HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCE

LEHRBRIEF

für das Modul

Betriebssysteme

Betriebssytemarchitektur

Autor: Prof. Ronny Bodach

Bearbeitungsstand: 25.05.2023

Hinweise

Herausgeber:

Hochschule Mittweida - University of Applied Sciences
Fakultät Computer- und Biowissenschaften
Technikumplatz 17
09648 Mittweida
1. Auflage (25.05.2023)

Redaktionelle Bearbeitung: Prof. Ronny Bodach

Inhaltsverzeichnis

1 Grundlagen Betriebssysteme / Betriebssystemarchitektur 4

1.1 Betriebssystemarchitektur 4

1.1.1 Einführung – Grundlegender Aufbau von Rechnern 4

1.1.2 Klassifikationsprinzipien 4

1.1.3 Leistungserhöhung 4

1.1.4 Von-Neumann-Rechnerarchitektur 5

1.1.5 Harvard-Rechnerarchitektur 5

1.1.6 Moderne Rechnerarchitektur 6

1.2 Prozesse, Tasks und Threads 6

1.2.1 Vom Programm zum Prozess 6

1.2.2 Mehrprozessbetrieb (Multitasking) 7

1.2.3 Mehrbenutzerbetrieb (*Multi-User*) 7

1.2.4 Kooperatives Multitasking 7

1.2.5 Präemptives Multitasking 7

1.3 Computerarchitektur nach Tanenbaum 7

1.3.1 Transistorebene 8

1.3.2 Logische Ebene 9

1.3.3 Microarchitektur (RISC, CISC) 15

1.3.4 Instruction Set Architecture 16

1.3.5 Betriebssystem 17

1.3.6 Betriebssystemarchitekturen 22

1.3.7 Prozesskontextwechsel (Process Context Switch) 28

1.3.8 Prozesskontrollblock (Process Control Block; PCB) 28

1.3.9 Prozesszustände 28

1.3.10 Zustandsdiagramm der Prozesszustände 29

1.3.11 Scheduling Auswahlstrategien 29

1.3.12 Multitasking für mehrere Prozesse einer Anwendung 32

1.3.13 Threads 33

1.4.13 Parallelität und Nebenläufigkeit 34

1.4.14 Wechselseitiger Ausschluss 36

1.4.15 Semaphore 38

1.5 Speicherverwaltung und Speicherzugriffe 40

1.5.12 Speicherverwaltung 40

1.5.13 E/A Zugriffe auf die Festplatte 51

1.6 Paketverwaltung 52
1.6.1 Umsetzung der Paketverwaltung in den jeweiligen Betriebssystemen 52
1.6.2 Paketverwaltung unter unixioden Betriebssystemen 53
1.6.3 Aufbau eines Paket-Containers am Beispiel eines .deb Paketes 54
1.6.4 Build-System 54

1 Grundlagen Betriebssysteme / Betriebssystemarchitektur

1.1 Betriebssystemarchitektur

1.1.1 Einführung – Grundlegender Aufbau von Rechnern

Bisher haben wir uns einen Einblick in die Partitionierung und die ersten Dateisysteme verschafft. Dieses ist jedoch nur ein Bestandteil in moderneren Computersystemen. Für die korrekte Funktion moderner Computer sind wesentlich mehr Bestandteile notwendig. Der Kernbestandteil ist das Betriebssystem.

Daher schauen wir uns nunmehr den grundlegenden Aufbau von Rechnern an, beginnend mit der Klassifizierung von Architekturen (Rechnerarchitektur).

1.1.2 Klassifikationsprinzipien

Die Klassifikation kann zum einen nach dem Rechenprinzip erfolgen:

- Von Neumann (Steuerfluss)
- Harvard-Architektur
- Datenfluss (Zündregel, Petrinetze)
- Reduktion (Funktionsaufruf)
- Objektorientiert (Methodenaufruf)

Zum anderen kann die Klassifikation auch nach dem Architektur-Grundkonzept erfolgen:

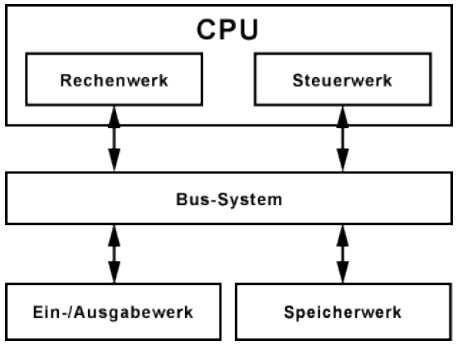
- Vektorrechner (Pipeline)
- Array-Computer (Data-Array)
- Assoziativ-Rechner (Assoziativ-Speicher)

1.1.3 Leistungserhöhung

Gebiet	Maßnahme
Architektur	Pipelines, Superskalarität, Spekulative Ausführung, Caches, Busbreite
Optimierung von Software	Compileroptimierung, Parallelisierung, effizientere Algorithmen
Siliziumbasis	Transistordichte und Taktraten

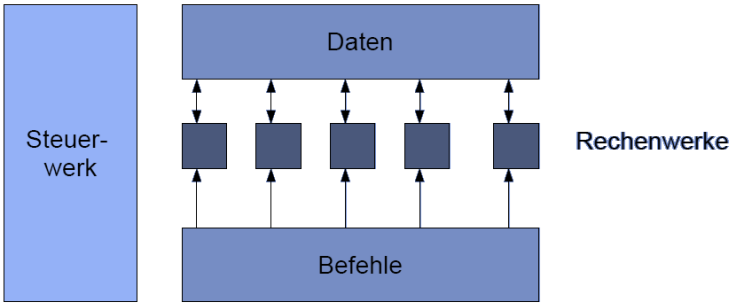
1.1.4 Von-Neumann-Rechnerarchitektur

Der Rechner besteht aus 5 Einheiten: Steuerwerk, Rechenwerk, Speicher, Ein- und Ausgabewerk. Die Struktur des Rechners ist unabhängig vom zu lösenden Problem (ist universell). Programme und Daten werden im gleichen Speicher abgelegt. Speicher sind in gleichgroße Zellen unterteilt, die fortlaufend nummeriert sind. Aufeinander folgende Befehle eines Programms werden in aufeinander folgenden Speicherzellen abgelegt. Weiterhin kann durch Sprungbefehle von der sequenziellen Bearbeitung abgewichen werden. Außerdem existiert ein Befehlssatz für arithmetische Befehle, logische Befehle, Transportbefehle, Verzweigungsbefehle und sonstige Befehle. Alle Daten werden binär kodiert.



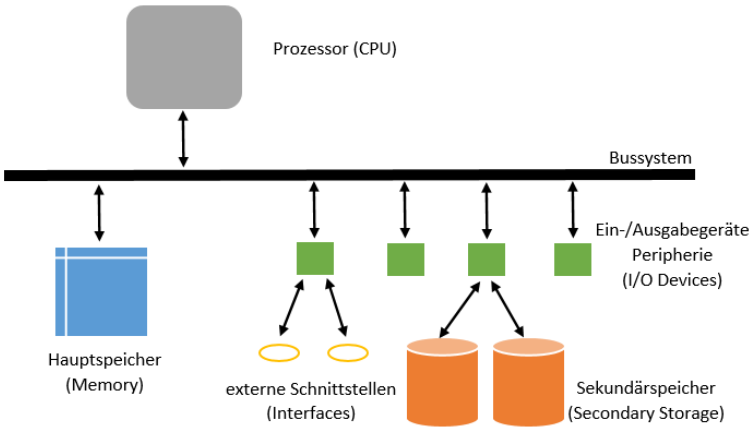
1.1.5 Harvard-Rechnerarchitektur

Bei der Harvard-Rechnerarchitektur ist der Befehlsspeicher physisch vom Datenspeicher getrennt. Dies bringt einige Vorteile mit sich. Gleichzeitiges Laden und Speichern von Befehlen und Daten ist möglich. Außerdem kann bei Softwarefehlern kein Programmcode überschrieben werden und die Datenwortbreite kann sich von Befehlswortbreite unterscheiden. Jedoch gibt es auch Nachteile. Die Harvard-Rechnerarchitektur ist komplexer als die von-Neumann-Architektur. Ein weiters Problem ist, dass der Datenspeicher nicht als Programmspeicher verwendet werden kann und umgekehrt.



1.1.6 Moderne Rechnerarchitektur

Moderne Mikroprozessoren verwenden eine Mischform aus der von-Neumann und der Harvard-Architektur (getrennte Daten- und Befehlsbereiche, getrennte Caches und MMU's, getrennte Busse, externen gemeinsamen Speicher).



1.2 Prozesse, Tasks und Threads

1.2.1 Vom Programm zum Prozess

Ein Prozess ist ein in Ausführung befindliches (nicht zwingend aktives) Programm (Maschinenprogramm, Anwendung). Während ein Maschinenprogramm nur eine Folge von Befehlen ist, wird dieses Programm zum Prozess, indem es in den Speicher geladen wird, einen eigenen Prozess-Kontext erhält und gestartet wird (Der Programmzähler wird auf die Anfangsadresse des Maschinenprogramms gesetzt). Insbesondere beinhaltet der Prozess den aktuellen Wert des Programmzählers sowie aktuelle Werte der Register und der Variablen.

Ein Prozess ist ein Programm, das sich in Ausführung befindet und seine aktuell genutzten Daten im Hauptspeicher ablegt. Wenn sich ein Programm in mehrfacher Ausführung befindet, dann gibt es auch mehrere Prozesse. Die konkrete Ausführungsumgebung für ein Programm mit den dazu notwendigen Betriebsmitteln ist der Speicher mit Rechten, Verwaltungsinformationen (verbrauchte Rechenzeit, ...), etc.

Ein Prozess stellt eine Ausführungsumgebung wie ein virtueller Prozessor, der ein Programm ausführt, bereit. Der Speicher stellt den virtuellen Adressraum und der Prozessor die Zeitanteile am echten Prozessor dar. Interrupts stehen für Signale zur Unterbrechung. Die I/O-Schnittstellen bilden das Dateisystem und weitere Kommunikationsmechanismen. Die Maschinenbefehle werden direkt durch den echten Prozessor ausgeführt.

1.2.2 Mehrprozessbetrieb (Multitasking)

Beim Mehrprozessbetrieb können mehrere Prozesse quasi gleichzeitig ausgeführt werden. Dabei werden für jeden Prozessor Zeitanteile der Rechenzeit an die Prozesse im *Time Sharing System* vergeben. Die Entscheidung der Umschaltung zwischen Prozessen und die Zuteilung von Rechenzeit trifft das Betriebssystem mit dem *Scheduler*. Weiterhin erfolgt die Umschaltung zwischen Prozessen durch das Betriebssystem mit Hilfe eines *Dispatchers*. Die Prozesse laufen nebenläufig, sprich das gerade in Ausführung befindliche Programm weiß nicht, wann und wo auf einen anderen Prozess umgeschaltet wird. Die Grundlage für die Arbeit im Multitasking/Multiprocessing sind Prozesse mit eigenständigen Befehlszählern je Prozess.

1.2.3 Mehrbenutzerbetrieb (*Multi-User*)

Im Mehrbenutzerbetrieb führen mehrere Benutzer mehrere Programme in einzelnen Prozessen aus. Das Prinzip ist ähnlich dem Multitasking-Betrieb, aber die Unterscheidung von Benutzern und evtl. Zugriffsberechtigungen wird durch das Betriebssystem realisiert.

1.2.4 Kooperatives Multitasking

Kooperatives Multitasking bezeichnet den Prozesswechsel unter Kontrolle der Prozesse. Der gerade laufende Prozess bestimmt den Zeitpunkt des Prozess-Kontextwechsels. Nachteile des kooperativen Multitaskings sind, dass die Wartzeiten die Arbeit anderer Prozesse beeinflussen (keine Transparenz), keine Fairness zwischen den Prozessen möglich ist und die Möglichkeit des Hängens beim Warten auf Ressourcen besteht.

Dennoch ist kooperatives Multitasking keineswegs überholt oder schlecht. Gerade im Bereich der Mikrocontroller und Echtzeitanwendungen gibt es viele gute Argumente dafür: Kooperatives Multitasking ist deterministischer, also besser zeitlich und logisch vorhersagbar und es ist besser simulierbar. Beispiele für Betriebssysteme mit kooperativem Multitasking sind Windows 3.x und MacOS vor Version 10.

1.2.5 Präemptives Multitasking

Der Prozesswechsel unterliegt dem Betriebssystem. Die Grundlage für den Prozesswechsel bilden zum einen der Wechsel nach/in Systemaufrufen und Unterbrechungen/Interrupts und zum anderen das Warten auf Ereignisse (z.B. Zeitpunkt, Nachricht, E/A-Operationen) wie das Beenden von Prozessen, das Ende der Zeitzuteilung im Time Sharing Betrieb oder der Prozess erhält den Status Abarbeitungsfähig.

