hochschule hof

Fakultät Informatik und Ingenieurwissenschaften

Vorlesung Informatik im 6. Semester

„Echtzeitsysteme“

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

Raum: **FB 131**
Tel.: **09281/409-444**
Fax: **09281/409-400**
Email: **Horst.Heineck@fh-hof.de**

Sprechstunde **Sommersemester 2009**
Mittwoch: **12^{00} Uhr – 13^{00} Uhr**

Literaturverzeichnis

/Heineck-84/ Heineck, H.:
Ein Beitrag zum Entwurf von dialogfähigen Mikrorechnerprogrammentwicklungssystemen, Dissertation, TH Ilmenau, Sektion Technische und biomedizinische Kybernetik, 1984

/Krapp-90/ Krapp, M.:
Digitale Automaten, 2. Auflage, Verlag Technik, Berlin 1990

/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

/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

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

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

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

/Stallings-02/ Stallings, W.:
Betriebssysteme, Prinzipien und Umsetzung, Pearson Studium, 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

/Wuttke-02/ Wuttke, H-D, Henke, K.:
Schaltsysteme, Eine automatenorientierte Einführung, Pearson Studium, 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

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

/Ehses-05/ Ehses, E., Köhler, L., Riemer, P., Stenzel, H., Victor, F.:
Betriebssysteme, Verlag Pearson Studium, 2005

/Hammer-05/ Hammerschall, U.:
Verteilte Systeme und Anwendungen, Verlag Pearson Studium, 2005

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

/Wörn-05/ Wörn H., Brinkschulte U.:
Echtzeitsysteme, Springer-Verlag Berlin Heidelberg 2005

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

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. Einführung

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.

1.1. Grundbegriffe

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

Definition 1: 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.

Definition 2: 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.

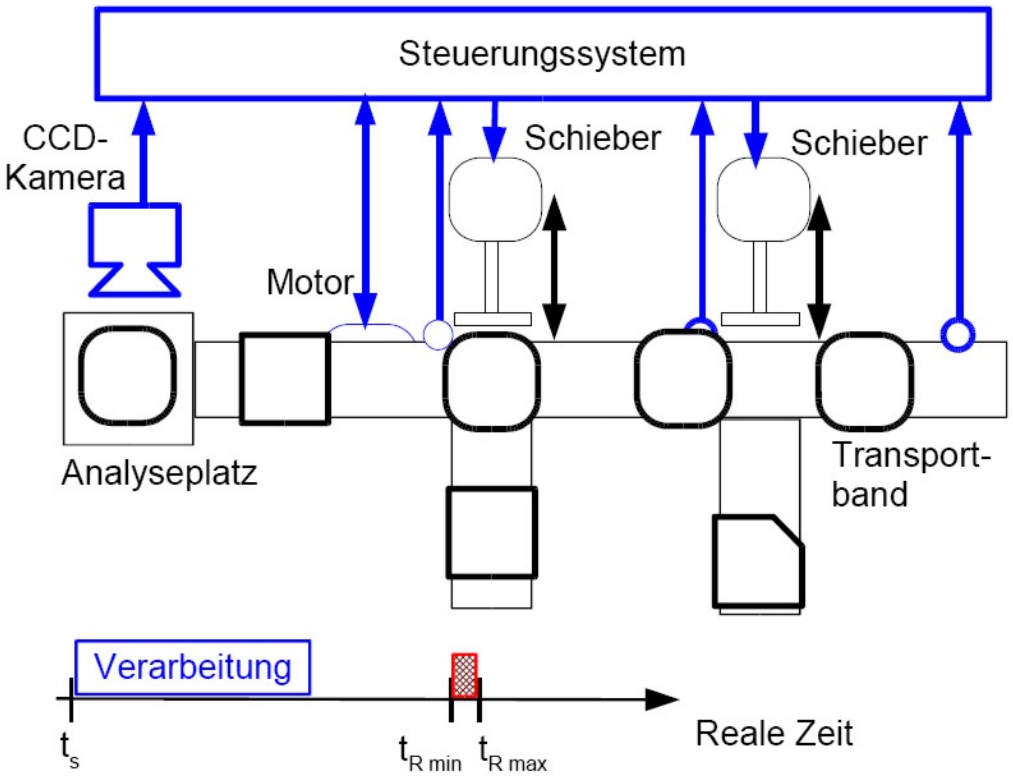


Abbildung 1: Beispiel für ein Echtzeitsystem

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

In Abbildung 2 ist der Zusammenhang zwischen den Regelkreis, siehe Abbildung 3, mit den Signalen x_i und y_i im Vergleich zum Zeitverhalten dargestellt.

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.

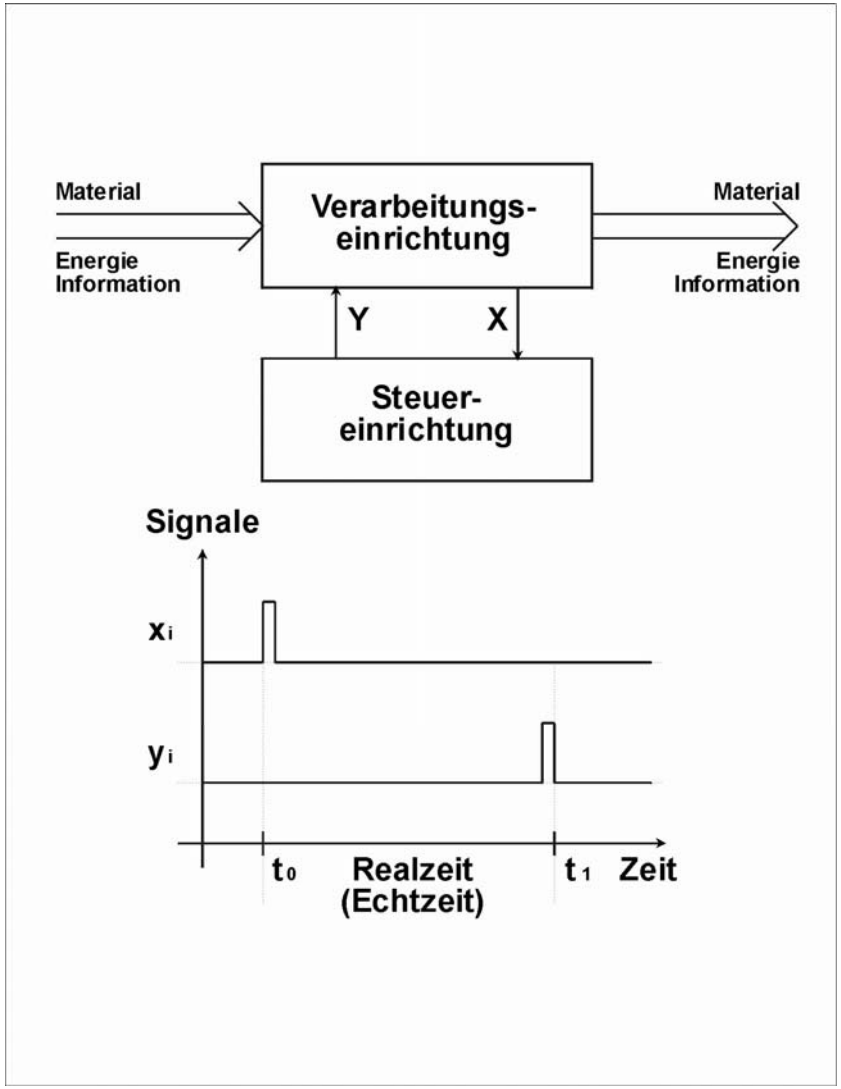


Abbildung 2: Echtzeit

An ein Echtzeitsystem stellt man im Allgemeinen Anforderungen nach Rechtzeitigkeit, Gleichzeitigkeit und zeitgerechter Reaktion auf spontane Ereignisse, vergleiche Tabelle 5.

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.

Definition 5: Gleichzeitigkeit

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

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

Definition 6: 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 1: 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).

1.2. 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.

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

Bei dieser Darstellung, siehe Abbildung 3, wird noch nicht zwischen Steuern und Regeln unterschieden.

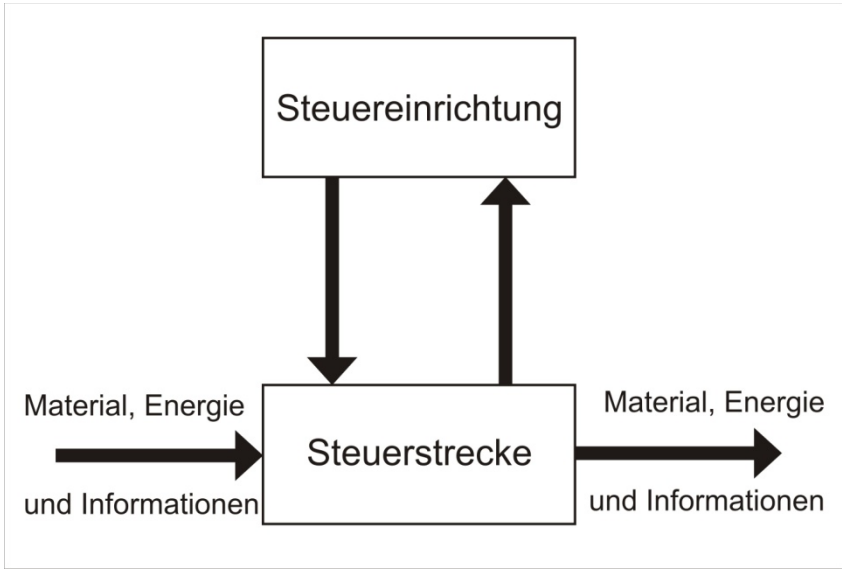


Abbildung 3: Regelkreis

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 4 zeigt das Grundprinzip der Automatisierung.

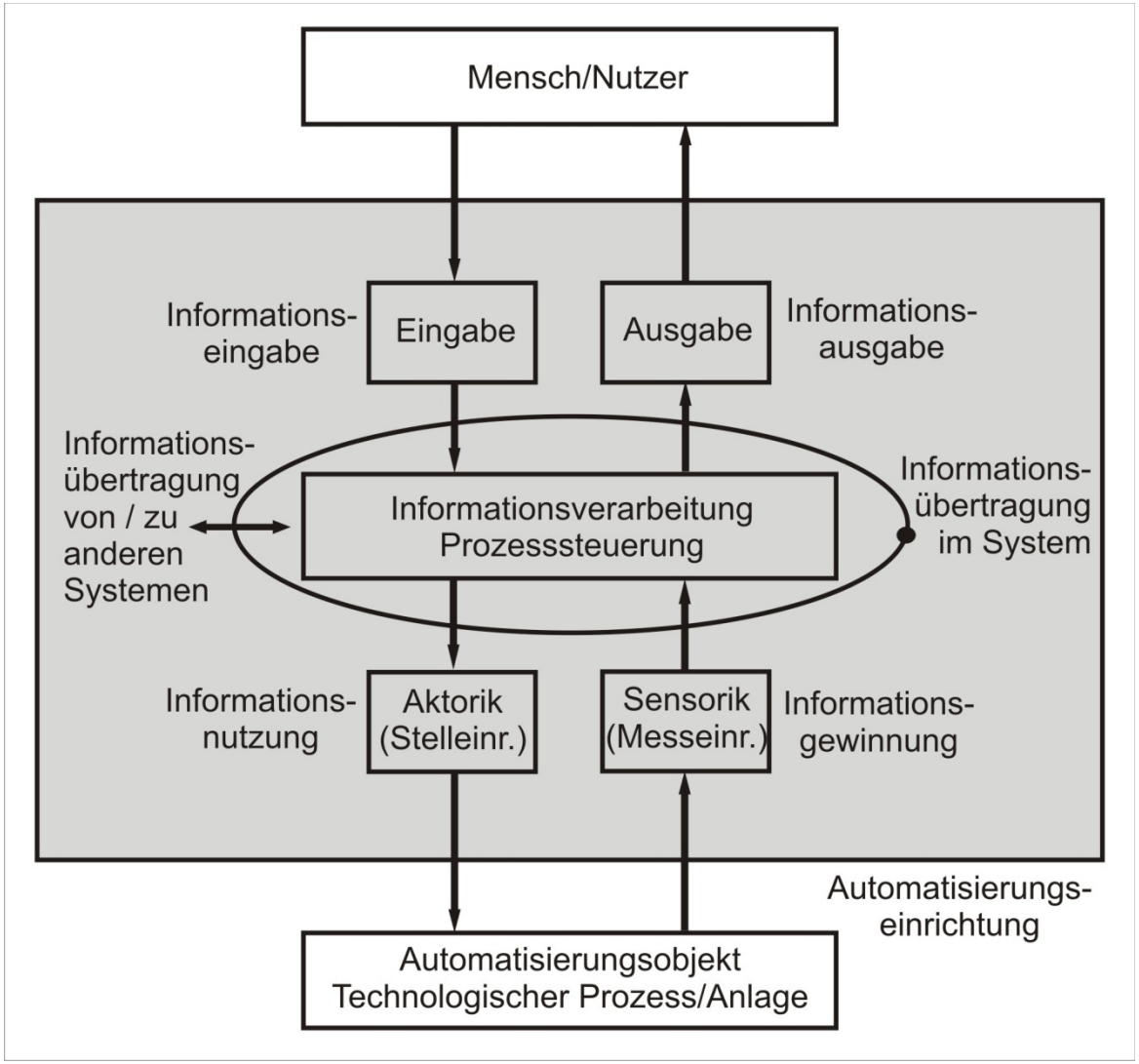


Abbildung 4: Grundprinzip der Automatisierung

1.3. Technologischer Prozess

Definition 7: 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.

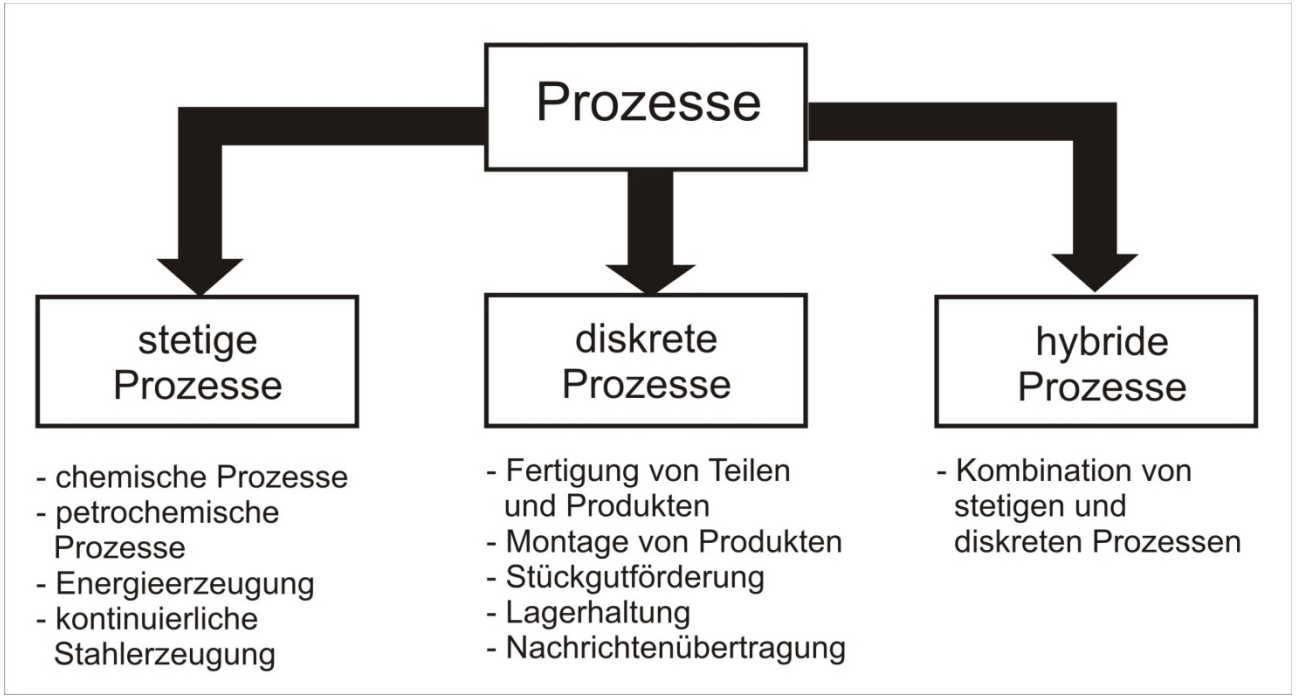


Abbildung 5: Unterteilung von technologischen Prozessen

Man kann stetige, diskrete und hybride Prozesse unterschieden, siehe Abbildung 5. Häufig werden technologische Prozesse durch Automatisierungsanlagen gesteuert.

1.4. Steuerung und Regelung

Prinzipiell lassen sich Steuerungssysteme mit offener Wirkungskette und Steuerungssysteme mit geschlossener Wirkungskette unterscheiden. In Abbildung 6 ist die prinzipielle Struktur einer Steuerung mit offener Wirkungskette dargestellt.

Definition 8: 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 6: 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 9: 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.

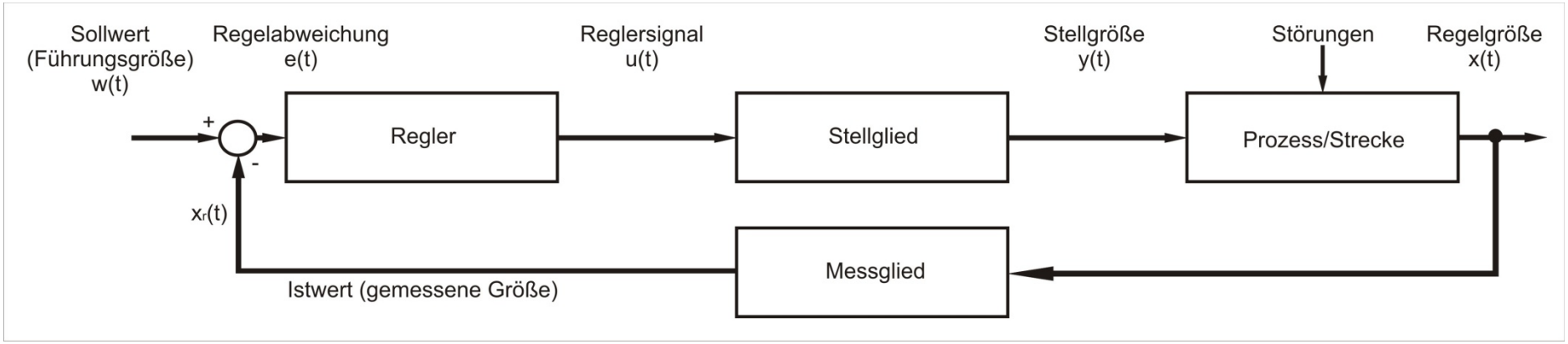


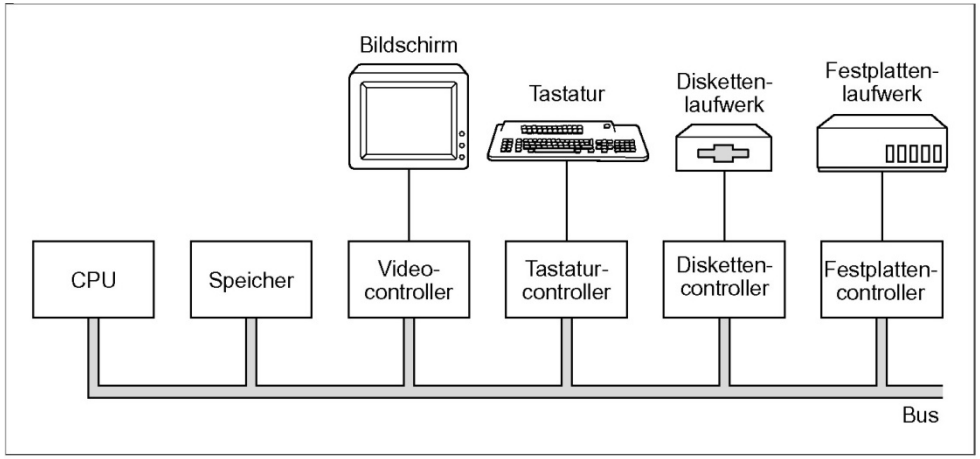
Abbildung 7: Wirkungskette einer Regelung

2. Rechnerarchitekturen für Echtzeitsysteme

Das Zeitverhalten eines mehrschichtigen Systems hängt von allen seinen Ebenen ab. Das Gesamtsystem ist nur dann echtzeitfähig, wenn alle Ebenen echtzeitfähig sind. So werden die Echtzeiteigenschaften eines Rechnersystems sowohl von der Hardware wie auch von der Software bestimmt.

Ein auf vorhersagbares Zeitverhalten hin entwickeltes Programm wird unvorhersagbar, wenn es auf einem Rechnersystem ausgeführt wird, dessen Befehlsausführungszeiten nicht vorhersagbar sind. Die Hardware des Rechnersystems bildet somit die Basis für ein Echtzeitsystem.

2.1. Architektur von Mikrorechnersystemen



Ein Betriebssystem ist sehr eng mit der Hardware eines Rechnersystems verbunden und erweitert den Befehlssatz des Rechners und verwaltet dessen Ressourcen. Die Zusammenarbeit zwischen Hardware und Betriebssystem entscheidet über die Leistungsfähigkeit des Gesamtsystems.

Eine sehr einfache Rechnerarchitektur ist in Abbildung 8 zur sehen.

Abbildung 8: Einfache Architektur eines Mikrorechnersystems

Grundsätzlich bauen die Strukturen auch heutiger Rechnersysteme prinzipiell auf der John von Neumann-Architektur auf, siehe Abbildung 9.

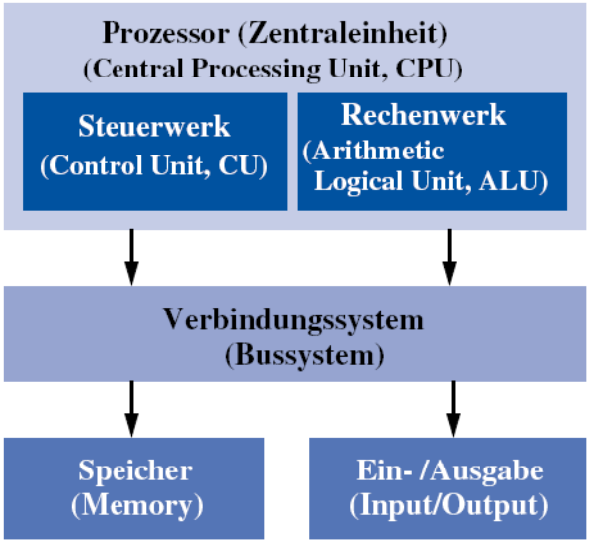


Abbildung 9: von-Neumann-Architektur

Im Folgenden sollen einige Hardwarekomponenten kurz vorgestellt werden.

2.1.1. Prozessoren

Das Herzstück eines jeden Computers ist der Prozessor, als **CPU** (**C**entral **P**rocessor **U**nit) bezeichnet. Hier gibt es unterschiedliche Ansätze in der Entwicklung. Jede CPU verfügt über einen Befehlssatz, d.h. eine Anzahl von Anweisungen, die sie ausführen kann. Auf Grund der Verschiedenartigkeit dieser Befehlssätze ist verständlich, warum beispielsweise ein **Intel-Pentium-Prozessor** nicht die Programme eines **DEC-Alpha-Prozessors** oder

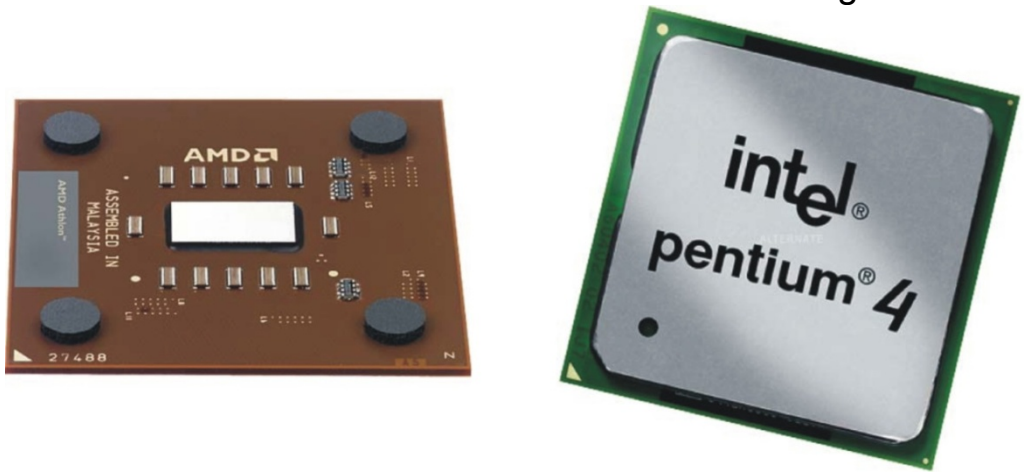
eines **SUN-Sparc-Prozessors** verarbeiten kann und umgekehrt der Alpha-Prozessor nicht die Programme eines Intel-Pentium-Prozessors, genauso, wie auch der Sparc-Prozessor diese nicht verarbeiten kann.

Bezüglich der Anzahl verfügbarer Befehle haben sich zwei grundsätzliche Strukturen entwickelt:

- **CISC** (***c***omplex ***i***nstruction ***s***et ***c***omputer = CPU mit komplexen Befehlssatz) und
- **RISC** (***r***educed ***i***nstruction ***s***et ***c***omputer = CPU mit reduziertem Befehlssatz).

Die **RISC** brauchen weniger Transistorfunktionen, sind auf kleinerem Platz auf dem **Chip** unterzubringen und damit schneller und aufgrund der geringeren Komplexität zuverlässiger.

Auch wenn die fehlenden Befehle durch mehrere einfache Befehle ersetzt werden müssen, stellt sich letztlich heraus, das **RISC** wesentlich schneller sind als vergleichbare **CISC**. Der IBM-Forscher **John Cocke** bewies 1974, dass ganze 20 Prozent aller möglichen Rechenoperationen eines Prozessors 80 Prozent der Rechenarbeit leisten. Daraufhin wurde die RISC-Technologie entwickelt.

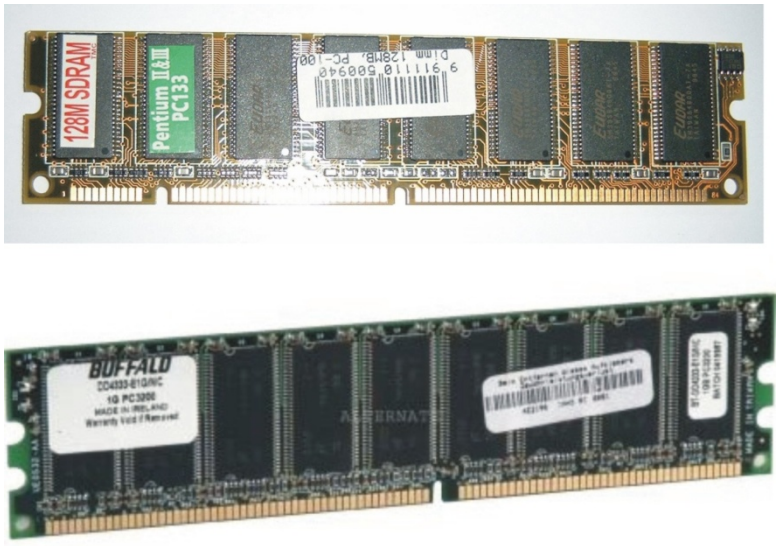


Im Bereich der Personalcomputer (CISC) haben sich zwei Konkurrenten in der Prozessorentwicklung herausgebildet, die sich ständig durch neue Entwicklungen und Marketingstrategienen neu Marktanteile sichern wollen. In der Abbildung 10 sind links ein **AMD Athlon** XP 3200+ und rechts der Konkurrent **Intel Pentium 4** Extreme Edition 3400 MHz zu sehen.

Abbildung 10: Moderne Prozessoren für den Personalcomputer

2.1.2. Hauptspeicher

Die Anzahl der Register eines Prozessors ist sehr beschränkt. Die zur Bearbeitung benötigten Daten und die Programme müssen in einem separaten, größeren Speicher, dem Hauptspeicher, untergebracht werden. Ein Prozessor benötigt einen direkten, adressbezogenen Zugriff auf einzelne Speicherzellen des Arbeitsspeichers, sowohl um einzelne Befehle zu lesen, als auch um einzelne Daten zu lesen oder zu schreiben.
Idealerweise sollte der Hauptspeicher genauso schnell wie die CPU sein, da er sonst diese „ausbremst". Weiterhin sollte er auch sehr groß und sehr billig sein. Da diese Kriterien sehr schwer zu verwirklichen sind, muss man für diesen Bereich den Kompromiss zwischen der Speichergröße, -geschwindigkeit und dem Preis eingehen.



Die schnellste Speicherstruktur ist in der CPU selbst untergebracht und wird als Registerstruktur bzw. Registersatz bezeichnen.

Etwas langsamer ist der Cache, der teilweise ebenfalls auf dem Chip oder in unmittelbarer Nähe zu diesem angeordnet ist. Die Kosten für diese Speicherart sind vergleichbar extrem hoch, da sie sehr schnell sind.
Um einen sehr großen, aber trotzdem preiswerten Speicherbereich zu haben, wird in den Computern Hauptspeicher eingesetzt.

In Abbildung 11 ist oben ein **SDRAM** mit 128 Mbyte und 133 MHz Taktrate und unten ein **DIMM** mit 1 Gbyte **DDR** mit 400 MHz Taktrate zu sehen.

Abbildung 11: Speicherbänke

Heute erreicht man bei „modernen“ Personalcomputern schon Größenordnungen bis 4 Gbyte Speicherraum. Da dies nicht komplett ausreichend ist, werden weitere Speichermedien extern, als Ein- / Ausgabegeräte eingesetzt.

2.1.3. Ein- / Ausgabegeräte

Die klassischen externen Speichermedien haben eine magnetische Schicht zur Speicherung der Daten. Die langsamste und kapazitiv kleinste Möglichkeit sind Disketten mit 1,44 MByte bzw. 2,88 Mbyte, oder erweitert bis zu 200 Mbyte Speicherkapazität. Abbildung 12 zeigt ein Standarddisketten-laufwerk.

Abbildung 12: Diskettenlaufwerk

Werden sehr große Speicherbereiche für relativ kleinen Preis benötigt, sind natürlich Festplattenlaufwerke eingesetzt.

Immer wieder kann man in der Fachpresse von neuen Kapazitätsgrenzen lesen, in die die Hersteller vorgestoßen sind.

In Abbildung 13 sind zwei Vertreter in heutigen Computern, links für Standardpersonalcomputer von Western Digital WD2500LB mit einer Speicherkapazität von 250 Gbyte und rechts für Laptops von Toshiba MK8026GAX mit 80 Gbyte Speicherkapazität zu sehen.

Abbildung 13: Festplattenlaufwerke

Eine recht interessante Alternative zum Austausch von Daten und Informationen zwischen einzelnen Computersystemen sind die so genannten USB-Sticks, die heute schon in Kapazitäten von mehreren Gbyte vordringen, siehe Abbildung 14.

Abbildung 14: Kingston 1 Gbyte USB-Stick

In dieser Aufzählung sind bewusst nur externe Speichermedien benannt, deren Liste sich natürlich auch noch fortsetzen lässt. Andere Ein- / Ausgabegeräte wie Bildschirm, Tastatur und Maus sind natürlich für den Computer sehr notwendig, sollen aber an dieser Stelle nicht weiter betrachtet werden.

2.1.4. Bussysteme

Bei den Betrachtungen der einzelnen Komponenten wird deutlich, dass ihre Anforderungen an die Geschwindigkeit der Datenübertragungen sehr unterschiedlich sind und sich noch weiter auseinander entwickeln werden. Dabei ist eine Struktur, wie sie Abbildung 8 zeigt nicht mehr zufrieden stellend.

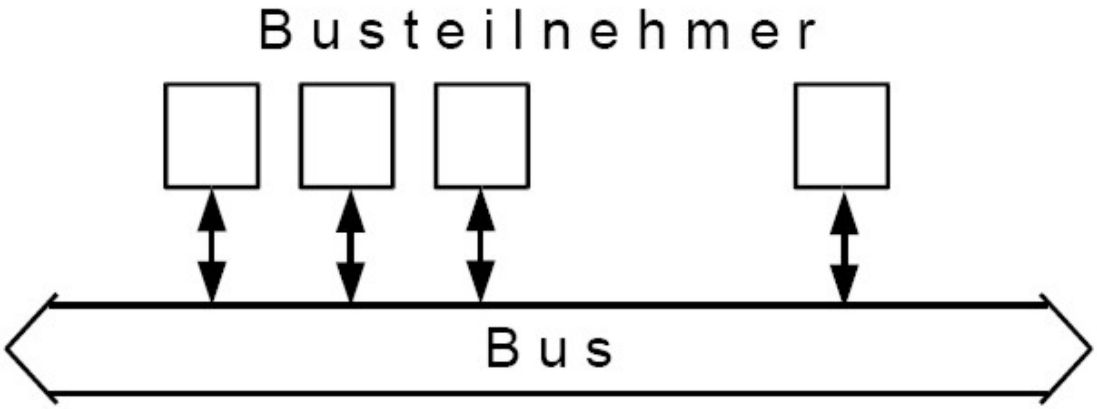


Abbildung 15: Bussystem allgemein

Die Entwicklung ist weg gegangen von der Ein-Bus-Struktur. In Abbildung 16 sieht man die Struktur eines Pentium-System mit mehreren verschiedenen Bus-Arten, die sich in Funktion, Taktrate, Übertragungsbreite und damit in der Übertragungsrate start unterscheiden:

- **ISA**-Bus (**I**ndustry **S**tandard **A**rchitecture),
- **PCI**-Bus (**P**eripheral **C**omponent **I**nterconnect),
- Cache-Bus,
- Lokaler Bus,
- Speicher-Bus
- **IDE**-Bus (**I**ntegrated **D**rive **E**lectronics),
- **SCSI**-Bus (**S**mall **C**omputer **S**ystem **I**nterface),
- **USB**-Bus (**U**niversal **S**erial **B**us),
- Firewire-Bus (IEEE 1394)

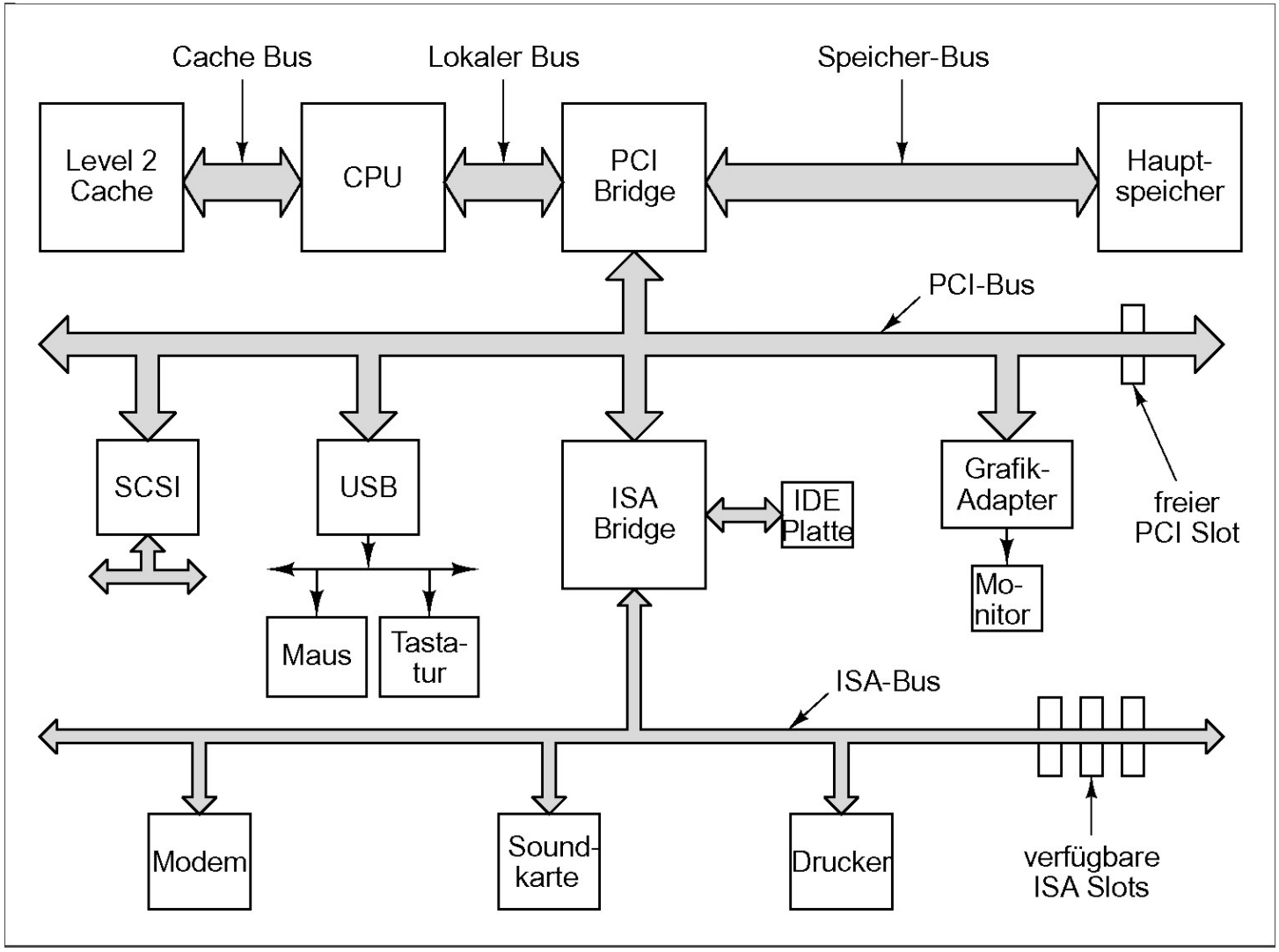
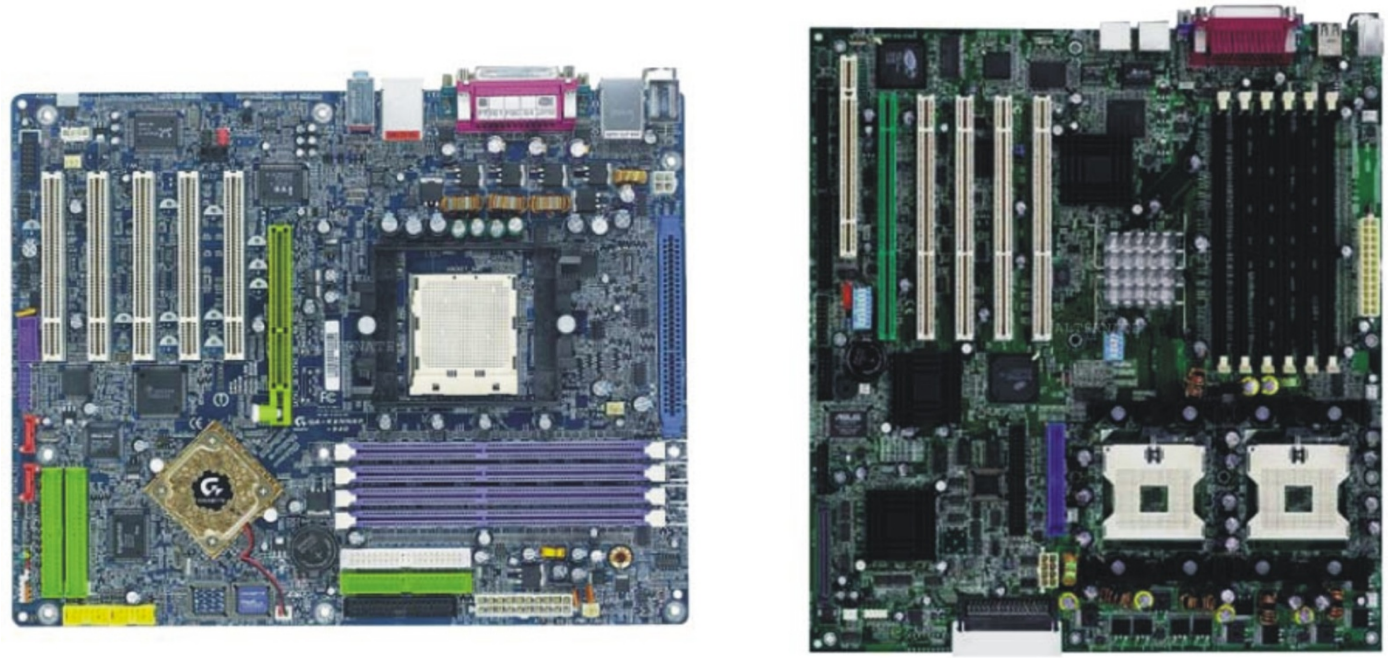


Abbildung 16: Typische Bus-Struktur eines Personalcomputers

Aus Kosten- und Geschwindigkeitsgründen werden die meisten Baugruppen zusammen auf einer Platine untergebracht, den sogenannten Mainboards.



