
```

Bei den Ergänzungen fällt sofort auf, dass ein Unterprogramm **Shell_Up** im Hintergrund, als asynchron zu **Shell_Proz** gestartet werden soll. Dieses Unterprogramm muss mit einem Editor erstellt werden:

Shell_Up:

```
# !/bin/ksh
# copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
echo "--> Mein Name ist $0"
echo "--> Meine PID = $$"
ps -f
```

Danach muss das Shell-Skript **Shell_Up** noch ausführbar gemacht werden. Dazu wird folgender Befehl eingegeben:

```
$ chmod u+x Shell_Up
```

Daran anschließend wird das Shell-Skript **Shell_Proz,** wie oben gezeigt, gestartet und erzeugt die entsprechenden Ausgaben auf dem Bildschirm:

```
$ Shell_Proz emil erna paul fred otto
Mein Name ist ./Shell_Proz
Mir wurden 5 Parameter übergeben
1. Parameter = emil
2. Parameter = erna
3. Parameter = paul
Alle Parameter zusammen:
emil erna paul fred otto
Meine Prozessnummer PID = 611
Das Unterprogramm Shell_Up
mit der PID=612 wurde gestartet
--> Mein Name ist ./Shell_Up
--> Meine PID = 612
Mo, 10 Nov 1994 19:09:50 MEZ
Das Unterprogramm wird jetzt terminiert
UID PID PPID C STIME TTY TIME COMMAND
hheineck 456 1 13 18:50:59 04 0:01 -ksh
hheineck 614 1 7 19:09:50 04 0:00 ps -f
$
```

In der Ausgabe sieht man sehr deutlich die Mischung der beiden Ausgaben von **Shell_Proz** und **Shell_Up**. Daran ist ablesbar, in welcher Reihenfolge die beiden Shell-Skripts durch das Prozesssystem gestartet und abgearbeitet werden.

Interessant ist auch die Tatsache, dass im abschließenden Beobachten der Prozesstabelle, mittels Kommando **ps -f** sowohl das Shell-Skript **Shell_Proz** mit der **PID = 611** als auch das Unterprogramm **Shell_Up** mit der **PID = 612** zum Zeitpunkt der Ausführung des Kommandos nicht mehr enthalten sind.

Ebenfalls erkennbar ist die Tatsache, dass ein Prozess, dessen **Vaterprozess** selbst terminiert, als Vater-PID **(PPID)** die Prozessnummer des so genannten **Init-Prozess, PID = 1** erhält.

1.3. Ersetzungsmechanismen

Die Shell als Kommandointerpreter kennt so genannte **Metazeichen**, z.B. zur Dateinamen-Expandierung. Wie aus anderen Betriebssystemumgebungen sind die Zeichen ***** und **?**, und darüber hinaus in der Shell die Intervallangaben **[...]** bzw. **[!...]** bekannt. Die Sonderbehandlung dieser Metazeichen lässt sich auch sperren. Dazu verwendet man

- für ein Zeichen den **Backslash **
- für eine Zeichenkette
  a) "..."
  b) '...'

Nun gibt es in der Shell als Programmiersprache auch das Metazeichen zur Referenzierung von Shellvaribalen **$**.

Folgende Befehlssequenz soll den Unterschied zwischen den o.g. Möglichkeiten **a)** und **b)** demonstrieren:

```
$ A="Inhalt von A"
$ X="Zugriff auf $A"
$ echo $X
Zugriff auf Inhalt von A

$ X='Zugriff auf $A'
$ echo $X
Zugriff auf $A
$
```

Während die Maskierung des Metazeichens **$** mittels der Möglichkeit **b) '...'** gelingt, hat die Variante **a) "..."** bezüglich der Variablenreferenzierung keinen Effekt.

"$variable" **Die Shell wertet den Inhalt von variable aus!**
'$variable' **Die Shell wertet den Inhalt von variable nicht aus!**

Die Shell kennt darüber hinaus noch eine Möglichkeit der **doppelten Wertreferenzierung.**
Diese Möglichkeit basiert auf der Verwendung des Befehls **eval** und soll an einem Beispiel demonstriert werden:

```
$ text="Hallo Freunde"
$ zeiger="\$text"
$ echo $text
Hallo Freunde
$ echo $zeiger
$text
$ eval echo $zeiger
Hallo Freunde
$
```

Einen ähnlichen Mechanismus, wie das Kommando **eval** verwendet die Korn-Shell beim Aktualisieren der Umgebungsvariablen. Durch das Kommando

```
PS1="\$PWD> "
```

wird in Abhängigkeit von der Veränderung der Umgebungsvariablen **PWD** durch das Kommando **cd**, auch die Umgebungsvariable **PS1** mitgeändert.

Sollen in einem Shell-Skript in der Dialogausgabe der Text zusammengesetzt werden, so bietet die Shell eine weitere Möglichkeit der Notation der Variablenreferenzierung an.

```
$ a="Variable"
$ echo "$a A"
Variable A
$ echo "$anwert = A"
$
$ echo "${a}nwert = A"
Variablenwert = A
$
```

Bezüglich der Möglichkeit, den Namen einer Variablen durch die Zeichen **{}** kenntlich zu machen, sind weitere spezielle Variablenzugriffsmechanismen in der Shell implementiert worden:

${variable:-wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird auf diesen Wert referenziert.
b) Die Variable **variable** hat keinen Wert, dann wird bei der Referenzierung der Wert **wort** eingesetzt.

${variable:=wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird auf diesen Wert referenziert.
b) Die Variable **variable** hat keinen Wert, dann wird der Variablen **variable** der Wert **wort** zugewiesen und bei der Referenzierung der Wert **wort** eingesetzt.

${variable:?wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird auf diesen Wert referenziert.
b) Die Variable **variable** hat keinen Wert, dann wird die Fehlermeldung **wort** ausgegeben und die Shell-Prozedure abgebrochen.

${variable:+wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird der Wert **wort** referenziert.
b) Die Variable **variable** hat keinen Wert, dann bleibt dieser Zustand erhalten.

1.4. Kommandoersetzung

In der gleichen Art und Weise, wie Kommandos ihre Ausgabe auf den Bildschirm tätigen, sollte es natürlich möglich sein, dies in einer Variablen zu speichern.

Die folgende Form der Eingabe führt jedoch nicht zum Ziel

```
$ VERZEICHNIS=pwd
```

denn damit wird der **Variablen** VERZEICHNIS nur der Wert **pwd** zugewiesen, wie die Kontrolle deutlich macht:

```
$ echo $VERZEICHNIS
pwd
$
```

Für die Kommandosyntax wurde ein weiteres Metazeichen definiert, dessen Verwendung die folgende Befehlssequenz demonstriert:

```
$ VERZEICHNIS=`pwd`
$ echo $VERZEICHNIS
/home/hheineck/shell
$
```

Somit ist es möglich, im Laufe eines Programmablaufes auf Zwischenwerte in Kommandos zurückzugreifen, in der Form:

```
$ cd $VERZEICHNIS
$
```

Somit ist eine gängige Möglichkeit beschrieben, den Zustand beim Eintreten in ein Shell-Skript in Variablen zwischenzuspeichern und beim Austritt wieder einzustellen, ohne dass Zwischenzustände nach außen für den Benutzer sichtbar werden.

1.5. Verarbeitungsstrukturen

Zu einer Programmiersprache in der 3. Generation gehört auch ein prozeduraler Aspekt. Dieser wird durch die Verarbeitungsstrukturen in der Shell-Programmierung realisiert.

Begonnen werden soll mit der bedingten und einfachen Fallunterscheidung.

1.5.1. Bedingte und einfache Fallunterscheidung

Da die Syntaxdiagramme für die Verarbeitungsstrukturen von anderen Programmiersprachen bekannt sind, soll an dieser Stelle darauf verzichtet werden und nur die reine Syntax in ihrer Schreibweise verwendet werden.

Dabei werden die Schlüsselwörter der Sprache **blau** dargestellt. Dabei ist festzustellen, dass diese Schlüsselwörter in dieser Sprache die Begrenzer für die **rot** dargestellten syntaktischen Einheiten sind.

```
if kommando_1 ; kommando_2 ; ...
   then kommando_a ; kommando_b ; ...
   else kommando_A ; kommando_B ;...
fi
```

Das Kennzeichnende an den Verarbeitungsstrukturen in der Shell ist es, dass nach dem Schlüsselwort **if** keine Bedingung formuliert wird.

Vielmehr werden die Kommandos **kommando_1 ; kommando_2 ; ...** abgearbeitet und der Exitstatus des letzten Kommandos entscheidet, welcher Zweig danach durchlaufen wird.
Dabei gelten folgende Festlegungen:

a) bezüglich des Exit-Status eines Programms:

1. Programm ist fehlerfrei abgelaufen → Exitstatus = 0
2. Programmablauf war fehlerbehaftet → Exitstatus $\neq 0$

b) bezüglich der zu durchlaufenden Programmzweige:

1. Der Exitstatus des letzten Kommandos aus **kommando_1 ; kommando_2 ; ...** ist = 0
   → Kommandos **kommando_a ; kommando_b ; ...** werden abgearbeitet.
2. Der Exitstatus des letzten Kommandos aus **kommando_1 ; kommando_2 ; ...** ist ≠ 0
   → Kommandos **kommando_A ; kommando_B ; ...** werden abgearbeitet.

Für die Verwendung von verschachtelten Bedingten und Einfachen Fallunterscheidungen gibt es noch ein weiteres Schlüsselwort **elif**, das eine Reduktion des Schreibaufwandes bedingt:

```
if kommandoliste
   then kommandoliste
   elif kommandoliste
        then kommandoliste
        else kommandoliste
fi
```

Ein kleines Beispiel-Shell-Skript soll die Verwendung dieser Verarbeitungsstruktur verdeutlichen:

```
#!/bin/ksh
# copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
PROG=mail
DIR=`pwd`
if cd /bin ; ls | grep $PROG >/dev/null
   then echo "Das Programm $PROG ist im Verzeichnis /bin!"
   elif cd /usr/bin ; ls | grep $PROG >/dev/null
        then echo "Das Programm $PROG ist im Verzeichnis /usr/bin!"
```

```
        else echo "Kann $PROG nicht finden!"
fi
cd $DIR
```

1.5.2. Kommando test

Das Kommando **test** ist ein shellinternel Kommando und wird in Shell-Skripten sehr häufig verwendet, um den Test auf eine Bedingung zu formulieren. Dazu wird der Exitstatus des Kommandos entsprechend der Auswertung der Bedingung gesetzt:

a) Die Bedingung ist erfüllt → Exitstatus = 0
b) Die Bedingung ist nicht erfüllt → Exitstatus ≠ 0

Für die Verwendung des Kommandos gibt es zwei Notationsformen, die äquivalent verwendet werden können:

```
test [!] ausdruck
[ [!] ausdruck ]
```

Welche Art von Bedingungen können mittels Kommando **test** geprüft werden:

1. Eigenschaften von Dateien,
2. Eigenschaften und Vergleiche von Zeichenketten und
3. algebraische Vergleiche ganzer Zahlen

1.5.2.1. Eigenschaften von Dateien

ausdruck	Bedeutung
-r <datei>	datei existiert und Leserecht
-w <datei>	datei existiert und Schreibrecht
-x <datei>	datei existiert und Ausführungsrecht
-f <datei>	datei existiert und ist einfache Datei
-d <datei>	datei existiert und ist Verzeichnis
-h <datei>	datei existiert und ist symbolische Link
-c <datei>	datei existiert und ist zeichenorientiertes Gerät
-b <datei>	datei existiert und ist blockorientiertes Gerät
-p <datei>	datei existiert und ist benannte Pipe
-u <datei>	datei existiert und für Eigentümer s-Bit gesetzt
-g <datei>	datei existiert und für Gruppe s-Bit gesetzt
-k <datei>	datei existiert und t- oder sticky-Bit gesetzt
-s <datei>	datei existiert und ist nicht leer
-t <dateikennzahl>	dateikennzahl ist einem Terminal zugeordnet

Tabelle 9: test – Eigenschaften von Dateien

Beispiel-Shell-Skript zur Verdeutlichung:

```
if [ -r /etc/passwd ]
   then more /etc/passwd
   else echo "Kann Datei nicht lesen oder existiert nicht"
fi
```

1.5.2.2. Eigenschaften und Vergleiche von Zeichenketten

Zeichenketten in der Shell sind beliebige Folgen von Zeichen. Dabei ist auch die leere Menge möglich. Diese sollte jedoch unbedingt in Anführungszeichen "" eingeschlossen sein. Dadurch wird verhindert, dass das Kommando test keinen Parameter an einer beliebigen Stelle übergeben bekommt und dies durch den Exitstatus 1 und Abbruch quittieren wird.

Man sollte sich deshalb generell das Einschließen von Zeichenketten in den Anführungszeichen angewöhnen.

ausdruck	Bedeutung
-n <zeichenkette>	wahr, wenn zeichenkette nicht leer
-z <zeichenkette>	wahr, wenn zeichenkette leer ist
<zeichenkette1>= <zeichenkette2>	wahr, wenn Zeichenketten gleich sind
<zeichenkette1>!= <zeichenkette2>	wahr, wenn Zeichenketten verschieden sind
<zeichenkette>	wahr, wenn zeichenkette nicht leer

Tabelle 10: test – Eigenschaften von Zeichenketten

Beispiel-Shell-Skript zur Verdeutlichung:

```
if [ "$USER" = "otto" ]
   then echo "Guten Morgen Otto!"
        exit
fi
```

1.5.2.3. Algebraische Vergleiche ganzer Zahlen

In Variablen der Shell können bekanntlich nur Zeichenketten gespeichert werden. Dennoch kann die Shell eine Menge von Ziffern in beliebiger Größe als ganze Zahl behandeln und verarbeiten.

Die Syntax des Kommandos test bei dieser Möglichkeit reduziert sich auf folgende Notationsform:

```
ausdruck: <zahl1> <operator> <zahl2>
```

Die Syntax zeigt, dass es für die möglichen Vergleiche unterschiedliche Operatoren **operator** gibt:

operator	Bedeutung
-eq	equal - gleich
-ne	not equal - ungleich
-ge	greater than or equal - größer gleich
-gt	greater than - größer
-le	less than or equal- kleiner gleich
-lt	less than - kleiner

Tabelle 11: test – Vergleichsoperatoren

Beispiel-Shell-Skript zur Verdeutlichung:

```
#!/bin/ksh
# copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
# Möglichkeit der Verwendung eines algebraischen Vergleiches
# Test auf die Übergabe der richtigen Anzahl Parameter
# und Test auf gültigen 2. Parameter
#
if [ "$#" -ne "2" ]
then echo "usage: $0 -1Amn <file>"
     exit
fi
if [ ! -r "$2" ]
then echo "Cannot read file \'$2\': No such file"
     exit
fi
# gleiche Funktionalität, jedoch zeitlich günstiger, da keine
# Zahlenkonvertierungen notwendig sind!
if [ "$#" != "2" ]
then echo "usage: $0 -1Amn <file>"
     exit
fi
if [ ! -r "$2" ]
then echo "Cannot read file \'$2\': No such file"
     exit
fi
```

1.5.2.4. Logische Verknüpfung von Bedingungen

Das Kommando test lässt ebenfalls zu, dass mehrere Bedingungen zu einem Ausdruck ausdruck logisch verknüpft werden können.

Bei den logischen Verknüpfungen von Bedingungen sind die Möglichkeiten der booleschen Algebra mit folgender Notation verwendbar:

UND:	`<bedingung1>`	`-a <bedingung2>`
ODER:	`<bedingung1>`	`-o <bedingung2>`
Klammern:	`\( <ausdruck> \)`	
Negation:	`! <ausdruck>`	

Tabelle 12: logische Verknüpfungen

Shell-Skript:

```
#!/bin/ksh
#copyrigth by hheineck, Hochschule Hof, Fakultät Informatik
#
OLDDIR=`pwd`
BASE=`basename $1`
DIR=`dirname $1`
DATEI=`cd $DIR;pwd`/$BASE
if [ -s "$DATEI" -a -f "$DATEI" -a -r "$DATEI" ]
   then if (file "$DATEI" | grep "c program text" || \
             file "$DATEI" | grep "commands text" || \
```

```
            file "$DATEI" | grep "assembler program text" || \
            file "$DATEI" | grep "English text" || \
            file "$DATEI" | grep "ascii text") >/dev/null 2>&1
           then (pr -o10 -l64 -h "$DATEI" "$DATEI" | \
                 lp ) >/dev/null 2>&1
           else echo "Datei $DATEI hat nicht druckbaren Typ!"
        fi
   else echo "Datei $DATEI existiert nicht oder nicht zugreifbar"
fi
cd $OLDDIR
```

1.5.3. Mehrfache Fallunterscheidung

Eine mehrfache Fallunterscheidung in abgeänderter Bedeutung kann mittels case realisiert werden:

```
case wort in
   muster_1 ) kommandoliste_1;;
   muster_2 ) kommandoliste_2;;
   ...
esac
```

Für die einzelnen Fälle ist es nicht möglich, einzelne Bedingungen zu formulieren, sondern es wird mit Zeichenmustern die Unterscheidung realisiert. Die Zeichenkette **wort** wird dabei in der Reihenfolge der Anweisungen mit den vorgegebenen Mustern **muster_1**, **muster_2** usw. verglichen und bei Übereinstimmung wird die nachfolgende Kommandoliste ausgeführt. Die Kommandoliste mus mit einem doppelten Semikolon (**;;**) enden.

Folgendes Beispiel verdeutlicht die Verwendung der mehrfachen Fallunterscheidung:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
if echo '\c' | grep c > /dev/null 2>&1
   then N='-n'
   else C='\c'
fi
WEITER=nein
while [ "$WEITER" = "nein" ]
do
   echo ${N} "Bitte geben Sie J oder N ein: $C"
   read ANTWORT
   case $ANTWORT in
        y | Y | j | J ) echo "Sie haben J eingegeben"
                        WEITER=ja ;;
        n | N )         echo "Sie haben N eingegeben"
                        WEITER=ja ;;
        * )             echo "Sie können nicht lesen"
                        ;;
   esac
done
```

1.5.4. Abweisende und nichtabweisende Schleife

In der gleichen Art und Weise, wie bei der einfachen und bedingten Fallunterscheidung, steht auch bei den Schleifen keine Bedingung nach den Schlüsselwörtern **while** und **until**, sondern eine Kommandoliste.

Die Verwendung der beiden Schleifen ist analog zur Notation der Fallunterscheidung und sieht wie folgt aus:

1.5.4.1. Abweisende Schleife

```
while kommandoliste_1
do
   kommandoliste_2
done
```

1.5.4.2. Nichtabweisende Schleife

```
until kommandoliste_1
do
   kommandoliste_2
done
```

Ein kleines Beispiel soll die Verwendung in Shell-Skripten demonstrieren:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
# Unterschiedliche Möglichkeiten der Unterdrückung
# von Newline
#
if echo '\c' | grep c > /dev/null 2>&1
   then N='-n'
   else C='\c'
fi
#
# Beispiel für die abweisende Schleife
# Gestaltung eines sicheren Bedienungsdialoges
#
WEITER="j"
while [ "$WEITER" = "j" ]
do
   echo ${N} "Bitte die Gruppe eingeben: $C"
   read GRUPPE
   if grep "$GRUPPE" /etc/group >/dev/null 2>&1
      then WEITER="n"
      else echo "--> Fehleingabe!"
           echo
   fi
done
```

1.5.5. Zählschleife

Die Notation und Verwendung der Zählschleife in Shell-Skripten unterscheidet sich sehr stark von ihrer Verwendung in anderen Programmiersprachen. Dabei ist zu beachten, dass keine Laufvariable in bestimmten Grenzen hoch- bzw. heruntergezählt wird.

Vielmehr wird in der Shell der Zählschleife eine Menge von Werten übergeben. Bei jedem Durchlauf wird einer Variablen der nächste Wert aus der Menge von Werten übergeben.

Die Anzahl von Durchläufen ist an die Menge von übergebenen Werten gebunden:

```
for name in wert_1 wert_2 wert_3 wert_4
do
   kommandoliste
done
```

Für viele Anwendungen ist es notwendig, für die in den Positionsparametern übergebenen Parameter in einer Schleife die gleichen Kommandos auszuführen.

D.h., es wäre schön, wenn anstelle der **wert_1 wert_2 wert_3 wert_4** ein Positionsparameter übergeben werden kann.

Dazu gibt es folgende Vereinbarung:

Wenn der Positionsparameter **$*** verwendet werden soll, wird die Zählschleife wie folgt notiert:

```
for name
do
   kommandoliste
done
```

Damit sind folgende Zugriffsmöglichkeiten, z.B. auf Dateien in einem bestimmten Verzeichnis wie folgt möglich:

```
set - bsp*
for i
do
   if [ -s "$i" -a -r "$i" -a -f "$i" ]
      then more $i
   fi
done
```

Ein etwas umfangreicheres Beispiel soll an dieser Stelle die bisher eingeführten Möglichkeiten der Shell vereinigen und demonstrieren. Als zentrale Ablaufsteuerung wird dabei die Rekursion verwendet, bei der sich ein Programm selbst wieder aufruft:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
OLDDIR=`pwd`
BASE=`basename $0`
DIR=`dirname $0`
PROGRAMM=`cd $DIR;pwd`/$BASE
if [ "$#" != "3" ]
   then echo "usage : $PROGRAMM <directory> <UID> <GID>"
        exit 1
fi
if [ ! \( -d "$1" -a -r "$1" \) ]
   then echo "Cannot read directory \`$1\`: No such directory"
        exit 2
fi
if grep "^$2" /etc/passwd >/dev/null 2>&1
   then :
   else echo "Cannot read UID \`$2\`: No such User"
        exit 3
fi
if grep "^$3.*$2" /etc/group >/dev/null 2>&1
   then :
   else echo "Cannot read GID \`$3\`: No such Group \
              or User \`$2\` not member"
        exit 4
fi
USER="$2"
GROUP="$3"
```

```
BASE=`basename $1`
DIR=`dirname $1`
HOMEDIR=`cd $DIR;pwd`/$BASE
echo "program change $HOMEDIR"
cd $HOMEDIR
set - *
(chown $USER *;chgrp $GROUP *) >/dev/null 2>&1
for i
do
   if [ -d $i -a -r $i -a "$i" != "*" ]
      then $PROGRAMM $i $USER $GROUP
   fi
done
cd $OLDDIR
exit 0
```

1.5.6. Das Kommando expr

Da es innerhalb der Shell direkt nicht möglich ist, werden zu diesem Zweck mehrere „Rechenprogramme“ unter Unix zur Verfügung gestellt. Zu den gebräuchlichsten gehört das Kommando expr, das die übergebenen Parameter als Ausdruck interpretiert und diesen auswertet.

Damit sind arithmetische Operationen wie Addition, Subtraktion, Multiplikation, Division und Restwertberechnung durchführbar und reguläre Ausdrücke können ausgewertet werden.

Die Notationsform kann wie folgt geschrieben werden:

```
expr ausdruck_1 operator ausdruck_2
variable=`expr ausdruck_1 operator ausdruck_2`
```

Dabei sind folgende Operatoren möglich:

1. Vergleichsoperatoren `<, <=, >, >=, =, !=`
2. arithmetische Operatoren `+, -, *, /, %`
3. spezielle Operatoren `|, &, :`

1.5.6.1. Vergleichsoperatoren

Beispiele für Vergleichsoperationen:

```
$ a=5
$ b=6
$ expr "$a" \> "$b"
0
$ expr "$a" \< "$b"
1
```

1.5.6.2. Arithmetische Operatoren

Beispiele für arithmetische Operationen:

```
$ n=`expr "$n" + 1`
$ echo $n
25761
```

```
$ x=`expr "$#" \* 2`
$ echo $x
4
```

Bei den meisten Operatoren ist zu beachten, dass sie innerhalb der Shell meistens eine Sonderbedeutung haben und als Metazeichen vor der Shell versteckt (maskiert) werden müssen.

1.5.6.3. Spezielle Operatoren

a) **`expr ausdruck_1 | ausdruck_2`**

Wenn ausdruck_1 weder die leere Zeichenkette noch 0 ist, ist das Ergebnis ausdruck_1 sonst ausdruck_2.

b) **`expr ausdruck_1 & ausdruck_2`**

Wenn weder ausdruck_1 noch ausdruck_2 die leere Zeichenkette oder 0 ist, ist das Ergebnis ausdruck_1 sonst 0.

c) **`expr ausdruck_1 : ausdruck_2`**

Die Zeichenketten ausdruck_1 und ausdruck_2 werden miteinander verglichen, beginnend mit dem ersten Zeichen in jeder Zeichenkette und endet mit dem letzten Zeichen in ausdruck_2. ausdruck_2 kann in der Form von regulären Ausdrücken angegeben werden.

Beispiele für c):

```
$ expr "Hallo Freunde" : "Hallo"
5
$ expr "Hallo" : "allo"
0
$ expr "$PATH" : ".*"
65
$ LANG=`expr "$VARIABLE" : ".*"`
```

Innerhalb der regulären Ausdrücke stehen dabei:

`.`	ein beliebiges Zeichen und
`*`	eine beliebig häufige Wiederholung des vorangestellten Zeichens
`^`	das folgende Textmuster steht am Zeilenanfang

1.6. Funktionen

Seit dem Unix System V Release 2 können in der Bourne-Shell auch Funktionen definiert und verarbeitet werden. Diese Funktionen werden im Datenbereich der Shell gespeichert und direkt von dieser abgearbeitet. Damit sind folgende Vorteile gegeben:

- kein Zugriff auf das Dateisystem um ein Shell-Skript zu laden, die Funktion ist im Speicher der Shell vorhanden.
- es wird kein neuer Prozess gestartet

```
funktionsname()
   {kommando_1;kommando_2;...}
```

Ein kleines Beispiel soll die Möglichkeiten verdeutlichen, die mittels Funktionsdefinitionen realisiert werden können:

Das Kommando cp muss mit mindestens zwei Parametern aufgerufen werden. Der 1. Parameter ist im Allgemeinen die Quelldatei, der 2. Parameter ist die Zieldatei, bei mehr als zwei übergebenen Parametern das Zielverzeichnis. Werden jedoch weniger als zwei Parameter übergeben, so bricht das Kommando cp mit dem Fehler ab:

```
cp: insufficient arguments (0)
usage: cp f1 f2
       cp f1 ... fn d1
```

Die Aufgabe besteht nun darin, eine Funktion zu schreiben, die die fehlenden Parameter im Dialog abfragt, damit der Fehler nicht auftritt:

```
cp()
    {
     case "$#" in
          0)  echo "Bitte Quelldatei eingeben: \c"
              read QUELLE
              cp $QUELLE;;
          1)  echo "Bitte Zieldatei eingeben: \c"
              read ZIEL
              cp $1 $ZIEL;;
          *)  /bin/cp $*
     esac
     }
```

Fakultätsberechnung mittels Funktion unter Verwendung des expr-Kommandos:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
fak()
   {
   if [ "$#" -eq 1 ]
      then if [ "$1" -ge 1 ]
              a=1
              then n=$1
                   while [ "$n" -gt 1 ]
                   do
                      a=`expr "$a" \* "$n"`
                      n=`expr "$n" - 1`
                   done
           fi
           echo $a
      else echo "usage fak:  fak <integer>"
   fi
   return 0
   }
```

Abschlussbeispiel für Shellprogrammierung:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, Fakultät Informatik
#
HOMEDIR=/home/seminare/1110
DEFDATEI=/home/hheineck/profile/Datei
DIR=`pwd`
clear
echo "Kopieren beliebiger Dateien für Nutzer"
echo "======================================"
echo
if echo '\c' | grep c > /dev/null 2>&1
   then N='-n'
   else C='\c'
fi
WEITER="j"
while [ "$WEITER" = "j" ]
  do
    echo ${N} "Bitte die Gruppe eingeben: $C"
    read GRUPPE
    if grep "$GRUPPE" /etc/group >/dev/null 2>&1
       then WEITER="n"
       else echo "--> Fehleingabe!"
            echo
    fi
  done
echo ${N} "Bitte Namen für Quelldatei (default=$DEFDATEI) angeben: $C"
```

```
read QUELLDATEI
if [ "$QUELLDATEI" = "" ]
   then QUELLDATEI=$DEFDATEI
   else BASE=`basename $QUELLDATEI`
        QDIR=`dirname $QUELLDATEI`
        QUELLDATEI=`cd $QDIR;pwd`/$BASE
fi
echo ${N} "Bitte Namen für Zieldatei angeben: $C"
read ZIELDATEI
if [ "$ZIELDATEI" = "" ]
   then ZIELDATEI="  "
fi
cd $HOMEDIR
set - *
for i
  do
    if ls -ld $i |grep "^d.*$i.*$GRUPPE.*$i" >/dev/null
       then  echo  ${N}  "Soll  für  $i  Datei  nach  \"$ZIELDATEI\"  kopiert
werden (j/n): $C"
            read antwort
            case $antwort in
                 j|J|y|Y) cp $QUELLDATEI $HOMEDIR/$i/"$ZIELDATEI"
                          chown $i $HOMEDIR/$i/"$ZIELDATEI"
                          chgrp $GRUPPE $HOMEDIR/$i/"$ZIELDATEI";;
                 n|N    ) ;;
                 *      ) echo
                          exit 1;;
            esac
```

```
            echo
        fi
      done
cd $DIR
echo
echo "--> fertig!"
echo
exit 0
```

2. Einführung in Echtzeitbetriebssysteme

Zeitkritische Systeme, so genannte Echtzeitsysteme, spielen in einer Vielzahl von Anwendungsbereichen eine bedeutende Rolle. Zu nennen sind hier z.B.:

- Die Fabrikautomation,
- die Robotik,
- die Medizintechnik,
- die Kraftfahrzeugtechnik oder
- die Mobilkommunikation.

2.1. Grundlagen und ~begriffe

Die Grenzen zwischen reinen Echtzeitsystemen und Informationssystemen ohne zeitliche Randbedingungen sind in der Regel fließend, siehe Tabelle 13. Manche Systeme scheinen auch bewusst in ihrer Verarbeitung verzögert zu arbeiten. So sollte z.B. eine Überweisung auf üblichem Bankenweg mit heutiger Rechenleistung nicht Tage dauern, sondern in Sekunden oder höchstens Minuten ablaufen. Für den Kunden wirkt sich dies so aus, dass er nach erfolgter Überweisung von seiner Bank auf eine Fremdbank oder umgekehrt nach etwa ein bis zwei Tagen mit der Abbuchung oder Gutschrift rechnen darf. Interessanterweise kommt die äußerst mangelhafte Echtzeitfähigkeit von Bankensoftware nahezu ausnahmslos den Banken zu Gute.

Informationssysteme (nicht Echtzeitsysteme)	**Echtzeitsysteme**
datengesteuert	Ereignisgesteuert
komplexe Datenstrukturen	einfache Datenstrukturen
große Mengen an Eingangsdaten	meist kleine Mengen an Eingangsdaten
Ein- / Ausgabeintensiv	Rechenintensiv
maschinenunabhängig	Auf eine Hardwarearchitektur zugeschnitten

Tabelle 13: Systemvergleich konventionell - Echtzeit

2.1.1. Echtzeitsysteme

Definition 1: Echtzeitsysteme

Bei Echtzeitsystemen ist neben der Korrektheit der Ergebnisse genauso wichtig, dass Zeitbedingungen erfüllt werden. Es wird unter Echtzeit- bzw. Realzeitbetrieb der Betrieb eines Rechnersystems verstanden, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die anfallenden Daten oder Ergebnisse können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu bestimmten Zeitpunkten auftreten.

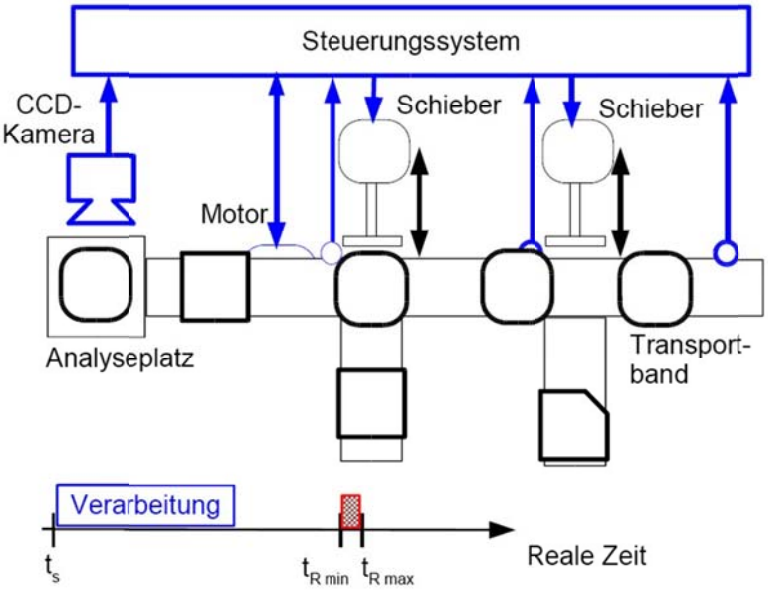


Abbildung 5: Beispiel für ein Echtzeitsystem

Ein Echtzeitsystem besteht gemeinhin aus der Hardware, einem Echtzeitbetriebssystem mit den für die Anwendung angepassten Bestandteilen und der Applikationssoftware. In der Applikation werden die vom Anwender gewünschten Aufgaben erfüllt.

Diese drei Komponenten müssen im Zusammenspiel das beschriebenen Antwortverhalten, d.h. den zeitlichen Determinismus garantieren. Hier gilt, je härter die zeitlichen Anforderungen sind, desto schneller muss die eingesetzte Hardware sein desto simpler muss das eingesetzte Betriebssystem bzw. die eingesetzte Software sein.

Viele technische Prozesse und technische Systeme werden unter harten Zeitbedingungen von sogenannten Echtzeitsystemen geleitet, gesteuert und geregelt.

2.1.2. Nicht-Echtzeitsysteme

Definition 2: Nicht-Echtzeitsysteme
Bei Nicht-Echtzeitsystemen kommt es ausschließlich auf die Korrektheit der Datenverarbeitung und der Ergebnisse an.

Für Nicht-Echtzeitsysteme lässt sich eine Vielzahl von Beispielen aufführen:

- Mathematische Berechnungen,
- betriebswirtschaftliche Kalkulationen,
- Textverarbeitung und vieles mehr.

2.1.3. Echtzeit:

Definition 3: Echtzeit

Der Begriff Echtzeit beschreibt die Zeitspanne t_1 minus t_0, die vergeht, bevor auf ein Eingangssignal der Steuereinrichtung, (Ausgangssignal des technologischen Prozesses) ein Ausgangssignal der Steuereinrichtung (Eingangssignal des technologischen Prozesses) berechnet und ausgegeben wird. Diese Echtzeit muss unter allen Umständen durch die Steuereinrichtung (die Automatisierungseinrichtung) garantiert werden. Die geforderte Echtzeit hängt weitestgehend vom technologischen Prozess ab.

Beispiele:

- Ernte von Tomaten und
- CNC-Werkzeigmaschine.

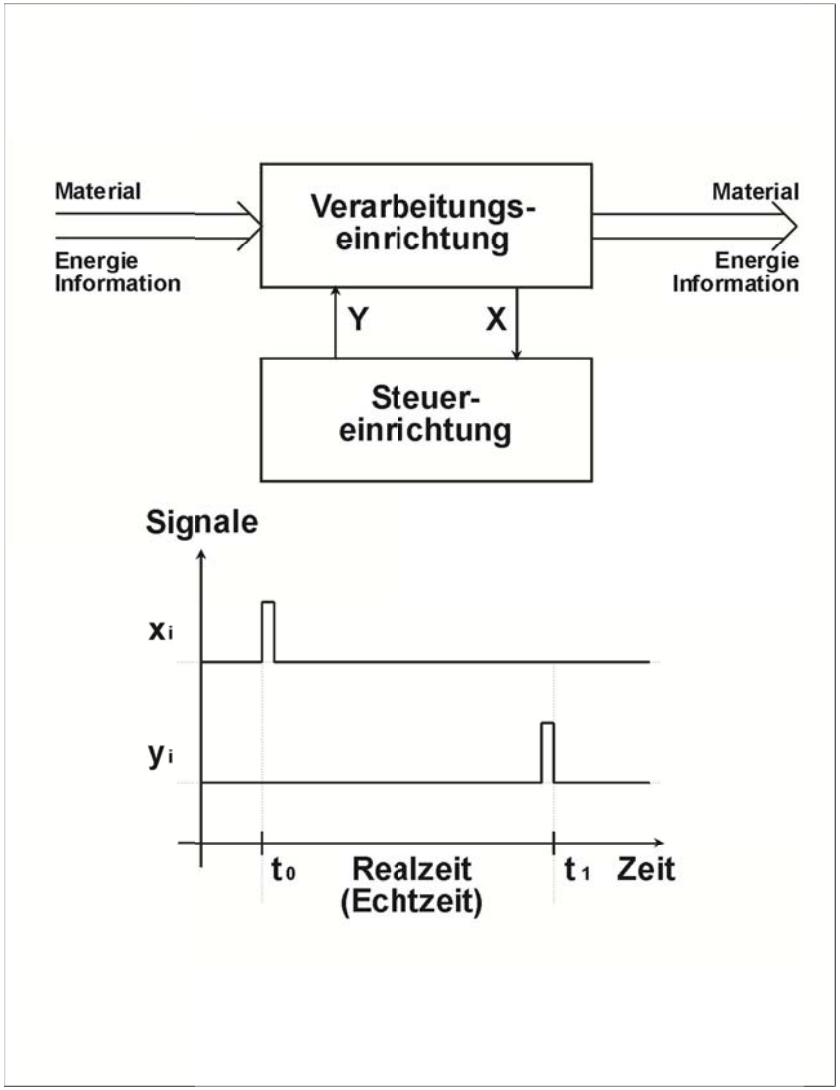


Abbildung 6: Echtzeit

2.1.4. Rechtzeitigkeit

Der Begriff Rechtzeitigkeit auf die Datenverarbeitung angewandt bedeutet, dass Eingangsdaten rechtzeitig im Sinne von nicht zu spät zur Verfügung stehen müssen. Die Berechnung der Ausgabedaten muss ebenso rechtzeitig zu verwertbaren Ausgangsdaten führen. Treten im Verlauf der Bearbeitungskette zwischen Aus- und Eingangsdaten, bei einem technischen Prozess, Verzögerungen auf, so kann daraus ein Fehlverhalten resultieren. Dies ist darin begründet, dass die eingelesenen Daten zum Zeitpunkt der Ausgabe des Ergebnisses keine Gültigkeit mehr besitzen. Je nach den zeitlichen Anforderungen des technischen Prozesses oder allgemein der realen Ergebnisse verlieren die Eingangsdaten unterschiedlich schnell ihre Gültigkeit. Das heißt die Eingangsdaten beschreiben den aktuellen Systemzustand nicht mehr korrekt.

Misst man zum Zeitpunkt t_0 die Temperatur eines Raumes zum Zweck der Reglung, so sollte zwischen dem Stellbefehl an das Reglerventil (Datenausgabe) zum Zeitpunkt t_1 und der Messung (Dateneingabe) t_0 nicht zwei Stunden liegen. Zwischenzeitlich könnte jemand das Fenster geöffnet haben, so dass die ursprüngliche gemessene Temperatur nicht mehr stimmt. Die Folge wäre ein falscher Stellwert und somit eine Fehlregelung.

Definition 4: Rechtzeitigkeit

Rechtzeitigkeit fordert, dass das Ergebnis für den zu steuernden Prozess rechtzeitig, d.h. innerhalb einer vorgegebenen Zeit vorliegen muss. Zum Beispiel müssen Zykluszeiten und Abtastzeitpunkte genau eingehalten werden.

2.1.5. Gleichzeitigkeit

Typischerweise müssen Echtzeitsysteme für technische Prozesse in der Regel mehrere Eingangsgrößen parallel auswerten. Der Grund liegt darin, dass reale Ereignisse sich nur mit einer Vielzahl von Daten beschreiben lassen.

Betrachtet man eine Kraftwerksregelung so laufen dort pro Sekunde mehrere tausend Daten ein, die zum Teil in Zusammenhang stehen. Ein Datenverarbeitungssystem muss dort in der Lage sein, mittels geeigneter Mechanismen die ankommenden Daten gleichzeitig zu verarbeiten. Die Erfüllung dieser Aufgabe verlangt nach Möglichkeiten der Parallelisierung von Berechnungsaufgaben und nach Unterscheidungsmöglichkeiten in verschiedene Wichtungen der Aufgaben. Die Hilfsmittel zur Lösung dieser Aufgaben sind typische Bestandteile und Eigenschaften von Echtzeitbetriebssystemen.

Definition 5: Gleichzeitigkeit

Gleichzeitigkeit bedeutet, dass viele Aufgaben parallel, bzw. pseudoparallel, jede mit ihren eigenen Zeitanforderungen bearbeitet werden müssen.

2.1.6. Determiniertheit

Die Rechtzeitigkeit und Gleichzeitigkeit bedingen, dass Berechnungsaufgaben zum Teil parallel, nach Wichtigkeit geordnet und im Normalfall unterbrechbar erledigt werden. Unterbrechbar bedeutet, dass Echtzeitsysteme sogenannte asynchrone Ereignisse erzeugen, auf die das Steuerungssystem auch asynchron reagieren muss.
Ein entsprechendes Echtzeitbetriebssystem muss über entsprechende Betriebsmittel verfügen. Die Parallelität, die Gewichtung der Aufgaben und die asynchrone Unterbrechnungsmöglichkeit dürfen nicht dazu führen, dass die Rechtzeitigkeit verletzt wird. Da die Rechtzeitigkeit meist eine relative Größe ist, d.h. auf Zeitpunkte bezogen ist, wird eine Verallgemeinerung eingeführt, die besagt, dass ein Echtzeitsystem in vorgebbaren zeitlichen Schranken deterministisch arbeiten muss.

Definition 6: Determiniertheit

Ein Echtzeitsystem arbeitet zeitlich determiniert, wenn zu jeder Kombination von Eingangsgrößen die Reaktionszeit des Echtzeitsystems in festen zeitlichen Grenzen vorhersagbar ist.

Die Gültigkeitsdauer von Eingangsdaten bzw. Änderungsgeschwindigkeiten von externen Ereignissen bestimmen die zeitlichen Deadlines der Echtzeitsysteme. Man unterscheidet:

Eine Robotersteuerung muss z.B. parallel das Anwenderprogramm interpretieren, die Führungsgrößen erzeugen, bis zu 20 und mehr Achsen regeln, Abläufe überwachen, usw.

Definition 7: spontane Reaktion auf Ereignisse

Spontane Reaktion auf Ereignisse heißt, dass das Echtzeitsystem auf zufällig auftretende interne oder externe Ereignisse aus dem Prozess innerhalb einer definierten Zeit reqagieren muss.

Ein Echtzeitsystem besteht aus Hard- und Softwarekomponenten. Diese Komponenten erfassen und verarbeiten interne und externe Daten und Ereignisse. Die Ergebnisse der Informationsverarbeitung müssen zeitrichtig an den Prozess, an andere Systeme bzw. an den Nutzer weitergegeben werden. Dabei arbeitet das Echtzeitsystem asynchron bzw. zyklisch nach einer vorgegebenen Strategie.

Randbedingungen	Forderungen
Dynamik der Vorgänge (Zeitanforderungen)	Rechtzeitigkeit und vorhersehbares Verhalten
- Zeitverhalten durch reale Prozesse bestimmt => „Echtzeit“ - reale Prozesse bestimmen Zeitpunkte für Ereignisse (E-Daten, Takt)	- „Schritthalten mit dem realen System“ notwendig - Reaktion innerhalb vorgegebener Zeitschranken (auch auf sporadische Ereignisse) - Synchronisation mit realer Zeit - vorhersagbare max. Reaktionszeiten (max. Systemlast, gleichzeitige Ereignisse **"worst case"-Zeiten !!!**) - Zugriff auf alle erforderlichen Ressourcen sichern
gleichzeitige Vorgänge	Gleichzeitigkeit
Bsp.: Transportband-Steuerung, Datenerfassung von der CCD-Kamera	- gleichzeitig laufende Programme ("Prozesse") - Multitask- / Multirechner-Betrieb
Kopplung der Vorgänge	Wechselwirkung
- Teilprozesse der physikalischen Systeme sind gekoppelt (Konkurrenz, Kooperation)	- Koordinierung der Prozesse notwendig (Kommunikation und -Synchronisation)

Tabelle 14: Echtzeitanforderungen

Beispiele für interne Ereignisse:

- Alarm aufgrund des Statuswechsels einer Hardwarekomponente – Hardwareinterrupt oder
- einer Softwarekomponente – Softwareinterrupt.

Beispiele für externe Ereignisse:

- Zeittakt eines externen Zeitgebers bzw. einer Uhr,
- eine Anforderung eines Sensors,
- eines Peripheriegerätes oder
- eines Nutzers (Mensch-Maschine-Kommunikation).

Entsprechend den technischen Anforderungen besteht somit die Aufgabe darin, die vom technischen Prozess gegebenen Echtzeitanforderungen zu analysieren und ein passendes Echtzeitbetriebssystem in Kombination mit einem geeigneten Rechner auszuwählen.

Definition 8: Echtzeitdatenverarbeitung
~ muss von der Datenaufnahme (Input), über die Datenverarbeitung bis hin zur Datenausgabe (Output) stets die zeitlichen Anforderungen, die von realen Ereignissen bestimmt werden, erfüllen.

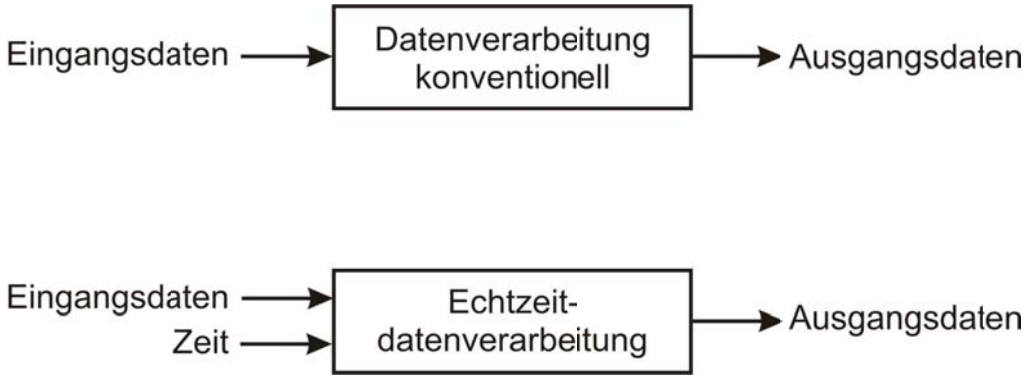


Abbildung 7: Zeit als zusätzliche Eingangsgröße

Die Hardware, die in Echtzeitsystemen eingesetzt wird, zeichnet sich durch hohen Integrationsgrad und Kompaktheit aus. Zum Teil wird die Hardware bereits in das (technische) Prozesssystem eingebaut. Hierfür hat sich der Begriff **embedded systems** eingebürgert.

2.1.7. harte Echtzeitsysteme

Hiermit sind Systeme wie z.B. Flugzeuge, Kraftwerke oder Werkzeugmaschinen gemeint, die harte zeitliche Schranken vorgeben. Die entsprechende Echtzeitdatenverarbeitung muss in jedem Fall sicherstellen, dass fest vorgegebene deadlines eingehalten werden, da es sonst zu schwerwiegenden Systemausfällen kommen kann. Hier können Verletzungen von Zeitvorgaben nicht toleriert werden. Das Datenverarbeitungssystem muss höchste Anforderungen an die Rechtzeitigkeit bei unter Umständen sehr hohem Datendurchsatz erfüllen können. Derartige Echtzeitsysteme müssen in der Regel 100% deterministisch arbeiten.

2.1.8. weiche Echtzeitsysteme

Hierunter fallen Systeme, die prinzipiell als Echtzeitsysteme zu betrachten sind, die aber sehr dehnbare Zeitlimits besitzen. Ein typisches Beispiel sind Bankterminals.
Dort steht der Datenverarbeitung der Mensch als technischer Prozess gegenüber. Da der Mensch aufgrund seiner Sinneswahrnehmung generell sehr langsam reagiert und Verzögerungen selbst im Sekundenbereich gut verkraften kann, sind die Anforderungen an das Datenverarbeitungssystem sehr gering. Die Erfüllung der Rechtzeitigkeit ist meist mit der unmittelbaren Kundenakzeptanz verbunden.

Das entsprechende Echtzeitsystem muss aber trotzdem deterministisch arbeiten, wobei die vorhersehbaren zeitlichen Grenzen sehr große Toleranzen aufweisen.

2.2. Automatisierung von technischen Prozessen

Die Automatisierungstechnik hat die Aufgabe, technologische Prozesse mit Automatisierungseinrichtungen zu steuern und zu überwachen. Die allgemeinste Darstellung bezieht sich auf den allgemeinen Regelkreis, der die Steuereinrichtung (Automatisierungseinrichtung, in der Regel ein Rechnersystem) mit der Steuerstrecke (technologischer Prozess) verbindet.

2.2.1. Der Regelkreis

Innerhalb des technologischen Prozesses kommt es zu Umwandlung von Material, Energie und / oder Informationen.

Bei dieser Darstellung, siehe Abbildung 8, wird noch nicht zwischen Steuern und Regeln unterschieden.