1.3 Computerarchitektur nach Tanenbaum

Den Aufbau eines Rechnersystems kann man in einzelnen Ebenen betrachten. Tanenbaum hat dazu eine Auflistung, welche als *Computerarchitektur* bezeichnet wird, erstellt.

Folgende Ebenen eines Rechnersystems kann man einteilen:

- Anwendungsebene (Anwendungssoftware)
- Assemblerebene (Beschreibung von Algorithmen, Link und Bind)
- Betriebssystem (Speichermanagement, Prozesskommunikation)
- Instruction Set Architecture (6502, 8080, z80, x86/ia32/x64/ia64, Bitbreite)
- Microarchitektur (RISC, CISC)
- Logische Ebene (Aritmethik, Register, Schieberegister)
- Transistorebene (Transistoren, MOS, Flip Flop)

1.3.1 Transistorebene

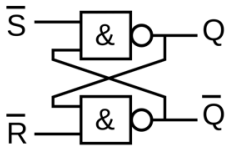
Für die Informationsverarbeitung in der Hardware kennen Computer nur zwei Zustände deren Informationsgehalt folgender ist:

- an, aus
- Strom fließt, fließt nicht
- Wahr (true), falsch (false)
- Magnetfeldänderung, keine Magnetfeldänderung
- Höhenänderung, keine Höhenänderung
- 1, 0

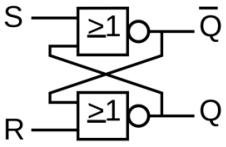
Computer-Hardware arbeitet (fast) ausschließlich auf dieser digitalen binären Grundlage. Die Informationsverarbeitung lässt sich vollständig auf Rechenoperationen mit Booleschen Operatoren UND/ODER/NOT zurückführen. Weiterhin erfolgt die Informationsspeicherung Bit-weise, sprich in Form einzelner Nullen und Einsen.

1.3.1.1 Speicherung von Bits

Die Speicherung einzelner Bits kann sehr einfach durch die Speicherung von elektrischer Ladung realisiert werden, beispielsweise durch Kondensatoren oder Flip-Flops. Einzelne Flip-Flops (Transistor/Bausteine) speichern eine Ziffer (1 Bit). Wenn elektrische Ladung vorhanden ist, wird eine Eins gespeichert. Wenn keine elektrische Ladung vorhanden ist, wird eine Null gespeichert.



Flip-Flop mit NAND-Gattern



Flip-Flop mit NOR-Gattern

1.3.1.2 Register

Mehrere Flip-Flops (Register) speichern eine komplette Zahl in Registerbreite:

- 8 Bit = 1 Byte
- Wort = 16 Bit / 32 Bit (Word)
- Doppelwort = 32 Bit / 64 Bit (Double Word)
- Vierfachwort = 64 Bit / 128 Bit (Quad Word)

Die Wortbreite hängt vom Prozessor und seiner Architektur ab ebenso das Byte-Ordering. Alle konventionellen Rechner sind Byte-Adressiert, sprich das Worte (egal ob 8, 16 oder mehr Bit) aus einer Folge (aufsteigender) Bytes bestehen. Dabei gilt das erste Byte als die Adresse des Wortes. Nimmt die Wertigkeit mit aufsteigender Adresse zu, ist es das Little-Endian-Format, umgekehrt das Big-Endian-Format.

1.3.1.2 Endian

Beim Little-Endian-Format kommt die niedrigste Wertigkeit zuerst. Dieses Format wird von x64 verwendet. Beispielsweise wird eine Hundertzwanzig als 021 dargestellt.

Das Big-Endian Format hat an der ersten Stelle die höchste Wertigkeit. Es verhält sich wie ein alltägliches Dezimalsystem. Hundertzwanzig würde im Big-Endian-Format als 120 dargestellt werden.

1.3.2 Logische Ebene

1.3.2.1 Informationsverarbeitung in Rechnern

Für die Rechnerarithmetik in Rechnern stehen Funktionen wie Addition, Subtraktion, Multiplikation und Division zur Verfügung. Außerdem wird in Rechnern noch die Festkomma- und Gleitkommaarithmetik verwendet.

Weiterhin werden natürliche Zahlen verwendet. In der positionellen Darstellung werden die Ziffern mit der Position der Ziffern als Gewichtung verwendet. Außerdem sind die Zahlen in jeder beliebigen Basis b darstellbar und umrechenbar.

1.3.2.2 Zahlenbasis

Basis	Zeichen	Beispiel 140
2 (binär)	0, 1	10001100
8 (oktal)	0, 1, 2, 3, 4, 5, 6, 7	214
10 (dezimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	140
16 (hexadezimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	8C

1.3.2.3 Boolsche Operationen

Unter Boolschen Operationen werden die Konjunktion UND, die Disjunktion ODER, die Negation NOT oder die Kontravalenz XOR verstanden.

A	B	A and B	A or B	not A	A xor B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

Beispielhaft die schriftliche Binäraddition – das Verfahren ist mit boolschen Operatoren mit Übertrag wie beim Dezimalsystem durchführbar:

```
      10011
    + 1001
      11  Übertrag
Ergebnis  11100
```

Der Übertrag wird auch Carry bezeichnet.

1.3.2.4 Overflow und Underflow

Probleme liefert hier eine feste Registerbreite bei der Addition:

Es gibt verschiedene Lösungen für das Problem der festen Registerbreite bei der Addition. Eine Variante ist die Verwendung eines Halbaddierers mit Carrybit. Außerdem könnte ein Volladdierer mit Carrybit in neuen Eingang eingesetzt werden. Als letzte Variante wäre auch ein paralleles Addierwerk aus Volladdierern denkbar.

1.3.2.5 Darstellung negativer ganzer Zahlen

Negative ganze Zahlen werden durch das Vorzeichen und den Betrag dargestellt. Das Vorzeichen wird hierbei durch ein Bit repräsentiert. Die anderen Bits repräsentieren den Betrag der Zahl.

Beispiel:

010012 = **+**9
110012 = **-**9

Der Nachteil besteht darin, dass das Vorzeichen für eine Berechnung explizit ausgewertet werden muss.

Einerkomplement

Eine negative Zahl -x wird binär durch das bitweise Komplement der entsprechenden positiven Zahl x dargestellt.

Komplementierung (6) = 0110 wird zu (−6) = 1001.

Beispiel für n=4:

Dezimal	0	1	2	3	4	5	6	7
Binär	0000	0001	0010	0011	0100	0101	0110	0111

Dezimal	-0	-1	-2	-3	-4	-5	-6	-7
Binär	1111	1110	1101	1100	1011	1010	1001	1000

Die Addition/Subtraktion muss in mehreren Schritten erfolgen. Als erstes wird eine normale Binäraddition durchgeführt. Daraufhin erfolgt eine zusätzliche Addition eines eventuellen Übertrags aus der ersten Binäraddition. Im letzten Schritt werden die bei der 2. Addition entstehenden Überträge gestrichen.

$$\begin{array}{r} 5 \\ +-6 \\ \hline = -1 \end{array} \qquad \begin{array}{rr} 0101 & (5) \\ +1001 & (-6) \\ \hline = 1110 & (-1) \end{array}$$

$$\begin{array}{r} 5 \\ +-3 \\ \hline = 2 \end{array} \qquad \begin{array}{rr} 0101 & (5) \\ +1100 & (-3) \\ \hline = 1|0001 & (-14) \\ +1 & (1) \\ \hline = \not{1}|0010 & (2) \end{array}$$

Quelle: D. W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009

Zweierkomplement

Beim Zweierkomplement werden die negativen Zahlen durch Bestimmung des Einerkomplements und einer zusätzlichen Addition von Eins gebildet.

2er Komplementierung (6) = 0110 wird zu 1001 + 0001 = (-6) = 1010

Beispiel für n=4:

Dezimal	0	1	2	3	4	5	6	7
Binär	0000	0001	0010	0011	0100	0101	0110	0111

Dezimal	-1	-2	-3	-4	-5	-6	-7	-8
Binär	1111	1110	1101	1100	1011	1010	1001	1000

Zur Addition kann die normale vorzeichenlose Binäraddition verwendet werden. Die Subtraktion entspricht einer Negation und anschließender Addition, wie z.B. (5 − 6) = (5 + (−6)).

5	0101	(5)
$+-6$	$+1010$	(-6)
$=-1$	$=1111$	(-1)

5	0101	(5)
$+-3$	$+1101$	(-3)
$=2$	$=\not{1}\|0010$	(2)

1.3.2.6 Festkomma-Darstellung von Zahlen

Die Festkomma-Darstellung von Zahlen wird durch eine feste Kommaposition bei dessen Darstellung erreicht. Die Festkommadarstellung kommt nur bei spezieller Hardware zum Einsatz, wie z.B. digitalen Signalprozessoren (DSPs).

Beispiel: n = 4, Komma an Position k = 2. Der Registerinhalt 0110 bedeutet 01, 10 binär und 1,5 dezimal (0 * 2^1 + 1 * 2^0 + 1 * 2^-1 + 0 * 2^-2).

Für negative Zahlen funktioniert dies analog (Vorzeichen erstes Bit).

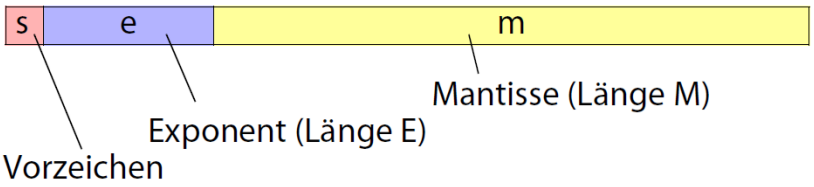
1.3.2.7 Gleitkommadarstellung

Das Ziel der Gleitkommadarstellung (engl. Floating point) ist die Darstellung großer und kleiner Zahlen mit dem gleichen Verfahren. Datentypen, welche als Gleitkommazahl dargestellt werden, sind beispielsweise *real*, *float* oder *double* aus den gängigen Programmiersprachen. Die Gleitkommazahlen entsprechen nicht reellen Zahlen im mathematischen Sinne, sondern stellen eine bloße Annäherung dar.

Die Idee hinter der Umsetzung ist, eine wissenschaftliche Potenzschreibweise zu nutzen. Dabei wird eine Anzahl von Ziffern *(Mantisse)* gemeinsam mit der Position des Kommas (Gleitkomma), welche durch Exponenten dargestellt wird, abgebildet. Bei der Potenzschreibweise im wissenschaftlichen Bereich wird das Komma durch den Exponenten in der Stelle verschoben: Für kleine Zahlen erfolgt eine Verschiebung nach rechts ($0{,}00021 = 0{,}21 * 10^{-4}$), wohingegen für große Zahlen eine Verschiebung nach links erfolgt ($1600000{,}00021 = 0{,}160000000021 * 10^{7}$). Die binäre Gleitkommadarstellung erweitert die Festkommadarstellung um einen binär-kodierten Exponenten.

IEEE 754 Standard der Gleitkommadarstellung

Der IEEE 754 Standard befasst sich mit der Vereinheitlichung der unterschiedlichen Darstellungen. Der Aufbau einer IEEE 754 Gleitkommazahl ist wie folgt vorgesehen:



IEEE 754 Gleitkommazahl Formatdefinitionen

	Single Precision	***Double Precision***	***Quad Precision***
Gesamtlänge *(N)*	32 Bit	64 Bit	128 Bit
Vorzeichen	1 Bit	1 Bit	1 Bit
Mantisse *(M)*	23 Bit	52 Bit	112 Bit
Exponent *(E)*	8 Bit	11 Bit	15 Bit
Bias *(B)*	127	1023	16383
$\|x_{min}\|$ (norm.)	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$	$2^{-16382} \approx 10^{-4932}$
$\|x_{min}\|$ (denorm.)	$2^{-149} \approx 10^{-45}$	$2^{-1074} \approx 10^{-324}$	$2^{-16492} \approx 10^{-4965}$
$\|x_{max}\|$	$(2 - 2^{-23}) * 2^{127} \approx 10^{38}$	$(2 - 2^{-52}) * 2^{1023} \approx 10^{308}$	$(2 - 2^{-112}) * 2^{16383} \approx 10^{4932}$

1.3.2.8 Zusammenfassung für die Binäre Arithmetik

Die Boolesche Algebra bildet die Grundlage für die Computer-Hardware. Die Werte 0 und 1 werden zum Rechnen mit Operationen der Konjunktion, Disjunktion und Negation verwendet. Für die Addition gibt es zwei verschiedene Addierer. Ein Halbaddierer addiert zwei Bits, wobei er eine Summe und ein Carry produziert. Der Volladdierer hingegen addiert zwei Bits und den Carry. Üblicherweise bestehen Addierwerke aus Volladdierern. Die Subtraktion lässt sich anhand des 2er-Komplements durchführen.