In Abbildung 17 sind Mainboards für die beiden typischen Prozessoren, die in Personalcomputer zum Einsatz kommen abgebildet. Links das Mainboard Gigabyte für AMD-64-Bit-Prozessoren und rechts das Mainboard Asus für 2 Intel-Xeon-Prozessoren. Damit sind jedoch Leistungen erreichbar, wie sie typischerweise in Servern gebraucht werden.

Abbildung 17: Mainboards für den Personalcomputer

2.2. 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, siehe Abschnitt 2.1. Architektur von Mikrorechnersystemen, auf und werden durch folgend aufgeführte Komponenten erweitert.

2.2.1. 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.

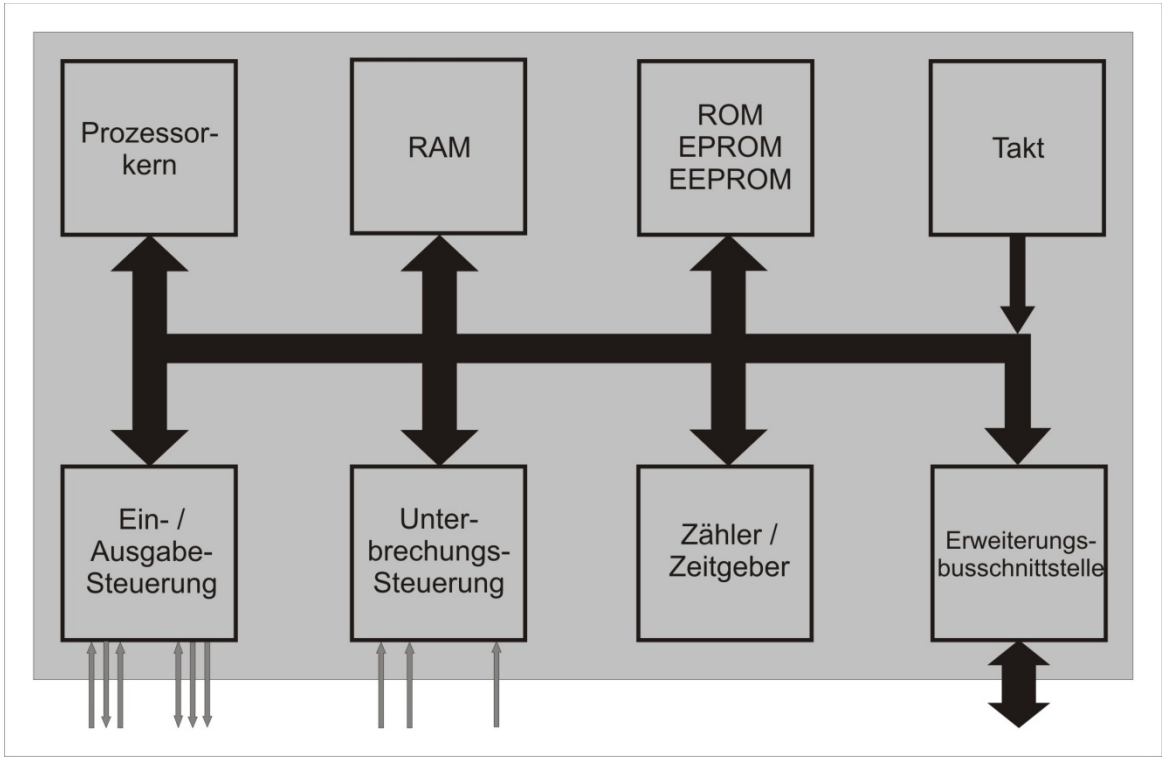


Abbildung 18: Prinzipieller Aufbau eines Mikrocontrollers

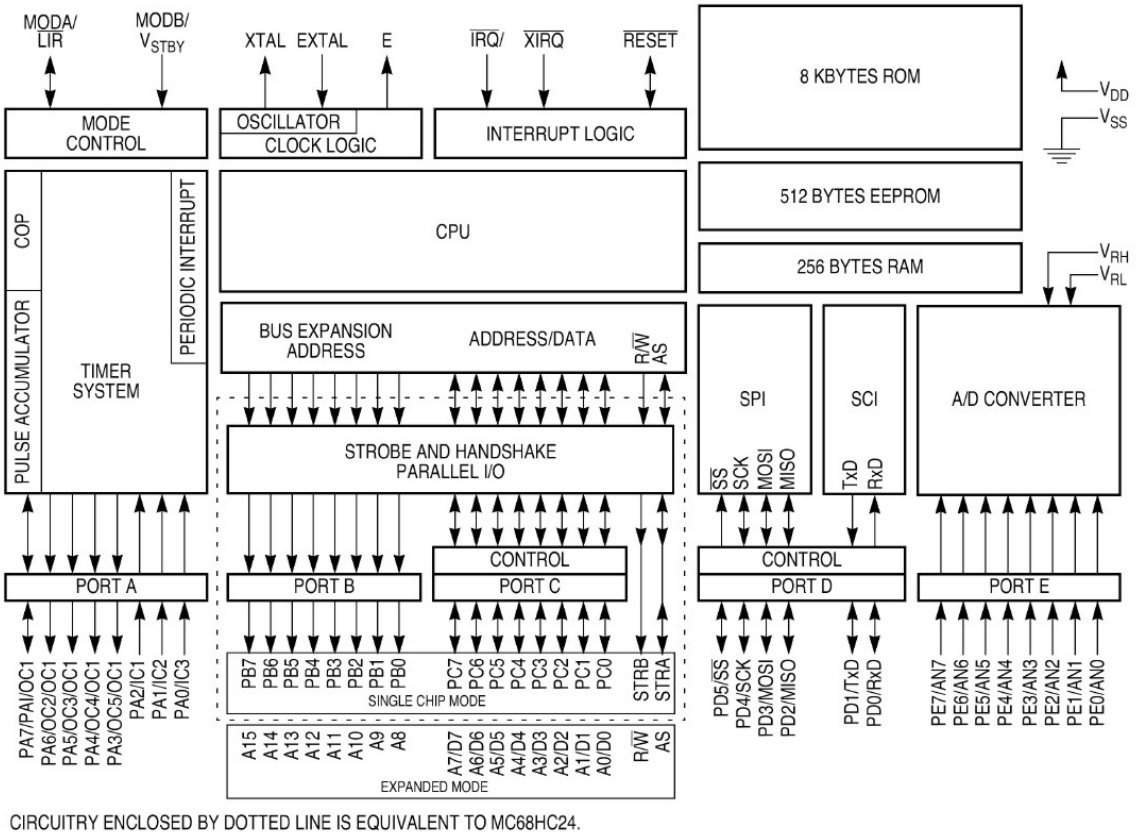


Abbildung 19: Blockschaltbild des Mikrocontrollers 68HC24

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 18 gibt einen Überblick.

Als Beispiel für einen Mikrocontroller sei der der Firma Motorola 68HC24, siehe Blockschaltbild in Abbildung 19 dargestellt.

2.2.1.1. 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 20.

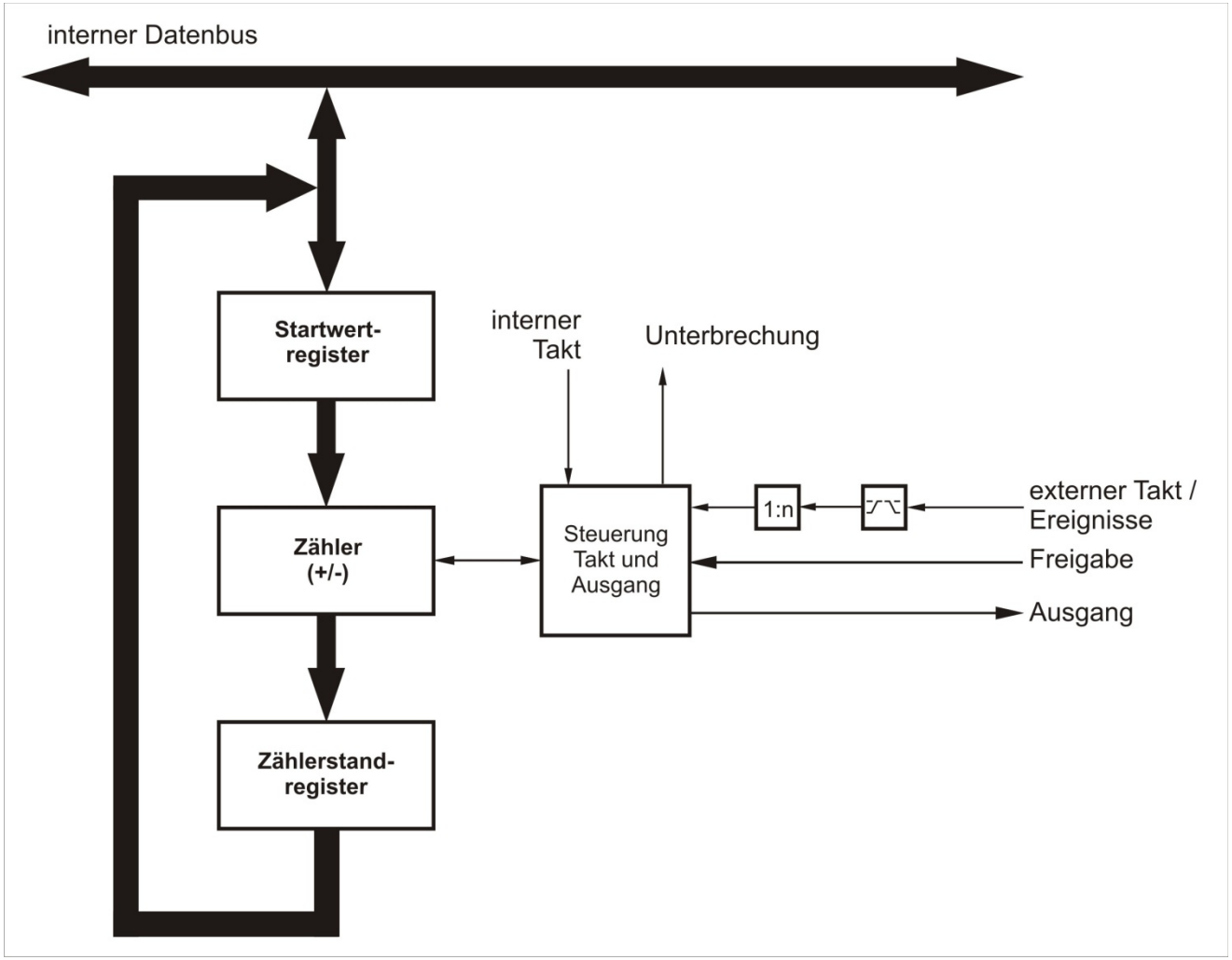


Abbildung 20: Zähler- / Zeitgebereinheit

2.2.1.2. 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 21 gezeigt.

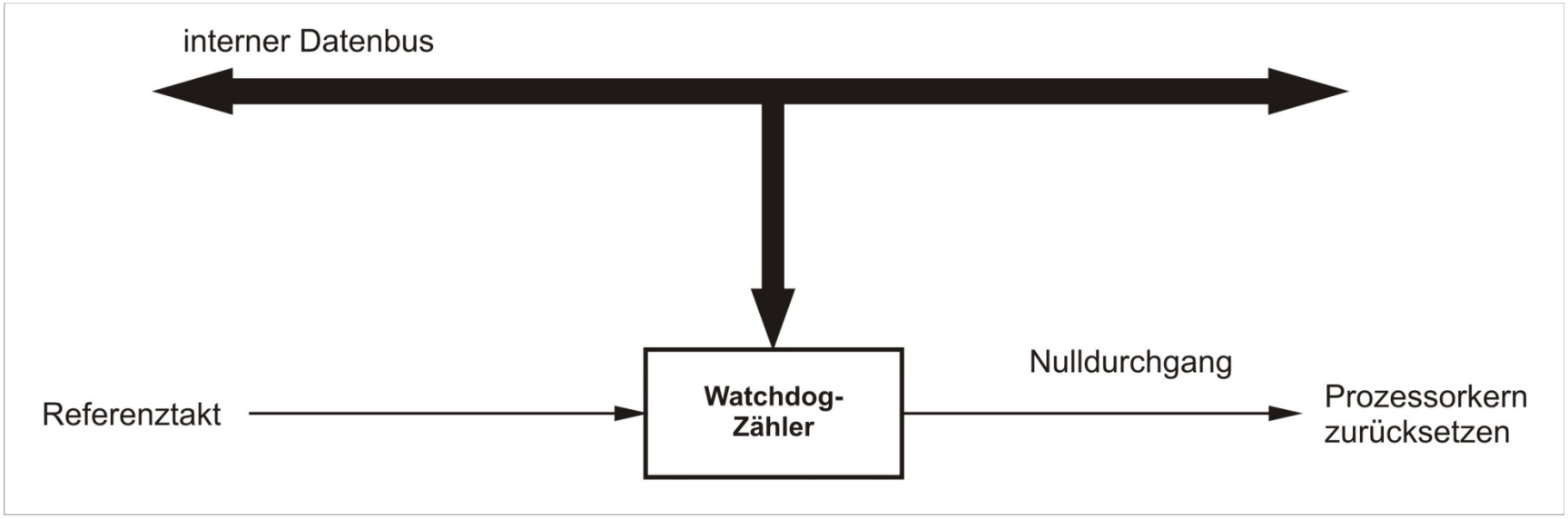


Abbildung 21: Watchdog

2.2.1.3. 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.

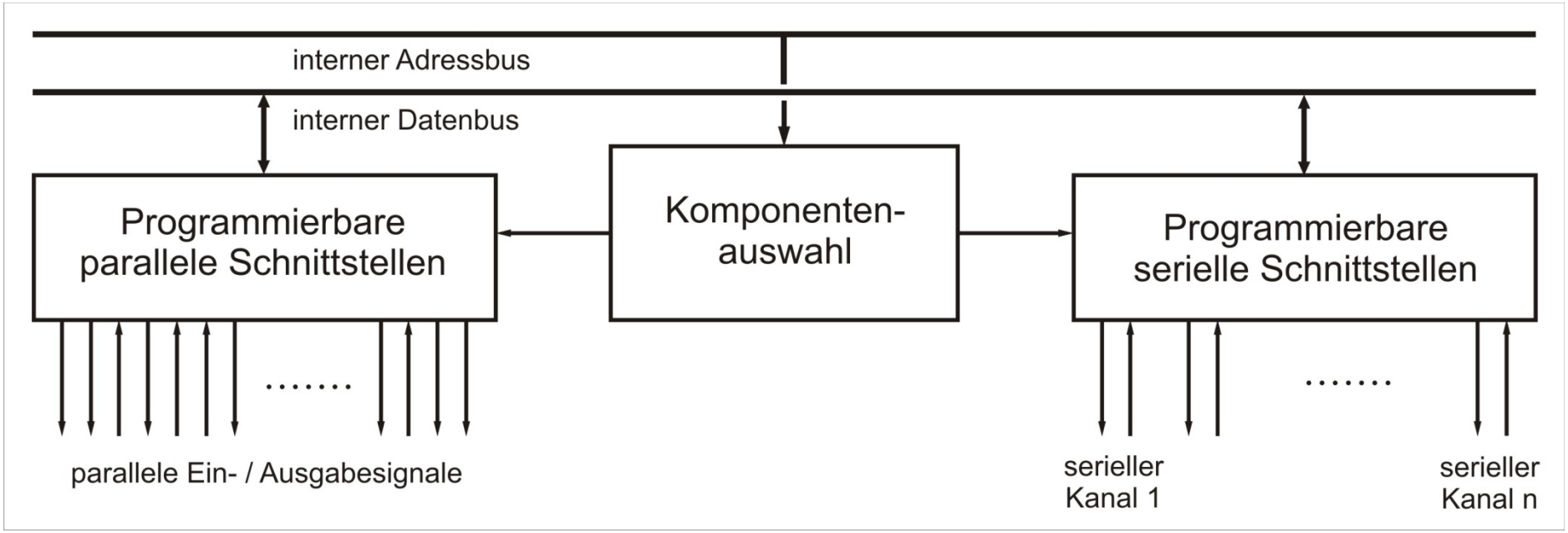


Abbildung 22: Serielle und parallele Ein- / Ausgabekanäle

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.

2.2.1.4. 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.

2.2.1.5. 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**.

2.2.2. 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.

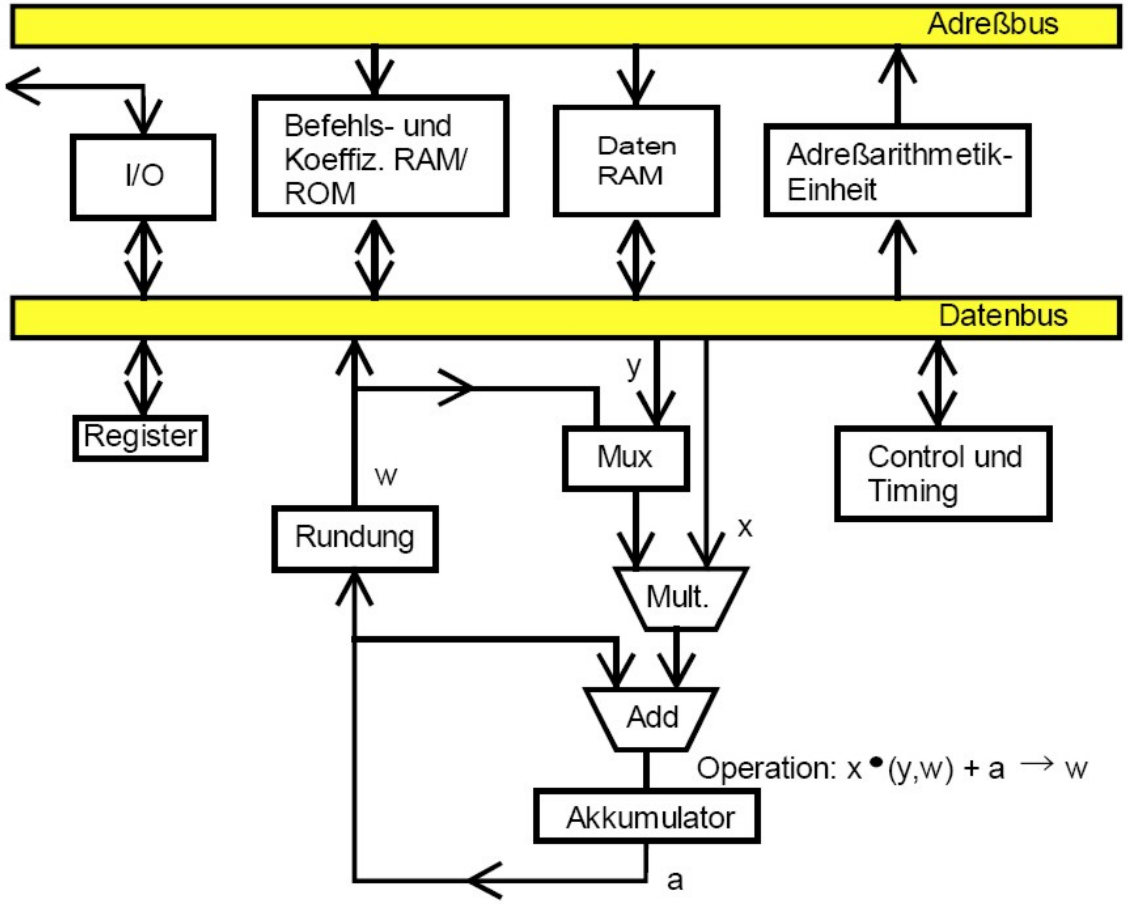


Abbildung 23: Grundlegender Aufbau eines Signalprozessors DSP

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 2: Unterschiede zwischen Mikroprozessoren und Signalprozessoren

2.2.3. Bussysteme für Echtzeitsysteme

Wie im Abschnitt 2.1.4. Bussysteme 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:

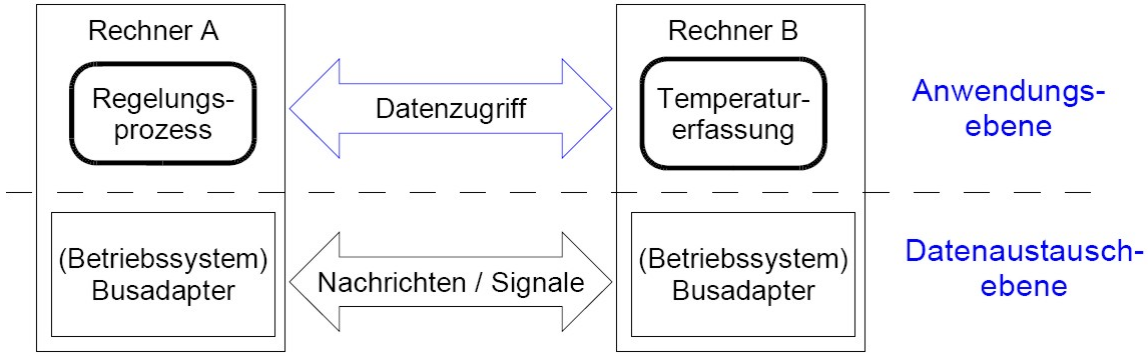
a) Bussysteme werden dort eingesetzt, wo es auf Flexibilität hinsichtlich der Anpassung von Systemen auf sich ändernde bzw. ähnliche Aufgaben ankommt.

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.

2.2.3.1. 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 24.

Abbildung 24: 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 25.

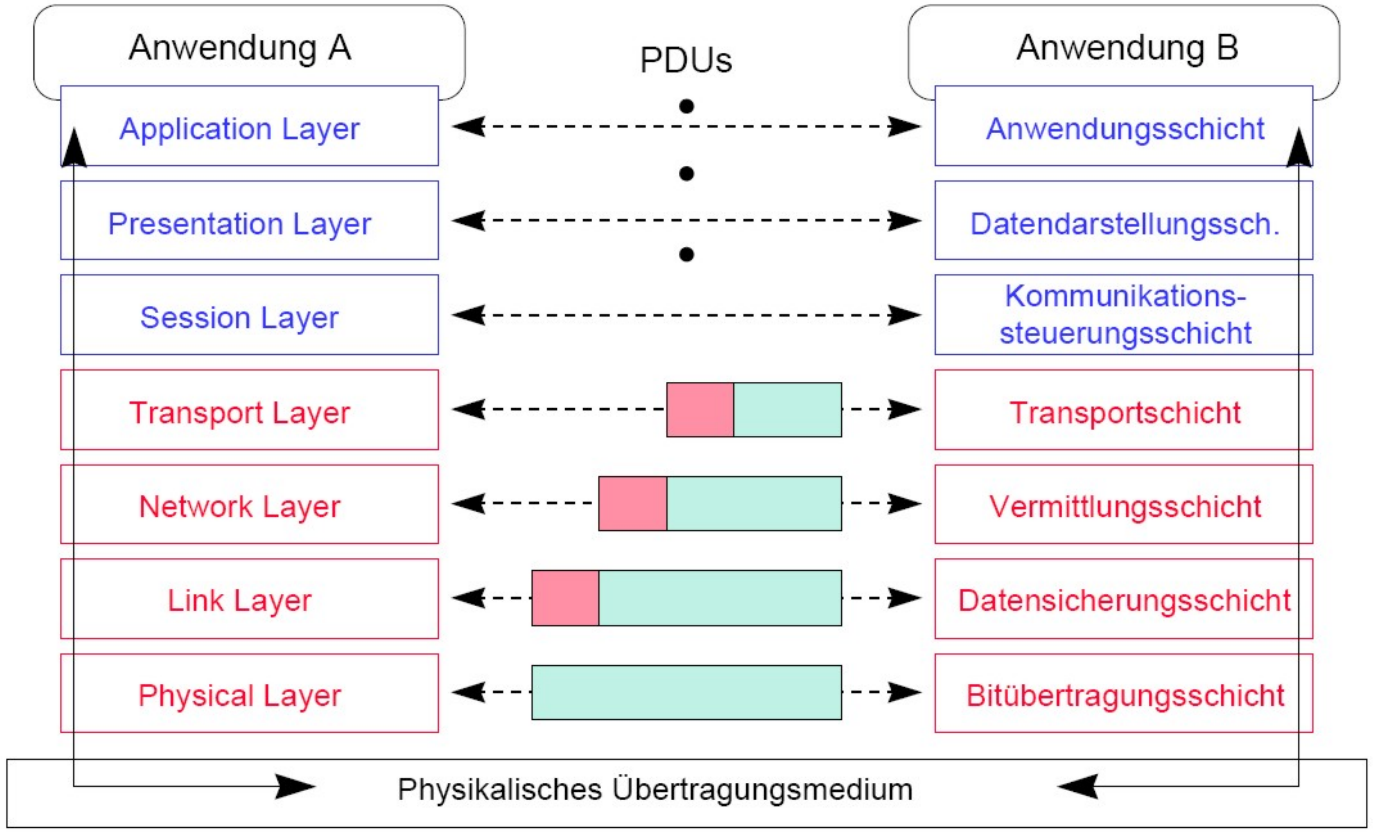


Abbildung 25: OSI-Modell

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 26 und Abbildung 27.

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 26: in Bussystemen benötigte Schichten des OSI-Modells

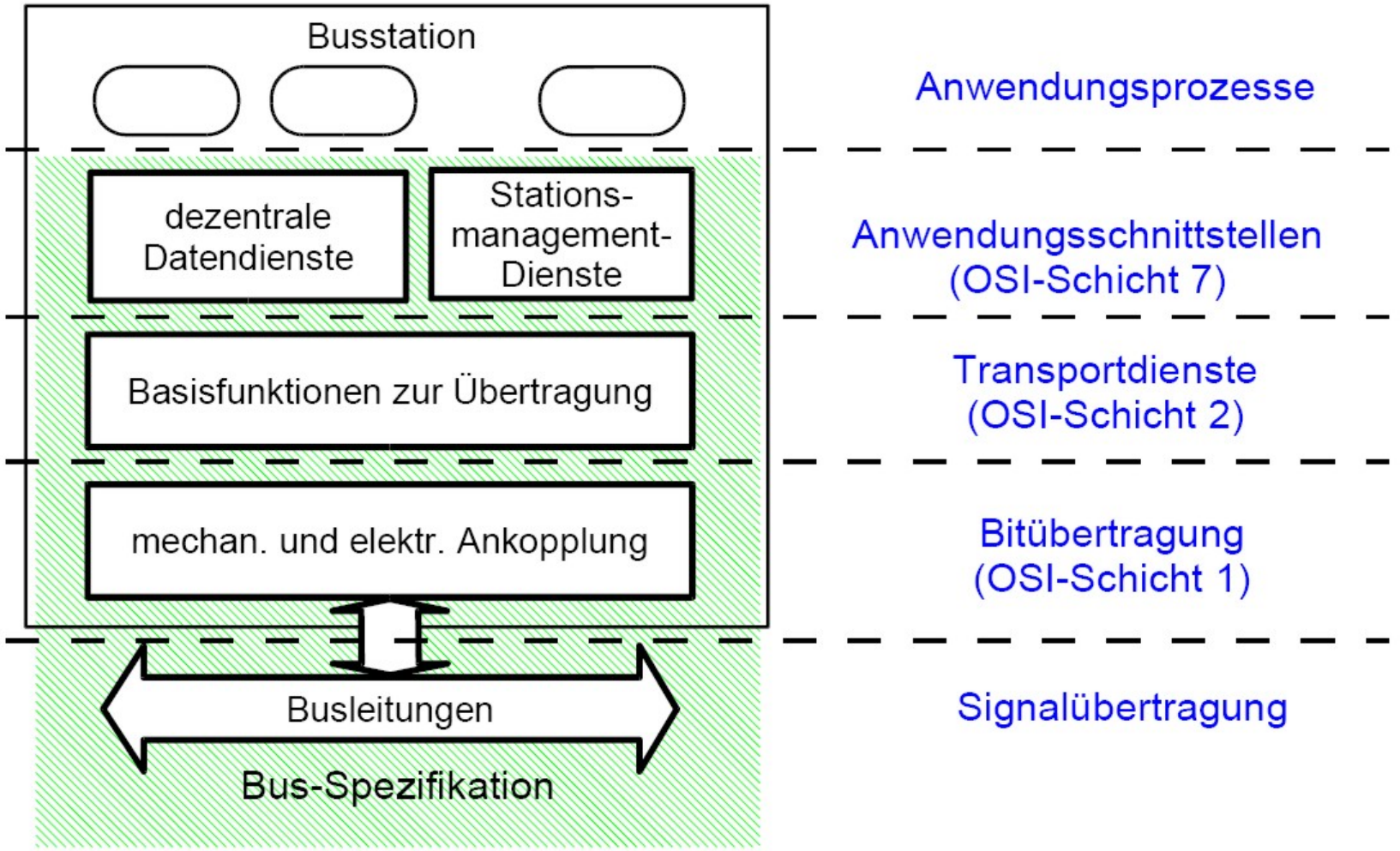
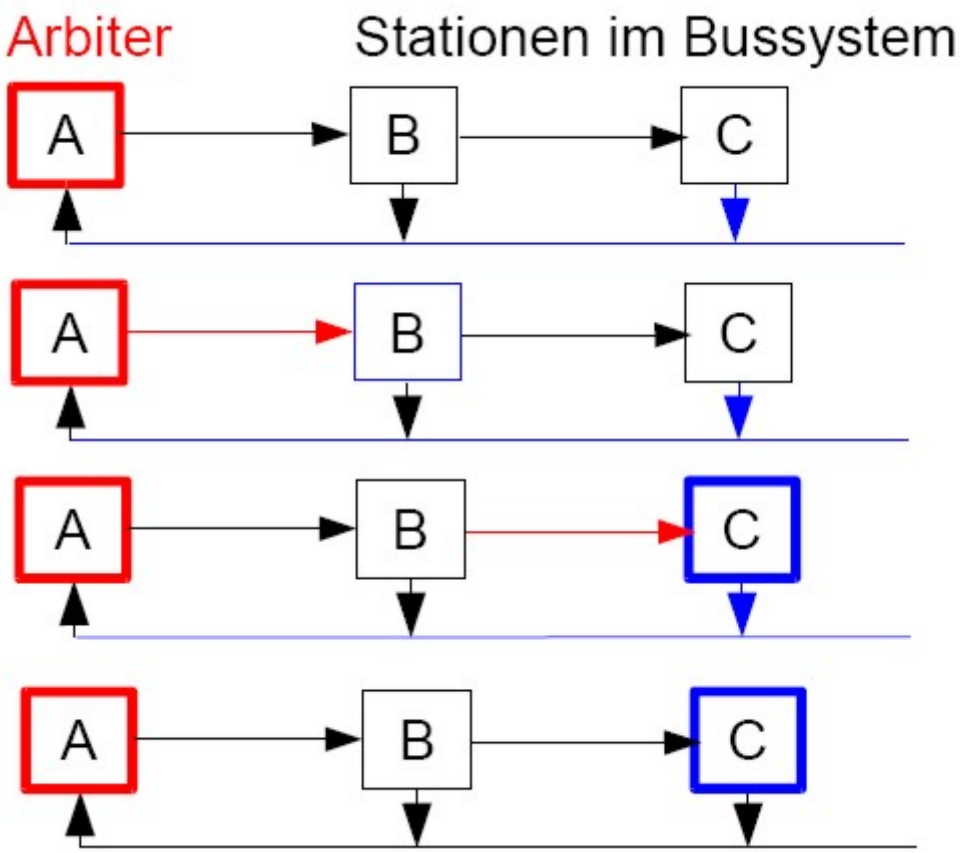


Abbildung 27: Kommunikation in Bussystemen

2.2.3.2. 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.



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 28 ist ein Verfahren mit einem Arbiter dargestellt. Der Zugriff durch Master C erfolgt dabei in vier Schritten.

Abbildung 28: Ablauf der Buszuteilung

2.2.3.2.1. 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.

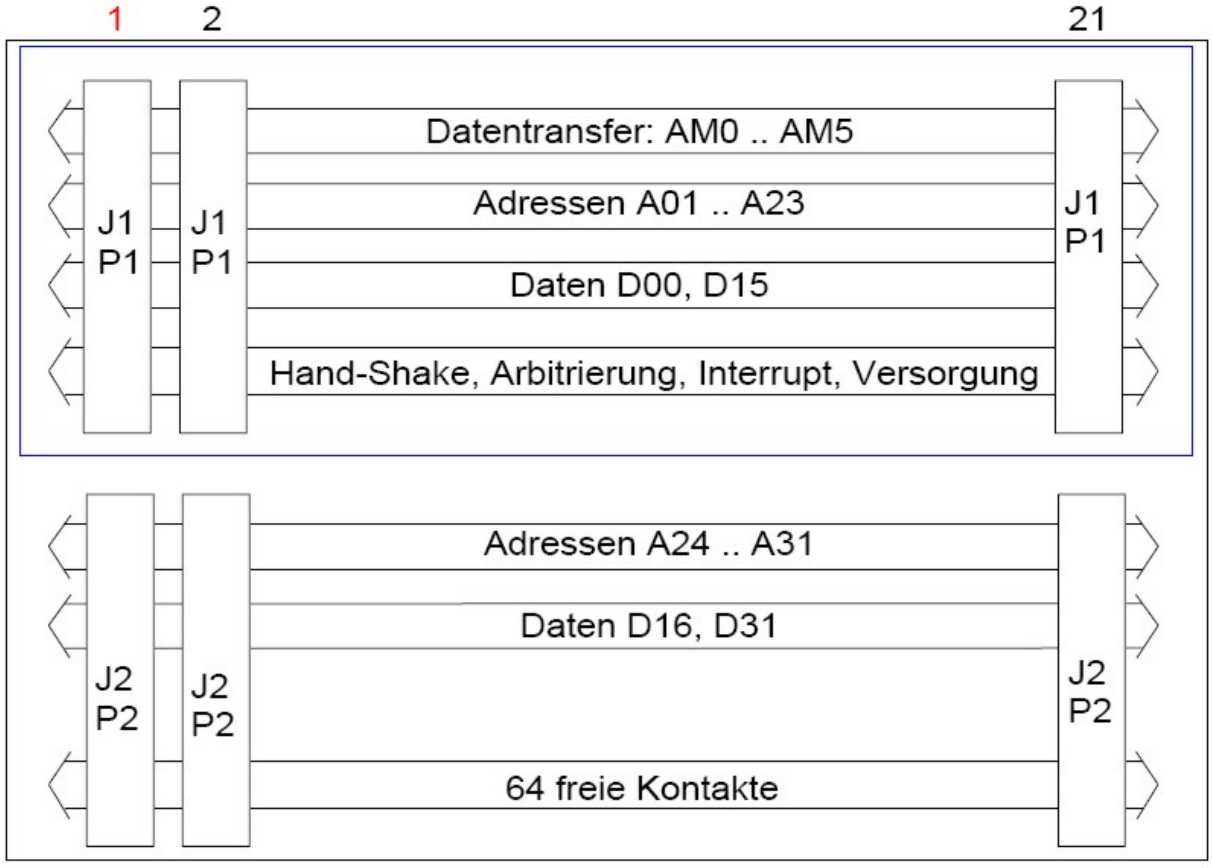
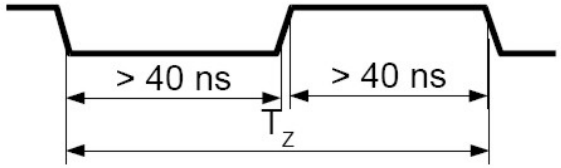


Abbildung 29: Ankopplung des VMEbusses

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 30: 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 30. Damit bleibt der VMEbus deutlich hinter dem PCI-Bus zurück.

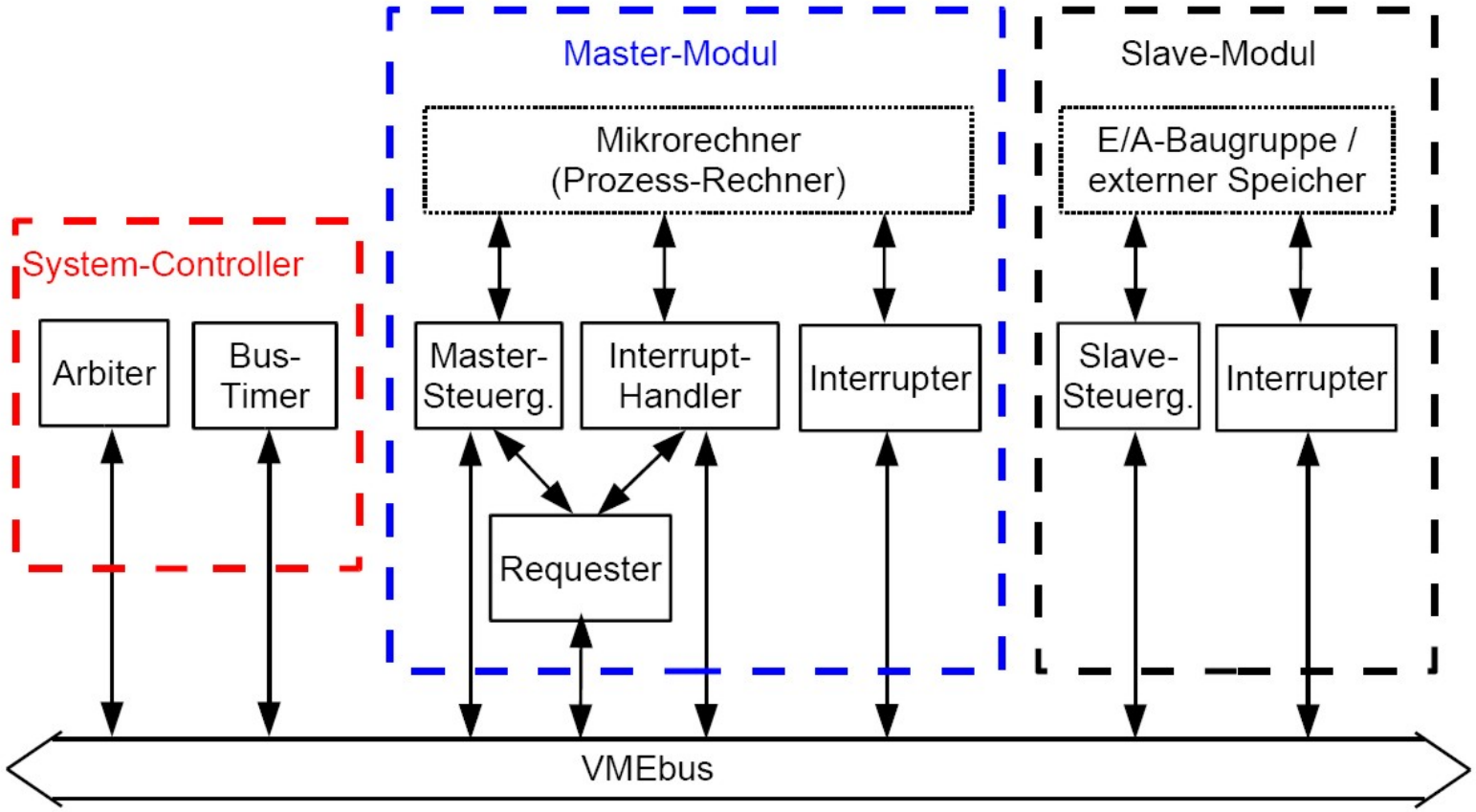


Abbildung 31: Module des VMEbusses

Die grundlegende Architektur des VMEBusses ist in Abbildung 31 dargestellt.