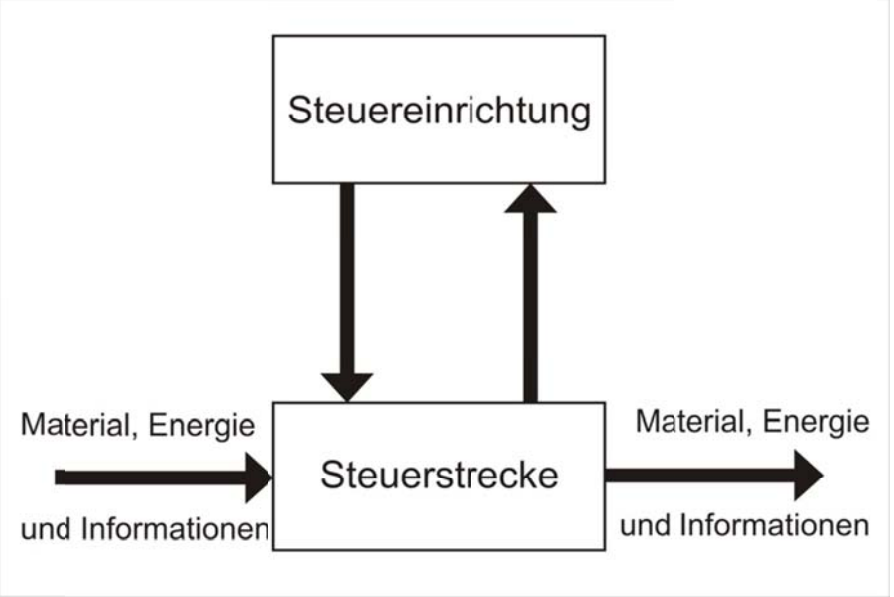


Abbildung 8: Regelkreis

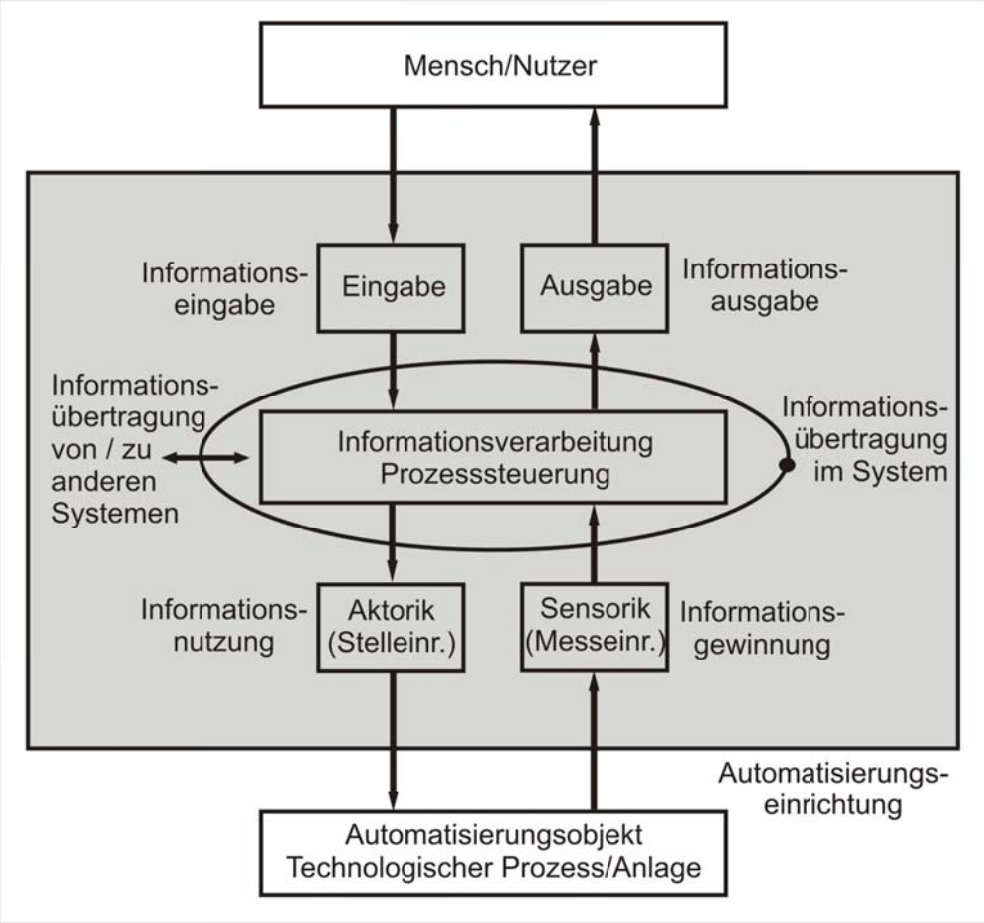


Abbildung 9: Grundprinzip der Automatisierung

Wichtige Aufgabe der Steuereinrichtung (Prozesssteuerung, Automatisierungseinrichtung) ist es, den technologischen Prozess bzw. die Anlage so zu steuern, dass ein größtmöglicher autonomer Ablauf gewährleistet ist.

Abbildung 9 zeigt das Grundprinzip der Automatisierung.

2.2.2. Technologischer Prozess

Definition 9: technologischer Prozess
Ein technologischer Prozess ist die Umformung und / oder der Transport von Material, Energie und / oder Information.

Die Zustandsgrößen technologischer Prozesse können mit technischen Mitteln gemessen, gesteuert und / oder gereget werden.

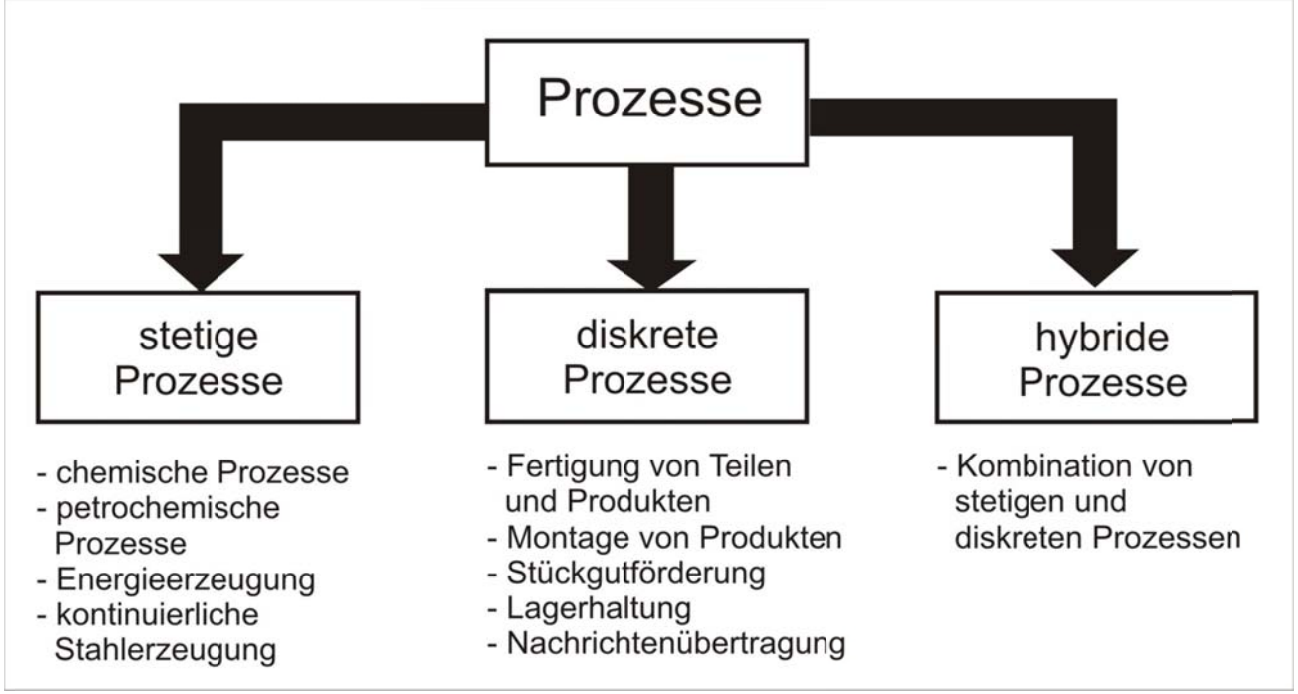


Abbildung 10: Unterteilung von technologischen Prozessen

Man kann stetige, diskrete und hybride Prozesse unterschieden, siehe Abbildung 10. Häufig werden technologische Prozesse durch Automatisierungsanlagen gesteuert.

2.2.3. Steuerung und Regelung

Prinzipiell lassen sich Steuerungssysteme mit offener Wirkungskette und Steuerungssysteme mit geschlossener Wirkungskette unterscheiden. In Abbildung 11 ist die prinzipielle Struktur einer Steuerung mit offener Wirkungskette dargestellt.

Definition 10: Steuerung

Die Steuerung ist ein Vorgang in einem abgegrenzten System, bei dem eine oder mehrere Größen als Eingangsgrößen w(t) andere Größen als Ausgangsgrößen x(t) aufgrund der dem System eigenen Gesetzmäßigkeiten beeinflussen.

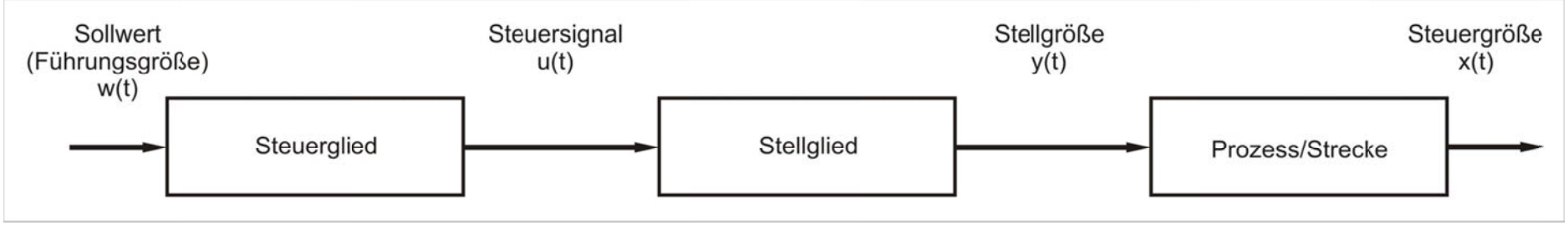


Abbildung 11: Wirkungskette einer Steuerung

Bei der Steuerung mit offener Wirkungskette ist keine Rückkopplung der Prozessgrößen x(t) vorhanden. Das Steuerglied (die Automatisierungseinrichtung) berechnet das Steuersignal u(t) ohne Kenntnis des aktuellen

Wertes der Ausgangsgröße x(t). Für die Berechnung muss ein Modell der Strecke bekannt sein, welches die Zusammenhänge zwischen x(t) und y(t) beschreibt.

Definition 11: Regelung

Die Regelung ist ein Vorgang in einem abgegrenzten System, bei dem eine technische oder physikalische Größe, die sogenannte Regelgröße oder der Istwert x(t), fortlaufend erfasst und durch Vergleich ihres Signals mit dem Signal einer anderen von außen vorgegebenen Größe, der Führungsgröße oder dem Sollwert w(t), im Sinne einer Angleichung an die Führungsgröße w(t) beeinflusst wird.

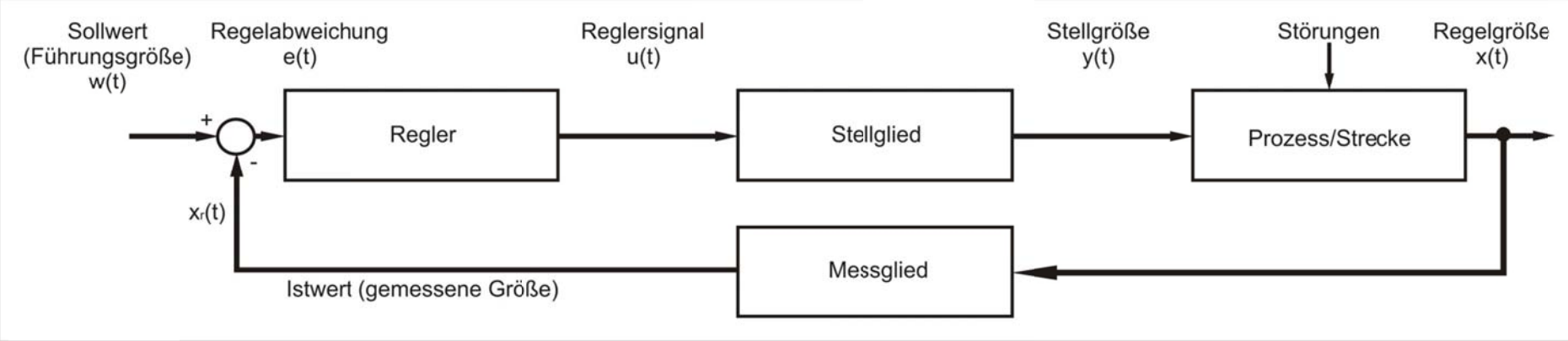


Abbildung 12: Wirkungskette einer Regelung

Zentraler Begriff bei der Betrachtung von Echtzeitsystemen bzw. Echtzeitbetriebssystemen ist der Begiff der Echtzeit.

In Abbildung 6 ist der Zusammenhang zwischen den Regelkreis, siehe Abbildung 8, mit den Signalen x_i und y_i im Vergleich zum Zeitverhalten dargestellt.

2.3. Ergänzungen der Architektur für Echtzeitsysteme

Für den Betrieb von Rechnersystemen als Echtzeitsysteme sind einige weiterführende Hardwarekomponenten notwendig. Grundsätzlich bauen die Rechnersysteme für den Echtzeitbetrieb auf den vorhergenannten Komponenten auf und werden durch folgend aufgeführte Komponenten erweitert.

2.3.1. Mikrocontroller

Mikrocontroller sind spezielle Mikrorechner auf einem Chip, die auf spezifische Anwendungsfälle zugeschnitten sind. Meist sind dies Steuerungs- und / oder Kommunikationsaufgaben, die einmal programmiert und dann für die Lebensdauer des Mikrocontrollers auf diesem ausgeführt werden. Die Anwendungsfelder sind hierbei sehr breit gesteut und reichen vom Haushalt über die Kfz-Technik und Medizintechnik bis hin zur Automatisierung.

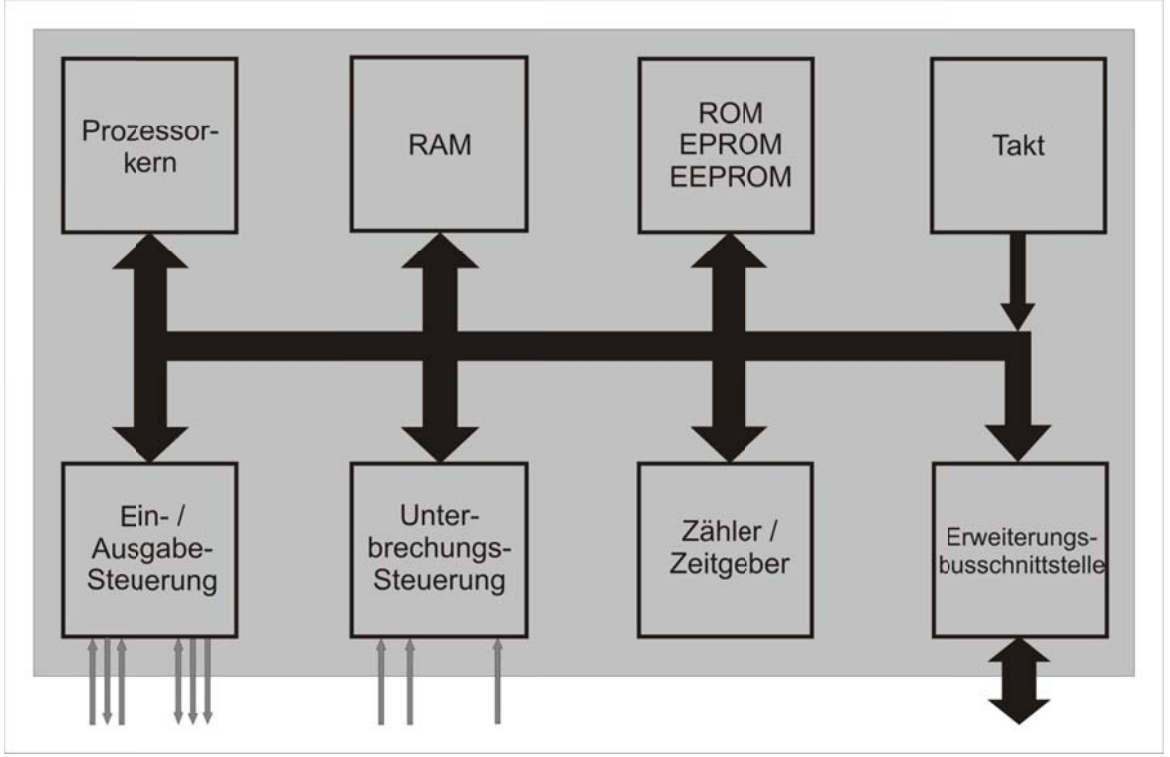


Abbildung 13: Prinzipieller Aufbau eines Mikrocontrollers

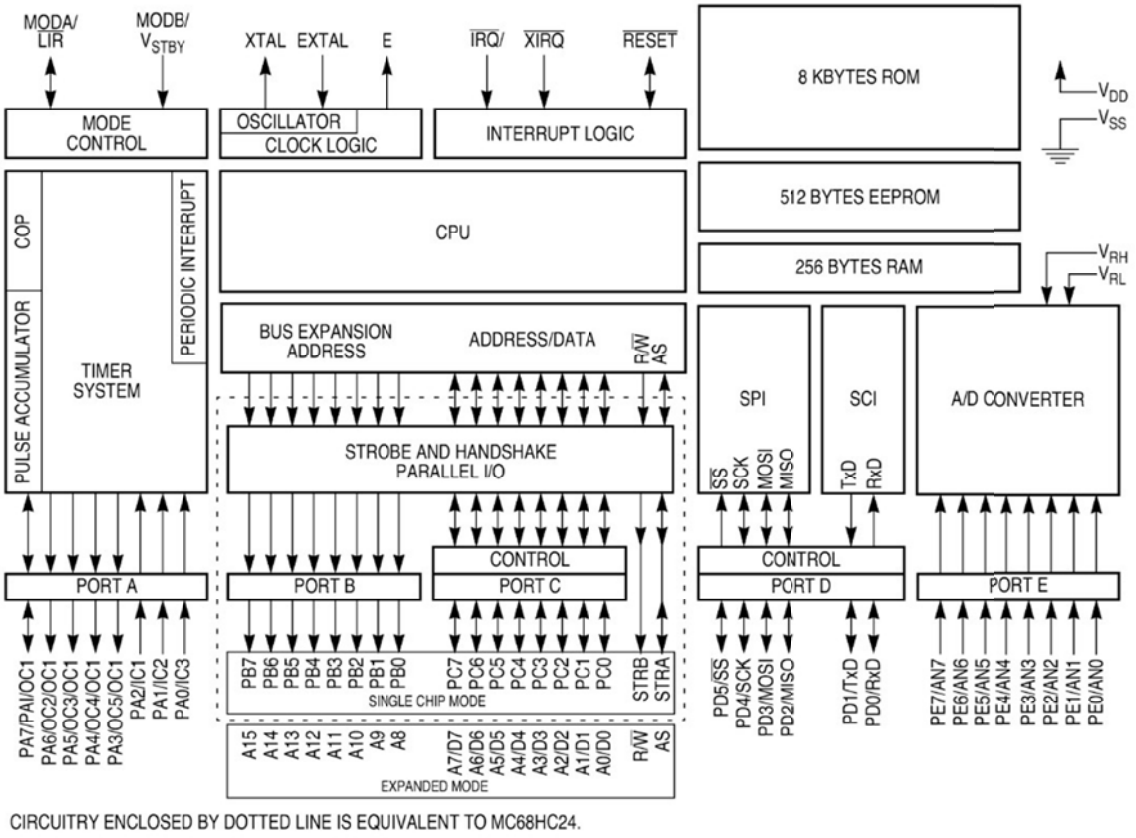


Abbildung 14: Blockschaltbild des Mikrocontrollers 68HC24

In vielen dieser Anwendungsbereiche müssen Zeitbedingungen eingehalten werden, sei es bei den vielen elektronischen Helfern im Fahrzeug wie etwa ABS und ESP oder bei der Steuerung von Robotern in der Automatisierung. Mikrocontroller sind daher wichtige und häufige Bestandteile von Echtzeitsystemen.

Einfach gesagt kann ein Mikrocontroller als ein Ein-Chip-Mikrorechner mit speziell für Steuerungs- und / oder Kommunikationsaufgaben zugeschnittener Peripherie betrachtet werden.

Abbildung 13 gibt einen Überblick.

Als Beispiel für einen Mikrocontroller sei der der Firma Motorola 68HC24, siehe Blockschaltbild in Abbildung 14 dargestellt.

2.3.1.1. Zähler und Zeitgeber

Für den Einsatz von Mikrocontrollern im Echtzeitbereich stellen Zähler und Zeitgeber wichtige Komponenten dar. Hiermit lassen sich eine Vielzahl von mehr oder minder umfangreichen Aufgaben lösen. Während beispielsweise das einfache Zählen von Ereignissen oder das Messen von Zeiten jeweils nur eine Zähler bzw. einen Zeitgeber erfordern, werden für komplexe Aufgaben wie Schrittmotorensteuerungen und Fequenz,- Drehzahl- oder Pulsweitenmessungen mehrere dieser Einheiten gleichzeitig benötigt. Typischerweise werden in Echtzeitsystemen mehrere Einheiten, zumeist in Mikrocontrollern angesiedelt, verwendet, siehe Abbildung 15.

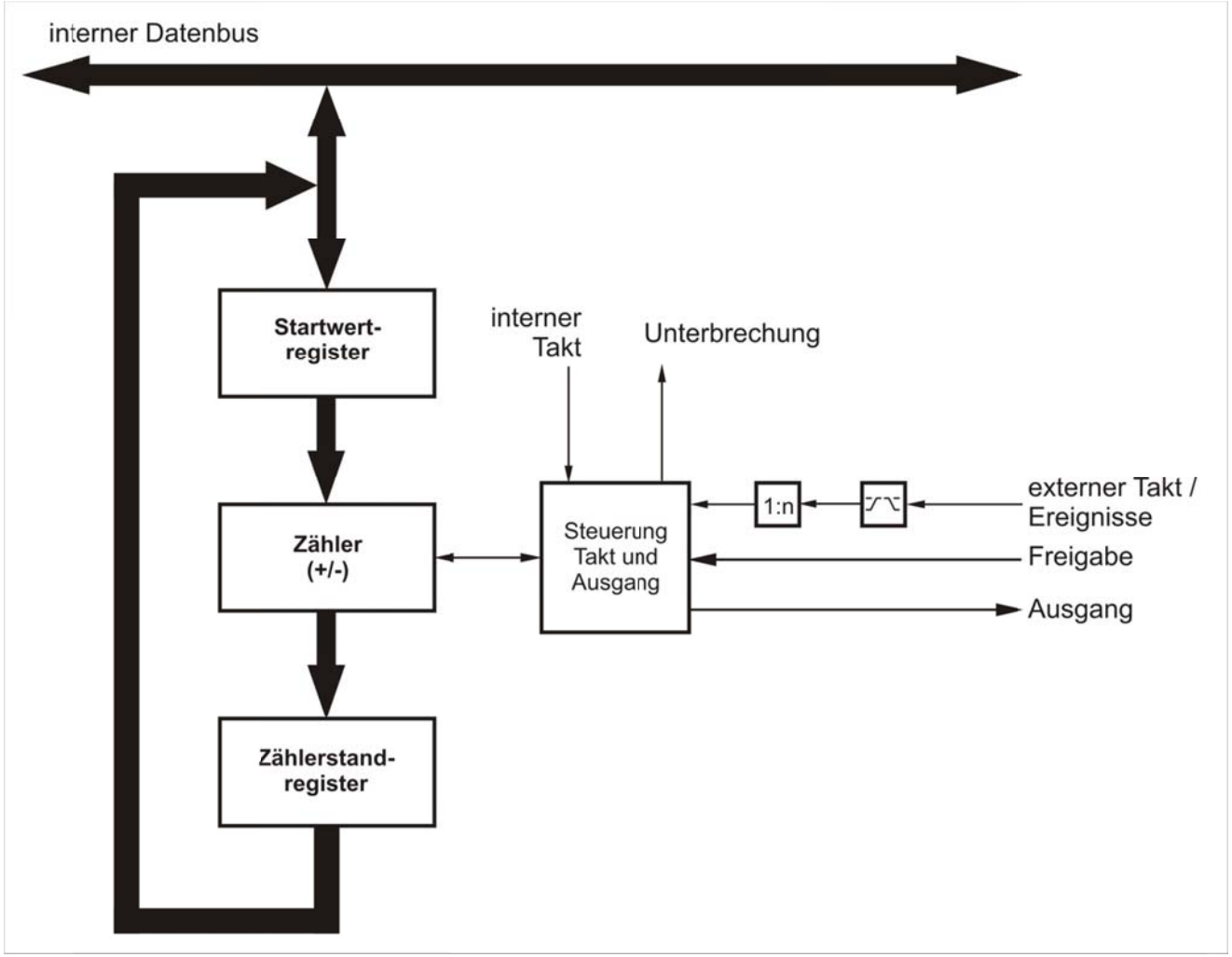


Abbildung 15: Zähler- / Zeitgebereinheit

2.3.1.2. Watchdogs

Wachhunde Watchdogs sind ebenfalls im Echtzeitbetrien gerne eingesetzte Komponenten zur Überwachung der Programmaktivitäten eines Mikrocontrollers. Ihre Aufgabe besteht darin, Programmverklemmungen oder Abstürze zu erkennen und darauf zu reagieren. Hierzu muss das Programm in regelmäßigen Abständen ein „Lebenszeichen“ an den Watchdog senden, z.B. durch Schreiben oder Leseneines bestimmten Ein-/Ausgabekanals. Bleibt das Signal über eine definierte Zeitspanne aus, so geht der Watchdog von einem abnormalen Zustand des Programms aus und leitet eine Gegenmaßnahme ein. Dies besteht im einfachsten Fall aus dem Rücksetzen des Mikrocontrollers und dem damit verbundenen Programmneustart. Der prinipielle Aufbau eines Watchdogs ist in Abbildung 16 gezeigt.

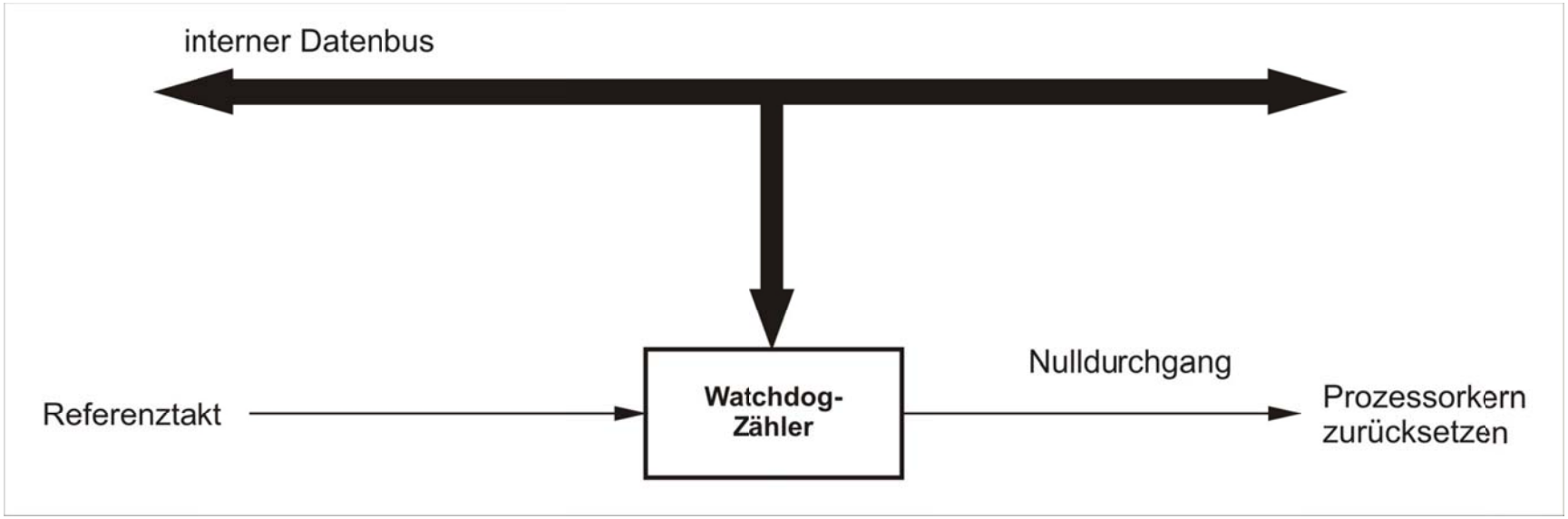


Abbildung 16: Watchdog

2.3.1.3. Serielle und parallele Ein- / Ausgabekanäle

Serielle und parallele Ein- / Ausgabekanäle (IO-Ports) sind die grundlegenden digitalen Schnittstellen eines Mikrocontrollers. Über parallele Ausgabekanäle können eine bestimmte Anzahl digitaler Signale gleichzeitig gesetzt oder gelöscht werden, z.B. zum Ein- und Ausschalten von peripheren Komponenten. Parallele Eingabekanäle ermöglichen das gleichzeitige Lesen von digitalen Signalen, z.B. zur Erfassung der Zustände von digitalen Sensoren wie etwa Lichtschranken. Die Richtung der parallelen Kanäle ist meist bitweise oder in Bitgruppen programmierbar.

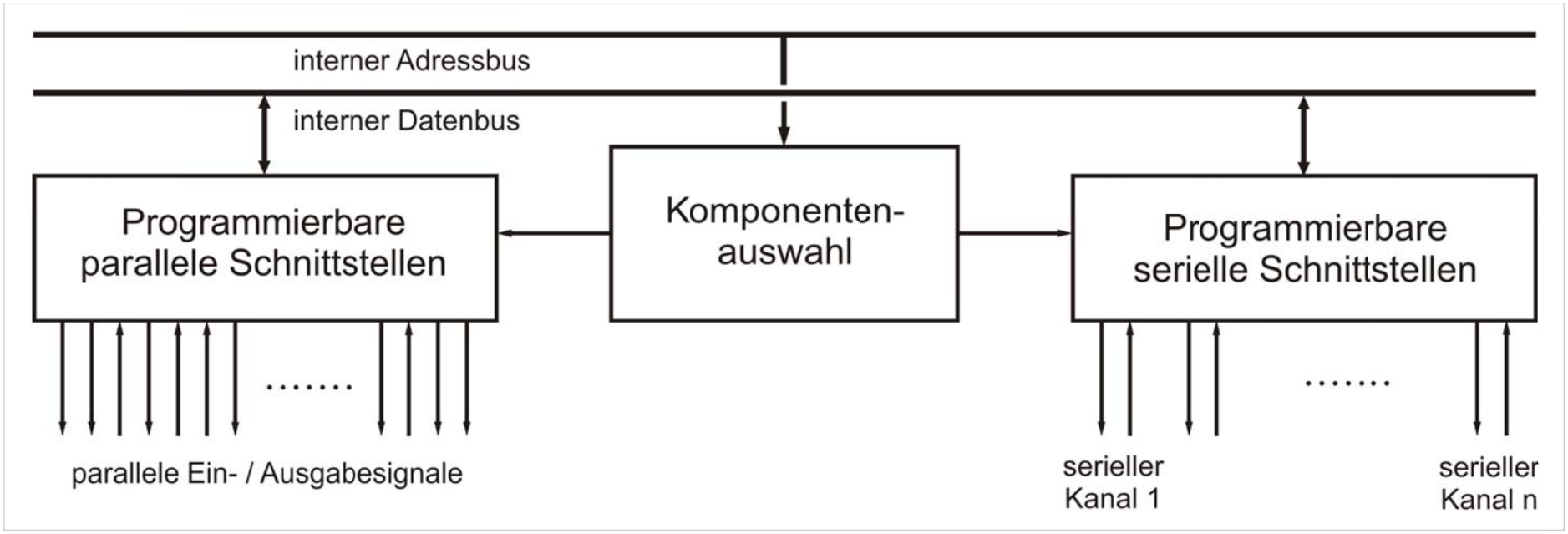


Abbildung 17: Serielle und parallele Ein- / Ausgabekanäle

Serielle Ein- / Ausgabekanäle dienen der Kommunikation zwischen Mikrocontrollern und Peripherie (oder anderer Mikrocontroller) unter Verwendung möglichst weniger Leitungen. Die Daten werden hierzu nacheinander (seriell) über eine Leitung verschickt. Je nach Art und Weise der Synchronisation zwischen Sender und Empfänger unterscheidet man synchrone und asynchrone serielle Kanäle.

2.3.1.4. Echtzeitkanäle

Echtzeitkanäle (Real Time Ports) sind eine für Echtzeitsysteme nützliche Erweiterung von parallelen Ein- / Ausgabekanälen. Hierbei wird ein paralleler Kanal mit einem Zeitgeber gekoppelt. Bei einem normalen Ein- / Ausgabekanal ist der Zeitpunkt einer Ein- oder Ausgabe durch das Programm bestimmt. Eine Ein- oder Ausgabe erfolgt, wenn der entsprechende Ein- / Ausgabebefehl im Programm ausgeführt wird. Da durch verschiedene Ereignisse die Ausführungszeit eines Programms verzögert werden kann, verzögert sich somit auch die Ein- oder Ausgabe. Bei periodischen Abläufen führt die zu unregelmäßigem Ein- / Ausgabezeitverhalten, man spricht von einem ***Jitter***.

Bei Echtzeitkanälen wird der exakte Zeitpunkt der Ein- oder Ausgabe nicht vom Programm, sondern von einem Zeitgeber gesteuert. Hierdurch lassen sich Unregelmäßigkeiten innerhalb eines gewissen Rahmens beseitigen und Jitter vermeiden.

2.3.1.5. AD- / DA-Wandler

Analog / Digital-Wandler (AD-Wandler) und Digital / Analog-Wandler (DA-Wandler) bilden die grundlegenden analogen Schnittstellen eines Mikrocontrollers. AD-Wandler wandeln anliegende elektrische Analog-Signale, z.B. eine von einem Temperatursensor erzeugte der Temperatur proportionale Spannung, in vom Mikrocontroler verarbeitbare digitale Werte um.

DA-Wandler überführen in ungekehrter Richtung digitale Werte in entsprechende elektrische Analog-Signale.

Die Wandlung selbst ist eine lineare Abbildung zwischen einem binären Wert, meist einer Dualzahl, und einer analogen elektrischen Größe, meist einer Spannung. Besitzt der binäre Wert eine Breite von ***n*** Bits, so wird der analoge Wertebereich der elektrischen Größe in 2^n gleich große Abschnitte unterteilt. Man spricht von einer ***n-Bit Wandlung*** bzw. von einem ***n-Bit Wandler***.

Die Abbildung zwischen der Dualzahl und der elektrischen Größe (Spannung) läßst sich durch eine Treppenfunktion darstellen. Dabei ist der kleinste Schritt der DualZahl gleich 1, der kleinste Spannungsschritt beträgt:

$$U_{LSB} = (U_{\max} - U_{\min}) / 2^n$$

U_{max} und U_{min} sind die maximalen bzw. minimalen Spannungen des analogen Wertebereiches. LSB steht für ***L**east **S**ignificant **B**it*, das niederwertigste Bit der Wandlung.

Die Wandlungsfunktion einer Digital / Analog-Wandlung der Dualzahl Z in eine Spannung U lautet somit:

$$U = (Z \bullet U_{LSB}) + U_{\min}$$

Umgekehrt ergibt sich die Wandlungsfunktion einer Analog / Digitalwandlung der Spannung U in eine Dualzahl Z zu:

$$Z = (U - U_{\min}) / U_{LSB}$$

U_{LSB} definiert somit die theoretisch maximale Auflösung der Wandlung. Werden beispielsweise **U_{max} = 5 Volt**, **U_{min} = 0 Volt** und **n = 12 Bit** gewählt, so ergibt sich eine maximale Auflösung von **1,221 Millivolt**.

2.3.2. Signalprozessoren

Signalprozessoren sind spezielle Mikrorechnerarchitekturen für die Verarbeitung analoger Signale, z.B. im Audio- und Video-Bereich. Die Anwendungsfelder reichen von digitalen Filtern über Spektralanalyse, Spracherkennung, Sprach- und Bildkompression, Signalaufbereitung bis zur Verschlüsselungstechnik. Signalprozessoren sind daher für eine effitiente Verarbeitung analoger Signale konzipiert. Hierzu besitzen sie eine spezielle, für die Signalverarbeitung optimierte Hochleistungsarithmetik. Darüber hinaus verfügen Sie über ein hohes Maß an Parallelität. Diese Parallelität steht weitgehend unter der Kontrolle des Programmierers. Dies ist ein Enterschied zu Mikroprozessoren und Mikrocontrollern, bei denen das Steuerwerk die Parallelität kontrolliert.

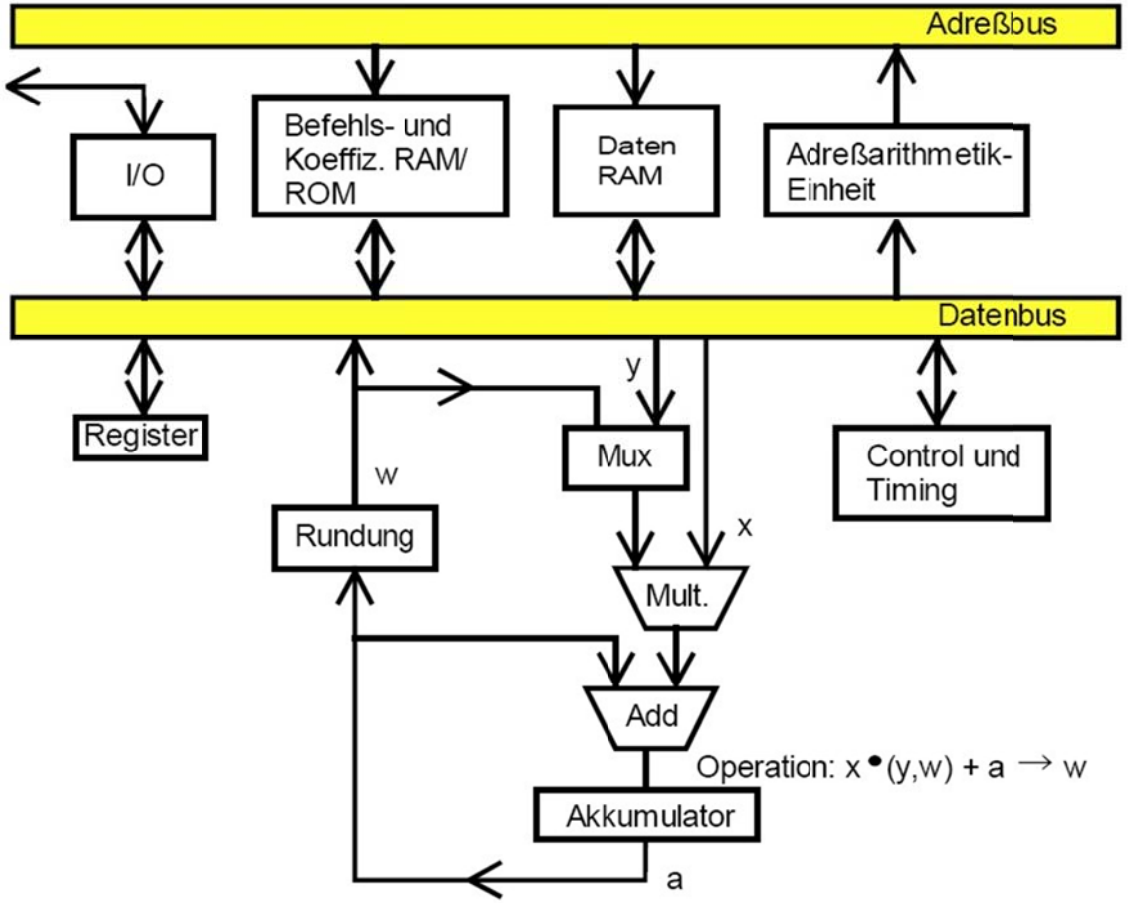


Abbildung 18: Grundlegender Aufbau eines Signalprozessors DSP

Die wesentlichen Eigenschaften von Signalprozessoren lassen sich wie folgt zusammenfassen:

a) Konsequente Harvard-Architektur, d.h. Trennung von Daten- und Befehlsspeicher.
b) Hochgradiges Pipilining sowohl zur Befehlsausführung wie für Rechenoperationen.
c) Oft mehrere Datenbusse zum parallelen Transport von Operanden zum Rechenwerk.
d) Hochleistungsarithmetik, optimiert für aufeinanderfolgende Multiplikationen und Additionen.
e) Hohe, benutzerkontrollierbare Parallelität.
f) Ggf. spezielle Peripherie zur Signalverarbeitung, z.B. Schnittstellen zur Digital / Analog- oder Analog / Digital-Wandlung.

Mikroprozessor	Signalprozessor
Ausgelegt für Betriebssysteme (Unix), viel Speicher, MMU	keine Betriebssystemunterstützung, keine MMU
von-Neumann/modif. Harvard Arch.	Harvard Architektur
64-Bit Arithmetik	32-Bit Arithmetik
Takt bis 200 MHz	< 60 MHz
Kein DMA	DMA on chip
Langsamer Taskwechsel	Schnellerer Taskwechsel
Standard Interruptbehandlung	Schnellere Interruptbehandlung
Pipeline Gleitkomma-Einheit	1-Zyklus Gleitkomma Multiplik.
z.T. Grafikeinheit bzw. -befehle	nicht vorhanden
Gleitk.-Multipl. in mehreren Zyklen	Multipl.& Addition in 1 Zyklus
Universeller Befehlssatz	Befehle optim. f. Arithm., spez. FFT

Tabelle 15: Unterschiede zwischen Mikroprozessoren und Signalprozessoren

2.3.3. Bussysteme für Echtzeitsysteme

Wie in der Vorlesung Betriebssysteme I dargestellt, werden die einzelnen Komponenten und Baugruppen eines Rechnersystems über verschieden Bussysteme miteinander verbunden. Dabei lassen sich allgemein serielle Bussysteme und Parallelbusse unterscheiden. Bei Letzterem werden Daten, Adressen und Steuersignale über parallele Leitungen übertragen. Sie sind das Verbindungsglied zwischen Mikroprozessoren, Speicher und Ein- / Ausgabeeinheiten. Damit verbinden sie die Einzelkomponenten zu einem System und werden daher als Systembus bezeichnet. Systembusse bestimmen in wesentlichen Teilen das Zeitverhalten des Gesamtsystems. Ein noch so leistungsfähiger Mikroprozessor ist nutzlos, wenn ein langsamer Bus ihn ausbremst.

Im Gegensatz zu oft seriellen Peripheriebussen, bei denen die Kosten im Vordergrund stehen, sind Parallelbusse deshalb auf eine möglichst hohe Übertragungsrate ausgelegt. Dabei ist von Vorteil, dass die zu überbrückenden Entfernungen sehr kurz sind, sie liegen im Allgemeinen unter 50 cm.

Eine möglichst hohe Datenrate ist jedoch nur ein Aspekt, der für ein Echtzeitsystem viel wichtigere Teil ist die zeitliche Vorhersagbarkeit. Um dies zu gewährleisten, sind spezielle Maßnahmen erforderlich.

Dazu sollen zunächst einige Überlegungen zum Einsatz von Bussystemen im industriellen Umfeld angestellt werden:

- Anlagen arbeiten kontinuierlich (möglichst 24 Standen pro Tag, 365 Tage pro Jahr).
- Sie müssen teilweise unter extremen Umgebungsbedingungen funktioniern:
  - Tempratur,
  - Feuchtigkeit,
  - Aggressive Gase oder Flüssigkeiten
- Erhebliche Störungen beeinflussen die Ein- / Ausgabesignale (elektromagnetische Störungen, Kabelbruch, Kurzschluss).

- Komponenten der Anlagen oder des Rechnersystems können ausfallen.
- Fehlbedienung

Fazit:

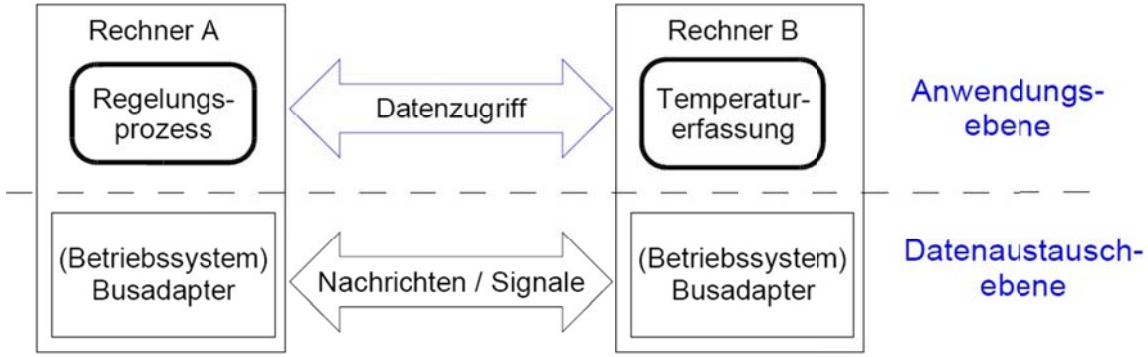
a) Bussysteme werden dort eingesetzt, wo es auf Flexibilität hinsichtlich der Anpassung von Systemen auf sich ändernde bzw. ähnliche Aufgaben ankommt.

b) Bussysteme sind als flexible Baukastensysteme konzipiert mit standardisierten Kommunikationsabläufen zwischen den Baugruppen.

c) Bussysteme in der Automatisierungstechnik müssen Echtzeitanforderungen genügen, robust gegenüber Störungen sein und unter widrigen Umgebungsbedingungen arbeiten.

Im Folgenden werden Beispiele für Bussysteme in Echtzeitsystemen betrachtet.

2.3.3.1. Technische Grundlagen



Der Grundansatz für das Verständnis von Bussystemen ist in der Rechnerkommunikation zu sehen. Dabei wird bereits deutlich, dass die Kommunikation in mehreren Ebenen stattfindet, siehe Abbildung 19.

Abbildung 19: Rechnerkommunikation

Ein weitergehender allgemeiner Ansatz wird durch das OSI-Modell zur Standardisierung der Kommunikation zwischen verschiedenen Rechnern beschrieben. Dieser Ansatz muss auch für die Kommunikation in Bussystemen gelten, siehe Abbildung 20.

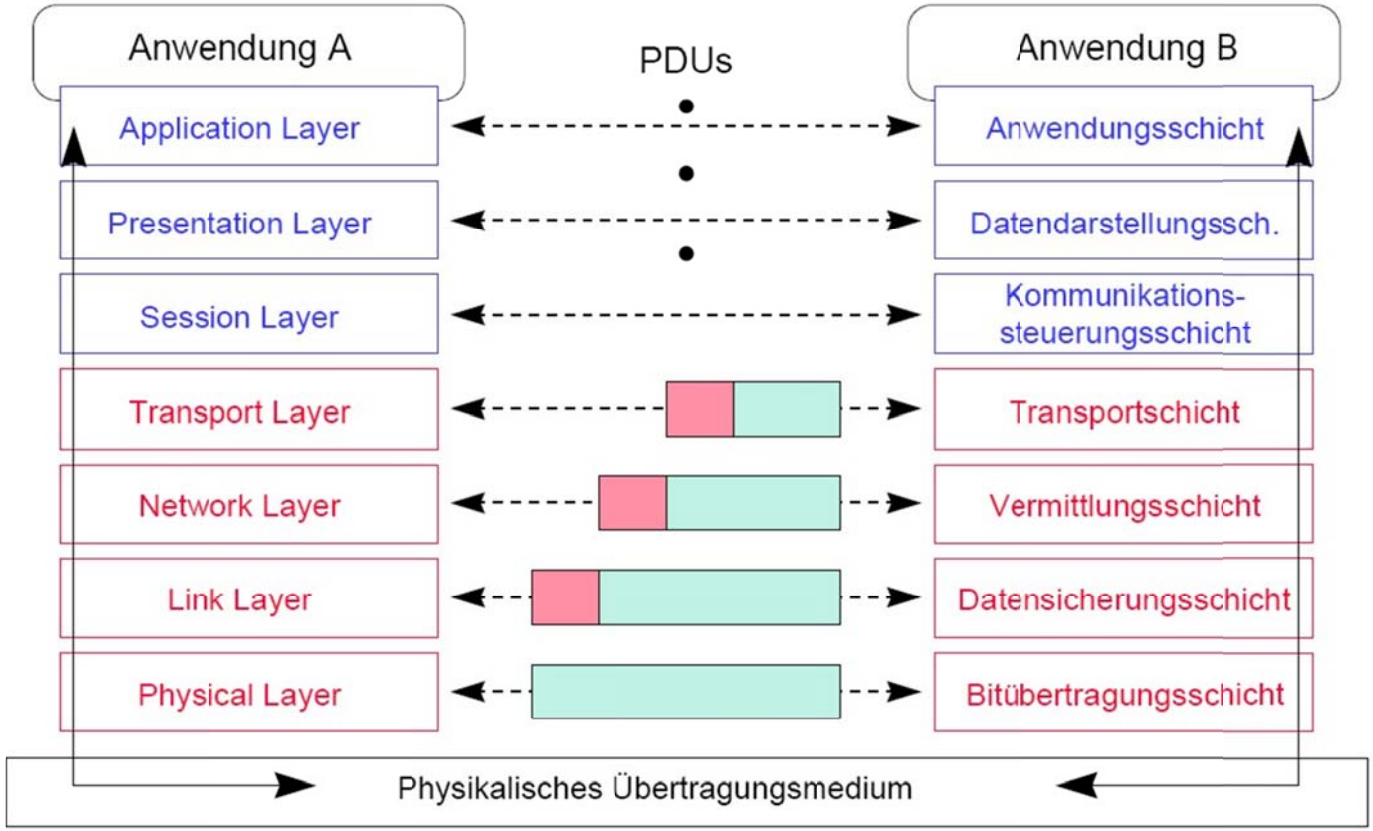


Abbildung 20: OSI-Modell

Die Frage, die sich dabei stellt, bezieht sich auf die notwendigen Schichten des allgemeinen OSI-Modells bezogen auf Bussysteme. Es bleiben interessanterweise von dem 7-Ebenen-Model nur die untersten 3 Schichten, sowie die Anwendungsschicht, die in Bussystemen relevant sind, siehe Abbildung 21 und Abbildung 22.