1.3.2.9 Repräsentation von Texten

Ein einfacher Ansatz für die Repräsentation von Texten wäre eine Codierung A=0, B=1, ... und dann eine fortgesetzte Binärcodierung. Jedoch kommen mit diesem Ansatz einige Probleme auf. Es ist fraglich, welche Zeichen wie codiert werden sollen. Außerdem muss überlegt werden, welche Sonderzeichen im Zeichensatz existieren. Ein weiteres Problem stellt die Frage dar, wie Daten/Texte zwischen Computern ausgetauscht werden können. Die Lösung für diese Probleme bieten standardisierte Zeichensätze.

1.3.2.10American Standard Code for Information Interchange – ASCII

Der American Standard Code for Information Interchange wurde 1963 verabschiedet und von der American Standards Organization für die USA entwickelt. Dabei handelt es sich um einen 7-Bit Code, welcher Zeichen codiert und Steuercodes zur Kontrolle von Geräten liefert. Beispielsweise wird der Code 0x0D *CR (carriage return)* für den Wagenrücklauf bei Druckern verwendet.

Code	**...0**	**...1**	**...2**	**...3**	**...4**	**...5**	**...6**	**...7**	**...8**	**...9**	**...A**	**...B**	**...C**	**...D**	**...E**	**...F**
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

1.3.2.11ISO-8859-1 / ISO Latin 1

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*nicht belegt*															
1...																
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8...	*nicht belegt*															
9...																
A...	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
B...	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D...	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E...	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F...	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Das Problem ist, dass bei ASCII viele wichtige Zahlen fehlen. Ein Ansatz, um dieses Problem zu beheben ist die „Längere Codierung“, wobei es sich um eine Erweiterung des Zeichensatzes durch den ISO-8859-1 / ISO Latin 1 (ISO = International Organization for Standardization
) handelt.
Mithilfe dieser Erweiterung entsteht ein 8-Bit Code, der viele Sonderzeichen für westeuropäische Sprachen enthält (z.B. ß, ä, â, à, á, î, ì, í, ö, ê, è, ë, é, æ, €, etc.). Dennoch fehlen auch hier einige wichtige Zeichen wie französische Sonderzeichen, das €-Symbol oder Emojis.

1.3.2.12Unicode

Der Unicode wird vom Unicode-Konsortium (http://www.unicode.org) verwaltet. Er unterstützt verschiedene Codierungsformate (UTF - Unicode Transformation Format) wie UTF-8 (8 Bits), UTF-16 (16 Bits) und UTF-32 (32 Bits). Dabei ergänzen längere Unicode-Formate kürzere Formate.
Bei Verwendung des Unicodes sind auch weitere Informationen, z.B. die Schreibrichtung, angegeben. Der Unicode ist kontinuierlich, um neue Zeichen ergänzen zu können. Ziel des Unicodes ist die Codierung aller in Gebrauch befindlicher Schriftsysteme und Zeichen.

Der Unicode ist derzeitig der defacto Unicode Standard. Es gibt 17 Planes zu je 65.536 Zeichen. 6 Planes werden derzeit genutzt und die restlichen Planes sind für eine spätere Nutzung vorgesehen.

Die Basic Multilingual Plane (BMP, 0) ist ein grundlegender mehrsprachiger Codebereich, welcher aktuell gebräuchliche Schriftsysteme, Satzzeichen und Symbole enthält. Die Supplementary Multilingual Plane (SMP, 1) beinhaltet historische Schriftsysteme und weniger gebräuchliche Zeichen (z.B. Domino- und Mahjonggsteine). Die Supplementary Ideographic Plane (SIP, 2) dient als ergänzender ideographischer Bereich für selten benutzte fernöstliche CJK-Schriftzeichen (CJK = China, Japan, Korea).

Unicode – UTF – Unicode Transformation Format forensische Bedeutung

Veränderung (de)

```
56 00 00 00|65 00 00 00|72 00 00 00|E4 00 00 00|6E 00 00 00|64 00 00 00    | UTF-32LE
00 00 00 56|00 00 00 65|00 00 00 72|00 00 00 E4|00 00 00 6E|00 00 00 64    | UTF-32BE
V          |e          |r          |ä          |n          |d               | Veränd
65 00 00 00|72 00 00 00|75 00 00 00|6E 00 00 00|67 00 00 00                | UTF-32LE
00 00 00 65|00 00 00 72|00 00 00 75|00 00 00 6E|00 00 00 67                | UTF-32BE
e          |r          |u          |n          |g                           | erung
56 00|65 00|72 00|E4 00|6E 00|64 00|65 00|72 00|75 00|6E 00|67 00          | UTF-16LE
00 56|00 65|00 72|00 E4|00 6E|00 64|00 65|00 72|00 75|00 6E|00 67          | UTF-16BE
V    |e    |r    |ä    |n    |d    |e    |r    |u    |n    |g               | Veränderung
56|65|72|C3 A4|6E|64|65|72|75|6E|67                                        | UTF-8
V |e |r |ä    |n |d |e |r |u |n |g                                        | Veränderung
```

1.3.3 Microarchitektur (RISC, CISC)

CISC und RISC sind unterschiedliche Konzepte für Computer oder Prozessoren. CISC steht für einen Prozessor, der einen umfangreichen Befehlssatz hat (spezialisiert komplex). Dem gegenüber steht RISC für einen Prozessor, der einen reduzierten Befehlssatz hat (Limitiert einfach).

Klassische RISC-Prozessoren werden hauptsächlich in Smartphones und Tablets eingesetzt. Die ARM-Architektur ist ein typischer Vertreter. Intel- und AMD-Prozessoren werden als typische CISC-Prozessoren verstanden, die intern aber auch mit RISC-artigen Strukturen arbeiten.

1.3.3.1 CISC – Complex Instruction Set Computing

CISC steht für Complex Instruction Set Computing. Der CISC-Prozessor zeichnet sich durch einen großen Befehlsumfang und komplexe Adressierungsmöglichkeiten aus. Die Anweisungen erfolgen mit komplexen Aufgaben. Weiterhin bietet CISC komplexe Adressierungsmöglichkeiten und ist auf Spezialaufgaben optimiert. Ein Beispiel ist der AMD x64 / Intel ia64, welcher intern jedoch auch RISC durchführt.

Funktionsprinzip eines CISC-Prozessors

Bei CISC wird jeder Befehl durch einen Mikrocode definiert, welcher im ROM-Speicher des CISC-Prozessors abgelegt ist. Bei jedem Befehlsaufruf wird von der Decoder-Einheit der Befehl in den Maschinenbefehl, die Adressierungsart, die Adressen und das Register aufgeteilt. Für die eigentliche Befehlsausführung werden kleine Anweisungen, der Mikrocode, an den Nanoprozessor geschickt, der den Mikrocode in seinen komplexen Schaltkreisen ausführt.

1.3.3.2 RISC – Reduced Instruction Set Computing

RISC steht für Reduced Instruction Set Computing. Übersetzt ist das ein Computer (Prozessor) mit einem reduzierten Befehlssatz, der nur wenige elementare Befehle enthält. Somit setzt RISC auf wenige Anweisungen mit einfachen, elementaren Instruktionen und ist auf Pipelining optimiert. Diese wenigen einfachen Befehle haben einen einfacheren Prozessoraufbau zur Folge, wobei sich im Prozessor einfache digitale fest verdrahtete Schaltungen befinden.

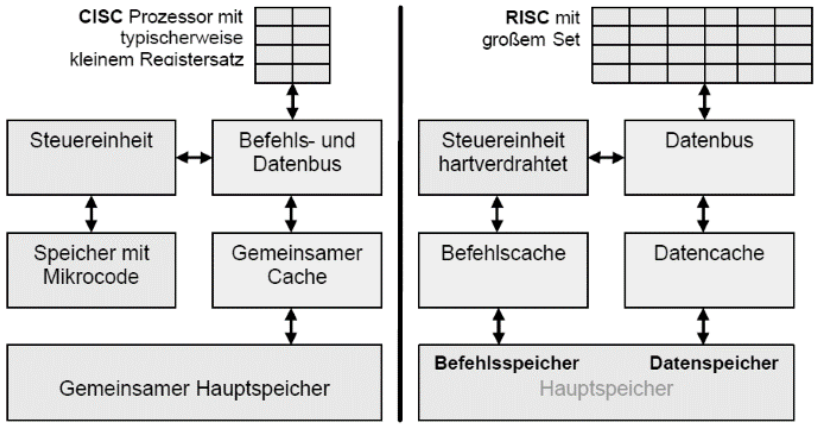
Die meisten Befehle können innerhalb weniger Taktzyklen ausgeführt werden. Ziel ist ein Takt pro Befehl (superskalar). Weiterhin sind RISC-Befehle viel schneller geladen als CISC-Befehle, weil sie nicht in kleinere Mikrocodes dekodiert werden müssen.

Beispiele für Mikroprozessoren, welche RISC anwenden, sind ARM aarch64, PowerPC oder OpenRISC.

Funktionsprinzip eines RISC-Prozessors

RISC setzt auf unabhängige Verarbeitungseinheiten. Zudem werden mehrere getrennte interne Bussysteme eingesetzt und der Fokus liegt auf der Parallelverarbeitung der Befehle. Alle RISC-Befehle haben das gleiche Format und es gibt nur eine Möglichkeit sie zu laden oder zu speichern.

1.3.3.3 Gegenüberstellung der Architektur von CISC und RISC



1.3.4 Instruction Set Architecture

Zur formalen Spezifikation einer Befehlssatzarchitektur gehört die Beschreibung des Befehlssatzes und dessen binärer Kodierung genauso wie eine Beschreibung der Verhaltensweise der CPU während bestimmter Betriebszustände und beim Eintreten bestimmter Ereignisse. Dabei geht es um das Verhalten der CPU bei einer Unterbrechungsanforderung, die Startadresse der Befehlsabarbeitung, die Initialisierung der Register nach einem Reset und den Aufbau wichtiger Datenstrukturen.

1.3.4.1 Formen der Implementierung

Mikroprozessor

Man spricht davon, dass ein Mikroprozessor eine Befehlssatzarchitektur implementiert bzw. unterstützt, wenn er alle im Sinne der Regeln dieser Befehlssatzarchitektur gültigen Programme in der vorgesehenen Art und Weise ausführen kann.

Viele real existierende Befehlssatzarchitekturen haben niemals eine formale Spezifikation erfahren, da sie historisch gewachsen sind. Auch eine Abwärtskompatibilität ist nicht immer gegeben gewesen. In der Praxis werden also häufig auch manche in den Datenblättern nicht dokumentierte Eigenschaften oder vermeintlich unbedeutende Details einer konkreten CPU zum Bestandteil einer Befehlssatzarchitektur erhoben.

Virtuelle Maschine

Da eine Befehlssatzarchitektur lediglich eine formale Definition ist, muss sie nicht zwangsweise oder gar ausschließlich als Prozessor implementiert werden. Sie lässt sich auch in Form von Software als eine so genannte virtuelle Maschine implementieren. Der Hypervisor übernimmt in diesem Fall die Virtualisierung. Die Prozessoremulierung erfolgt auf einer abweichenden Architektur. Auf diese Art lässt sich auch Software für eine Befehlssatzarchitektur ausführen und testen, bevor die zugehörige CPU überhaupt gebaut wurde, sodass eine Softwarekompatibilität gegeben ist.

So wurden große Teile der IA-64-Unterstützung für den Betriebssystemkern Linux programmiert, bevor der erste Itanium Intels Fabriken verließ. Das ist auch der Grund, warum Linux bereits kurz nach Verfügbarkeit der ersten Testmuster auf der Itanium-CPU lauffähig war. Ein weiteres Beispiel ist die Konsolenemulation von PS1, GB und GBA.

1.3.5 Betriebssystem

1.3.5.1 Definitionen Betriebssystem (OS)

DIN 44300

„... die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen."

Tanenbaum

„... eine Software-Schicht ..., die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle … anbietet, die einfacher zu verstehen und zu programmieren ist [als die direkte Programmierung der Hardware]."

Silberschatz/Galvin

„... ein Programm, das als Vermittler zwischen Rechnernutzer und Rechner-Hardware fungiert. Der Sinn des Betriebssystems ist eine Umgebung bereitzustellen, in der Benutzer bequem und effizient Programme ausführen können."