2.2.3.2.2. 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 3: Unterschied des compactPCI zum PCI

Die Unterschiede des compactPCI-Busses zum PCI-Bus sind in Tabelle 3 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.

2.2.3.3. 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 32.

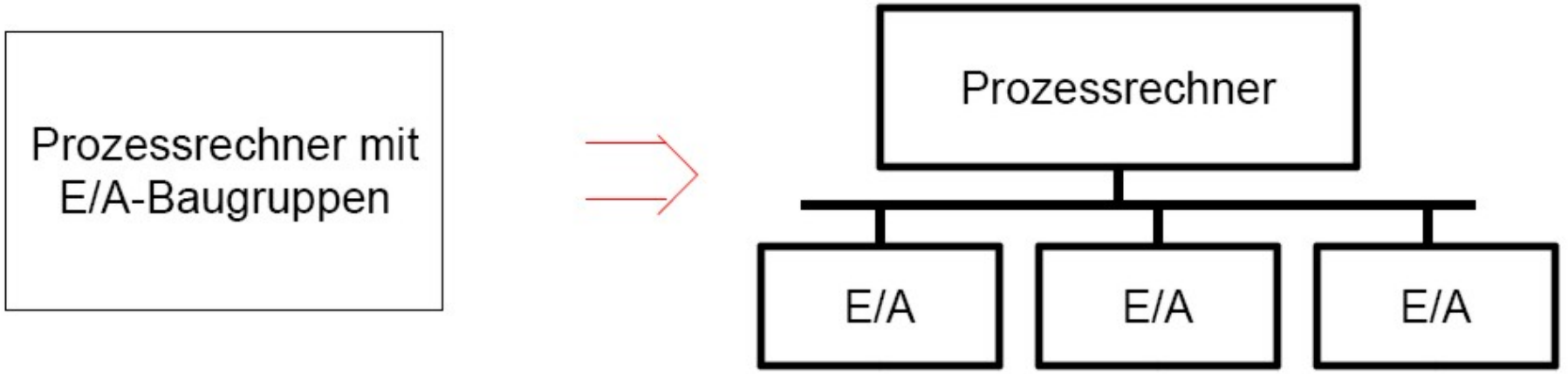


Abbildung 32: 1. Schritt der Entwicklung der Systemarchitektur von Prozessrechnern

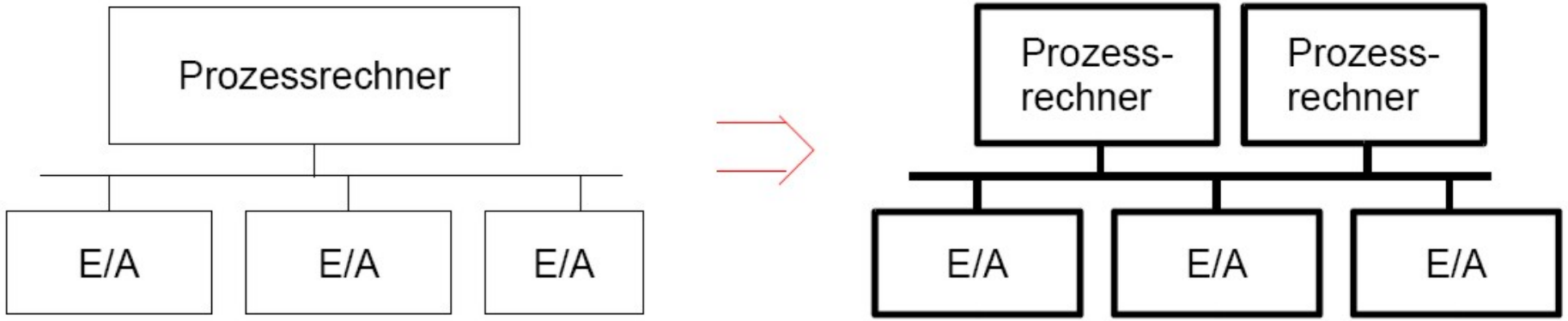


Abbildung 33: 2. Schritt der Entwicklung der Systemarchitektur von Prozessrechnern

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 33. Es entstanden parallele Bussysteme für den Multiprozessorbetrieb. Als ein Beispiel ist der VMEbus zu nennen.

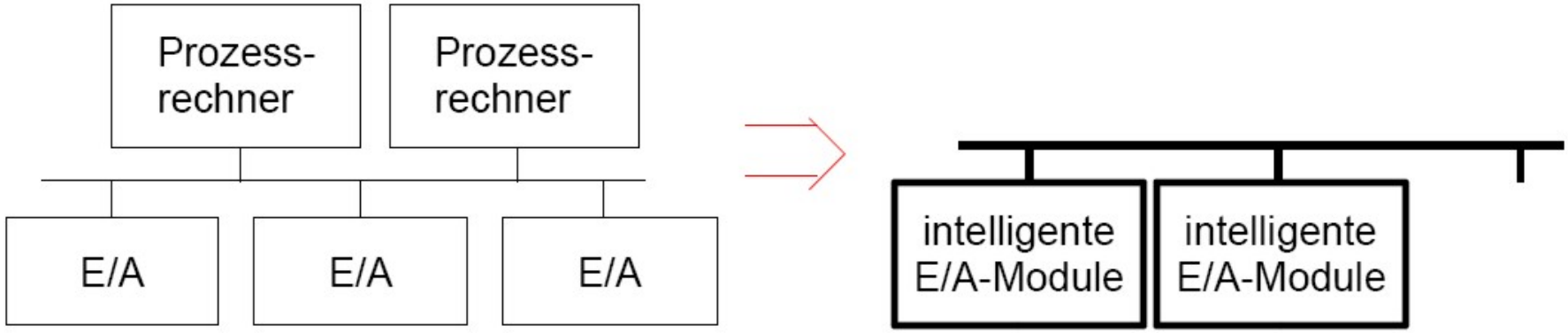


Abbildung 34: 3. Schritt der Entwicklung der Systemarchitektur von Prozessrechnern

Im dritten Schritt, siehe Abbildung 34, 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 4 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 4: Randbedingungen für Feldbussysteme

2.2.3.3.1. Profibus (PROcess FIeld 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 35:

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

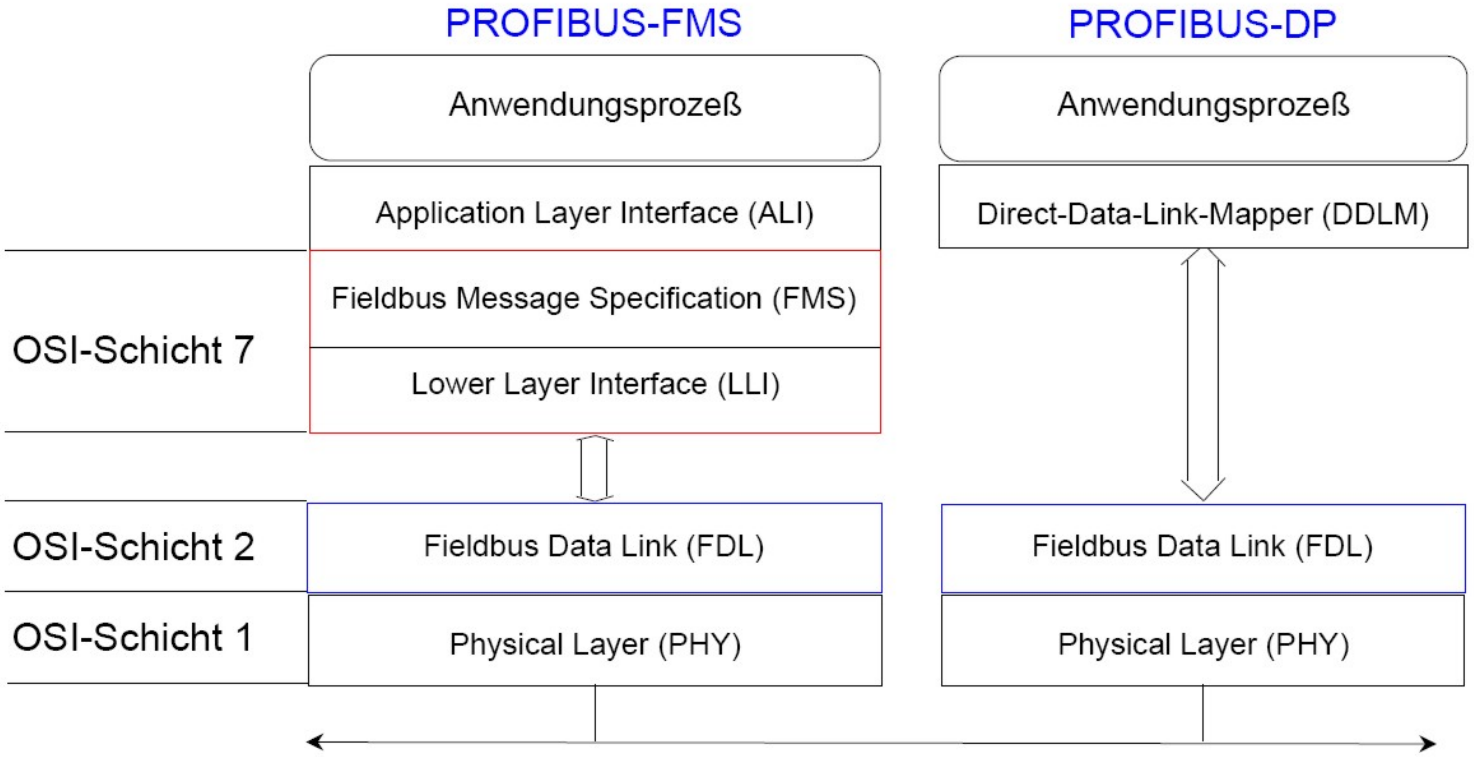


Abbildung 35: Architektur von PROFI-Busstationen

2.2.3.3.2. 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 36 abgebildet.

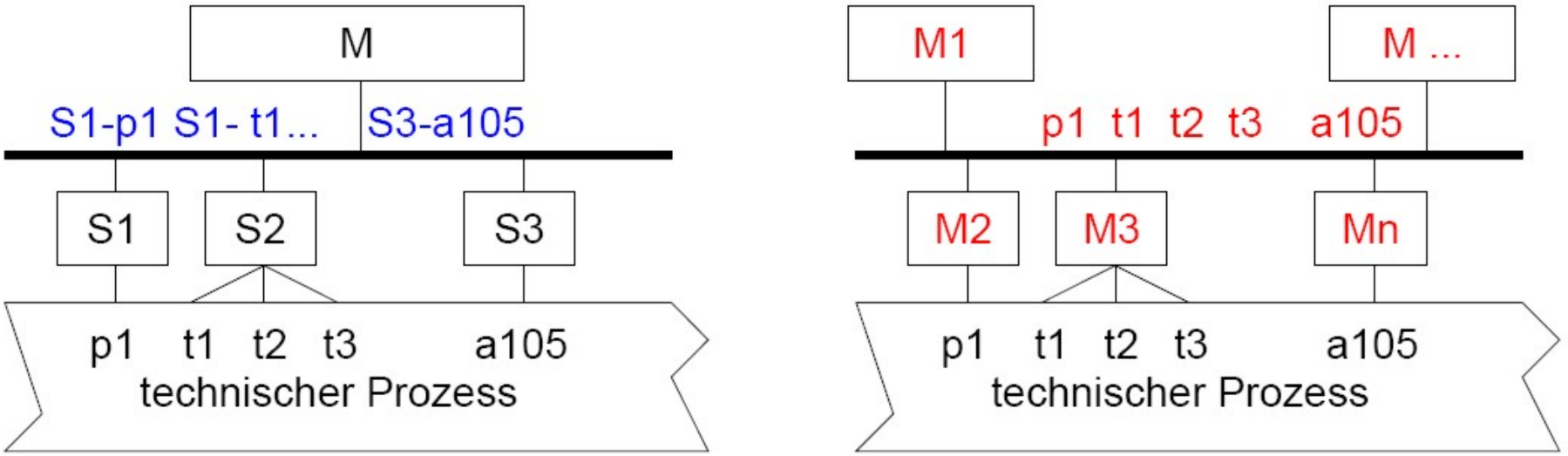


Abbildung 36: Kommunikationsidee im CAN-Bus

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)**.

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 10: 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 11: 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 12: 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 37.

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.

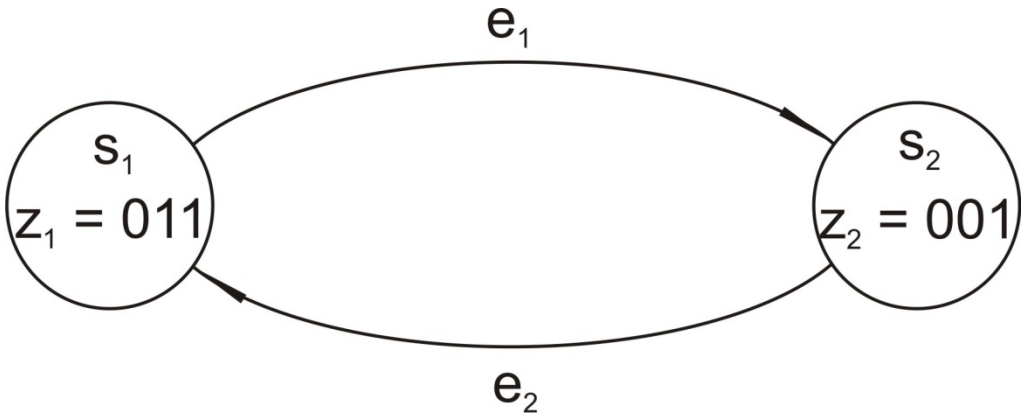


Abbildung 37: Zustandsdiagramm bzw. Automatengraph

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 13: 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 38.

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“

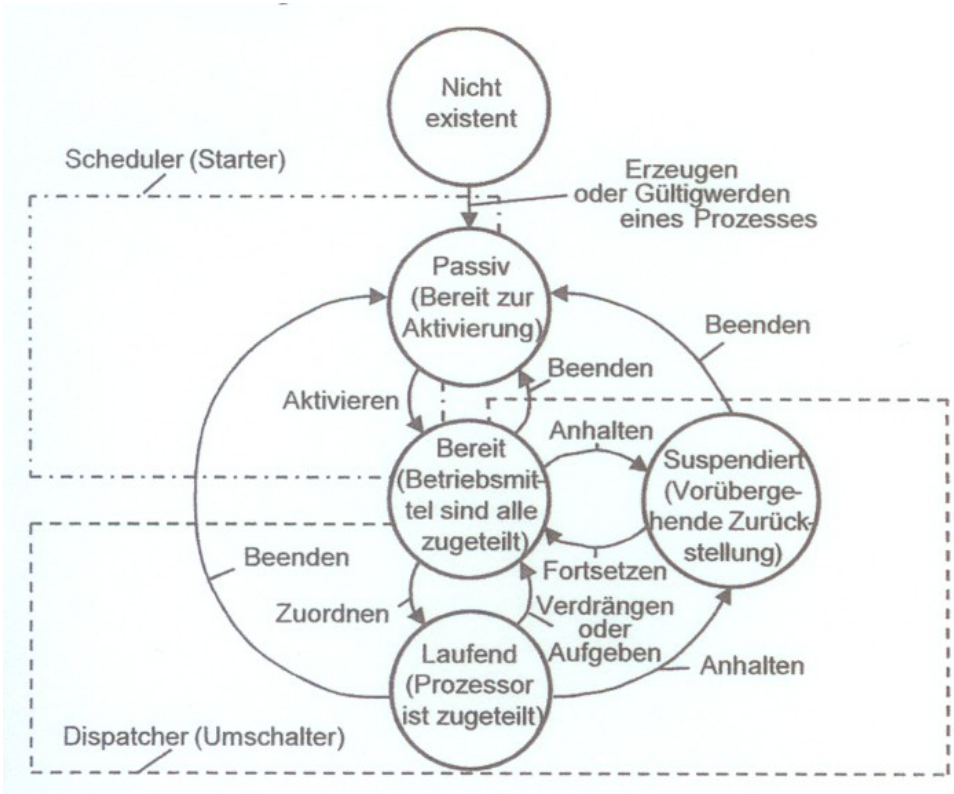


Abbildung 38: Prozessmodell in Betriebssystemen

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“).

4. Betriebssysteme

4.1. Zitat von Microsoftgründer Bill Gates und die Reaktion von General Motors

Auf einer Computermesse hat **Bill Gates** die Computerindustrie mit der Autoindustrie verglichen und das folgende Statement gemacht:

> **"Wenn General Motors (GM) mit der Technologie so mitgehalten hätte wie die Computerindustrie, dann würden wir heute alle 25-Dollar-Autos fahren, die 1000 Meilen pro Gallone Sprit fahren würden".**

Als Antwort darauf veröffentlichte General Motors (von Mr. Welch selbst) eine Presseerklärung mit folgendem Inhalt:

> **Wenn General Motors eine Technologie wie Microsoft entwickelt hätte, dann würden wir alle heute Autos mit folgenden Eigenschaften fahren:**
>
> 1. Ihr Auto würde ohne erkennbaren Grund zweimal am Tag einen Unfall haben.
> 2. Jedes Mal, wenn die Linien auf der Straße neu gezeichnet würden, müsste man ein neues Auto kaufen.
> 3. Gelegentlich würde ein Auto ohne erkennbaren Grund auf der Autobahn einfach ausgehen, und man würde das akzeptieren, neu starten und weiterfahren.
> 4. Wenn man bestimmte Manöver durchführen würde, wie z.B. eine Linkskurve, würde das Auto einfach ausgehen und sich weigern, neu zu starten. Man müsste dann den Motor erneut installieren.
> 5. Man könnte nur alleine in dem Auto sitzen, es sei denn, man würde "Car95" oder "CarNT" kaufen. Aber dann müsste man jeden Sitz einzeln bezahlen.
> 6. Macintosh würde Autos herstellen, die mit Sonnenenergie fahren, zuverlässig laufen, fünfmal so schnell und zweimal so leicht zu fahren wären, aber sie würden nur auf 5% der Straßen laufen.

7. Die Kontrollleuchte und die Warnlampe für Temperatur und Batterie würde durch eine "Genereller Auto-Fehler"-Warnlampe ersetzt werden.
8. Neue Sitze würden erfordern, dass alle dieselbe Gesäßgröße haben.
9. Das Airbag-System würde "Sind Sie sicher?" fragen, bevor es auslöst.
10. Gelegentlich würde das Auto Sie ohne jeden erkennbaren Grund aussperren. Sie könnten nur wieder mit einem Trick aufschließen, und zwar müsste man gleichzeitig den Türgriff ziehen, den Schlüssel drehen und mit einer Hand an die Radioantenne fassen.
11. General Motors würde Sie zwingen, mit jedem Auto einen De-luxe-Kartensatz der Firma Rand McNally (seit Neustem eine GM-Tochter) mitzukaufen, auch wenn Sie diesen Kartensatz nicht brauchen oder möchten. Wenn Sie diese Option nicht wahrnehmen, würde das Auto sofort 50% langsamer werden (oder schlimmer). Darüber hinaus würde GM deswegen ein Ziel von Untersuchungen der Justiz werden.
12. Immer dann, wenn ein neues Auto von GM vorgestellt werden würde, müssten alle Autofahrer das Autofahren neu erlernen, weil keiner der Bedienhebel genau so funktionieren würde, wie in den alten Autos.
13. Man müsste den "Start-Knopf" drücken, um den Motor auszuschalten.

4.2. Was ist ein Betriebssystem

Um ein Rechnersystem für bestimmte Aufgaben benutzen zu können, benötigt man ein **vollständiges Programm**. Vollständig bedeutet dabei, dass das Programm alle Anweisungen enthält, um die zu verarbeitenden Daten einzulesen (von Tastatur, Diskette, Scanner, usw.), Algorithmen auf sie anzuwenden und die Ergebnisse geeignet darzustellen (Bildschirm, Drucker, Festplatte usw.). Die Anweisungen müssen dazu im Maschinenbefehlssatz des verwendeten Prozessors vorliegen.

Ein vollständiges Programm zu einer bestimmten Aufgabe (z.B. Verwaltung der Konten einer Bank, Informationsserver im Internet, Steuerung einer CNC-Werkzeugmaschine u.v.m.) ist damit sehr komplex und aufwendig herzustellen.

Bedenkt man, dass heutige Rechnersysteme sich selbst innerhalb einer Rechnerfamilie vielfältig in Speicherausstattung, Art und Umfang der angeschlossenen Geräte unterscheiden, so wird klar, dass die Erstellung vollständiger Programme im obigen Sinne für jede mögliche Rechnerkonstellation ein praktisch undurchführbares Unternehmen ist.

Die Lösung dieses Problems heißt auch hier wie in anderen Ingenieurbereichen: **Modularisierung**.

Programme werden in **Module** zerlegt, die zueinander über definierte **Schnittstellen** in Beziehung stehen. Somit ist es möglich, innerhalb eines Programms einen Modul durch einen anderen mit gleicher Schnittstelle zu ersetzen, um das Programm an eine andere Rechnerkonstellation anzupassen.

Definition 14: Betriebssystem

> Betriebssystem (operating system) heißt das vollständige Programm, das zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bildet und insbesondere die Abwicklung von Programmen steuert und überwacht.

In einem vollständigen Programm gibt es aufgabenspezifische Module, bezogen auf das Einsatzgebiet und Module, die von der konkreten Aufgabe relativ unabhängig und für andere Aufgaben in der gleichen Weise einsetzbar sind.

Definition 15: Betriebssystem (2. Möglichkeit)

> Als Betriebssystem wird eine Menge von logisch zusammengehörenden Programmen bezeichnet, die die funktionale Verbindung zwischen der Gerätetechnik (Rechenanlage, Hardware) einerseits und anwendungsspezifischen Programmen sowie dem Anwender (Nutzer) andererseits herstellt.

Die Auswahl und Zusammenstellung der allgemeingültigen Module wird bestimmt durch die eingesetzte Hardware und die Art der Programme, die durch diese Module unterstützt werden sollen.
Sie ist für viele Programme, die auf einem Rechner abgearbeitet werden sollen, gleich und unterscheiden sich wiederum etwas von Rechner zu Rechner.

Es bietet sich daher an, diese allgemeingültigen Module jeweils für einen Rechner zu einem System zusammenzufassen, dem **Betriebssystemkern**.
Die Zusammenfassung der aufgabenspezifischen Module einer Anwendung bildet das **Anwendungsprogramm**, welches über die definierten Schnittstellen auf die Funktionen des Betriebssystemkerns zugreift.

Um eine möglichst bequeme Nutzung von Betriebssystemkern und Anwendungsprogramm zu ermöglichen, erhält der Anwender zum Betriebssystem eine Reihe von nützlichen und unbedingt notwendigen Anwendungsprogrammen, so genannte **Dienstprogramme** dazu.

Betriebssystemkern und Dienstprogramme bilden zusammen das **Betriebssystem**. Die einzelnen Betriebssysteme unterscheiden sich im konkreten Funktionsumfang des Betriebssystemkerns und seiner Programmschnittstellen sowie der Dienstprogramme. Dennoch findet man viele Grundprinzipien in allen Betriebssystemen wieder. In Abbildung 39 ist der Zusammenhang zwischen den Komponenten eines Rechnersystems in dem Schalen- bzw. Schichtenmodell zusammengefasst. Dienstprogramme und das Anwendungsprogramm sind einer gemeinsamen Schale bzw. Schicht zugeordnet.

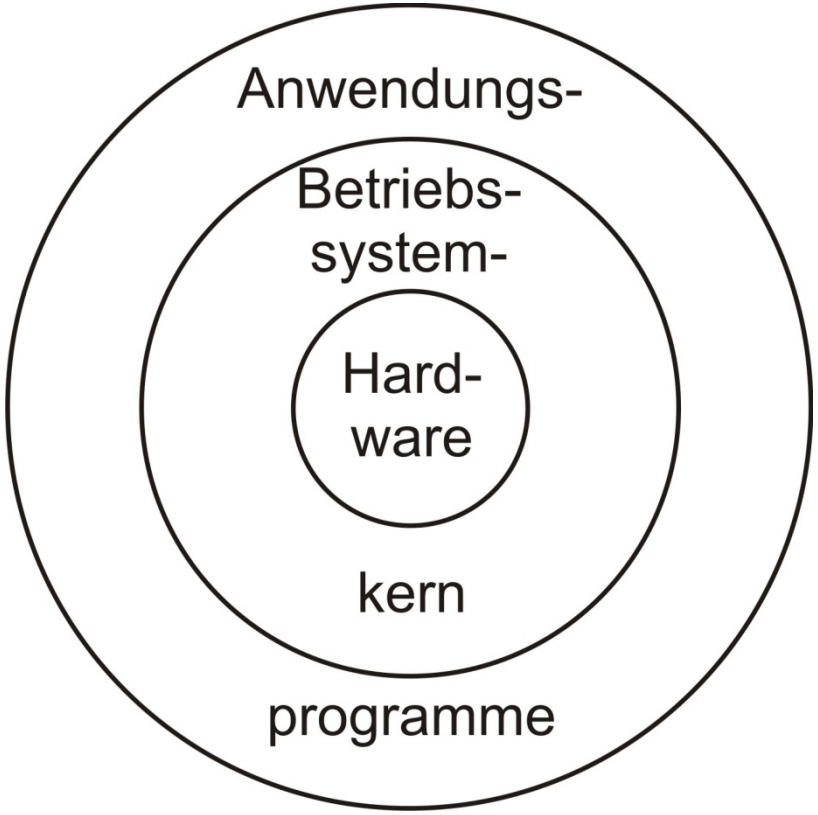


Abbildung 39: Schalen- bzw. Schichtenmodell eines Rechners

4.2.1. 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 Abbildung 9.

4.2.2. 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 5.3.2.8.

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

Beispiel einer einfachen Steuerungsaufgabe:

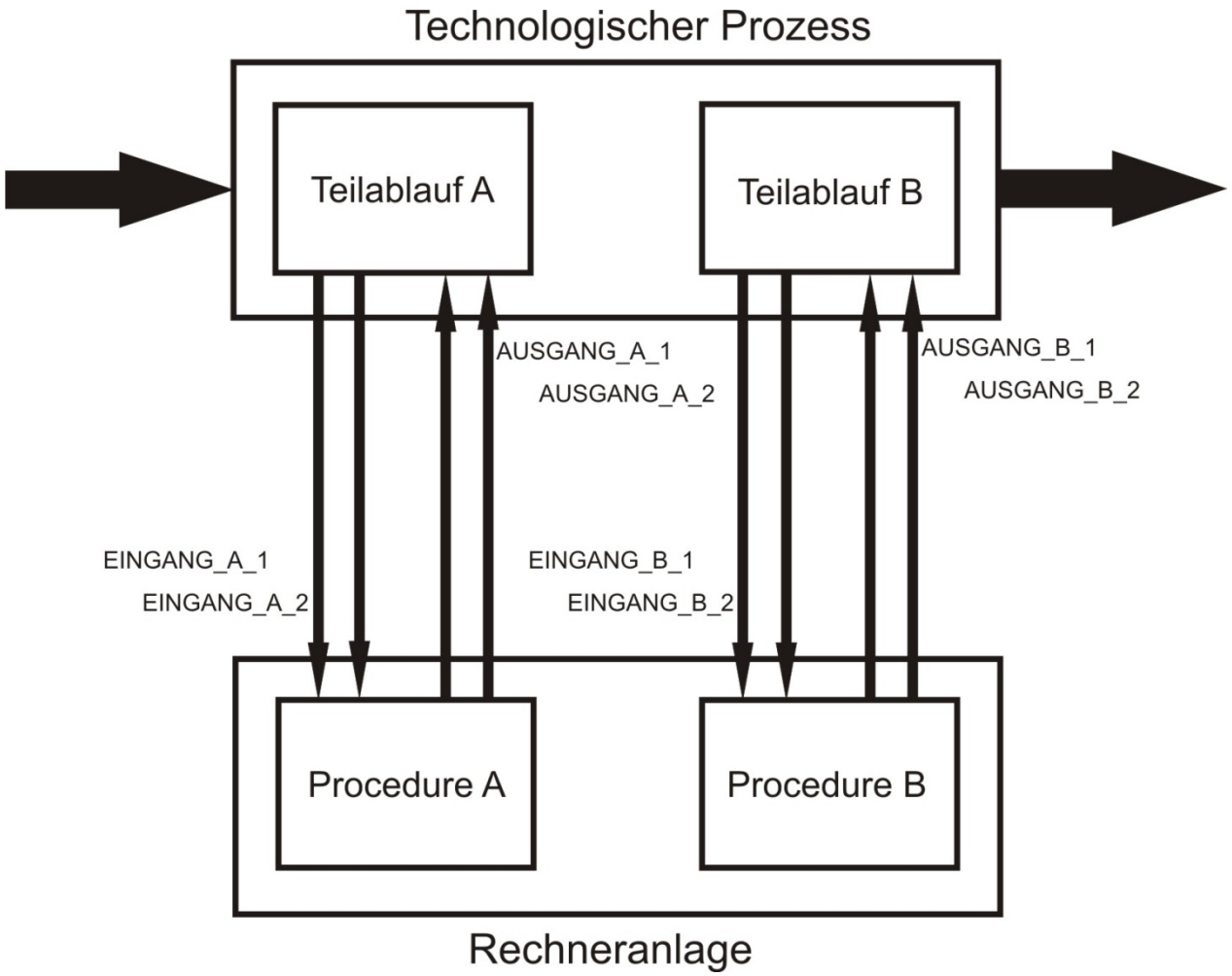


Abbildung 40: einfache Steuerungsaufgabe

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.