Anwendungsschicht	Protokolle für häufig verwendete Anwendungen (Zugriff auf Prozess-Informationen, Management von Stationen, ...)
Darstellungsschicht	Kodierung und Präsentation der Daten (Zeichenkodes, Geometrie für Textdarstellung, Verschlüsselung ...)
Sitzungsschicht	Ablauf der logischen Kommunikation (Verhaltensregeln, Wer spricht wann?)
Transportschicht	Gewährleistung einer Ende-zu-Ende-Verbindung in geforderter Güte
Vermittlungsschicht	Bestimmung des Weges der Nachrichten (Pakete) im Netz
Datensicherungsschicht	sichere Übertragung von Daten (frames) zwischen benachbarten Stationen
Bitübertragungsschicht	Übertragung von Bits - „Bitstrom“

Abbildung 21: in Bussystemen benötigte Schichten des OSI-Modells

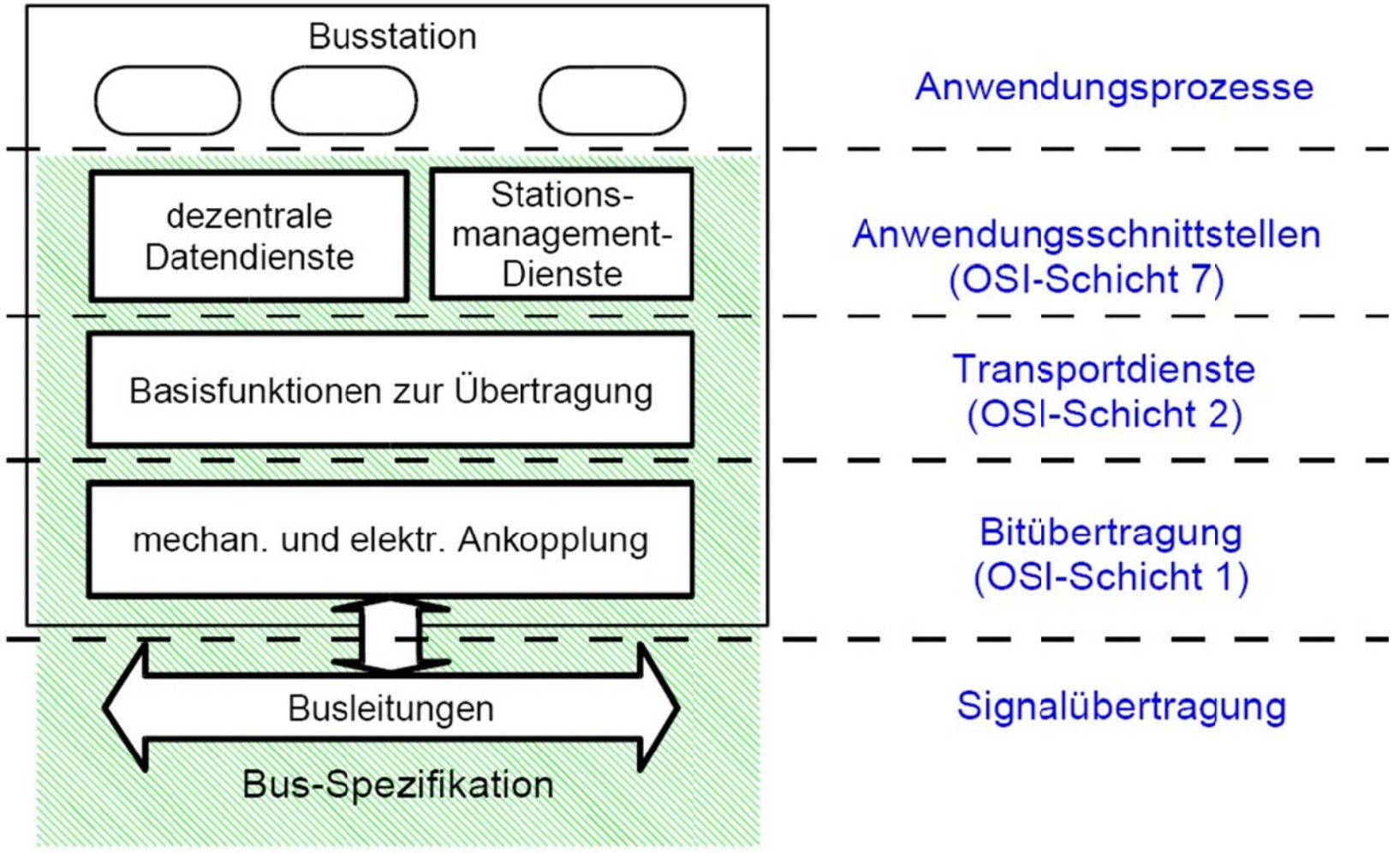


Abbildung 22: Kommunikation in Bussystemen

2.3.3.2. Parallele Bussysteme

Eine Eigenschaft von Systembussen ist die Übertragungsart. Hier wird zwischen Multiplex- und Nicht-Multiplex-Betrieb unterschieden. Bei diesemn teilen sich Bussignale eine Busleitung. Eine der häufigsten Varianten ist der Daten- / Adressmultiplex, bei dem Daten- und Adresssignale in verschiedenen Takten über dieselben Leitungen übertragen werden.

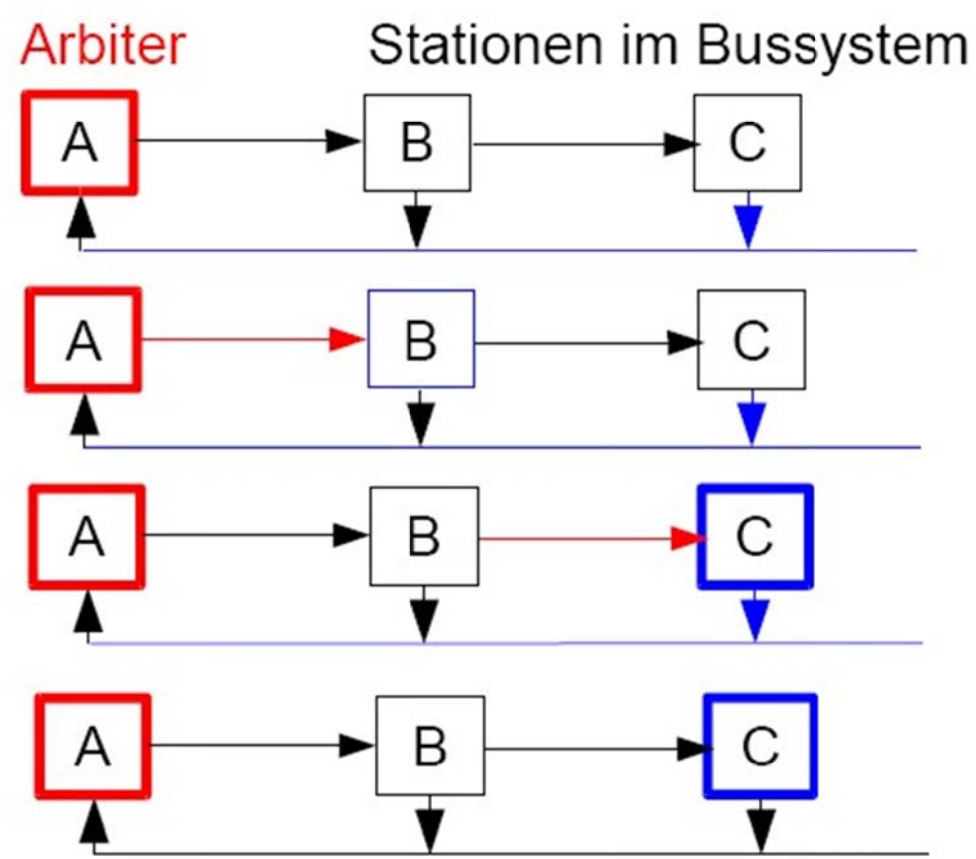


Abbildung 23: Ablauf der Buszuteilung

Weiterhin unterscheiden sich die verschiedenen Bussysteme in der Anzahl der Master und Slaves. Während in einfachen Bussystemen nur ein Master existiert, kann es in komplexen Bussystemen mehrere Master geben, die nicht alle immer aktiv sein können. Für die Buszuteilung muss es entsprechende Verfahren geben, wie der Zugriff der einzelnen Master auf den Bus erfolgen kann. In Abbildung 23 ist ein Verfahren mit einem Arbiter dargestellt. Der Zugriff durch Master C erfolgt dabei in vier Schritten.

2.3.3.2.1. VMEbus (Versa Module Europa)

Der VMEbus (Versa Module Europa) ist eine Weiterentwicklung des VersalBus, den die Firma Motorola für die 680XX-Prozessorfamilie definiert hatte.

Obwohl er für eine Prozessorfamilie definiert wurde, gilt der VMEbus als prozessorunabhängig und war lange Zeit dominierend in Bussystemen für Automatisierungsrechner. Er wird jedoch zunehmend durch den PCI-Bus, vorwiegend den compactPCI-Bus verdrängt, da dieser höhere Übertragungsraten ermöglicht. Die Eigenschaften des VMEbus lassen sich wie folgt zusammenfassen:

- prozessorunabhängig,
- signalorientiert,
- asynchron,
- nicht gemultiplext,
- 32-Bit-Bus mit 64-Bit Burst,
- mulimasterfähig,
- auf sieben Ebenen interrupt-fähig und
- echtzeitfähig.

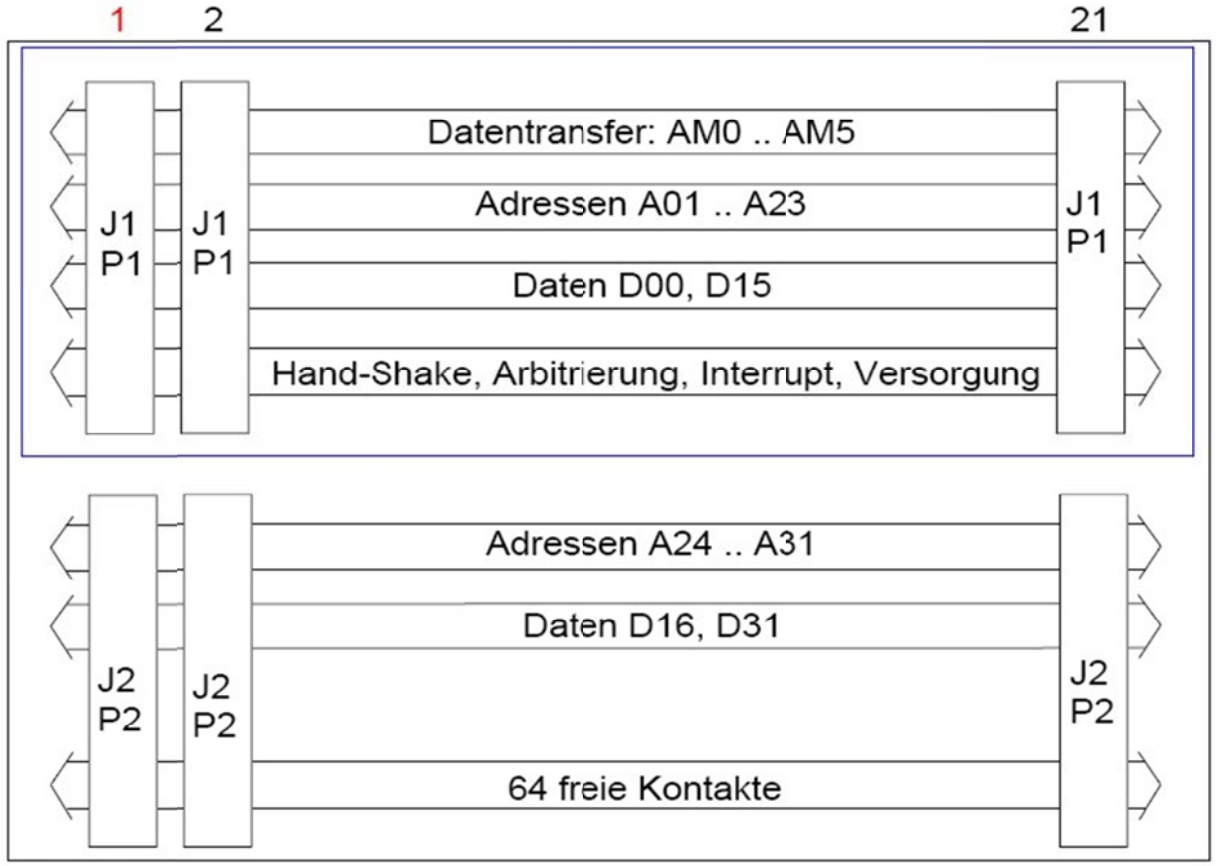
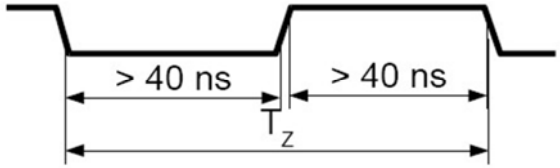


Abbildung 24: Ankopplung des VMEbusses

Zeitvorgaben des VMEbus-Standards:

- Signal zur Anzeige stabiler Adressen (AS*)
  muss mindestens 40 ns H-Pegel bzw. L-Pegel haben bis zum nächsten Pegelwechsel!



- Verzögerungszeiten für Treiber und Empfänger betragen ca. 10 ns

$$T_Z \geqslant 2 \cdot T_{AS} + 2 \cdot T_V = 2 \cdot 40\,ns + 2 \cdot 10\,ns$$

$$T_Z \geqslant 100\,ns$$

Transferrate: 10 MHz

- "Datenbusbreite" ist variabel (D8, D16, D32)

$$D_{max} = \frac{I}{T_Z} = \frac{4\,Byte}{100\,ns} = 40\,MByte/s$$

praktisch erreichbare Raten:

- 30 MByte/s (bei optimalem Hardware- und Software-Design)

Abbildung 25: Maximal Übertragungsrate des VMEbusses

Die Datentransferraten am VMEbus werden durch die Zeitbedingungen für die Handshake-Signale bestimmt. Entscheiden ist, dass die Zeit zwischen zwei Aktivierungen TZ nicht kürzer als 100 ns werden darf. Damit ergeben sich bei einem 32-Bit-Transfer maximale Datenübertragungsraten von 40 Mbyte/s, siehen Abbildung 25. Damit bleibt der VMEbus deutlich hinter dem PCI-Bus zurück.

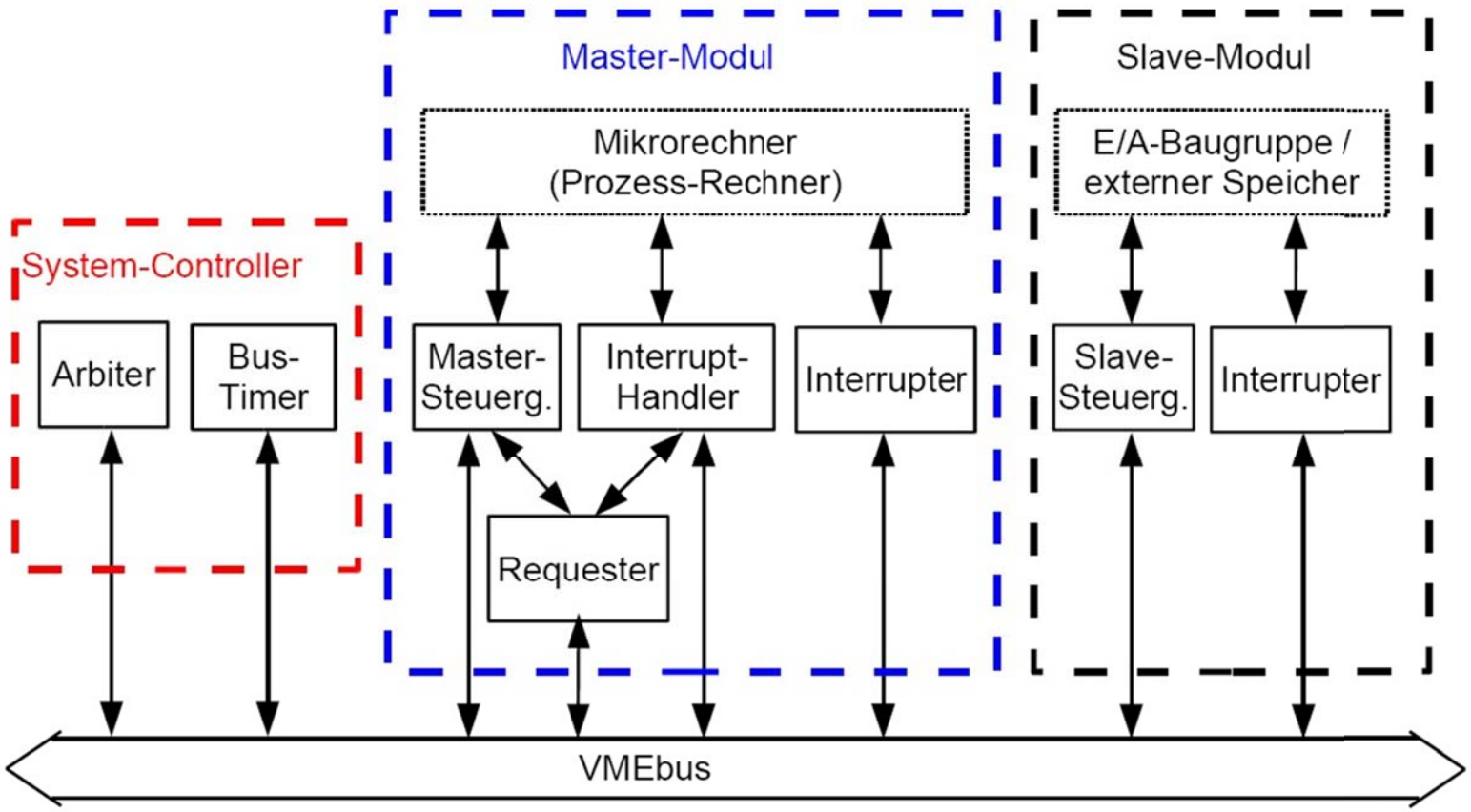


Abbildung 26: Module des VMEbusses

Die grundlegende Architektur des VMEBusses ist in Abbildung 26 dargestellt.

2.3.3.2.2. compactPCI (Peripheral Component Interconnect)

Der PCI-Bus (Peripheral Component Interconnect) ist ein entkoppelter, prozessorunabhängiger Bus, der in einer 32-Bit-Version und einer 64-Bit-Version zur Verfügung steht. Ursprünglich von Intel entwickelt, wird der PCI-Bus heute von der PCI Special Interest Group weiterentwickelt und gepflegt.

mechanisch	- Europa-Karten-Format (einfache = 3U, doppelte = 6U) - Steckverbinder (IEC 917 und IEC 1076-4-101) 2mm-Raster, 5 Reihen mit 47 Kontakten - Befestigung der Module
Bus-Struktur	- max. 8 Steckplätze 1 System-Controller 5 Master- oder Slave-Module („intelligente“ oder einfache) 2 Slave („intelligente“ oder einfache) - 32/64-bit-Module bei 3U-Baugruppen - Zusatz-Busse bei 6U-Baugruppen auf freien Signal-Leitungen
Signale	- Zusätzliche Signale für spezielle Zwecke - manuelles RESET (PRST#) - Betriebsspannungsstatus (DEG#, FAL#) - Systemplatzkennung (SYSEN#) - System-Numerierung (ENUM#) - IDE-Interrupt-Unterstützung (INTP, INTS) - geografische Adressierung
Sonstige	- Modulaustausch bei laufendem Betrieb („Hot-Swap“)

Tabelle 16: Unterschied des compactPCI zum PCI

Die Unterschiede des compactPCI-Busses zum PCI-Bus sind in Tabelle 16 zusammengefasst. Als maximale Datenübertragungsraten sind beim PCI-Bus 266 Mbyte/s bei 32-Bit-Transfer und 533 Mbyte/s bei 64-Bit-Transfer möglich.

Folgende Eigenschaften kennzeichnen den PCI-Bus:

- prozessorunabhängig,
- entkoppelt,
- kommandoorientiert,
- synchron,
- gemultiplext,
- per Software konfigurierbar,
- 32 oder 64 Bit breit,
- burstfähig,
- fehlererkennend,
- multimasterfähig und
- echtzeitfähig.

2.3.3.3. Feldbussysteme

Um die Einordnung von Feldbussystemen zu verstehen, muss die Entwicklung der Systemarchitektur von Prozessrechnern untersucht und dargestellt werden. In einem ersten Schritt kam es zur vertikalen Strukturierung. Der Prozessrechner wurde über einem Bussystem mit den Ein- / Ausgabeeinheiten verbunden. Damit kam es zur effektiven Verkablung und flexibler Peripherie, siehe Abbildung 27.

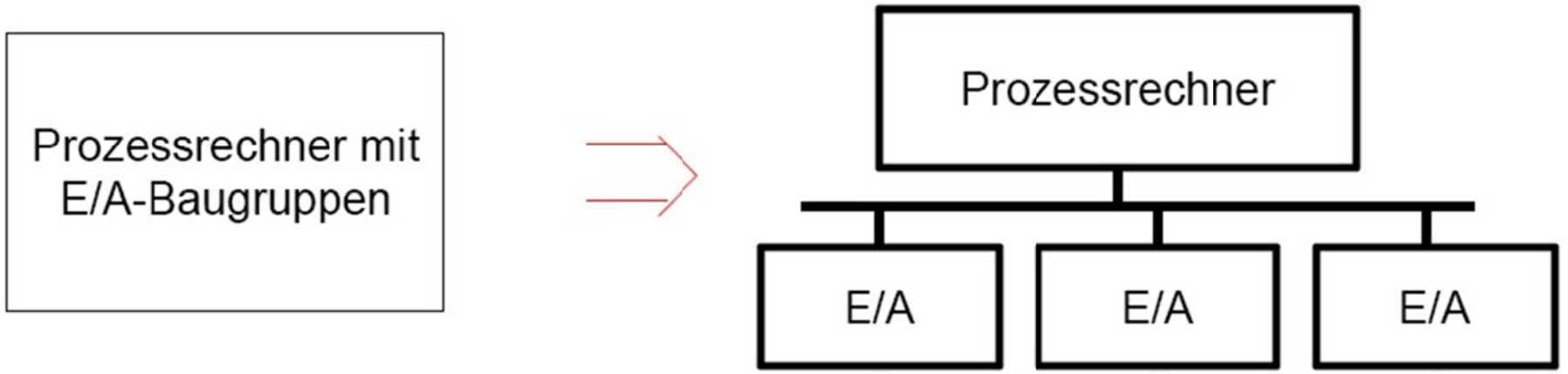


Abbildung 27: 1. Schritt der Entwicklung der Systemarchitektur von Prozessrechnern

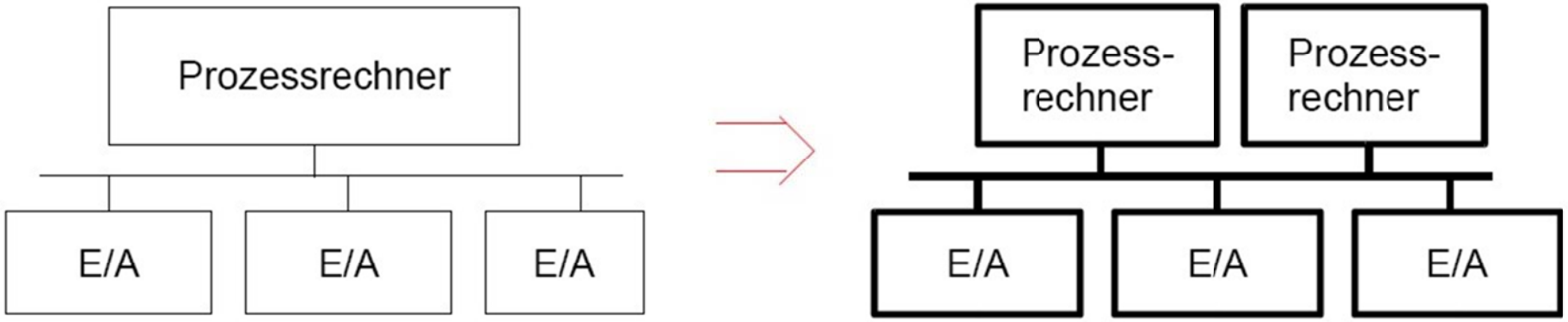


Abbildung 28: 2. Schritt der Entwicklung der Systemarchitektur von Prozessrechnern

Im zweiten Schritt wurde die parallele Verarbeitung eingeführt. D.h., mehrere Prozessrechner waren mit einem gemeinsamen Bussystem mit den einzelnen Ein- / Ausgabeeinheiten verbundnen, siehe Abbildung 28. Es entstanden parallele Bussysteme für den Multiprozessorbetrieb. Als ein Beispiel ist der VMEbus zu nennen.

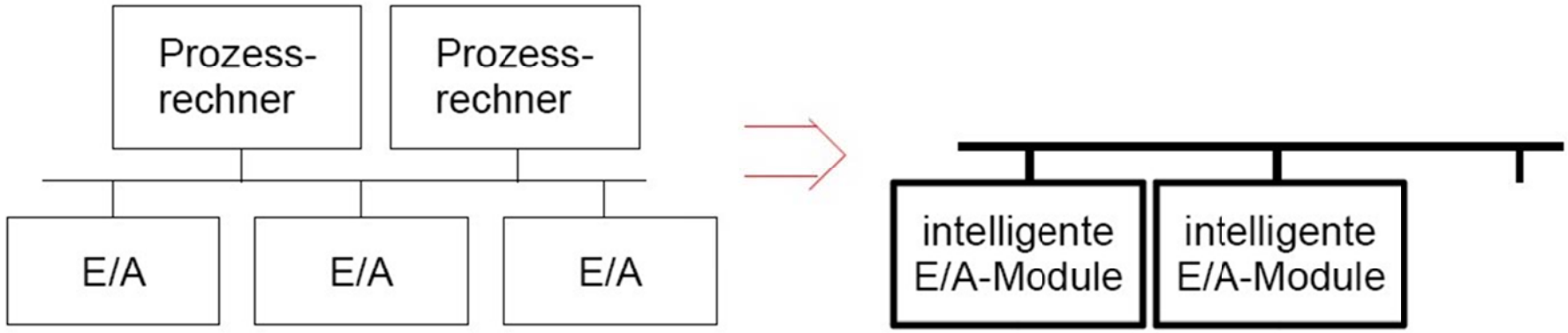


Abbildung 29: 3. Schritt der Entwicklung der Systemarchitektur von Prozessrechnern

Im dritten Schritt, siehe Abbildung 29, kam es wieder zur vollständigen Dezentralisierung. Über sogenannte Feldbussysteme wurden flexible intelligente Ein- / Ausgabeeinheiten (Mikroprozessorrechner) und effektiver Verkabelung miteinander verbunden. Es entstehen offene Kommunikationssysteme, die in der Automatisierung von:

- Industrieanlagen,
- Gebäuden (INSTA-BUS) und
- Fahrzeigen eingesetzt werden.

Die Anforderungen an Feldbussysteme sind in Tabelle 17 aufgelistet.

Echtzeitforderungen	- max. Zugriffsverzögerung - Prioritätssteuerung - periodische Aktionen / aperiodische Ereignisse
Kommunikationsverhalten	- kurze Nachrichten - relativ hohe Datenraten
Kommunikationsmuster	- klassisch: Master-Slave (1 : N) - Trend: Multi-Master (M : N)
typische Anwendungen	- Transport von Prozeßdaten - Diagnose, Parametrierung

Tabelle 17: Randbedingungen für Feldbussysteme

2.3.3.3.1. Profibus (PROcess Fleld BUS)

Der Profibus wurde 1987 als Projekt des BMFT mit 13 Firmen und 5 Instituten entwickelt. 1991 erfolgte die Normierung als DIN 19245 und er ist in die europäische Feldbusnorm EN 50170 teil 3 eingegangen.

Der Profibus ist ein offener Standard, der sich ebenfalls am OSI-Modell orientiert und lizenzfrei ist. Die Koordinierung der Aktivitäten wird durch die PNO Profibus Nutzer Organisation übernommen. Es kommen traditionelle Prinziplien der Kommunikation zum Einsatz, so

- hierarchische Steuerung (Master-Slave) für autonome periodische Prozesse und
- gleichberechtigter Zugang für Master.

Es erfolgt die Standardisierung auch von Anwendungsdiensten.
Es existieren mehrere Varianten des Profibus, siehe Abbildung 30:

- Profibus-FMS (volle Norm – Schichten 1, 2 und 7),
- Profibus-DP (dezentrale Peripherie – auf Geschwindigkeit optimiert) und
- Profibus-PA (eigensichere Übertragungstechnik).

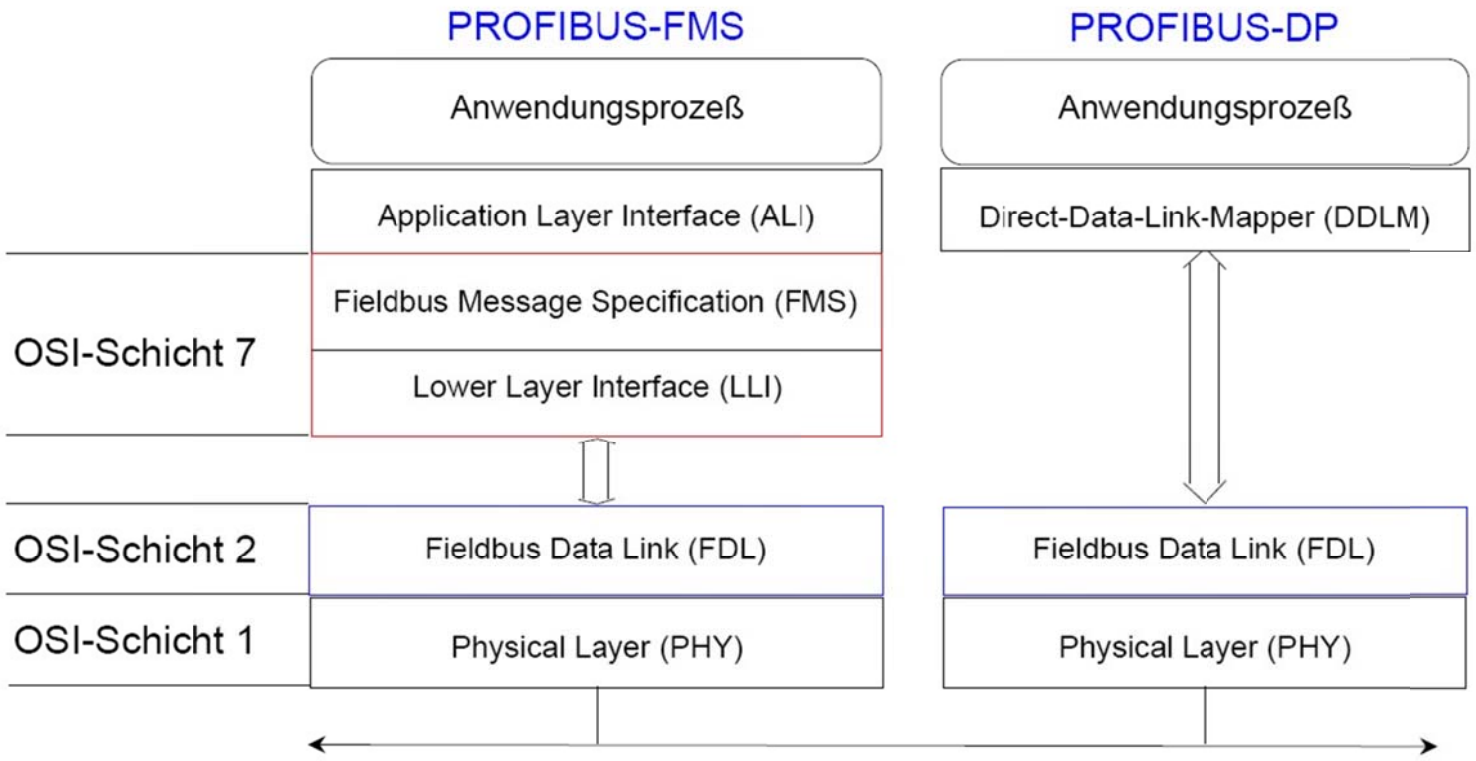


Abbildung 30: Architektur von PROFI-Busstationen

2.3.3.3.2. CAN-Bus (Controller Area Network)

Der CAN-Bus wurde ursprünglich von der Firma Bosch für die Fahrzeugautomatisierung entwickelt (ABS, Motorsteuerung u.ä.). Es gilt das nachrichtenorientierte Prinzip mit Prioritäten für die einzelnen Nachrichten. Er ist ein Multimaster-Bussystem mit hoher Übertragungssicherheit. Dazu sind Fehlersignalisierung und Lokalisieren ausgefallener Stationen enthalten.

Das Grundprinzip der Kommunikation im CAN-Bus ist in Abbildung 31 abgebildet.

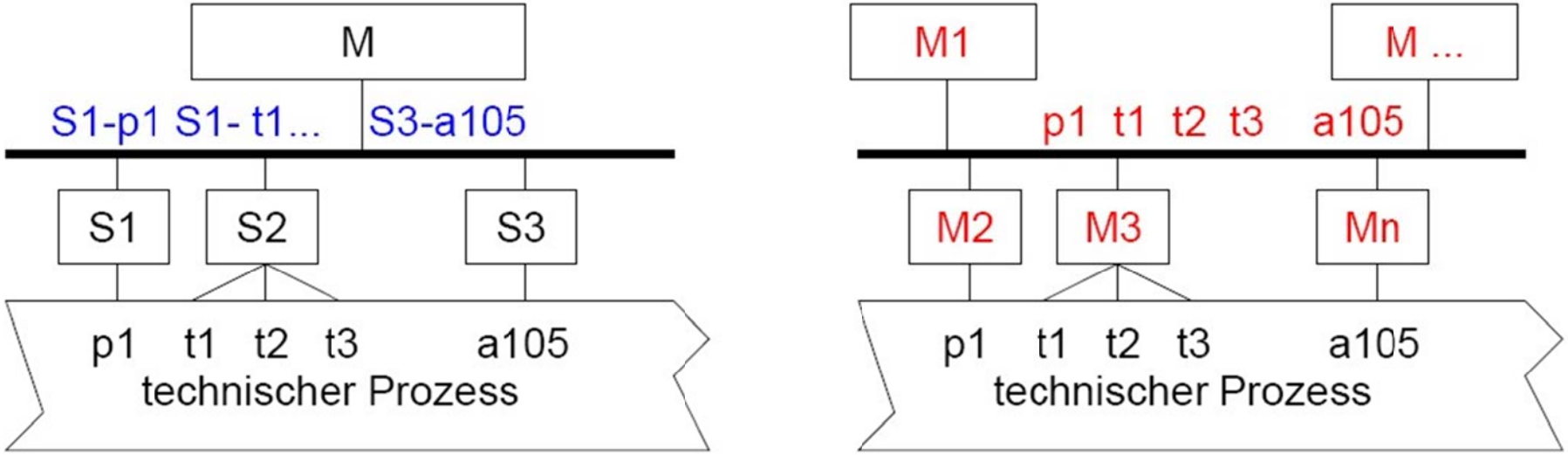


Abbildung 31: Kommunikationsidee im CAN-Bus

2.4. Standards für Echtzeitsystemen

POSIX (IEEE)	= Portable Operating System Interface based on uniX (POSIX 1003.x) - Elemente in POSIX 4 sind Optionen!!! (hinterfragen, was konkret vorhanden ist)
POSIX 1	- grundlegender Betriebssystem-Standard
POSIX 1b (alt: POSIX 4)	- prioritätsgesteuertes preemptives Scheduling - Timer mit höherer Auflösung - Signale mit verbessertem Verhalten - Message Queues und Shared Memory - Semaphoren - „memory locking“ (Verhindern des Swapping best. Speicherbereiche) - asynchrone E/A
POSIX 1c	- Thread-Funktionalität

Tabelle 18: Standards für Echtzeitsysteme

2.5. Der Betriebssystemkern – Ergänzungen für Echtzeitbetriebssysteme

2.5.1. Das Prozesssystem

Die grundlegenden Aussagen und Begriffe zum Prozesssystem sind bereits in der Vorlesung „Betriebssysteme“ angesprochen worden.

2.5.1.1. Prozessumschalter – Scheduler

Der Teil des Betriebssystemkerns, der die verschiedenen Prozesse steuert, nennt man Scheduler. Dieser Scheduler arbeitet mit einem so genannten Scheduler Algorithmus. Folgende gängige Scheduler Verfahren sind möglich:

1. Prioritätsbasiertes Scheduling
   Diese Strategie basiert auf den benutzerdefinierten Prozessprioritäten. Die Priorität repräsentiert dabei die Wichtigkeit der jeweiligen Aufgabe, die mit dem Prozess gelöst wird. Grundsätzlich kommt bei prioritätsbasierten Systemen immer der Prozess zur Ausführung, der zum Zeitpunkt der Einplanung (Scheduling) die aktuell höchste Priorität besitzt. Der Aufruf des Schedulers kommt entweder per Systemcall oder nach Interrupts vor.

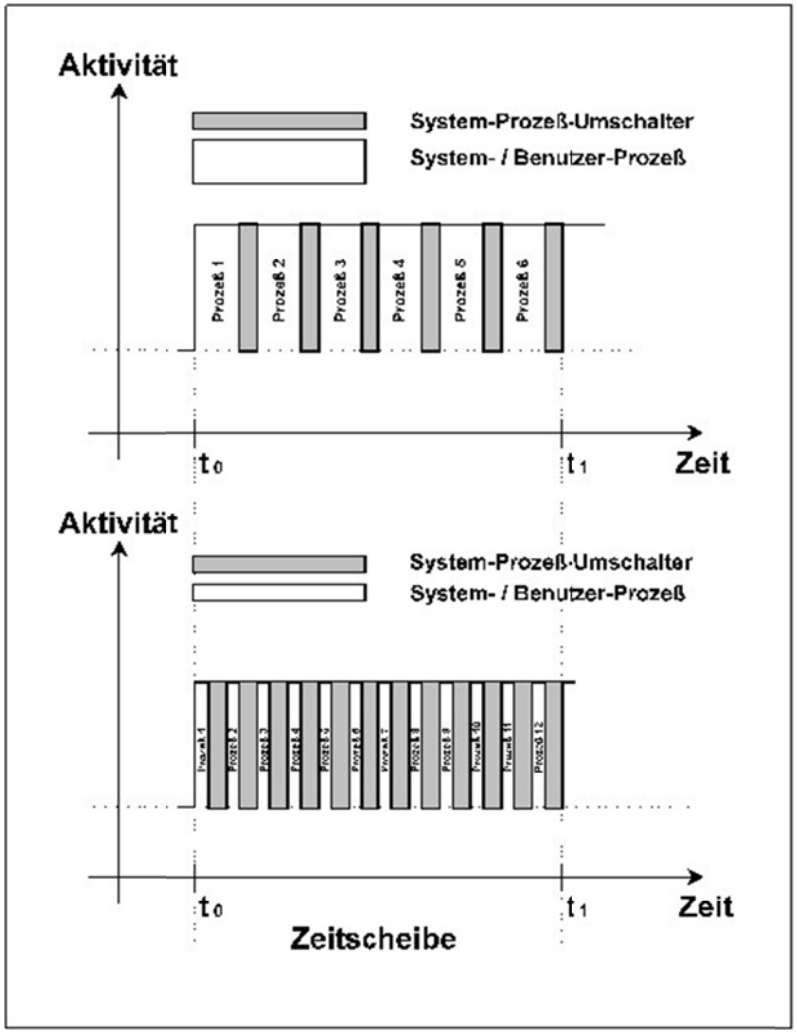


Abbildung 32: gerechtes Scheduling-Verfahren

2. Round-Robin Scheduling
   Bei dieser Strategie ist das Ziel, die Leistung des Prozessors möglichst gleichmäßig auf alle Prozesse zu verteilen. Man spricht auch von so genannten **gerechten Verfahren**.
   Alle Prozesse besitzen die gleiche Priorität. Grundlage der Round Robin Technologie ist ein Zeitscheibensystem. Dabei wird jedem Prozess ein gleicher Teil der Zeitscheibe zugeteilt. Erst wenn alle Prozesse ihren Anteil an der Zeitscheibe an Bearbeitungszeit erhalten haben, wird der erste Prozess wieder gestartet.

Echtzeitbetriebssysteme realisieren alle Tasks in demselben Speicherbereich. Damit wird die Prozesskommunikation sehr stark vereinfacht und der Kontextwechsel verläuft im Prinzip ohne besonderen Adressierungsoverhead. Der Nachteil liegt im fehlenden Schutz der Prozesse untereinander.

Die gemeinsame Nutzung von dem Programmcode durch mehrere Prozesse setzt eine wichtige Eigenschaft voraus. Der gemeinsame Code muss reentrant (wiedereintrittsfähig) sein. Das bedeutet, der gemeinsame Code muss über Mechanismen verfügen, die erlauben, zwar den Programmcode gemeinsam zu benutzen, den Zugriff auf Variablen und Daten jedoch für jeden Prozess getrennt zu verwalten. Folgende Vorkehrungen machen den Programmcode reentrant:

1. Die ausschließliche Verwendung von dynamischen Stackvariablen. Das bedeutet. Die aufrufende Taskfunktion übergibt die Variable als Zeiger auf den taskeigenen Speicher. Damit ist sichergestellt, dass im gemeinsam benutzten Programmcode keine Variablenkonflikte auftreten.

2. Schutz des gemeinsamen Programmcodes durch Semaphore.

2.5.1.2. Echtzeitscheduling

Die Hauptaufgabe der Taskverwaltung besteht in der Zuteilung des bzw. der Prozessor(s)(en) an die ablaufwilligen Tasks. Hierzu gibt es die verschiedensten Strategien. Diese Zuteilungsstrategien werden auch Schedulingverfahren genannt, die Zuteilung selbst erfolgt innerhalb der Task-(Prozess-)verwaltung durch den Scheduler.

Ein Echtzeitscheduler hat die Aufgabe, den Prozessor zwischen allen ablaufwilligen Tasks derart aufzuteilen, dass – sofern überhaupt möglich – alle Zeitbedingungen eingehalten werden. Diesen Vorgang nennt man Echtzeitscheduling. Die Menge der durch den Echtzeitscheduler verwalteten Tasks heißt auch das Taskset.

Zur Bewertung verschiedenen Schedulingverfahren müssen folgende grundlegenden Fragen beantwortet werden:

a) Ist es für ein Taskset überhaupt möglich, alle Zeitbedingungen einzuhalten? Wenn ja, existiert zumindestest ein sogenannter **Schedule**, d.h. eine zeitliche Aufteilung des Prozessors an die Task, der die Aufgabe löst.
b) Wenn dieser Schedule existiert, kann er in endlicher Zeit berechnet werden?
c) Findet das verwendete Schedulingverfahren diesen Schedule, wenn er existiert und in endlicher Zeit berechnet werden kann?

Allgemein kann davon ausgegangen werden, dass es durchaus möglich ist, dass ein solcher Schedule existiert und auch in endlicher Zeit berechnet werden kann, das Schedulingverfahren ihn aber nicht finden kann. Ein solches Schedulingverfahren ist nicht optimal.

Man spricht von **optimalen Schedulingverfahren**, wenn es immer dann, wenn ein Schedule existiert, diesen auch in endlicher Zeit finden kann.

Um im Fall einer konkreten Anwendung im Voraus sagen zu können, ob sie unter allen Umständen ihre Zeitbedingungen einhalten wird, muss das zugehörige Taskset unter Berücksichtigung des verwendeten Schedulingverfahrens analysiert werden. Diese Analyse, die meist mit mathematischen Methoden und Modellen durchgeführt wird, nennt man **Scheduling Analyse**. Aus dieser Analyse heraus kann auch eine Bewertung des Schedulingverfahrens erfolgen, da diese Analyse zeigt, ob ein Schedule existiert und ob das Schedulingverfahren in findet.

Eine zentrale Größe für solche Analysen ist die sogenannte **Prozessorauslastung**. Allgemein kann die Prozessorauslastungwie folgt definiert werden:

Definition 12: Prozessorauslastung
Die Prozessorauslastung H ergibt sich aus

$$H = \frac{\textit{benötigte Prozessorzeit}}{\textit{verfügbare Prozessorzeit}}$$

An einem einfachen Beispiel soll dies verdeutlicht werden. Besitzt eine periodische Task eine Ausführungszeit von 100 ms und eine Periodendauer von 200 ms, so verursacht diese Task eine Prozessorauslastung von 50%. Kommt zu dieser Tast eine zweite periodische Task mit einer Ausführungszeit von 50 ms und einer Periodendauer von 100 ms hinzu, so steigt die Prozessauslastung auf 100%.

Es ist leichtn einzusehen, dass bei einer Prozessorauslastung von mehr als 100% und nur einem verfügbaren Prozessor das Taskset nicht mehr ausführbar ist, d.h. kein Schedule existiert, der die einhaltung aller Zeitbedingungen erfüllt. Bei einer Prozessorauslastung von weniger als 100% sollte hingegen ein solcher Schedule existieren. Es ist jedoch nicht vorhersehbar, ob das verwendete Schedulingverfahren diesen auch findet.

Bei diesem Beispiel wurden nur periodische Task betrachtet, was für viele Anwendungen eine vernünftige Einschränkung darstellt. Die Untersuchung der Prozessorauslastung kann aber auch auf komplexers Taskmodelle angewandt werden, man spricht dann von **Processor Demand Analysis**.

Echtzeitschedulingverfahren lassen sich in verschiedene Klassen aufteilen. Abbildung 33 zeigt die Klassifizierungsmerkmale

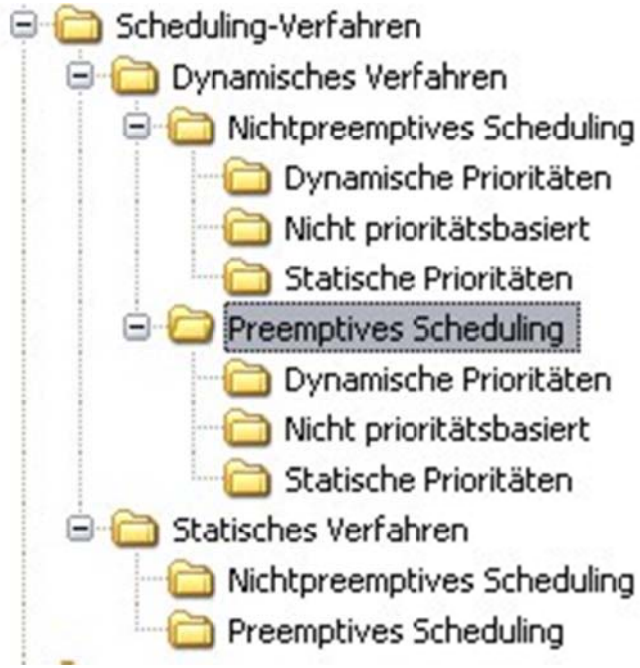


Abbildung 33: Klassifizierungsmerkmale von Schedulingverfahren

Bei **statischem Scheduling** berechnet das Schedulingverfahren vor Ausführung der Task eine Zuordnungstabelle. In sdieser Tabelle sind die Startzeiten der einzelnen Tasks vermerkt. Es entsteht ein

minimaler Overhead durch den Prozessumschalter, da zur Laufzeit keine Entscheidungen mehr getroffen werden müssen. Es sind damit aber auch alle Nachteile verbunden wie Starrheit und Beschränkung auf periodische Ereignisse.

Dynamisches Scheduling berechnet die Zuordnung der Tasks an den Prozessor zur Laufzeit. Auch hier werden je nach Verfahren unterschiedlich komplexe Analysen über Zeitschranken, Ausführzeiten oder Spielräume durchgeführt. Dies führt jedoch zu erhöhtem Laufzeitoverhead. Dadurch wird ein flexibler Ablauf und die Möglichkeit der Reaktion auf aperiodische Ereignisse erreicht.

Nicht zu verwechseln ist **statisches** und **dynamisches Scheduling** mit statischen und dynamischen Prioritäten. Beide Techniken gehören zum dynamischen Scheduling und benutzen Prioritäten der Tasks, um die Zuordnung von Tasks zum Prozessor zu bestimmen.

Bei **statischen Prioritäten** werden die Prioritäten der einzelnen Tasks einmal zu Beginn der Ausführung festgelegt und während der Laufzeit nie verändert. Bei **dynamischen Prioritäten** können die Prioritäten zur Laufzeit an die Gegebenheiten angepasst werden.

Daneben existieren Schedulingverfahren, die völlig auf Prioritäten verzichten.

Ein weiteres Unterscheidungsmerkmal ist die Fähigkeit zur **Preemption.** Dies heißt übersetzt „Vorkaufsrecht“**.** **Preemptives Scheduling** bedeutet, dass eine unwichtige Task zur Laufzeit von einer wichtigen Task verdrängt werden kann. Dadurch kommt die wichtigste bereite Task sofort zur Ausführung. Eine unwichtige Task wird erst dann fortgesetzt, wenn alle wichtigen Tasks abgearbeitet oder blockiert sind.

2.5.1.3. Taskmodell

Ein Rechenprozess, auch Task genannt, ist ein auf einem Rechner ablaufendes Programm zusammen mit allen zugehörigen Variablen und Betriebsmitteln. Eine Task ist somit ein vom Betriebssystem gesteuerter Vorgang der Abarbeitung eines sequentiellen Programms. Hierbei können mehrere Tasks quasi-parallel vom Betriebssystem bearbeitet werden, der tatsächliche Wechsel zwischen den einzelnen Tasks wird vom Betriebssystem mittels Programmumschalter gesteuert. Da die von einer Task zu bewältigenden Aufgaben oft sehr komplex sind, führen viele Betriebssysteme hier eine weitere Hierarchiestufe von parallel ausführbaren Objekten ein, die so genannten Threads.