Brinch Hansen

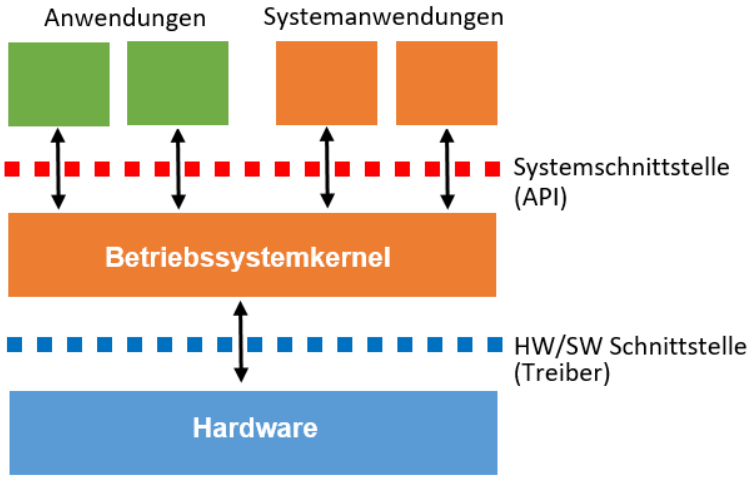
„... der Zweck eines Betriebssystems [liegt] in der Verteilung von Betriebsmitteln auf sich bewerbende Benutzer."

1.3.5.2 Zusammenfassung der einzelnen Definitionen

Das Betriebssystem stellt eine Software zur Steuerung und Überwachung der Betriebsarten dar. Weiterhin ist es eine Software zur Betriebsmittelverwaltung und agiert als Vermittler zwischen Hardwareebene und Benutzerebene. Es kann als eine einfache Abstraktionsebene für bequeme und effiziente Programmnutzung und als Benutzerschnittstelle für einfache Programmierung angesehen werden. Insgesamt lässt sich das Betriebssystem als Fundament für Anwenderprogramme beschreiben.

1.3.5.3 Schematischer Aufbau Betriebssystem – OS

Das Betriebssystem vermittelt zwischen Benutzer, Anwendung und Hardware.



1.3.5.4 Zentrale Aufgaben eines Betriebssystems

Die zentrale Aufgabe eines Betriebssystems ist die Betriebsmittelverwaltung. Unter einem Betriebsmittel (oder allgemein einer Ressource) eines Rechners versteht man eine beliebige Hardware- oder Software-Ressource. Neben der Betriebsmittelverwaltung gehören folgende Aufgaben zum Umfang eines Betriebssystems:

- Verwaltung mehrerer Prozesse
- Aus- und Einlagern von Prozessen
- Verwaltung des Hauptspeichers
- Versorgung aller Prozesse mit benötigten Teilen des Hauptspeichers
- Auswahl des jeweils nächsten Prozesses für die CPU (CPU-Scheduling)
- Datei- und Verzeichnisverwaltung auf Datenträgern
- Verwaltung der Ein- und Ausgabegeräte

1.3.5.5 Ressourcenverwaltung durch Betriebssysteme

Bei physikalischen Ressourcen handelt es sich um Hardwareressourcen. Dazu zählen Bestandteile wie die CPU, der Speicher, Ein- und Ausgabegeräte, Externe Schnittstellen und der Festspeicher.

Die virtuellen Ressourcen sind Ressourcen, die logisch sind und erst vom Betriebssystem geschaffen wurden. Dazu gehören Speichersegmente, Dateien und Gerätedateien.

Aufgaben für das Betriebssystem

Bei der Ressourcenverwaltung ergeben sich verschiedene Aufgaben für das Betriebssystem. Das Betriebssystem ist für die Belegung der Ressourcen je nach Anforderung zuständig. Auch das Multiplexen von Ressourcen für unterschiedliche Benutzer bzw. Prozesse gehört zu dessen Aufgaben. Weiterhin werden vom Betriebssystem Schutzmechanismen angewandt. Dabei geht es sowohl um den wechselseitigen Ausschluss von Ressourcenzugriffen als auch um die Umsetzung von Zugriffsberechtigungen. Schlussendlich ist das Betriebssystem für die Fehlerbehandlung und Fehlertoleranz verantwortlich.

Die Koordinierung der Nutzung von Ressourcen

Die Nutzung von Ressourcen lässt sich in drei Kategorien einteilen. Zum einen in aktive, zeitlich aufteilbare Ressourcen wie wie die des Prozessors oder der Grafikkarte. Zum anderen gibt es passive, ausschließlich exklusiv nutzbare Ressourcen, beispielsweise E/A-Geräte wie Drucker. Zuletzt gibt es noch passive, räumlich teilbare Ressourcen. Beispiele hierfür sind Speicher wie der Festspeicher.

Für die Ressourcenverwaltung werden Einzelkomponenten des Betriebssystems genutzt. Dazu gehören die Anwendungs- und Prozessverwaltung, Dateisysteme, die Speicherverwaltung und die Verwaltung der Ein-/Ausgabegeräte.

Die Steuerung der Ressourcenverwaltung erfolgt durch den Kommandozeileninterpreter (Shell) das GUI (Graphical User Interface), die Programmablaufsteuerungen und Systemkonfigurationen und Systemdienste.

1.3.5.6 Umsetzungen von Betriebssystemen

Die Realisierungen sind zum einen abhängig von der Hardware und dessen Konfiguration (Verteilungen und Abstraktionen). Hierbei kommt es auf die Basisarchitektur und die Betriebsarten an. Zum anderen sind Realisierungen abhängig von den verwendeten Programmiersprachen und Programmstrukturierungen. Dabei geht es sowohl um Ablaufsteuerungen und deren Komponenten (Tasks, Prozesse, Threads) als auch um Interaktionsmöglichkeiten wie Programmaufrufe, Benachrichtigungen, Datenaustausch und Datenkommunikation.

1.3.5.7 Klassifizierungsansätze von Betriebssystemen

Hinsichtlich der Betriebssysteme finden sich in der Literatur unterschiedliche Klassifizierungsansätze. Tanenbaum beschreibt 2009 eine Artenliste, die sich insbesondere an der Hardware orientiert, für welche die jeweilige Betriebssystemart konzipiert wurde. Mandl 2013 klassifiziert in den Kapiteln eher nach dem Einsatzszenario, unter dem das jeweilige Betriebssystem nach seiner Installation betrieben werden soll.

1.3.5.8 Klassifikation nach Betriebsarten

Eine Möglichkeit besteht in der Einteilung von Betriebssystemen nach Betriebsarten. Die Betriebsart (Nutzungsform, operation mode) legt die Art und Weise der Kommunikation mit dem Benutzer fest.

Hier eine Liste von möglichen Betriebsarten:

- Stapelverarbeitung (batch processing)
- Dialogbetrieb (interactive processing)
- Zeitgesteuerter Betrieb (time sharing processing)
- Echtzeitverarbeitung (real time processing)
- Verteilte Verarbeitung (distributed processing)

Stapelverarbeitung (batch processing)

Stapelverarbeitung (batch processing): Abarbeitung einer Folge von Stapelaufträgen (Jobs), die vom Benutzer mit allen erforderlichen Programmen, Daten und Anweisungen zur Ablaufsteuerung (Job Control Language, JCL) zusammengestellt werden. Ein solcher Batch Job wird ohne Interaktion des Benutzers vollständig abgearbeitet. Beispiele hierfür sind *IBM OS/370, OS/390, MVS* oder *BS 2000.*

Dialogbetrieb (interactive processing)

Bei dem Dialogbetrieb (interactive processing) erfolgt ein ständiger Wechsel zwischen Eingabeaktionen des Benutzers (z.B. Kommandoeingaben) und solchen des Systems (z.B. Kommandoausführung). Der Nutzer kann dabei den Arbeitsablauf durch den Dialog jederzeit beeinflussen. Beispiele hierfür sind *MS-DOS, MS Windows 95/98/ME/NT/2000/XP* oder *UNIX/Linux.*

Zeitgesteuerter Betrieb

Im Time-Sharing Betrieb wird die Rechenzeit auf mehrere Benutzer oder Programme aufgeteilt. Dabei gibt es feste Timeslots für die Ressourcenzuordnung. Beispiele für diese Betriebsart sind Spezialhardware mit fester Arbeitstaktung oder Backbone Router.

Echtzeitverarbeitung (real time processing)

Für die Echtzeitverarbeitung (real time processing) wird ein Computersystem zur Steuerung und Überwachung von technischen Prozessen eingesetzt, wobei durch das Echtzeit-Betriebssystem vor allem die Rechtzeitigkeit (Einhaltung von Zeitbedingungen) gesichert werden muss. Beispiele sind *VxWorks, VRTX, pSOS, LynxOS, Enea OSE, MS Windows CE, QNX, OSEK/VDX (speziell im Automobilbau)* aber auch SCADA Systeme gehören dazu.

Verteilte Verarbeitung (distributed processing)

Ein solches verteiltes System besteht aus mehreren miteinander gekoppelten Computern. Das Betriebssystem dient dabei vorrangig der Verteilung von Daten, Ressourcen und der Arbeitslast. Vertreter für die verteile Verarbeitung in Form verteilter Betriebssysteme sind *Amoeba, CHORUS* und *MACH*. Netzwerk-Betriebssysteme mit einer verteilten Verarbeitung sind unter anderem *Novell Netware* und *IBM AS-400*. Auch einige Parallelrechner-Betriebssysteme wie *UNICOS, SPP-UX* und *KSR-OS* setzen auf eine verteilte Verarbeitung.

1.3.5.9 Klassifikation nach Anzahl der Benutzer

Die Klassifikation erfolgt anhand der Anzahl der Benutzer, die zur selben Zeit am System bedient werden können. Dabei wird zwischen Einzelnutzer- und Mehrnutzer-Systemen unterschieden. Einzelnutzer-Systeme haben einen einfachen Aufbau, eine vereinfachte Rechteverwaltung, sind ressourcensparender und ermöglichen eine schnellere Ausführung. Mehrnutzer-Systeme hingegen ermöglichen mehrere Nutzer gleichzeitig und haben unterschiedliche Rechte für Programme.

Einzelnutzer-System (single user System)

Eine Authentifizierung und Vergabe von Rechten an mehrere Nutzer sind auf einem Einzelnutzer-System möglich, jedoch kann stets nur ein Nutzer am System arbeiten. Somit erfolgt keine parallele Erkennung bzw. Verwaltung von mehreren Nutzern. Beispiele für Einzelnutzer-Systeme sind *MS-DOS, MS Windows 3.X/95/98/ME* oder Linux im single user mode.

Mehrnutzer-System (multi user System)

Mehrnutzer-System (multi user System) bieten eine gleichzeitige Systemnutzung durch mehrere Benutzer, z.B. über mehrere Terminals oder einzelne Anmeldungen. Somit können Programme für verschiedene Nutzer laufen, aber auch gleiche Programme mit verschiedenen Rechten. Beispiele für Mehrnutzer-Systeme sind *UNIX, IBM OS/390, OS/400, BS2000, OpenVMS, Linux* und *Windows XP/Vista/2000/Server/7/8/10*.

1.3.5.10 Klassifikation nach der Anzahl der Prozesse

Einzelprozess-System (single tasking System)

Das Einzelprozess-System (single tasking System) kann jeweils nur einen einzigen Auftrag (Task) bearbeiten. Ein weiterer Auftrag kann erst nach Beendigung des aktuellen aufgenommen werden. Beispiele für Einzelprozess-Systeme sind *CP/M* und *MS-DOS*.

Mehrprozess-System (multitasking System / multi programming System)

Bei einem Mehrprozess-System (multi tasking System / multi programming System) werden gleichzeitig mehrere verschiedene Aufträge verwaltet und parallel oder zumindest quasi-parallel (zeitlich geschachtelt) verarbeitet. Beispiele hierfür sind *MS Windows 95/98/ME/NT/2000/CE/XP/ SERVER/VISTA/7/8/10, UNIX/Linux, IBM OS/390, OS/400, OS/2, BS2000, OpenVMS, VxWorks, VRTX, LynxOS* und *Enea OSE.*

1.3.5.11 Betriebssystem Modi

Die Verwaltung und Absicherung von Ressourcen in Single Prozess Systemen gestaltet sich auf Grund der Batch Verarbeitung einfach. Wenn Prozesse jedoch parallel oder quasi-parallel auf einem System ablaufen, ist die Verwaltung und Sicherheit der Ressourcen komplexer. Jeder einzelne Prozess besitzt eigene Daten, und es muss dafür Sorge getragen werden, dass nur der jeweilig berechtigte Prozess Zugriff auf diese Daten hat. Für alle anderen Prozesse muss sichergestellt sein, dass sie nicht unberechtigt Zugriff auf Prozessfremde Daten erhalten. Multiple Prozesse benötigen einen fairen Wechsel der Ressourcen damit alle Prozesse ihre Aufgabe erfüllen können. Dazu wird eine zentrale Stelle benötigt, die diese Wechsel durchsetzt und verwaltet.