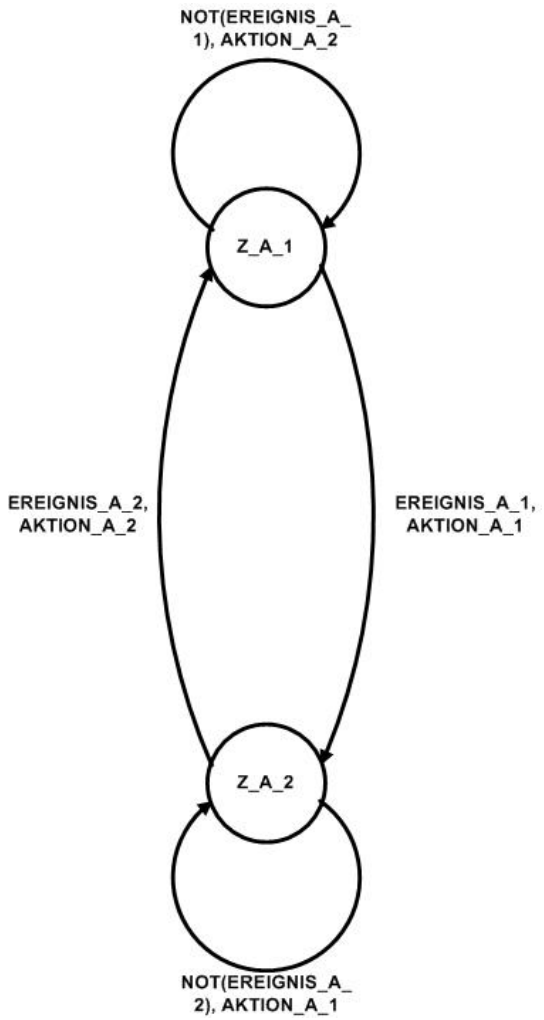
Prozedur A für Teilablauf A:

```
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 41 und Abbildung 42, mit je zwei Zuständen aufgefasst werden. Dabei hat jeder Zustand **Z_A_i** und **Z_B_i** jeweils eine wegführende Kante und eine Eigenschleife.

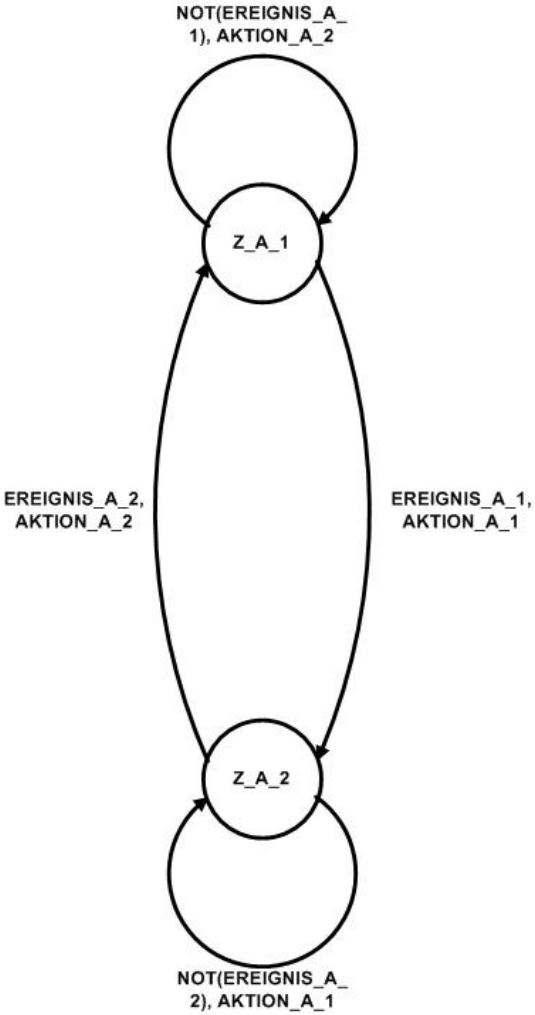


Abbildung 41: Automatengraph für Procedure A

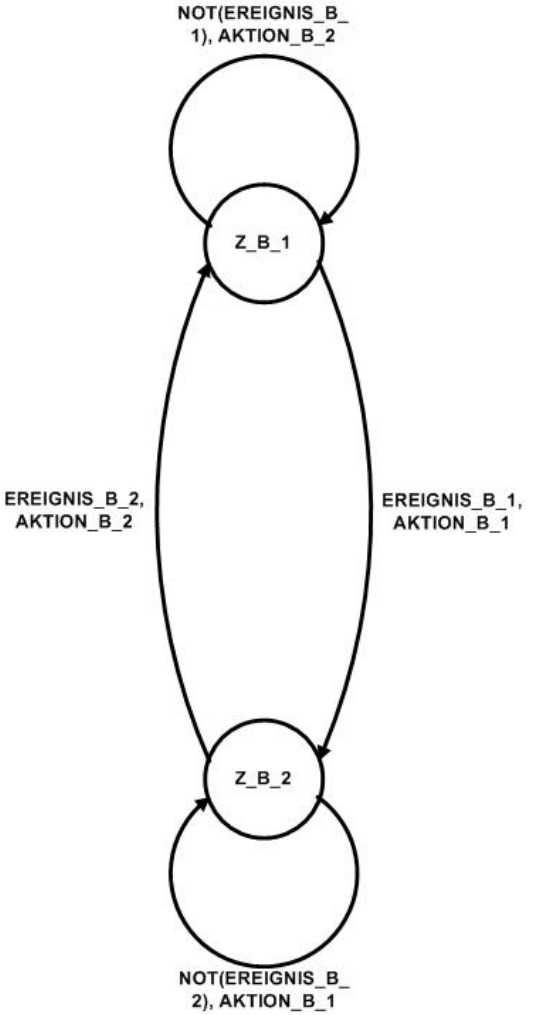
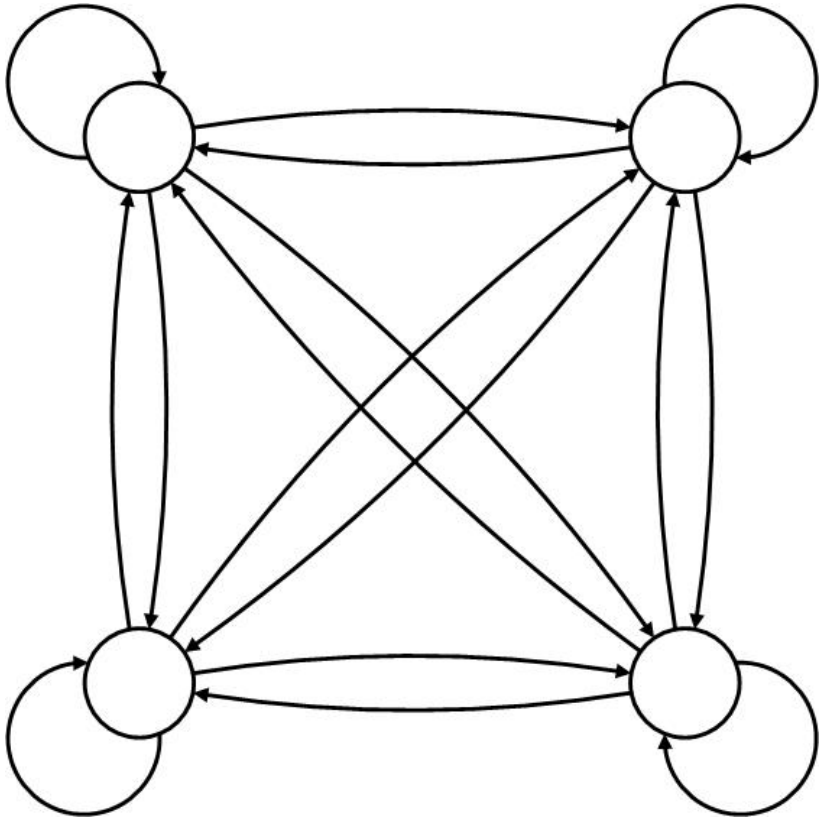


Abbildung 42: Automatengraph für Procedure B



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 43.

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 43: Automatengraph mit vier Zuständen

4.3. Arten von Betriebssystemen

Die ganze Geschichte und Entwicklung hat eine Vielzahl von unterschiedlichen Betriebssystemen hervorgebracht, die selten in ihrer Gesamtheit wahrgenommen wird. Überblicksweise sollen diese hier kurz vorgestellt und angesprochen werden. Auf einzelne Arten wird in nachfolgenden Abschnitten wieder eingegangen.

Diese Unterteilung ist nicht gleichbedeutend mit der Klassifizierung von Betriebssystemen, wie sie im Abschnitt 4.4. Klassifizierung von Betriebssystemen vorgenommen wird.

4.3.1. Mainframe-Betriebssysteme

Im Highend-Bereich aller Systeme sind Betriebssysteme für Mainframes. Diese raumgrossen Geräte sind auch heute noch in großen Rechenzentren von Firmen zu finden. Sie zeichnen sich durch eine sehr hohe Ein- / Ausgabebandbreite gegenüber allen anderen Systemen aus. Ausstattungen mit 1000 Festplatten und mehreren Terabyte sind in diesem Bereich nicht ungewöhnlich.

In der heutigen Zeit werden solche Systeme als große Webserver, Server für E-Commerce oder Server für Business-to-Business-Anwendungen eingesetzt.

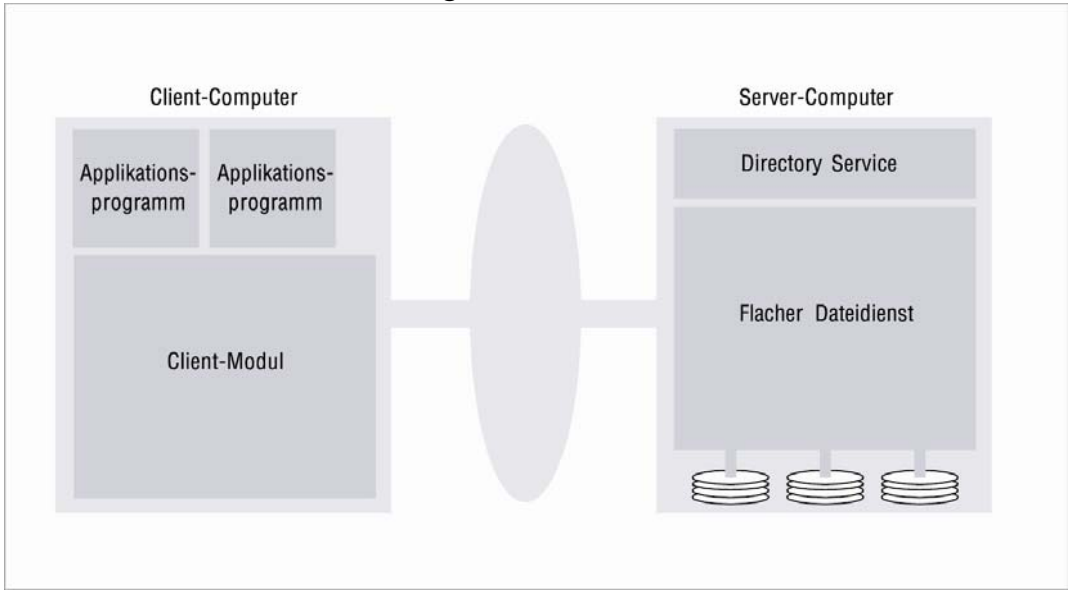
Das Betriebssystem muss eine Vielzahl von Prozessen mit einem hohen Bedarf an schneller Ein- / Ausgabe verwalten. Bezüglich der Prozessverwaltung sind alle Möglichkeiten in diesem Bereich zu finden:

- Batchbearbeitung,
- Transaktionssteuerung und
- Zeitaufteilungsverfahren

Als typischer Vertreter dieser Betriebssystemart ist das Mainframe-Betriebssystem **OS/390** der **Firma IBM** als Nachfolgesystem des bereits erwähnten **OS/360**.

4.3.2. Server-Betriebssysteme

Eine Ebene unter den Mainframe-Betriebssystemen sind Server-Betriebssysteme angesiedelt, die auf sehr großen Personalcomputern, Workstations oder sogar Mainframes laufen. Sie bedienen eine Vielzahl von Benutzern, die über das Netzwerk zur gleichen Zeit auf verteilte Hardware- und Softwareressourcen zugreifen. Dazu können sie Druckdienste, Dateifreigaben und Webdienste anbieten.



Typische Vertreter der Serverbetriebssysteme sind Unix, Linux und Windows-Server 2003.

Ein Beispiel für einen Dateiserver ist in Abbildung 44 dargestellt.

Abbildung 44: Beispiel für einen Dateiserver

4.3.3. Multiprozessor-Betriebssysteme

Eine neuere Entwicklung erhöht die Rechenleistung durch Zusammenschalten mehrer Prozessoren in einem oder mehreren Rechnern. In Abhängigkeit der Verschaltung untereinander nennt man diese Systeme:

- Parallelcomputer,
- Multicomputer oder
- Multiprozessorcomputer

Die neusten Entwicklungen gehen dabei in Richtung von

- Computernetzwerken bzw. –clustern oder
- Grid-Computing.

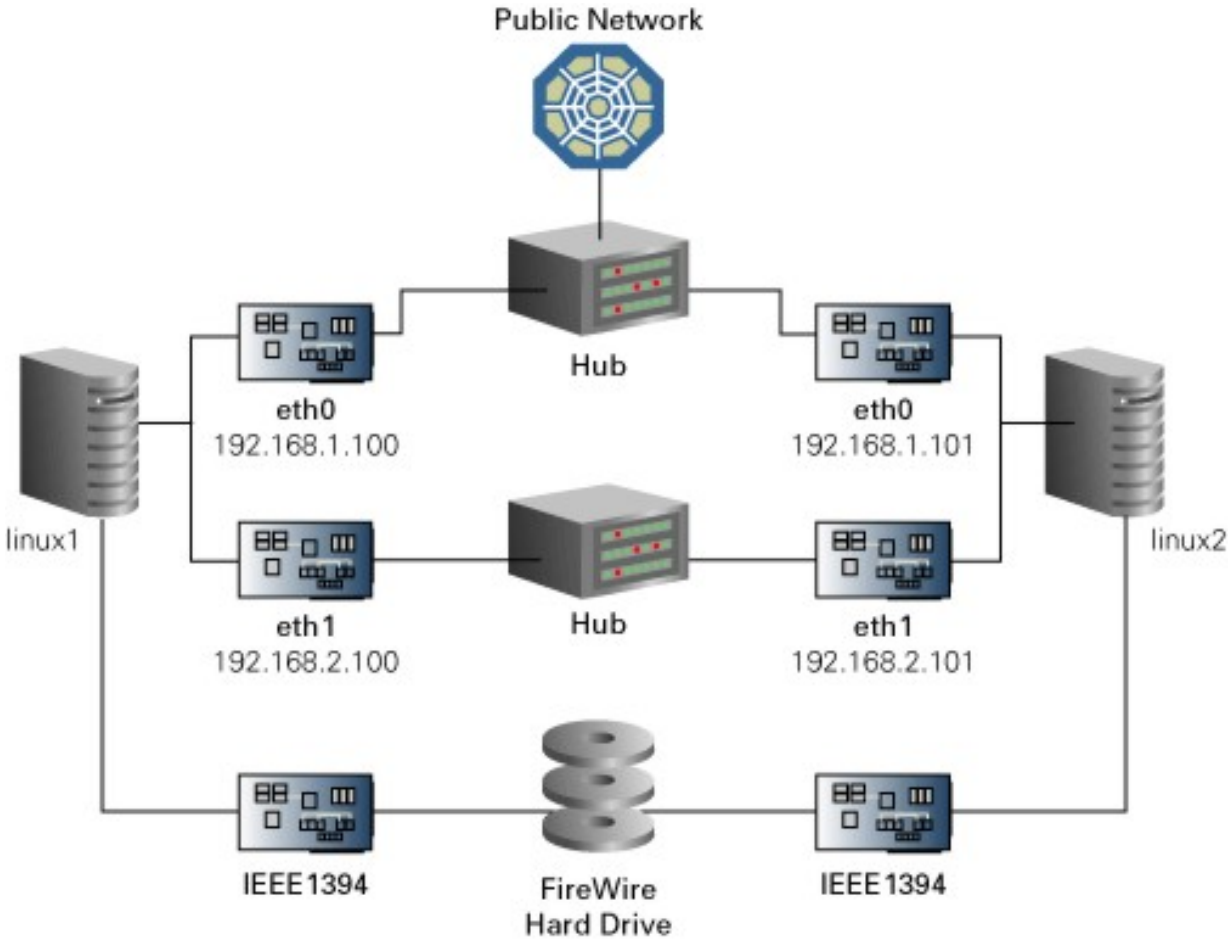


Abbildung 45: Beispiel für ein Linux-Cluster

Als Beispiele für diese Betriebssystemart sind die meisten Unix- oder Linuxdistributionen oder Windows-Server-systeme zu nennen.

4.3.4. Betriebssysteme für den Personalcomputer

Den kommerziell am meisten verbreiteten Bereich der Betriebssysteme stellen solche für den Personalcomputer dar. Mit ihnen können die Benutzer im professionellen und privaten Umfeld die verschiedensten Aufgaben von der Textbearbeitung, der Bearbeitung von Filmen, oder alle Aufgaben rund um das Internet bewältigen.

Die Verbreitung dieser Systeme ist so gigantisch, dass es wohl nur wenige Menschen gibt, die keinen Umgang mit solchen Systemen haben. Mache davon wissen teilweise überhaupt nicht, dass es noch andere Systeme gibt.

Als Beispiele für diese Betriebssystemart sind die Microsoftprodukte Windows, das Betriebssystem für den Macintosh der Firma Apple oder mit steigender Bedeutung das Linux-Betriebssystem zu nennen.

4.3.5. Echtzeitbetriebssysteme

Die Art der Echtzeitbetriebssysteme zeichnet sich dadurch aus, dass die Zeit ein sehr wichtiger Parameter bei der Ressourcenvergabe ist. Meistens gibt es sehr wichtige Zeitpunkte, die unbedingt eingehalten werden müssen. Einsatzgebiete sind in vielen Bereichen der industriellen Produktion, zum Beispiel der Fertigungs- oder der Robotersteuerung.

Aus der Vielzahl an möglichen und sehr spezifischen Systemen sind zu nennen:

- OS9 der Firma Microware Systems Corp.,
- LynxOS der Firma Lynx Real Time Systems oder
- QNX der Firma QNX Software Systems.

4.3.6. Betriebssysteme für eingebettete Systeme

Geht man in Richtung kleiner Systeme, dann kommt man zu den Palmtops bzw. **PDA** (**P**ersonal **D**igital **A**ssistant) und den eingebetteten Systemen. Diese Systeme übernehmen zum einen die Funktionen eines Betriebssystems für den Personalcomputer, zum anderen sind sie in Steuerungen anderer Systeme wie Auto, Waschmaschine, Telefonen und der gleichen mehr eingesetzt und haben Funktionen von Echtzeitbetriebssystemen.

Abbildung 46: Beispiele für PDAs

Die Entwicklung solcher Systeme ist durch die Eigenschaften geringe Größe, kleinen Arbeitsspeicher und geringen Stromverbrauch bestimmt.

Beispiele für diese Systeme sind Tungsten mit **PalmOS** links in Abbildung 46: Beispiele für PDAs oder Toshiba Pocket PC e800 mit **Windows Pocket 2003**.

4.3.7. Betriebssysteme für Chipkarten

Die kleinsten Betriebssysteme laufen auf Smart Cards, diese haben die Größe einer Kreditkarte und besitzen einen eigenen Prozessor. Dabei sind sehr harte Bedingungen an die Rechenleistung, den Stromverbrauch und die Robustheit zu stellen.

Viele dieser Systeme sind Java-orientiert und besitzen eine **JVM** (**J**ava **V**irtual **M**aschine). So genannte Java-Applets werden auf die Smart Card geladen und können durch die JVM sofort bearbeitet werden.

4.4. Klassifizierung von Betriebssystemen

Die Klassifizierung von Betriebssystemen gibt Auskunft darüber, wie bestimmte Funktionen in dem Betriebssystemkern realisiert werden bzw. werden müssen. Oder anders ausgedrückt, sind für bestimmte Klassifikationsmerkmale bestimmte Funktionen, die der Betriebssystemkern realisiert, unbedingt notwendig.

Diese angesprochenen Funktionen liegen vor allem in dem Prozesssystem des Betriebssystemkerns bzw. des betrachteten Betriebssystems.

Jedoch müssen alle Funktionsgruppen des Betriebssystemkernes, unabhängig von der Klassifikation und dem Funktionsumfang, enthalten sein.

4.4.1. Klassifikation nach dem Anwendungsgebiet

4.4.1.1. Betriebssysteme für allgemeine Anwendungen

- Programmentwicklung,
- betriebswirtschaftliche Aufgaben,

- wissenschaftlich-technische Aufgaben,
- Multimediaanwendungen,
- **CAD** - **C**omputer **A**ided **D**esign u.s.w.

4.4.1.2. Echtzeitbetriebssysteme

- Prozess-Steuerungen,
- Labor- und Geräteautomation,
- **CAM** - **C**omputer **A**ided **M**anufacturing

Wesentliches Kriterium für ein Echtzeitbetriebssystem ist, dass eine minimale (möglichst kleine) Reaktionszeit angebbar ist, nach der auf ein äußeres Ereignis maximal reagiert werden kann. In Abschnitt 5. Echtzeitbetriebssysteme wird aus diese Art von Betriebssystemen genauer eingegangen.

Prozess	**Antwortzeiten**
Maschinenreglung	1 … 10 ms
Prozessreglung	10 … 100 ms
Prozessführung	0,1 … 1 s
Prozessüberwachung / Auskunftssysteme	1 … 10 s

Tabelle 5: Echtzeitbedingungen

4.4.2. Klassifikation nach der vorhandenen Anzahl von Prozessoren

Wie bereits eingeführt, hat John von Neumann einen Vorschlag für den Aufbau von Digitalrechnern vorgelegt, nach dessen Vorbild bisher fast alle Rechner aufgebaut sind. Von Neumann geht dabei von einer Verarbeitungseinheit

aus, da es zur Zeit der Entwicklung seiner Idee noch nicht die Möglichkeiten gab, mehrere Verarbeitungseinheiten in einem Rechner zusammenzufassen.

In heutigen Systemen werden für fast alle Geräteansteuerungen eigene Prozessoren verwendet. So z.B. für

- Grafikprozessoren für CAD-Anwendungen,
- Prozessoren in den Ansteuerkarten für Festplatten, CD-ROM-Laufwerke usw.
- Prozessoren für den Netzzugang usw.

Bei der genannten Klassifizierungsart geht es jedoch nicht darum, wie viel Prozessoren allgemein in einem Rechner verwendet werden, sondern wie viel **Universalprozessoren** für die Verarbeitung der Daten zur Verfügung stehen. Somit ergibt sich folgende Unterscheidung:

4.4.2.1. Ein-Prozessor-Betriebssystem

Die meisten Rechner, die auf der von-Neumann-Architektur aufgebaut sind, verfügen über **nur** einen Universalprozessor. Aus diesem Grund unterstützen auch die meisten Betriebssysteme für diesen Anwendungsbereich nur einen Prozessor.

Beispiele:

- **Microsoft-DOS für Intel 80x86,**
- **die meisten Unix-Versionen für verschiedene Prozessoren,**
- **Linux für verschiedene Prozessoren,**
- **OS/2 für Intel 80x86,**
- **Microsoft-Windows 95/98/ME für Intel 80x86 und**
- **Microsoft-Windows NT/2000/XP für Intel 80x86**

4.4.2.2. Mehr-Prozessor-Betriebssystem

Für diese Klassifizierung der Betriebssysteme, entsprechend der Rechnerhardware, ist noch keine Aussage über die Kopplung der einzelnen Prozessoren getroffen worden. Auch gibt es keinen quantitativen Hinweis über die Anzahl der Prozessoren, nur, dass es mehr als ein Prozessor ist.

Mehrprozessorsysteme werden für spezielle Anwendungsgebiete geschaffen:

Beispiele.

- **Datenbankserver in Informationssystemen,**
- **Hochleistungsrechner für Applikationen mit hohem Datenvorkommen und hohem Rechenaufwand usw.**

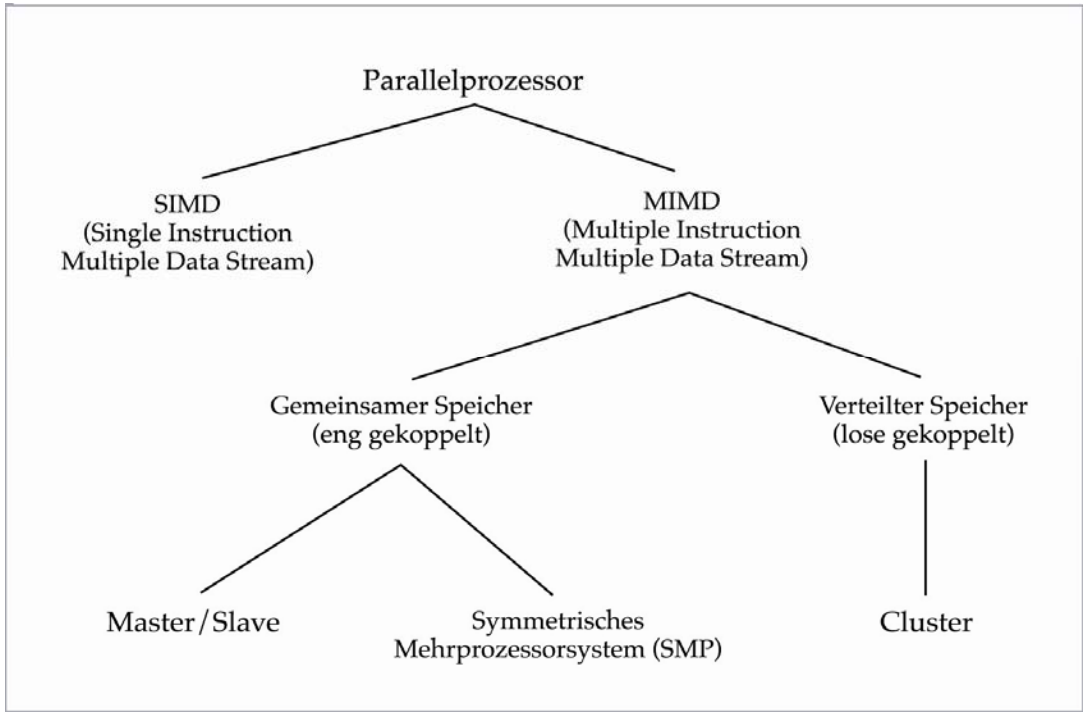


Abbildung 47: verschiedene Multiprozessorstrukturen

Für die Realisierung der Betriebssysteme für die Mehrprozessorsysteme gibt es zwei Herangehensweisen:

1. Jedem Prozessor wird durch das Betriebssystem eine eigene Aufgabe zugeteilt. D.h., es können zu jedem Zeitpunkt nur soviel Aufgaben bearbeitet werden, wie Prozessoren zur Verfügung stehen. Somit entstehen Koordinierungsprobleme, wenn die Anzahl der Aufgaben nicht gleich der Anzahl verfügbarer Prozessoren ist.

2. Jede Aufgabe kann prinzipiell jedem Prozessor zugeordnet werden. D.h., die Verteilung der Aufgaben zu den Prozessoren ist nicht an die Bedingung gebunden, dass die Anzahl der Aufgaben gleich der Anzahl Prozessoren ist. Sind mehr Aufgaben zu bearbeiten, als Prozessoren vorhanden sind, so bearbeitet ein Prozessor mehrere Ausgaben **quasi-parallel**. Sind mehr Prozessoren als Aufgaben vorhanden, dann bearbeiten mehrere Prozessoren eine Aufgabe. Das Betriebssystem kann dabei seinerseits auch auf mehrere Prozessoren verteilt sein. Man spricht dann von **verteilten Betriebssystemen**.

4.4.3. Klassifikation nach der Anzahl parallel ablaufender Programme

Für einen Anwender, der mittels Betriebssystem seine Aufgaben bearbeitet haben möchte, ist von großem Interesse, wie viel Aufgaben kann er parallel bearbeiten bzw. bearbeiten lassen. Dabei spielt die Klassifikation nach 1.2.2. keine Rolle.

In dieser Klassifikation kommt der Begriff **Task** vor. Alternativ kann der deutsche Begriff **Prozess** Verwendung finden. Eine genauere Definition folgt später. Aus Anwendersicht kann an dieser Stelle auch der Begriff **Aufgabe** bzw. **Auftrag** verwendet werden.

Man unterscheidet:

4.4.3.1. Single-Task-Betriebssystem

Auch wenn für den Anwender die Anzahl vorhandener Prozessoren bei dieser Klassifikation keine Rolle spielt, sind typische **Single-Task-Betriebssysteme** auch gleichzeitig **Ein-Prozessor-Betriebssysteme**.

Beispiel:
MS-DOS für INTEL 80x86

Kennzeichnend ist, dass der Anwender immer nur eine Anwendung starten kann. Erst wenn er diese beendet hat, kann die nächste Aufgabe bearbeitet werden.

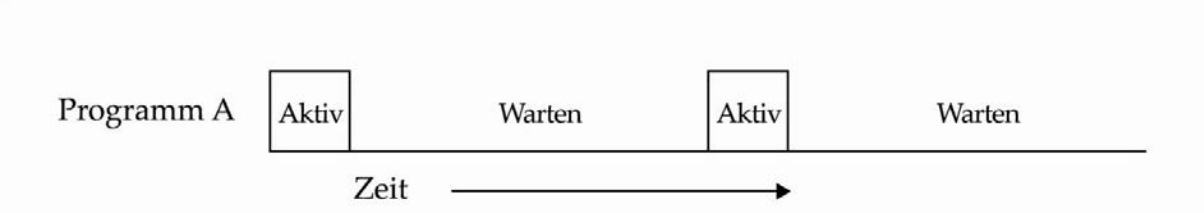


Abbildung 48: Single-Task-Betrieb

4.4.3.2. Multi-Task-Betriebssystem

Bei **Multi-Task-Betriebssystemen** kann der Anwender zu jedem beliebigen Zeitpunkt mehrere Aufgaben durch das Betriebssystem bearbeiten lassen. Dabei wird im Allgemeinen nur eine Aufgabe direkt zu jedem Zeitpunkt **aktiv** bearbeitet.

Die anderen Aufgaben oder Aufträge können nach anfänglicher Eingabe von Parametern u.ä. durch das Betriebssystem selbständig bearbeitet werden. Beziehungsweise der Anwender führt gerade keinen **Dialog** mit den Anwendungen.

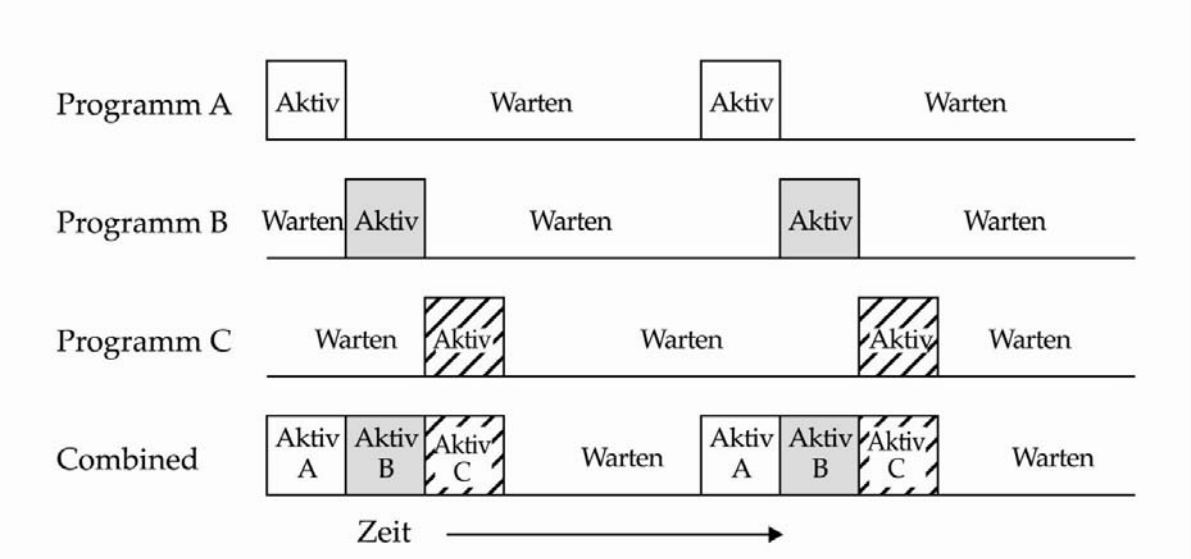


Abbildung 49: Multi-Task-Betrieb

Beispiele:

- **Unix-Versionen für verschiedene Prozessoren,**
- **Linux für verschiedene Prozessoren,**
- **OS/2 für Intel 80x86,**
- **Microsoft-Windows 95/98/ME für Intel 80x86 und**
- **Microsoft-Windows NT/2000/XP für Intel 80x86**

4.4.4. Klassifikation nach der Anzahl gleichzeitig aktiver Nutzer

Wenn man die geschichtliche Entwicklung der Rechentechnik betrachtet, waren die verfügbaren Rechner in den Anfängen recht unhandlich und teuer. Deshalb konnte man es sich nicht leisten, einen Rechner für einen Anwender anzuschaffen. Mit der Einführung des **P**ersonal **C**omputers durch die Firma **IBM 1984** änderte sich das schlagartig. Die Entwicklung war soweit fortgeschritten, dass dem einzelnen Mitarbeiter ein persönlich zugeordneter Rechner angeschafft werden konnte.

Mit der zunehmenden Vernetzung und Verflechtung der Rechner in einem Unternehmen und darüber hinaus weltweit, machte sich wieder ein umgekehrter Trend bemerkbar, dass moderne Betriebssysteme wieder die Nutzung durch mehrere Anwender zulassen.

Man unterscheidet:

4.4.4.1. Single-User-Betriebssystem

Entsprechend der Zielsetzung und des Einsatzgebietes dieser Art von Betriebssystemen bzw. denen der Rechner, kann zu einem beliebigen Zeitpunkt nur ein Anwender mit dem Betriebssystem arbeiten und Aufgaben lösen.

Dabei kann nicht ausgeschlossen werden, dass zu einem anderen Zeitpunkt ein anderer Anwender mit dem gleichen System arbeitet. **Single-User-Betriebssysteme** zeichnen sich durch einen geringen oder keinen Schutz des Systems vor dem Zugriff anderer Anwender oder Personen aus.

Der Benutzer zu einem Zeitpunkt hat Zugriff auf alle Ressourcen des Systems.

Beispiel:

- **MS-DOS für Intel 80x86,**
- **Microsoft-Windows 95/98/ME für Intel 80x86 und**
- **Microsoft-Windows NT/2000/XP für Intel 80x86.**

4.4.4.2. Multi-User-Betriebssystem

Arbeiten mehrere Anwender gleichzeitig an einem Rechner, so sind Leistungen durch das Betriebssystem zu erbringen, die in einem **Single-User-Betriebssystem** nicht oder nur in geringem Umfang vorhanden sein müssen:

1. Die Leistungen des Rechners sind allen Anwendern gerecht zur Verfügung zu stellen.
2. Der Anwender muss annehmen, dass er ganz allein an dem Rechner arbeitet.
3. Es sind besondere Schutzmechanismen zum Schutz der Daten der einzelnen Anwender notwendig. Dabei sind folgende Möglichkeiten getrennt zu realisieren:

   - Schutz **gegen den Rest der Welt**
   - Schutz untereinander.

Beispiele:

- **Unix für verschiedene Prozessoren und**
- **Linux für verschiedene Prozessoren.**

4.5. Betriebsmittel

Definition 16: Betriebsmittel

Als Betriebsmittel werden alle realen physikalischen Einheiten des Rechnersystems (Gräte, Speicher, Prozessoren) und weiterhin alle bei der Zusammenarbeit von Programmen entstehenden logischen Objekte (Datenbereiche) bezeichnet, die für die Nutzung durch Programme von Bedeutung sind.

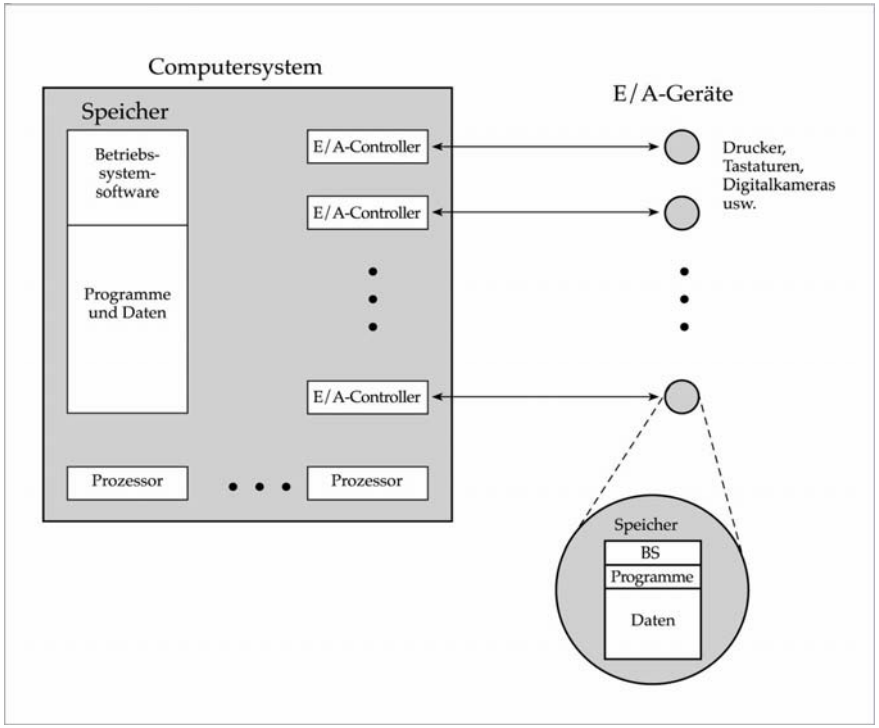


Abbildung 50: Betriebsmittelverwaltung

Damit ist ein Betriebssystem eine Sammlung von Programmen zur geregelten Verwaltung und Benutzung von Betriebsmitteln.

Es ergeben sich zwei fundamentale Aufgabenbereiche für Betriebssysteme:

1. Zuordnung von Betriebsmitteln: In der Regel gibt es mehr Interessenten an Betriebsmitteln als es Betriebsmittel gibt. **Wettbewerb um die Betriebsmittel → Belegung** und **Freigabe** von **Betriebsmitteln**
2. Betrieb von Betriebsmitteln **sichere Steuerung und gute Ausnutzung von Betriebsmitteln**

Folgende Aufteilung der Betriebsmittel ist denkbar:

4.5.1. Aktive Betriebsmittel, zeitlich aufteilbar

Dies sind die **programmierbaren Logiken (CPU)**. Sie haben als einzige Einheit des Rechnersystems eine **Sonderstellung**, da sie in der Lage sind, Programme auszuführen.

4.5.2. Passive Betriebsmittel, exklusiv benutzbar

Darunter sind Einzelgeräte, wie Drucker, Terminals, Bandgeräte, usw. zu verstehen, die für einen bestimmten Zeitraum **nur einem Interessenten zugeordnet** sind.

4.5.3. Passive Betriebsmittel, räumlich aufteilbar

Dies sind **interne und externe Speicher** (bestimmte Bereiche sind verschiedenen Interessenten als Datenspeicher zugeordnet)

1. exklusive Nutzung
2. mehrfache Nutzung

a) Programmcode von Prozeduren
b) Datenbereich mit NUR-Lesen der Daten

Eine weitere Aufteilung der passiven Betriebsmittel:

1. wieder verwendbare Betriebsmittel (Geräte, physikalische Speicher usw.)
2. zum „Verbrauch" bestimmte Betriebsmittel (Dateninhalte)

5. Echtzeitbetriebssysteme

Bereits in den Abschnitten 1. Einführung, 4.3.5. Echtzeitbetriebssysteme oder 4.4.1.2. Echtzeitbetriebssysteme wurde auf grundlegende Eigenschaften von Echtzeitbetriebssystemen eingegangen. Bezüglich der Aufgaben muss ein Echtzeitbetriebssystem dieselben Aufgaben erfüllen, wie ein Standardbetriebssystem, siehe Abschnitt 5.3.1. Grundlegende Funktionen des Betriebssystemkerns. Diese Aufgaben sind bei Echtzeitbetriebssystemen genau wie bei Standardbetriebssystemen je nach Typ mehr oder minder ausgeprägt. Neben den klassischen Betriebssystemaufgaben haben Echtzeitbetriebssysteme zwei wesentliche zusätzliche Aufgaben, die

- Wahrung der Rechtzeitigkeit und Gleichzeitigkeit, und die
- Wahrung der Verfügbarkeit.

Die Grenzen zwischen reinen Echtzeitsystemen und Informationssystemen ohne zeitliche Randbedingungen sind in der Regel fließend, siehe Tabelle 6. 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	**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 6: Systemvergleich konventionell - Echtzeit

Die wichtigste Definition ist die allgemeine Definition von Echtzeitsystemen.

Definition 17: Echtzeitsysteme

Echtzeitsysteme erlauben die Verarbeitung von Daten innerhalb festgelegter und reproduzierbarer zeitlicher Toleranzen. Das bedeutet, ein Echtzeitsystem arbeitet in seinem zeitlichen Verhalten in vorherbestimmten Grenzen, die meist durch das technische System gegeben sind, deterministisch.

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.

5.1. Grundlagen

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

Definition 18: Echtzeitdatenverarbeitung
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.

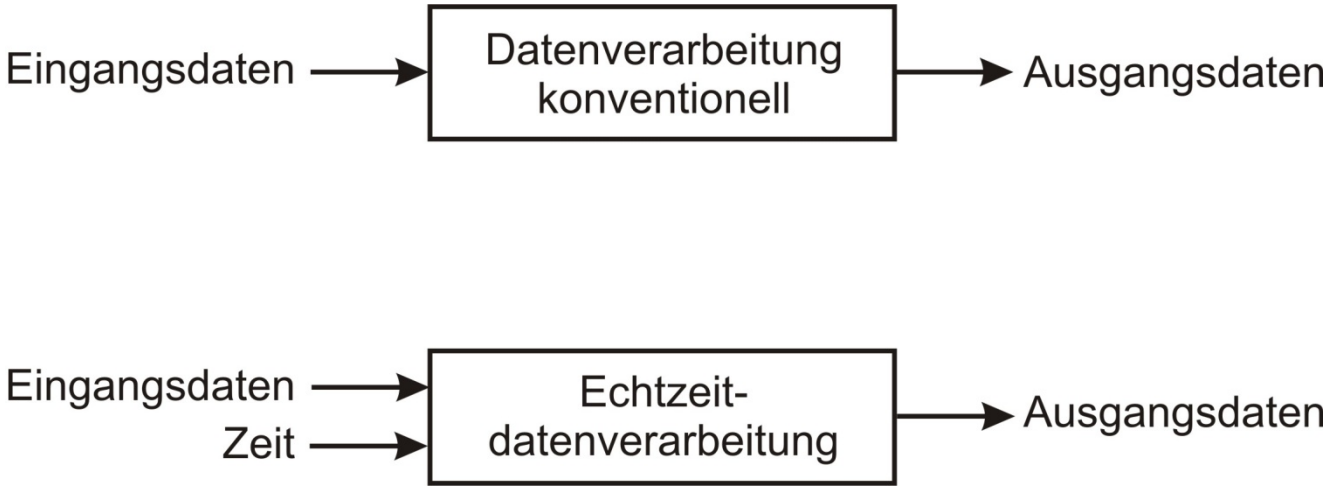


Abbildung 51: Zeit als zusätzliche Eingangsgröße

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.

5.1.1. 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.

5.1.2. 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.

5.1.3. 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 19: 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:

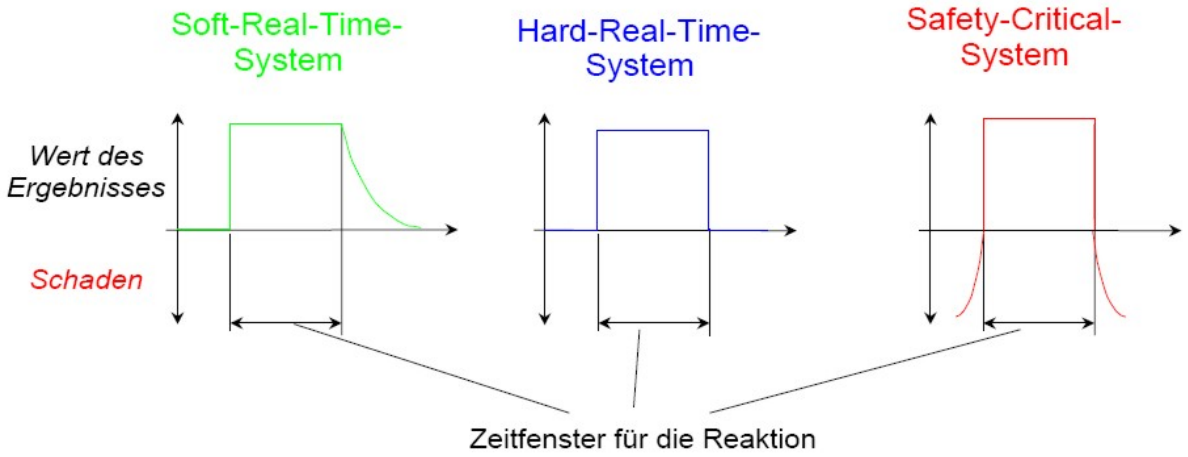
5.1.3.1. harte Echtzeitsysteme

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.

5.1.3.2. 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.



Klasse	Soft-Real-Time-Systeme	Hard-Real-Time-Systeme	Safety-Critical-Systeme
Auswirkungen von Zeitfehlern	- höhere Kosten (mindern Wert des Ergebnisses)	machen Ergebnis wertlos	führen zur Zerstörung Gefahr für Menschen
Entwurfskriterium	mittlerer Durchsatz	worst-case-Parameter	worst-case-Parameter + Zuverlässigkeit
Beispiele	Dialogsysteme Flugreservierung, Betriebsdatenerfassung	Datenerfassung Fertigungssteuerung	Kernkraftwerk-Steuerung Steuerung eines Raumschiffes

Abbildung 52: Klassen von Echtzeitsystemen

5.2. 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 7: Standards für Echtzeitsysteme

5.3. Der Betriebssystemkern

5.3.1. Grundlegende Funktionen des Betriebssystemkerns

Definition 20: Betriebssystemkern

Unter einem Betriebssystemkern versteht man eine Sammlung von Programmen, die wohldefinierte Funktionen über eine definierte Schnittstelle zur Verfügung stellen. Sie greifen steuernd auf die Hardware des Rechnersystems zurück.

Für alle Typen von Betriebssystemen hat der Betriebssystemkern gleiche Funktionen, die sich jedoch erheblich im Umfang der Realisierung der einzelnen Funktionen in Abhängigkeit von dem Einsatzgebiet des Betriebssystems unterscheiden.

Folgende vier Funktionen bzw. Funktionsgruppen werden unterschieden:

- Prozesssystem,
- E/A-System,
- Dateisystem und
- Speicherverwaltungssystem.

5.3.2. Das Prozesssystem

5.3.2.1. Programme, Prozeduren, Prozesse und Instanzen

Definition 21: Programm

Die Lösung einer Programmieraufgabe (=Algorithmus) wird in Form eines Programms realisiert. Teillösungen werden dabei als Prozeduren (Unterprogramme) formuliert, welche nach Beendigung ihrer Arbeit zum aufrufenden übergeordneten Programm zurückkehren.

Damit die Leistungen des Betriebssystemkerns problemlos in Anwenderlösungen eingebunden werden können, sind sie ebenfalls als Prozeduren realisiert.

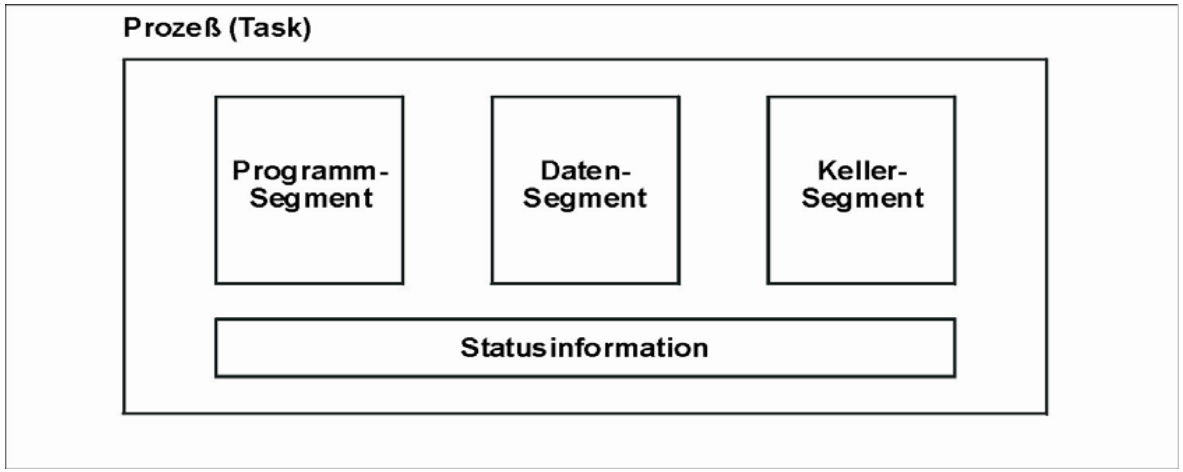
Ein Programm (Prozedur, Unterprogramm) besteht aus

1. Befehlen (**Codebereich**, Textbereich)
2. Programmdaten (**Datenbereich**)

beide Komponenten sind **problemorientiert**.

Definition 22: Prozess
Wird ein Programm (Prozedur) unter der Kontrolle eines Betriebssystems (genauer gesagt unter der Kontrolle eines Betriebssystemkerns) ausgeführt, so wird dieser Ablauf als Prozess (engl. Task) bezeichnet.

Diese Betrachtungsweise macht es möglich, dass mehrere Programme gleichzeitig als Prozesse **parallel** auf einem **sequentiell arbeitenden Rechnersystem** (unabhängig von der realen Anzahl Prozessoren) ablaufen können.



Als Sonderfall gilt die Ausführung mehrerer Prozesse auf einem Prozessor. (ACHTUNG: Die Funktionalität des Betriebssystemkerns bezüglich der Verwaltung von Prozessen ist bei der Ausführung mehrerer Prozesse **n** auf einem Prozessor **1** äquivalent der Verwaltung auf mehreren Prozessoren **m**. Dabei kann das Verhältnis **m : n** sowohl statisch als auch dynamisch änderbar sein.)

Abbildung 53: Komponenten eines Prozesses

Bei der Ausführung von Prozessen entstehen Daten, die durch den Betriebssystemkern verwaltet werden. Diese werden **Statusinformationen** genannt und sind **systemabhängig**.

Als weitere Komponente wird beim Ablauf eines Programms ein Kellerspeicher (**Stack**) aufgebaut. Somit lässt sich ein Prozess modellhaft wie in Abbildung 53. darstellen.

In Tabelle 8 sind typische Informationen enthalten, die typischerweise in den Statusinformationen, in einer Prozesstabelle hinterlegt sind.

Prozessmanagement	**Speichermanagement**	**Dateiverwaltung**
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Kellersegment	Dateideskriptor
Kellerzeiger		Benutzer-ID
Prozesszustand		Gruppen-ID
Priorität		
Scheduling-Parameter		
Prozess-ID		
Elternprozess		
Prozessfamilie		
Signale		
Startzeit des Prozesses		
Benutzte CPU-Zeit		
CPU-Zeit des Kindes		
Zeitpunkt des nächsten Alarms		

Tabelle 8: Prozesstabelle

Alle vier Komponenten, die bei der Ausführung eines Programms (einer Prozedur) beteiligt sind, werden als **Instanz** zusammengefasst.

Definition 23: Instanz
Eine Instanz umfasst das Tupel (P, D, K, S)
P: Programmsegment → problemorientiert
D: Datensegment → problemorientiert
K: Kellersegment → system- / problemorientiert
S: Statusinformationen → systemorientiert

Die physische Anordnung dieser Komponenten im Arbeitsspeicher eines Rechners kann in unterschiedlichen Betriebssystemen verschieden sein.

In Abbildung 54 ist die Problematik der Zuordnung von mehreren Prozessen zu einem Prozessor dargestellt.

Der Betriebssystemkern steuert die Prozesse des Rechnersystems. Er realisiert das Ablaufen der Prozesse. Bei Single-Prozessor-Systemen bringt der Betriebssystemkern die einzelnen Prozesse so zum Ablaufen auf der CPU, dass der Eindruck entsteht, die Prozesse liefen quasi parallel ab. Innerhalb von Multi-Prozessor-Systemen verteilt der Betriebssystemkern die Prozesse auf die einzelnen Prozessoren.

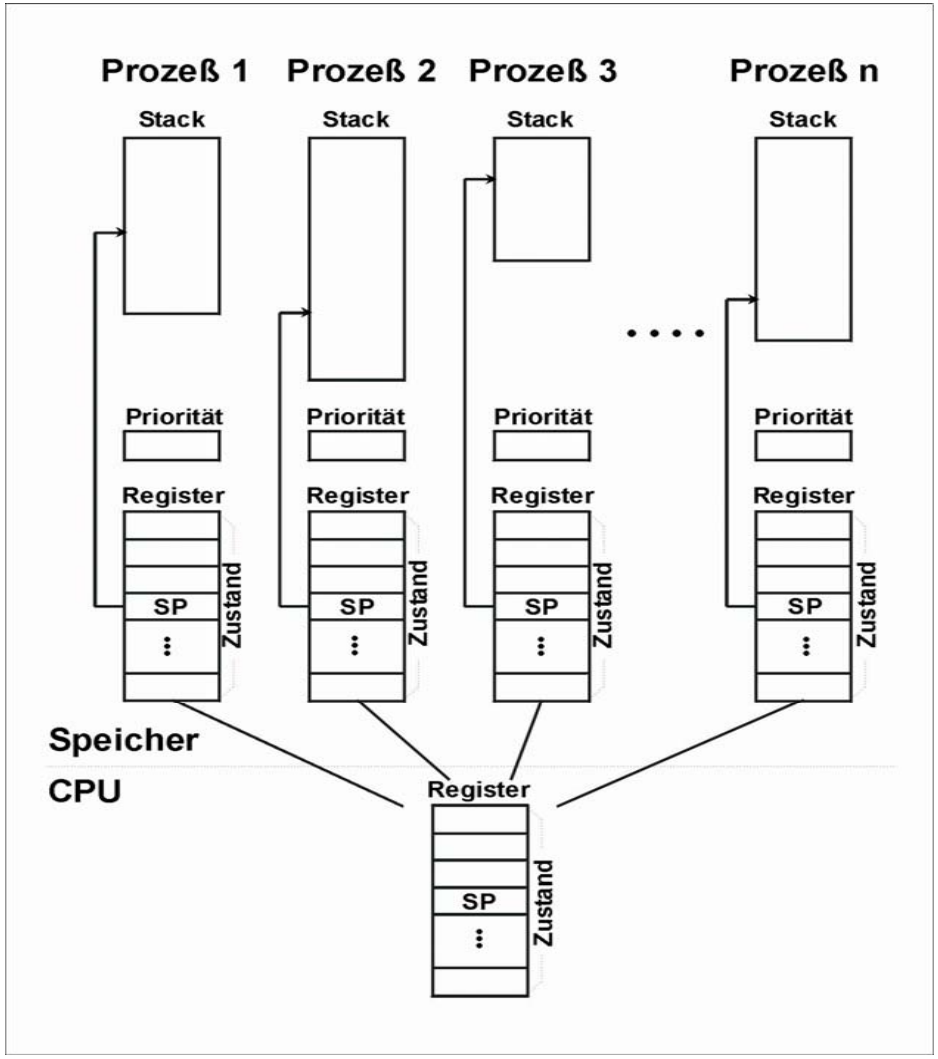


Abbildung 54: Verwaltung von n Prozessen auf 1 Prozessor

5.3.2.2. Prozesserzeugung

Betriebssysteme benötigen ein Verfahren, um sicherstellen zu können, dass alle notwendigen Prozesse existieren. Prinzipiell gibt es dazu zwei Vorgehensweisen:

1. Alle Prozesse werden beim Systemstart erzeugt und
2. Es existieren Verfahren, die Prozesse im laufenden System je nach Bedarf zu erzeugen und zu beenden.

Dabei ist von Interesse, welche Ereignisse zur Erzeugung eines Prozesses dienen können:

1. Initialisierung des Systems
2. Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
3. Benutzeranfrage, einen neuen Prozess zu erzeugen
4. Initiierung einer Stapelverarbeitung (**Batch-Job**)

Beim Starten des Betriebssystems wird in der Regel eine Reihe von Prozessen bereits gestartet. Nach ihrer Arbeitsweise unterscheidet man zwischen:

1. Vordergrundprozesse – Prozesse, die im Dialog zum Anwender stehen und Aufgaben für diese erledigen.

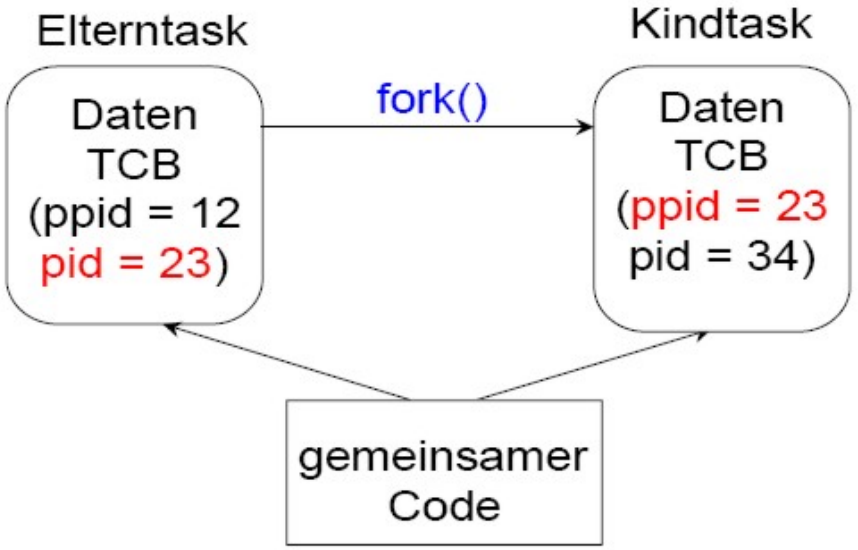


Abbildung 55: Erzeugen von Prozessen (Klonen)

2. Hintergrundprozesse – Prozesse für bestimmte Aufgaben, z.B. Verwalten der Warteschlange für den Drucker, die keine Dialogeingabe seitens eines Anwenders benötigen und die meiste Zeit nichts tun. Diese Prozesse werden auch als **Dämons** bezeichnet.

In interaktiven Systemen können Benutzer Programme durch Eingabe eines Kommandos oder das (Doppel-) Klicken eines **Icons** starten. Jede dieser Aktionen startet einen neuen Prozess und lässt das gewählte Programm darin ablaufen.

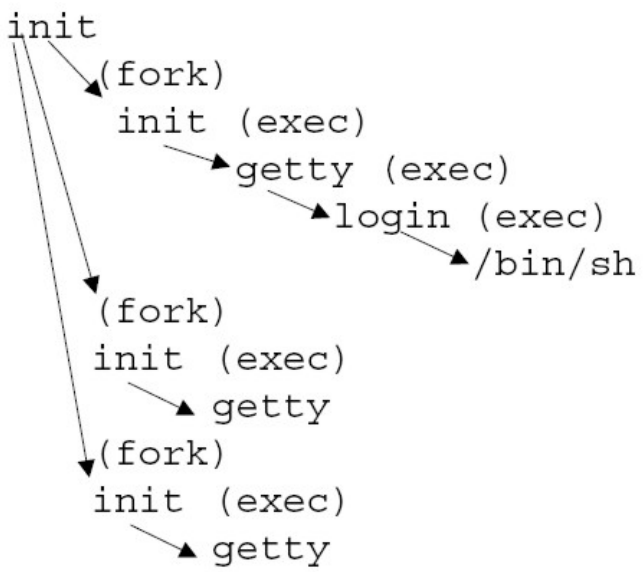
Technisch gesehen wird in jedem Fall ein neuer Prozess erzeugt, indem ein bestehender Prozess einen Systemaufruf zur Erzeugung eines neuen Prozesses ausführt, siehe Abbildung 55 und Abbildung 56.