Definition 13: Task

Eine Task ist ein sogenannter schwergewichtiger Prozess, der eigene Variablen und Betriebsmittel enthält und von den anderen Tasks durch das Betriebssystem abgeschirmt wird. Sie besitzen einen eigenen Adressraum und kann nur über Kanäle der Prozesskommunikation mit anderen Tasks kommunizieren.

Definition 14: Thread

Ein Thread ist ein so genannter leichtgewichtiger Prozess, der innerhalb einer Task existiert. Er benutzt die Variablen und Betriebsmittel der Task. Alle Threads innerhalb einer Task teilen sich denselben Adressraum. Die Kommunikation kann über beliebige globale Variablen innerhalb der Task erfolgen.

Hieraus ergeben sich eine Reihe unterschiedlicher Eigenschaften von Tasks und Threads. Eine Task bietet größtmöglichen Schutz, die Beeinflussung durch andere Tasks ist auf vordefinierte Kanäle beschränkt. Threads hingegen können sich innerhalb einer Task beliebig gegenseitig stören, da keine „schützenden Mauern“ zwischen ihnen errichtet sind. Dies ermöglicht aber auf der anderen Seite hohere Effizienz.

Die Kommunikation zwischen Threads ist direkter und schneller. Ein weiterer Vorteil von Threads ist der schnelle Kontexwechsel. Das bedeutet, der Wechsel zwischen zwei Tasks ist in der Regel bedeutend langsam als der Wechsel zwischen zwei Threads. Dies lässt sich sogar noch weiter beschleunigen, wenn ein Prozessor verwendet wird, der die Umschaltung zwischen Threads per Hardware unterstützt.

Die Scheduling Algorithmen setzen voraus, dass sich die Prozesse bzw. Task in verschiedenen Zuständen befinden. Das Schema, das die Beziehungen zwischen den einzelnen Zuständen und die Überführung zwischen den Zuständen beschreibt, nennt man das Taskmodell.

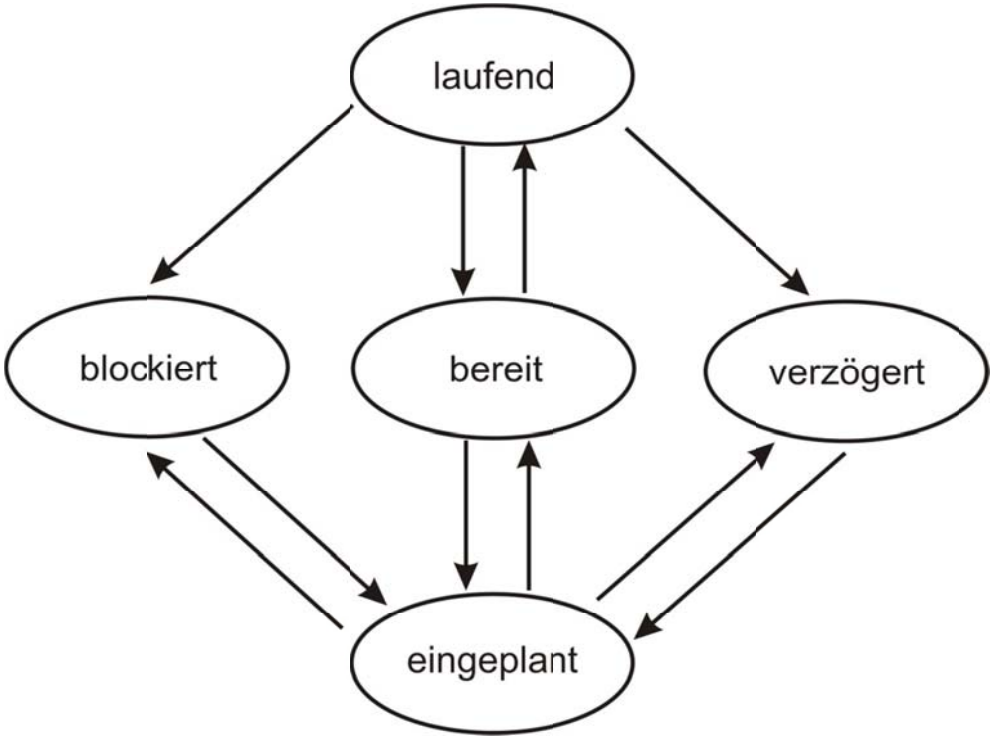


Abbildung 34: Taskzustände im Taskmodell

2.5.1.4. Taskzustände

Die einzelnen Zustände haben dabei folgende Bedeutung:

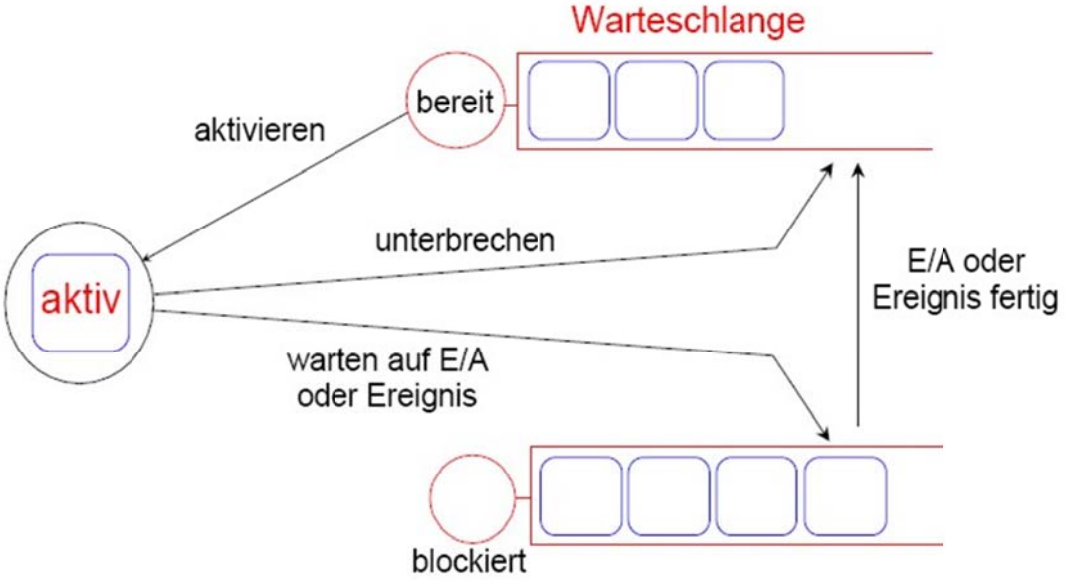
laufend Der Prozess ist dem Prozessor zugeordnet und wird aktuell bearbeitet.

bereit Der Prozess ist ausführbereit, alle Bedingungen sind erfüllt. Derjenige Prozess, der beim nächsten Scheduleraufruf die höchste Priorität besitzt und gleichzeitig im Zustand bereit ist, bekommt den Prozessor zugeteilt.

blockiert Der Prozess ist blockiert, er wartet auf ein Ereignis. Der blockierte Zustand wird dann angenommen, wenn zwei Prozesse synchronisiert werden sollen oder mehrere Prozesse auf ein Betriebsmittel zugreifen wollen. Man bezeichnet dies als Warte- oder Schlafzustand.

verzögert Der Prozess ist verzögert, er wartet eine bestimmte Zeit. Hierbei handelt es sich ebenfalls um einen Warte- oder Schlafzustand, der allerdings nur zeitliche Randbedingungen beinhaltet.

eingeplant Der Prozess ist eingeplant oder wurde neu erzeugt. Damit ist der Prozess dem System bekannt. Meistens kann der Prozess nicht unmittelbar in den laufenden Zustand überführt werden.



Die Übergänge zwischen den einzelnen Taskzuständen unbewertet. In einem bewerteten Taskmodell lassen sich als Kantenbewertungen entsprechende Systemdienste des Echtzeitkerns angeben. Somit sind die Kantenbewertungen dann systemabhängig, während die Taskzustände in der Regel allgemeiner und damit meistens nicht systemabhängig sind.

Abbildung 35: Verwaltung von Prozessen

2.5.1.5. Statisches und dynamisches Taskmodell

In der Vorlesung Betriebssysteme wurde auf die Prozesserzeugung bzw. –beendigung eingegangen. Nach der Art der Erzeugung der Prozesse wird zwischen einem statischen und einem dynamischen Taskmodell unterschieden.

Werden sofort nach dem Start des Echtzeitsystems alle notwendigen Prozesse erzeugt und in einen Wartezustand versetzt, spricht man von einem statischen Taskmodell. Die Prozesse werden auch mit dem Herunterfahren des Echtzeitsystems mitbeendet. Während des Betriebs des Echtzeitsystems werden keine Prozesse erzeugt bzw. beendet.

Definition 15: Statische Taskverwaltung

Von statischer Taskverwaltung spricht man, wenn nach dem Start des Echtzeitsystems alle Tasks bekannt sind und durch den Echtzeitkern verwaltet werden. In der gesamten Laufzeit ist keine Manipulation der Taskanzahl möglich.

Bei Echtzeitsystemen mit dynamischem Taskmodell laufen zu jedem beliebigen Zeitpunkt nur die unmittelbar benötigten Prozesse, plus eine Reserve. Werden zusätzliche Prozesse benötigt, müssen diese erst erzeugt werden, wenn die Reserve von Prozessen verbraucht ist. Im Gegensatz dazu werden nicht benötigte Prozesse nach einer gewissen Zeit (idle time) wieder beendet, bzw. werden als Reserve verwendet.

Definition 16: Dynamische Taskverwaltung

Dynamische Taskverwaltung liegt vor, wenn der Echtzeitkern nach dem Start des Echtzeitsystems nur die Informationen über eine Task vorliegen. Alle anderen Tasks befinden sich im Zustand nicht existent und können durch aktive Tasks generiert (d.h. gestartet) werden. Die Anzahl der verwalteten Tasks ist in der gesamten Laufzeit variabel.

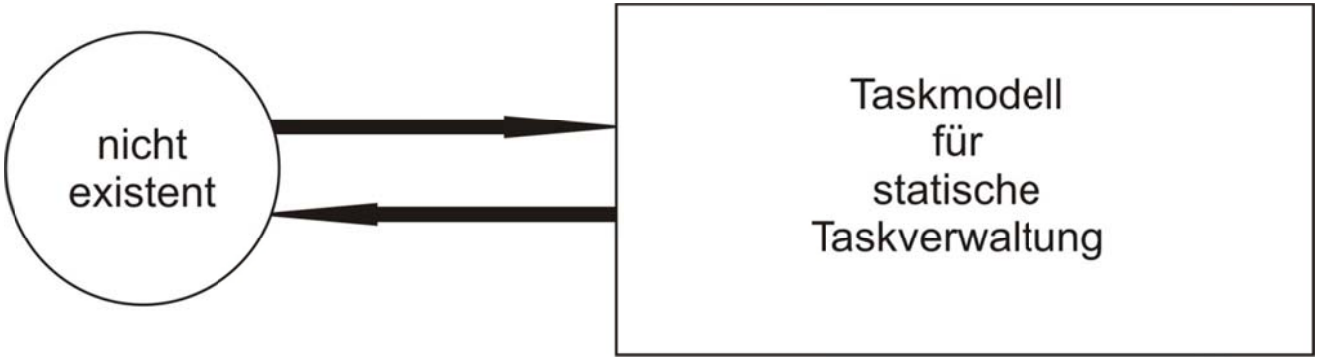


Abbildung 36: Statische und dynamische Taskverwaltung

2.5.1.6. Synchronisation und Verklemmung

Das Problem der Synchronisation von Tasks entsteht, wenn Tasks nicht unabhängig voneinander sind. Abhängigkeiten ergeben sich, wenn gemeinsame Betriebsmittel genutzt werden, auf die der Zugriff koordiniert werden muss. Gemeinsame Betriebsmitten können sein:

a) **Daten**. Mehrere Tasks greifen lesend und schreibend auf gemeinsame Variablen oder Tabellen zu. Bei unkoordiniertem Zugriff könnte eine Task inkonsistente Werte lesen, z.B. wenn eine andere Task erst einen Teil einer Tabelle aktualisiert hat.

b) **Geräte**. Mehrere Tasks benutzen gemeinsame Geräte wie etwa Sensoren oder Aktoren. Auch hier ist eine Koordinierung erforderlich, um z.B. keine widersprüchlichen Kommandos von zwei Tasks an einen Schrittmotor zu senden.

c) **Programme**. Mehrere Tasks teilen sich gemeinsame Programme, z.B. Gerätetreiber. Hier ist dafür Sorge zu tragen, dass konkurierende Aufrufe dieses Programms so abgewickelt werden, dass keine inkonsistenten Programmzustände entstehen.

Es gibt zwei grundlegende Varianten der Synchronisation:

a) Die **Sperrsynchronisation**, auch wechselseitiger Ausschluss, **Mutual Exclude** – **Mutex** genannt stellt sicher, dass zu einem Zeitpunkt immer nur eine Task auf ein gemeinsames Betriebsmittel zugreift.

b) Die **Reihenfolgesynchronisation**, auch **Kooperation** genannt, regelt die Reihenfolge der Taskzugriffe auf gemeinsame Betriebsmittel. Anders als bei der Sperrsynchronisation, die nur ausschließt, dass ein gemeinsamer Zugriff stattfindet, aber nichts über die Reihenfolge der Zugriffe aussagt, wird bei der Reihenfolgesynchronisation genau diese Reihenfolge der Zugriffe exakt definiert.

Bereiche, in denen Tasks synchronisiert werden müssen heißen **kritische Bereiche**.

2.5.2. Interprozesskommunikation

Prozesse müssen ständig mit anderen Prozessen kommunizieren. In einer Shell-Pipe, unter Unix zum Beispiel muss die Ausgabe des ersten Prozesses an den zweiten Prozess weitergereicht werden. Damit gibt es die Notwendigkeit für die Kommunikation zwischen Prozessen, vorzugsweise in einer gut strukturierten Weise, die keine Unterbrechungen verwendet.

Bei der Interprozesskommunikation (IPC) geht es im Wesentlichen um drei Aspekte:

1. Wie ein Prozess Informationen an einen anderen Prozess weiterreichen kann,
2. Sicherzustellen, dass zwei oder mehr Prozesse sich nicht gegenseitig in die Quere kommen, wenn kritische Aktivitäten anliegen (konkurierender Zugriff auf gleiche Resourcen) und
3. Betrifft den sauberen Ablauf, wenn Abhängigkeiten vorliegen (z.B. die Synchronisation zwischen erzeugendem und verbrauchendem Prozess).

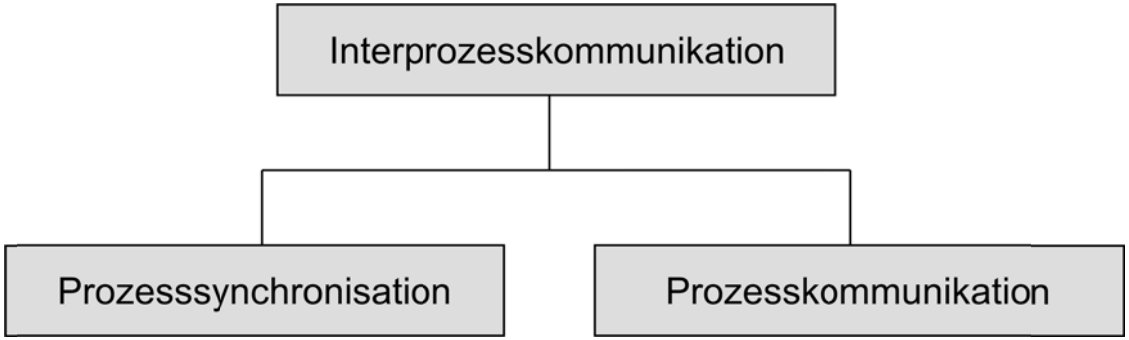


Abbildung 37: Interprozesskommunikation

Synchronisation und Kommunikation von Tasks gehören eng zusammen. Man kann die Synchronisation auch als informationslose Kommunikation betrachten. Auf der anderen Seite kann die Kommunikation zur Synchronisation benutzt werden. Ein Beispiel hierfür wäre etwa das Warten auf einen Auftrag. Mit dem Auftrag wird Information übermittelt, durch den Zeitpunkt der Auftragsvergabe ist eine Synchronisation möglich.

Es gibt zwei grundlegende Varianten der Taskkommunikation:

a) **Gemeinsamer Speicher**. Der Datenaustausch erfolgt über einen gemeinsamen Speicher. Die Synchronisation, d.h. wann darf wer lesend oder schreibend auf die Daten zugreifen, geschieht über Semaphore.

b) **Nachrichten (Messages)**. Der Datenaustausch und die Synchronisation erfolgt über das Verschicken von Nachrichten.

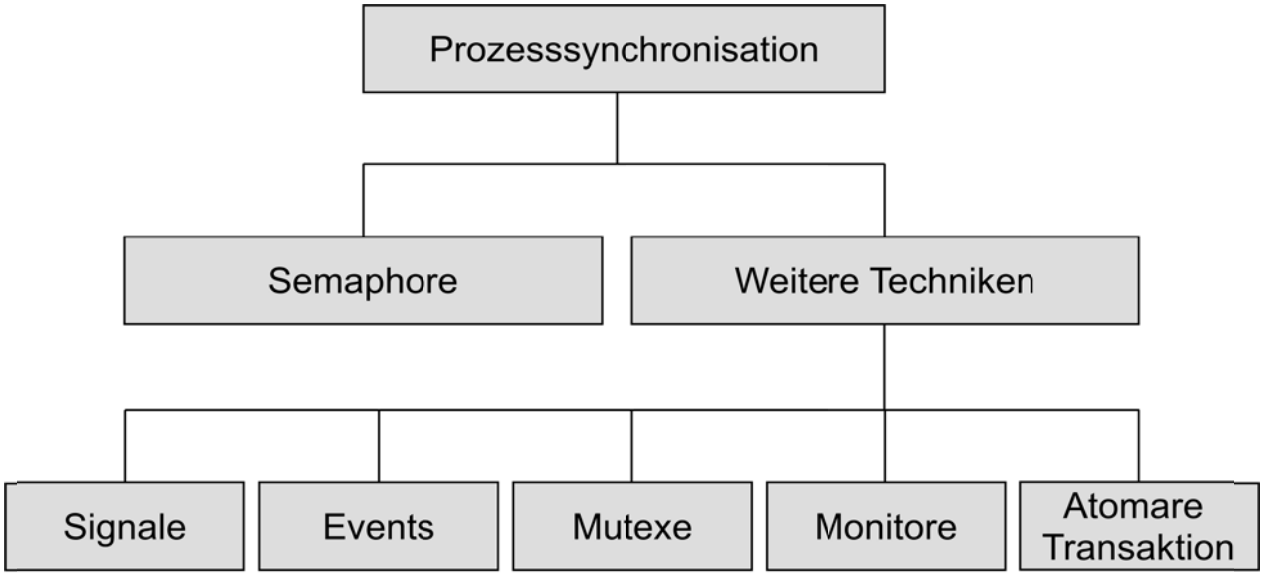


Abbildung 38: Möglichkeiten der Prozesssynchronisation

Ein besonderes Merkmal der Kommunikation in Echtzeitsystemen ist **prioritätsbasierte Kommunikation**. Hierbei besteht die Möglichkeit, Nachrichten oder über gemeinsamen Speicher übertragene Informationen mit Prioritäten zu versehen. So können hochpriorisierte Informationen solche mit niedriger Priorität überholen. Prioritätsbasierte Kommunikation ist wichtig, um bei der Kooperation von Tasks die Prioritätskette lückenlos aufrecht zu erhalten. Man spricht hier von der Wahrung der **End-zu-End-Priorität**.

Ein weiteres Merkmal bei der Kommunikation ist die zeitliche Koordinierung. Hier kann man unterscheiden zwischen:

a) **Synchroner Kommunikation**. Es existiert mindestens ein Zeitpunkt, an dem Sender und Empfänger gleichzeitig an einer definierten Stelle stehen, d.h. ein Teilnehmer wird in der Regel blockiert (Aufruf einer Funktion ***Warten_auf_Nachricht***).

b) **Asynchroner Kommunikation**. Hier müssen die Tasks nicht warten, der Datenaustausch wird vom Kommunikationssystem gepuffert. Die Tasks können jederzeit nachsehen, ob neue Daten für sie vorhanden sind (Aufruf einer nichtblockierenden Funktion ***Prüfe_ob_Nachricht_vorhanden***).

2.5.2.1. Semaphore

Semaphore sind typische Elemente von Echtzeitsystemen zur Steuerung des Systemverhaltens. Sie sind systemweit bekannte Objekte, die zur Synchronisation der im System vorhandenen Prozesse verwendet werden. Synchronisation bedeutet in diesem Zusammenhang:

1. dass ein Prozess auf Daten eines anderen Prozesses wartet,
2. dass sich zwei Prozesse ein Betriebsmittel teilen müssen oder
3. dass eine Ereignissteuerung im Gegensatz zum Polling realisiert werden soll.

Der Niederländer **E.W.Dijkstra** schlug 1965 vor, eine ganzzahlige Variable einzuführen, die er **Semaphore** nannte. Ein Semaphor könnte den Wert 0 besitzen, wenn kein Event vorliegt. Mit irgendeinem positiven Wert wird die Anzahl der Events, von verschiedenen Prozessen kommend angezeigt. Zur Bearbeitung sollten zwei Operationen dienen, **down** und **up**. Die **down-Operation** eines Semaphors prüft, ob der Wert größer 0 ist. Falls dem so ist, erniedrigt sie den Wert um eins (z.B. um ein Event zu bearbeiten) und macht einfach weiter. Falls der Wert 0 ist, wird der Prozess sofort schlafen gelegt, ohne **down** vollständig auszuführen.

Die **up-Operation** erhöht den Wert, der von dem Semaphor adressiert wird, um eins. Falls ein oder mehrere Prozesse wegen eines Semaphors schlafen sollten, unfähig eine frühere **down-Operation** abzuschließen, wird vom System per Zufall ein Prozess gewählt, der seine **down-Operation** vervollständigen kann. Somit bleibt zwar nach der **up-Operation** an einem Semaphore, auf den schlafende Prozesse warten, der Wert des Semaphors immer noch auf dem Wert 0, aber es gibt einen Prozess weniger, der wegen des Semaphors schläft.

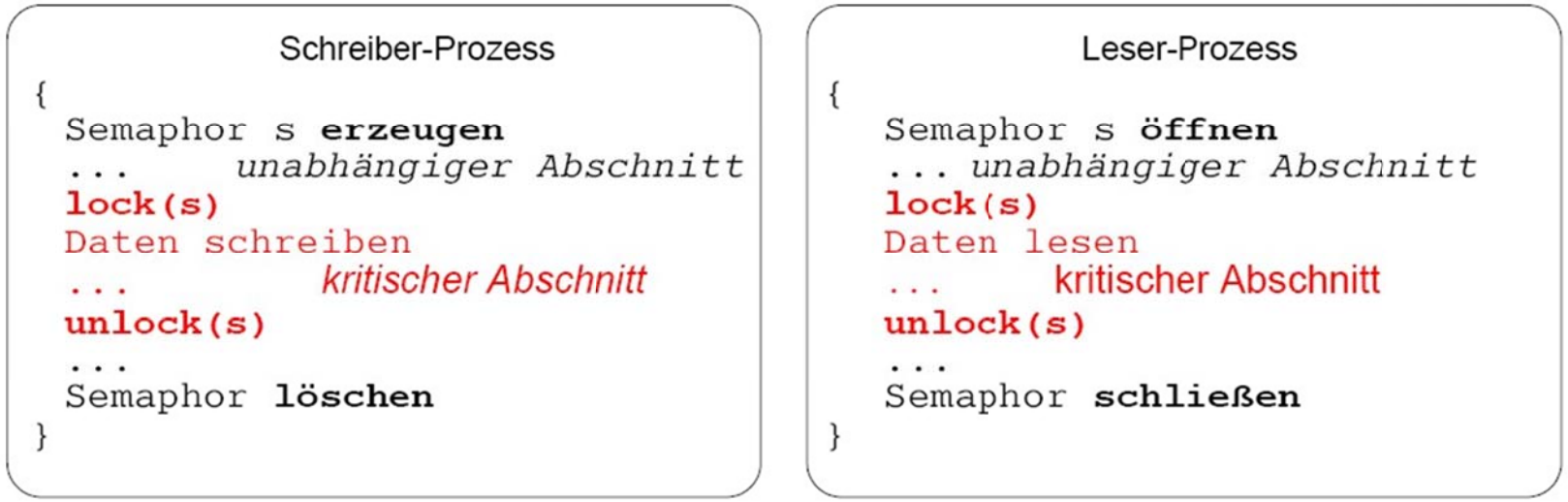


Abbildung 39: Semaphore

Typischerweise werden sie aufgrund der Besonderheit von Echtzeitbetriebssystemen, dass Prozesse im gleichen Adressraum ausgeführt werden, zu deren Schutz verwendet.

Der Schlüssel zur Vermeidung von Problemen bei der gemeinsamen Nutzung von Speicherbereichen, Dateien oder anderen gemeinsam genutzen Ressourcen ist es, einen Weg zu finden, der es einem Prozess verbietet, gemeinsam mit einem anderen Prozess auf die gemeinsam genutzen Ressourcen zuzugreifen.

Es wird ein **wechselseitiger Ausschluss** gebraucht, der aber nur für eine gewisse Zeit, den **kritischen Abschnitt** benötigt wird. Dazu müssen folgende Bedingungen erfüllt sein, damit parallele Prozesse richtig und effizient zusammen arbeiten können:

1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Abschnitten sein.
2. Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPU's gemacht werden.
3. Kein Prozess, der außerhalb seiner kritischen Abschnitte läuft, darf andere Prozesse blockieren.
4. Kein Prozess sollte ewig darauf warten müssen, in seinen kritischen Abschnitt einzutreten.

Auf eine abstrakte Weise wird das gewünschte Verhalten in dargestellt.

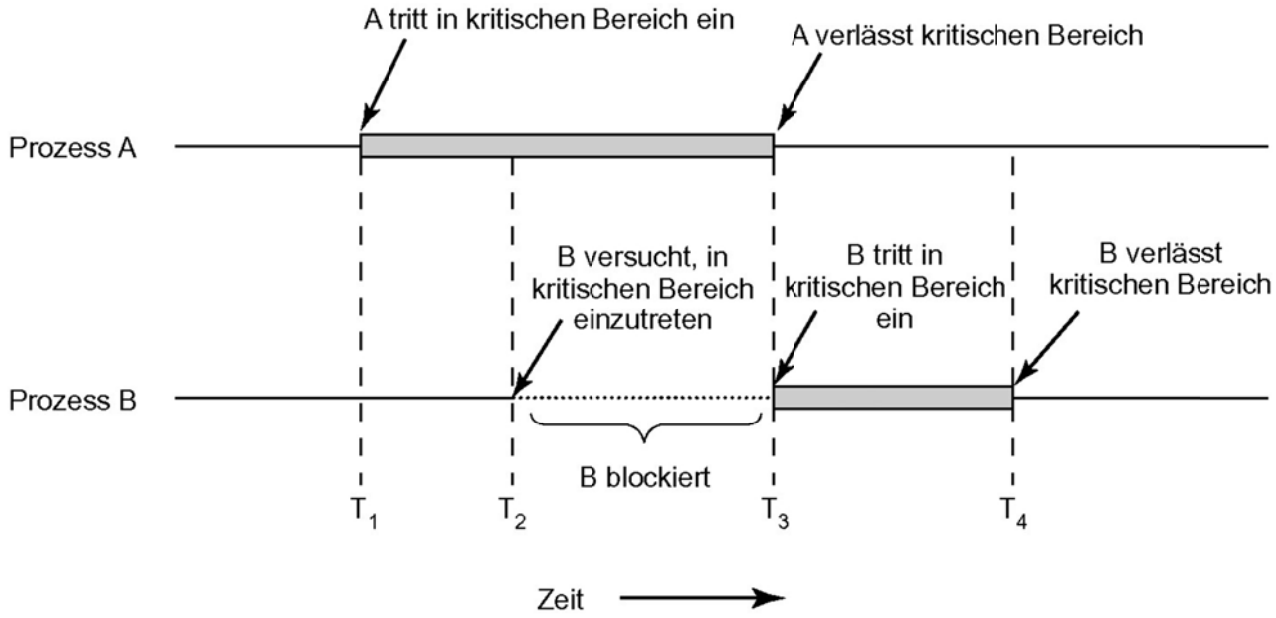


Abbildung 40: Wechselseitiger Ausschluss unter Verwending kritischer Bereiche

Im Allgemeinen unterscheidet man drei verschiedene Typen von Semaphoren:

1. Binäre Semaphore,
2. Ausschlusssemaphore (Mutexe) und
3. Zählsemaphore.

Wie in Abbildung 41 dargestellt, besteht ein Semaphor aus einer Zählvariablen sowie zwei nicht unterbrechbaren Operationen **„Passieren“** – P (**up-Operation**) und **„Verlassen“** – V (**down-Operation**).

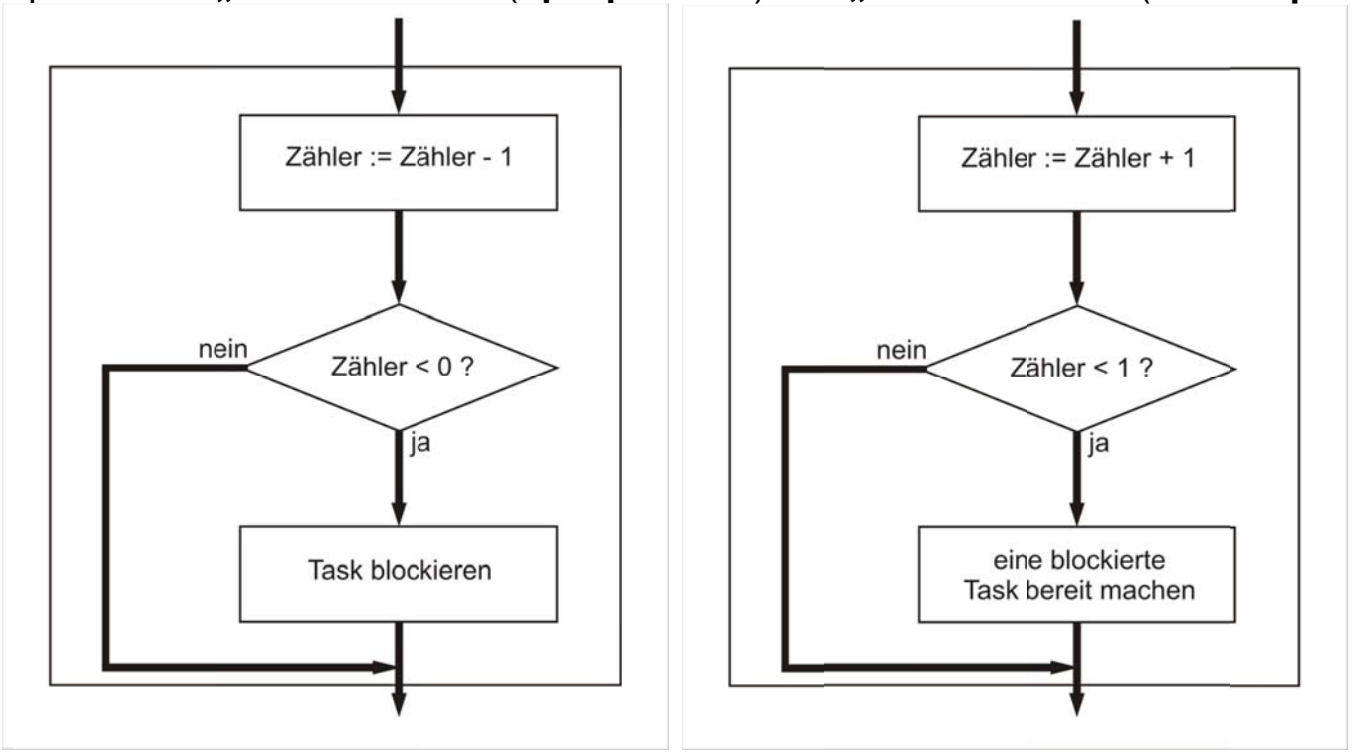


Abbildung 41: Die Operationen „Passieren“ und „Verlassen“ eines Semaphors

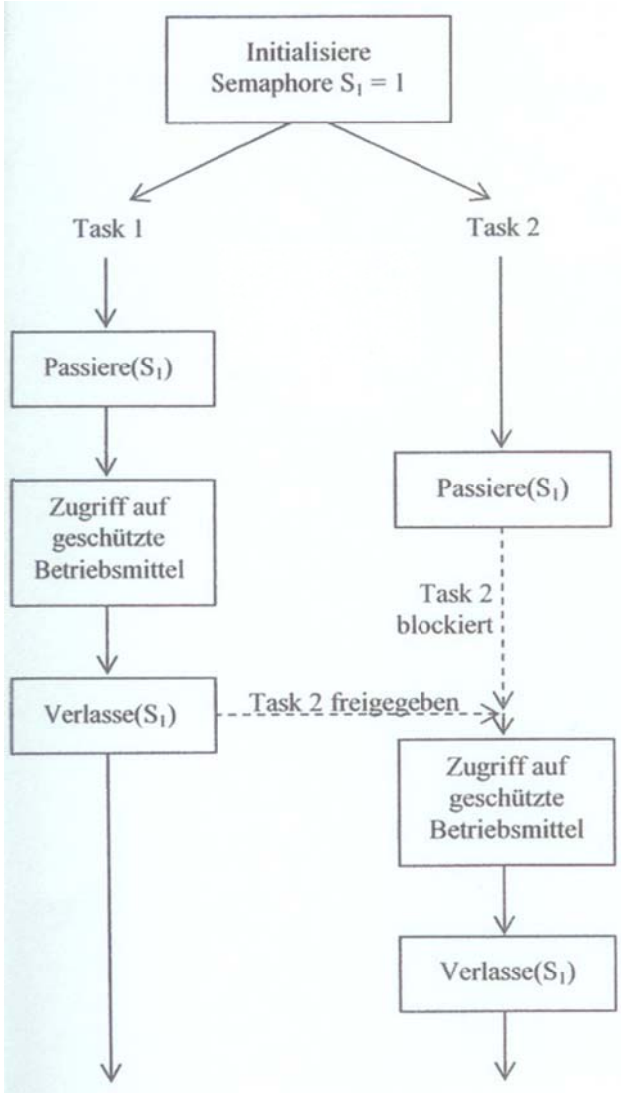
Als Demonstrationen der Operationen sind PL/SQL-Prozeduren in Abbildung 42 und Abbildung 43 enthalten.

```
create or replace procedure passieren (p_semaphore in number)
is
  v_zaehler  semaphore.zaehler%TYPE;
begin
  select zaehler
  into    v_zaehler
  from   semaphore
  where semaphore = p_semaphore;
  v_zaehler := v_zaehler - 1;
  update semaphore
  set zaehler = v_zaehler
  where semaphore = p_semaphore;
  commit;
  loop
    select zaehler
    into    v_zaehler
    from   semaphore
    where semaphore = p_semaphore;
    exit when v_zaehler >= 0;
  end loop;
exception
 when others then null;
end;
```

Abbildung 42: Die Operationen „Passieren“ als PL/SQL-Prozedur

```
create or replace procedure verlassen (p_semaphore in number)
is
  v_zaehler  semaphore.zaehler%TYPE;
begin
  select zaehler
  into    v_zaehler
  from   semaphore
  where semaphore = p_semaphore;
  v_zaehler := v_zaehler + 1;
  update semaphore
  set zaehler = v_zaehler
  where semaphore = p_semaphore;
  commit;
exception
  when others then null;
end;
```

Abbildung 43: Die Operationen „Verlassen“ als PL/SQL-Prozedur



Ruft eine Task die Operation **„Passieren“** eines Semaphors auf, so wird die zugehörige Zählvariable erniedrigt. Erreicht die Zählvariable einen Wert < 0, so wird die aufrufende Task blockiert. Bei der Initialisierung des Semaphors gibt somit der positive Wert der Zählvariablen die Anzahl der Tasks an, die die Semaphore passieren dürfen und somit in den durch den Semaphore geschützten kritischen Bereich eintreten dürfen.

Bei Aufruf der Operation **„Verlassen“** wird die Zählvariable wieder erhöht. Ist der Wert der Zählvariablen nach dem Erhöhen noch kleiner als 1, so wartet mindestens eine Task auf das Passieren. Aus der Liste dieser blockierten Tasks wird eine Task freigegeben und in den Zustand bereit gebracht. Sie kann nun passieren. Ein negativer Zählerwert gibt somit die Anzahl der Tasks an, denen der Eintritt bisher verwehrt wurde.

Es ist von entscheidender Bedeutung, dass die Operationen **„Passieren“** und **„Verlassen“** atomar sind, d.h. nicht von der jeweils anderen Operation unterbrochen werden können. Nur so ist eine konsistente Handhabung der Zählvariablen sichergestellt.

Ein wesentlicher Unterschied von Semaphoren in Standardbetriebssystemen und Echtzeitbetriebssystemen besteht darin, welche der blockierten Task bei Verlassen eines Semaphors in den Zustand bereit versetzt wird. Bei Standardbetriebssystemen ist dies eine beliebige Task aus der Liste der blockierten Tasks (die Auswahl hängt allerdings von der Implementierung des Betriebssystems ab).

Abbildung 44: Sperrsynchronisation mit einem Semaphor

Bei Echtzeitbetriebssystemen wird die blockierte Task mit der höchsten Priorität (oder der engsten Zeitschranke bzw. dem engsten Spielraum) zur Ausführung gebracht.

Ein Semaphor kann zur Realisierung beider Synchronisationsarten benutzt werden:

a) Zur **Sperrsynchronisation** (Mutex) wird die Zählvariable des Semaphors mit 1 initialisiert. Dies bewirkt, dass immer nur eine Task passieren kann und somit Zugriff auf das geschützte Betriebsmittel erhält. Abbildung 44 zeigt das Prinzip. Task 1 passiert die Semaphore, die Zählvariable wird hierdurch auf 0 reduziert. Beim Versuch ebenfalls zu passieren wird daher die Task 2 blockiert, bis Task 1 den Semaphore wieder verlässt und die Zählvariable zu 1 wird. Die geschützten Betriebsmittel können so immer nur von einer Task belegt werden.

b) Eine **Reihenfolgesynchronisation** kann durch wechselseitige Belegung mehrerer Semaphore erreicht werden. Abbildung 45 zeigt ein Beispiel, in dem die Ablaufsteuerung zweier Tasks durch zwei Semaphore geregelt wird. Zu Beginn einer Aktivität versuchen beide Tasks, ihren Semaphore (Task 1 $\rightarrow S_1$, Task 2 $\rightarrow S_2$) zu passieren. Am Ende der Aktivität verlässt jede Task den Semaphore der jeweils anderen Task (Task 1 $\rightarrow S_2$, Task 2 $\rightarrow S_1$). Semaphore S_1 wird mit 0 und Semaphore S_2 mit 1 initialisiert.

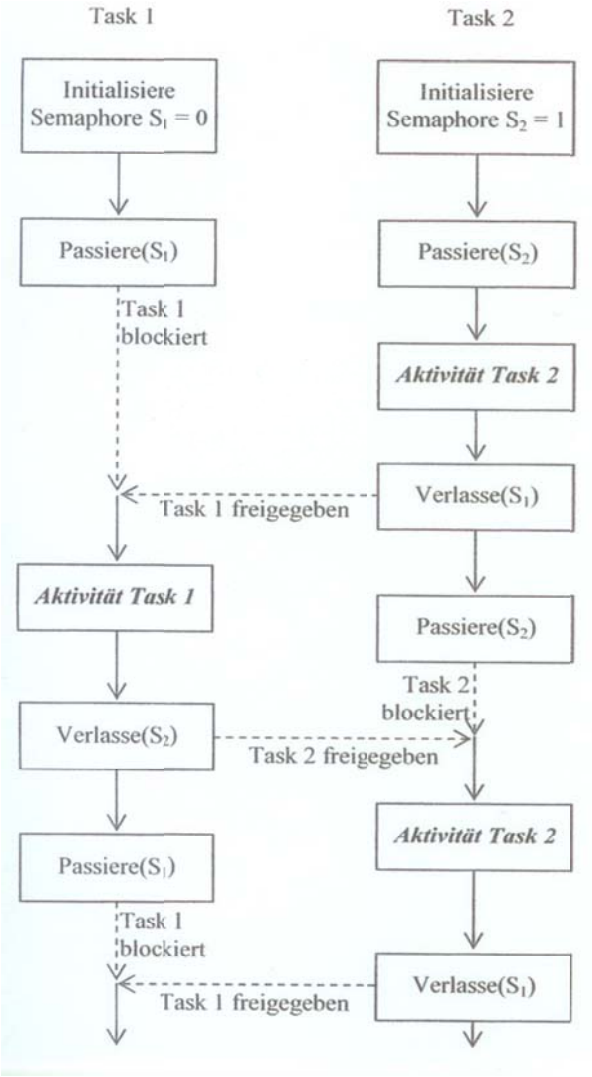


Abbildung 45: Reihenfolgesynchronisation mit zwei Semaphoren

Bei der Synchronisation von Tasks kann es zu **Verklemmungen** kommen, d.h. die Fortführung einer oder mehrerer Tasks wird auf die Dauer blockiert. In der Taskverwaltung führt das zum **Deadlock**.

Bei einem Deadlock warten mehrere Tasks auf die Freigabe von Betriebsmitteln, die sich gegenseitig blockieren. Dies führt zu einem Stillstand der Tasks, da sie auf Ereignisse warten, die nicht mehr eintreten können.

Vermeidung von Deadlocks:
Müssen mehrere Betriebsmittel gleichzeitig geschützt werden, so müssen alle Tasks die zugehörigen Semaphore in der gleichen Reihenfolge passieren.

2.5.2.2. Binäre Semaphoren - Mutex

Benötigt man die Möglichkeit eines Semaphors zu zählen nicht, wird manchmal eine vereinfachte Version eines Semaphors, **Mutex** genannt, verwendet. Mutexe dienen nur der Verwaltung des gegenseitigen Ausschlusses von irgendeiner gemeinsam genutzten Ressource. Sie sind einfach und effizient zu realisieren, was sie besonders in Thread-Paketen nützlich macht, die komplett im Benutzeradressraum realisiert sind.

Ein Mutex ist eine Variable, die zwei Zustände annehmen kann: nicht gesperrt oder gesperrt. Folglich wird nur 1 Bit benötigt. Zwei Prozeduren werden mit Mutexen verwendet. Wenn ein Thread (oder ein Prozess) Zugang zu einem kritischen Abschnitt braucht, ruft er *mutex_lock* auf. Falls der Mutex gerade nicht gesperrt ist (was bedeutet, dass der kritische Abschnitt verfügbar ist), ist der Aufruf erfolgreich und dem aufrufenden Thread steht es frei, in den kritischen Abschnitt einzutreten.

Falls der Mutex jedoch bereits gesperrt ist, wird der aufrufende Thread so lange gesperrt, bis der andere Thread den kritischen Abschnitt durch Aufruf von *mutex_unlock* verlässt. Wenn mehrere Threads wegen des Mutex gesperrt sind, wird einer von ihnen per Zufall (oder Priorität) ausgewählt und ihm erlaubt, die Sperre zu erwerben.

```
mutex_lock:
        TSL REGISTER, MUTEX     | kopiere Mutex in Register, Mutex = 1
        CMP REGISTER, #0        | war Mutex Null?
        JZE ok                  | wenn Null, Mutex war belegt, Rücksprung
        CALL thread_yield       | Mutex belegt; führe anderen Thread aus
        JMP mutex_lock          | versuche es später
ok:     RET                     | Rücksprung, in kritischen Bereich eingetreten

mutex_unlock:
        MOVE MUTEX, #0          | speichere 0 im Mutex
        RET                     | Rücksprung
```

Abbildung 46: Realisierung von mutex_lock und mutex_unlock

2.5.2.3. Interrupts

Der reguläre Betrieb des Echtzeitsystems kann von externen (**Interrupts**) oder internen (**Signalen**) Ereignissen unterbrochen werden. Interrupts sind im Prinzip elektrische Signale, die direkt oder über so genannte Interrupt Controller an die CPU gemeldet werden. Setzt man voraus, dass Echtzeitsysteme für kürzeste Reaktionszeiten konstruiert sind, so muss jeder externe Interrupt in der Lage sein, den aktuell ablaufenden Prozess zu unterbrechen. Die Zeit, die vom Anlegen des Interupts bis zum Start der entsprechenden **Interruptserviceroutine** vergeht, nennt man **Interruptlatenzzeit**. Sie wird oftmals zum Vergleich der verschiedenen Echtzeitsysteme herangezogen. Typische Interruptlatenzzeiten liegen bei modernen Echtzeitbetriebssystemen im Bereich um die 10 Mikrosekunden.

Signale sind interne Programmunterbrechungen, die grundsätzlich ebenfalls Prozesse unterbrechen. Damit Signale Prozesse unterbrechen können, muss innerhalb des Prozesses eine Art **Signalserviceroutine** (**Signal Handler**) installiert sein, die beim Eintreffen des passenden Signals aktiviert wird.

Signale sind im Prinzip **Softwareinterrupts**, die ein asynchrones Ereignis anzeigen. Der Signal Handler wird beim Auftreten des Signals ausgeführt. Nachdem die Signalserviceroutine abgelaufen ist, wird der Prozess an der Stelle der Unterbrechung fortgesetzt. Grundsätzlich stellen Signale die Möglichkeit dar, wichtige Operationen asynchron mit hoher Priorität zur Ausführung zu bringen.

2.5.2.4. Events

Events sind Ereignisse, auf die Prozesse warten und damit aus dem schlafenden Zustand zu erwachen. Prozesse können andererseits Events schicken um genau diesen Effekt bei anderen Prozessen zu erreichen. Z.B. werden im Unix Events durch Signale abgebildet. Das Kommando

```
kill -9 %1
```

sendet das Signal 9 (SIGKILL) an den Prozess %1. Dieser kann das Signal nicht ignorieren und muss sich beenden.

Events und Interrupts haben ein ähnliches Prinzip, aber auch deutliche Unterschiede. Interrupts finden auf reiner Hardwareebene statt, während Events von der Software ausgelöst werden.

Interrupts werden ausgelöst und abgearbeitet (siehe Abbildung 47).

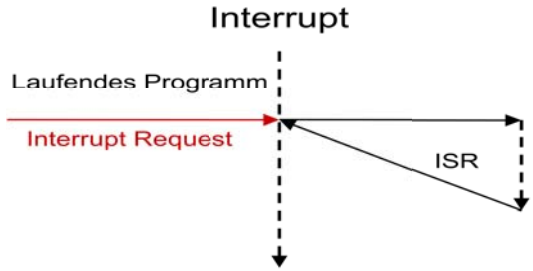


Abbildung 47: Interrupts

Events werden erst dann auf signalisiert gesetzt, wenn eine bestimmte Reaktion der Umwelt (x mal) eingetreten ist (siehe Abbildung 48).

Abbildung 48: Events

Wenn ein blockierender Zustand innerhalb eines Prozesses auftritt wird ein nicht signalisierter Event gesetzt. Der Prozess kann erst dann fortgesetzt werden, wenn der Event signalisiert ist, also eine Reaktion der Umwelt auf den blockierenden Zustand erfolgt ist.

- Ein Event kann genau eine bestimmte Reaktion erwarten: Boole‘sches Event.
- Ein Event kann aber auch mehrere Reaktionen erwarten: Zählendes Event.

Beispiel boole‘sches Event: Lieferwagen stellt Paket zu. Das Paket stellt hierbei den Event dar. Ein Paket gilt erst nach erfolgreicher Zustellung als zugestellt.

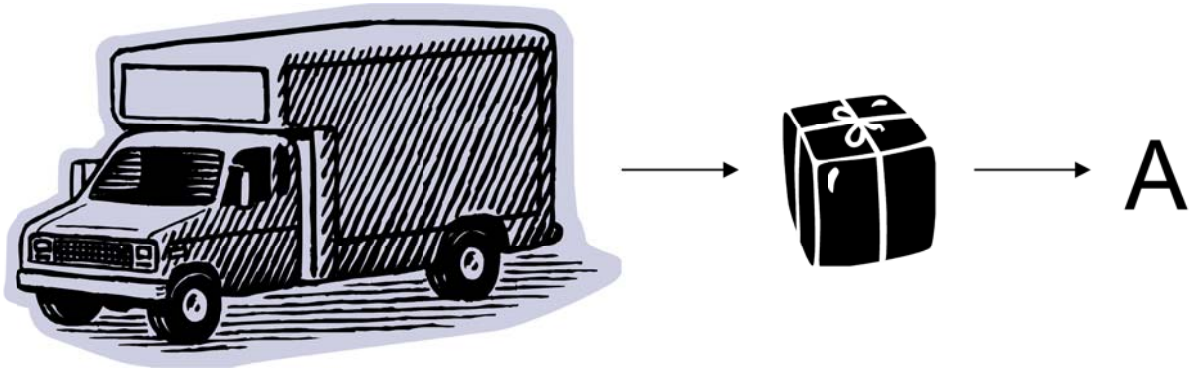


Abbildung 49: Beispiel boole’sches Event

Beispiel zählendes Event: Gleiches Beispiel - Lieferwagen stellt Pakete zu. Jede erfolgreiche Lieferung stellt hierbei einen zählenden Event dar. Der Lieferwagen hat seine Arbeit erst dann erledigt, wenn alle Pakete zugestellt sind.