Damit ein Betriebssystem die Verwaltung der Ressourcen und einzelnen Prozesse erfüllen kann, benötigt es auf einem Rechnersystem mehr Privilegien als jeder normale Anwendungsprozess.

Aus diesem Grund werden bei der Abarbeitung von Befehlen auf der CPU zwei Modi unterschieden:

1. Kernel-Mode
2. User-Mode

Für den Wechsel der Privilegien Level sind ebenfalls Techniken notwendig, die in modernen Betriebssystemen über sogenannte Systemaufrufe realisiert werden.

Kernel-Mode

Im Kernel-Mode ist jeder beliebige Befehl zur Ausführung zugelassen. Es kann auf sämtliche Speicherbereiche für Daten- und Programmtext, sowie auf alle Betriebsmittelressourcen zugegriffen werden. In diesem Level bestehen die höchsten Privilegien, man nennt diesen Modus deshalb auch den privilegierten Modus.

Durch ein Steuer- oder Kontrollregister auf der CPU wird der Kernel-Mode angezeigt. Das Betriebssystem arbeitet üblicherweise im Kernel-Mode und hat somit alle Möglichkeiten, seine vordefinierten Aufgaben zu erfüllen.

User-Mode

Im User-Mode ist nur ein eingeschränkter Befehlssatz zur Ausführung zugelassen. Weiterhin kann nicht auf alle Speicherbereiche und auch nicht auf alle Betriebsmittelressourcen zugegriffen werden.

Durch ein Steuer- oder Kontrollregister auf der CPU wird der User-Mode angezeigt. Anwendungsprogramme arbeiten üblicherweise im User-Mode und haben daher eingeschränkte Zugriffsmöglichkeiten.

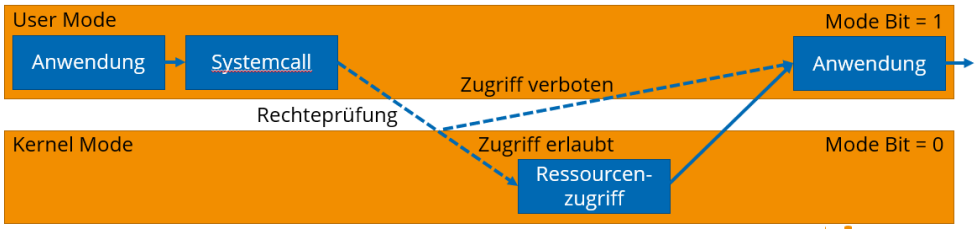
Ein Zugriff vom Kernel-Mode in den User-Mode ist unproblematisch, da hierbei die Rechte eingeschränkt werden und somit keine Sicherheitsbedenken bestehen. Anders verhält es sich beim Übergang vom User-Mode in den Kernel-Mode. Der Prozess eines Anwendungsprogramms darf nicht über erweiterte Rechte des Kernel-Modes verfügen.

Systemaufruf

Durch einen Systemaufruf (oder: Systemcall, Syscall) kann von einem im User-Mode ablaufenden Prozess ein Aufruf einer vom Betriebssystem zur Verfügung gestellten Funktion, welche nur im Kernel-Mode ausgeführt werden kann, durchgeführt werden.

Das Betriebssystem hat damit die Möglichkeit, durch vorherige Sicherheitsüberprüfungen festzustellen, ob der aufrufende Prozess zur Ausführung der gewünschten Funktion überhaupt berechtigt ist, und ob auch sonst keine anderen Gegebenheiten gegen eine Ausführung sprechen.

Ist die betreffende Funktion ausgeführt bzw. verweigert worden, so wird der Systemaufrufs wieder in den User-Mode zurückgeschaltet, und das Betriebssystem gibt die Kontrolle zurück an den aufrufenden Prozess, der auf das Resultat reagiert.



1.3.6 Betriebssystemarchitekturen

Architekturmodelle verdeutlichen die Anordnung der Komponenten des Betriebssystems und ihr funktionales Zusammenwirken. Sie geben zudem auch Aufschluss über die Portabilität des Systems. In der Literatur zu Betriebssystemen beschreibt Mandl 2013 unter anderem die Architektur des monolithischen Kernels und dessen Weiterentwicklung, den schichtenorientierten Kernel und stellt beiden Varianten die Mikrokern-Architektur gegenüber.

1.3.6.1 Architekturmodelle

Monolithische Architektur

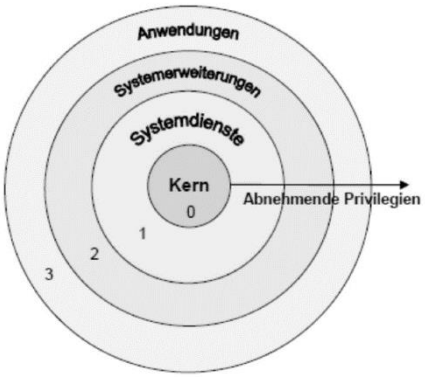
Alle wesentlichen Komponenten des Systems sind zu einem homogenen Gebilde zusammengefügt, das zwar u. U. effizient, aber nicht flexibel anpassbar ist. Kernel Mode (privilegiert) für das gesamte Betriebssystem und der User Mode (nicht privilegiert) für Anwendungsprogramme.

Kern-Schale-Architektur

Das System besteht aus dem privilegierten Kern (kernel), der die wichtigsten Komponenten vereint (z.B. die Prozessverwaltung), und einer Schale (shell) für ergänzende Bestandteile (z.B. Kommando-Interpreter). Typischer Vertreter dieses Modells ist UNIX.

Hierarchische Schichten (Mehrschichtenmodell):

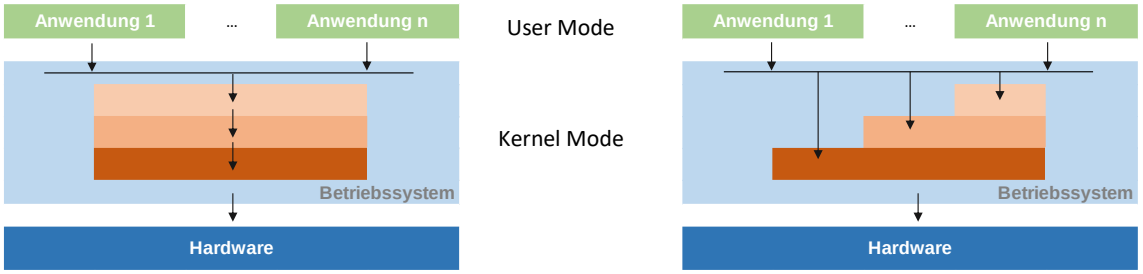
Das System wird modularisiert und in einzelne Schichten geteilt. Zwischen diesen sind Schnittstellen definiert, so dass sie austauschbar sind. Sie können mit abgestuften Privilegien ausgestattet sein. Diese Architektur ist weit verbreitet, z.B. bei MS-DOS, OS/2, und OpenVMS.



In der Hierarchie bietet jede Schicht ihre Dienstleistungen der (den) darüber liegenden Schicht(en) an und greift gleichzeitig zur Erfüllung ihrer Aufgaben auf die Dienste der tiefen liegenden Schicht(en) zurück. Die unterste Schicht ist im Allgemeinen die Hardware, die oberste oft ein Kommandointerpreter.

Konsistente Schichten sind leicht austauschbar, das strenge Durchlaufen aller Schichten führt aber oft zu Effizienzverlusten.

Bei quasi-konsistenten Schichten (Treppenstufenmodelle) kann dagegen bei Bedarf auf verschiedene tiefer liegende Schichten zugegriffen werden. Dies führt jedoch u. U. zu unkontrollierbaren Abläufen.



Konsistente Schichten

Quasi-konsistente Schichten

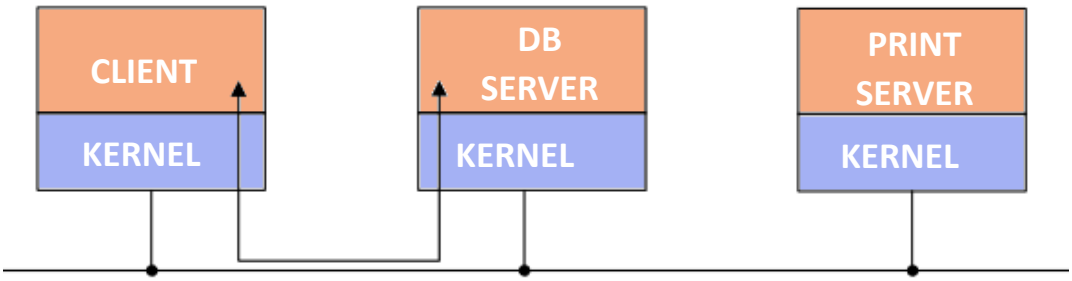
Mikrokern (micro kernel)

Er bildet nur noch eine Art Infrastruktur mit minimalem Funktionsumfang. Alle anderen Betriebssystemfunktionen werden durch Systemprozesse außerhalb des Kerns erbracht, die flexibel modifiziert oder erweitert werden können. Mikrokerne findet man z.B. bei MACH (OS X), CHORUS, QNX/Neutrino, z.T. bei MS Windows NT/XP/VISTA/2000/VISTA/7/8/10/SERVER.

Mikrokerne enthalten nur noch elementare Funktionen, z.B. zur Speicherverwaltung, IPC, Prozessverwaltung, Scheduling-Mechanismus, sowie einige hardwarenahe E/A-Funktionen. Systemprozesse realisieren dagegen den Rest der Verwaltung wie z.B. die Dateiverwaltung.

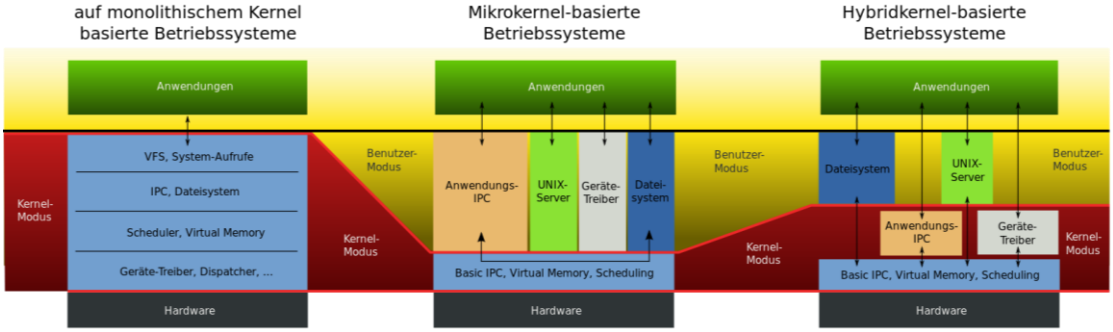
Client-Server-Modelle

Client-Server-Modelle basieren häufig auf Micro Kernel Architekturen. Dabei bietet ein Server Dienste an, die von den Clients genutzt werden. Die Kommunikation erfolgt hierbei durch einen Nachrichtenaustausch.



Hybridkernel (Makrokernel)

Der Hybridkernel ist ein Kompromiss zwischen einem Microkernel und einem monolithischen Kernel, bei dem aus Performancegründen Teile des monolithischen Kernels in den Kern integriert werden und dieser dadurch kein reiner Mikrokernel mehr ist.



Quelle: Golftheman übersetzt von Hagen

Der Vorteil des Hybridkernels besteht darin, dass er weniger fehleranfällig wie ein monolithischer Kernel ist, da nicht alle Treiber im privilegierten Modus laufen und bei einem Absturz das ganze System zum Absturz bringen. Ein weiterer Vorteil ist die höhere Geschwindigkeit des Kernels, da nicht so viele Kontextwechsel und Kommunikation nötig sind, wie bei einem Mikrokernel. Beispiele für den Einsatz von Hybridkernels sind *OSX* und *NT/XP/VISTA/2000/VISTA/7/8/10/SERVER*.

Virtuelle Maschinen

Einsatz eines Basis-Betriebssystems mit dem *Hypervisor,* auch als *Virtual Machine Monitor* bezeichnet, auf dem virtuelle Maschinen in Form unterschiedlicher verschiedener Betriebssysteme ablaufen. Diese sind von der realen Hardware und untereinander völlig entkoppelt. Beispiele hierfür sind IBM VM/370, ESXi, VMWare und VirtualBox. Dies ist unter anderem auch im Zusammenhang mit Java etwa bei der Dalvik Engine von Android von Bedeutung.