```
#define TRUE 1

while (TRUE) {                                  /* Endlosschleife */
      type_prompt( )                            /* Prompt ausgeben */
      read_command(command,parameters);         /* Befehl einlesen */

      if (fork () != 0)                         /* Kindprozess erzeugen */
          /* Elternprozess */
          waitpid(–1, &status, 0)               /* Auf Ende von Kind warten */
      } else {
          /* Kindprozess */
          execve(command,parameters,0)          /* Befehl ausführen */
      }
}
```

Abbildung 56: Erzeugung neuer Prozesse



In **Unix** existiert beispielsweise der Systemcall zur Erzeugung eines neuen Prozesses **fork**. Dieser Aufruf erzeugt eine exakte Kopie des aufrufenden Prozesses. Nach dem **fork** haben die beiden Prozesse, also Vater und Kind (oder Sohn), das gleiche Speicherabbild, die gleichen Umgebungsvariablen und die gleichen geöffneten Dateien. Üblicherweise führt der Kindprozess anschließend **execve** oder einen ähnlichen Systemcall aus, um sein Speicherabbild zu wechseln und ein neues Programm abzuarbeiten.

Abbildung 57: Prozesse in Unix

Im Gegensatz dazu wickelt unter **Windows** ein einziger Win32-Funktionsaufruf, nämlich **CreateProcess**, sowohl die Erzeugung des Prozesses als auch das Laden des richtigen Programms in den Prozess ab, wozu dieser Funktionsaufruf eine Reihe von Parametern erhält.

Sowohl in **Unix** als auch in **Windows** haben Vater und Kind nach der Prozesserzeugung je einen eigenen getrennten Adressraum. Wenn einer von beiden ein Speicherwort im Adressraum ändert, so ist diese Änderung für den anderen Prozess nicht sichtbar.

5.3.2.3. Prozesssynchronisation

Prozesse, z.B. durch den Systemcall fork unter Unix gestartet, werden laufen in der Regel synchron ab. D. h., es findet eine Synchronisation zwische ihnen statt.

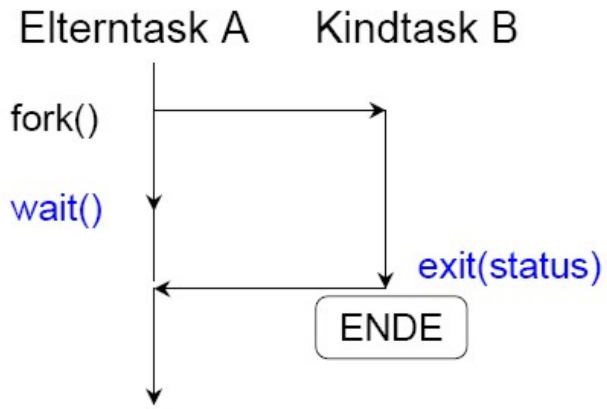


Abbildung 58: Prozesssynchronisation

5.3.2.4. Prozessbeendigung

Jeder der einmal erzeugten Prozesse wird früher oder später terminiert. Dies wird üblicherweise aufgrund einer der folgenden Bedingungen erfolgen:

1. Normales Beenden (freiwillig)
2. Beenden aufgrund eines Fehlers (freiwillig)
3. Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
4. Beenden durch einen anderen Prozess (unfreiwillig)

Die meisten Prozesse terminieren, nachdem sie ihre Aufgabe erledigt haben. Dazu wird ein Systemcall ausgeführt, der dem Betriebssystem mitteilt, dass die Arbeit erledigt ist. Diese Systemcall lautet unter **Unix exit** und unter **Windows exitProzess**.

Es ist denkbar, dass durch die Terminierung eines Prozesses gleichzeitig alle von diesem Prozess erzeugten Kindprozesse sofort terminiert werden.

5.3.2.5. Prozesszustände

Obwohl jeder Prozess eine unabhängige Einheit darstellt, müssen Prozesse häufig mit anderen Prozessen kommunizieren. Folgendes Beispiel einer Kommandozeile unter **Unix** soll dies verdeutlichen:

who | **wc -l**

In diesem Beispiel werden sofort zwei Prozesse mit den Programmen who und wc gestartet. Das Programm who listet alle z.Z. am System angemeldeten Benutzer auf. Das Programm wc mit der Option -l zählt die Anzahl Zeilen in der Ausgabe von who. Beide Prozesse sind durch eine Pipe verbunden, die der Kommunikation zwi-

schen den Prozessen dient. Es wird deutlich, dass der Prozess des Programms wc erst dann arbeiten kann, wenn der andere Prozess mit dem Programm who Daten geliefert hat. Dieser Prozess befindet sich in dem Prozesszustand blockiert.

Prinzipiell sind im einfachsten Fall die drei Zustände für einen Prozess möglich, die in Abbildung 59 dargestellt sind, wenn man davon ausgeht, dass nicht für jeden Prozess ein eigener Prozessor zur Verfügung steht.

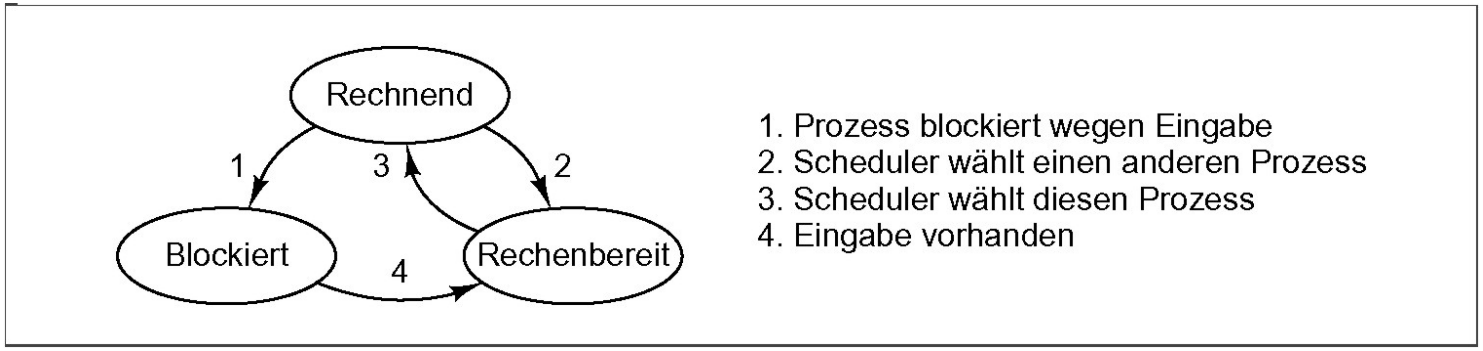


Abbildung 59: Taskmodell mit 3 Zuständen

Folgende Zustände können dabei minimal unterschieden werden:

1. **rechnend** (die Befehle werden in diesem Moment auf der CPU ausgeführt)
2. **rechenbereit** (kurzzeitig gestoppt, um einen anderen Prozess rechnen zu lassen)
3. **blockiert** (nicht lauffähig bis ein bestimmtes externes Ereignis eintritt)

Sicherlich ist es möglich, die Art der Blockade von Prozessen weiter zu spezifizieren. So sind Prozesszustandsmodelle mit weit mehr als drei Zuständen denkbar. Als Beispiel kann die Abbildung 60 dienen.

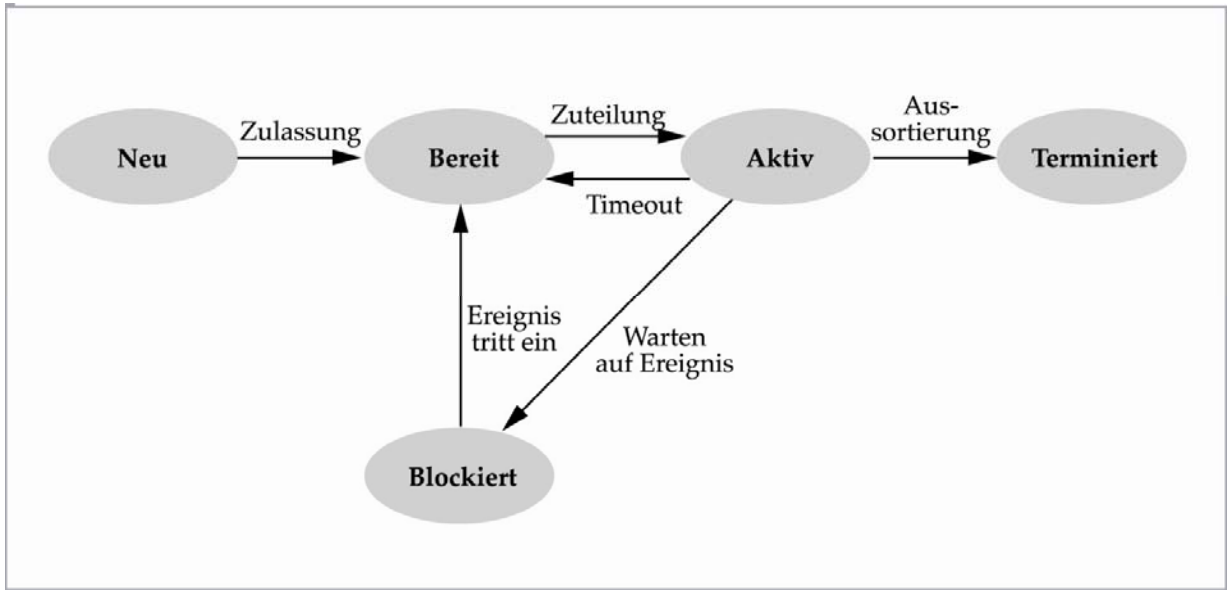
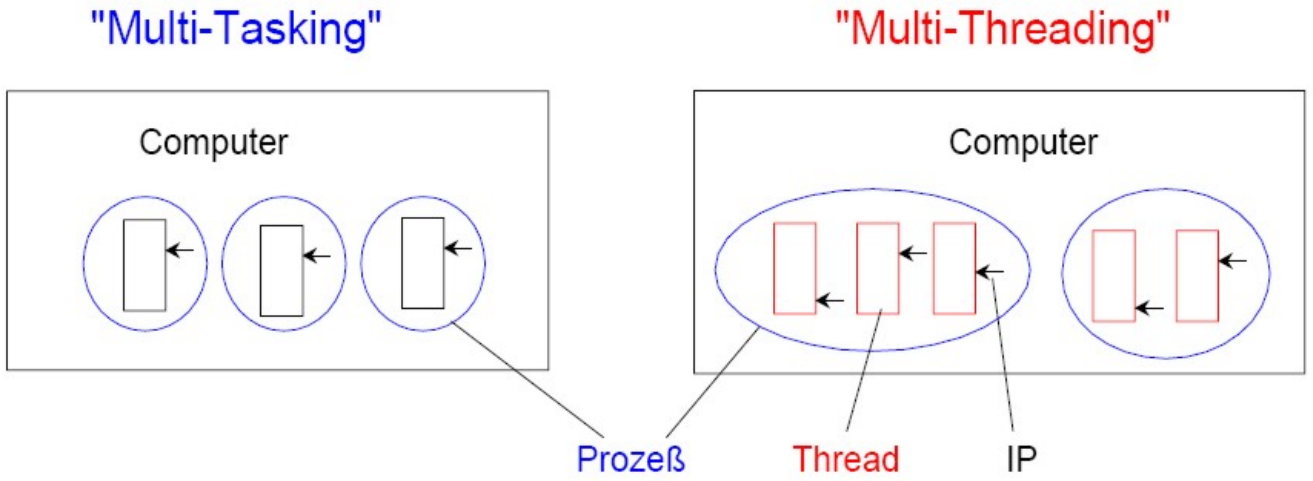


Abbildung 60: Taskmodell mit 5 Zuständen

5.3.2.6. Threads

In traditionellen Betriebssystemen hat jeder Prozess einen eigenen Adressraum und einen einzigen Ausführungsweg. Jedoch gibt es viele Situationen, wo es wünschenswert ist, mehrere Ausführungswege in ein und demselben Adressraum quasiparallel ablaufen zu lassen, als ob es einzelne Prozesse wären, abgesehen von dem gleichen Adressraum.

Threads erweitern das Prozessmodel um die Möglichkeit, mehrere Ausführungswege, die sich in hohem Grade unabhängig voneinander verhalten, in derselben Prozessumgebung ablaufen zu lassen. Abbildung 62 dokumentiert die Unterschiede zwischen dem klassischen Prozess und der Aufteilung in mehrere Threads.



Ganz klar ist bei die Idee, dass Prozesse Parallelarbeit in verschiedenen Adressräumen während Threads Parallelarbeit im gleichen Adressraum bedeutet.

Abbildung 61: Multi-Tasking und Multi-Threading

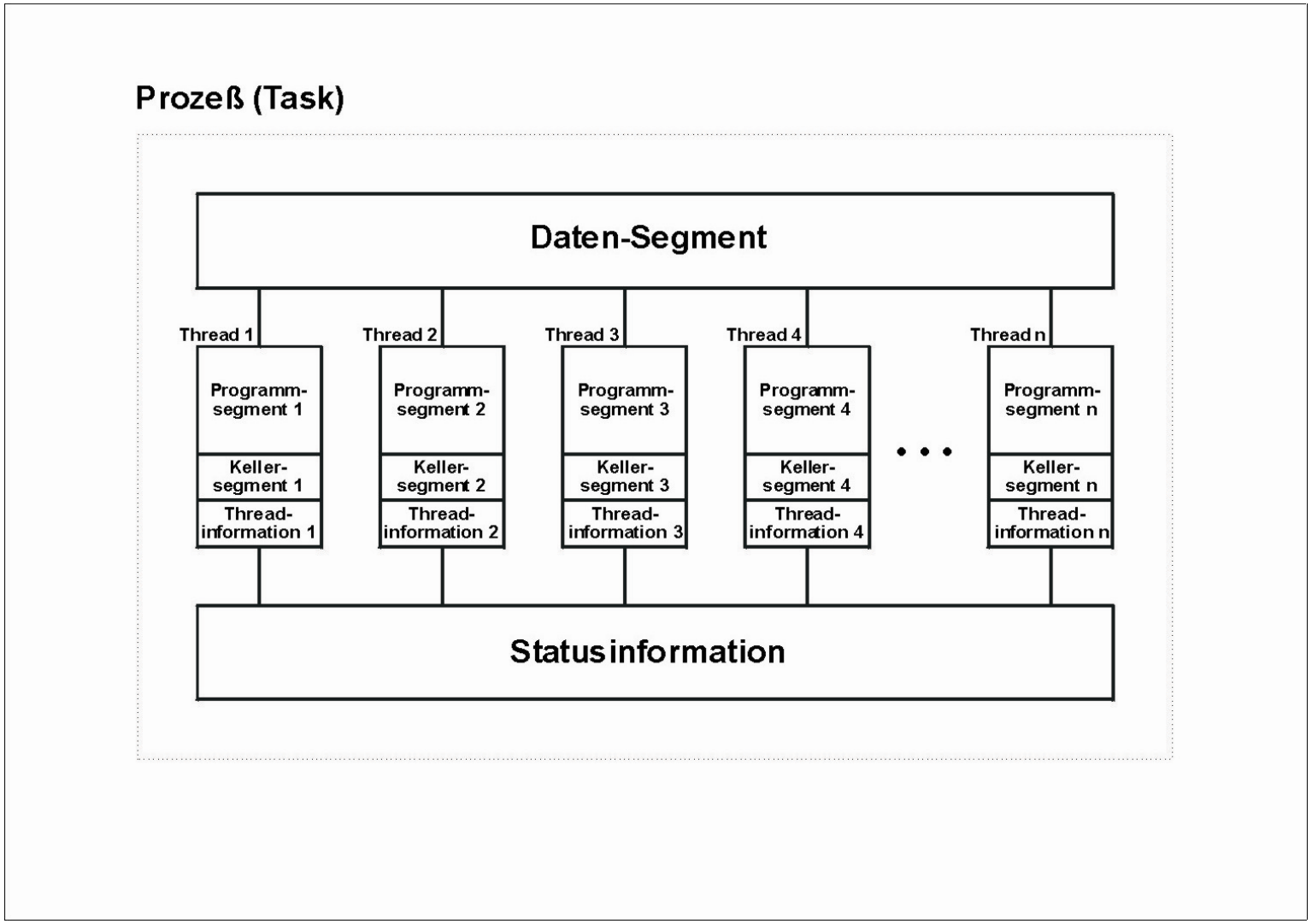


Abbildung 62: Beziehung zwischen Prozess und Thread

Task	Thread
- "Schwergewichtsprozess"	- "Leichtgewichtsprozess"
- **eigener Adressraum für jeden Task**	- **gemeinsamer Adressraum für alle Threads** - globale Daten für alle Threads „sichtbar“ - Stack-Bereiche nicht geschützt
- Interprozess-Kommunikation aufwändig	- Kommunikation zw. Threads einfacher
- Task-Switch erfordert MMU-Aktivitäten	- Thread-Switch erfordert nur Registertausch
- separate Verwaltung offener E/A-Kanäle	- gemeinsame Liste offener E/A-Kanäle (aus „Prozessumgebung“)
- Verwaltungsdaten enthalten Register, Stack, Task-ID, Priorität, Schedulingverfahren, MMU-Parameter, E/A-Daten	- Verwaltungsdaten enthalten Register, Stack, Thread-ID, Priorität, Schedulingverfahren
	- gemeinsamer User-ID

Tabelle 9: Task und Thread

5.3.2.7. 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.

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**.

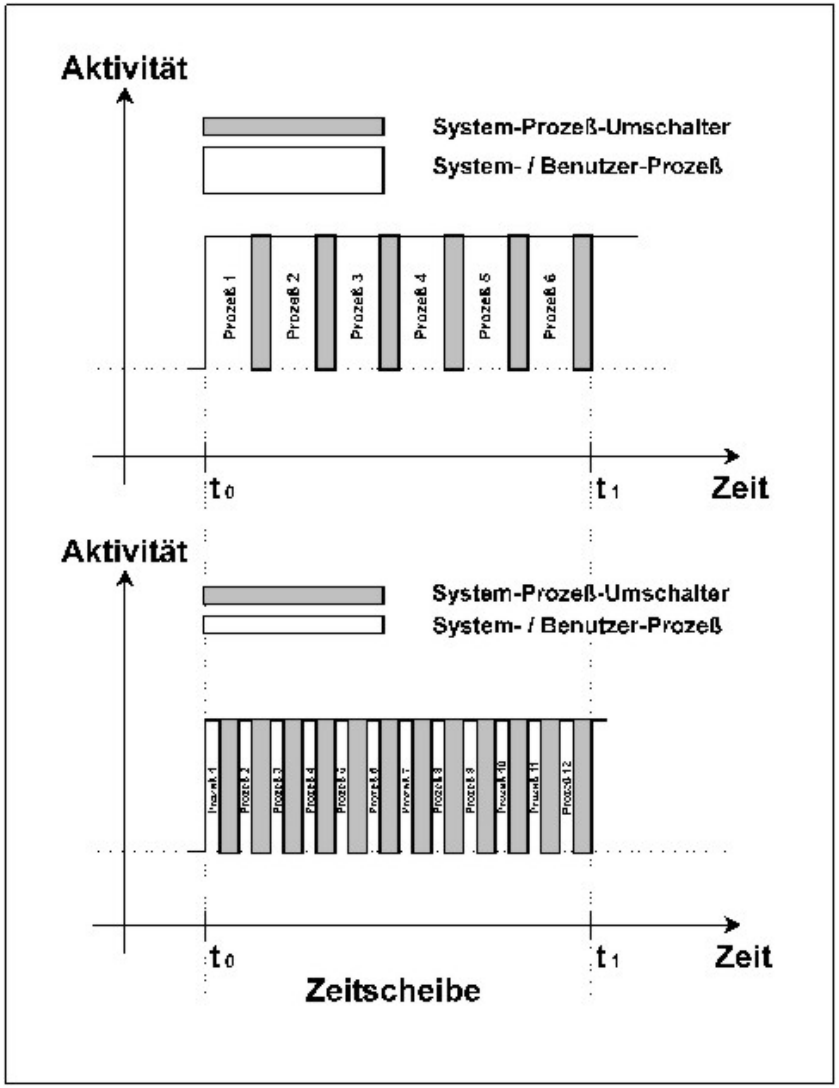


Abbildung 63: gerechtes Scheduling-Verfahren

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.

In Betriebssystemen wie Unix oder Windows besitzt jeder Prozess einen eigenen Speicherbereich, der exklusiv reserviert wird. Dort erhält jeder Prozess seinen eigenen Datenbereich. Der Programmcode kann von mehreren Prozessen genutzt werden, siehe 5.3.2.1. Programme, Prozeduren, Prozesse und Instanzen.

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.

5.3.2.8. 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 24: 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 64 zeigt die Klassifizierungsmerkmale

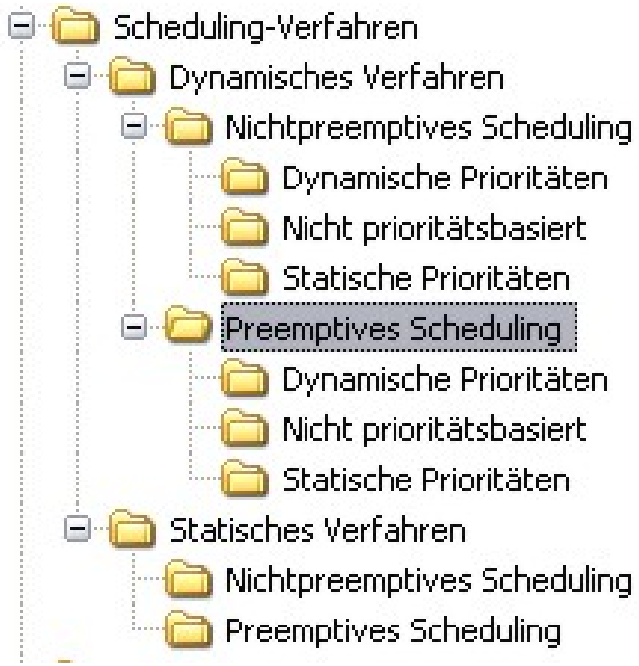


Abbildung 64: Klassifizierungsmerkmale von Schedulingverfahren

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.

5.3.2.9. 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, siehe Abschnitt 5.3.2.1. Programme, Prozeduren, Prozesse und Instanzen und Abschnitt 5.3.2.6. Threads.

Definition 25: 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 26: 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**, siehe Abbildung 59, Abbildung 60 oder Abbildung 65.

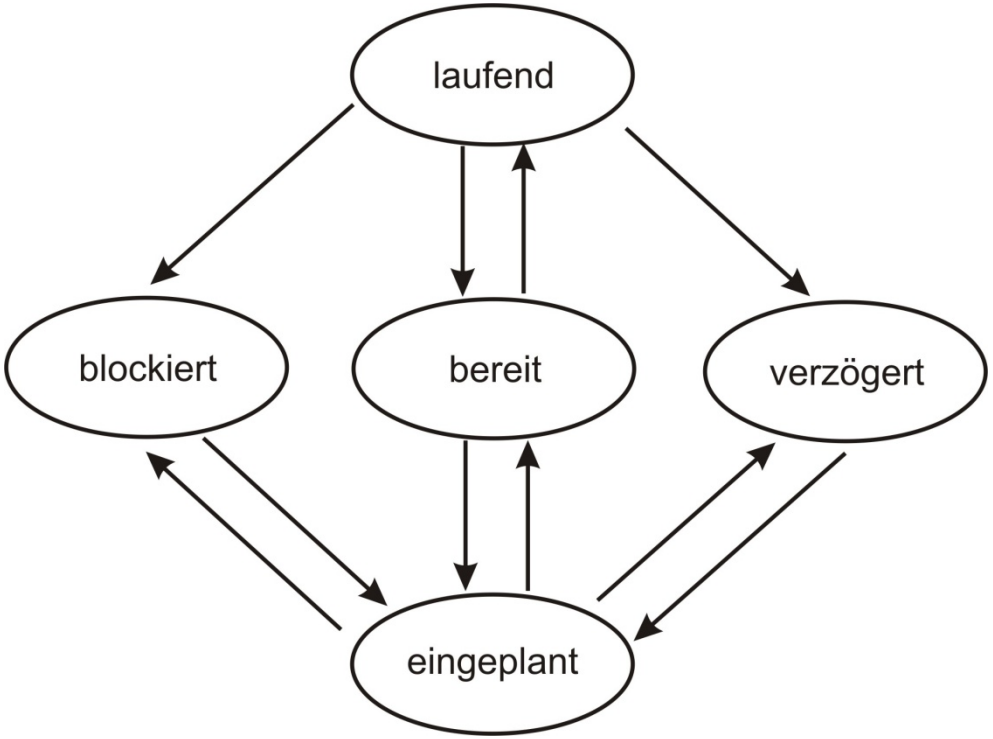


Abbildung 65: Taskzustände im Taskmodell

5.3.2.10. Taskzustände

Die einzelnen Zustände haben dabei folgende Bedeutung:

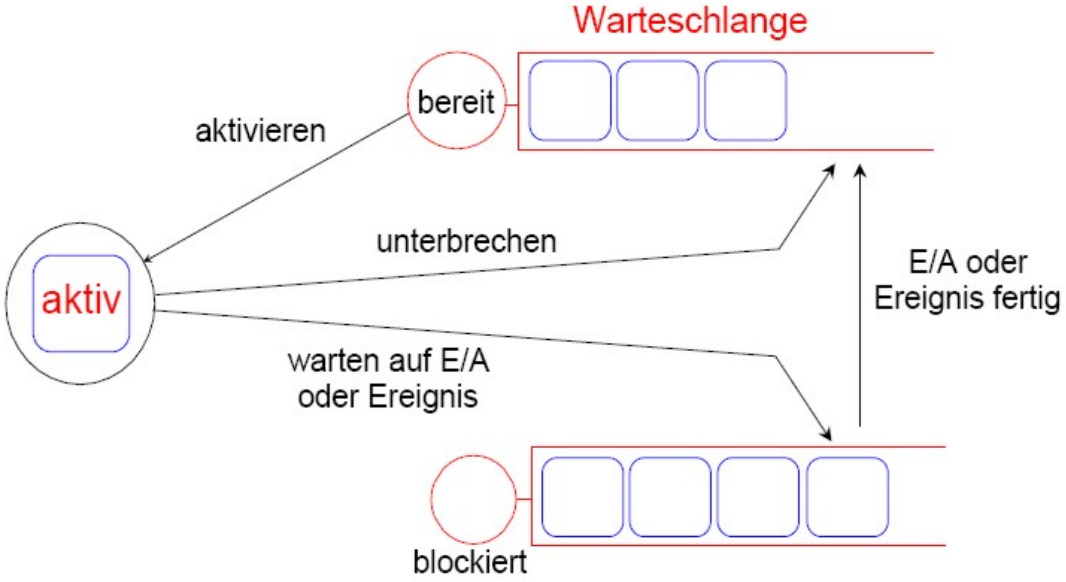
laufend Der Prozess ist dem Prozessor zugeordnet und wird aktuell bearbeitet.

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.



In Abbildung 65 sind 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 66: Verwaltung von Prozessen

5.3.2.11. Statisches und dynamisches Taskmodell

Im Abschnitt 5.3.2. Das Prozesssystem 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 27: 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 28: 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.

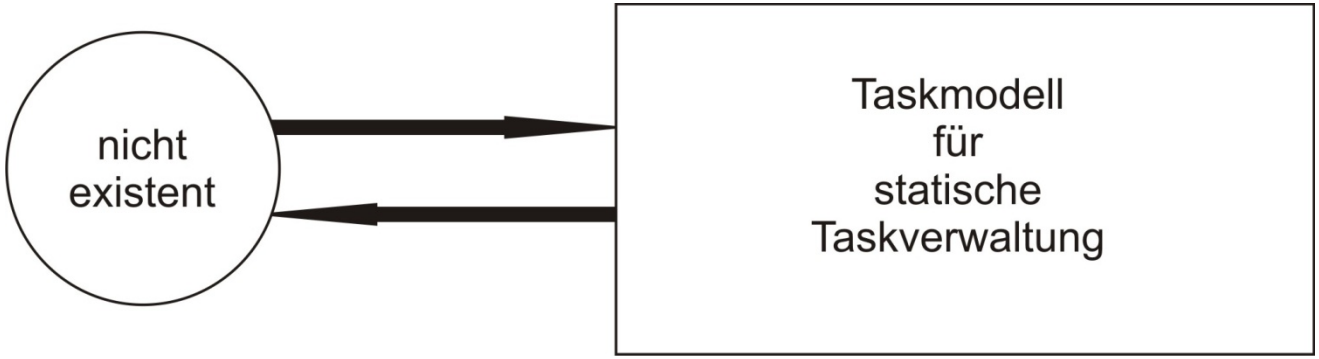


Abbildung 67: Statische und dynamische Taskverwaltung

5.3.2.12. 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**.

5.3.3. 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).

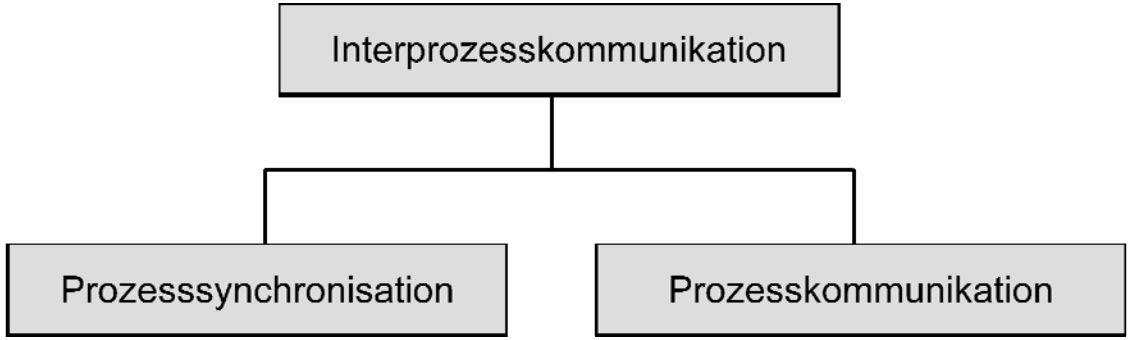


Abbildung 68: Interprozesskommunikation

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.

Folgendes Beispiel aus Abschnitt 5.3.2.5. Prozesszustände, demonstriert diesen Sachverhalt:

who | wc -l

In diesem Beispiel wird deutlich, dass dabei sowohl eine **Datenkommunikation** als auch eine **Prozesssynchronisation** stattfindet.

Das who-Kommando listet alle momentan am System angemeldeten Benutzer auf und gibt sie auf der Standardausgabe aus. Es werden damit Daten **erzeugt**. Im Kommando wc mit der Option -l werden alle Daten auf der Standardeingabe geprüft und die Anzahl Zeilen gezählt und am Ende ausgegeben. Durch die Pipe werden die erzeugten Daten von Prozess 1 (who) im Prozess 2 (wc –l) **verbraucht**. Es handelt sich um ein so genanntes **Erzeuger-Verbraucher-Problem**.

Gleichzeitig findet auch eine Synchronisation zwischen den Prozessen statt. Der Prozess 1 (who) kann nur endlich viele Daten in die Pipe schicken, wenn man sich diese als Puffer im Hauptspeicher vorstellt. Ist der Puffer voll, wird der Prozess 1 angehalten, man sagt, er **schläft**. Andererseits kann der Prozess 2 (wc -l) ohne Daten seine Aufgaben nicht erledigen. Ist dies der Fall, dann schläft er ebenfalls.

Das Beispiel führt nur dann zum Ziel, wenn alle Daten, die Prozess 1 erzeugt, durch Prozess 2 verbraucht werden. Ist Prozess 1 fertig, sendet er ein **Signal** (**EOF** – **E**nd **o**f **F**ile) an Prozess 2 und beendet sich. Nachdem Prozess 2 dieses Signal empfangen hat, ist dies für ihn das Zeichen, die ermittelte Zeilenanzahl auszugeben und sich dann ebenfalls zu beenden.

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.

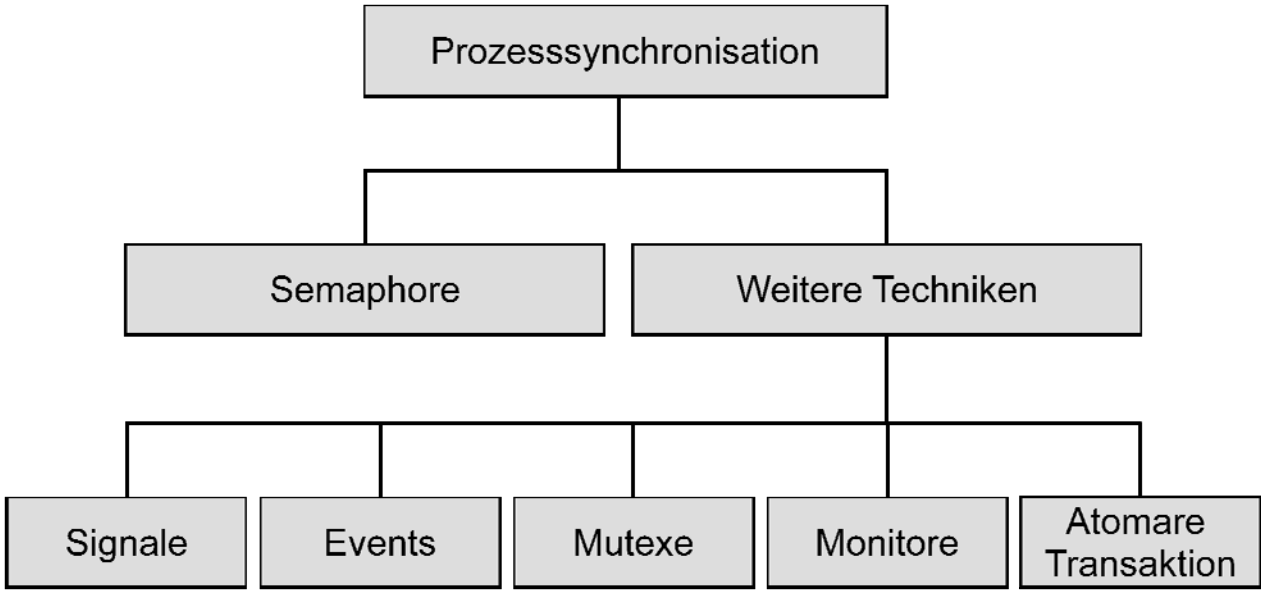


Abbildung 69: Möglichkeiten der Prozesssynchronisation

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***).

5.3.3.1. 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.

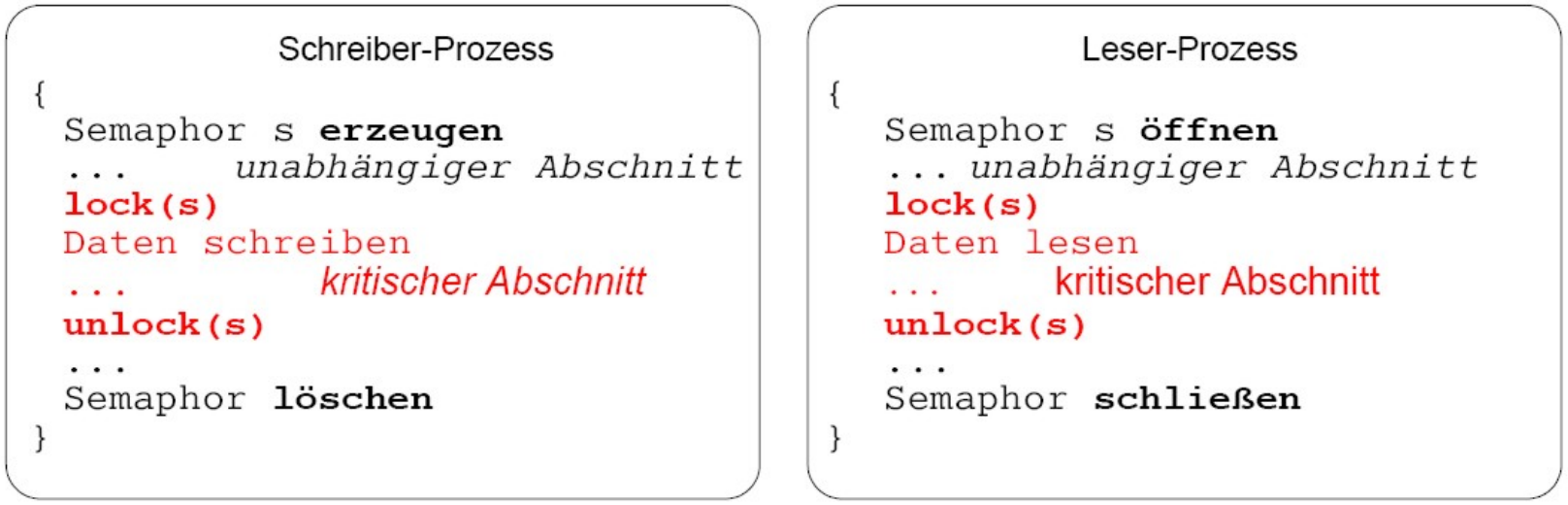


Abbildung 70: Semaphore

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.

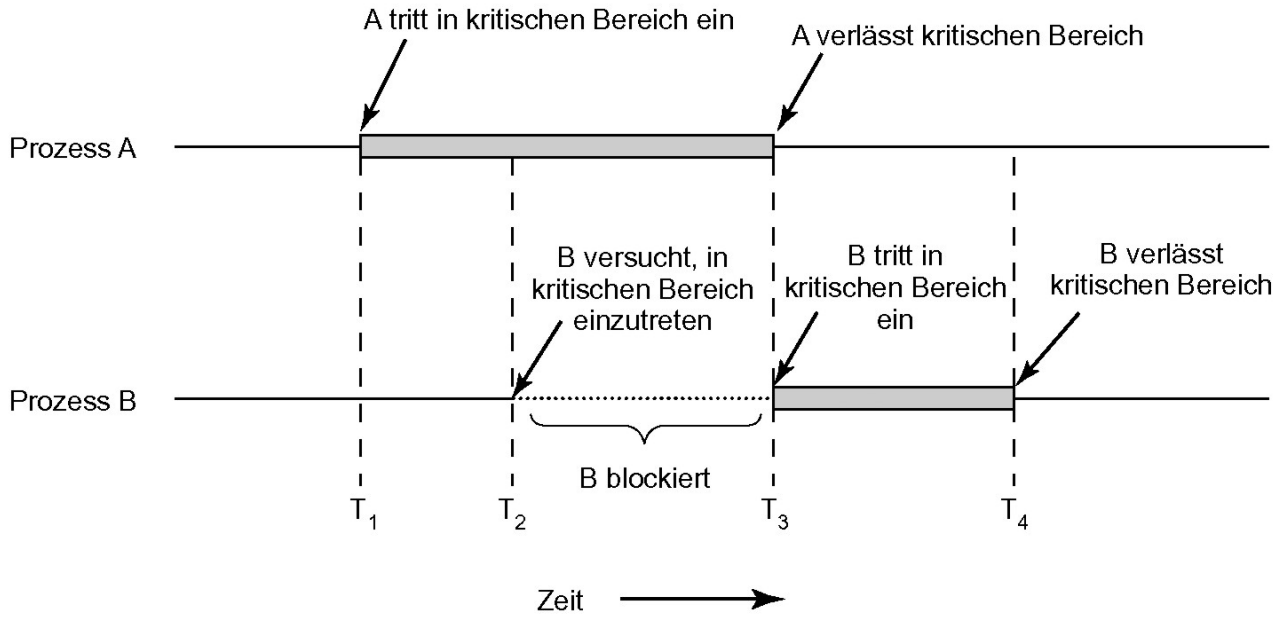


Abbildung 71: Wechselseitiger Ausschluss unter Verwending kritischer Bereiche

Im Allgemeinen unterscheidet man drei verschiedene Typen von Semaphoren:

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

Wie in Abbildung 72 dargestellt, besteht ein Semaphor aus einer Zählvariablen sowie zwei nicht unterbrechbaren Operationen **„Passieren“** – P (**up-Operation**) und **„Verlassen“** – V (**down-Operation**).

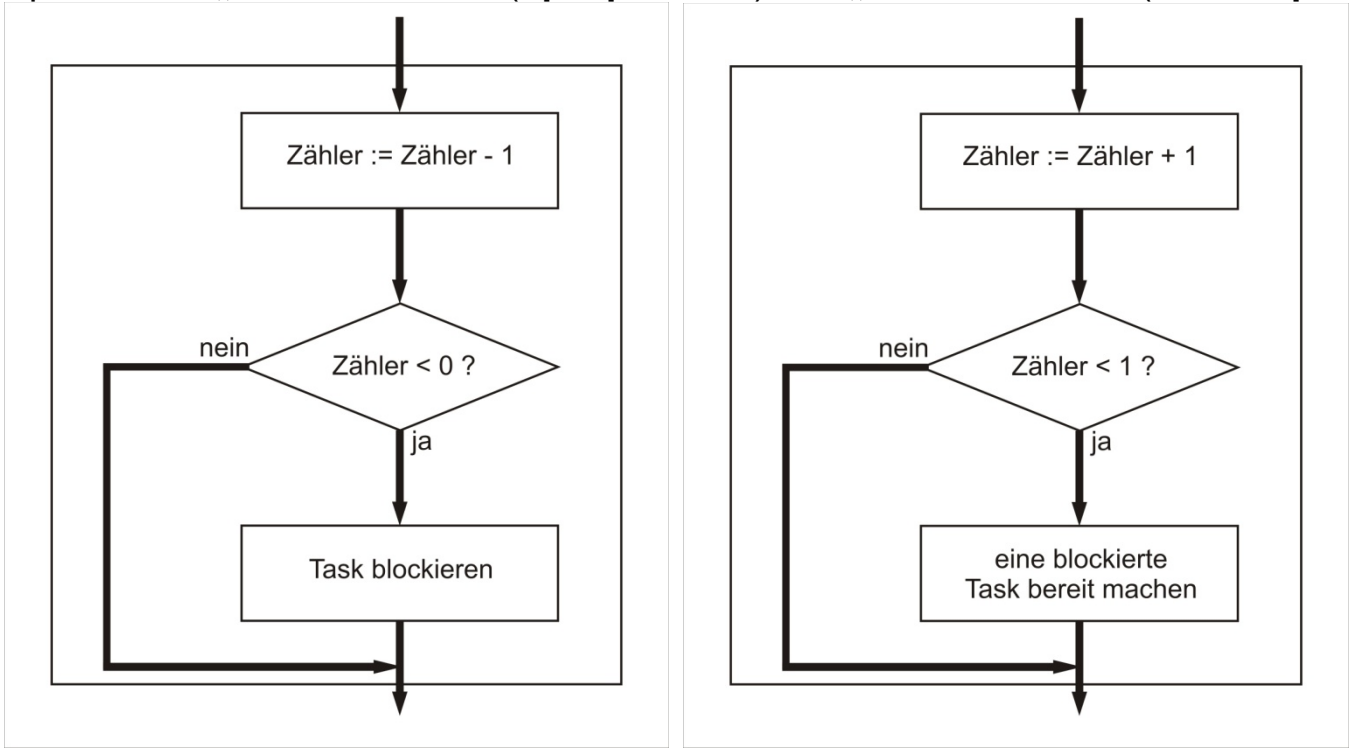


Abbildung 72: Die Operationen „Passieren“ und „Verlassen“ eines Semaphors

Als Demonstrationen der Operationen sind PL/SQL-Prozeduren in Abbildung 73 und Abbildung 74 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 73: 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 74: 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).

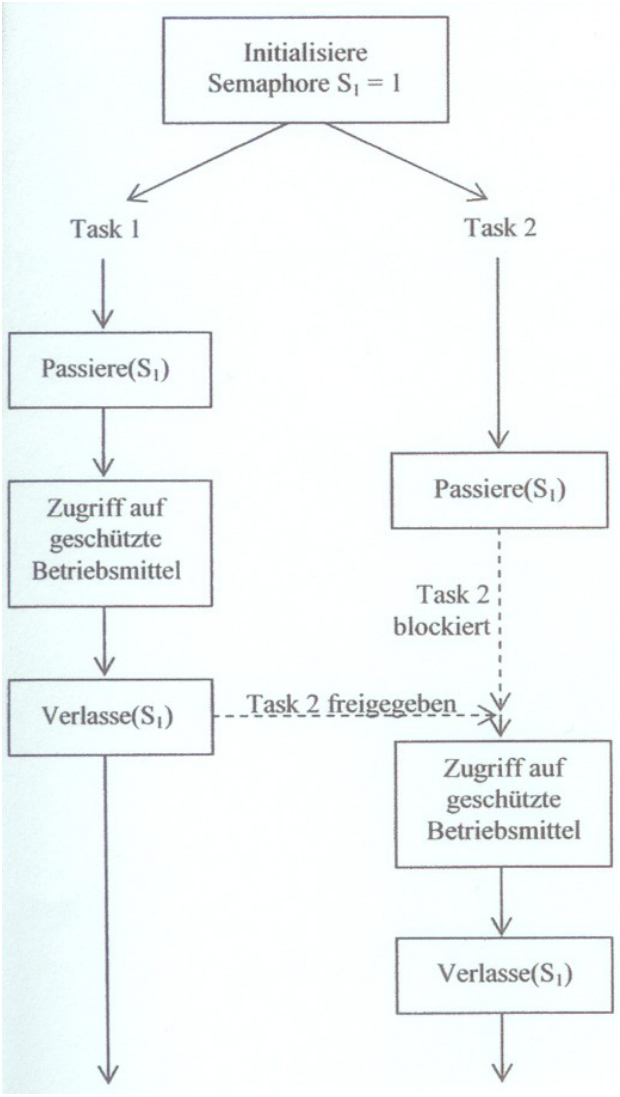
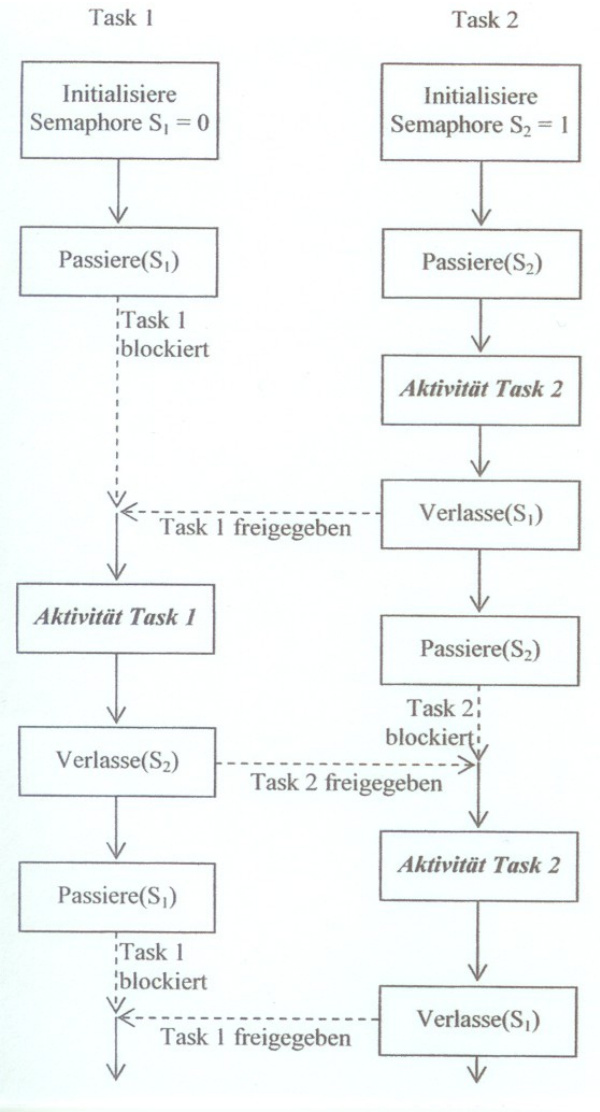


Abbildung 75: Sperrsynchronisation mit einem Semaphor



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 75 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 76 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 76: 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.

5.3.3.2. 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 77: Realisierung von *mutex_lock* und *mutex_unlock*

5.3.3.3. 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.

5.3.3.4. 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 78).

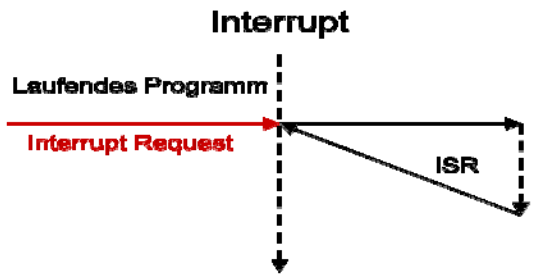


Abbildung 78: Interrupts

Events werden erst dann auf signalisiert gesetzt, wenn eine bestimmte Reaktion der Umwelt (x mal) eingetreten ist (siehe Abbildung 79).

Abbildung 79: 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.

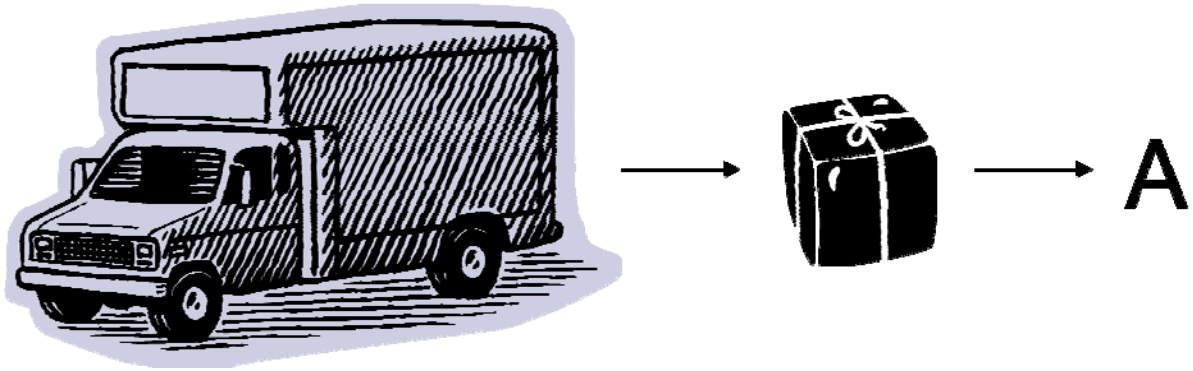


Abbildung 80: Beispiel boole’sches Event