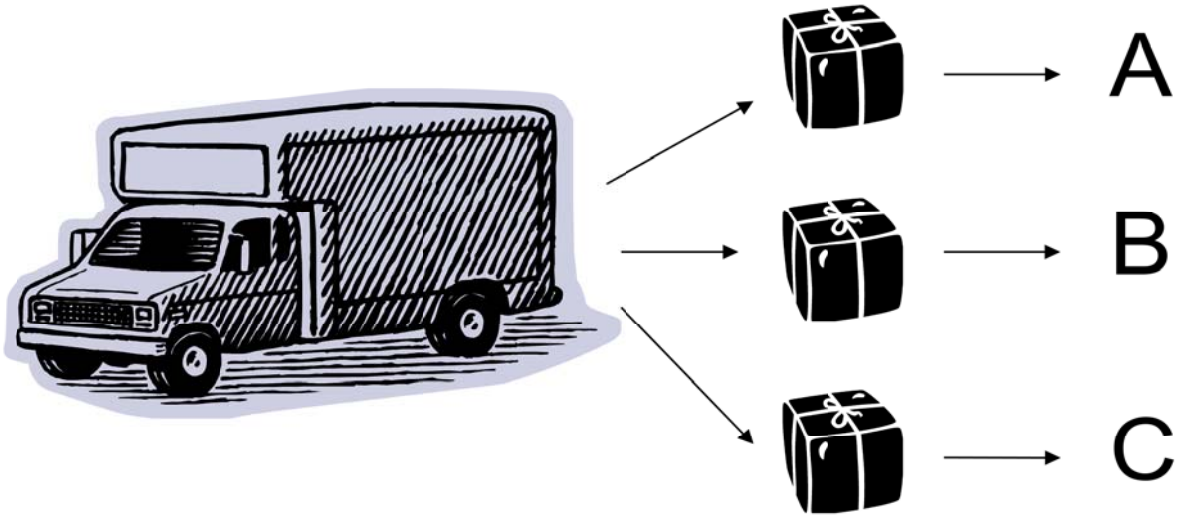


Abbildung 50: Beispiel zählendes Event

2.5.2.5. Signale

Eine weitere Möglichkeit der Prozesssynchronisation stellen Signale dar. Ein Signal ist ein asynchrones Ereignis und bewirkt eine Unterbrechung auf der Prozessebene.

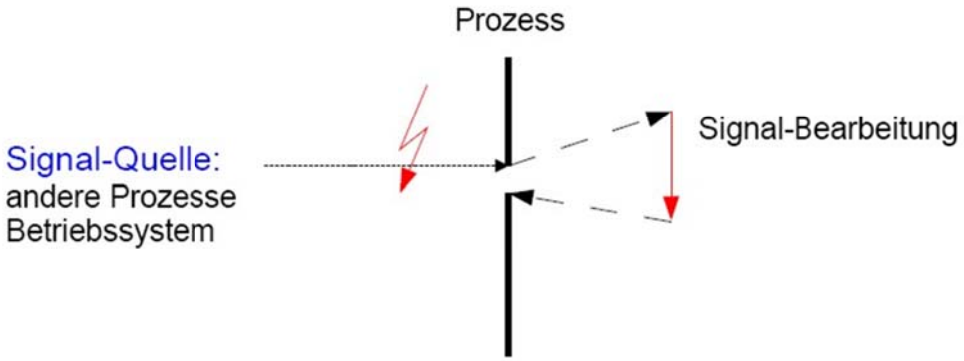


Abbildung 51: Signale

Signale können entweder von außen durch den Benutzer an den Terminal oder durch das Auftreten von Programmfehlern (Adressfehler, Division durch Null usw.) erzeugt oder durch externe Unterbrechung hervorgerufen werden. Unter Unix kann auch ein anderer Prozess mittels Systemcall kill (SIGNAL, PID) ein Signal senden. In Tabelle 19sind die Signale, wie sie in Unix definiert sind aufgelistet.

Vergleich:	Hardware-Interrupt	Signale
Auslöser	durch externes Ereignis	durch einen anderen Prozess (oder durch das Betriebssystem)
Zeitraster	asynchron zum normalen Programm	asynchron (oder synchron)
Selektion der Quelle	möglich durch Maskieren von Interrupt-Quellen	möglich (Maskieren, Ignorieren)
Ort der Bearbeitung	Interrupt Service Routine (ISR)	Signal-Handler (Trap-Handler)

Tabelle 19: Signale und Interrupts

2.5.2.6. Messages

Während Semaphore dazu dienen die Synchronisation der einzelnen Prozesse zu übernehmen, braucht es zusätzlich komplexere Mechanismen zur Kommunikation der einzelnen Prozesse untereinander. In Echtzeitbetriebssystemen dienen sogenannte **Message Queues** zur Interprozesskommunikation. Hierbei unterscheidet man zwischen der Kommunikation unter den Prozessen, die ausschließlich auf einem Prozessor ablaufen und der Interprozesskommunikation in Multiprozessorsystemen. Letzteres bedarf je nach System zusätzliche Hilfsmittel, die meist das Speicherverwaltungssystem betreffen.

Bezüglich der Anzahl beteiligter Prozesse, sowohl auf der Sender- als auch der Empfängerseite können verschiedene Formen des Nachrichtenaustausches unterschieden werden:

Abbildung 52: Nachrichtenaustausch in Form 1:1

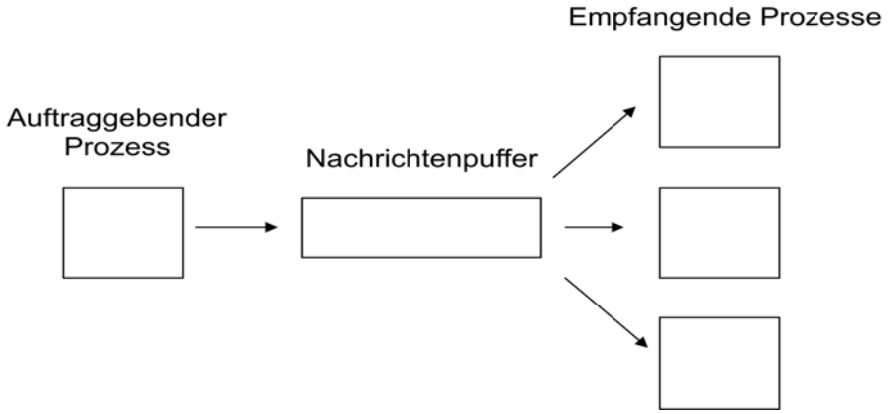


Abbildung 53: Nachrichtenaustausch in Form 1:n

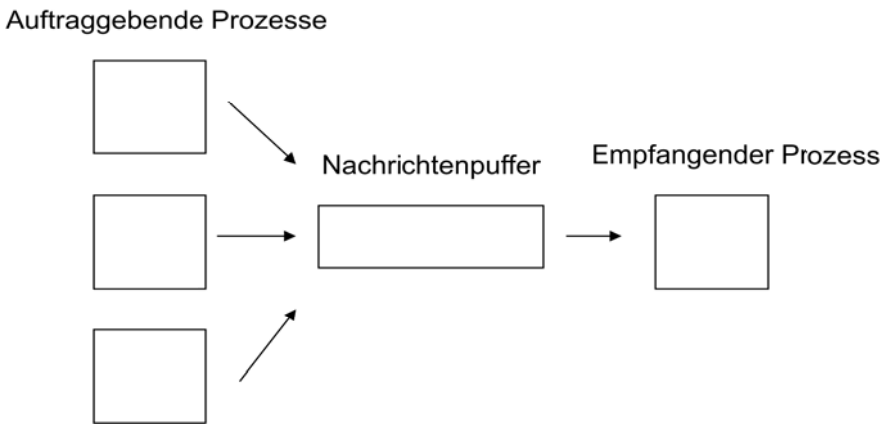


Abbildung 54: Nachrichtenaustausch in Form m:1

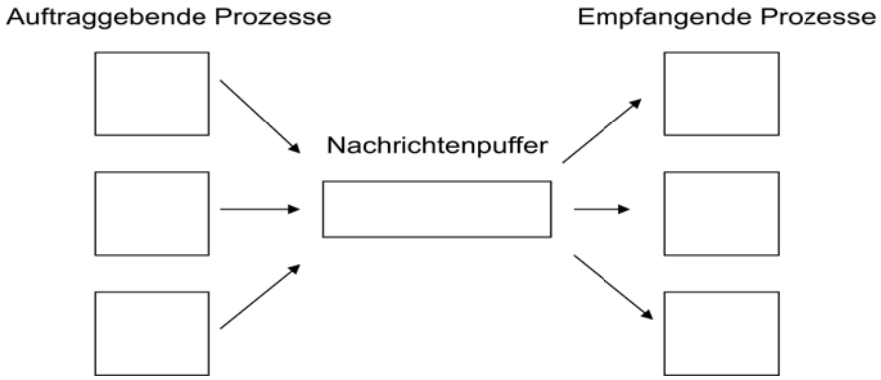


Abbildung 55: Nachrichtenaustausch in Form m:n

Message Queues erlauben das Verschicken von einer beliebigen Anzahl von Mitteilungen mit beliebiger Länge. Dem Mechanismus liegt meist eine **FIFO** (**F**irst **I**n **F**irst **O**ut) Algorithmus zu Grunde. D.h. Mitteilungen, die zuerst

geschickt werden kommen auch zuerst an. Die nachfolgenden Mitteilungen werden in eine Warteschlange eingetragen.

Grundsätzlich können mehrere Prozesse Mitteilungen über eine Message Queue verschicken bzw. mehrere Prozesse können über dieselbe Message Queue Mitteilungen empfangen. Zur Realisierung eines **full-duplex** Modus braucht es allerdings in der Regel zwei Message Queues, jeweils für eine Richtung. Es wird eine **full-duplex** Kommunikation über zwei Message Queues gezeigt.

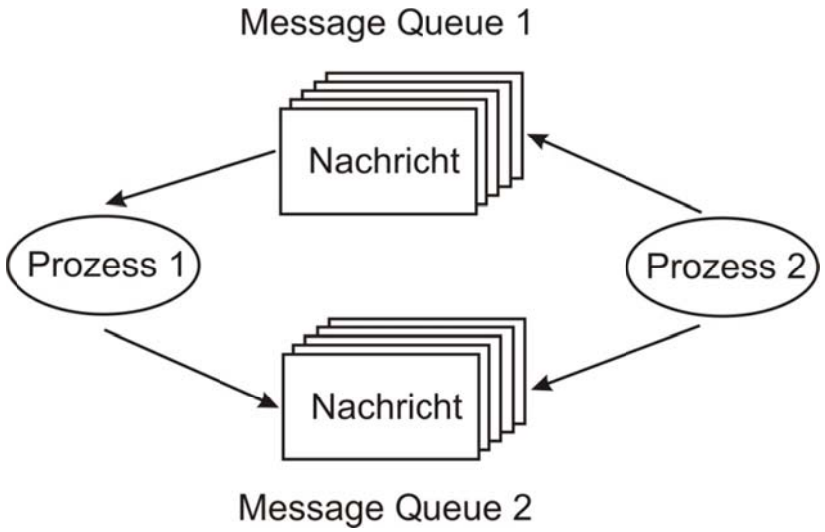


Abbildung 56: Full Duplex Interprozesskommunikation mit Message Queues

2.5.2.7. Spezialisierte Warteschlangen (Pipes)

Sogenannte Pipes können als spezialisiertes Interface zu Message Queues verstanden werden. Pipes existieren im Betriebssystem Unix, wie auch in Echtzeitbetriebssystemen. Pipes sind dort sogenannte virtuelle Geräte. Pipes werden analog zu Message Queues erzeugt, ausgelesen, beschrieben und bei Bedarf wieder gelöscht. Die Verwaltung geschieht durch das Betriebssystem. Es sind die Unterschiede zwischen Message Queues und Pipes dargestellt.

Message Queues	**Pipes**
Möglichkeiten das Blockierungsverhalten zu beeinflussen	keine Beeinflussungsmöglichkeiten
Möglichkeiten die Mitteilungen zu priorisieren	keine Beeinflussungsmöglichkeiten
weniger Kommunikationsoverhead, deswegen schneller	Langsamer
können bei Bedarf während der Laufzeit dynamisch entfernt werden	in der Regel ist die Entfernung erst nach Systemneustart möglich
Anzeigeroutinen sind meistens vorhanden	keine Anzeige
freier Zugriff, weniger formalisiert	nutzt die Standard E/A Schnittstelle des Betriebssystems
keine Möglichkeit	kann für das Umlenken des Standard E/A genutzt werden

Tabelle 20: Unterschiede zwischen Message Queues und Pipes

2.5.2.8. Deadlock

Deadlocks können nicht nur durch die Reservierung von E/A-Geräten (allgemein Betriebsmitteln), sondern auch in vielen anderen Situationen entstehen. In einer Datenbankanwendung könnte eine Anwendung z.B. die Datensätze, an denen sie arbeitet, sperren. Wenn Anwendung A den Datensatz DS1 und die Anwendung B den Datensatz DS2 sperren und anschließend jede Anwendung versucht, den Datensatz der anderen Anwendung zu sperren, entsteht ebenfalls ein Deadlock.

Deadlocks können also sowohl durch Hardware- als auch durch Softwareressourcen entstehen.

Abbildung 57: Deadlock

3. Methode zur Modellierung und zum Entwurf

Für den Entwurf von diskreten Steuerungen für diskrete Prozesse und Abläufe werden Beschreibungsverfahren benötigt. Von den verschiedenen Möglichkeiten seien an dieser Stelle nur drei genannt:

1. Beschreibung und Modellierung durch **Petri-Netze**,
2. Beschreibung und Modellierung durch **Programmablaufgraphen** und
3. Beschreibung und Modellierung durch **Automaten(-graphen)**.

3.1. Automaten

Eine Methode zur Beschreibung und Modellierung von diskreten Prozessen und Abläufen verwendet als Grundlage Automaten. Oft werden auch Automaten zur Steuerung von technischen Prozessen eingesetzt.

Zunächst sollen die prinzipiellen Gemeinsamkeiten verschiedenartiger Automaten herausgearbeitet werden. Es geht insbesondere darum, wie sich die Arbeitsweise (das Verhalten) der Automaten losgelöst von den physikalischen Komponenten eines realen Automaten hinreichend genau beschreiben und modellieren lässt.

Typisch für einen Automaten ist, dass er von außen bedient wird, d.h. er wird mit Eingabedaten versorgt. Die Eingabe oder Bedienung wird in der Umgangssprache durch Formulierungen wie *„Knopf - Fahrziel 1. Etage“*, *„Knopf - Tür schließen“* oder *„Knopf - nach oben fahren“* ausgedrückt. Diese Eingabemöglichkeiten werden durch Zeichen des sogenannten **Eingabealphabetes** repräsentiert. Man verwendet die folgende Sprechweise:

Definition 17: Eingabe

Es gibt eine endliche Menge E von Eingabezeichen, die aus endlich vielen Zeichen e_i bestehen. Bei einer bestimmten Eingabe „wirkt“ dann ein Zeichen $e_i \in E$. Die endliche Menge E heißt Eingabealphabet.

Ähnliches gilt für die inneren Zustände. Ein Automat befindet sich stets in einem bestimmten Zustand. Unter Einwirkung der Eingabe kann er die Zustände wechseln und eine Abfolge von Zuständen durchlaufen. Bei der umgangssprachlichen Beschreibung der Automaten werden die Zustände durch Formulierungen der Art „*Tür offen*“, „*Fahren nach oben*“ oder „*Halt*“ repräsentiert. Die Zustände werden ebenfalls durch Zeichen dargestellt. In der Automatentheorie wird folgende Sprechweise verwendet:

Definition 18: Zustand

Es gibt eine endliche Menge S von internen Zuständen. Ein Zustand wird durch s_i repräsentiert. Der Automat befindet sich in einem bestimmten Zustand si oder er geht von Zustand s_i in den neuen Zustand s_j über s_i, $s_j \in S$. Die endliche Menge S heißt Zustandsmenge.

Die Ausgaben, die der Automat im Laufe seiner Arbeit erzeugt, werden umgangssprachlich durch „*Lampe Etage 3 an*“, „*Motor linkrum drehen*“ oder „*Lampe Fahrtrichtung nach oben an*“ beschrieben. In der Automatentheorie gelten hierfür folgende Konventionen:

Definition 19: Ausgabe

Es gibt eine endliche Menge Z von Ausgabezeichen, die aus endlich vielen Zeichen $z_i \in Z$ bestehen. Bei einer bestimmten Ausgabe wird das Zeichen $z_i \in Z$ vom Automaten ausgegeben. Die endliche Menge Z heißt Ausgabealphabet.

Mit diesen Konventionen können Automaten noch nicht vollständig beschrieben werden. Es ist noch nicht möglich, den dynamischen Ablauf oder das Verhalten eines Automaten darzustellen. Zur Repräsentation des Verhaltens wird das Zustandsdiagramm bzw. der Automatengraph eingeführt, siehe Abbildung 58.

Der Automatengraph ergibt sich, wenn man für jeden Zustand einen Knoten zeichnet. Von jedem Knoten gehen so viele gerichtete Kanten aus, wie es Eingabezeichen gibt. Die Kante endet bei demjenigen Knoten, in den der Automat beim Lesen des Zeichens übergeht.

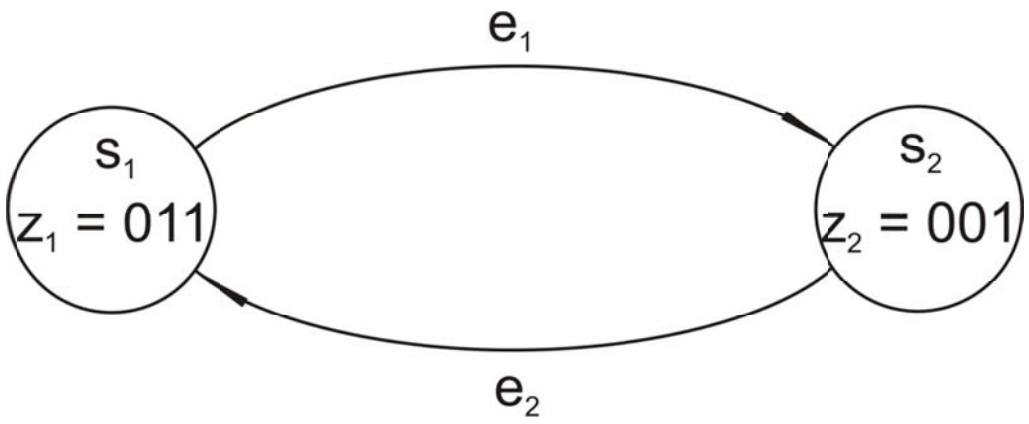


Abbildung 58: Zustandsdiagramm bzw. Automatengraph

Die Kante wird mit diesem Eingabezeichen beschriftet. Die Zustände s_i werden gemeinsam mit den Zeichen der Ausgabe z_i in den Knoten übernommen.

Nachdem nun alle Komponenten eines endlichen Automaten eingeführt sind, lässt sich unmittelbar eine exakte Definition angeben:

Definition 20: endlicher Automat

Ein (endlicher) Automat ist ein Sechstupel
$\alpha = (E, Z, S, u, o, s_0)$ mit
$E = \{e_1, e_2, \ldots\}$ eine endliche, nichtleere Menge, das Eingabealphabet,
$Z = \{z_1, z_2, \ldots\}$ eine endliche, nichtleere Menge, das Ausgabealphabet,
$S = \{s_1, s_2, \ldots\}$ eine endliche, nichtleere Menge, die Zustandsmenge,
u: Zustandsüberführungsfunktion,
o: Ausgabefunktion und
s_0: der Anfangszustand.

Für die Modellierung von diskreten Prozessen verwendet man häufig endliche Automaten ohne Ausgabe. Bei diesen wird auf das Ausgabealphabet und die Ausgabefunktion verzichtet.

Verallgemeinert kann festgestellt werden, dass sich Automaten immer dort einsetzen lassen, wo Prozesse und Abläufe durch diskrete Zustände mit Zustandswechseln gekennzeichnet sind. Ein Beispiel hierfür ist das Prozessmodell eines Betriebssystems, siehe Abbildung 59.

Hier können die einzelnen Prozesszustände als Zustände in einem endlichen Automaten definiert werden. Das Prozessmodell besteht aus fünf Zuständen:

1. Zustand „Nicht existent“,
2. Zustand „Passiv“,
3. Zustand „Bereit“,
4. Zustand „Laufend“ und
5. Zustand „Suspendiert“

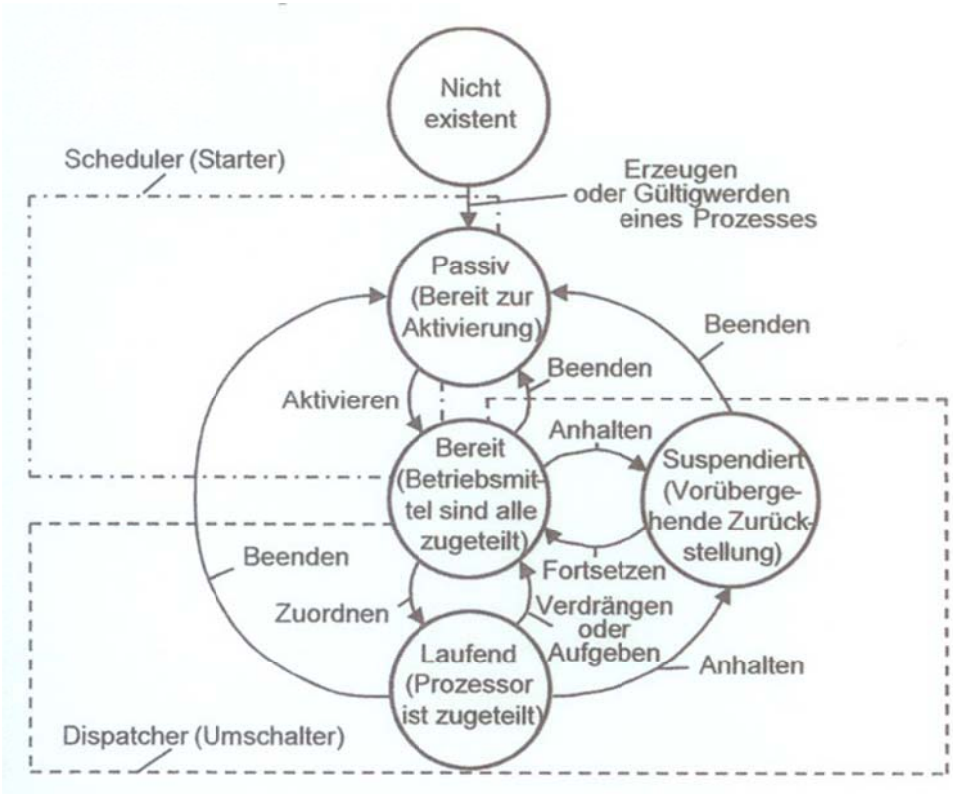


Abbildung 59: Prozessmodell in Betriebssystemen

Zu Beginn sind alle Prozesse im Zustand „Nicht existent“ und wechseln dann nach der Erzeugung in den Zustand „Passiv“.

Im Zustand „Passiv“ ist der Prozess nicht ablaufbereit. Die Zeitvoraussetzungen sind nicht erfüllt, oder es fehlen die notwendigen Betriebsmittel, um starten zu können. Sobald diese Bedingungen erfüllt sind, wechselt der Prozess in den Zustand „Bereit“.

Im Zustand „Bereit“ sind alle Betriebsmittel zugeteilt. Bekommt der Prozess auch den Prozessor zugeteilt, so wechselt er in den Zustand „Laufend“.

Wird einem laufenden Prozess ein Betriebsmittel entzogen, so wird der Prozess suspendiert (Zustand „Suspentiert“).

3.2. Abbildung der Benutzerwelt auf die Maschinenwelt

Der Anwender wird von Detailkenntnissen über die physikalisch-technischen Parameter des Rechnersystems entlastet. (Man kann mit einem Computer arbeiten, ohne die Vorgänge am Halbleiter-PN-Übergang zu kennen.)

Es erfolgt eine Transformation aus der logisch-organisatorisch opportunen Umwelt eines Problems in die physikalisch-technisch notwendige Umwelt der Rechneranlage und umgekehrt.

Solche Transformationen spielen sich im Wesentlichen im Ein- / Ausgabesystem der Rechneranlage ab, siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**.

3.3. Koordination und Organisation des Betriebsablaufes

Bei der Steuerung von technologischen Abläufen durch eine Rechneranlage gibt es häufig technologisch voneinander unabhängige Teilabläufe. Es ist günstig, wenn die Steuerung der Teilabläufe durch gleichzeitig im Rechner abzuarbeitende Programme erfolgt, siehe Abschnitt 0.

Technologische Zerlegungen des technischen Ablaufes widerspiegeln sich in ähnlichen Zerlegungen des steuerenden Programmsystems.

Beispiel einer einfachen Steuerungsaufgabe:

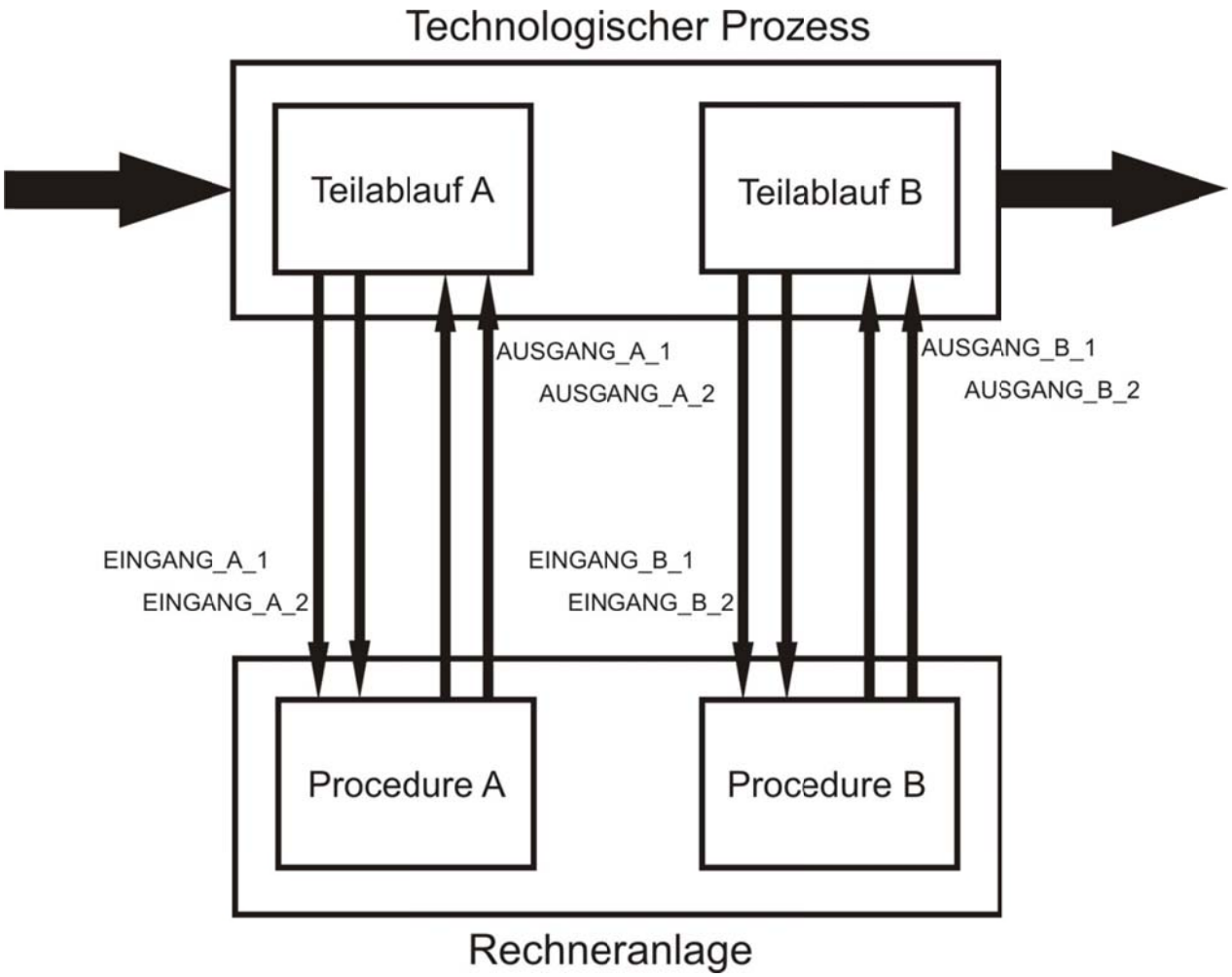


Abbildung 60: einfache Steuerungsaufgabe

EREIGNIS_A_1, EREIGNIS_A_2, EREIGNIS_B_1 und EREIGNIS_B_2 sind PL/SQL-Funktionen, die das Eintreten jeweils eines Ereignisses (EINGANG_A_i bzw. EINGANG_B_i) im technischen Ablauf testen und den booleanschen Wert TRUE oder FALSE zurückgeben:

```
FUNCTION EREIGNIS_A_1 RETURNS BOOLEAN
 Argument Name                  Typ                     In/Out Defaultwert?
 ------------------------------ ----------------------- ------ --------
 EINGANG_A_1                    BOOLEAN                 IN

FUNCTION EREIGNIS_A_2 RETURNS BOOLEAN
 Argument Name                  Typ                     In/Out Defaultwert?
 ------------------------------ ----------------------- ------ --------
 EINGANG_A_2                    BOOLEAN                 IN

FUNCTION EREIGNIS_B_1 RETURNS BOOLEAN
 Argument Name                  Typ                     In/Out Defaultwert?
 ------------------------------ ----------------------- ------ --------
 EINGANG_B_1                    BOOLEAN                 IN

FUNCTION EREIGNIS_B_2 RETURNS BOOLEAN
 Argument Name                  Typ                     In/Out Defaultwert?
 ------------------------------ ----------------------- ------ --------
 EINGANG_B_2                    BOOLEAN                 IN
```

AKTION_A_1, AKTION_A_2, AKTION_B_1 und AKTION_B_2 sind PL/SQL-Prozeduren, die bestimmte Aktionen im technologischen Ablauf auslösen:

```
PROCEDURE AKTION_A_1
 Argument Name                  Typ                       In/Out Defaultwert?
 ------------------------------ ------------------------- ------ --------
 AUSGANG_A_1                    BOOLEAN                   OUT

PROCEDURE AKTION_A_2
 Argument Name                  Typ                       In/Out Defaultwert?
 ------------------------------ ------------------------- ------ --------
 AUSGANG_A_2                    BOOLEAN                   OUT

PROCEDURE AKTION_B_1
 Argument Name                  Typ                       In/Out Defaultwert?
 ------------------------------ ------------------------- ------ --------
 AUSGANG_B_1                    BOOLEAN                   OUT

PROCEDURE AKTION_B_2
 Argument Name                  Typ                       In/Out Defaultwert?
 ------------------------------ ------------------------- ------ --------
 AUSGANG_B_2                    BOOLEAN                   OUT
```

Die Funktionen EREIGNIS_A_i und Prozeduren AKTION_A_i einerseits und die Funktionen EREIGNIS_B_i und Prozeduren AKTION_B_i andererseits sind zwei voneinander unabhängigen Teilabläufen (A und B) zugeordnet.

Die Ereignisfunktionen der beiden Teilabläufe nehmen unabhängig voneinander und von den Aktionsprozeduren des jeweils anderen Teilablaufes die Werte true und false ein.

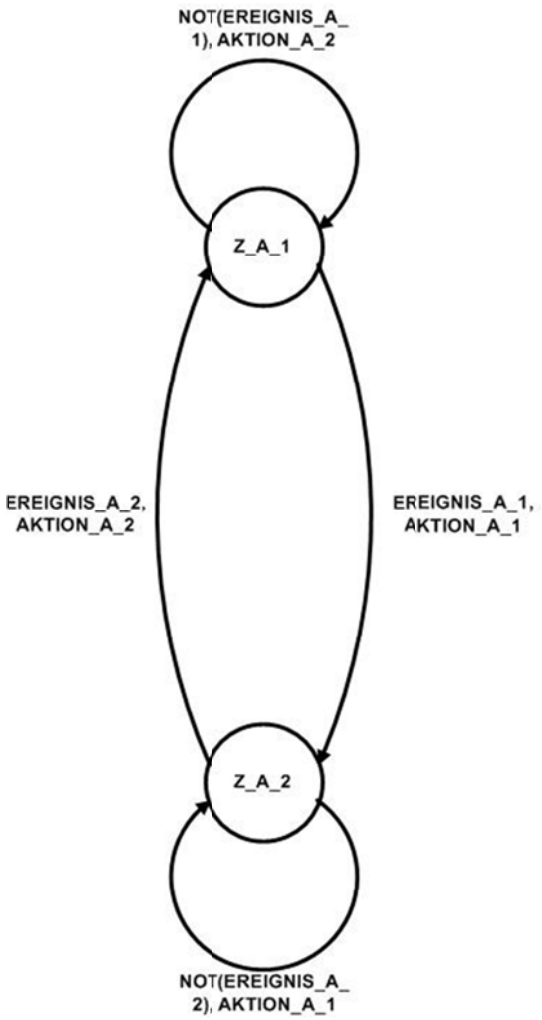
Prozedur A für Teilablauf A:

```
procedure A
is
begin
  loop
    while not EREIGNIS_A_1(V_EINGANG_A_1) loop
      AKTION_A_2(V_AUSGANG_A_2);
    end loop;
    AKTION_A_1(V_AUSGANG_A_1);
    while not EREIGNIS_A_2(V_EINGANG_A_2) loop
      AKTION_A_1(V_AUSGANG_A_1);
    end loop;
    AKTION_A_2(V_AUSGANG_A_2);
  end loop;
end;
```

Prozedur B für Teilablauf B:

```
procedure B
is
begin
  loop
    while not EREIGNIS_B_1(V_EINGANG_B_1) loop
      AKTION_B_2(V_AUSGANG_B_2);
    end loop;
    AKTION_B_1(V_AUSGANG_B_1);
    while not EREIGNIS_B_2(V_EINGANG_B_2) loop
      AKTION_B_1(V_AUSGANG_B_1);
    end loop;
    AKTION_B_2(V_AUSGANG_B_2);
  end loop;
end;
```

Die beiden Prozeduren **A** und **B** können in zwei gleichzeitig durch das Betriebssystem abzuarbeitenden Programmen benutzt werden.



Jede Prozedur kann als Realisierung eines Automaten, siehe

Abbildung 61 und Abbildung 62, mit je zwei Zuständen aufgefasst werden. Dabei hat jeder Zustand **Z_A_i** und **Z_B_i** jeweils eine wegführende Kante und eine Eigenschleife.

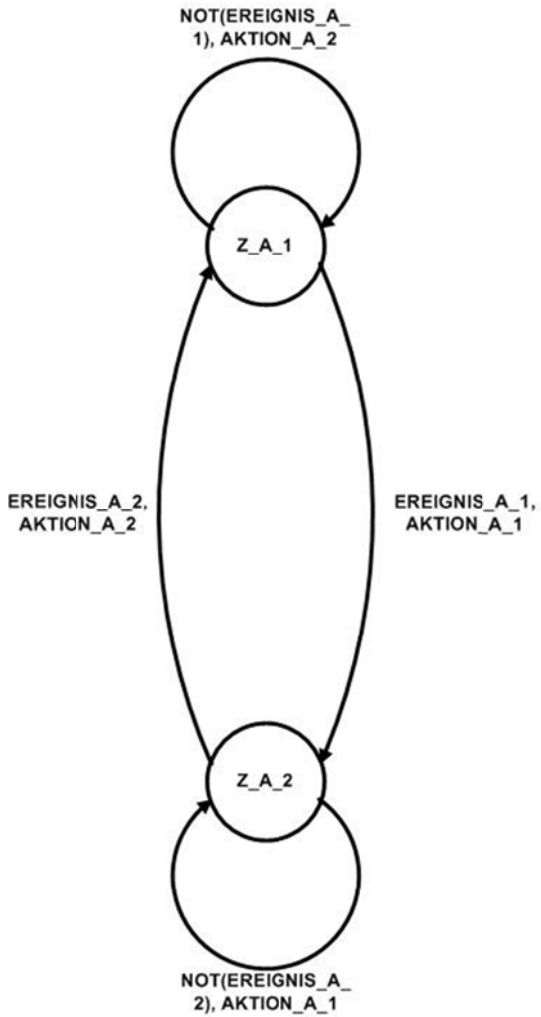


Abbildung 61: Automatengraph für Procedure A

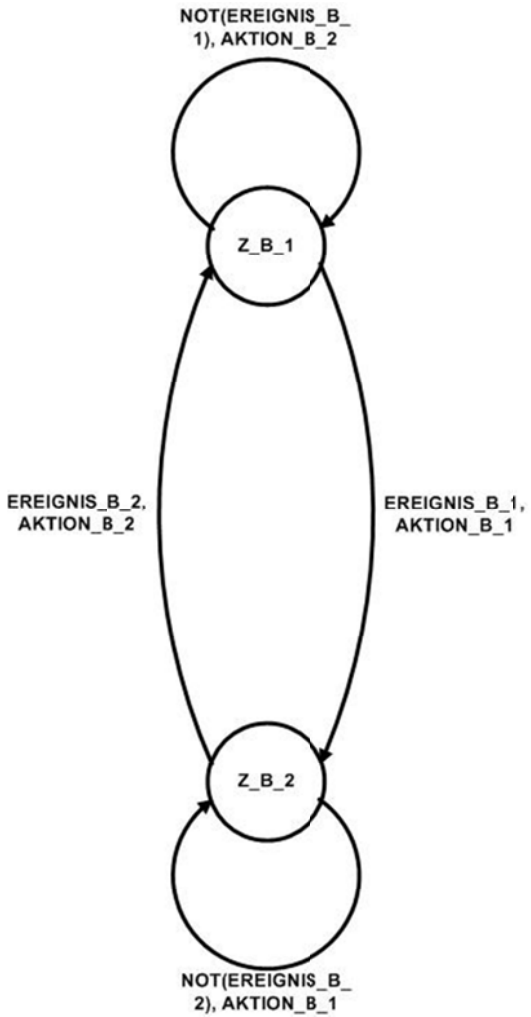
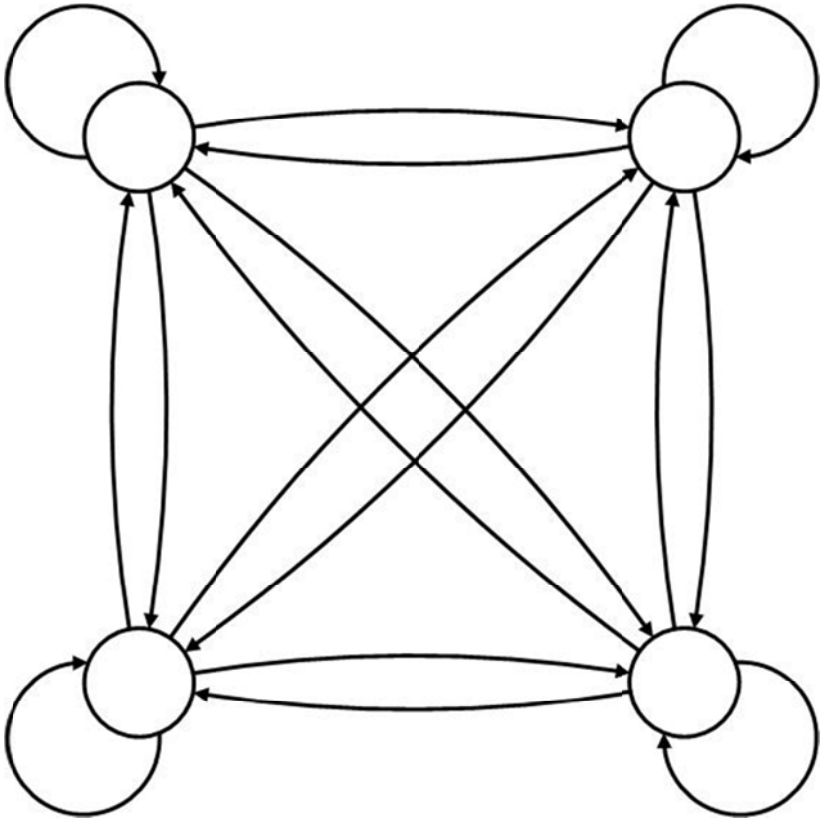


Abbildung 62: Automatengraph für Procedure B



Soll die Steuerung in einer Prozedur dargestellt werden, so ist ohne Verwendung von Unterprogrammen (Unterautomaten) die Komposition beider Teilautomaten zu realisieren. Es entsteht ein Automat mit vier Zuständen und Kantengewichten zwischen allen Zuständen, siehe Abbildung 63.

Die entstehenden Kantengewichte entstehen aus der Konjunktion der Eingangsfunktionen:

EREIGNIS_A_i^EREIGNIS_B_i,AKTION_A_i u AKTION_B_i

Vereinigung der Wirkung – Setzen von Ausgangsvariablen auf disjunkten technischen Teilprozessen.

Es entsteht eine wesentlich komplexere und unnötige, unübersichtliche Struktur.

Abbildung 63: Automatengraph mit vier Zuständen

4. Multiprozessorsysteme

Einsteins spezieller Relativitätstheorie zufolge kann sich kein elektrisches Signal schneller als Licht fortbewegen, also ungefähr 30 cm/ns im Vakuum und ca. 20 cm/ns in Kupfer oder Lichtleiter. Das bedeutet, dass in einer 10 GHz-CPU das Signal in einem Takt nicht weiter als insgesamt 2 cm kommen kann. Diese Wegstrecke beträgt bei einem 100 GHz-Computer dann höchstens noch 2 mm.

Um das Signal in einem Zyklus von dem einen Ende des Chips zum anderen und zurückkommen zu lassen, müsste ein 1000-GHz-Computer kleiner als 100 Mikron sein. Damit tritt ein weiteres Problem auf, die Hitzeentwicklung. Je schneller der Computer, desto größer seine Hitzeabstahlung. Umgekehrt gilt, je keiner der Chip ist, desto schwieriger wird es, die Hitze abzuleiten. Schon jetzt ist auf einen High-End-Pentium-System der Kühler größer als die CPU selbst.

Ein Ansatz für mehr Geschwindigkeit, bzw. mehr Leistung basiert auf massiver Parallelisierung von Computern. Diese Maschinen bestehen aus vielen CPUs, jede einzelne mit „normaler“ Geschwindigkeit, gegenwärtig um die 3 GHz. Damit erreicht das Gesamtsystem eine wesentlich höhere Rechenleistung als eine einzelne CPU. Systeme mit 1000 CPUs sind schon kommerziell verfügbar. Solche mit 1 Million CPUs sind nur eine Frage der Zeit.

Hochgradig parallele Computer werden oft für extreme numerische Berechnungen benötigt. Probleme wie die Wettervorhersage, Simulationen bzgl. Der Weltwirtschaft oder das Verständnis um die Funktionsweise von Rezeptoren im Gehirn sind sehr rechenintensiv.

Eine andere relevante Entwicklung ist die unglaublich schnelle Ausbreitung des Internets. Dieses erfüllt Aufgaben in vielen Bereichen. Eine davon ist es, Tausende von Rechnern in der ganzen Welt zu verbinden, um gemeinsam an einem wissenschaftlichen Problem zu arbeiten, z.B. am CERN in der Schweiz. Grundsätzlich besteht kein Unterschied zwischen einem System von 1000 Computern, die auf der ganzen Welt verstreut sind, und einem System mit 1000 Computern in einem Raum, obwohl es selbstverständlich Unterschiede in der Verbindungsgeschwindigkeit und anderen technischen Details gibt.

Ein Extrem sind Shared-Memory-Multiprozessoren, Systeme, in denen zwischen zwei und 1000 CPUs über einen gemeinsamen Speicher kommunizieren, siehe Abbildung 64 (a). Der Zugriff benötigt in der Regel 10-50 ns. Das Modell sieht zwar einfach aus, seine Implementierung ist aber alles andere als das.

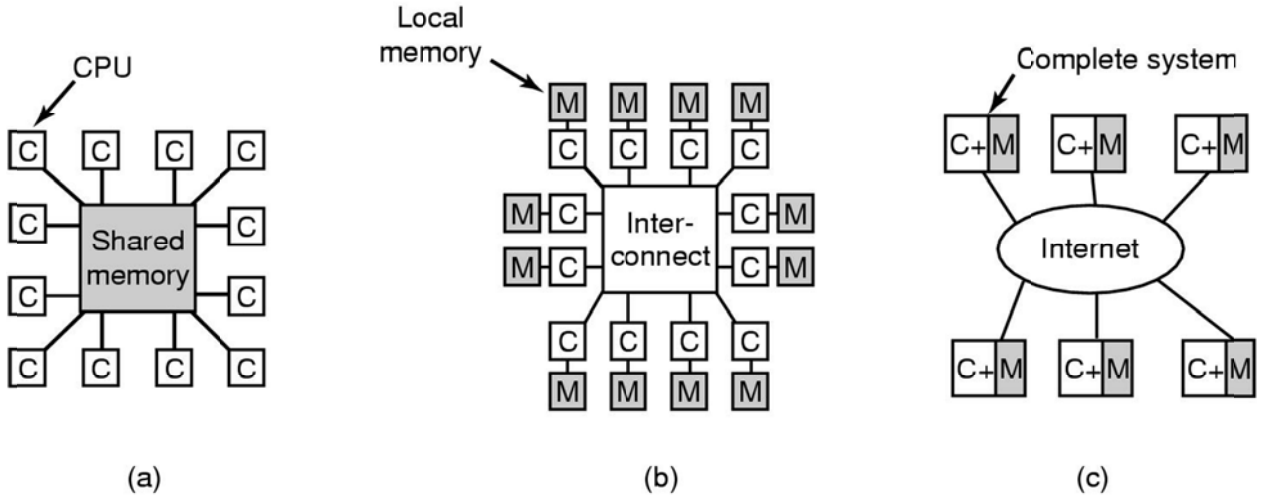


Abbildung 64: Arten von Multiprozessorsystemen

In Abbildung 64 (b) sind CPU-Speicher-Paare durch Hochgeschwindigkeitsverbindungen untereinander verknüpft. Ein solches System wird Message-Passing-Multicomputer genannt. Jede Speichereinheit ist lokal mit

einer CPU verbunden und der Zugriff kann nur durch diese CPU erfolgen. Mit guten Verbindungen können kurze Mehrwort-Nachrichten in 10-50 µs verschickt werden. Das ist zwar um den Faktor 1000 mal langsamer, aber da es im Design keinen gemeinsamen Speicherbereich gibt, sind diese Systeme viel leichter zu realisieren, jedoch schwerer zu programmieren.

Das dritte Modell, siehe Abbildung 64 (c), verbindet ganze Computeranlagen über das Internet. Das Ergebnis ist ein sogenanntes verteiltes System. Jeder dieser Computer hat natürlich seinen eigenen Speicher und die Systeme kommunizieren durch Nachrichtenaustausch miteinander.

4.1. Multiprozessoren

Ein Shared-Memory-Multiprozessoren – kurz Multiprozessor genannt – ist ein Computersystem, in welchem sich zwei oder mehr CPUs unbeschränkten Zugriff auf einen gemeinsamen RAM teilen. Gleichgültig, auf welcher CPU ein Programm abläuft, es sieht den normalen virtuellen Adressraum. Die einzige ungewöhnliche Eigenschaft dieses Systems: Eine CPU kann einen Wert in den Speicher schreiben, dann von der gleichen Speicheradresse laden und einen anderen Wert erhalten, weil eine andere CPU jenen Wert geändert hat. Korrekt organisiert stellt diese Eigenschaft die Grundlage für Interprozesskommunikation dar – eine CPU schreibt Daten in den Speicher und eine andere liest die Daten aus.

Multiprozessor-Betriebssysteme sind größtenteils gewöhnliche Betriebssysteme. Sie behandeln Systemaufrufe, verwalten den Speicher, stellen ein Dateisystem zur Verfügung und steuern Ein-/Ausgabe-Geräte.

Die einfachsten Multiprozessoren basieren auf einem einzigen Bus, siehe Abbildung 65 (a). Zwei oder mehrere CPUs und ein oder mehrere Speichermodule benutzen gemeinsam denselben Bus für die Kommunikation. Soll eine CPU ein Speicherwort lesen, so überprüft sie zuerst, ob der Bus belegt ist. Sobald er frei ist, kann das Speicherwort gelesen werden. Ansonsten wartet die CPU einfach, bis der Bus wieder frei ist. Und darin liegt auch das Problem dieses Designs. Bei zwei oder drei CPUs ist der gemeinsame Zugriff auf den Bus noch zu

behandeln, mit 32 oder 64 CPUs wird der Zugriff auf den Bus jedoch unmöglich. Das System wird dann komplett durch die Bandbreite des Busses beschränkt und die meisetn CPUs sind den größten Teil der zeit ungenutzt.

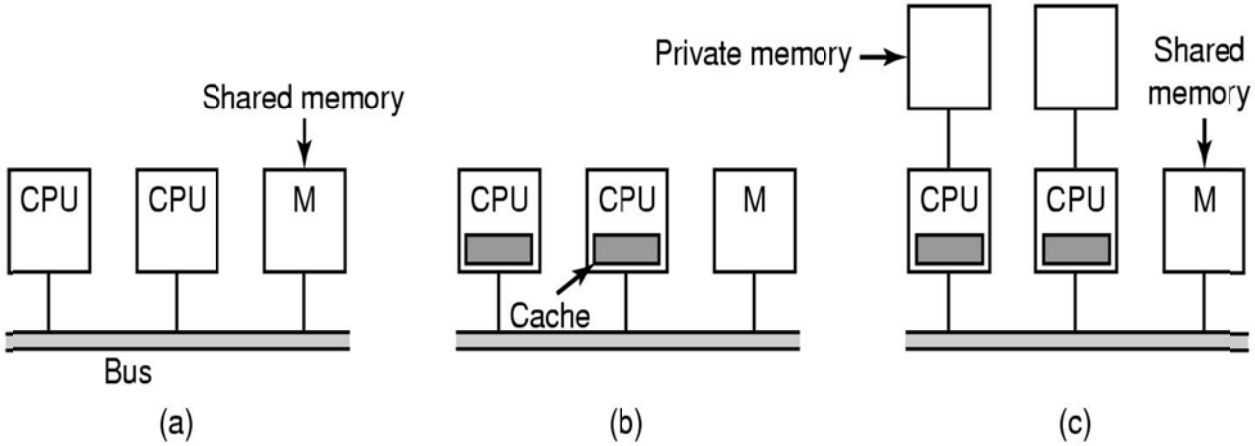


Abbildung 65: Arten von Multiprozessoren

Die Lösung für dieses Problem ist, wie in Abbildung 65 (b) gezeigt, ein Cache für jede CPU. Da nun viele Lesezugriffe aus dem lokalen Cache bedient werden können, gibt es viel weniger Zugriffe auf den Bus und das System ist in der Lage, mehr CPUs zu unterstützen. Im Allgemeinen werden nicht einzelne Speicherwörter gecached. Vielmehr werden ganze Blöcke von üblicherweise 32 bis 64 byte zwischengespeichert. Wenn ein Wort referenziert wird, so wird im Zuge dessen sein ganzer Block in den Cache der betreffenden CPU geladen.