1.3.6.2 Klassifizierung anhand der Betriebssystemarten

Auch eine Klassifizierung anhand der Systemarten ist möglich:

1) Betriebssysteme für Großrechner
2) Betriebssysteme für Server
3) Betriebssysteme für Laptops und Personal Computer
4) Echtzeitbetriebssysteme
5) Betriebssysteme für Embedded Systems
6) Betriebssysteme für Chipkarten

Betriebssysteme für Großrechner

Großrechner (Mainframes) verarbeiten im Regelfall eine sehr hohe Zahl an Aufgaben in sehr kurzer Zeit. Meist sind sie optimiert auf eine sehr hohe Ein/Ausgabe-Rate und verfügen optional über eine sehr große Speicherkapazität auf angeschlossenen Speichersystemen.

Mainframe-Betriebssysteme arbeiten üblicherweise im Stapelbetrieb (Batch-Jobs) oder sind transaktionsorientiert, was bedeutet, dass eine sehr große Zahl an Transaktionen in sehr kurzer Zeit ausgeführt werden kann. Verschiedene Subsysteme ermöglichen den gleichzeitigen Betrieb im Dialog und im Batchbetrieb.

Ein Beispiel für ein Mainframe-Betriebssystem ist z/OS von IBM. Es ist u.a. auf Großrechnern bei Banken und Versicherungen im Einsatz.

Betriebssysteme für Server

Server werden überwiegend von vielen verschiedenen Clients über ein Netzwerk angesprochen. Charakterisiert werden Server-Betriebssystemseine dadurch, dass sie eine möglichst schnelle Reaktionszeit mit nur minimaler Verzögerung für die Datenbereitstellung benötigen. Zudem werden rechenintensive Aufgaben auf die zentrale Server Architektur ausgelagert (Print/Datenbank/Terminal).

Vertreter für Server-Betriebssysteme sind Windows Server, Unix bzw. Linux aber auch Mac OS X. Eine grafische Oberfläche ist nicht immer vorhanden.

Die Einsatzzwecke für Server-Betriebssysteme liegen beispielsweise bei File-Servern (Dateiablagen), Print-Servern, E-Mail-Servern, Terminal-Servern und Datenbank Servern.

Betriebssysteme für Laptops und Personal Computer

Mit dieser Art Rechnersystem dürften die meisten täglich in Berührung kommen. Durch verschiedene Benutzer können verschiedene Anwendungen quasi-parallel genutzt werden. Es handelt sich also um eine überwiegend dialogorientierte Nutzungsart: Der Anwender tätigt eine Eingabe und das Betriebssystem reagiert darauf.

In der Praxis werden heute überwiegend Betriebssysteme mit grafischer Oberfläche eingesetzt, z.B. Windows, MacOS X oder Linux mit KDE, Gnome bzw. Unity.

Echtzeitbetriebssysteme

Spezielles Betriebssystem bei denen die Verarbeitung von Informationen zeitkritisch ist und diese in nahezu Echtzeit erfolgen muss. Das Erfordernis eines Echtzeitbetriebssystem ergibt sich immer dann, wenn Rechner mit der physikalischen Welt messend und/oder steuernd in Verbindung stehen.

Das bedeutet das gesicherte Verarbeiten von Anfragen eines Anwendungsprogramms oder dem Eintreffen von Signalen über Hardware-Schnittstellen innerhalb einer im Vorhinein bestimmbaren Frist (Timeout, t_{max}). So sind zum Beispiel bei einer Heizungssteuerung längere t_{max} -Werte anwendbar als bei der Auslösung eines Airbags. Beispiele hierfür sind Steuerungsanlagen, Ampelsteuerungen und Robotertechnik.

für Embedded Systems

Hierbei handelt es sich um spezielle Betriebssysteme für Mikrocomputersysteme. Diese sind typischerweise bei tragbaren Geräten aber auch bei integrierten Bauteilen mit CPU im Einsatz. Betriebssysteme, die für Embedded Systems eingesetzt werden, sind Windows CE, Windows Mobile, Palm OS, Android und Linux. Einige Anwendungsbeispiele sind im Folgenden aufgelistet:

- Automobilindustrie
- Industriesteuerungsanlagen
- tragbare Navigationsgeräte
- Waschmaschinen, Haushaltsgeräte
- IoT Geräte

Betriebssysteme für Chipkarten

Spezialbetriebssysteme für Chipkarten mit meist sehr reduziertem Funktionsumfang. Lauffähig auf Chipkarten wie etwa Smartcards von Bezahldiensten oder SIM-Karten von Telefonanbietern.

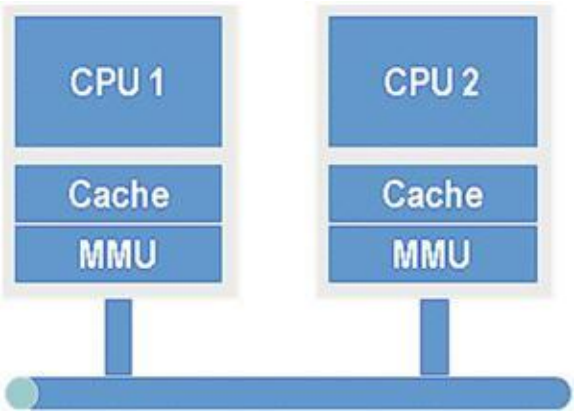
Vorrangig werden dabei Rechenoperationen abgebildet, die durch das Chipkarten Betriebssystem durchgeführt werden, um Verschlüsselungen und Speicherzugriffe zu realisieren.

1.3.6.3 Betriebssysteme und deren Hardwareunterstützung

Moderne Betriebssysteme werden durch verschiedene Hardwarebestandteile unterstützt, wie die CPU/Pozessor, die Memory Managment Unit (MMU), den DMA Controller (Direct Memory Access) und Ein-/Ausgabe Controller (USB, PCIe, SATA etc…).

Der Prozessor ist der Kernbestandteil modernen Hardware für Betriebssysteme. Er enthält die digitalen Schaltungen für die Rechenarithmetriken, die Register für die Daten und Befehle und führt die einzelnen Instruktionen auf Grund des Programmzählers aus. Dabei arbeitet der Prozessor wie bereits festgestellt in zwei Modi. Dem Benutzermodus (User Mode) mit eingeschränktem Befehlssatz und dem privilegierten Modus (Kernel Mode) der die Ausführung privilegierter Befehle erlaubt.

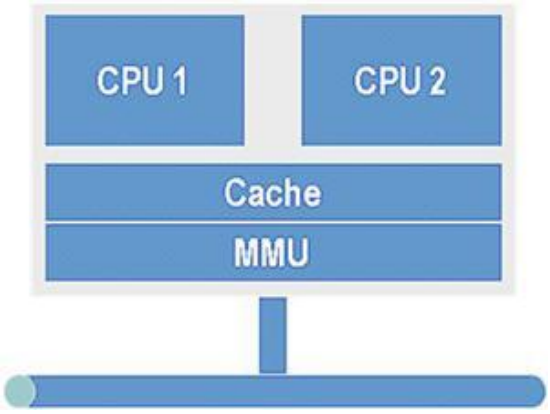
Multiprozessoren



Quelle: National Instruments

Multiprozessorsysteme enthalten mehrere Prozessoren, die sich auf verschiedenen Chips befinden. Diese Systeme verbreiteten sich in den 1990ern, v. a. im Bereich von IT-Servern. Zu dieser Zeit waren das meist Prozessorplatinen, die in einen Rack-Mount-Server installiert wurden. Heute befinden sich Multiprozessoren oft auf derselben Platine und werden über eine Hochgeschwindigkeits-Kommunikation angebunden.

Multicore-Prozessoren



Quelle: National Instruments

Multicore-Prozessoren werden von einer „Familie“ von Prozessoren gebildet, die mehrere Prozessoren, z. B. 2, 4 oder 8, auf einem Chip integrieren.

Allerdings stellen Multicore-Prozessoren Softwareentwickler vor einer Herausforderung. Höhere Geschwindigkeiten und damit bessere Leistung sind direkt davon abhängig, wie parallel der Quellcode einer Anwendung geschrieben wurde.

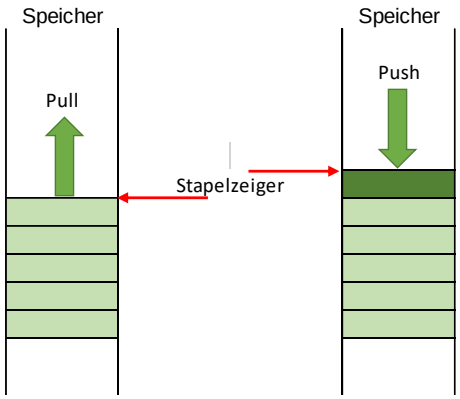
Bei der Prozessausführung in Prozessoren kann es notwendig sein, auf vordefinierte Ereignisse wie Fehlerbedingung, ankommende Netzwerknachrichten oder die Rückmeldung durch externe Geräte zu reagieren. Diese Unterbrechungen werden als Interrupts bezeichnet.

1.3.6.4 Externe Unterbrechungen durch Interrupts

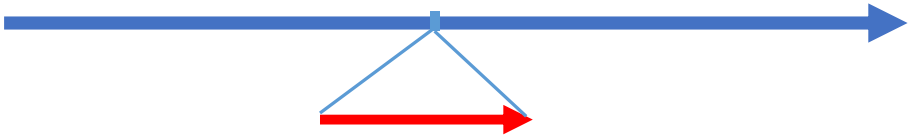
Der Prozessor unterbricht die laufende Bearbeitung und führt eine definierte Befehlsfolge aus, welche vom privilegierten Modus aus konfigurierbar ist. Im Vorfeld werden alle Register einschließlich Programmzähler gesichert (z.B. auf einem *Stapelspeicher/Stack*). Nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden. Weiterhin werden die Unterbrechungen im privilegierten Modus bearbeitet. Das Signalisieren der Unterbrechung erfolgt durch Interrupts *(Interrupt Request, IRQ).* Die Ansprungadresse der Unterbrechungsbehandlung steht fest und ist meist konfigurierbar. Die Rückkehr von der Unterbrechungsbehandlung erfolgt durch die Nutzung eines speziellen Befehles, z.B. RTI (Return from Interrupt). Es erfolgt eine Befehlsausführung bei einem RTI, sowie die Wiederherstellung gesicherter Register und die Rückschaltung in den alten Modus. Anschließend wird die Abarbeitung ab der alten Befehlsfolge wieder aufgenommen.

1.3.6.5 Einsatz eines Stapelspeichers (Stack)

Ein Stapelspeicher wird für Interrupts verwendet. Der Zugriff erfolgt mittels der Operationen Push und Pull. Push speichert ein Datum „oben“ auf dem Stapel, wohingegen Pull das zuletzt gespeicherte Datum vom Stapel holt.



1.3.6.6 Charakteristika für Externe Unterbrechungen durch Interrupts



Die unterbrochene Befehlsfolge bleibt in der Regel unberührt, da solche Unterbrechungen transparent sind und den Programmablauf nicht berühren. Außerdem sind verschachtelte Unterbrechungen sind möglich, sprich die erneute Unterbrechung der Unterbrechungsbehandlung ist möglich. Auch hier erfolgt eine koordinierte Handhabung der gesicherten Register. Weiterhin ist die Unterbrechung beliebiger Betriebsmodi möglich und somit auch die Unterbrechung des privilegierten Modus.

1.3.6.7 Interne Unterbrechungen durch Interrupts

Auch interne Unterbrechungen im Programmablauf sind möglich. Damit sind interne Unterbrechungen der Befehlsausführung durch *Exceptions* gemeint. Bei bestimmten Fehlersituationen (z.B. Division durch Null) unterbricht der Prozessor die laufende Befehlsbearbeitung und führt eine vorher definierte Befehlsfolge aus (ähnlich wie bei externen Unterbrechungen).

1.3.7 Prozesskontextwechsel (Process Context Switch)

Eine Prozess Umschaltung erfolgt durch mehrere Schritte. Zuerst erfolgt das Sichern der vom Prozess genutzten Register und Befehlszähler. Anschließend wird ein Prozess ausgewählt und die Prozessumgebung generiert. Die gesicherten Registerinformationen und Befehlszäher werden geladen und abschließend werden die Befehle des Befehlszähler abgearbeitet. Die notwendigen Prozessinformationen erhält das Betriebssystem dabei aus einem Prozesskontrollblock (PCB), welcher dem Prozess im Hauptspeicher vorgelagert ist.