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.

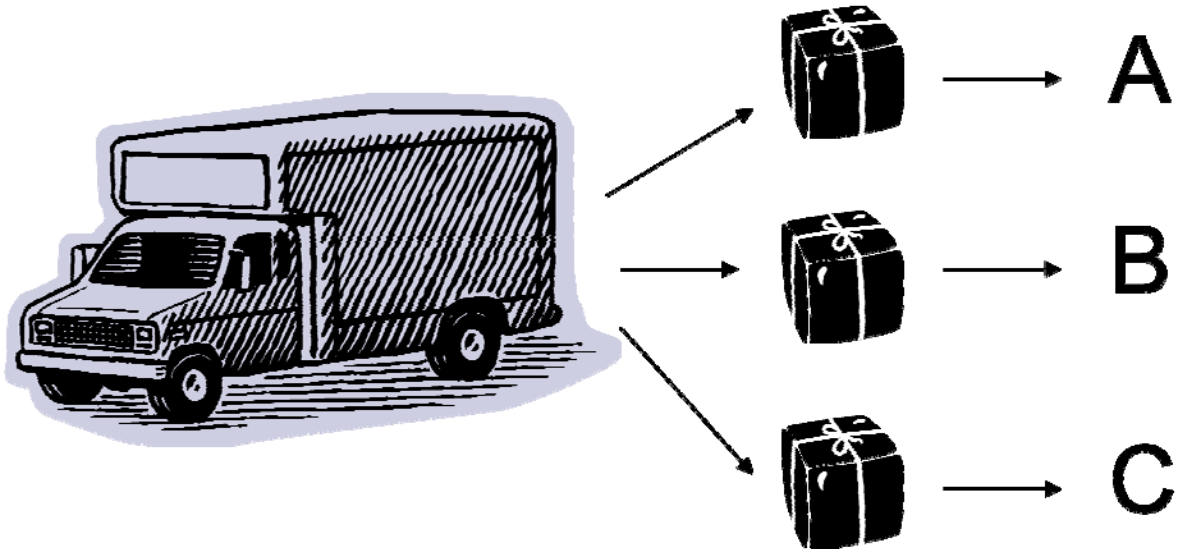


Abbildung 81: Beispiel zählendes Event

5.3.3.5. Signale

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

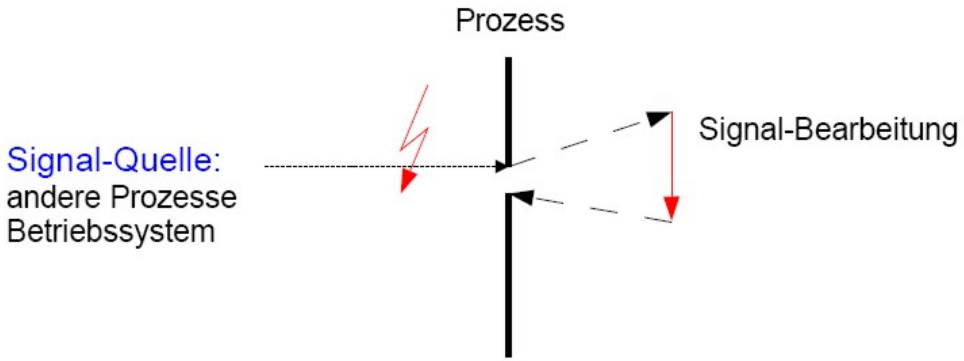


Abbildung 82: Signale

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 11 sind 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 10: Signale und Interrupts

Signalname	Signalnummer	Bedeutung
SIGHUP	1	Abbruch einer Terminalleitung (analog zum Aufhängen beim Telefon
SIGINT	2	Unterbrechung von dem Terminal
SIGQUIT	3	Abbruch vom Terminal
SIGILL	4	Ausführung eines ungültigen Befehls
SIGTRAP	5	Trace trap Unterbrechung (Einzelschrittausführung)
SIGFPE	8	Ausnahmesituation bei einer Gleitkommaoperation
SIGKILL	9	Kill-Signal (kill -9 PID)
SIGBUS	10	Fehler auf dem Systembus
SIGSEGV	11	Speicherzugriff mit unerlaubtem Segmentzugriff
SIGSYS	12	Ungültiges Argument beim Systemaufruf
SIGPIPE	13	Es wurde in eine Pipe geschrieben, von der nicht gelesen wird
SIGALARM	14	Ein Zeitintervall ist abgelaufen
SIGTERM	15	Signal zur Programmbeendigung
SIGCLD	18	Signalisiert die Beendigung des Sohnprozesses
SIGPWR	19	Signalisiert einen Spannungsausfall

Tabelle 11: Signale in Unix

5.3.3.6. 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 83: Nachrichtenaustausch in Form 1:1

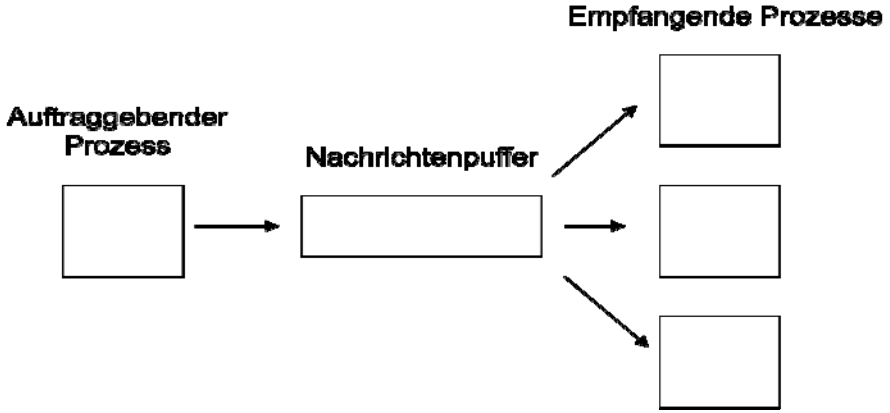


Abbildung 84: Nachrichtenaustausch in Form 1:n

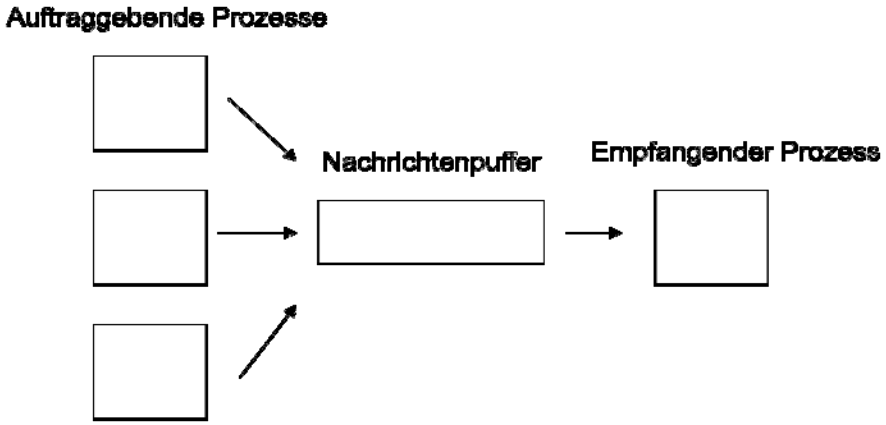


Abbildung 85: Nachrichtenaustausch in Form m:1

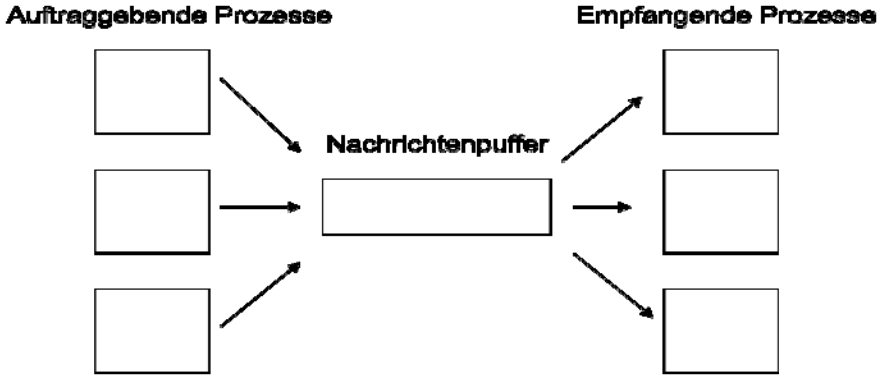


Abbildung 86: Nachrichtenaustausch in Form m:n

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. Die Abbildung 87 zeigt eine **full-duplex** Kommunikation über zwei Message Queues.

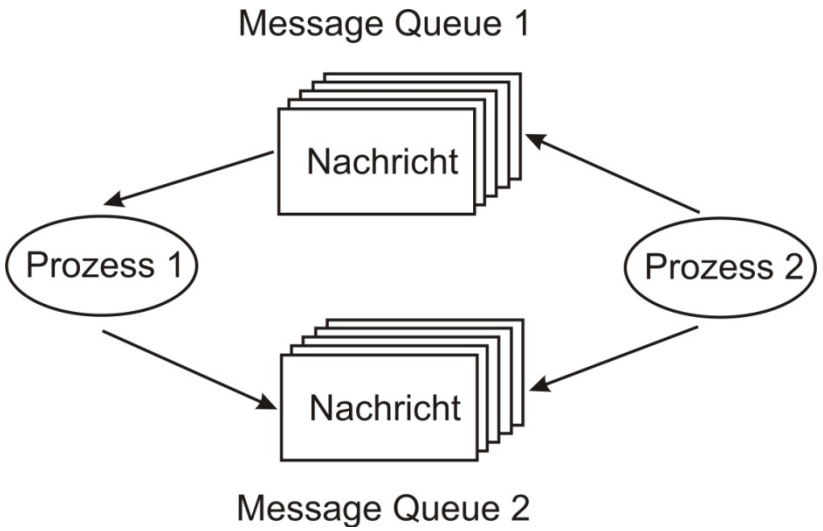


Abbildung 87: Full Duplex Interprozesskommunikation mit Message Queues

5.3.3.7. 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. In Tabelle 12 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 12: Unterschiede zwischen Message Queues und Pipes

5.3.3.8. 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 88: Deadlock

5.3.3.9. Prozedurfernaufruf – Remote Procedure Call

Eine Variante des elementaren Nachrichtenaustauschmodells stellt der Prozedurfernaufruf (RPC, Remote Procedure Call) dar. Hierbei handelt es sich um eine inzwischen allgemein akzeptierte und gängige Methode für die Kommunikationskapslung in einem verteilten System. Im Wesentlichen funktioniert diese Technik so, dass sie es Programmen auf verschiedenen Rechnern ermöglichen, mit Hilfe einer einfachen Prozeduraufruf- / -übergabesemantik miteinander zu kommunizieren und zu koordinieren, als befänden sich beide Programme auf demselben Rechner. Das bedeutet, dass der Prozeduraufruf für den Zugriff auf externe Dienste verwendet wird.

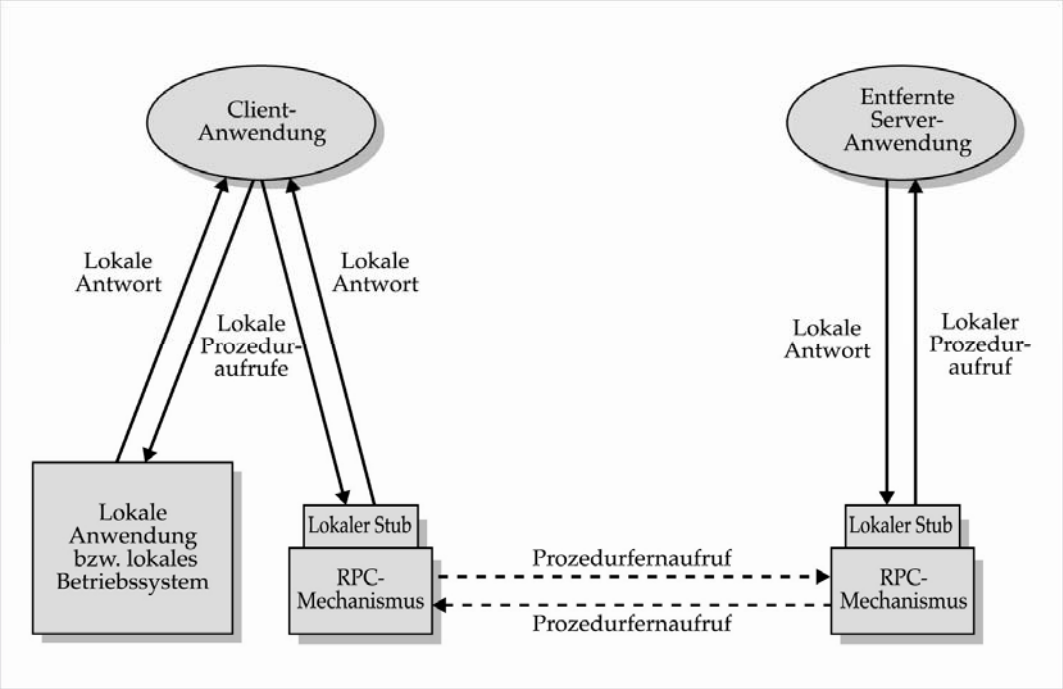


Abbildung 89: Remote Procedure Call

Die Beliebtheit dieses Ansatzes ist den folgenden Vorteilen zuzuschreiben:

1. Der Prozeduraufruf ist eine allgemein akzeptierte, genutzte und verstandene Abstraktion.
2. Mit Hilfe von Prozedurfernaufrufen können entfernte Schnittstellen in Form benannter Operationen mit zugeordneten Argument- und Ergebnisdatentypen definiert werden. Somit kann die Schnittstelle klar dokumentiert und verteilte Programme können statisch auf Typfehler geprüft werden.
3. Da eine standardisierte und präzise definierte Schnittstelle festgelegt ist, kann der Kommunikationscode für eine Anwendung automatisch generiert werden.
4. Da eine standardisierte und präzise definierte Schnittstelle festgelegt ist, können Entwickler Client- und Server-Module schreiben, die mit geringfügigen Anpassungen und Codeänderungen zwischen Computern und Betriebssystemen verschoben werden können.

5.3.4. Das Ein- / Ausgabe-System

Eine der Hauptaufgaben eines Betriebssystems ist es, alle Ein- / Ausgabegeräte eines Computers zu überwachen und zu steuern. Es müssen Kommandos an die Geräte weitergeleitet, Unterbrechungen (**Interrupts**) abgefangen und etwaige Fehler behandelt werden.

Außerdem sollte eine Schnittstelle zwischen den Geräten und dem Rest des Systems zur Verfügung stehen, die einfach und leicht zu benutzen ist. Soweit wie möglich ist, sollte die Schnittstelle für alle Geräte gleich sein (**Geräteunabhängigkeit**).

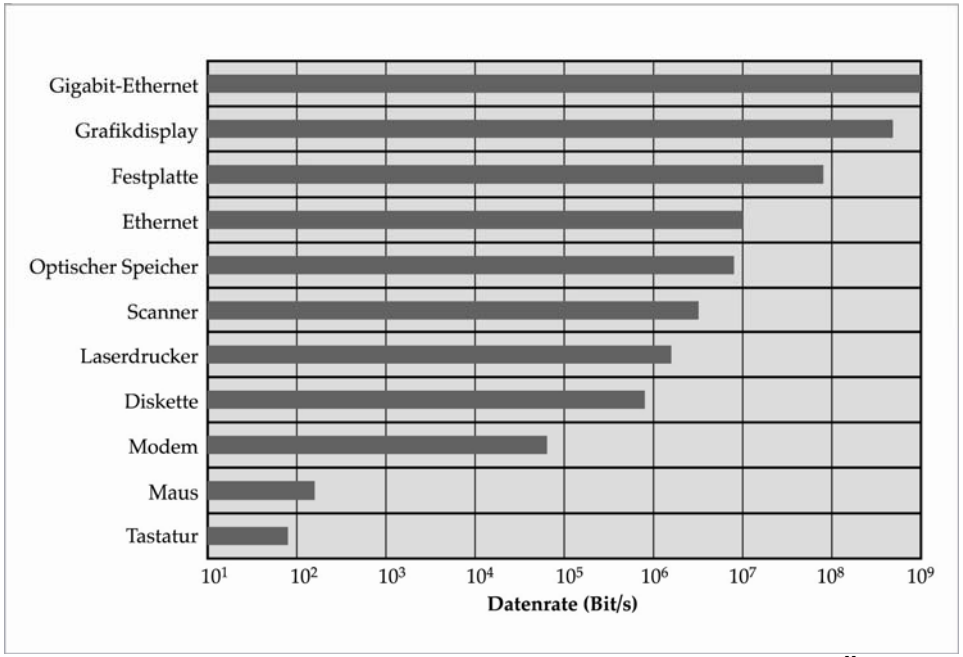


Tabelle 13: typische Ein- / Ausgabegeräte und die Übertragungsraten

5.3.4.1. Ein- / Ausgabegeräte

Im Sinne der Betriebssystembetrachtungen ist die Schnittstelle der Geräte zur Software wichtig – die Kommandos, die die Hardware kennt, die Funktionen, die sie ausführt und mögliche Fehlermeldungen.

Eine Reihe von Geräten (wie Tastatur, Bildschirm, Drucker u.ä.) zeichnen sich dadurch aus, dass der Zugriff auf diese Geräte **zeichenweise sequentiell** (**zeichenorientierte Geräte**) erfolgt.

D.h., einzelne Dateien bzw. Zeichen werden in einer bestimmten, wohldefinierten Reihenfolge von diesen Geräten **gelesen** bzw. auf diese Geräte **geschrieben**.

Funktionen zur zeichenorientierten Ein- und Ausgabe nutzen diesen Umstand aus und ermöglichen dem Anwendungsprogramm auf relativ einheitliche Weise, ein breites Spektrum an **Peripheriegeräten** zu bedienen.

Ein Teil dieser Geräte (z.B. Drucker) verarbeiten die Zeichen zur Ein- bzw. Ausgabe wesentlich langsamer, als sie der Rechner zur Verfügung stellt bzw. verarbeiten kann. In diesem Fall muss eine Synchronisation zwischen der CPU des Rechners und dem Gerät herbeigeführt werden. Aus diesem Grund kann zu jedem Zeitpunkt der Zustand des peripheren Gerätes durch den Rechner abgefragt werden.

Weitere Geräte (wie Disketten-, Festplatten- oder CD-ROM-Laufwerke) ermöglichen die direkte Adressierung der Daten. Diese Geräte verarbeiten die Daten nicht typischerweise zeichenorientiert, sondern **blockweise** (**blockorientierte Geräte**).

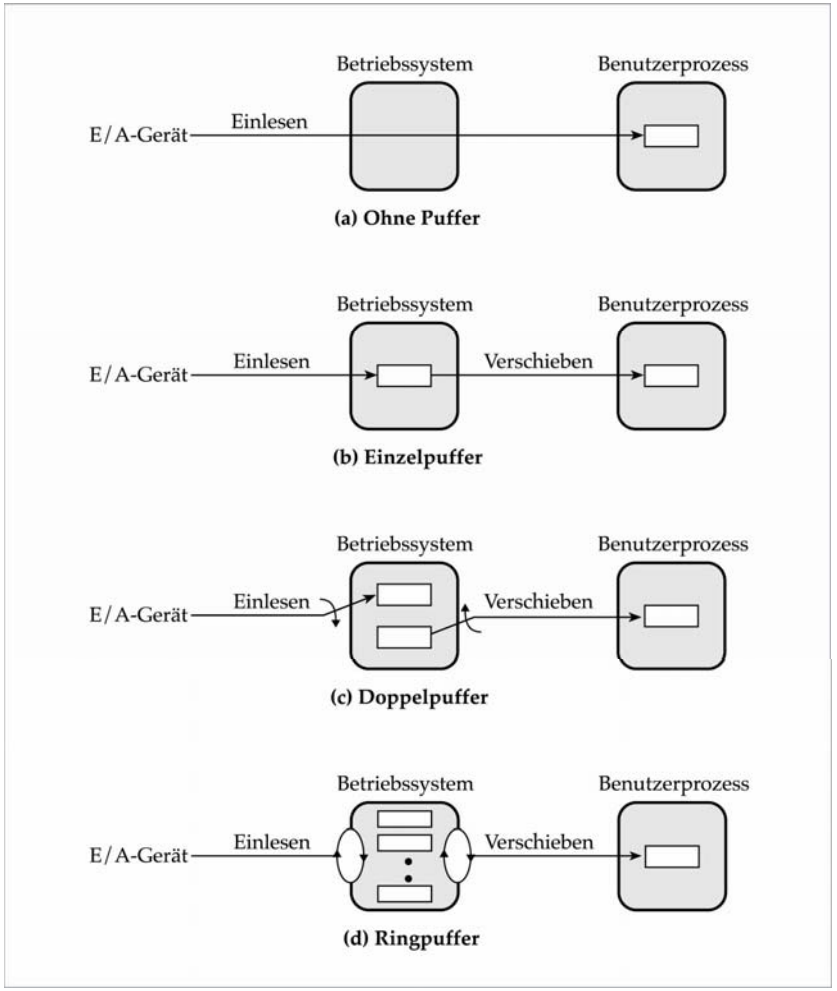


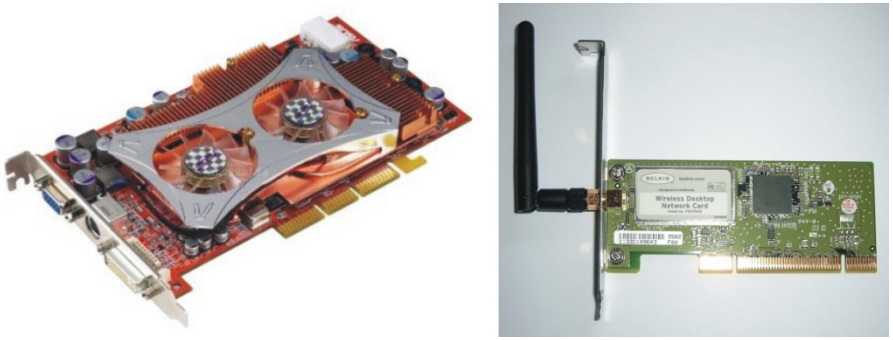
Abbildung 90: E/A-Pufferverfahren (Einlesen)

Ein Zugriff auf die Daten ist nur in Einheiten von Blöcken bzw. Sektoren möglich, wobei die Sektoren entweder durch eine fortlaufende Nummer adressiert werden oder durch Angabe der physischen Speicherzelle auf dem

Datenträger (Laufwerk, Kopfnummer, Zylindernummer, Sektornummer). Die physische Ansteuerung dieser Geräte wird in Funktionen zur blockorientierten Ein- und Ausgabe zusammengefasst.

5.3.4.2. Steuereinheiten

Die Ein- / Ausgabeeinheiten bestehen typischerweise aus einer **mechanischen** und einer **elektronischen Komponente**. Meistens ist es möglich, diese beiden Komponenten voneinander zu trennen, so dass man einen modularen und allgemeinen Entwurf erhält. Die elektronische Komponente wird als **Controller** oder **Adapter** bezeichnet. Bei Personalcomputern ist es häufig als Einsteckkarte realisiert. Diese Struktur wird in Abbildung 8 dargestellt.



In Abbildung 91 sind zwei Beispiele für verschiedene Einsteckkarten dargestellt. Links eine **Grafikkarte Asus A9800 Pro** und rechts eine **Belkin Wireless Desktop Network Card**.

Abbildung 91: Einsteckkarten

Jeder Controller besitzt einige Register, die für die Kommunikation mit dem Prozessor über den standardisierten Bus vorhanden sind. Damit können die Befehle an den Controller übermittelt werden. Weiterhin besitzen die meisten Controller zusätzlich Datenpuffer, die durch das Betriebssystem gelesen und geschrieben werden können.

Für den Zugriff auf diese Register existieren zwei Möglichkeiten:

1. Jedem Register wird eine sogenannte **Ein- / Ausgabe-Port-Nummer** zugewiesen, die ein 8-Bit- oder 16-Bit-Integer-Wert sein kann.

2. Jedem Register wird eine Hauptspeicheradresse zugewiesen, an der sich kein Hauptspeicher befindet. Dieses wird **Memory-Mapped Input Output** bezeichnet.

5.3.4.3. Direct Memory Access

Die Aufgabe für den Prozessor besteht darin, den Controller zu adressieren, um Daten mit diesen auszutauschen. Die CPU könnte die Daten vom Controller zeichenweise holen, aber das würde sehr viel Rechenzeit verschwenden, weshalb oft ein anderes Verfahren eingesetzt wird – **DMA** (Direct Memory Access).

Das Betriebssystem kann DMA nur dann verwenden, wenn die Hardware mindestens einen DMA-Controller zur Verfügung stellt. Dieser ist in den meisten Rechnern vorhanden und kann

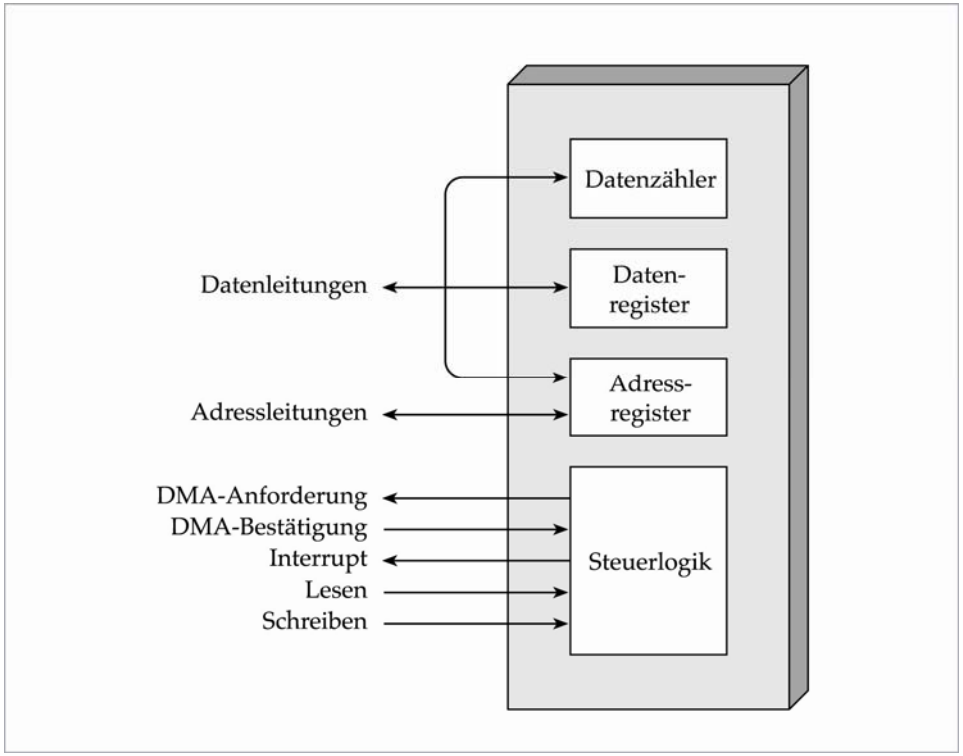


Abbildung 92: DMA-Blockdiagramm

- entweder direkt am Gerät an der Steuereinheit oder
- zentral auf dem Mainboard vorhanden sein.

Der DMA-Controller hat unabhängig von der CPU immer Zugriff auf den Systembus, wie in Abbildung 93 zu sehen ist.

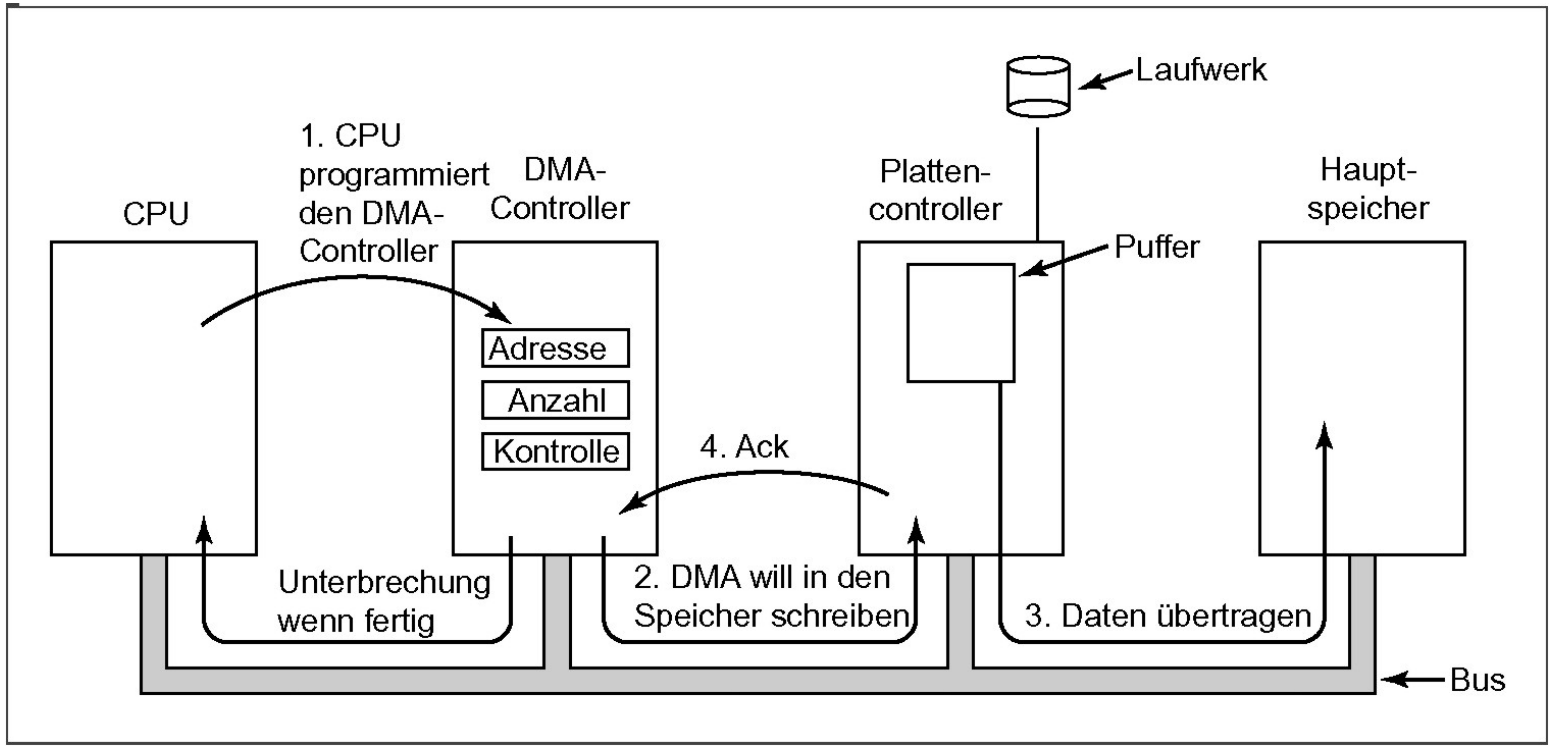


Abbildung 93: DMA-Transfer

5.3.4.4. Interrupts

Aus den Angaben in Tabelle 13 ist ersichtlich, dass die einzelnen Geräte mit sehr unterschiedlicher Geschwindigkeit die Daten übertragen können. Es muss eine Synchronisation zwischen den Geräten und der CPU kommen. In der Theorie sind dazu prinzipiell zwei Verfahren möglich:

1. Polling – Die CPU überträgt ein Zeichen auf das oder von dem Gerät und wartet dann in einer Schleife bis das Gerät für den nächsten Transfer bereit ist. In dieser Zeit kann die CPU nichts anderes tun.

2. Interrupts – Die CPU beauftragt den Controller mit der Aufgabe der Übertragung und arbeitet normal weiter. Ist das Gerät fertig, unterbricht es die laufende Arbeit der CPU und diese kann jetzt den nächsten Transfer anstoßen.

Die prinzipielle Arbeitsweise bei Interrupts ist in Abbildung 94 zu sehen.

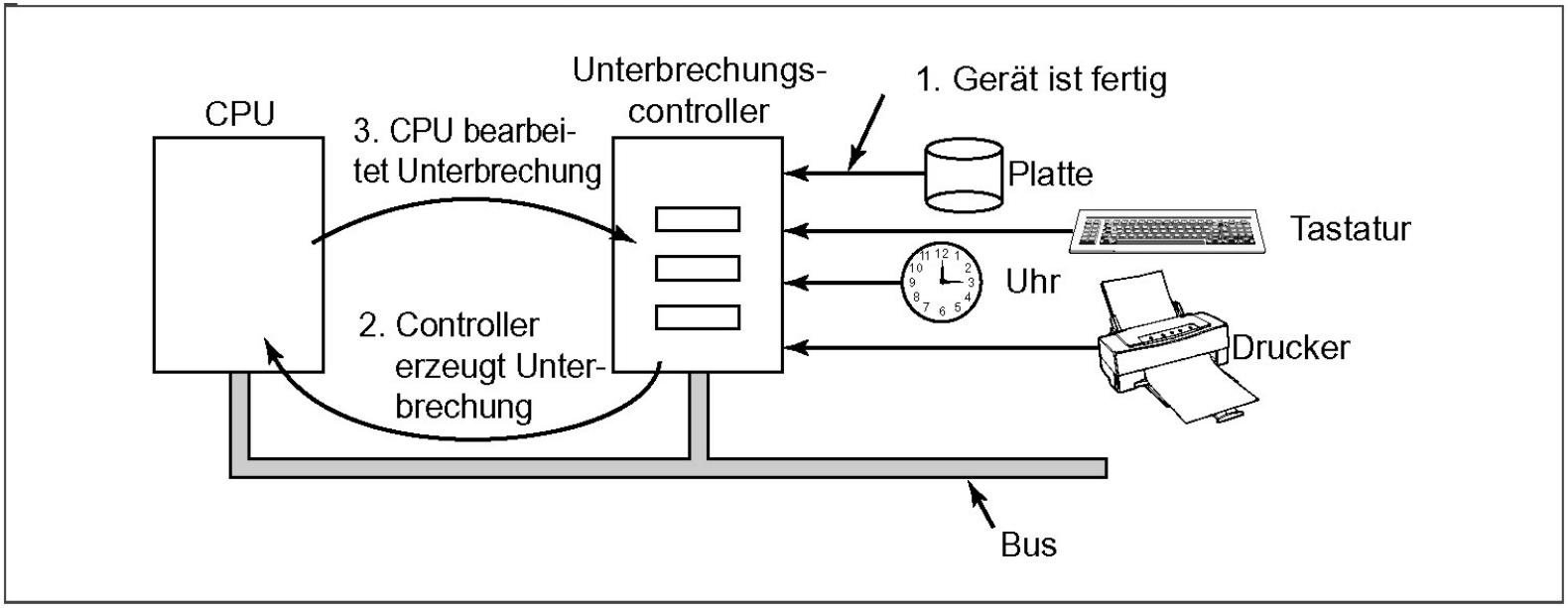


Abbildung 94: Ablauf einer Unterbrechung (Interrupt)

5.3.5. Systemuhren und Zeitgeber

Spezifisch für Echtzeitsysteme sind die Einbindung von **Zeitgebern** in Anwendungsprogramme. Ferner sind Mechanismen vorhanden, um den Programmablauf zeitlich zu steuern. Wie im Abschnitt 5.1.1. Rechtzeitigkeit bereits dargestellt, hängt die Richtigkeit des Ergebnisses bei der Echtzeitdatenverarbeitung neben der logischen Richtigkeit ganz wesentlich von der Rechtzeitigkeit des Ergebnisses ab.

Bezüglich der Systemuhren und Zeitgeber kann man folgendes unterscheiden:

1. Systemuhr oder Systemzeitgeber (**System Clock**)

2. Hilfsuhr (**Auxiliary Clock**)
3. Alarmzeitgeber (**Watchdog Timer**)

Je nach Hardware findet man tatsächlich drei verschiedene Zeitgeber (Timer) oder man bedient zwei oder mehrere Zeitgeber mit einem **Timerbaustein**. Ein Timerbaustein besteht aus einem Quarz, der mit einer definierten Frequenz schwingt.

5.3.6. Dateien und Dateisystem

Alle Anwendungsprogramme hantieren mit Mengen logisch zusammenhängender Daten, den Dateien. Das muss derart geschehen, dass dem Benutzer die Details über das **Wo** und **Wie** der Speicherung verborgen bleiben.

Die wahrscheinlich wichtigste Eigenschaft jeder Abstraktion ist die Art und Weise, wie die zu verwaltenden Objekte benannt werden.

5.3.6.1. Dateinamen

Die Speicherung dieser Dateien erfolgt meistens auf Geräte, den so genannten externen Speichermedien (extern, da sie außerhalb des Prozessors liegen). Und die Verwaltung, d.h. die Aufzeichnung, welche Dateien gespeichert sind, wo die Daten stehen, wann die Dateien erzeugt oder geändert wurden, welche Zugriffsrechte vergeben sind usw., übernimmt das **Dateisystem** (Filesystem) des Betriebssystemkerns.

Definition 29: Datei
Eine Datei ist eine Menge an Daten (unstrukturiert oder strukturiert), die auf einem Speichermedium unter einem Namen abgelegt sind.

Eine Anwendung (später als Prozess bezeichnet) erzeugt eine Datei, indem sie Daten unter einem wohlbestimmten Namen auf einem Speichermedium speichert. In der Regel existiert diese Datei auch nach Beenden der Anwendung und eine andere Anwendung kann ihrerseits auf diese Daten über den ihr bekannten Namen zugreifen.

Für den Anwender und damit die Anwendungen ist somit die Regel für die Vergabe von Dateinamen von großer Bedeutung und diese unterscheiden sich bei den verschiedenen Betriebssystemen nach folgenden Kriterien:

- Maximale Anzahl Zeichen,
- zugelassene Zeichen,
- Schreibweise (Unterscheidung zwischen Groß- und Kleinschreibung oder keine Unterscheidung),
- Anzahl Teile (maximal zwei), die zusammen den Dateinamen bilden (d.h. mit oder ohne Dateierweiterung) und
- Kennzeichnung für besondere Dateien (z.B. Systemdateien).