Eine weitere Möglichkeit ist in Abbildung 65 (c) illistriert. Jede CPU hat nicht nur einen Cache, sondern auch einen lokalen privaten Speicher, auf welchen über einen privaten dedizierten Bus zugegriffen wird. Um eine derartige Konfiguration optimal zu nutzen, sollte der Compiler den Programmtext, Strings, Konstante und andere readonly Daten, Stacks und lokale Variablen in den privaten Speicher legen. Der gemeinsame Speicher sollte nur für veränderbare gemeinsame Variablen benutzt werden.

4.1.1. Betriebssysteme für Multiprozessoren

Bezüglich der Verwendung von Betriebssystemen im Multiprozessoren gibt es verschiedene Organisationsmöglichkeiten, die im folgenden näher betrachtet werden sollen.

4.1.1.1. Jede CPU hat ihr eigenes Betriebssystem

Der einfachste Weg, ein Multiprozessor-Betriebssystem zu organisieren, sieht wie folgt aus: Man unterteilt den Speicher in so viele Partitionen, wie es CPUs gibt, und teilt jeder CPU ihren eigenen Speicher und ihre eigene Kopie des Betriebssystems zu. Im Endeffekt arbeiten die **n** CPUs dann wie **n** unabhängige Computer. Eine offensichtliche Optimierung wäre die gemeinsame Nutzung des Systemcodes, sodass jede CPU nur über private Kopien der Daten verfügen muss. Abbildung 66 verdeutlicht diesen Ansatz.

Obige Möglichkeit ist immer noch besser als **n** eigenständige Computer, da alle Maschinen gemeinsamen Zugriff auf die Datenträger und die Ein-/Ausgabe-Geräte haben, ebenso kann der Speicher gemeinsam und flexibel genutzt werden.

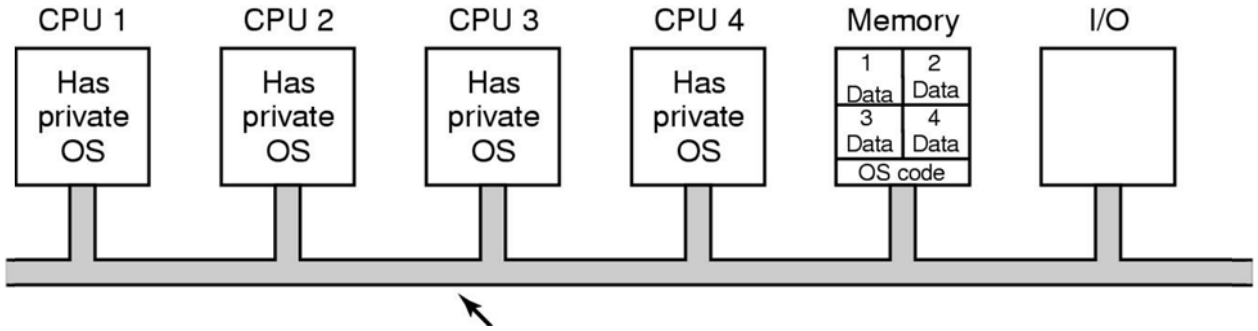


Abbildung 66: jede CPU hat ihr eigenes Betriebssystem

Der gemeinsame Speicher kann dynamisch, je nach zu bearbeitenden Aufgaben den einzelnen CPUs zugewiesen werden und die Kommunikation zwischen den einzelnen Prozessen beschränkt sich auf das Ablegen von Daten im gemeinsamen Speicher, von dem ein anderer Prozess die Daten holen kann.

Die vier folgenden Aspekte sind nicht offensichtlich und sollen hier eine Erwähnung finden.

1. Wenn ein Prozess einen Systemaufrufruf auslöst, so wird dieser von der eigenen CPU abgefangen und unter Verwendung der Datenstrukturen in den Tabellen ihres Betriebssystems behandelt.
2. Da jedes Betriebssystem seine eigenen Tabellen hat, hat es auch seine eigenen Prozesse, die es eigens verwaltet. Prozesse werden nicht gemeinsam genutzt, d.h., loggt sich ein Benutzer auf CPU1 ein, so laufen auch alle seine Prozesse auf dieser CPU ab. Als Konsequenz kann es passieren, dass CPU2 nichts zu tun hat, während CPU1 komplett ausgelastet ist.
3. Speicherseiten werden nicht gemeinsam genutzt. So kann der Fall eintreten, dass CPU1 freie Speicherseiten zur Verfügung stehen, während CPU2 ständig auslagern muss. Für CPU2 gibt es keine Möglichkeit, sich Seiten von CPU1 zu borgen, da die Speicherallokation fix ist.
4. Der schlimmste Fall ergibt sich aus der Verwendung von Caches für kürzlich verwendete Festspeicherblöcke. Alle Betriebssysteme verwalten solche Caches unabhängig voneinander. Folglich kann es sein, das sich ein bestimmter Block geändert in mehreren Caches befindet. Inkonsistenzen sind unvermeidbar.

4.1.1.2. Master-Slave-Multiprozessoren

Aus den genannten Gründen wird obiges Modell kaum noch verwendet. In den Anfängen der Multiplozessorsystemen kam es noch zum Einsatz, mit dem Ziel, ein bestehendes Betriebssystem möglichst schnell an neue Multiprozessoren anzupassen.

Einen weiteren Modellansatz zeigt Abbildung 67. Hier befindet sich eine Kopie des Betriebssystems und seine Tabellen auf einer CPU und auf keiner anderen. Alle Systemaufrufe werden auf CPU1 umgeleitet und dort bearbeitet. Ist Rechenzeit verfügbar, so können auf CPU1 durchaus auch Benutzerprozesse ablaufen.

Dieses Modell wird Master-Slave genannt. CPU1 ist der Master, die anderen CPUs die Slaves. Das Master-Slave-Modell löst die meisten Probleme des ersten Modells.

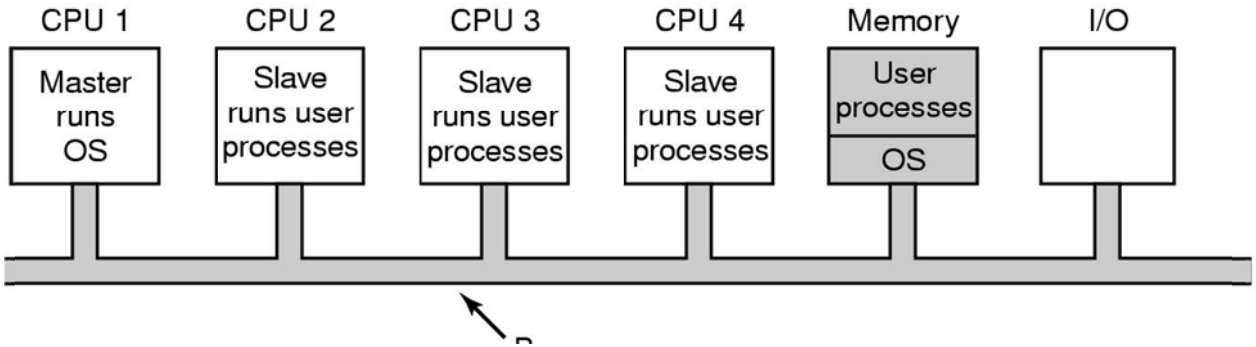


Abbildung 67: Master-Slave-Multiprozessoren

Eine einzige Datenstruktur – z.B. eine Liste oder eine Menge von Listen mit zugrunde liegender Ordnung – hält die Informationen über alle rechenwilligen Prozesse. Hat eine CPU Rechenzeit zur Verfügung, so fordert sie vom betriebssystem einen rechenbereiten Prozess an und bekommt alsbald einen zugeteilt. Folglich kann es nicht passieren, dass eine CPU ausgelastet ist, während eine andere nichts zu tun hat. In ähnlicher Weise werden die Speicherseiten unter allen Prozessen dynamisch verteilt und da nur ein Festspeicher-Cache existiert, sind inkonsistenzen ausgeschlossen.

Das Problem dieses Ansatzes ist jedoch, dass bei vielen CPUs der Master zum Flaschenhals wird. Schließlich muss er die Systemaufrufe aller CPUs behandeln. Daher ist dieser Ansatz einfach und praktikabel für kleine Multiprozessorsysteme, für große jedoch nicht.

4.1.1.3. Symmetrische Multiprozessoren

Das dritte Modell, die symmetrischen Multiprozessoren, eleminieren obige Asymmetrie. Eine Kopie des Betriebssystems befindet sich im Speicher, doch jede CPU kann sie ausführen. Tritt ein Systemaufruf auch, so wechselt diejenige CPU, auf der der Systemaufruf stattfand, in den Kernelmodus und behandelt den Aufruf. Abbildung 68 zeigt den entsprechenden Modellansatz.

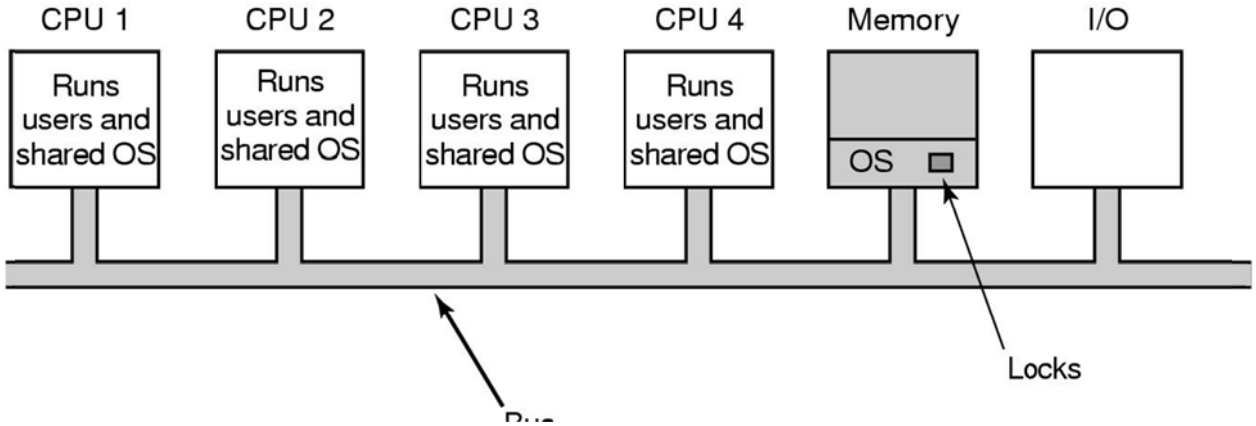


Abbildung 68: Symmetrische Multiprozessoren

Dieser Ansatz balanciert Prozesse und Speicher dynamisch aus, da nur ein Satz der Betriebssystemtabellen vorhanden ist. Außerdem umgeht man den Master-Flaschenhals.

Die Lösung hat jedoch ihre eigenen Probleme. Insbesondere, wenn zwei oder mehr CPUs Betriebssystemcode ausführen, mommt es zur Katastrophe. Vorstellbar ist es, dass zwei CPUs, die den gleichen Prozess ausführen oder die gleiche freie Speicherseite beanspruchen. Der einfachste Weg , dieses Problem zu umgehen, ist, einen Mutex – d.h. einen Lock – einzuführen, der aus dem ganzen Betriebssystem einen einzigen kritischen Abschnitt macht. Möchte eine CPU Betriebssystemcode ausführen, so muss sie zuerst den Mutex passieren. Ist der Mutex gesprerrt, dann wartet sie einfach. Auf diese Weise kann jede CPU das Betriebssystem ausführen, jedoch nur eine gleichzeitig.

Dieser Ansatz funktioniert, doch er ist fast so schlecht wie das Master-Slave-Modell. Glücklicherweise kann das leicht verbessert werden, da viele teile des Betriebssystems unabhängig voneinander sind. Es gibt z.B. kein Problem, wenn eine CPU mit dem Scheduler arbeitet, während eine andere einen Systemaufruf bzgl. Des dateisystems behandelt und eine dritte CPU einen Seitenfehler behandelt.

Diese Beobachtung führt zur Aufteilung des Betriebssystems in unabhängige kritische Abschnitte, die untereinander nicht interagieren. Jeder kritische Abschnitt ist durch seinen eigenen Mutex geschützt, so dass ihn immer nur eine CPU zu einem zeitpunkt ausführen kann. Mit diesem Vorgehen erreicht man nun weit mehr Parallelität.

Die meisten modernen Multiprozessorsysteme benutzen diese Vereinbarungen. Der schwierige Teil bei der Entwicklung eines betriebssystems für eine solche Maschine ist nicht die Programmierung, der Code ist nicht gravierend anders als bei einem regulären Betriebssystem. Viel schwieriger ist die Aufteilung des Betriebssystems in die oben beschriebenen unabhängigen kritischen Abschnitte, da man hier sehr viel subtile und indirekte Möglichkeiten von Abhängigkeiten betrachten muss. Zusätzlich muss jede Tabelle, die von mehreren kritischen Abschnitten benutzt wird, eigens durch ihren eigenen Mutex geschützt werden und natürlich muss jeder Code, der solche Tabellen benutzt, auch den richtigen Mutex verwenden.

Des Weiteren muss sehr sorgfältig bei der Vermeidung von Deadlocks verfahren werden. Benötigen zwei kritische Abschnitte die Tabellen **A** und **B** und beanspruchen sie jeweils verschiedene Tabellen zuerst, dann ist eine Deadlock, dessen Ursachen später niemand mehr erkennen wird, früher oder später die Folge.

4.1.2. Multiprozessorsynchronisation

Von Zeit zu Zeit müssen die CPUs in einem Multiprozessor synchronisiert werden. Eine solide Basis der Synchronisation ist wirklich nötig.

Wenn ein Prozess auf einem Einprozessorsystem einen Systemaufruf auslöst, der Zugriff auf kritische Kernel-Tabellen verlangt, so kann der Kern einfach die Interrupts sperren, bovor die Tabelle angerührt wird. Wissend, dass sich kein anderer Prozess dazwischenschieben und die Tabelle benutzen kann, ist der Prozess in der Lage, seine Arbeit zu verrichten.

Auf einem Multiprozessorsystem berührt die Sperrung von Interrupts nur diejenige CPU, auf der sie ausgeführt wurde, während andere CPUs weiterhin in Betrieb sind und auf kritische Tabellen zugreifen können. Notwendigerweise muss ein geeignetes Mutex-Protokoll verwendet und von allen CPUs respektiert werden, um eine korrekte Arbeitsweise des wechselseitigen Ausschlusses gewährleisten zu können.

Man bedenke, was auf einem Multiprozessorsystem passieren kann. In Abbildung 69 sieht man das schlechteste Timing, bei dem das Speicherwort **1000**, hier als Lock verwendet, anfangs den Wert **0** hat. Im Schritte 1 liest CPU1 das Wort und erhält eine **0**. In Schritt 2 kommt CPU2 dazwischen und liest ebenfalls das Wort zu **0** aus, bevor CPU1 die Möglichkeit hat, das Wort zurückzuschreiben und dabei auf **1** zu setzen. In Schritt 3 schreibt schreibt CPU1 nun eine **1** in das Wort. In Schritt 4 schreibt CPU2 ihrerseits auch eine **1** in das Wort. Beide CPUs bekamen von der TSL-(**test and set lock**)-Instruktion eine 0 zurückgeliefert, folglich haben beide nun Zugang zu dem kritischen Abschnitt. Der wechselseitige Ausschluss muss fehlschlagen.

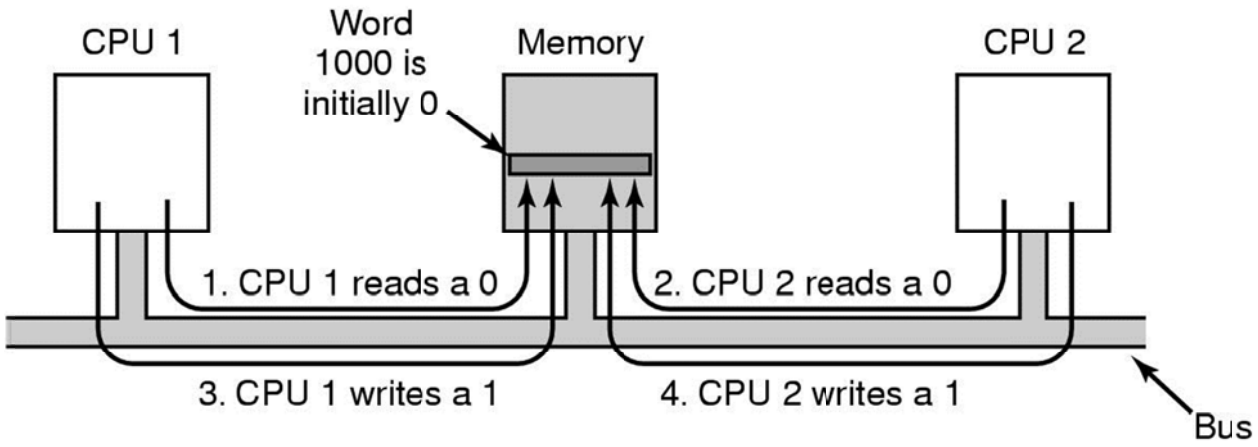


Abbildung 69: Test- und Set Lock Ablauf

Um dieses Problem zu vermeiden, muss die TSL-Instruktion zuerst den Bus sperren, um anderen CPUs den Zugriff auf ihn zu verwehren. Danach werden zwei Speicherzugriffe ausgeführt und schließlich der Bus wieder freigegeben. Typischerweise sperrt man den Bus unter Verwendung des gebräuchlichen Bus-Protokolls. Zuerst erlangt man Zugriff auf den Bus und setzt dann eine spezielle Leitung des Busses bis beide Zyklen vollendet sind. Solange diese spezielle Leitung gesetzt ist, wird keine ander CPU Zugriff auf den Bus gewährt.

Eine noch bessere Idee ist es, jeder CPU, die den Mutex beanspruchen will, ihre eigene Sperrvariable zum Testen zu geben, Abbildung 70 verdeutlicht diese Idee. Um Konflikten aus dem Weg zu gehen, sollte die Varible in einem sonst ungenutzen Cache-Block untergebracht werden.

Folgendermaßen arbeitet der Algorithmus: Eine CPU, die die Sperre nicht in Besitz nehmen kann, reserviert eine Sperr-Variable und hängt sich an das Ende einer Liste von wartenden CPUs. Verlässt der momentane Besitzer nun den kritischen Abschnitt, so gibt er die private Sperre, die die erste CPU in der Liste der wartenden CPUs testet – in ihrem eigenen Cache, frei. Diese CPU kann nun den kritischen Abschnitt betreten. Ist sie fertig, so gibt auch sie die private Sperre des Nachfolgers frei, usw.

Obwohl das Protokoll einigermaßen kompliziert ist – um zu verhindern, dass sich zwei CPUs gleichzeitig an das Ende der Warteschlange hängen, ist es effizient und es kann keine CPU verhungern.

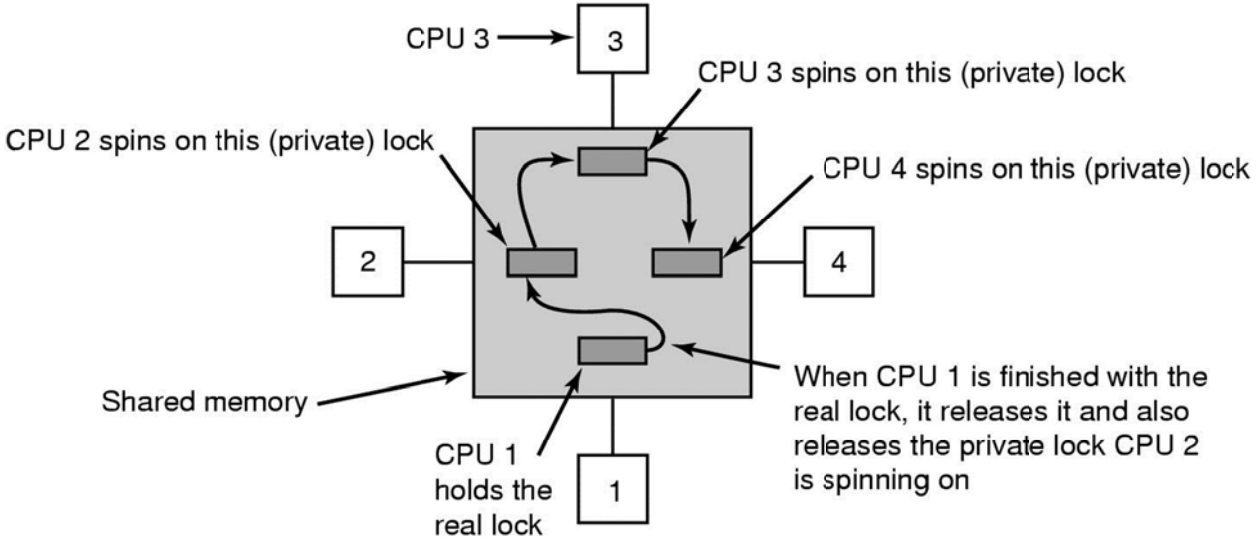


Abbildung 70: Sperren, um Fehler zu vermeiden

Bis jetzt wurde angenommen, dass eine CPU, die einen gesperrten kritischen Abschnitt betreten muss, einfach wartet. Dies ist entweder durch kontinuierliches Polling, durch unterbrochenes Polling oder durch Anhängen an eine Liste wartender CPUs geschehen. In manchen Fällen gibt es in der Tat keine Alternative, als einfach zu warten. Man stelle sich z.B. vor, dass eine CPU nichts zu tun hat und nun Zugriff auf die gemeinsame Liste von rechenwilligen Prozessen benötigt. Ist diese Liste gesperrt, kann sich diese CPU nicht einfach entscheiden und die momentane Arbeit auszusetzen und einen anderen Prozess ausführen, da gerade das durch den Zugriff auf die Liste der rechenwilligen Prozesse bedingt ist. Die CPU muss warten, bis der Zugriff auf die Liste gewährt wird.

Dennoch, in anderen Fällen gibt es Alternativen. Wenn z.B. ein Prozess auf einer CPU Zugriff auf einen Dateisystem-Cache benötigt und dieser im Moment gesperrt ist, dann kann die CPU zu einem anderen Prozess wechseln.

Die Frage, ob aktives Warten oder das Wechseln von Prozessen vorzuziehen ist, war lange Gegenstand intensiver Forschungen. Man beachte, dass diese Frage auf einem Einprozessorsystem nicht relevant ist. Aktives Warten wäre nicht sinnvoll, wenn es keine andere CPU gibt, die die Sperre freigeben könnte.

Wenn ein Prozess versucht, sich die Sperre zu Eigen zu machen, und dabei versagt, ist er dauerhaft blockiert. Der Eigentümer der Sperre hat die Möglichkeit, seine Aufgabe zu vollenden und dann die Sperre wieder freizugeben.

4.1.3. Multiprozessorscheduling

Auf einem Einprozessorsystem ist das Scheduling eindimentional. Die einzige Frage, die – wiederholt – beantwortet werden muss, ist ***„welcher Prozess soll als Nächstes ausgeführt werden?“***

Auf einem Multiprozessor hingegen ist das Scheduling zweidimensional. Der Scheduler muss entscheiden, welcher Prozess als Nächstes ausgeführt werden muss, und vor allem auf welcher CPU. Diese weitere Dimension kompliziert das Scheduling enorm.

Ein zusätzlicher erschwerender Faktor ist, dass auf manchen Systemen alle Prozessoren voneinander unabhängig sind, auf anderen sind sie jedoch gruppiert. Ein Beispiel für die erste Situation ist ein Timesharing-System, bei dem unabhängige Benutzer unabhängige Prozesse starten. Die Prozesse sind unabhängig und können ohne Rücksicht auf die anderen vom Scheduler behandelt werden.

4.1.3.1. Timesharing

Zunächst sei der Fall betrachtet, dass der Scheduler unabhängige Prozesse behandelt. Der einfachste Scheduling-Algorithmus für den Umgang mit unabhängigen Prozessen bedient sich einer systemweiten Datenstruktur für die rechenwilligen Prozesse. Diese Datenstruktur ist möglicherweise eine Liste, wahrscheinlich jedoch eher eine Menge von Listen für Prozesse mit verschiedenen Prioritäten. Als Anhaltspunkt dient Abbildung 71 (a).

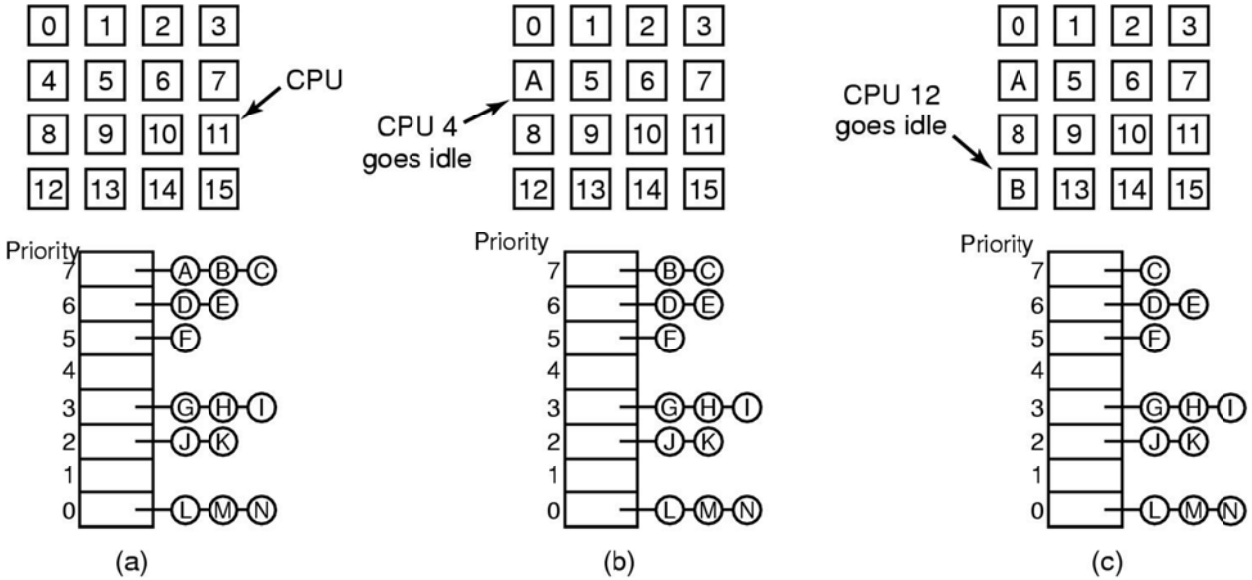


Abbildung 71: Eine einzige Datenstruktur für das Scheduling

Die 16 CPUs sind dort ständig beschäftigt und eine priorisierte Menge von 14 Prozessen warten darauf, ausgeführt zu werden. In der Abbildung ist CPU4 die erste CPU, die ihre momentane Arbeit beendet.

Sie sperrt daraufhin die Scheduling-Warteschlange und wählt den nächsten Prozess höchster Priorität, hier A, aus, siehe Abbildung 71 (b). Als Nächstes wird CPU12 frei und wählt, wie aus Abbildung 71 (c) ersichtlich, Prozess B aus. Solange die Prozesse unabhängig voneinander sind, ist dieses Scheduling sinnvoll.

Der Gebrauch einer einzigen Datenstruktur, die von allen CPUs benutzt wird, unterteilt die Rechenzeit genauso, wie es auf einem Einprozessorsystem der Fall wäre. Sie automatisiert gleichzeitig die Lastverteilung, da es niemals passieren kann, dass eine CPU nicht zu tun hat, während andere überlastet sind.

Zwei Nachteile dieses Ansatzes sind die potentielle Konkurrenz um die Datenstruktur mit wachsender Anzahl der CPUs und der bekannte Overhead beim Kontextwechsel, wenn ein Prozess aufgrund einer Ein-/Ausgabe-Operation blockiert wird.

Ein Kontextwechsel kann auch auftreten, wenn ein Prozess sein Zeitquantum überschreitet. Auf einem belegt, dann Multiprozessor sind einige Eigenschaften, die es auf einem Universalprozessor nicht gibt, von Belang. Es ist auf einem Multiprozessor nicht ungewöhnlich, dass ein Prozess eine Sperre mit aktivem Warten hält und andere CPUs dann auf die Freigabe dieser Sperre warten. Sie verschwenden Zeit, wenn sie darauf warten müssen bis der Prozess erneut vom Scheduler ausgewählt wird und dann die Sperre eventuell freigibt.

Bei Einprozessorsystemen wird die Ausführung eines Prozesses ausgesetzt, während er einen Mutex belegt, dann blockiert ein anderer Prozess, der in diesen kritischen Bereich eintreten will, sofort. Insgesamt wird dabei nur wenig Zeit verschwendet.

Um diese Anomalie zu umgehen, benutzen manche Systeme das sogenannte **intelligente Scheduling**. Ein Prozess, der gegenwärtig eine Sperre hält, auf die aktiv gewartet wird, setzt ein systemweites Flag, um diese Tatsache kund zu tun. Gibt er die Sperre frei, so wird das Flag ebenfalls gelöscht. Der Scheduler hält einen Prozess, der eine solche Sperre belegt nicht einfach an, sondern gewährt ihm mehr Zeit, um diesen kritischen Abschnitt zu verlassen und die Sperre freizugeben.

4.1.3.2. Space-Sharing

Ein anderes Ziel in Bezug auf das Multiprozessor-Scheduling kann verfolgt werden, wenn Prozesse in bestimmter Art und Weise voneinander abhängen. Es kommt oft vor, dass ein einzelner Prozess mehrere Threads erzeugt, die zusammenarbeiten. Für diese Zwecke ist ein Job, bestehend aus mehreren, voneinander abhängigen Threads, und ein Prozess, bestehend aus mehreren Kernel-Threads, im Wesentlichen dasselbe. Das Scheduling mehrerer Threads zur gleichen Zeit über mehrere CPUs hinweg wird **Space-Sharing** genannt.

Angenommen, dass eine Gruppe von in Verbindung stehender Threads auf einmal erzeugt wird. Zum Zeitpunkt, da diese Gruppe entstand, überprüft der Scheduler, ob genauso viele freie CPUs wie Threads vorhanden sind. Ist dem so, bekommt jeder Thread seine eigene CPU und alle Threads starten zur gleichen Zeit. Gibt es nicht genügend CPUs, so startet kein Thread, bis genügend frei CPUs vorhanden sind. Jeder Thread bleibt solange seiner CPU zugeordnet, bis er terminiert. Danach gliedert sie sich wieder in den Kreis freier CPUs ein. Wird einem Thread der Zugriff auf das Ein-/Ausgabe-System verwehrt – blockiert –, so bleibt er seiner CPU, die dann nichts mehr zu tun hat, zugeordnet, und zwar solange, bis er wieder aufgeweckt wird. Der gleiche Algorithmus startet abermals, wenn die nächste Gruppe von Thread erscheint.

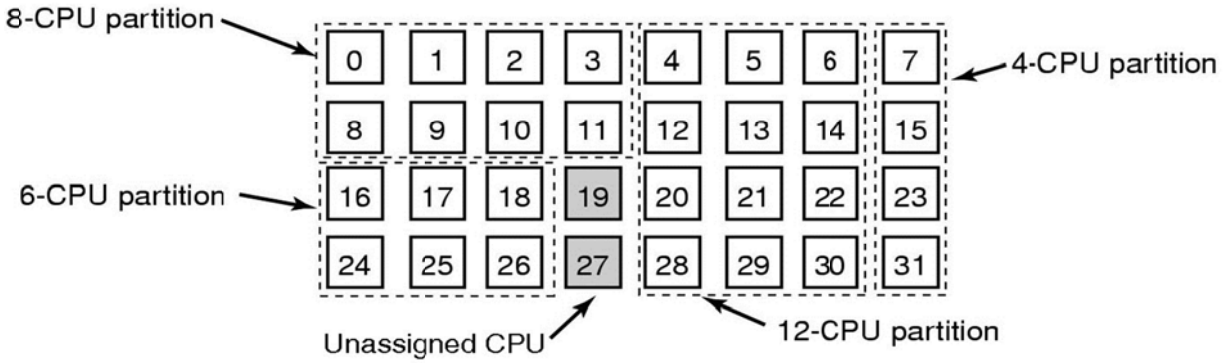


Abbildung 72: 32 CPUs, in 4 Partitionen aufgeteilt und 2 verfügbare CPUs

Zu jedem Zeitpunkt ist die Menge der CPUs statisch in eine Anzahl von Partitionen aufgeteilt und jede Partition führt Threads eines Prozesses aus. In Abbildung 72 sind Partitionen mit 4, 6, 8, 12 CPUs dargestellt. Zwei der CPUs sind im Beispiel nicht zugewiesen. Im Verlaufe der Zeit werden sich, je nach Kommen und Gehen der Prozesse, Anzahl und Größe der Partitionen ändern.

Der klare Vorteil des Space-Sharing ist die Elimination der Multiprogrammierung, welche den Overhead, der beim Kontextwechsel entsteht, beseitigt. Dennoch, ein gleichermaßen offensichtlicher Nachteil ist die verschwendete Zeit, die entsteht, wenn eine CPU blockiert und nichts zu tun hat, bis sie wieder freigegeben wird.

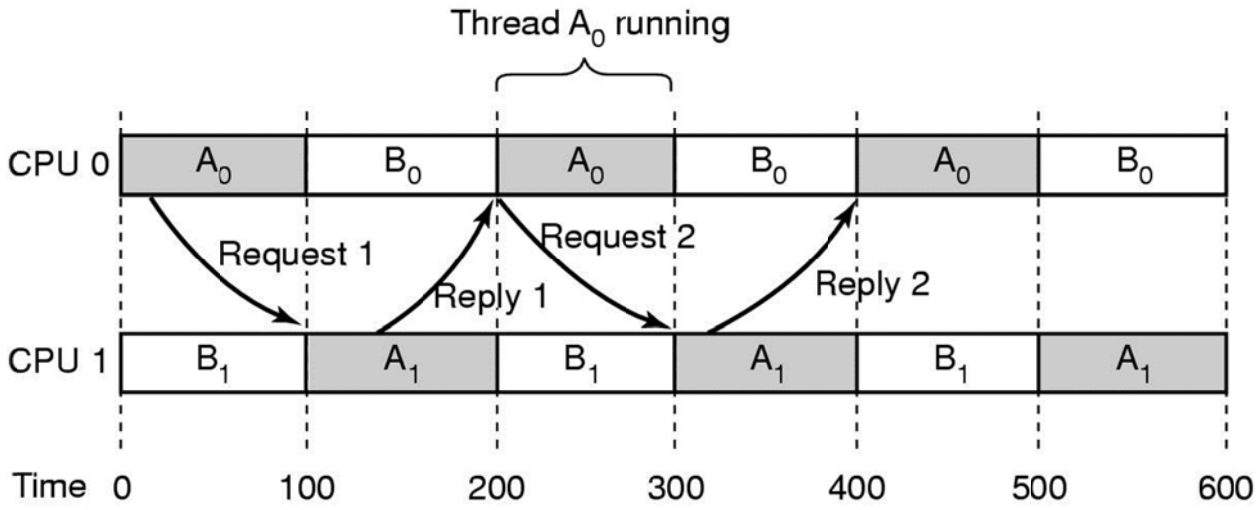


Abbildung 73: Kommunikation zwischen 2 Threads

4.1.3.3. Gang-Scheduling

Um die Art des Problems zu erkennen, welches entsteht, wenn die Threads eines Prozesses unabhängig voneinander zugeteilt werden, wird ein System mit den Threads $\mathbf{A_0}$ und $\mathbf{A_1}$, die zu Prozess **A** gehören, und die Threads $\mathbf{B_0}$ und $\mathbf{B_1}$, die zu Prozess **B** gehören. Die Threads $\mathbf{A_0}$ und $\mathbf{B_0}$ teilen sich die Zeit auf CPU0 und die Threads $\mathbf{A_1}$ und $\mathbf{B_1}$ teilen sich auf CPU1. $\mathbf{A_0}$ und $\mathbf{A_1}$ müssen oft kommunizieren. Das Kommunikationsschema sieht vor, dass $\mathbf{A_0}$ eine Nachricht an $\mathbf{A_1}$ sendet und $\mathbf{A_1}$ dann $\mathbf{A_0}$ eine Antwort zurücksendet. Danach erfolgt eine weitere derartige Sequenz.

Als Annahme, wie in Abbildung 73 zu sehen ist, dass glücklicherweise $\mathbf{A_0}$ und $\mathbf{B_1}$ zuerst starten.

4.2. Verteilte Systeme

Diese Systeme sind Multicomputern ähnlich in der Tatsache, dass jeder Knoten seinen eigenen privaten Speicher besitzt und es keinen gemeinsamen physikalischen Speicher im System gibt. Dennoch, verteilte Systeme sind oft loser verbunden als Multicomputer.

Die Knoten eines Multicomputers haben im Allgemeinen eine CPU, RAM, ein Netzwerkinterface und vielleicht eine Festplatte für das Paging. Im Gegensatz dazu sind die Knoten in einem verteilten System komplette Computer mit Peripheriegeräten. Weiterhin sind die Knoten eines Multicomputers typischerweise in einem einzigen Raum versammelt, damit sie über ein eigenes ausgezeichnetes Hochgeschwindigkeitsnetzwerk kommunizieren können. Die Knoten eines verteilten Systems hingegen können über die ganze Welt verteilt sein. Alle Knoten eines Multicomputers betreiben schließlich das gleiche Betriebssystem, teilen sich ein gemeinsames Dateisystem und unterstehen einer gemeinsamen Verwaltung. Die Knoten in einem verteilten System können jedoch verschiedene Betriebssysteme nutzen, haben ihr eigenes Dateisystem und werden eigens verwaltet. Ein typisches Beispiel für einen Multicomputer sind 512 Knoten. Ein typisches verteiltes System besteht aus Tausenden von Maschinen.

Tabelle 21 vergleicht Multiprozessorsysteme, Multicomputer und verteilte Systeme auf der Grundlage voran genannter Kriterien.

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

Tabelle 21: Vergleich der 3 Arten von Multiprozessorsystemen

Definition 21: Verteiltes System

Ein verteiltes System ist eine Ansammlung unabhängiger, loser verbundenen Computer, die den Benutzer wie ein einzelnes kohärentes System erscheinen.

Diese Definition hat mehrere wichtige Aspekte. Der erste besagt, dass ein verteiltes System aus Komponenten – also Computern – besteht, die autonom sind. Der zweite Aspekt ist, dass die Benutzer glauben, sie hätten es mit einem einzigen System zu tun.

Bis zu einem gewissen Ausmaß ist die lose Verbindung der Computer in einem verteilten System gleichermaßen Stärke wie Schwäche. Es ist eine Stärke, da die Computer für eine Vielzahl von Anwendungen benutzt werden können. Es ist aber auch eine Schwäche, denn die Programmierung dieser Anwendungen ist schwierig, da ein gemeinsames zugrundeliegendes Modell fehlt.

Typische Internetapplikationen beinhalten den Zugriff auf entfernte Computer – unter Verwendung von ssh –, den Zugriff auf entfernte Informationen – unter Benutzung des **World Wide Web** und FTP, des **File Transfer Protocol**, die Person-zu-Person-Kommunikation – **Email** und **ICQ** – und viele andere Anwendungen.

Verteilte Systeme fügen dem zugrunde liegenden Netzwerk ein gemeinsames Paradigma – Modell – hinzu, welches einen allgemeinen Blick auf das ganze System ermöglicht. Die Intention eines verteilten System ist es, die lose verbundene Ansammlung von Computern in ein kohärentes System, basieren auf einem einzelnen Konzept, zu verwenden.

Ein einfaches Beispiel für ein verallgemeinerndes Paradigma in einem leicht andern Kontext findet sich in Unix. Dort sehen alle Ein-/Ausgabe-Geräte wie Dateien aus. Tastaturen, Drucker und serielle Schnittstellen in derselben Art und Weise, auf derselben Grundlage zu betreiben, vereinfacht den Umgang mit ihnen viel mehr, als sie alle konzeptionell getrennt zu verwalten.

Ein Weg, ein gewisses Maß an Uniformität für die unterschiedliche zugrunde liegende Hardware und das Betriebssystem zu erreichen, ist, eine Softwareschicht über dem Betriebssystem einzuführen. Diese Schicht wird **Middleware** genannt und ist in Abbildung 74 dargestellt.

Diese Schicht stellt bestimmte Datenstrukturen und Operationen zur Verfügung, die es Prozessen und Benutzern auf weitverteilten Maschinen erlauben, konsistent zu interagieren.

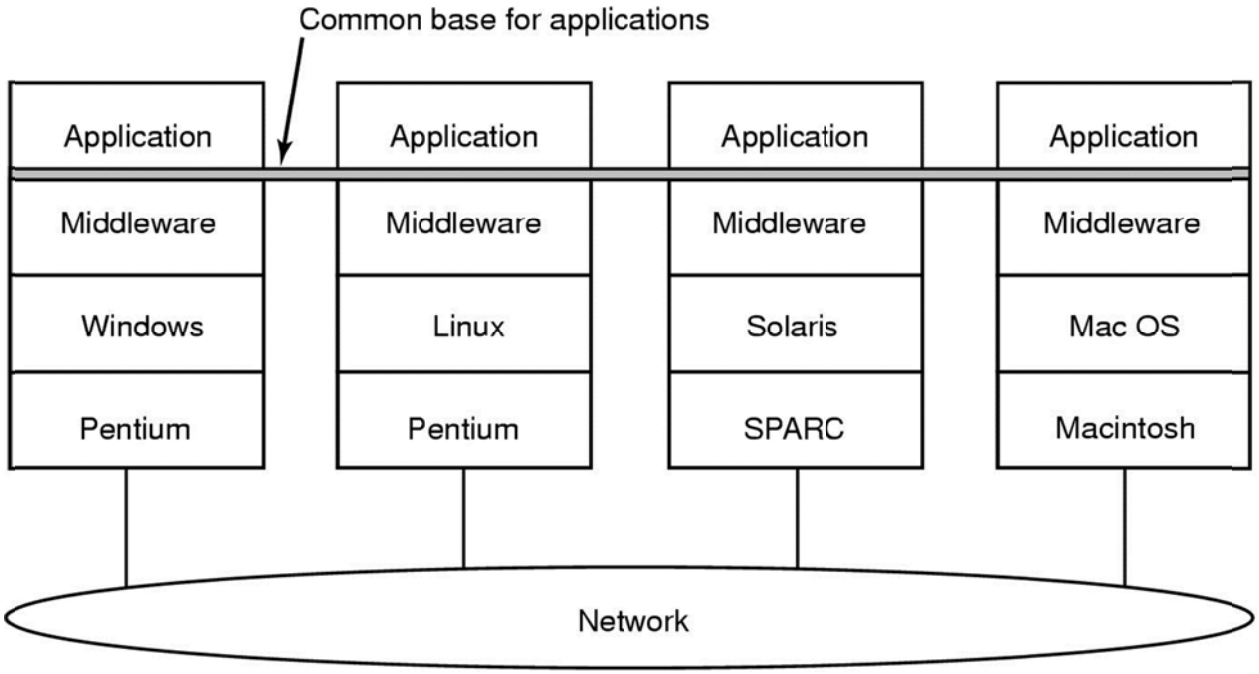


Abbildung 74: Middleware in einem verteilten System

In gewissem Sinne ist die Middleware auf der einen Seite wie ein Betriebssystem für ein verteiltes System anzusehen. Auf der anderen Seite ist sie gerade kein Betriebssystem.

5. Virtualisierung

In der Informatik ist die eindeutige Definition des Begriffs **Virtualisierung** nicht möglich, da der Begriff in vielen unterschiedlichen Anwendungsfällen anders ausgeprägt ist. Es gibt viele Konzepte und Technologien im Bereich der Hardware und Software, die diesen Begriff verwenden. Ein sehr offener Definitionsversuch lautet wie folgt:

Definition 22: Virtualisierung
Virtualisierung bezeichnet Methoden, die es erlauben, Ressourcen eines Computers (insbesondere im Server-Bereich) zusammenzufassen oder aufzuteilen.

Primäres Ziel ist, dem Benutzer eine Abstraktionsschicht zur Verfügung zu stellen, die ihn von der eigentlichen Hardware – Rechenleistung und Speicherplatz – isoliert. Eine logische Schicht wird zwischen Anwender und Ressource eingeführt, um die physischen Gegebenheiten der Hardware zu verstecken.

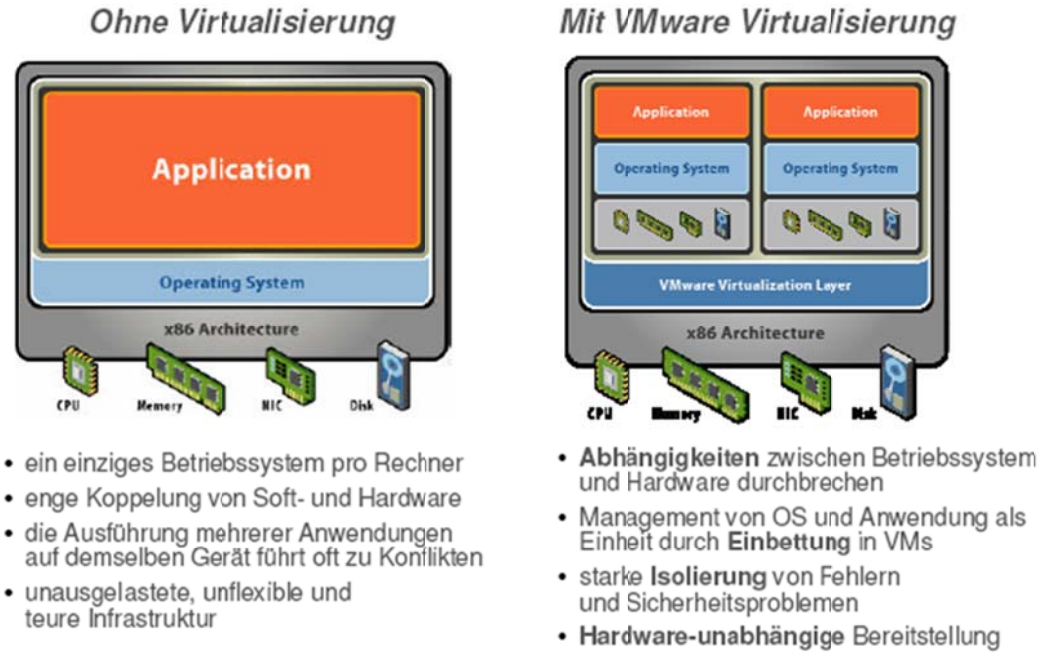


Abbildung 75: Gründe für die Virtualisierung

Dabei wird jedem Anwender (so gut es geht) vorgemacht, dass er (a) der alleinige Nutzer einer Ressource sei, bzw. (b) werden mehrere (heterogene) Hardwareressourcen zu einer homogenen Umgebung zusammengefügt. Die für den Anwender unsichtbare bzw. transparente Verwaltung der Ressource ist dabei in der Regel die Aufgabe des Betriebssystems.