1.3.8 Prozesskontrollblock (Process Control Block; PCB)

Der Prozesskontrollblock ist die Datenstruktur des Betriebssystems, die alle nötigen Daten für einen Prozess hält. Der PCB in Windows Umgebungen ist nicht offiziell veröffentlicht, da Windows zur Gattung der "Closed-Source"-Software gehört. *Russinovich et.al. 2012 (P1)* beschreiben, dass jeder unter Windows erzeugter Prozess durch eine Instanz der Datenstruktur EPROCESS (engl.: executive process structure) repräsentiert wird. Der PCB in UNIX Umgebungen enthält die Prozessnummer *(PID),* die verbrauchte Rechenzeit, den Erzeugungszeitpunkt, den Kontext (Register etc.), die Speicherabbildung, den Eigentümer *(UID, GID)* etc.

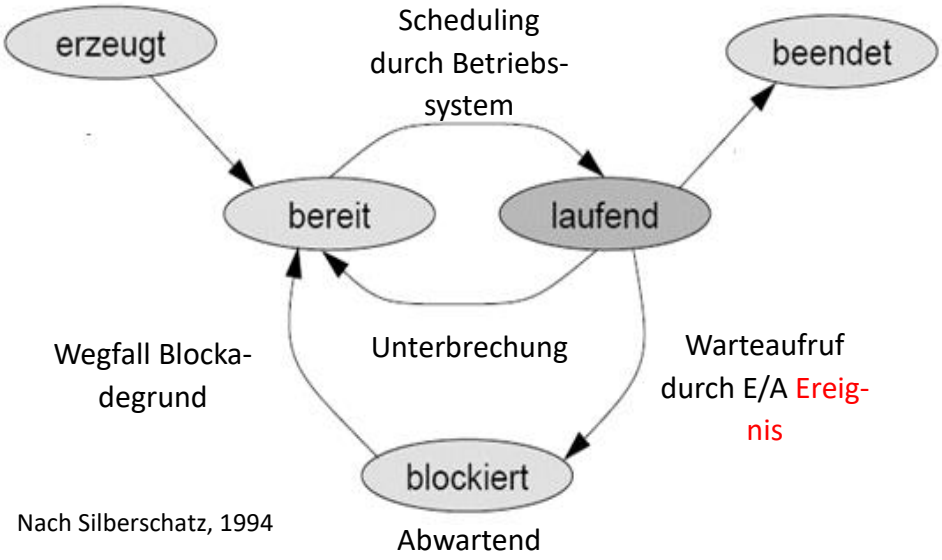
1.3.9 Prozesszustände

Ein Prozess befindet sich in einem der folgenden Zustände:

- **Erzeugt** *(New)*
  Prozess ist erzeugt, aber ist noch nicht im Besitz aller nötigen Betriebsmittel
- **Bereit** *(Ready)*
  Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
- **Laufend** *(Running)*
  Prozess wird vom realen Prozessor ausgeführt
- **Blockiert** *(Blocked/Waiting)*
  Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabe-operation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht);
  zum Warten wird er blockiert

- **Beendet** *(Terminated)*
  Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

1.3.10 Zustandsdiagramm der Prozesszustände



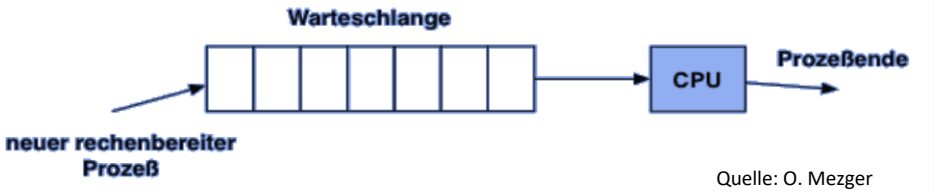
Nach Silberschatz, 1994

1.3.11 Scheduling Auswahlstrategien

Es gibt verschiedene Kriterien für die Auswahl durch den Scheduler. In Bezug auf die CPU-Auslastung ist eine 100% ausgelastete CPU gewünscht. Weiterhin gilt der Durchsatz als Kriterium. Pro Zeiteinteilung soll ein Optimum an bearbeiteten Prozessen erreicht werden. Auch die Verweilzeit kann betrachtet werden, sodass eine möglichst geringe Gesamtbearbeitungszeit eines Prozesses vorliegt. Ein weiteres Kriterium ist die Wartezeit. Eine geringe Verweildauer eines Prozesses im Zustand „bereit“ ohne Abarbeitung soll erzielt werden. In Bezug auf die Antwortzeit gilt für den interaktiven Betrieb eine schnelle Reaktion auf Eingaben. Außerdem sind First Come First Served, Shortest Job First, Prioritäts-Scheduling (non präemptiv), Round Robin und Prioritäts-Scheduling (präemptiv) mit Multilevel-Queue Scheduling oder Multilevel-Feedback-Queue Scheduling Kriterien für die Auswahl durch den Scheduler.

1.3.11.1First Come First Served

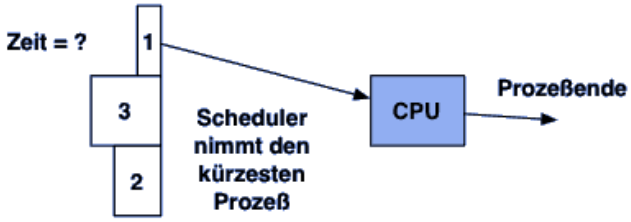
First Come First Served (FCFS) wird auch First In First Out (FIFO) genannt. Herbei geht es um die Eingangsreihenfolge, sodass der erste eingetroffene Befehl auch als erstes abgearbeitet wird. Ein Prozess verfügt über den Prozessor, bis er beendet ist, erst dann kann ein neuer Prozess gestartet werden. Der Vorteil dieses Prinzips ist die Einfachheit. Jedoch sind hierbei lange Antwortzeiten möglich.



Quelle: O. Mezger

1.3.11.2Shortest Job First

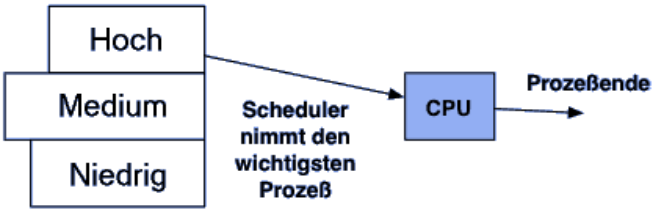
Bei Shortest Job First (SJF) erfolgt die Auswahl anhand der Prozesslänge. Das Ziel ist die Gesamtwartezeit aller Prozesse zu minimieren. Die Voraussetzung für dieses Prinzip ist, dass sich die Rechenzeit der Prozesse abschätzen lässt, damit die CPU den kürzesten Prozess zuerst nimmt.



Quelle: O. Mezger

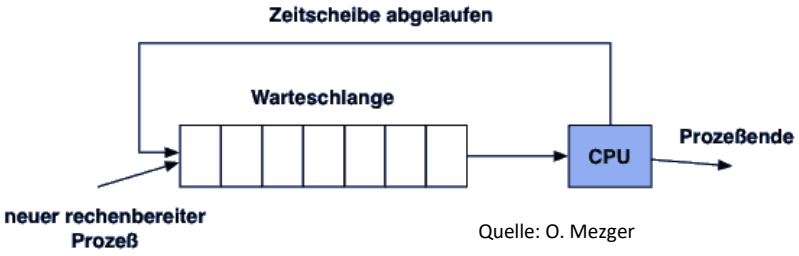
1.3.11.3Prioritäten

Bei Highest Priority First (nicht präemptiv) (HPF-n) wird nach dem Ende eines Prozesses aus den wartenden Prozessen der mit der höchsten Priorität ausgewählt und gestartet. Die Probleme dieses Prinzips bestehen in der Prioritätsumkehr *(Priority Inversion)* und der Aushungerung *(Starvation).*



1.3.11.4Round Robin

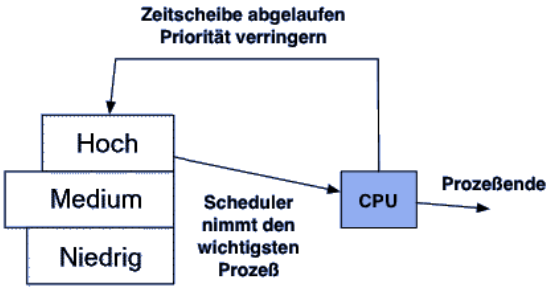
Round-Robin-Scheduling (RR) ist ein Zeitscheibenverfahren. Alle rechenbereiten Prozesse werden in einer Warteschlange angeordnet. Der vorderste Prozess wird aus der Schlange genommen und bekommt ein festes Intervall (eine Zeitscheibe) Prozessorzeit. Ist er nach dieser Zeit nicht fertig, wird er wieder hinten in die Schlange angestellt. Neue Prozesse werden ebenfalls hintenangestellt. Die Implizite Annahme besteht darin, dass alle Prozesse gleich wichtig sind.



Quelle: O. Mezger

1.3.11.5Prioritäten-Scheduling

Beim Prioritäts-Scheduling (präemptiv) wird das Prinzip Highest Priority First (HPF-p) angewendet. Prozesse werden nach ihrer Priorität ausgewählt. Nach Ablauf der Zeitscheibe wird die Priorität verringert. Die neuen Prioritäten veranlassen daraufhin einen Prozesswechsel. Bei Fällen gleicher Priorität wird Round-Robin als Standard-Verfahren eingesetzt. Nach einer "Epoche" (relativ lange Zeiteinheit bzgl. Zeitscheibe) werden die Prioritäten wieder erhöht. Dies nennt man Aging.



1.3.11.6Multilevel-Queue Scheduling

Das Multilevel-Queue Scheduling wird durch verschiedene Scheduling-Klassen wie Hintergrundprozesse (Batch) oder Vordergrundprozesse (interaktive Prozesse) realisiert. Zwischen den Klassen gibt es als Scheduling-Strategie feste Prioritäten, wobei Vordergrundprozesse immer vor Hintergrundprozessen abgearbeitet werden. Jede Klasse besitzt ihre eigenen Warteschlangen, die sogenannten Queues, und verwaltet diese nach einer eigenen Scheduling-Strategie (Round Robin).

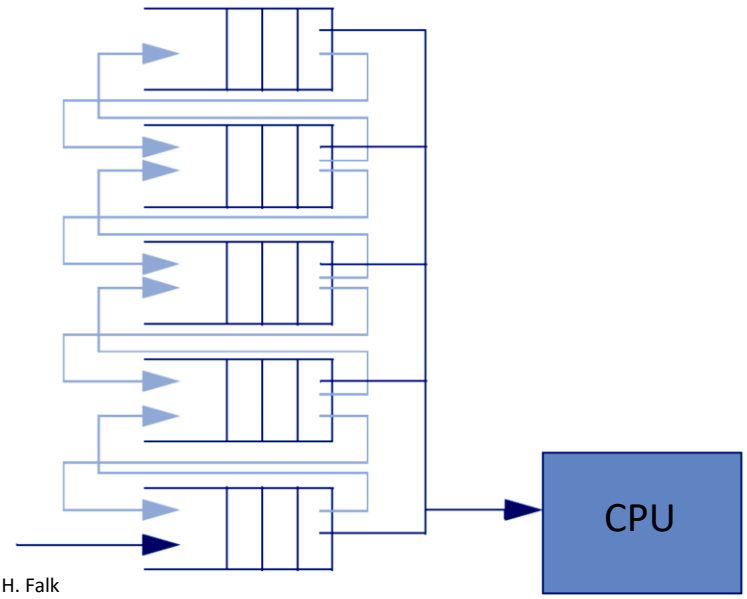
1.3.11.7Multilevel-Feedback-Queue Scheduling

Das Multilevel-Feedback-Queue Scheduling funktioniert ähnlich wie das Multilevel-Queue Scheduling. Auch hier erfolgt eine Einteilung in Scheduling-Klassen, wobei die Scheduling Strategie innerhalb der Klassen gleich ist (z.B. Round Robin). Die Auswahl der Scheduling-Klassen erfolgt mittels Prioritäten

(z.B. 1. Klasse vor 2. Klasse). Zusätzlich gibt es bei der Multilevel-Feedback-Queue ein Scheduling (MLFB). Der Transfer von Prozessen von einer Klasse in die andere wird als Feedback bezeichnet.

Beispiel:

- fünf Scheduling-Klassen mit eigener Priorität, Queue und Strategie
- Transfer von Prozessen zwischen den fünf Scheduling-Klassen möglich



Quelle: H. Falk

1.3.12 Multitasking für mehrere Prozesse einer Anwendung

Die Aufgabenverteilung in modernen Betriebssystemen wird in aller Regel auf mehrere Prozesse verteilt. Diese Prozesse nutzen gemeinsame Betriebsmittel und kooperieren dabei untereinander.