In Tabelle 14 sind einige typische Dateierweiterungen dargestellt, die ihren Ursprung im Betriebssystem MS-DOS haben, aber wegen ihrer weiten Verbreitung auch in anderen Betriebssystemen äquivalent verwendet werden, die keine Dateierweiterung verwenden. Mit einer solchen Standardisierung der Dateierweiterungen kann man diesen bewusst eine Anwendung zuordnen, mit der diese Art von Dateien immer bearbeitet werden kann.

Erweiterung	Bedeutung
Name.bak	Backup-Datei
Name.c	C-Quelltextdatei
Name.gif	Compuserve Graphical Interchange File Format
Name.hlp	Hilfedatei
Name.html	Hypertext-Markup-Language-Datei für das WWW
Name.jpg	Bilddatei nach dem JPEG-Standard
Name.mp3	Musik nach MPEG Layer 3 kodiert
Name.mpg	Film nach MPEG-Standard kodiert
Name.o	Objektdatei (übersetzt, aber nicht gebunden)
Name.pdf	Portable-Document-Format-Datei
Name.ps	PostScript-Datei
Name.tex	Eingabedatei für TeX
Name.txt	Allgemeine Textdatei
Name.zip	Komprimiertes Archiv

Tabelle 14: typische Dateierweiterungen

5.3.6.2. Dateizugriff

Ein weiteres Kriterium ist die Art und Weise, wie ein Betriebssystem die einzelnen Dateien auf dem Speichermedium strukturiert ablegt. Dabei sind folgende Forderungen zu erfüllen:

- Die Dateigröße sollte nicht begrenzt sein (nur die physikalische Grenze des Speichermediums darf begrenzend wirken.

- Die einzelnen Bereiche der Datei sollten möglichst schnell auffindbar und für die Anwendung in den Hauptspeicher ladbar sein.

Auch in dieser Strukturierung unterscheiden sich die einzelnen Betriebssysteme wesentlich.

Sehr stark von der gewählten Art der Strukturierung der Dateien sind die Möglichkeiten des Zugriffs auf die Dateien abhängig. Bei bestimmter Strukturierung kann nur **sequenziell** zugegriffen werden. Eine Anwendung kann alle Datei in ihrer Reihenfolge als Bytes oder Records nacheinander beim Anfang beginnend einlesen. Überspringen oder Zugriffe außerhalb der Reihenfolge sind nicht möglich. Diese Strukturierung ist dann sehr komfortabel, wenn auch die Strukturierung des Speichermediums sequenziell ist, z.B. Magnetbänder u.s.w.

Voraussetzung für viele Anwendungen ist jedoch ein **wahlfreier Zugriff** auf die Bytes oder Records innerhalb der Datei, Random-Access-Dateien. Dabei treten die o.g. Forderungen sehr stark in den Vordergrund. Welche Möglichkeiten dabei entwickelt wurden, wird später beleuchtet.

5.3.6.3. Dateiattribute

Jede Datei hat einen Namen und ihre Daten. Darüber hinaus ist er ratsam, noch weitere Informationen über eine Datei abzuspeichern, z.B. Erstellungsdatum, Eigentümer oder Länge der Datei, um nur wenige zu nennen. Diese Zusatzinformationen werden auch **Dateiattribute** bezeichnet.

In Tabelle 15 sind mögliche Dateiattribute mit ihrer Bedeutung zusammengefasst, die denkbar sind. Die einzelnen Betriebssysteme benutzen jeweils immer nur einen Teil dieser Attribute.

Attribut	Bedeutung
Schutzinfos	Wer kann wie auf die Datei zugreifen
Passwort	Passwort für den Zugriff auf die Datei
Ersteller	ID des Erzeugers der Datei
Eigentümer	Aktueller Eigentümer
Read-only Flag	0: Lesen/Schreiben; 1: nur Lesen
Hidden Flag	0: normal; 1: in Listen nicht sichtbar
System Flag	0: normale Datei; 1: Systemdatei
Archive Flag	0: wurde gesichert; 1: muss noch gesichert werden
ASCII/binary Flag	0: ASCII-Datei; 1: Binärdatei
Random access Flag	0: sequentielle Datei; 1: wahlfreier Zugriff
Temporary Flag	0: normal; 1: Datei bei Prozessende löschen
Lock Flags	0: nicht gesperrt; nicht 0: Datei gesperrt
Record-Länge	Anzahl der Bytes in einem Record
Schlüsselposition	Offset des Schlüssels in einem Record
Schlüssellänge	Anzahl der Bytes in einem Schlüssel
Erstellungszeit	Datum und Zeit der Dateierstellung
Zeit des letzten Zugriffs	Datum und Zeit des letzten Zugriffs
Zeit der letzten Änderung	Datum und Zeit der letzten Dateiänderung
Aktuelle Größe	Anzahl der Bytes in einer Datei
Maximale Größe	Anzahl der Bytes für maximale Größe einer Datei

Tabelle 15: mögliche Dateiattribute

5.3.6.4. Dateioperationen

Im Sinne des objektorientierten Paradigmas sind Dateien Objekten mit den dazu definierten Methoden, die im Falle von Dateien Dateioperationen genannt werden. Die folgenden Dateioperationen (Systemaufrufe) sind denkbar, aber stark mit der Dateistruktur verbunden und von dieser abhängig:

Create Die Datei wird ohne Daten erzeugt. Der Sinn und Zweck dieses Aufrufes besteht darin, für die kommenden Daten den Speicherraum vorzubereiten. Gleichzeitung werden die Dateiattribute auf Defaultwerte gesetzt.

Delete Wird die Datei nicht länger benötigt, wird sie gelöscht, um den von ihr belegten Platz auf dem Speichermedium wieder freizugeben.

Open Vor der Nutzung einer Datei muss die Anwendung diese öffnen. Damit verbunden werden durch das Betriebssystem die Dateiattribute in den Hauptspeicher geladen um bei folgenden Zugriffen auf die Datei performanter zugreifen zu können.

Close Sind alle Zugriffe auf die Datei beendet, werden die Dateiattribute nicht mehr länger benötigt und können gelöscht werden, um Hauptspeicher freizugeben. Änderungen in den Dateiattributen und der Datei, z.B. der letzte Block werden vorher auf das Speichermedium geschrieben.

Read Die Daten werden aus der Datei gelesen. Gewöhnlich werden die Daten von der aktuellen Position beginnend gelesen. Dazu muss dieser Operation angegeben werden, wie viele Daten (Bytes oder Records) gelesen werden sollen und in welchen Pufferbereich im Hauptspeicher die Daten abgelegt werden sollen.

Write Die Daten werden in die Datei geschrieben. Gewöhnlich an der aktuellen Position. Ist die aktuelle Position am Dateiende, so erhöht sich die Dateigröße. Ansonsten werden die Daten in die Datei geschrieben, wobei andere Daten an dieser Stelle überschrieben werden und verloren gehen.

Append Eingeschränkte Write-Operation. Es wird immer am Dateiende geschrieben.

Seek Für Dateien mit wahlfreiem Zugriff wird diese Operation benötigt um die aktuelle Position für Read oder Write definiert zu verändern.
Get Attributes Die aktuelle Belegung der Dateiattribute wird erneut in den Hauptspeicher gelesen.
Set Attributes Die aktuelle Belegung der Dateiattribute im Hauptspeicher wird auf das Speichermedium geschrieben.
Rename Der Dateiname wird verändert, ohne Datei die Daten zu kopieren oder sonst wie zu verändern.

5.3.6.5. Verzeichnisse

Da mit der ständig steigenden Speicherkapazität der Speichermedien nicht nur die mögliche Größe einer Datei ständig anwächst, sondern auch die Anzahl von Dateien pro Speichermedium, bedarf es einer gewissen Ordnung. Diese wird durch Verzeichnisse oder Ordner erreicht. Die einfachste Form der Organisation sieht pro Speichermedium nur ein Verzeichnis vor, in dem alle Dateien des Datenträgers mit ihren Dateinamen und –attributen verzeichnet sind. Meistens ist in diesen Fällen die Anzahl Einträge begrenzt.

Eine weitaus flexiblere Organisation liegt bei einer Baumstruktur der Verzeichnisse vor. Dabei kann organisiert werden, dass pro Verzeichnis beliebig viele Einträge vorgenommen werden können und es auch beliebig viele Verzeichnisse auf dem Speichermedium geben kann. Da es bezüglich der Lage der Verzeichnisse zueinander dabei eine strenge Ordnung gibt spricht man in diesem Fall von einem hierarchischen Verzeichnissystem bzw. einem hierarchischen Dateisystem.

5.3.6.6. Realisierung von Dateien

Der wichtigste Aspekt bei der Realisierung eines Dateisystems ist die Organisationsform einer Datei auf dem Speichermedium. D.h., welche Plattenblöcke werden durch eine Datei belegt. Dabei sind die Forderungen aus Abschnitt 5.3.6.2. Dateizugriff in besonderer Weise zu beachten.

Die wohl einfachste Form der Abspeicherung einer Datei auf einem Speichermedium ist die zusammenhängende Belegung, wie sie in Abbildung 95 dargestellt wird. Dabei ist als Beispiel die Plattenbelegung von sieben Dateien zu sehen (a). In (b) sind die Dateien D und F gelöscht worden und hinterlassen freie Bereiche auf dem Speichermedium.

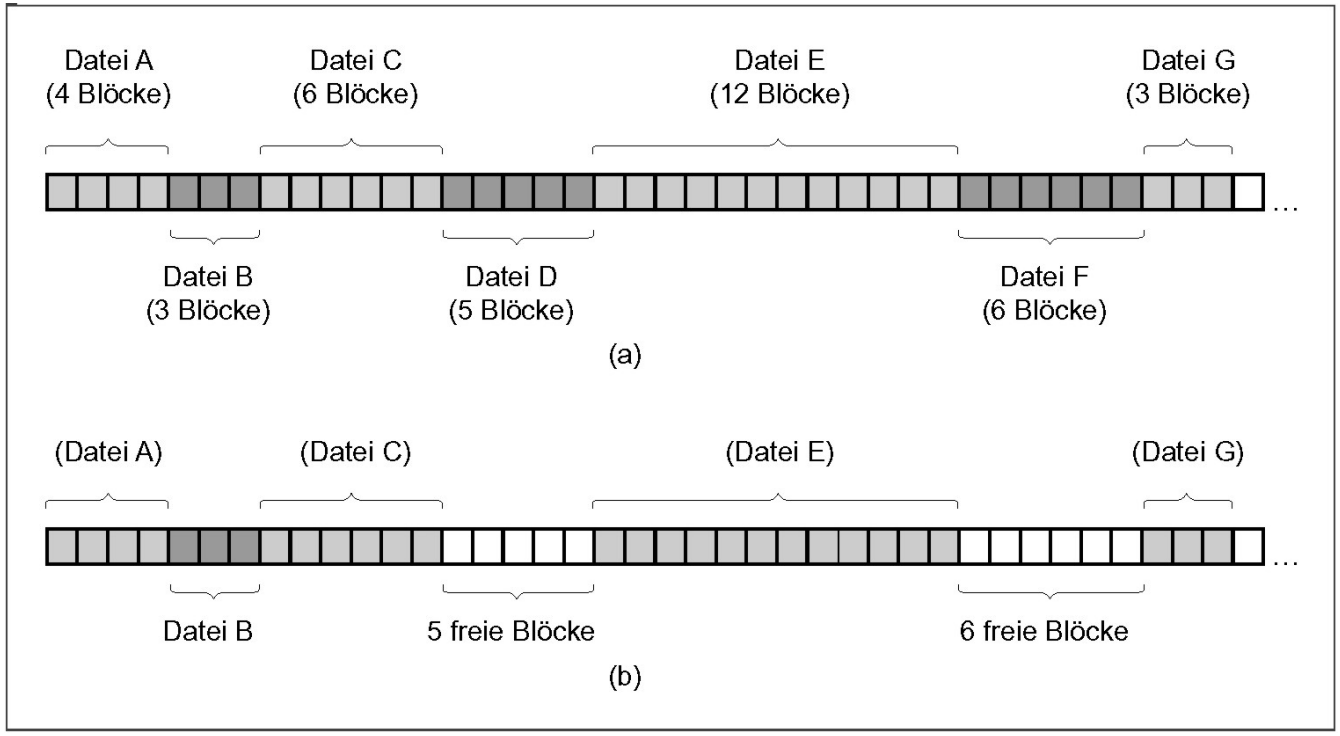
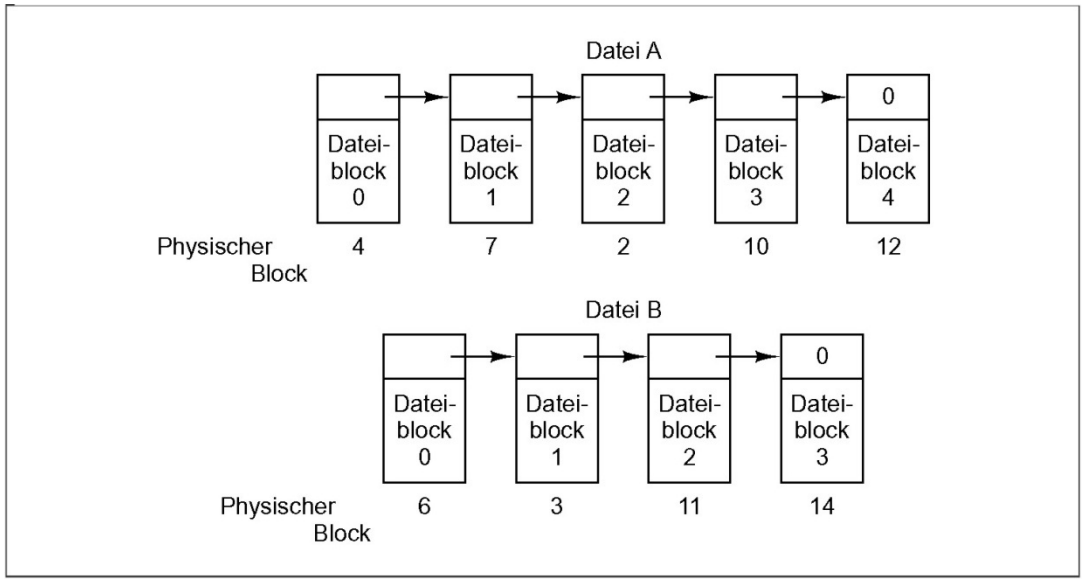


Abbildung 95: Zusammenhängende Belegung

Die Problematik der Arbeitsweise liegt bei dieser Speicherform auf der Hand. Die Verwaltung der freien Bereiche und deren Wiederverwendung für neue Dateien ist entweder durch Verschieben der noch auf dem Speichermedium befindlichen Dateien, hier Datei E und G, was in der Praxis undenkbar ist. Oder es müssen die freien Bereiche in einer Datenstruktur verwaltet werden, um neue Dateien an die passende Stelle zu platzieren, das wiederum voraussetzt, dass deren Dateigröße vorher bekannt sein muss.

Eine weitere Möglichkeit stellt die Belegung durch verkettete Listen dar. Dabei wird im ersten Fall in jedem Plattenblock an dessen Beginn ein Verweis auf den folgenden Block abgelegt, wodurch sich der restliche Platz im Plattenblock um 3, besser 4 Byte verringert und damit keine Potenz von Zwei ist. Die meisten Anwendungen lesen und schreiben aber Daten in Potenzen von Zwei, was bei dieser Speicherbelegung zusätzlichen Aufwand mit sich bringt.



Im Beispiel in Abbildung 96 belegt die Datei A die Plattenblöcke 4, 7, 2, 10 und 12 und die Datei B die Plattenblöcke 6, 3, 11 und 14.

Abbildung 96: verkettete Listen

Auch diese Form der Dateiabspeicherung auf einem Speichermedium ist nicht praktikabel und wurde weiterentwickelt. Die Form der verketteten Liste bleibt bestehen aber diese wird nicht zusammen mit den Daten in den Plattenblöcken abgelegt, sondern in einer Tabelle im Hauptspeicher bzw. im Verzeichnis. Diese Tabelle wird allgemein bekannt als **FAT** (**F**ile **A**llocation **T**able) bezeichnet. Diese Organisationsform wurde erstmals durch **Bill Gates** in MS-DOS eingebracht. Spricht man dabei von der **FAT**, so ist die ursprüngliche Form als **FAT-16** gemein. 16 bedeutet dabei die Anzahl Bits (16) zur Adressierung der einzelnen Plattenblöcke bzw. Cluster.

Damit ist relativ leicht die Begrenzung der Speichergröße eines Speichermediums durch die **FAT-16** auf 65536 Plattenblöcke mit jeweils 512 Byte = 32 Mbyte bzw. 256 Mbyte bei 4 Kbyte Clustergröße erkennbar. Für die Speichermedien der ersten Personalcomputer war das sicherlich ausreichend, musste aber im Laufe der Zeit mehrfach verändert werden. Die heutigen Windows-Betriebssysteme verwenden als ein Dateisystem die **FAT-32**.

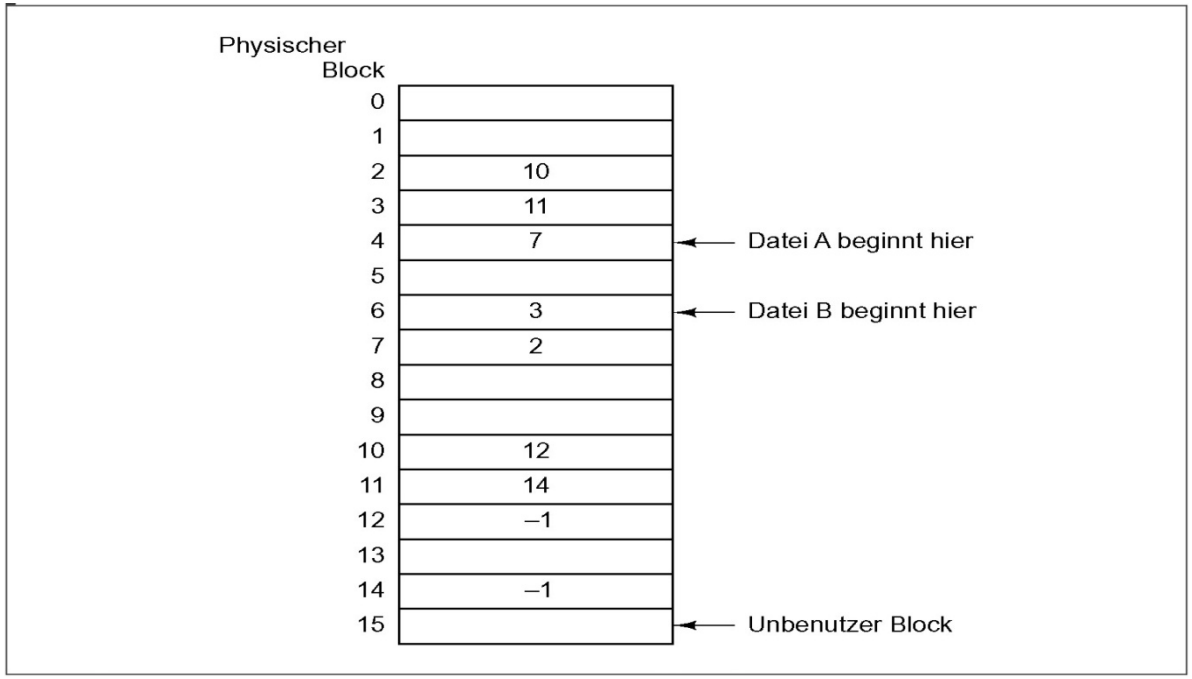


Abbildung 97: FAT - File Allocation Table

In Abbildung 97 sind die beiden Dateien A und B mit den gleichen Plattenblöcken enthalten wie in Abbildung 96.

Ein deutlich ältere, aber wesentlich intelligentere Speicherorganisation der Speichermedien stellt das Unix-Betriebssystem mit der Inode-Struktur bereit.

Anwendungsprogramme nutzen ausschließlich die Routinen des Dateisystems, um Dateien anzulegen, zu lesen, zu schreiben, zu löschen oder um Zugriffsrechte zu verändern. Nur so ist es möglich, dass Dateien, die von einem Anwendungsprogramm erzeugt wurden, von einem anderen weiterverarbeitet werden können. Das Dateisystem verwaltet die Namen der Dateien in so genannten Verzeichnissen (Directory).

Attribute der Dateien (Länge, Erstellungsdatum, Zugriffsrechte usw.), können in den Verzeichniseinträgen oder aber in separaten Listen abgelegt sein. Jedes Dateisystem verwaltet den verfügbaren Speicherbereich zur Ablage von Dateien (Freispeicherliste, Freiblockliste) ebenso, wie Informationen und die Größe und Lage des Wurzelverzeichnisses (root directory).

Informationen zur Freispeicherverwaltung und weitere Informationen werden im Kopf des Dateisystems abgelegt.

5.3.6.7. Das (physikalische) Unix-Dateisystem

Aus Anwendersicht erscheint der Unix-Dateibaum als eine homogene Struktur. Auf physischer Ebene setzt er sich jedoch aus mehreren Teilbereichen zusammen, den so genannten physischen Dateisystemen.

Ein physisches Dateisystem ist eine dateiorientierte Struktur auf einem logischen Datenträger (**Partition**, **Slice**). Unix bietet die Möglichkeit, mehrere dieser logischen Datenträger auf einem physischen Datenträger (**Festplattenlaufwerk**) zu verwalten. Eine mögliche Unterteilung ist in Abbildung 98 dargestellt.

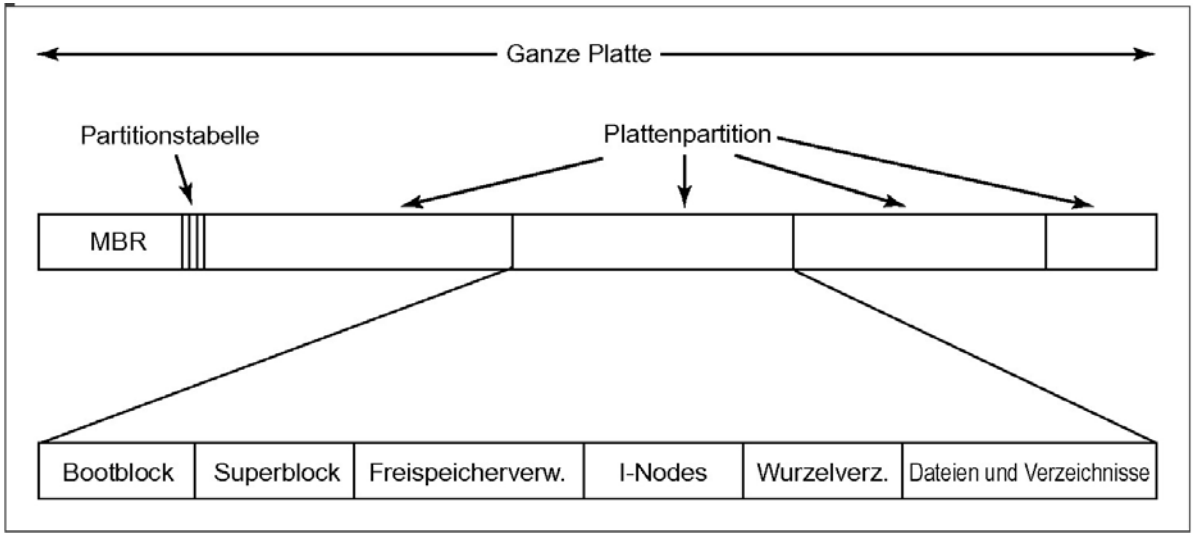


Abbildung 98.: Unterteilung einer Festplatte in Partitionen