5.1. Vorteile der Virtualisierung

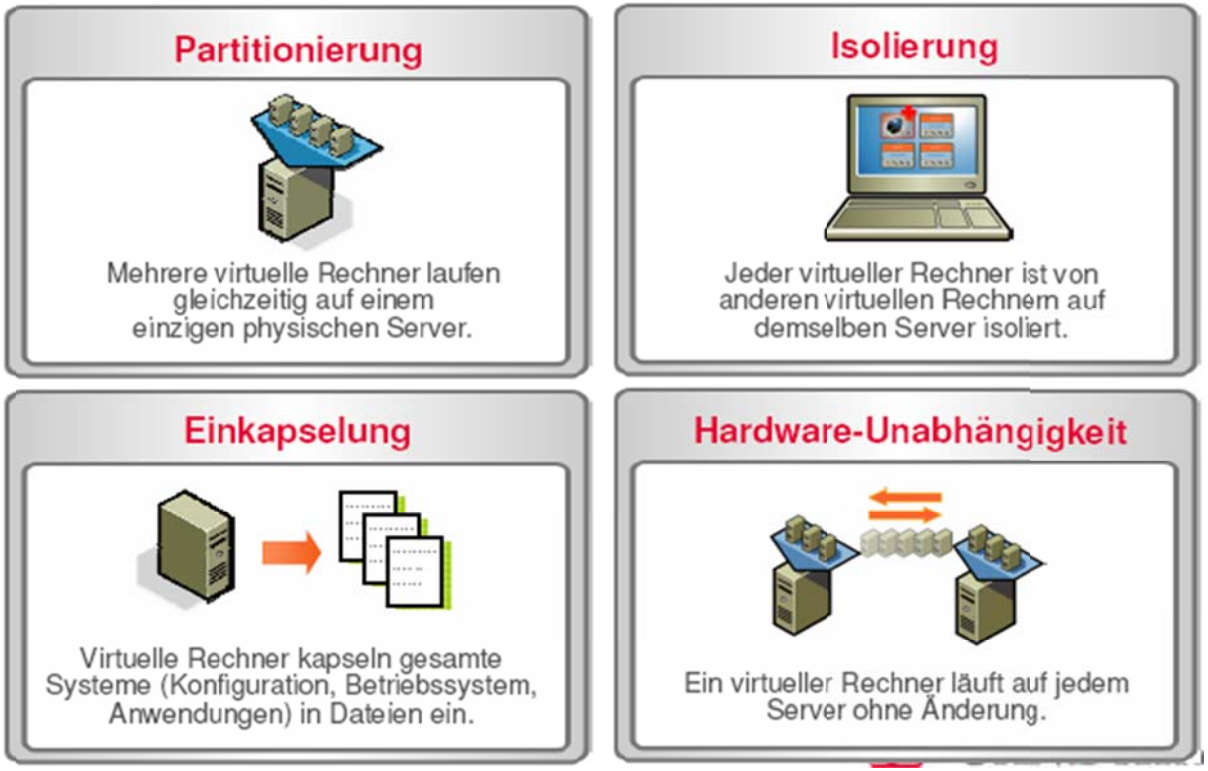
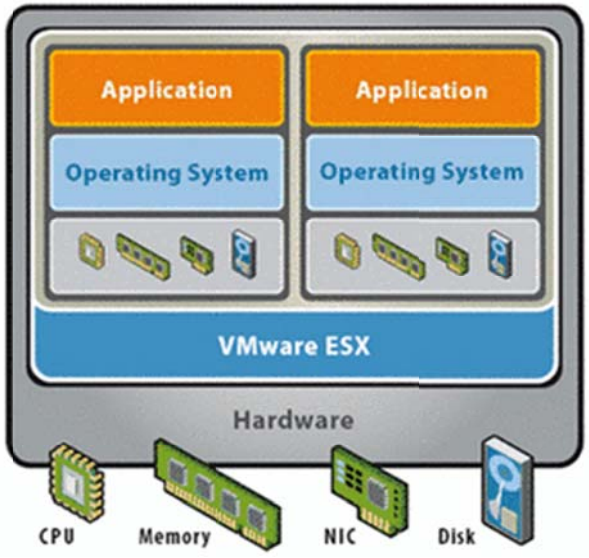


Abbildung 76: Vorteile der Virtualisierung

5.2. Softwarevirtualisierung



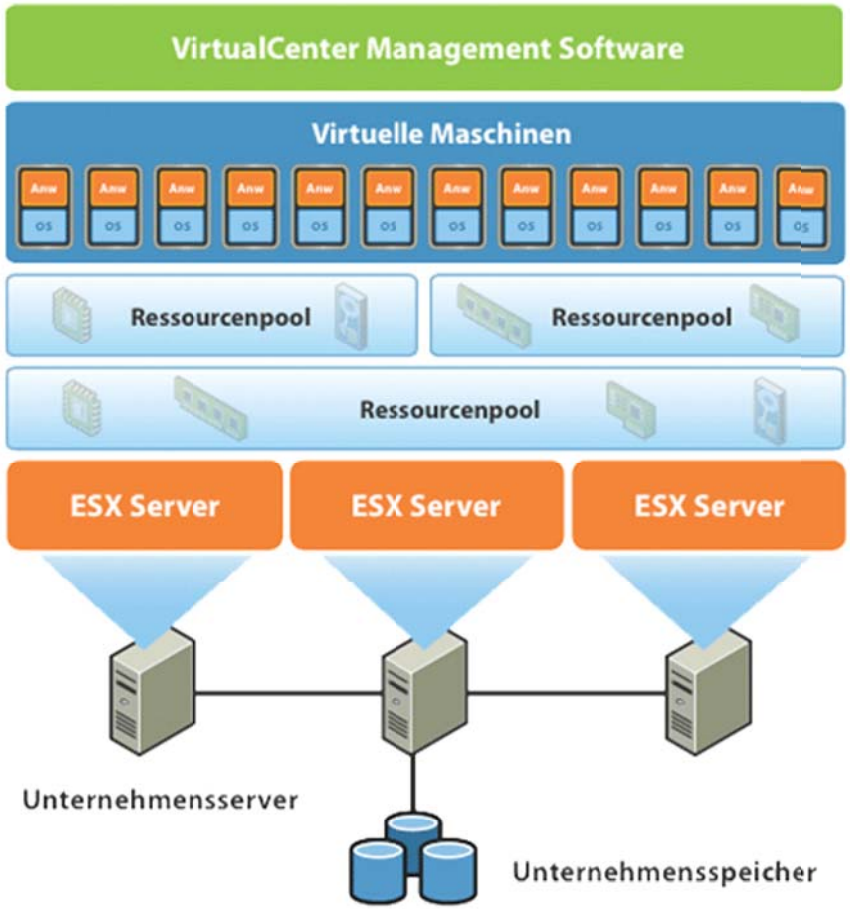
Die Softwarevirtualisierung kann für mehrere Zwecke eingesetzt werden, z. B. um ein Betriebssystem oder nur eine Anwendung zu simulieren.

Bei Virtualisierung auf Betriebssystemebene wird anderen Computerprogrammen eine komplette Laufzeitumgebung virtuell innerhalb eines geschlossenen Containers oder „jails“ zur Verfügung gestellt, es wird kein zusätzliches Betriebssystem gestartet, was zur Folge hat, dass es nicht möglich ist ein anderes OS als das Hostsystem zu betreiben. Die OS-Container stellen eine Teilmenge des Wirtbetriebssystems dar. Vorteil dieses Konzepts liegt in der guten Integration der Container in das Gastbetriebssystem. Der Nachteil dieses Konzepts liegt in den Containern. Aus den Containern heraus können keine Treiber geladen bzw. andere Kernel geladen werden.

Abbildung 77: Softwarevirtualisierung mit VMware

Bei der OS-Virtualisierung läuft immer nur ein Host-Kernel, wobei UML eine gewisse Sonderrolle zukommt, da dort ein spezieller User-Mode-Kernel unter der Kontrolle des Host-Kernels läuft. Beispiele: OpenSolaris, Zoning, BSD jails, Mac-on-Linux, OpenVZ, Virtuozzo, Linux-VServer, User Mode Linux und Xenserver von Citrix.

Bei Virtualisierung mittels eines Virtual Machine Monitors („virtuelle Maschine“) werden die bereitstehenden nativen (= real physisch verfügbaren) Ressourcen intelligent verteilt. Dies kann durch Hardware-Emulation, Hardware-Virtualisierung oder Virtualisierung mittels Hypervisors stattfinden.



Den einzelnen Gastsystemen wird dabei jeweils ein eigener kompletter Rechner mit allen Hardware-Elementen (Prozessor, Laufwerke, Arbeitsspeicher, usw.) vorgespiegelt. Der Vorteil ist, dass an den Betriebssystemen selbst (fast) keine Änderungen erforderlich sind und die Gastsysteme alle ihren eigenen Kernel laufen haben, was eine gewisse Flexibilität im Gegensatz zur Betriebssystemvirtualisierung mit sich bringt.

Wenn weder diese Hardware-Elemente noch die Betriebssysteme der Gastsysteme diese Form der Virtualisierung unterstützen, muss die Virtualisierungssoftware eine Emulationsschicht benutzen, um jedem Gastsystem vorzuspiegeln, es hätte die Hardware für sich allein. Diese Emulation ist oft weniger effizient als direkter Zugriff auf die Hardware, was dann zu einer verringerten Geschwindigkeit führen kann.

Abbildung 78: Verbund mehrerer ESX-Server

Beispiele: VMware Workstation, Microsoft Virtual PC, VirtualBox, Parallels Workstation.

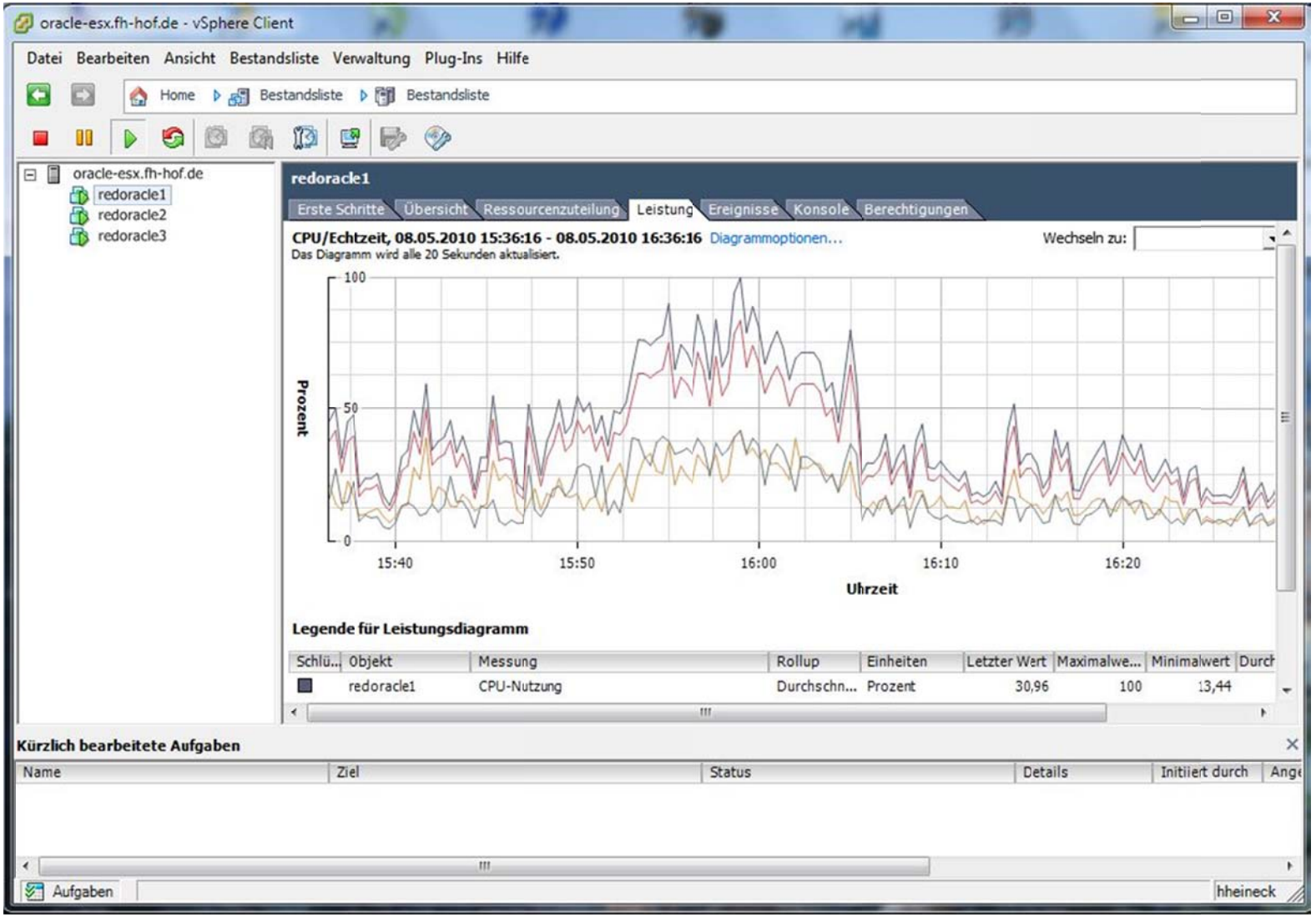


Abbildung 79: RedCluster auf einem ESXi-Server

5.3. Hardwarevirtualisierung

Hardware-Emulation (irreführend auch Full Virtualisation genannt):

Die virtuelle Maschine simuliert die komplette Hardware und ermöglicht einem nichtmodifizierten Betriebssystem, das für eine andere CPU ausgelegt ist, den Betrieb. Beispiele: Bochs (hier anstatt Emulation Simulation), PPC-Version von Microsoft Virtual PC.

Hardware-Virtualisierung (native Virtualisation, full Virtualisation):

Die virtuelle Maschine stellt dem Gastbetriebssystem nur Teilbereiche der physischen Hardware in Form von virtueller Hardware zur Verfügung. Diese reicht jedoch aus, um ein unverändertes Betriebssystem darauf in einer isolierten Umgebung laufen zu lassen. Das Gastsystem muss hierbei für den gleichen CPU-Typ ausgelegt sein. Beispiele: VMware, x86-Version von Microsoft Virtual PC, Xen 3.0 auf Prozessoren mit Hardware-Virtualisierungstechnologien: Intel VT-x oder AMD Pacifica.

Hierfür können entweder das ganze System (Partitioning mit LPAR, Domaining) oder einzelne seiner Komponenten wie z. B. CPU (Intels Vanderpool oder AMDs Pacifica) virtualisiert werden.

5.4. Netzwerkvirtualisierung

Durch Virtual Local Area Networks werden Geräte in einem lokalen Netzwerk in Gruppen aufgeteilt, zwischen denen Verbindungen grundsätzlich unterbunden sind, aber gezielt ermöglicht werden können. Ein Virtual Private Network bildet ein nach außen abgeschirmtes Netzwerk über fremde oder nicht vertrauenswürdige Netze. Software für den gleichzeitigen Betrieb mehrerer virtueller Betriebssysteme auf einem Computer kann ein virtuelles Netzwerk bereitstellen, über das diese kommunizieren. Es können auch mehrere Netze simuliert werden, über die beispielsweise zur Erprobung wiederum ein *Virtual Private Network* aufgebaut wird.

5.5. Virtualisierungslösungen

Kommerzielle Virtualisierungslösungen

- VMware
- Citrix XenServer
- Virtuozzo
- vAdmin

Freie Virtualisierungslösungen

- Xen
- lguest
- Oracle VM
- VirtualBox
- KVM
- OpenVZ
- Linux-VServer
- SandBoxIE
- Microsoft Hyper-V

5.6. Beispiel für die Virtualisierung eines Anwendungsclusters

An dieser Stelle soll ein praktisches Beispiel den Einsatz der Virtualisierung in der Praxis verdeutlichen. Zur Ergänzung der bereits bestehenden Oracle-Server „**sunoracle.fh-hof.de**“ und „**winoracle.fh-hof.de**“ an der Hochschule Hof, wurde angestrebt, einen Ersatz- und Reserve-Server aufzubauen. Gleichzeitig war geplant, verschiedene Technologienen auszuprobieren und zu testen. Neben der Virtualisierung betraf das den Aufbau eines Clusters, bzw. die Virtualisierung eines Clusters, verbunden mit einer darüber liegenden verclusterten Anwendung, einem Datenbankmanagementsystems, wie logischer Weise einen Oracle-Datenbank-Server.

Die daraus abgeleitete Entscheidung ging von folgenden Anforderungen aus:

1. Virtualisierung des aufzubauende Oracle-Server,
2. Einsatz des kostenlosen ESXi-Server der Firma VMware, Inc.,
3. Betriebssystem für den Datenbankserver das RedHat-Linux Oracle Enterprise Linux, OEL5,
4. Aufbau der Datenbank als Real Application Cluster – RAC – der Firma Oracle Corporation und
5. Oracle Server 11gR2, Version 11.2.0.1.0 als Datenbanksoftware.

Zunächst jedoch musste eine geeignete Hardware gefunden werden. Angeschafft wurde von der Firma Hewlett-Packard ein Server HP ProLiant DL380 G5 QC E5420 P400 und damit waren folgende Leistungsmerkmale vorhanden, die damit weitestgehend den Anforderungen genügen konnten:

1. 2 Quad-Core Intel Xeon E5420 Prozessors (2.50 GHz, 1333 FSB),
2. 8 GB PC2-5300 DDR2-667 SDRAM und
3. 1,3 Tbyte Festplattenkapazität.

Als erstes wurde das Betriebssystem für die Virtualisierung installiert. Dabei handelte es sich um den Vmware ESXi Version 4.0.0 für HP Maschinen. Anschließend wurde die Aufteilung für die virtuellen Maschinen geplant. Mit dem Server selbst und 3 virtuellen Maschinen wurden die Resourcen CPU und Hauptspeicher durch vier geteilt. Somit stehen für jede virtuelle Maschine 2 CPU-Core und 2 Gbyte Hauptspeicher zur Verfügung. Außerdem verfügt jede virtuelle Maschine über zwei 50 Gbyte Festplatten.

Die Aufteilung der Festplatten ist typisch für ein RedHat-Linux-Betriebssystem:

Gerätename	Verwendung	Größe	VMware-Name
/dev/sda1	/boot	100 Mbyte	redoraclex/system.vmdk
/dev/sda2	swap	16 GByte	redoraclex/system.vmdk
/dev/sda3	/	34 GByte	redoraclex/system.vmdk
/dev/sdb1	/u01	50 GByte	redoraclex/oracle.vmdk

Tabelle 22: Festplattenaufteilung je virtueller Maschine

Die virtuellen Maschinen wurden mit den üblichen Werkzeugen des ESXi-Servers erzeugt.

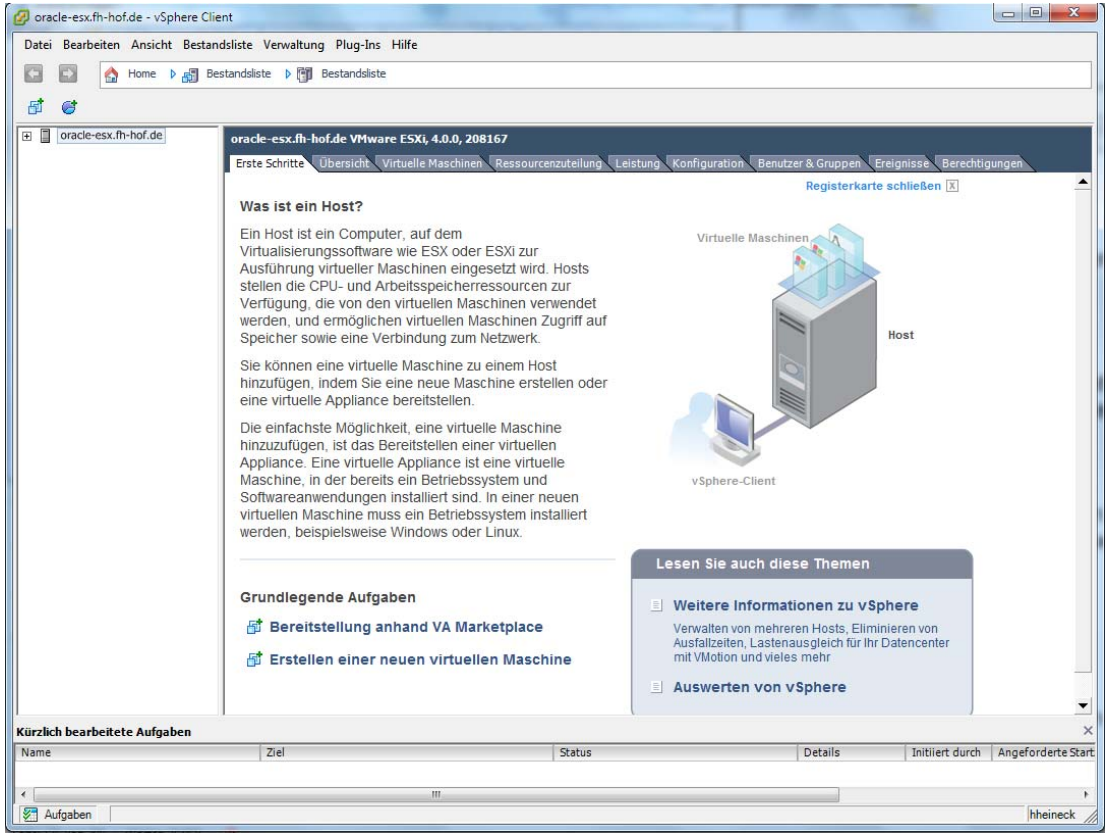


Abbildung 80: Verwaltung des ESXi-Servers über vSphere-Client

5.6.1. Einrichten der gemeinsam genutzten Festplattenlaufwerke

Die spannende Herausforderung für das Cluster war die Schaffung gemeinsam genutzer Plattenlaufwerke, die im Betriebssystem des ESXi-Servers als Dateien erzeugt werden müssen. Dazu wurde direkt auf dem Linux des Servers per Kommando für den Datenspeicher, neben den Ordnern der virtuellen Maschinen **redoracle1**, **redoracle2** und **redoracle3** ein weiterer Ordner **shared** angelegt. Danach mussten die Platten per Kommando erzeugt werden:

```
vmkfstools -c 150G -a lsilogic -d thin /vmfs/volumes/datastore/asm1.vmdk
mit:   –c Größe des Laufwerkes,
       -a Adaptertyp (lsilogic oder buslogic) und
       -d Typ des Laufwerkes (z.B. thin oder thick)
```

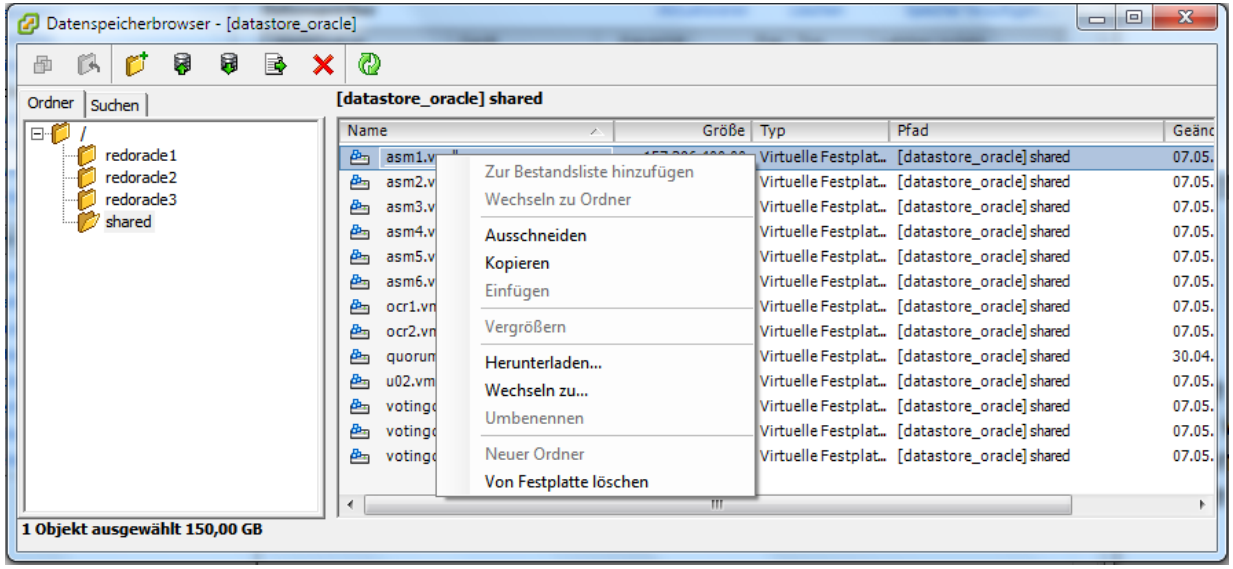


Abbildung 81: Vergrößern der gemeinsam genutzten Laufwerke

Nach Recherchen im Internet wurde der Typ „**thin**“ verwendet, der zwar die Laufwerksgröße als Eigenschaft hatte, aber mit 0 Byte angelegt wurde. Anschließend wurde die Vergrößerung im Datenspeicherbrowser der Monitoring-Software vSphere Client eingesetzt, siehe Abbildung 81, um die angegebene Größe der Festplattenlaufwerke zu erreichen. Ergebnis waren die Einträge im Linux-Betriebssystem, wie sie in Abbildung 82 dargestellt sind.

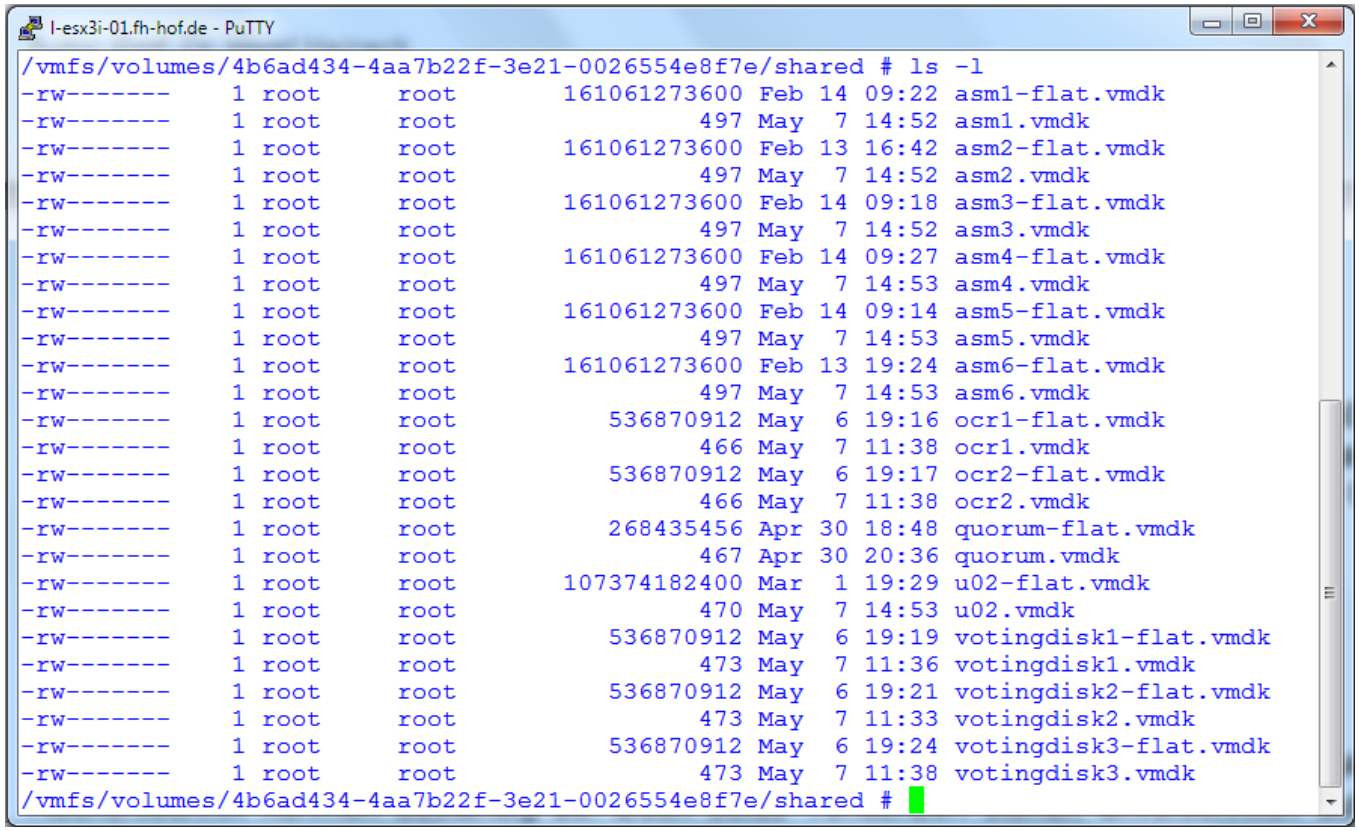
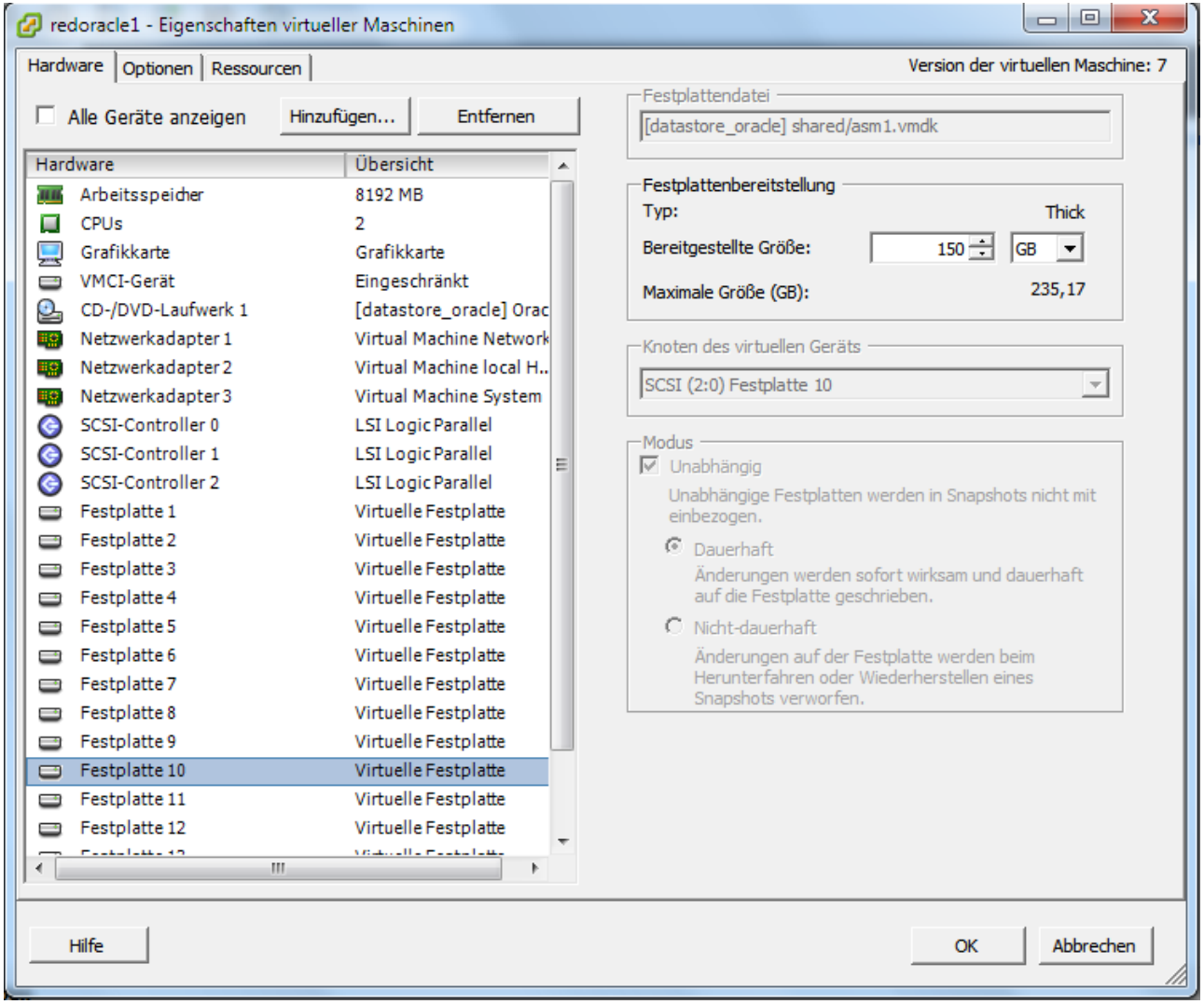


Abbildung 82: ls -l im shared-Ordner



Wie in der Darstellung, Abbildung 82 zu erkennen ist, besteht jedes „Festplattenlaufwerk“ aus den beiden Dateien „**xxx.vmdk**“ und „**xxx-flat.vmdk**“. Wobei die erste als Beschreibung des Festplattenlaufwerkes und die zweite für die Aufnahme der physischen Daten bestimmt ist.

Im weiteren Schritt mussten diese gemeinsam genutzten Laufwerke in die virtuellen Maschinen eingebunden werden. Das erfolgte durch das Bearbeiten der Eigenschaften der virtuellen Maschinen, siehe Abbildung 83.

Entscheidender Punkt dabei war das Einrichten weiterer SCSI-Controller, die den gemeinsamen Zugriff auf die angeschlossenen Laufwerke durch die Eigenschaft „**virtuell“** erlaubten.

Abbildung 83: Einbindung der gemeinsam genutzten Laufwerke

5.6.2. Einrichten der Netzwerkverbindungen

Üblicherweise sind in einem Cluster mehrere Netzwerkadapter pro virtueller Maschine notwendig:

1. Jede Maschine benötigt zunächst einen öffentlichen Zugang auf das Betriebssystem und die Anwendung. Diese Adresse, z.B. redoracle1.fh-hof.de ist auch im **Dynamic Names Service** – DNS – der Hochschule bzw. in der /etc/hosts-Datei im Betriebssystem eingetragen.
2. Wird auf dem virtualisierten Betriebssystem, wenn die Maschine in ein Cluster eingebunden wird, eine schnelle Interconnect-Verbindung notwendig.
3. Zusätzlich muss in der Regel auch die Anwendungssoftware, wenn sie in einem Cluster arbeitet einen weiteren Interconnect zwischen den virtuellen Maschinen besitzen.

Somit sind ein **public** und zwei **private** Netzwerke einzurichten, die wiederum mit den bekannten Werkzeugen zur Konfigurierung der virtuellen Maschinen eingerichtet werden. Üblicherweise werden dazu die Netzwerkadapter **eth0**, **eth1** und **eth2** verwendet.

Dabei bekommt **eth0** den Status **public** und **eth1** und **eth2** den Status **private**. Die beiden privaten Netzwerkadapter sind so zu konfigurieren, dass sie von außen nicht erreichbar sind, d.h., die angeschlossenen Switche dürfen keine Verbindung in die Netzwerkdomäne haben, oder anders ausgedückt, dürfen sie im ESXi-Server keine Verbindung zu einem der vorhandenen Netzwerkadapter haben. Damit wird gewährleistet, dass keine störenden und damit verlangsamenden Einflüsse des äußeren Netzwerkes auf diese beiden Netzwerke haben.

Auch ist es üblich, für die Verbindung innerhalb des Betriebssystems und der Anwendung jeweils getrennt Netzwerke aufzubauen. Im konkreten Fall wurden aus den genannten Gründen zwei Class-A-Netzwerke eingerichtet: **10.253.0.0** für **eth1** und **10.254.0.0** für **eth2**, wie in Abbildung 84 zu sehen ist.

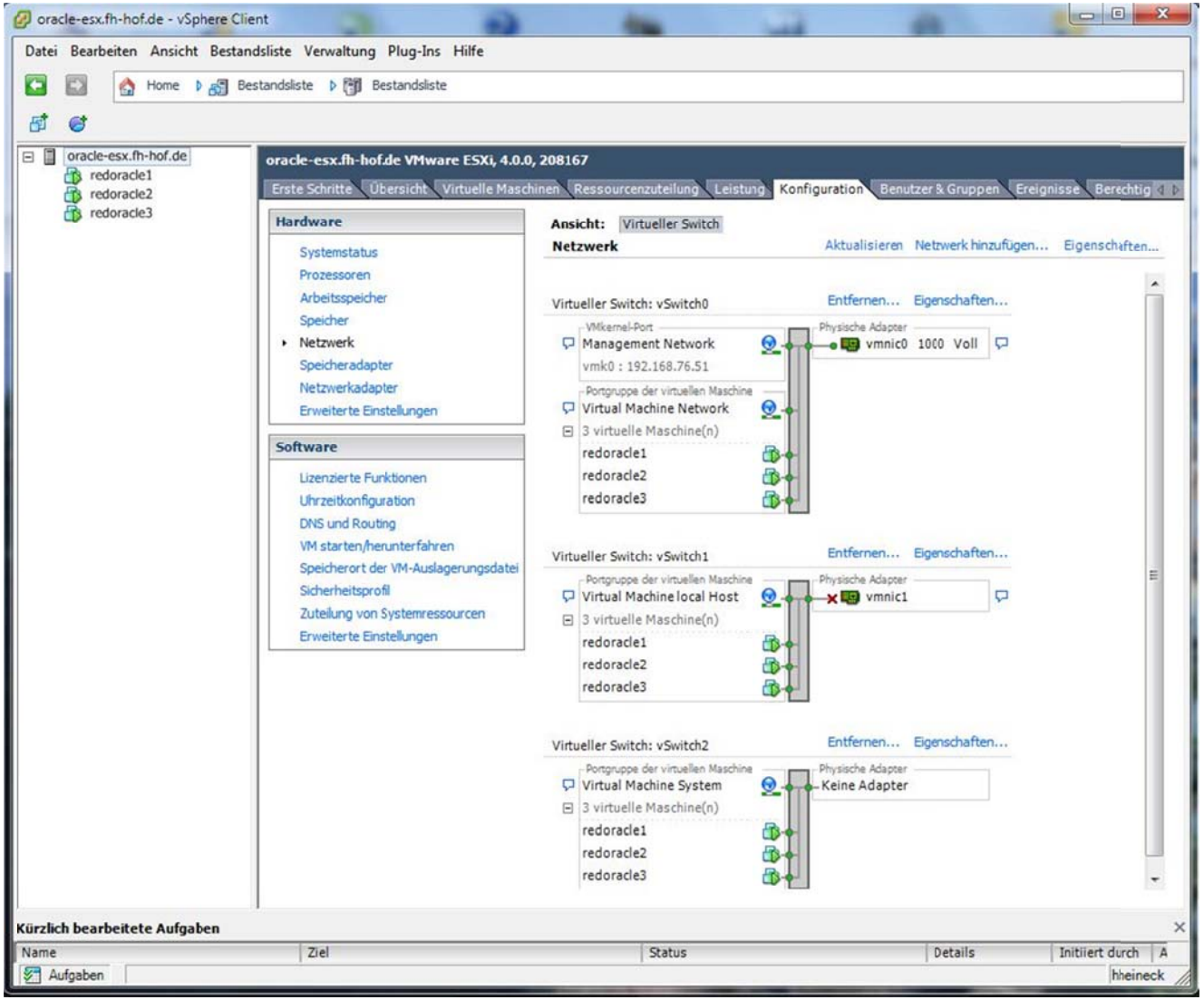


Abbildung 84: Konfiguration des gesamten Netzwerkes

5.6.3. Installation der Betriebssysteme und Anwendungssoftware

Nachdem die Konfigurations- und Einrichtungsarbeiten abgeschlossen sind, kann mit der Installation der virtuellen Maschinen begonnen werden. Diese Tätigkeiten sind nicht so spektakulär und sollen hier nicht weiter betrachtet werden. In Abbildung 85 ist schließlich zum Abschluss das Einrichten eine RAC-Datenbank „**REDKURS**“ mit dem Tool **Database Configuration Assistant** – **dbca** – zu sehen.

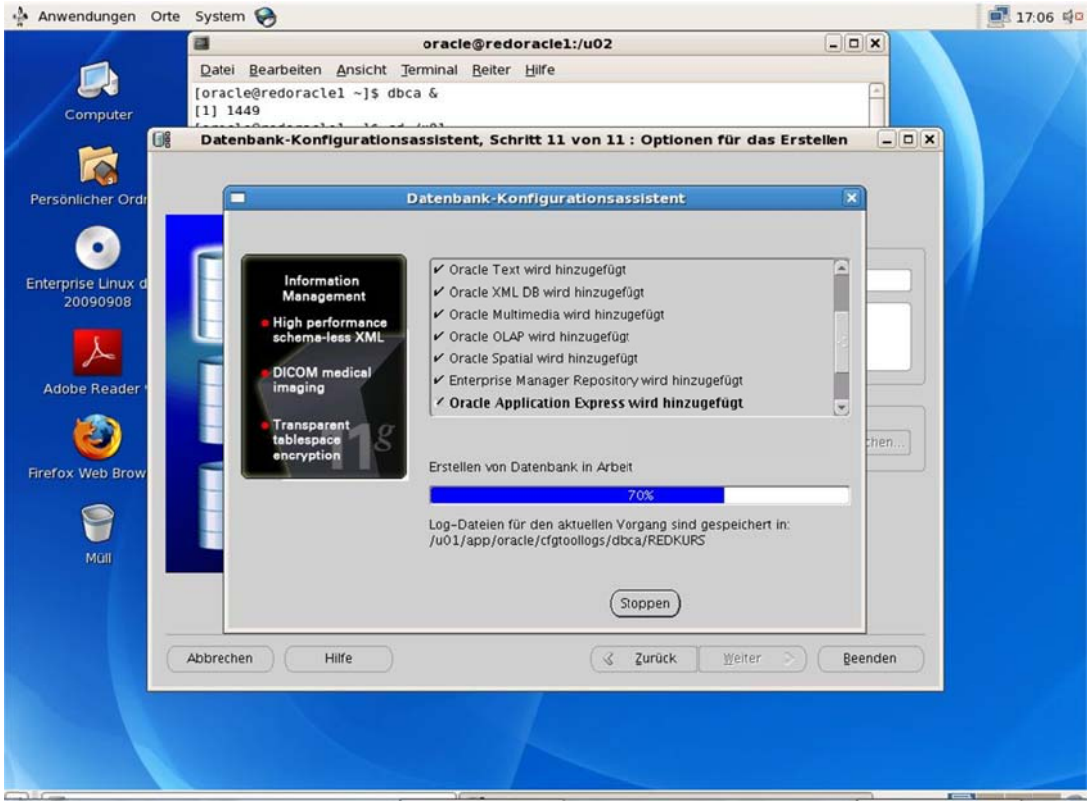


Abbildung 85: Arbeiten zur Installation des Betriebssystems und er Anwendungssoftware.

6. Hochverfügbarkeit

Wir stecken in einem Dilemma: Sowohl die Komplexität der IT wie auch unsere Abhängigkeit von dieser Rechentechnik nehmen beständig zu. Doch beide Tendenzen stehen unversöhnlich auf Kriegsfuß: Komplexität ist der Feind der Verfügbarkeit. Eine reibungslos funktionierende IT aber ist lange schon kein bloßer Wettbewerbsvorteil mehr, sondern eine Existenzfrage. Das ist keine leere Floskel, wie zahlreiche Untersuchungen von Marktforschern und Versicherungen belegen, denen zufolge 40 bis 70 Prozent der Firmen, die von schweren und länger anhaltenden Ausfällen ihrer IT betroffen waren, das darauf folgende Jahr nicht überlebten.

6.1. High Five – „hohe Fünf“

Also Verfügbarkeit um jeden Preis? Das wäre ein unbezahlbarer Ausweg. Denn besonders hohe Ausfallsicherheit ist nicht billig: Jede redundante Hardwarekomponente ist doppelt so teuer wie die einfache Ausführung, Failover-Software oft nicht nur kostspielig, sondern meist auch komplex. Das erfordert weitere Investitionen in Know-how und Training. Zum Glück ist es aber auch gar nicht nötig, dass jede Allerwelts-Applikation die marketingträchtigen Five Nines (99,999 Prozent Verfügbarkeit) anstrebt – denn es geht nicht um unbedingte, sondern um angemessene Verfügbarkeit. Was ist angemessen? Um die Anforderungen dafür zu sortieren, wurden verschiedene Klassifikationen entwickelt.

Tabelle 23 listet gängige Verfügbarkeitsklassen, die sich an der Anzahl der Neunen orientieren. Einen anderen und zugleich anschaulicheren Ansatz entwarf die Harvard Research Group (HRG), siehe Tabelle 24, die ein Schema entwickelt hat, das sich nicht an der Downtime-Quantität, sondern an der Qualität der Auswirkung eines Ausfalls orientiert.

Jährliche Verfügbarkeit	Downtime pro Jahr	Verfügbarkeitsklasse
90 %	36,50 Tage	
95 %	18,25 Tage	
98 %	7,30 Tage	
99 %	3,65 Tage	2 – stabil
99,5 %	1,83 Tage	
99,8 %	17,52 Stunden	
99,9 %	8,76 Stunden	3 – verfügbar
99,95 %	4,38 Stunden	
99,99 %	52,60 Minuten	4 – hochverfügbar
99,999 %	5,26 Minuten	5 – fehlerunempfindlich
99,9999 %	31,50 Sekunden	6 – fehlertolerant
99,99999 %	3,00 Sekunden	7 – fehlerresistent

Tabelle 23: Verfügbarkeitswerte

Anstelle eines blinden Verfügbarkeits-Aktionismus ist also eine durchdachte Strategie gefragt. Dazu kann man sich für einen ersten Überblick daran orientieren, dass sich alle Rechner grob in folgende Klassen mit verschiedenen Ansprüchen an die Ausfallsicherheit einordnen lassen:

6.1.1. Allzwecksysteme (General Purpose Computing)

In dieser Gruppe sind gelegentliche Ausfälle noch tolerabel, wenn sie sich schnell und automatisch beheben lassen. Die Spannweite der Anwendungen ist bei diesen Rechnern sehr groß; sie reicht von Fileservern oder Computern für wissenschaftliche oder technische Berechnungen bis hin zu Datenbanken oder zu ERP-Systemen.

In dieser Gruppe sind gelegentliche Ausfälle noch tolerabel, wenn sie sich schnell und automatisch beheben lassen. Die Spannweite der Anwendungen ist bei diesen Rechnern sehr groß; sie reicht von Fileservern oder Computern für wissenschaftliche oder technische Berechnungen bis hin zu Datenbanken oder zu ERP-Systemen.

6.1.2. Hochverfügbare Systeme (Highly Available Systems)

In dieser Klasse ist die Verfügbarkeit das absolut wichtigste Kriterium. Lokale Störungen, die bereits nach kurzer Zeit wieder repariert sind, lassen sich zwar auch hier noch hinnehmen, aber ein Totalausfall des Systems ist keinesfalls mehr zu akzeptieren. Derartige Rechner stehen beispielsweise in Banken oder auch in Telekommunikationsunternehmen.

6.1.3. Kritische Systeme (Critical Computational Systems)

Hier kann ein Ausfall Menschenleben gefährden oder hohe ökonomische Verluste nach sich ziehen, dementsprechend gilt es, diese Situation unter allen Umständen zu vermeiden. Beispiele finden sich etwa in der Medizin, der Flugsicherung, bei manchen Produktionssteuerungen oder im militärischen Bereich.

6.1.4. Langlebige Systeme (Long-life Systems)

In dieser Gruppe steht die Zuverlässigkeit im Vordergrund. Es handelt sich um Systeme, bei denen eine ungeplante Wartung nicht oder nur mit extremen Kosten möglich ist, weil sie autonom und weit entfernt operieren. Dazu zählen etwa Satelliten.

Den Computer für die Urlaubsplanung der Abteilung wie den Zentralrechner eines Kernkraftwerks zu sichern, wäre offensichtlicher Unsinn. Die meisten Rechner fallen in die Klasse der Allzwecksysteme, für die ein mittlerer Aufwand für die Vorbeugung vor lang andauernden oder häufigen Ausfällen und ein automatisches Recovery

gerechtfertigt ist. Das allein sagt aber noch nicht allzu viel darüber, wie dieses Ziel zu erreichen ist. Dafür ist es nötig, noch weiter in die Details vorzudringen.

Klasse	Auswirkung	Definition
ABC-0	Conventional	Für Geschäftsfunktionen, die unterbrochen werden dürfen und bei denen die Datenintegrität nicht so wichtig ist. Fallen Rechner dieser Klasse unvermittelt aus, unterbrechen sie die Arbeit der Anwender abrupt, und es kann zu Datenverlust kommen.
ABC-1	Highly Reliable	Für Geschäftsfunktionen, die unterbrochen werden dürfen, solange die Datenintegrität gewahrt bleibt. Auch hier ereignen sich plötzliche Ausfälle, es kommt aber nicht zu Datenverlusten.
ABC-2	High Availability	Für Geschäftsfunktionen, die nur eine kurzzeitige oder geplante Unterbrechung erlauben. Die Arbeit der Anwender stoppt, sie können sie aber unmittelbar danach wieder fortsetzen. Daten gehen nicht verloren, unter Umständen sind jedoch Transaktionen zu wiederholen, und die Performance kann vermindert sein.
ABC-3	Fault Resilient	Für Geschäftsfunktionen, die einen unterbrechungsfreien Betrieb erfordern. Die Anwender bleiben durchgängig online, müssen unter Umständen allerdings Transaktionen wiederholen und verminderte Performance in Kauf nehmen.
AEC-4	Fault Tolerant	Für Geschäftsfunktionen, die unterbrechungsfrei ablaufen müssen und bei denen Fehler für die Anwender transparent bleiben. In einem 24x7-Betrieb gibt es hier keine Unterbrechung, außerdem gehen keine Transaktionen verloren, und es ist keine Performanceeinbuße spürbar.
AEC-5	Disaster Tolerant	Für Geschäftsfunktionen, die unterbrechungsfrei ablaufen müssen und bei denen Fehler transparent bleiben. Es gehen weder Transaktionen verloren, noch kommt es zu einer spürbaren Verschlechterung der Performance. Das gilt für diese Klasse selbst im Fall eines Desasters, wie Feuer, Erdbeben, Sturm, Stromausfall oder Vandalismus.

Tabelle 24: HRG-Verfügbarkeitsklassen

6.2. Risiken identifizieren

Der erste Schritt bei der Entwicklung einer effektiven Verfügbarkeitsstrategie besteht darin, zusammenzutragen, was ausfallen kann und also potenziell schützenswert ist. Hier mag man gleich das eine oder andere unkritische System ausklammern können, für das keine Vorsorge nötig ist. Zu den Applikationen, die Unternehmen am häufigsten als unverzichtbar einstufen, zählten nach einer Umfrage der Aberdeen Group im Jahr 2007 in erster Linie E-Mail-Server mit 83 Prozent der Nennungen und Datenbanken mit 63 Prozent. ERP-Daten nannte ein Drittel der Befragten besonders wichtig, gefolgt von den Daten gerade in Arbeit befindlicher Projekte (30 Prozent). Sich allein auf solche Applikationen zu konzentrieren, reicht aber nicht aus. Es gilt, den Gesichtskreis zu erweitern, denn jeder Server hängt von der Infrastruktur ab, in die er eingebettet ist, und der beste Cluster ist vollkommen nutzlos, wenn Strom oder Kühlung ersatzlos ausfallen, im Brandfall keine Löschanlage existiert, Kondenswasser in die Serverschränke tropft, das Netzwerk zusammenbricht oder auch nur das DNS versagt.

Definition 23: Zuverlässigkeit (Reliability)

Die ~ ist eine Eigenschaft, die angibt, wie wahrscheinlich ein System während einer bestimmten Zeitspanne den ihm zugedachten Zweck erfüllt. Zuverlässigkeit ist nicht direkt messbar (im Unterschied etwa zu Größe oder Gewicht), man ermittelt sie entweder empirisch durch Beobachtung der Ausfallhäufigkeit oder analytisch aus der Zuverlässigkeit der einzelnen Komponenten.