Des Weiteren können neben den resistenten physischen Dateisystemen (Dateisysteme auf dem Systemlaufwerk) auch nicht resistente Dateisysteme auf montierbaren Datenträgern (Festplattenlaufwerke anderer Rechnersysteme, Disketten usw.) in den Dateibaum eingehängt bzw. entfernt werden. Dadurch wird ein hoher Grad an Flexibilität in der Systemarchitektur erreicht, der dem Anwender weitestgehend verborgen bleibt.

Schaut man sich ein physisches Dateisystem genauer an, so erkennt man den **Betriebssystemblock** als kleinste Einheit (im Bereich von **512 Byte** bis **16 KByte**). Das Betriebssystem legt nun über diese (durchnummerierte) Folge von **Blöcken** ein höheres Ordnungsschema, wodurch ein physisches Dateisystem in **vier Bereiche** mit unterschiedlicher Funktionalität aufgespaltet wird:

1. **Boot-Block**
   Er enthielt im **root-Dateisystem** einen Urlader-Programm, das das eigentliche Unix-System (Betriebssystemkern) in den Arbeitsspeicher lädt. Dieser Bereich ist bei heutigen Systemen leer.

2. **Super-Block**
   Er beinhaltet alle relevanten Verwaltungsinformationen zu einem physischen Dateisystem:
   - Name und Größe des physischen Dateisystems
   - Größe der nachfolgenden Bereiche des physischen Dateisystems (Inodeliste, Nutzdatenbereich)
   - Verweise auf die Freiblocklisten (Liste der freien Datenblöcke und Liste der freien Inodes)
   - Datum der letzten Sicherung und Modifikation
   - und weitere Angaben.

3. **Inodeliste**
   Sie stellt ein Inhaltsverzeichnis aller in dem Dateisystem existierenden Dateien dar. Sie besteht aus einer Folge von Inodes (Dateiköpfen), die die Verwaltungsdaten zu jeder Datei enthalten.

4. **Nutzdatenbereich**
   Hier befinden sich die freien Blöcke, Datenblöcke (Inhalte von Dateien und Verzeichnissen) und Referenzblöcke, die zur Adressierung der Datenblöcke eingesetzt werden.

Die Größe der einzelnen Bereiche wird bei der Initialisierung eines physischen Dateisystems spezifiziert und kann im Nachhinein nicht mehr dynamisch verändert werden (außer im Unix-System **AIX** von **IBM**).

Das dazu notwendige Kommando des Systemadministrators ist **mkfs** make file **s**ystem.

Die Bereiche eines physischen Dateisystems sind auf die Kapazität eines logisch Datenträgers und damit maximal auf die Gesamtkapazität eines Festplattenlaufwerks beschränkt, d.h. festplattenübergreifende physische Dateisysteme sind nicht möglich.

Die Inodeliste besteht aus einer Folge von Inodes, die durch einen eindeutigen Index, der Knotennummer, gekennzeichnet werden.

Da alle Inodes die identische Länge von 128 Byte haben, erübrigt sich die Abspeicherung der Knotennummer innerhalb des Inodes.

Ein Inode enthält die relevanten Verwaltungsattribute einer Datei:

1. Dateityp und Zugriffsrechte
2. Referenzzähler (**Linkcounter**)
3. Benutzernummer (**UID**) und Gruppennummer (**GID**)
4. Dateigröße in Byte
5. Zeitstempel
   - Erstellungsdatum
   - Datum der letzten Modifikation
   - Datum des letzten Zugriffs
6. Blockadressen der ersten zehn Datenblöcke
7. Blockadressen der Indirektionsblöcke

Zu beachten ist, dass der Name der Datei nicht aufgeführt wird.

Diese Systemarchitektur ermöglicht es, unter mehreren (verschiedenen) Namen als Einträge von Verzeichnissen die gleiche Datei (genauer gesagt Inode und Dateiinhalt) anzusprechen.

Der Inode fungiert somit als Bindeglied zwischen dem Namen und dem Inhalt einer Datei. Für die Adressierung der Inhalte einer Datei ergibt sich eine bestimmte Verweisstruktur.

Unter der Voraussetzung, dass eine Blockgröße von 512 Byte betrachtet wird, ergibt sich eine mögliche Dateistruktur und -größe, wie sie in Abbildung 99 dargestellt ist und liegt bei 2.113.673 Blöcken.

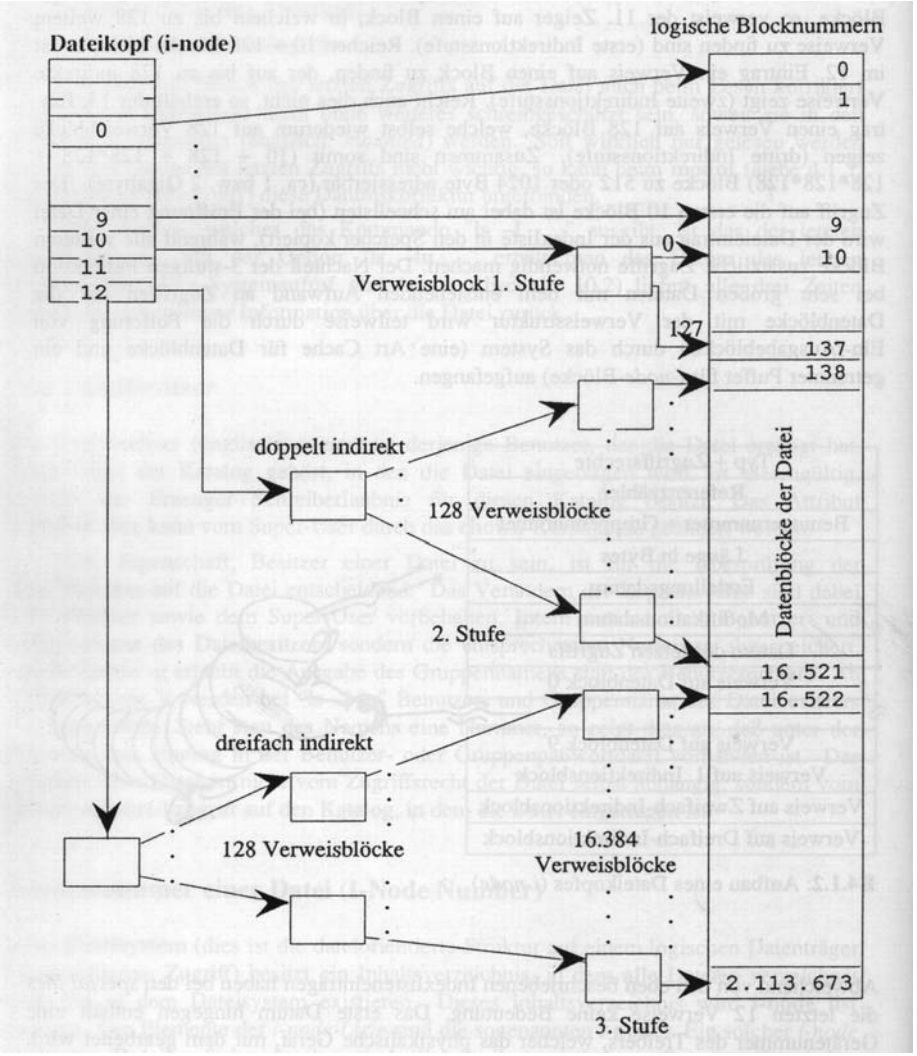


Abbildung 99.: Inode-Verweisstruktur

5.3.7. Das Speicherverwaltungssystem

Speicher ist eine wichtige Ressource, die sorgfältig verwaltet werden muss. Jeder Programmierer hätte am liebsten einen unendlich großen, unendlich schnellen Speicher, der auch noch nicht flüchtig (nonvolatile) ist, dessen Inhalt also nicht verloren geht, falls die Stromversorgung ausfällt.

> Von dem Mitgründer der Firma Microsoft **Bill Gates** stammt der Ausdruck ***„... ich kann mir nicht vorstellen, dass eine Anwendung mehr als 640 KByte Hauptspeicher benötigt. ...“***

Leider funktionieren reale Speicher so nicht. Aus diesem Grund haben die meisten Computer deshalb eine Speicherhierarchie, siehe 2.1.2. Hauptspeicher.
An der Spitze stehen die Register der CPU und ein sehr kleiner, sehr schneller, teurer und flüchtiger Cache-Speicher. Die nächsten Stufen bilden der einige hundert bis tausende Mbyte große mittelschnelle Hauptspeicher (**RAM**) und ein langsamer, billiger und nichtflüchtiger Plattenspeicher, der einige hundert Gbyte groß sein kann. Das Betriebssystem hat die Aufgabe, die Nutzung dieser verschiedenen Speicher zu koordinieren:

- Register,
- Cache-Speicher,
- Hauptspeicher (RAM) und
- Externe Speichermedien.

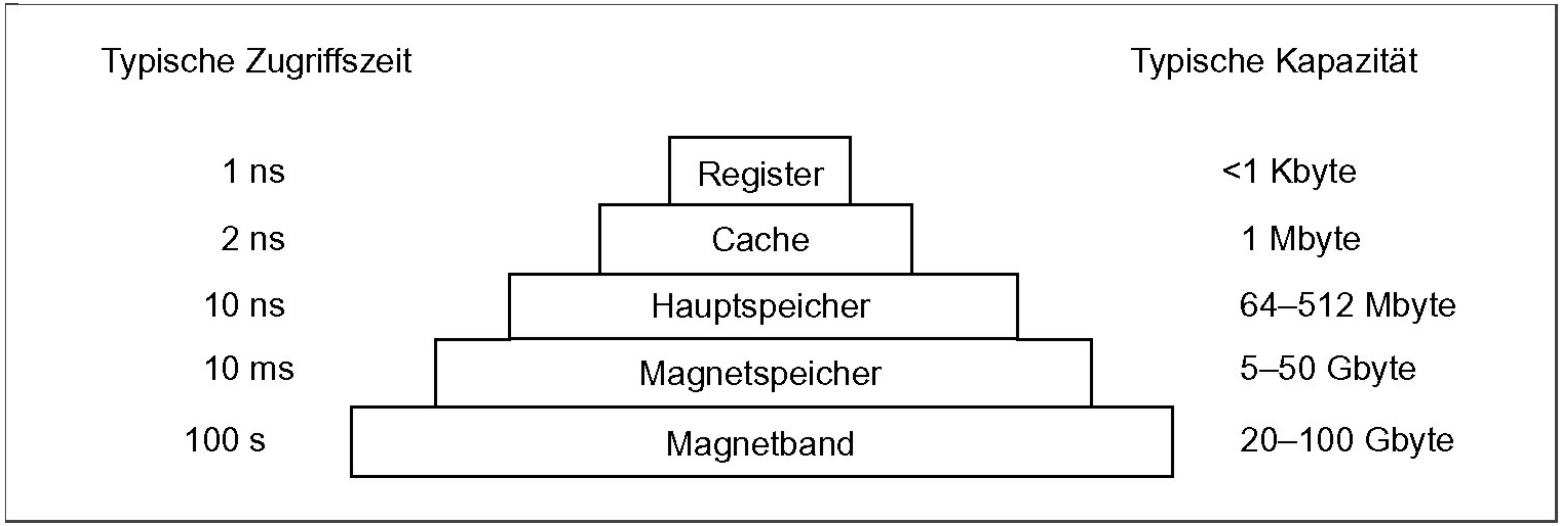


Abbildung 100: Speicherhierarchie

Die in Abbildung 100 enthaltenen Werte sind sehr grobe Schätzungen und werden sich durch die stetige Entwicklung auch ständig zu größeren Speicherkapazitäten und geringeren Zugriffszeiten verschieben.

a) Grundlagen der Speicherverwaltung

Es gibt zwei Klassen von Speicherverwaltungssystemen:

- Speicherverwaltung nur im Hauptspeicher und
- Speicherverwaltung des Hauptspeichers mit zu Hilfenahme von Speichermedien (**Swapping** und **Paging**)

Die einfachste Strategie ist, nur ein Programm laufen zu lassen. Verschiedene Anwendungen werden nacheinander, sequentiell bearbeitet. Verschiedene Varianten der Aufteilung des Hauptspeichers zwischen dem Betriebssystem und der Anwendung sind denkbar, siehe Abbildung 101.

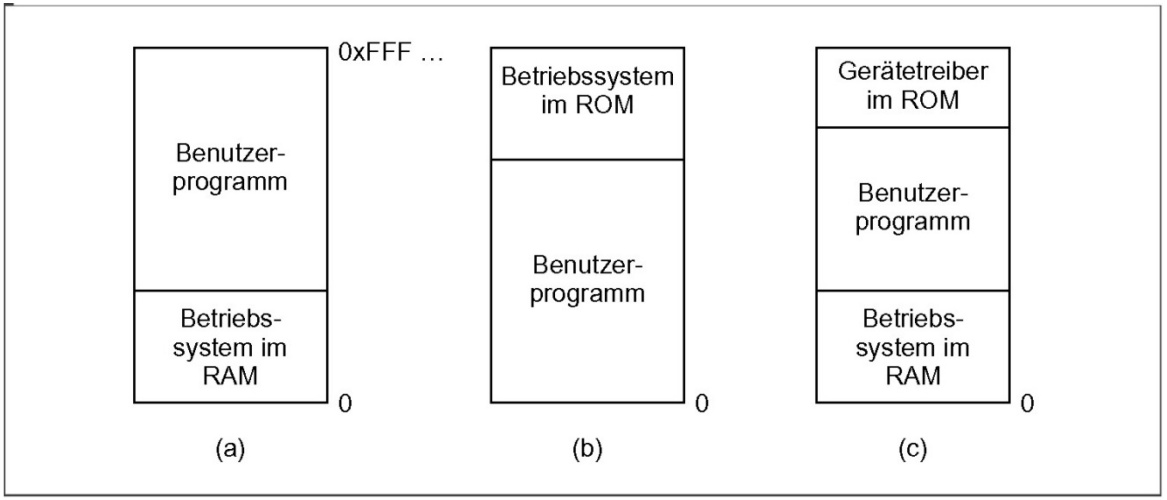


Abbildung 101: Aufteilung des Hauptspeichers zwischen Betriebssystem und Anwendung

Die Variante **(a)** wurde früher in Mainframes und Minicomputern verwendet und wird heute nicht mehr eingesetzt. Variante **(b)** wird in einigen Palmtops und eingebetteten Systemen benutzt. Die Variante **(c)** wurde von frühen Personalcomputern (z.B. unter MS-DOS) verwendet. Der Teil des Systems im ROM wird **BIOS** (Basic Input Output System) genannt.

In einem so strukturierten System kann immer nur eine Anwendung laufen. Sobald der Benutzer einen Befehl eingegeben hat, lädt das Betriebssystem das entsprechende Programm vom Speichermedium in den Hauptspeicher und startet es.

Die meisten modernen Betriebssysteme können mehrere Prozesse gleichzeitig ausführen. Dabei kann ein Prozess bearbeitet werden, wenn ein anderer z.B. auf eine Ein- / Ausgabeoperation wartet. Die einfachste Art dieser Multiprogrammierung ist, den Hauptspeicher einfach in n (möglicherweise verschieden große) Bereiche aufzu-

teilen. Diese Einteilung kann beim Systemstart in Abhängigkeit von der zur Verfügung stehenden Hauptspeichergröße erfolgen.

Wenn ein Prozess gestartet wird, kann er an eine Warteschlange für den kleinsten Hauptspeicherbereich abgehängt werden, der für diesen Prozess groß genug ist. In Abbildung 102 sind die Verwaltungsmöglichkeiten der festen Hauptspeicherbereiche dargestellt. In Variante **(a)** existiert für jeden Speicherbereich eine eigene Warteschlange, während in Variante **(b)** alle Speicherbereiche über eine Warteschlange verwaltet werden.

Dieses System mit festen Hauptspeicherbereichen wurde viele Jahre lang von **IBM OS/360** auf den großen Mainframes verwendet und heißt **MFT** (**M**ultiprogramming with a **f**ixed number of **T**asks)

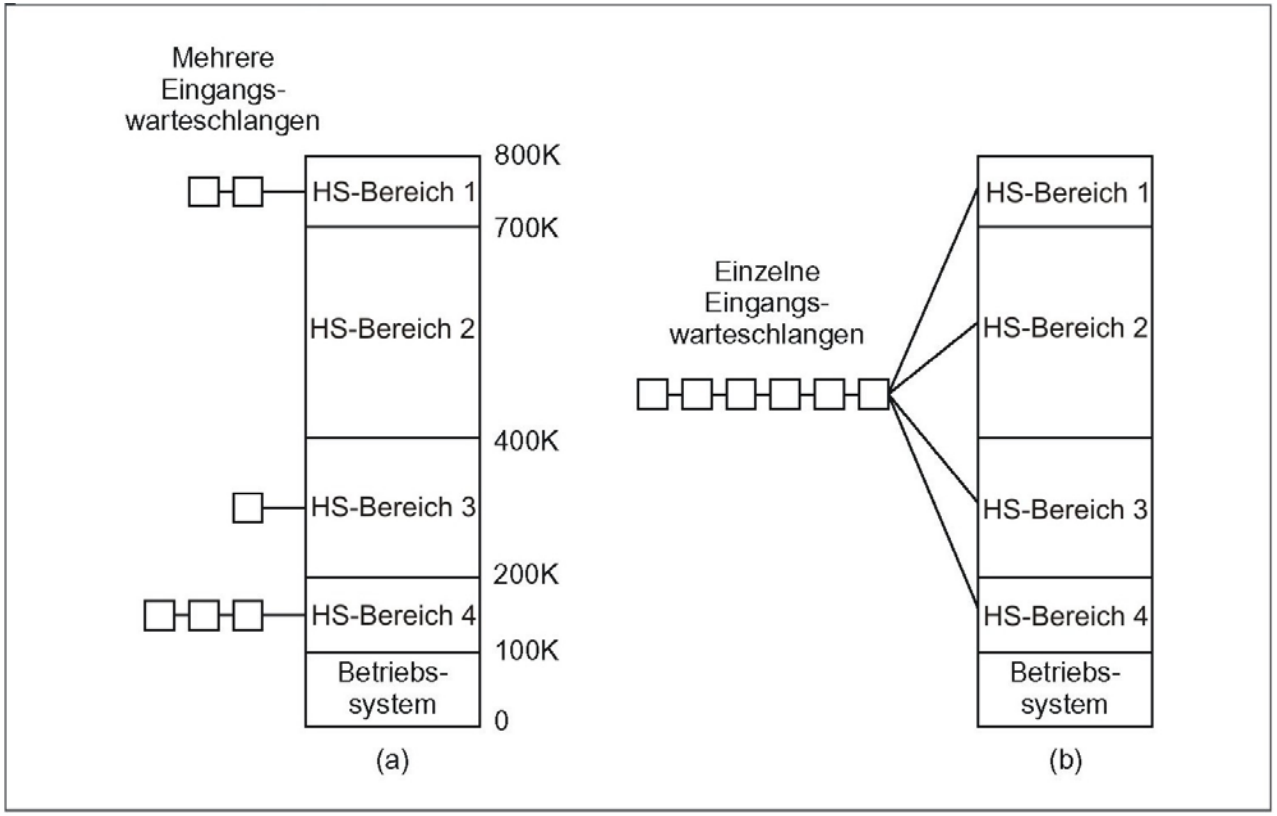


Abbildung 102: feste Hauptspeicherbereiche mit verschiedenen Warteschlangenverwaltungen

5.3.7.2. Swapping

Für Timesharing-Systeme oder Personalcomputer mit grafischer Benutzeroberfläche reicht die Speicherverwaltung mit festen Hauptspeicherbereichen nicht aus. Es kann vorkommen, dass der vorhandene Hauptspeicher

nicht für alle aktiven Prozesse ausreicht. Deshalb müssen einige der Prozesse auf ein Speichermedium ausgelagert werden und bei Bedarf dynamisch wieder in den Hauptspeicher zurückgeholt werden.

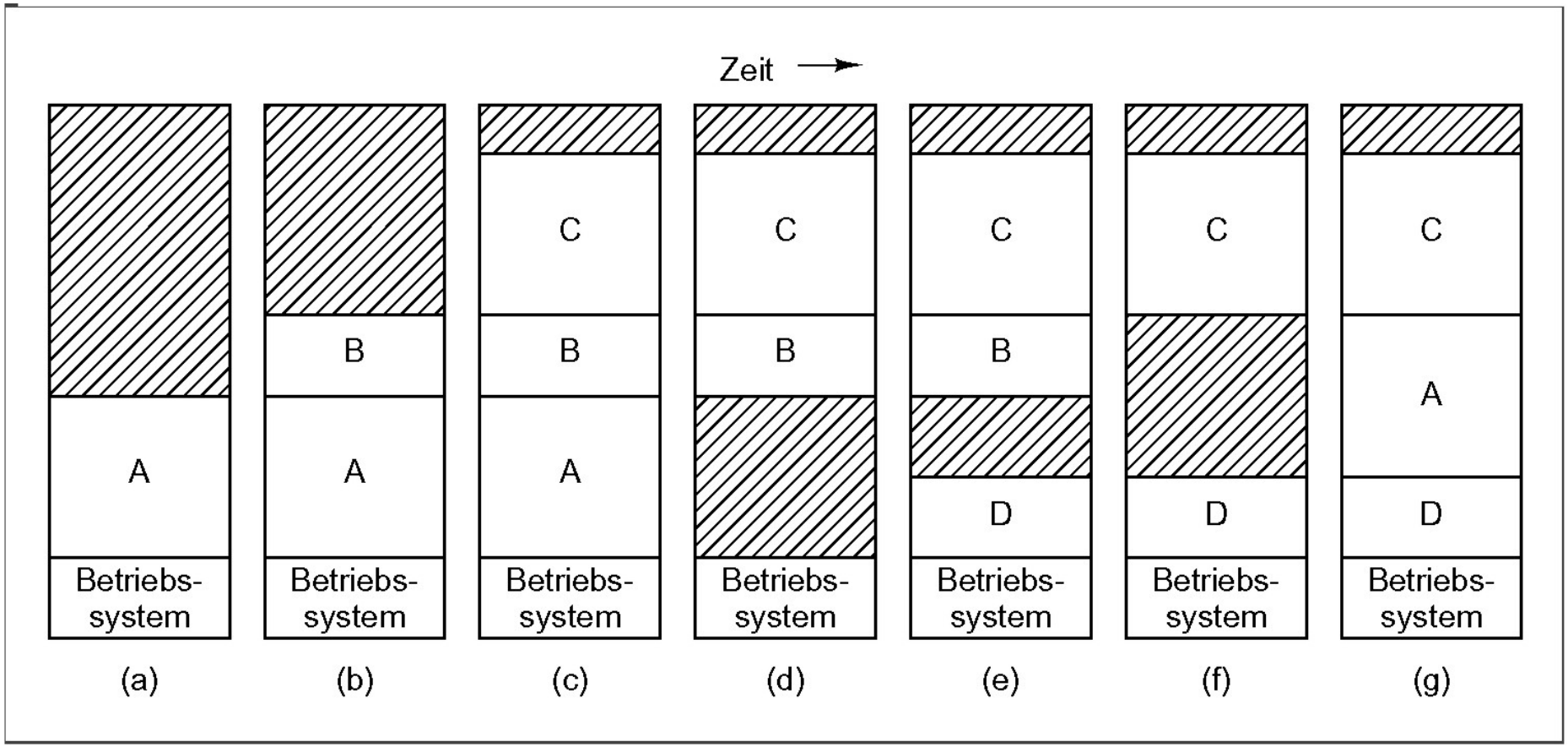


Abbildung 103: Swapping

Für diese Art von Speicherverwaltung gibt es zwei grundlegende Ansätze. Für welche man sich entscheidet, hängt teilweise von der verfügbaren Hardware ab. Die einfachste Strategie ist das sogenannte Swapping, siehe Abbildung 103, bei dem jeder Prozess komplett in den Hauptspeicher geladen wird, eine gewisse Zeit laufen darf und anschließend wieder auf das Speichermedium ausgelagert wird.

Bei der anderen Strategie, dem virtuellen Speicher, können auch dann Programme laufen, wenn sich nur ein Teil von ihnen im Hauptspeicher befindet, siehe Abschnitt 5.3.7.4. Paging.

Der Hauptunterschied zwischen den festen Hauptspeicherbereichen aus Abbildung 102 und den variablen Hauptspeicherbereichen aus Abbildung 103 besteht darin, dass Anzahl, Größe und Ort der Speicherbereiche hier nicht feststehen, sondern sich dynamisch ändern können. Diese Flexibilität verbessert die Speicherausnutzung, weil feste Speicherbereiche zu klein oder zu groß sein können, aber sie machen auch die Zuteilung, Freigabe und Verwaltung des Speichers komplizierter.

Eine wichtige Frage ist auch, wie viel Speicher für einen Prozess reserviert werden soll, wenn er gestartet oder wieder eingelagert wird. Bei fester Speichergröße ist diese Verwaltung trivial.

Wenn man jedoch davon ausgeht, dass die meisten Prozesse während ihrer Laufzeit wachsen, ist es sinnvoll immer etwas mehr Hauptspeicher zu reservieren, wenn ein Prozess eingelagert wird. Dadurch lässt sich der zusätzliche Aufwand vermeiden, wenn ein Prozess nicht mehr in seinen Speicherbereich passt. Wenn ein Prozess ausgelagert wird, sollte natürlich nur der wirklich benutzte Hauptspeicher auf das Speichermedium geschrieben werden.

In Abbildung 104 (a) ist eine Speicherkonfiguration zu sehen, in der für zwei Prozesse Platz reserviert wurde.

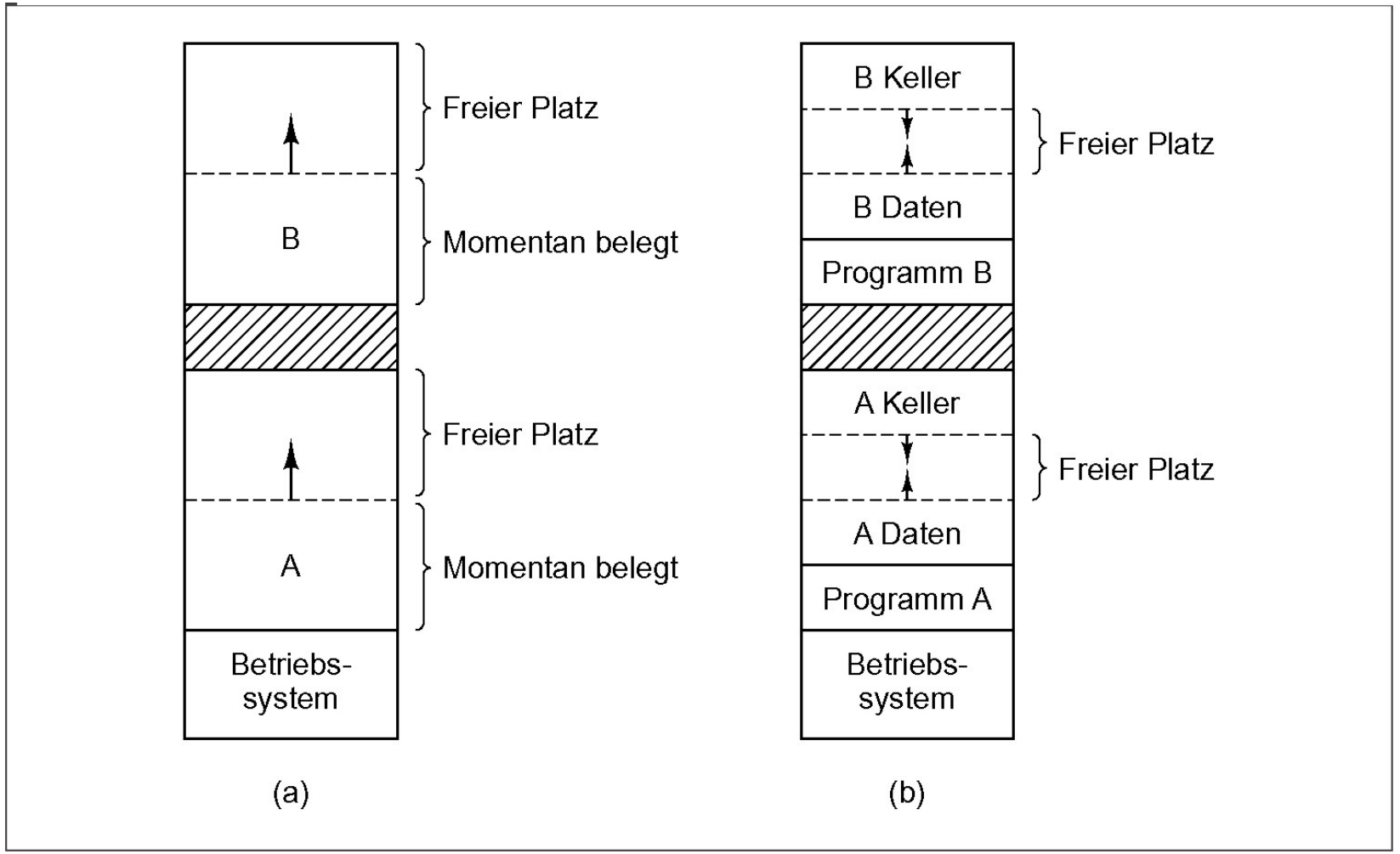


Abbildung 104: Reservierung von Hauptspeicherplatz

Wenn ein Prozess zwei wachsende Segmente haben kann, zum Beispiel ein Daten- und ein Kellersegment, dann liegt es nahe, die Anordnung aus Abbildung 104 (b) zu verwenden. Der freie Speicher in der Mitte kann für jedes der beiden Segmente genutzt werden. Wenn er nicht ausreicht, müssen die Prozesse in ein größeres Loch verschoben, ausgelagert oder abgebrochen werden.

5.3.7.3. virtueller Speicher

Vor langer Zeit stießen Programmierer zum ersten Mal auf das Problem, dass die Programme zu groß für den verfügbaren Speicher wurden. Normalerweise lösten sie dieses Problem, indem sie die Programme in mehrere Teile aufspalteten, so genannte **Overlays**. Zunächst wurde Overlay 0 ausgeführt, das dann, sobald es fertig war, ein anderes Overlay aufrief.

Obwohl die Overlays vom Betriebssystem verwaltet wurden, musste der Programmierer das Programm selbst aufteilen. Große Programme in kleine, modulare Teile aufzuspalten, war eine zeitaufwändige und langweilige Arbeit. Deshalb wurde diese Aufgabe bald durch den Computer erledigt.

Diese Methode wurde als **virtueller Speicher** bekannt. So kann man beispielsweise ein 16 Mbyte großes Programm auf einer Maschine mit 4 Mbyte Hauptspeicher ausführen, indem man die 4 Mbyte, die im Hauptspeicher liegen, zu jedem Zeitpunkt sorgfältig auswählt und Teile des Programms nach Bedarf ein- und auslagert.

Virtueller Speicher funktioniert auch für Systeme mit Multiprogrammierung. Dabei können dann Teile von mehreren verschiedenen Programmen gleichzeitig im Hauptspeicher liegen.

5.3.7.4. Paging

Die meisten Systeme mit virtuellem Speicher verwenden eine Technik namens **Paging**. Für jeden Computer gibt es eine Menge von Speicheradressen, die Programme erzeugen können. Adressen können u.a. mit Hilfe von Indizierung, Basisregistern und Segmentregistern generiert werden.

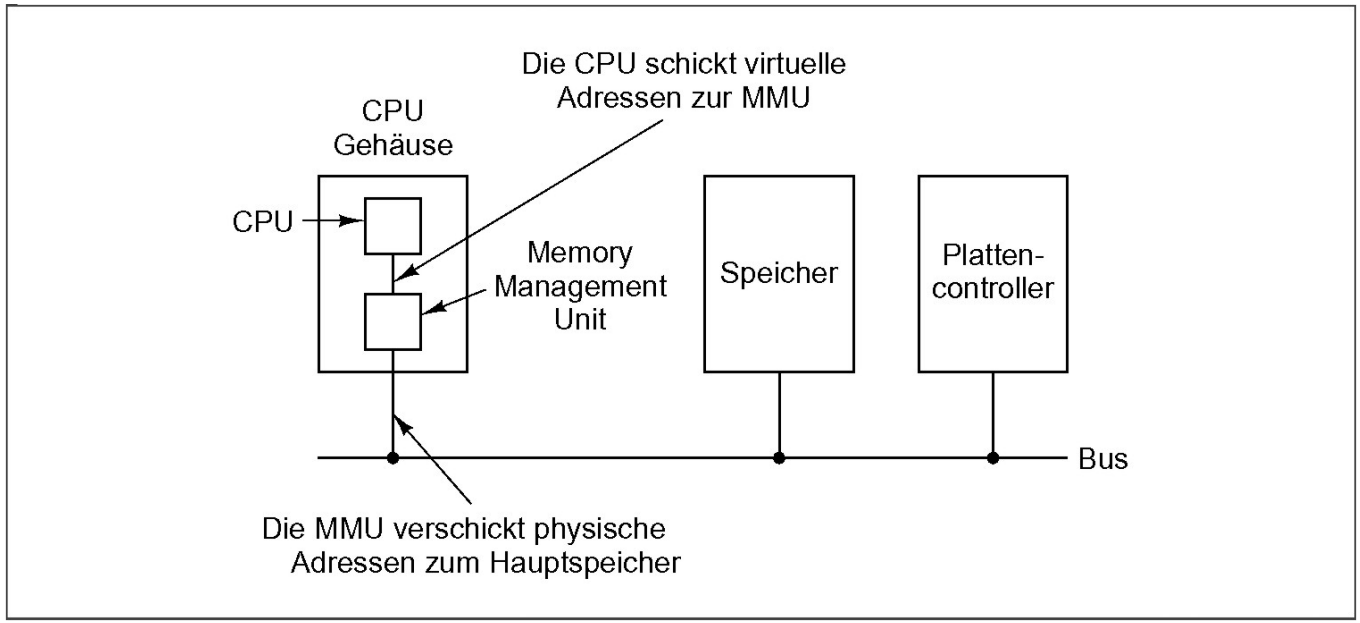


Abbildung 105: Funktion der MMU

Diese vom Programm generierten Adressen heißen virtuelle Adressen und bilden den virtuellen Adressraum. Bei Rechnern mit virtuellem Speicher gehen die Adressen nicht direkt an den Hauptspeicher, sondern an die MMU (Memory Management Unit), die die virtuelle Adresse auf eine physische Adresse abbildet, Abbildung 105.

Tabellenverzeichnis

Abbildungsverzeichnis

Definitionsverzeichnis

Index