Ein in der Technik häufig gebrauchtes Maß für die Zuverlässigkeit ist die Mean Time Between Failures - MTBF, die besagt, wie viel Zeit im Durchschnitt zwischen zwei aufeinanderfolgenden Fehlern vergeht. Ein häufiger Irrtum ist, zu erwarten, dass eine MTBF von 100 000 Stunden bedeutet, dass das Gerät 100 000 Stunden oder 4 166 Tage oder rund 11,5 Jahre durchläuft, ehe es zum ersten Fehler kommt. Das ist leider nicht der Fall. Die MTBF (ein statistischer Wert) kann wesentlich größer als die Lebensdauer sein, mit der sie nichts zu tun hat. So könnte beispielsweise eine Batterie einen Einsatzzeitraum (Useful Life) von vielleicht fünf Stunden, aber eine MTBF von 100 000 Stunden haben. Das bedeutete, dass von 100 000 Batterien eine pro Stunde Betriebsdauer

einen Fehler aufweist. Umgekehrt kann die MTBF kürzer als die Lebenszeit sein, in der sich dann im Mittel mehrere Fehler ereignen. Der Kehrwert der MTBF (1/ MTBF) ist die Fehlerrate. Die Fehlerrate ist in der Regel nur während des eigentlichen Einsatzzeitraums (Useful Life) annähernd konstant. Insgesamt folgt sie dagegen einer Badewannen-Kurve , siehe Abbildung 86. Den hohen Werten in der Anfangsphase versucht man oft durch einen Burn-In-Prozess zu begegnen, der Frühausfälle vor Beginn des produktiven Einsatzes aussondert. Den vermehrten Problemen am Lebensende kommt ein rechtzeitiger Austausch zuvor.

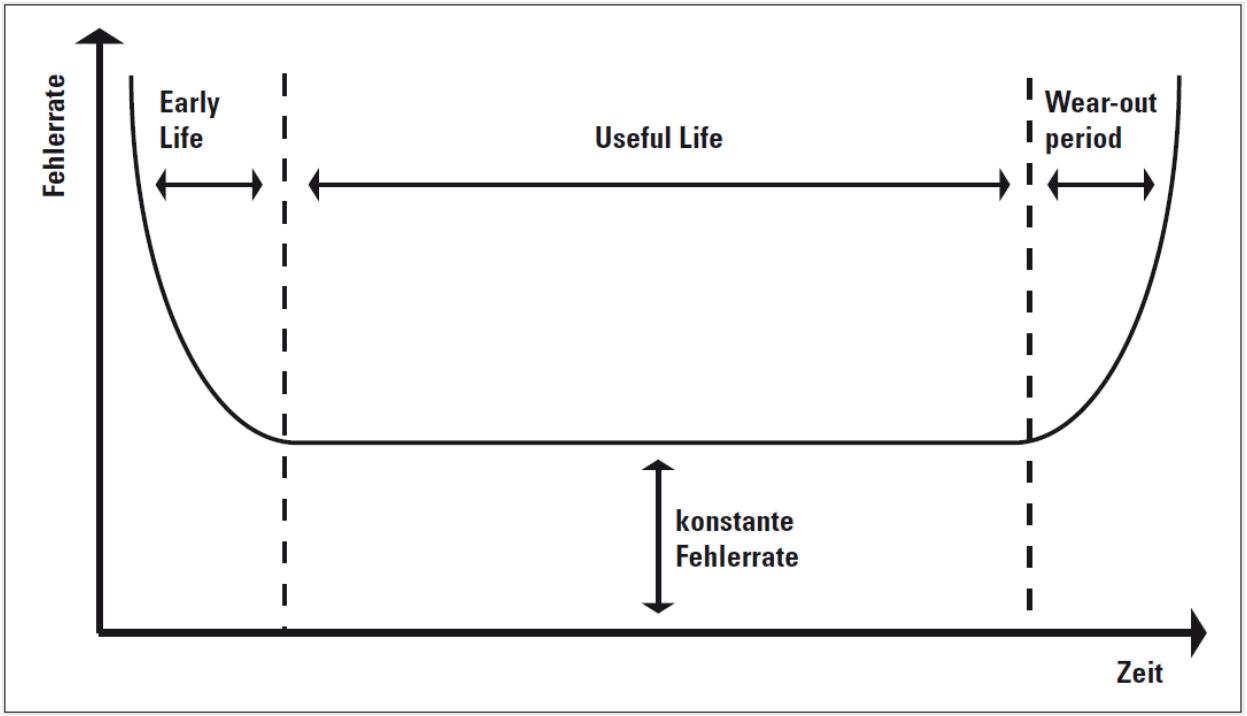


Abbildung 86: Fehlerrate in verschiedenen Lebensphasen

Definition 24: Verfügbarkeit (Availability)

Die ~ ist eine weitere Systemeigenschaft, die die Wahrscheinlichkeit dafür angibt, dass ein System während einer bestimmten Zeitspanne einen bestimmten Service unterbrechungsfrei anbieten kann. Das klingt so ähnlich wie die Definition der Zuverlässigkeit, ist aber nicht dasselbe. In die Verfügbarkeit gehen nämlich zusätzlich die Auswirkungen geplanter und ungeplanter Auszeiten (Downtime) ein.

Eine Messgröße für diese Downtime ist die Mean Time To Repair (MTTR), die angibt, wie lange die Wiederherstellung nach einem Ausfall durchschnittlich dauert. Daraus ergibt sich: Availability = (Gesamtzeit - Downtime) / Gesamtzeit oder V = MTBF / (MTBF + MTTR). Wie man sieht, ist die Verfügbarkeit somit eine Funktion von Zuverlässigkeit und Wartbarkeit (Serviceability). Ein System mit geringer Zuverlässigkeit kann dennoch hoch verfügbar sein, wenn die Zeit zur Wiederinbetriebnahme nach einem Fehler sehr kurz ist. Fällt ein Dienst etwa jeden Monat aus, ist aber nach einer Minute wieder funktionstüchtig, erreicht er damit eine jährliche Verfügbarkeit von 364 Tagen, 23 Stunden und 48 Minuten oder mehr als 99,99 Prozent der Gesamtzeit.

Bei der Ermittlung der Eintrittswahrscheinlichkeit kann man sich unter anderem an der MTBF orientieren, die Kosten der Downtime sind anwendungs- und branchensspezifisch, können aber beträchtliche Höhen erreichen. Der amerikanische Marktforscher Gartner etwa schätzt die durchschnittlichen Kosten für eine Stunde Netzwerkausfall auf 42 000 Dollar allein an Personalkosten für die zur Untätigkeit verdammten Mitarbeiter; noch gar nicht eingerechnet sind die Kosten für die Schadensbehebung selbst, verlorene Transaktionen, unzufriedene Kunden, mögliche Regressforderungen und so weiter. In bestimmten Branchen, etwa bei Banken oder Börsen, fallen sie jedoch noch um ein Vielfaches höher aus.

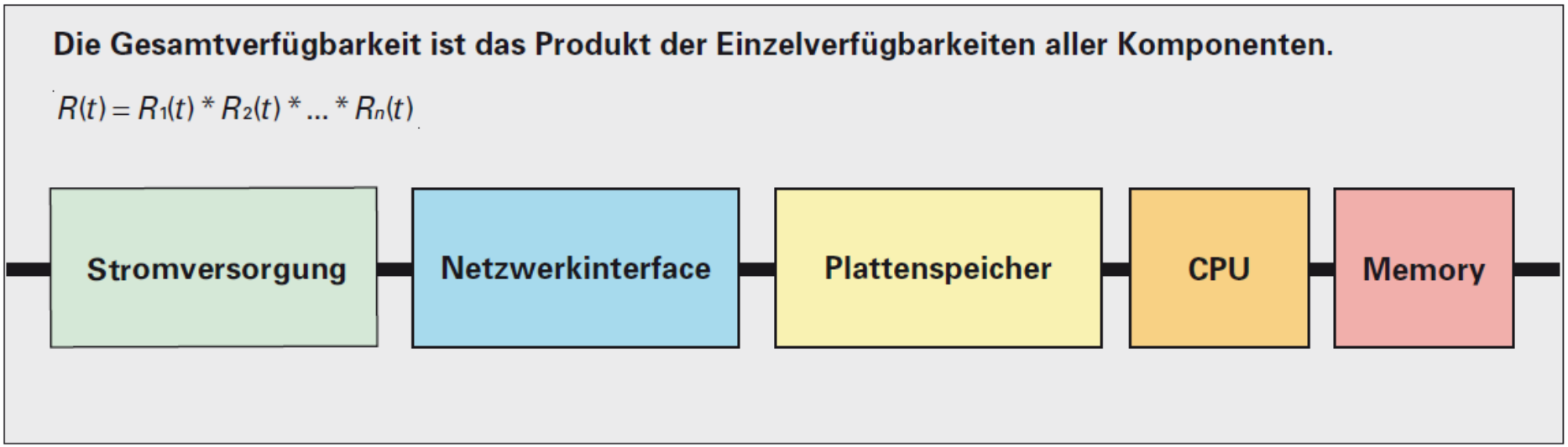


Abbildung 87: In Reihenschaltung von Komponenten multiplizieren sich die Einzelverfügbarkeiten

6.3. Aktionen festlegen

Für jedes Risiko, der nun sortierten Liste gibt es im Wesentlichen vier grundsätzliche Handlungsmöglichkeiten:

6.3.1. Risikovermeidung (Risk Avoidance)

Das scheint auf den ersten Blick die Pauschalantwort auf alle Risiken zu sein: Man tut etwas, damit sie gar nicht erst entstehen. Die pauschale Konsequenz im vorliegenden Fall wäre allerdings: Man arbeitet nicht mit Computern, denn auf andere Weise sind IT Risiken wohl kaum vollständig auszuschließen. Im kleinen Rahmen jedoch ist das Vermeiden sehr wohl ein probates Mittel. Eine RAID-Gruppe etwa hilft, das Risiko eines Datenverlusts nach einem Festplattenausfall zu vermeiden.

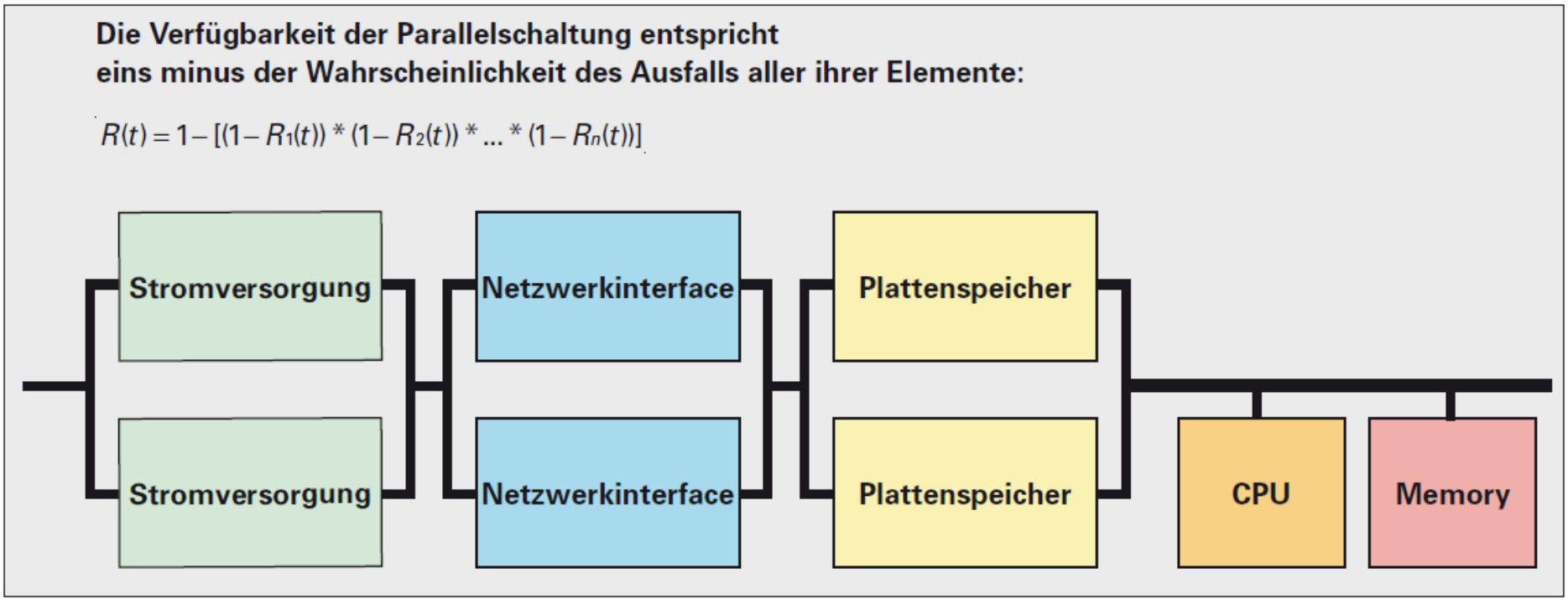


Abbildung 88: In Parallelschaltung von Komponenten multiplizieren sich die Ausfallwahrscheinlichkeiten

6.3.2. Risikominderung (Risk Reduction)

Kann man das Risiko nicht völlig vermeiden, so lässt es sich doch oft mindern. Beispielsweise minimiert ein funktionierendes Backup das Risiko, Daten zu verlieren, weil in der RAID-5-Gruppe eine zweite Platte ausfällt, bevor die erste ersetzt wurde. Eine Sprinkleranlage verringert das Brandrisiko. Allerdings kann hier der Schaden durch Löschwasser größer als der Nutzen sein – einen Ausweg böte eine Halon-Löschanlage, aber die ist teuer. Hier, wie in vielen Fällen, wären dann wieder die oben ermittelten Kosten im Schadensfall zum Vergleich heranzuziehen. Je höher der potenzielle Schaden, je eher rechtfertigt er eine kostspielige Gegenwehr.

6.3.3. Risikoverlagerung (Risk Transfer)

Dieses Herangehen kennt jeder, der je eine Versicherung abgeschlossen hat. Auch für IT-Risiken ist das möglich. Allerdings reduziert diese Strategie weder Gefahr noch Folgen, sondern nur die Kosten für den Versicherten.

6.3.4. Risikoakzeptanz (Risk Retention)

Das bedeutet: Man nimmt das Risiko in Kauf, ohne etwas dagegen zu unternehmen. Das kann in manchen Fällen durchaus sinnvoll sein, beispielsweise dann, wenn die Kosten der anderen Handlungsalternativen über dem Schaden liegen, der schlimmstenfalls eintreten könnte.

Unter diesen Möglichkeiten sind sicher Vermeidung und Minderung das Mittel der Wahl für den Entwickler einer guten HA-Strategie.

6.4. Redundanz

Weiß man nun, mit welchem Aufwand man versuchen kann, bestimmte Gefahren abzuwenden, stellt sich die Frage, wie das zu erreichen wäre. In einer ersten Näherung kann man sich dazu einen Rechner als Reihenschaltung von Komponenten vorstellen, siehe Abbildung 87. Fällt eine Komponente aus, dann ist – wie bei einem Stromkreis dieser Art – das gesamte System funktionsuntüchtig. Mathematisch betrachtet ist hier die Gesamtverfügbarkeit das Produkt der Einzelverfügbarkeiten. Da keine Komponente per se vor einem Ausfall geschützt ist, ihre Verfügbarkeit also stets kleiner als eins ist, nimmt die Gesamtverfügbarkeit mit jeder weiteren Komponente ab. Das ist der Beleg für die Eingangshypothese, wonach Komplexität mit Verfügbarkeit auf Kriegsfuß steht. Legt man nun einige Komponenten doppelt aus, dann entspricht jedes der dadurch entstehenden Komponentenpaare einer Parallelschaltung. Der Ausfall eines Glieds der Parallelschaltung

beeinträchtigt die Funktion des Pärchens nicht, weil nur ein Element der Gruppe tatsächlich nötig ist. Jedes weitere parallelgeschaltete Element taugt als potenzieller Ersatz nach einem Fehler siehe Abbildung 88. In einer Parallelschaltung entspricht die Gesamtausfallzeit $\mathbf{N_{gesamt}}$ dem Produkt der einzelnen Ausfallzeiten, das heißt, sie vermindert sich mit jeder weiteren parallelen Komponente. Im Gegenzug erhöht sich dabei die Verfügbarkeit entsprechend.

Damit ist die schärfste Waffe gefunden, die die Verfügbarkeit gegen die Komplexität ins Feld führen kann: Sie heißt Redundanz. Je redundanter ein System ausgelegt ist, je verfügbarer wird es sein. Allerdings: Redundanz erzeugt immer auch Komplexität, die wiederum eine potentielle Fehlerquelle ist. Deshalb ist auch hier abzuwägen, ob im Einzelfall nicht eine etwas weniger redundante, aber robustere Lösung einer stärker redundanen, dafür aber hochkomplexen vorzuziehen ist.

6.5. SPOF vermeiden

Im hier gezeigten Beispiel sind noch Solokomponenten ohne paralleles Pendant verblieben. Jede bildet einen so genannten Single Point of Failure - SPOF, den es nach Möglichkeit zu vermeiden gilt, weil dessen Ausfall zu einem Totalausfall des Systems führt. Redundanz vertreibt diese Schwachstellen automatisch. Allerdings ist das nicht immer möglich.

Beim Hauptspeicher kann man nur auf die interne Fehlerkorrektur (Error Correction Code - ECC) setzen und durch proaktives Monitoring der Bitfehlerraten versuchen, sich anbahnende Ausfälle frühzeitig zu erkennen, um das betroffene Modul vorher zu tauschen. Die Auswirkung jeder weiteren Parallelisierung auf die Zuverlässigkeit ist ausrechenbar. Auch für komplizierte Strukturen oder Komponenten, wie RAID-Gruppen mit Spare Disks, gibt es entsprechende Formeln. Für einfache Fälle bekommt man das mit dem Taschenrechner hin; für komplexe Anwendungen existiert spezielle Software , die unter anderem auch umfangreiche Datenbanken mit MTBF-Werten aller möglicher Komponenten enthält. Ein Beispiel für so ein Programm ist das Relex Reliability Studio.

So lässt sich bei Bedarf immer ermitteln, wie viel Redundanz sich rechnet und zu wie viel zustzlicher Verfügbarkeit sie führt.

In der Praxis reicht das doppelte Vorhandensein einer Komponente alleine meist noch nicht aus, um die Parallelschaltung zu realisieren. Stattdessen muss man den Reservespielern erst auf die Sprünge helfen. Ein zweites Netzteil schaltet der Rechner in Sekundenbruchteilen noch selbst ein, sollte das erste ausfallen, Netzwerkinterfaces lassen sich durch so genanntes Bonding zu hochverfügbaren Einheiten koppeln , beim Plattenspeicher gelingt das durch RAID-Konfigurationen oder Replikation, der komplette Server wird durch ein Standby-System redundant. Möchte man die Redundanz allerdings auf die Dienste ausdehnen, die der Server anbietetet, dann benötigt man einen Cluster. Mit den Typen, Grundbegriffen und Problemen der Cluster-Technologie beschäftigt sich der folgende Beitrag. Konkrete Anwendungsfälle dagegen demonstrieren viele weitere Artikel dieser Ausgabe. Dabei kommen weder Lösungen für den universellen Einsatz, noch spezielle Techniken zu kurz.

7. Computercluster und Grid Computing

7.1. Grundlagen der Computercluster

Cluster stellen als ein Ansatz, der ein hervorragendes Leistungsverhalten und ein hohes Maß an Verfügbarkeit bieten, eine Alternative zu SMP-Systemen dar, der insbesondere für Serveranwendungen attraktiv ist. Man kann ein Cluster als eine Gruppe von miteinander verbundenen ganzen Computern definieren, die als eine einheitliche Rechenressource so zusammenarbeiten, dass der Eindruck entsteht, es handle sich um einen einzigen Rechner.

Im Abschnitt 5.6. Beispiel für die Virtualisierung eines Anwendungsclusters, wurde bereits an einem Beispiel für die Virtualisierung auch auf den Kontext des Aufbaus eines Clusters eingegangen.

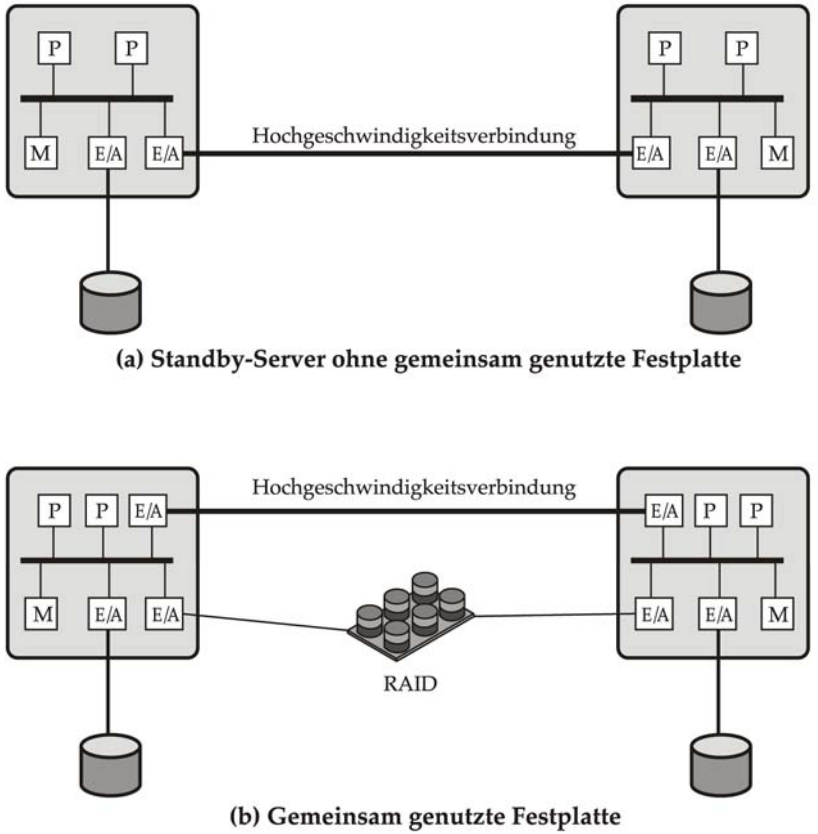


Abbildung 89: Cluster-Konfigurationen

Von besonderem Interesse ist dabei, wie der, in Abbildung 89 (b) dargestellt gemeinsame Zugriff auf die Plattenlaufwerke erfolgen soll. Aus der Literatur sind dazu drei verschiedene Möglichkeiten bekannt:

Verwendung von sogenannten Raw-Devices,
Nutzung eines Clusterfilesystems, wie es in Abbildung 90 exemplarisch dargestellt wird und
Speziell für die Nutzung der Datendateien in einer Oracle-Datenbank das ASM – Automatic Storage Management.

Welche Form der Realisierung in jedem speziellen Fall gewählt wird, hängt im starken Maße von dem Einsatzfall, den Kosten die für die jeweilige Lösung anfallen und im, nicht zu unterschätzenden Know How der jeweiligen Mitarbeiter ab.

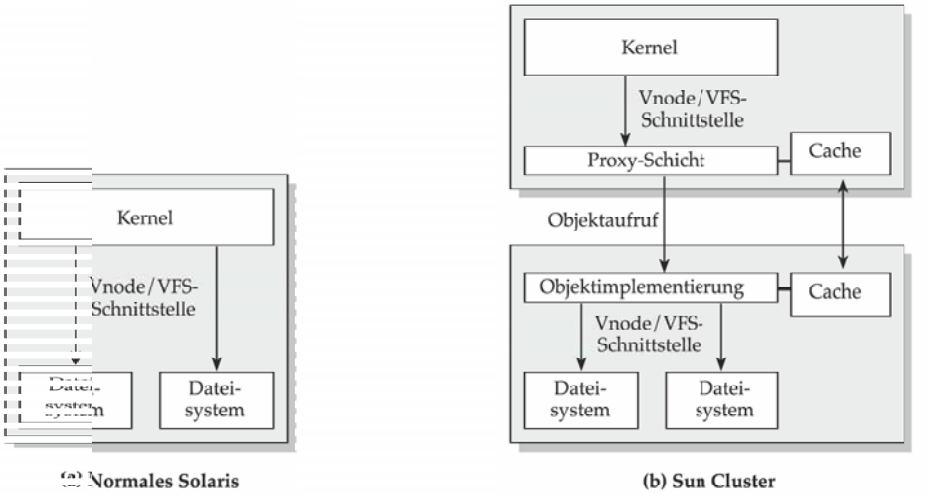


Abbildung 90: Dateisystemerweiterungen bei Sun Cluster

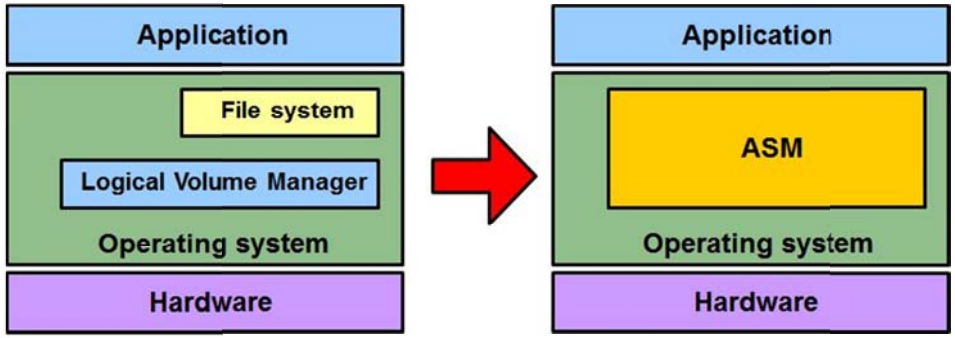


Abbildung 91: Verwendung Raw-Devices und ASM

Was sich hinter Oracle-ASM verbirgt und wie der prinzipielle Aufbau in einem Cluster funktioniert, wird in Abbildung 92 dargestellt.

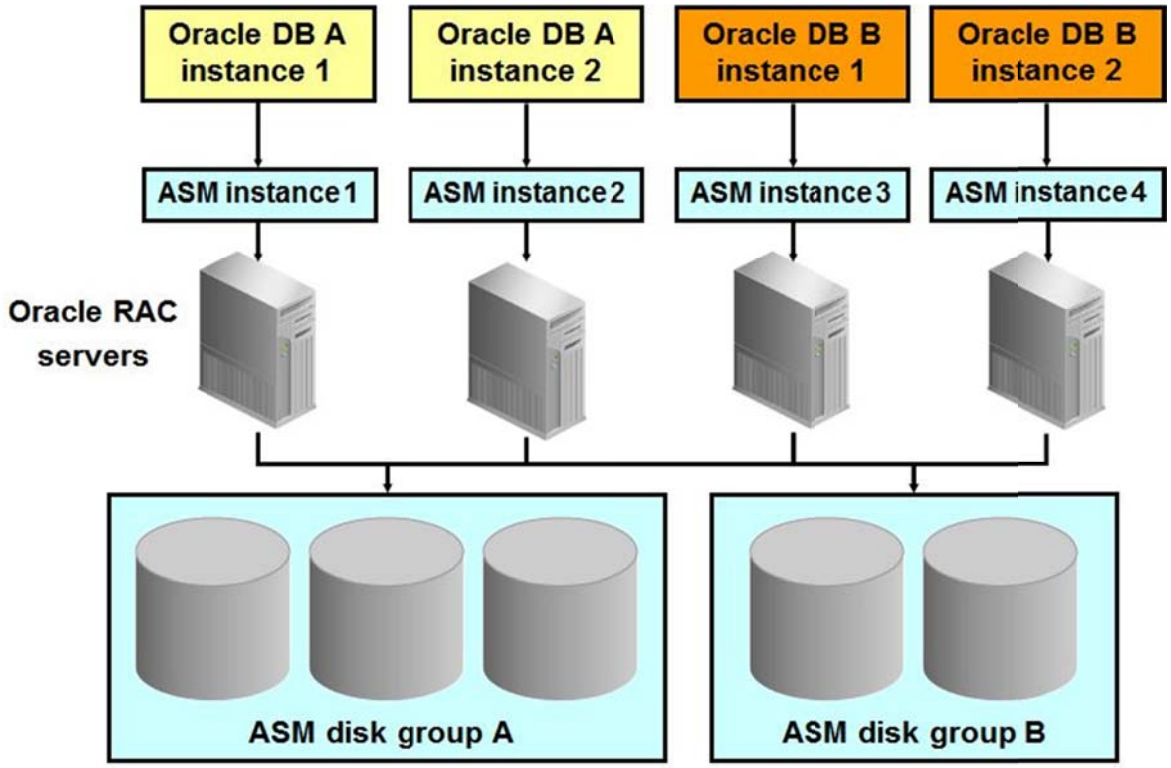


Abbildung 92: Aufbau von Oracle-ASM

Für die Verwendung von clusterfähigen Dateisystemen hat jeder namhafte Hersteller mindestens eine Lösung im Portfolio. In Tabelle 25 sind einige Beispiele zusammengefasst:

Hersteller	Name des Clusterfilesystems
Red Hat	GFS – Global File System
Quantum Corporation	SNFS – StorNext FileSystem
IBM	GPFS – General Parallel File System
Microsoft	CSV – Cluster Shared Volumes
Oracle	OCFS – Oracle Cluster File System
Oracle	ACFS – ASM Cluster File System
Sun Microsystems – jetzt Oracle-Sun	QFS – Quick File System
VERITAS Storage Foundation	VxCFS – Veritas Cluster File System
Vmware	VMFS – Virtual Machine File System

Tabelle 25: verschiedene Cluster File Systeme

Werden alle Aspekte zu Computer-Clustern zusammengefasst, ergibt sich eine allgemeine Konfiguration, wie sie in Abbildung 93 dargestellt ist.

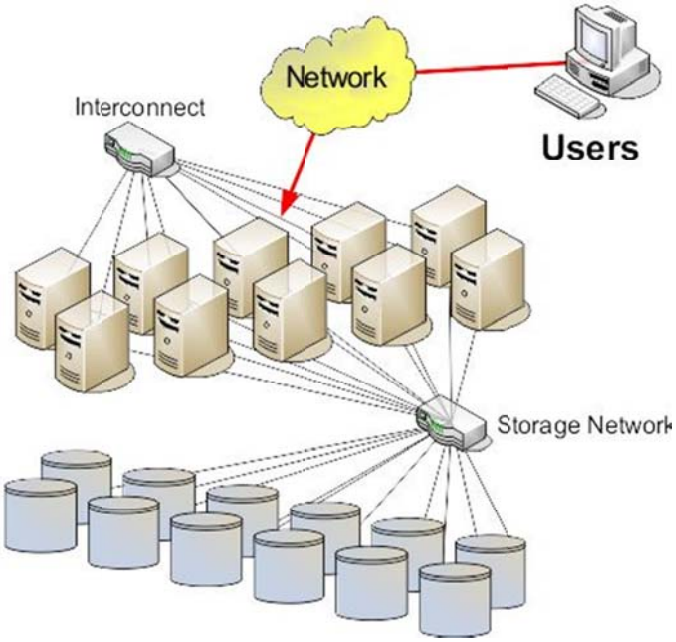


Abbildung 93: Allgemeine Cluster Konfiguration

7.2. Grundlagen von Grid Computing

Was ist Grid Computing → **Utility Computing**, **On-Demand Computing**, **Grid Computing** und viele andere Begriffe beschreiben die nächste Generation von Rechensystemen. Es ist wichtig, den Unterschied zwischen diesen verschiedenen Konzepten zu kennen.

Der Grundgedanke des **Grid Computing** basiert auf der Bereitstellung von Rechenleistung durch **Versorgungsbetriebe** (utility) ähnlich der Versorgung mit Wasser, Gas und Strom über ein öffentliches Netz. Man muss sich nicht darum kümmern, wo die eigenen Daten gespeichert sind oder mit welchen Computerprozessen man arbeiten. Es sollten Daten und Rechenleistung nach Bedarf anfordern können – in dem Umfang und zu dem Zeitpunkt, wenn diese benötigen. Dies entspricht der Arbeitsweise eines Energieversorgers, bei dem ja auch unbekannt ist, wo das Kraftwerk steht und wie die Stromleitungen vernetzt sind. Der Anwender fragt einfach nach Strom und bekommen ihn. Ziel ist es, Rechenleistung zu einem überall verfügbaren Versorgungsgut zu machen.

Es gibt viele verschiedene Möglichkeiten, **Utility Computing** umzusetzen. Die derzeit gängigste Methode besteht darin, einfach neue Lizenzregeln für vorhandene Technologien einzuführen. Einige Serverhersteller bieten beispielsweise die Partitionierung **symmetrischer Multiprozessorserver** (**SMP-Server**) an, so dass bei Bedarf zusätzliche Prozessoren eingeschaltet werden können. Dieses Modell wurde vor vielen Jahren bereits für Großrechner eingesetzt.

Während diese großen **SMP-Server** Rechenkapazität nach Bedarf (**on demand**) bereitstellen können und somit die Realisierung einer Form von Utility Grid Computing ermöglichen, führen solche Systeme nicht unbedingt zu deutlich niedrigeren Kosten.

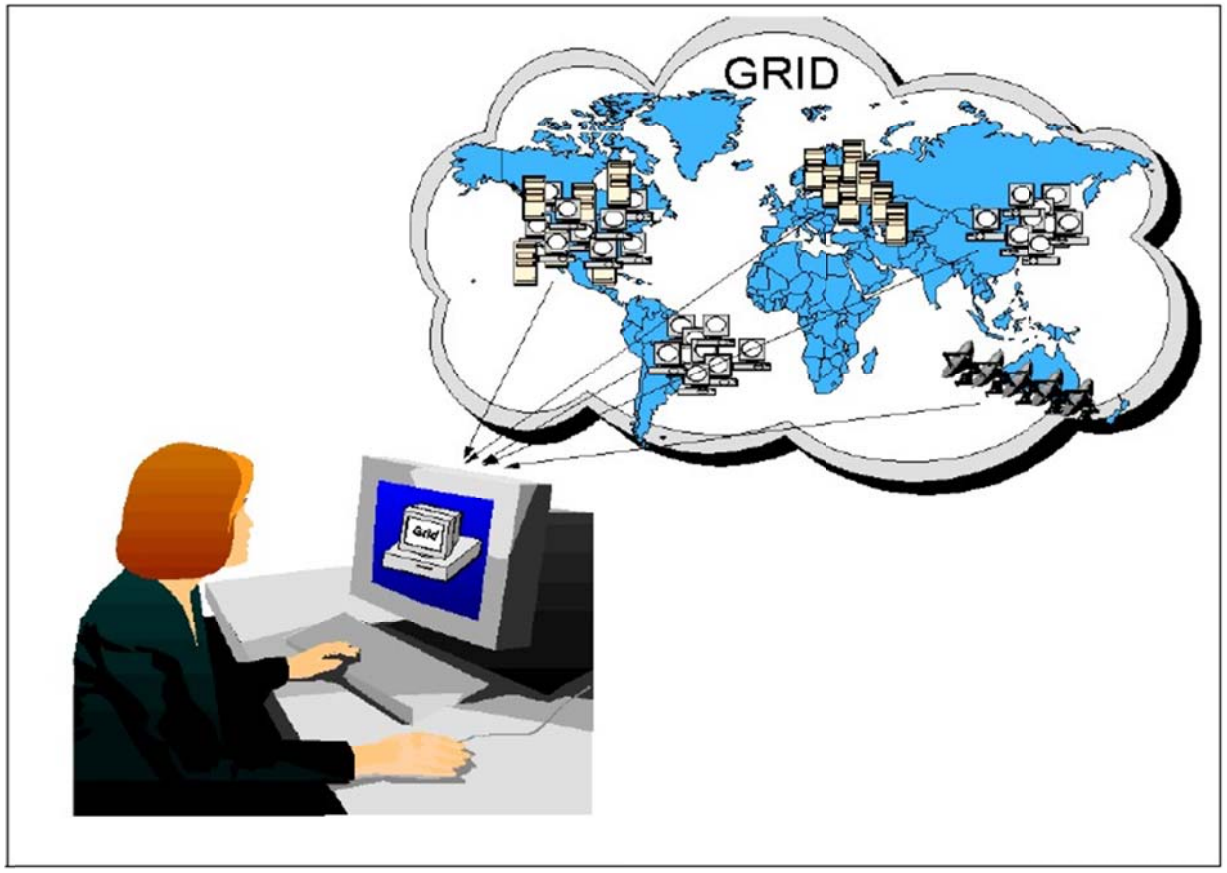


Abbildung 94: Ein Grid ist eine heterogene und geografisch weit verzweigte Organisationsform

Letztendlich bestehen **SMP-Server** immer noch aus exotischen und teuren Technologien und haben Skalierungsprobleme. Somit unterscheidet sich diese Art des **On-Demand Computing** nicht von einem Großrechner mit allen Kosten und Grenzen. Die eigentliche technologische Revolution liegt anderswo.

Grid Computing ist eine grundlegend neue Computerarchitektur zur Erfüllung von Utility-Computing-Bedürfnissen. Grid Computing schaltet eine große Anzahl von Servern und Speichern zu flexibel abrufbaren Ressourcen zusammen, die sich für alle Rechenbedürfnisse eines Unternehmens nutzen lassen.

Gängige Web Services für Identity Management oder Ressourcenbereitstellung bilden die Schnittstelle zwischen den Geschäftsanwendungen und der Grid-Computing-Infrastruktur. Letztere analysiert kontinuierlich die Nachfrage nach Ressourcen und passt das Angebot entsprechend an.

Derzeit wird Grid Computing hauptsächlich für spezielle wissenschaftliche Anwendungen eingesetzt. Aber immer mehr Unternehmen erwägen den Einsatz von Grid-Computing-Anwendungen.

- Grid Computing in der Wissenschaft (**scientific grid computing**) konzentriert sich auf hoch spezialisierte Probleme wie umfangreiche Berechnungen und technische Simulationen.
- Grid Computing in der Wirtschaft (**enterprise grid computing**) bündelt Rechenressourcen zur wirtschaftlichen Nutzung der stetig steigenden Leistungsfähigkeit von zu Clustern zusammengeschalteten Standardservern und -speichern.

Grid Computing hat folgende Vorzüge:

- Flexible Abdeckung wechselnder Geschäftsanforderungen, • hohe Qualität bei geringen Kosten
- Investitionsschutz und schnelle Rendite

Grid Computing gestattet es Unternehmen, ihre Geschäftsarchitektur über Service Level Agreements mit ihrer IT-Architektur zu verbinden. Die meisten Anwendungen nutzen nur die Ressourcen eines einzigen Servers an einem Standort. Ändern sich die Geschäftsprozesse, müssen neue Serverkapazitäten gekauft, muss neue Integrationssoftware geschrieben und müssen weitere Tests durchgeführt werden. So kann es sehr lange dauern, bis die Infrastruktur an die neuen Geschäftsanforderungen angepasst ist.

Grid Computing führt fortschrittliche Lastmanagementfunktionen ein. So können Anwendungen die Ressourcen mehrerer Server gemeinsam nutzen. Die Datenverarbeitungskapazität lässt sich nach Bedarf erhöhen oder reduzieren, und die Ressourcen lassen sich an einem Standort dynamisch bereitstellen. Web Services gestatten die schnelle Integration von Anwendungen zur Schaffung neuer Geschäftsprozesse. Ihre IT-Infrastruktur reagiert daher sofort auf jede Änderung Ihrer geschäftlichen Anforderungen.

Eine Grid-Computing-Architektur ermöglicht den schnellen und einfachen Aufbau einer großen Computerinfrastruktur aus kostengünstigen Standardkomponenten wie Server Blades und Massenspeichern. Durch zentrales Clustering und Lastausgleich bietet Grid Computing höchste Leistung, Skalierbarkeit, Zuverlässigkeit und Sicherheit.

- Grid Computing bietet hohe Leistung und Skalierbarkeit, da alle Rechenressourcen den Anwendungen nach Bedarf flexibel zugewiesen werden können
- Grid Computing ist unempfindlich gegenüber gängigen Ursachen für Systemausfälle wie Hardware-, Software-, Netzwerk- und Bedienungsfehlern
- Durch die Zentralisierung der Ressourcen und deren Behandlung als eine Einheit ermöglicht Grid Computing den Aufbau einer Infrastruktur, die viel sicherer ist als jede andere Architektur

Grid-Lösungen anderer Anbieter erfordern in der Regel maßgeschneiderte Anwendungen und eine vollkommen neue Infrastruktur.

Einige der heutigen Anwendungen sind zu teuer, um sie auf eigenen Systemen laufen zu lassen. Die Nutzung von Grid-Ressourcen für einen kurzen Zeitraum macht ihren Einsatz jedoch wirtschaftlich. Angenommen, ein Unternehmen möchte die Auslieferung eines Produkts an seine Vertriebsgesellschaften und Einzelhändler planen oder eine Behörde die Erbringung von Sozialleistungen für ihre Bürger. Hierbei handelt es sich um unglaublich komplexe Planungen, für die eine gewaltige Rechenleistung für einen begrenzten Zeitraum benötigt wird.

Ein anderes Einsatzgebiet sind Simulationen. Ein kleines Architekturbüro könnte Modelle für wesentlich komplexere Gebäude entwickeln als mit der Technologie, die es sich heute leisten kann. Wetter-, Finanz-, Marketing- und zahlreiche andere Modelle könnten Unternehmen zu deutlich niedrigeren Kosten bereitgestellt werden.

Außerdem gibt es viele umfangreiche Datenquellen, deren vorübergehende Einbindung in Anwendungen sinnvoll wäre – was aber heute technisch nicht machbar ist. Angenommen, eine Hypothekenbank möchte ein Direkt-Mailing an potenzielle Kunden versenden. In diesem Fall ermöglicht Grid Computing die schnelle Integration von Kredit-, Umfrage- und Marketingdatenbanken von Drittanbietern, um eine optimale Mailing-Liste zu erstellen.

- Leistung und Skalierbarkeit mit kostengünstigen Hardware-Clustern.
- Zuverlässigkeit. Enterprise Grids erfordern die kontinuierliche Verfügbarkeit von Daten und Anwendungen.
- Sicherheit und Datenschutz.
- Automatisierte Verwaltung. Angesichts ihrer Größe lassen sich die Infrastrukturen für Enterprise-Grid-Computing mit den heutigen Managementtools nicht mehr kontrollieren. Viele dieser Funktionen werden automatisiert , so dass ein einzelner Administrator mehrere Hundert Server verwalten kann.
- Verteiltes Rechnen. Durch Enterprise Grid Computing können Anwendungen und Daten überall im Netzwerk laufen.

Da immer mehr Unternehmen Cluster aus Standardservern einsetzen, entstehen ganz von alleine IT-Infrastrukturen, die Enterprise Grids ähneln. Durch den Trend, so viele Hardware- und Softwarekomponenten wie möglich zu standardisieren, werden stabile Grid-ähnliche Infrastrukturen zur zuverlässigsten und wirtschaftlichsten Lösung.

Als erste für Enterprise Grid Computing entwickelte Infrastruktursoftware sorgt man für einen sauberen Übergang von der aktuellen Infrastruktur zu einer Zeit- und Kosten sparenden Grid-Computing-Infrastruktur. Möglich wird Grid Computing durch die Kombination von vier Hauptinnovationsbereichen.

- Standardisierung auf kostengünstige, modulare Server und Speicher auf der Basis von Technologien wie den Intel-Itanium-Prozessoren, Blade-Servern und Linux
- Konsolidierung von Server- und Speicher-Clustern, die auf ein oder mehrere Datenzentren verteilt sind
- Automatisierung aller alltäglichen Verwaltungsaufgaben, so dass ein einzelner Administrator gleichzeitig mehrere Hundert Server in Clustern verwalten kann
- Optimierung durch Nutzung gängiger Infrastrukturdienste wie die Ressourcenbereitstellung und Identitätsverwaltung über verteilte Rechentechnologien wie Web Services

Jedes Unternehmen verwendet heute eine Mischung aus verschiedenen Server- und Speichertechnologien. Eine Enterprise-Grid-Computing-Infrastruktur ist dagegen aus einer Vielzahl kleiner standardisierter Server und Speicher aufgebaut. Der erste Schritt zur Senkung der Kosten durch Grid Computing ist die Bestimmung dieser Standardrechen- und -speichereinheiten, die die Grundlage eines neuen Grids bilden. Dies können sehr einfache Einheiten mit begrenzter Lebensdauer sein, da Redundanz eine hohe Verfügbarkeit sicherstellt. Je kostengünstiger diese Grundrechen- und -speichereinheiten sind, umso niedriger sind die Gesamtkosten der Infrastruktur.

Man kann mit einem einzelnen kleinen Server- und Speicher-Cluster anfangen, auf dem eine Anwendung läuft. Bei der Ablösung alter Systeme wird die Kapazität des Grids erhöhen und es werden weitere Anwendungen verlagert. Mit der Zeit vereinfacht sich die Infrastruktur und senkt kontinuierlich die Kosten.
Hier einige der Computertechnologien, die in den letzten fünf Jahren entwickelt wurden und die die Kosten von Server- und Speichersystemen revolutionieren:

- Prozessoren. Die neuen kostengünstigen 64-Bit-Hochleistungsprozessoren Intel Itanium 2, Sun SPARC und IBM PowerPC erreichen oder übertreffen die Leistung der exotischen Prozessoren, die in High-End-SMP-Servern zum Einsatz kommen

- Server. Blade-Server-Technologie senkt die Hardwarekosten und erhöht die Serverdichte (hierdurch reduziert sich der Flächenbedarf von Datenzentren weiter)
- Netzwerkspeicher. Die Plattenspeicherkosten fallen noch schneller als die Prozessorkosten. Bei Netzwerkspeichertechnologien wie Network Attached Storage (NAS) und Storage Area Networks (SANs) nutzen verschiedene Systeme gemeinsam Speicher – so kann ein Unternehmen die Effizienz steigern und gleichzeitig die Kosten reduzieren
- Netzwerkverbindungen. Gigabit-Ethernet- und -InfiniBand-Verbindungstechnologien senken die Kosten der Zusammenschaltung von Servern zu Clustern
- Linux-Betriebssystem. Das Linux-Betriebssystem ist ein stabiles und kostengünstiges Server-Betriebssystem, das sich für das gesamte Grid einsetzen lässt.

Der zweite Schritt zur Umsetzung von Grid Computing ist die Konsolidierung der Infrastruktur in einem oder einigen wenigen Datenzentren. Konsolidierung reduziert die Anzahl der Grids und erhöht die Ressourcen, welche den Anwendungen zur Verfügung stehen. Durch größere Grids steigt außerdem die Zuverlässigkeit, während die Verwaltungskosten sinken.

Der dritte Schritt ist die Automatisierung des Grid, um es mit zunehmender Größe noch effektiv verwalten zu können. Da Enterprise Grids aus mehreren Hundert oder sogar Tausend Servern bestehen können, sind sie einfach zu groß, um noch Server für Server per Hand zu verwalten. Selbst die größten Grid-Datenzentren können von einem oder einigen wenigen Administratoren verwaltet werden.

Der letzte Schritt zur Implementierung eines Grids ist die Anpassung der Anwendungen an die Nutzung der Grid-Services. Durch Nutzung des Grid werden die Anwendungen leistungsfähiger, skalierbarer und zuverlässiger. Und durch Upgrading der Anwendungen zur Nutzung dieser neuen, service-orientierten Architektur können die Anwendungsintegration und Verwaltung von Geschäftsprozessen weiter verbessern.

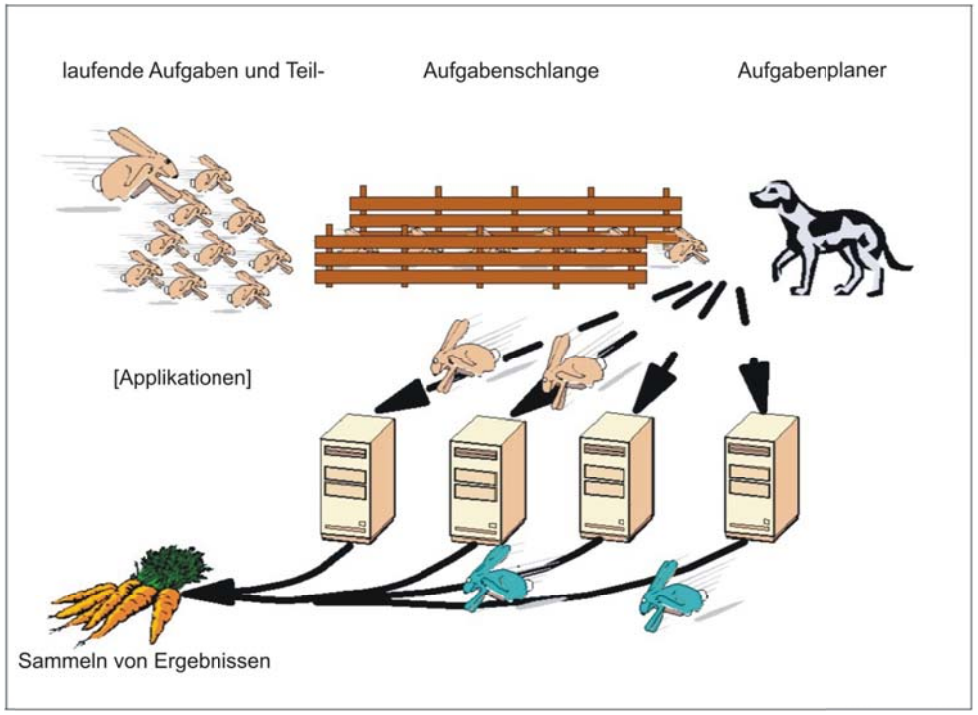


Abbildung 95: Aufgaben und Applikationen

Tabellenverzeichnis

Abbildungsverzeichnis

Definitionsverzeichnis

Index