Die Aufteilung einer Anwendung in Tasks kann ebenfalls mittels mehrerer Prozesse realisiert werden. Bei geeigneter Hardware ist die parallele Ausführung von Prozessen durchführbar. Solche Multiprozessor-Systeme werden etwa in Hochleistungsrechnen (Simulationen, Wettermodelle etc.) genutzt. Ein weiteres Beispiel sind Client-Server Anwendungen unter UNIX/LINUX die pro Verbindung einen eigenen Server-Prozess starten. Auch die Implementation von Echtzeitfähigkeiten in bestehende Betriebssysteme wird über geeignete Prioritätsgesteuerte Prozesse realisiert.

Die Kommunikation zwischen kooperierenden Prozessen einer Anwendung kann entweder über die gemeinsame Nutzung von Speicherbereichen oder die Inter Prozess Kommunikation (IPC) erfolgen.

Besondere Gesichtspunkte der Aufteilung einer Anwendung in unabhängige Tasks sind die einfache Implementation, die parallele Abarbeitung und Priorisierung von Tasks. Die Prozessverwaltungsstrukturen bedingen einen gewissen Overhead und Prozesswechsel sind aufwändig und nur in mehreren Schritten realisierbar. Alternativen zu Tasks stellen Threads (Aktivitätsträger), User-level Threads, Kernel-level Threads und Lightweight Processes (LWP) dar.

1.3.13 Threads

Der Grundgedanke der Threads ist die gemeinsame Nutzung von Ressourcen in Hinblick auf Instruktionen, Datenbereiche und E/A Ressourcen. Hingegen wird eine separate Nutzung von Programmzählern, Registersätzen und Stackbereichen angestrebt.

Das Umschalten zwischen zwei Threads einer Prozess-Gruppe kann einfacher erfolgen als ein Prozess-Kontextwechsel. Dabei müssen Programmzähler, Registersätze und Stackbereiche gewechselt werden. Jedoch bleiben die Datenspeicherinhalte unberührt und alle E/A Ressourcen bleiben weiterhin verfügbar.

1.4.12.1User-Level Threads

Die Implementierung von User-Level Threads erfolgt, indem Instruktionen im Anwendungsprogramm zwischen den Threads hin- und her schalten, ähnlich wie der Scheduler im Betriebssystem. Das Betriebssystem sieht nur einen Single-Thread-Prozess.

User-Level Threads bringen einige Vorteile mit sich. Beispielsweise sind keine Systemaufrufe zum Umschalten von Threads erforderlich und sie können auf jedem Betriebssystem ausgeführt werden. Weiterhin sind für das Thread-Wechseln keine Kernelmodus-Berechtigungen erforderlich. Thread-Wechsel obliegen dem Anwender und dessen Implementation (Scheduling-Strategie).

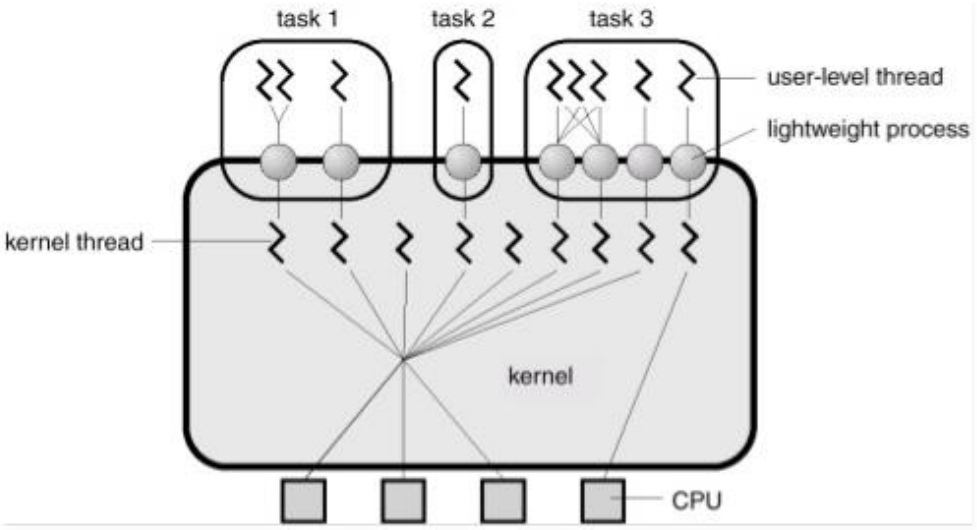
Jedoch gibt es auch einige Nachteile. Multithread-Anwendungen können Multiprocessing nicht nutzen. Außerdem wird der gesamte Prozess blockiert, wenn ein Thread einen Blockierungsvorgang ausführt.

1.4.12.2Kernel-Level Threads

Für die Implementierung von Kernel-Level Threads werden die Threads auf Kernelebene direkt vom Betriebssystem verarbeitet und die Threadverwaltung erfolgt durch den Kernel. Vorteile von Kernel-Level Threads sind, dass keine Blockierung unbeteiligter Threads bei blockierenden Systemaufrufen erfolgt. Weiterhin wird die Multiprozessor Hardware mit echten parallelen Abläufen unterstützt und das Betriebssystem kann ebenfalls Multithreading unterstützen. Nachteilig ist, dass ein Moduswechsel in den Kernelmodus für Threadumschaltung erforderlich ist. Auch ein gewisses Fairnessverhalten zwischen Prozessen mit vielen und solchen mit wenigen Threads ist nötig. Ebenfalls nachteilig ist, dass die Scheduling-Strategie meist vom Betriebssystem vorgegeben wird.

1.4.12.3 Lightweight Processes (LWP)

LWP ist eine Implementation wo das Scheduling und die Ausführung durch den Kernel erfolgt, jedoch die sonstige Implementation denen der User Level Threads entspricht.



Sun Solaris LWP nach Silberschatz 1999

1.4.13 Parallelität und Nebenläufigkeit

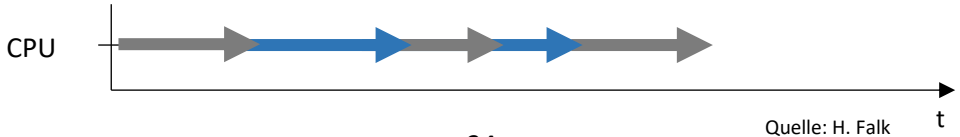
Durch die Anwendung von Technologien wie Mulitprocessing (Multitasking), Multithreading und Hyperthreading lässt sich die Rechenleistung signifikant erhöhen und es können echte parallele Aufgaben bzw. quasi-parallele Aufgaben in Nebenläufigkeit ausgeführt werden.

1.4.13.1Parallelität bei mehreren Prozessen oder Threads

Die Anweisungen zweier Prozesse/Threads werden parallel bearbeitet, wenn die Anweisungen unabhängig voneinander zur gleichen Zeit ausgeführt werden. Dabei erfolgt nur eine echte parallele Bearbeitung auf geeigneter Hardware wie Multiprozessoren/Mehrkernprozessoren. Eine parallele Abarbeitung von Befehlen auf Singleprozessoren ist nicht möglich.

1.4.13.2Nebenläufigkeit

Zwei Prozesse heißen nebenläufig, wenn ihre Anweisungen unabhängig voneinander abgearbeitet werden, wobei eine parallele Bearbeitung immer nebenläufig ist. Durch eine zeitliche Begrenzung einzelner Prozesse oder andere geeignete Scheduling Techniken ist die nebenläufige Bearbeitung auch auf Singleprozessoren möglich.



Quelle: H. Falk

1.4.13.3Probleme, die in Nebenläufiger Abarbeitung auftreten

Die in einer höheren Programmiersprache aufgeführten Befehle werden bei Abarbeitung nicht atomar (unteilbar) abgearbeitet, da die einzelnen Befehle in der Regel in mehrere Maschinenbefehle aufgeteilt werden. Bei einem Prozesswechsel innerhalb einer Befehlsausführung, in der höheren Programmstruktur oder zwischen zwei aufeinanderfolgenden Befehlen können Inkonsistenzen auftreten, die im schlimmsten Fall zu einem Absturz führen.

Grund der Inkonsistenzen ist in so einem Fall die gemeinsame Nutzung von Daten und Ressourcen. Dieses Problem kann man mit der Einführung sogenannter kritischer Abschnitte (Critical Sections) beseitigen. Man nutzt dabei die Methode des wechselseitigen Ausschlusses (Mutual Exclusion, Mutex). Ein Prozess/Thread erhält nur im wechselseitigen Einverständnis Zugang zu Daten und Ressourcen. Damit erreicht man, dass kritische Abschnitte zeitlich unteilbar (atomar) werden. Notwendig dafür sind Implementation die verhindern, dass mehrere Prozesse gleichzeitig im kritischen Abschnitt sind.

Damit dienen kritische Abschnitte der Beschränkung und Koordinierung der Nebenläufigkeit durch die Methode des wechselseitigen Ausschlusses in dem sie die zeitliche teilbare Ausführung von Befehlen höherwertiger Programmiersprachen einschränken, die Nebenläufigkeit für den Zeitraum der Abarbeitung dieser kritischen Abschnitte außer Kraft setzen und die Unabhängigkeit bei der Befehlsausführung unterbinden.

1.4.13.4Spezialfall Hyperthreading

Hyper-Threading Technology (kurz HTT, üblicherweise nur Hyper-Threading und dann HT genannt) ist eine spezielle Implementierung von hardwareseitigem Multithreading in Intel-Prozessoren, die auch von AMD übernommen wurde. Hierbei wird die Idee genutzt, die Rechenwerke eines Prozessors besser auszulasten, indem zwei Threads sich die Ressourcen teilen, die für einen vollständigen Kern notwendig wären. Hyper-Threading teilt mehrere vollständige Registersätze und ein komplexes Steuerwerk intern parallel arbeitenden Pipeline-Stufen (Siblings) mit zwei parallelen Befehls- und Datenströmen zu.

Beim Hyper-Threading werden die CPU-Ressourcen in drei Kategorien eingeteilt. Einmal in die *replicated resources* (replizierte Ressourcen). Diese werden von jedem Sibling unabhängig in eigener Kopie vorgehalten. Dazu zählen in jedem Fall der vollständige Registersatz inklusive Stackpointer und Befehlszähler. Außerdem in *partitioned resources* (unterteilte Ressourcen). Diese werden durch Unterteilung zwischen den Siblings aufgeteilt. Das heißt, sie sind zwar nur einmal vorhanden, aber einzelne Teile der Ressourcen lassen sich genau einem Sibling zuordnen. Zu diesen gehören die Instruction Queues, der Reorder Buffer und die Load/Store Buffer. Die letzte Kategorie bilden *shared resources* (geteilte Ressourcen). Alle übrigen Ressourcen gehören zu den geteilten Ressourcen und werden von beiden Siblings benutzt, meist dergestalt, dass sie nur von einem der Siblings gleichzeitig verwendet werden können. Hierzu zählen derzeit insbesondere die Arithmetisch-logische Einheit (ALU) und die Gleitkommaeinheit (FPU).

1.4.14 Wechselseitiger Ausschluss

Inkonsistenzen durch gleichzeitige Nutzung der Variablen counter:

```
int counter = 0;
```
Main Init

```
counter = counter +1;
```
Prozess1

```
printf counter;
counter = 0;
```
Prozess2

Hierbei kann es passieren, dass der Zähler noch zählt, bevor er auf Null gesetzt wird, da im Prozess 2 zwischen den beiden Befehlsausführungen Prozess 1 bereits wieder aktiv sein kann. Der Wert von counter hängt vom Scheduling der Prozesse ab.

Zwei Prozesse wollen regelmäßig den kritischen Abschnitt aus höherer Programmierung betreten. Mit der Grundannahme, dass Maschinenbefehle unteilbar (atomar) sind, wäre eine Möglichkeit die Nutzung einer gemeinsamen Variable:

```
int turn = 1;
```
Main Init

```
do {
   while (turn == 2);
   /* critical section */
   turn = 2;
}
```
Prozess1

```
do {
   while (turn == 1);
   /* critical section */
   turn = 1;
}
```
Prozess2

Das Problem dieser Lösung ist, dass der jeweils nächste Prozess nur dann den kritischen Abschnitt betreten kann, wenn der andere gegenläufige Prozess diesen durchlaufen hat. Die Implementierung ist somit unvollständig und kann durch das Ersetzen von turn durch zwei unabhängige Variablen (Thread-Zustand-Flags) behoben werden. *prepare1* zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist. Er kann diesen aber nur betreten, wenn Prozess 2 nicht ebenfalls diesen Zustand schon besitzt. *prepare2* zeigt an, dass Prozess 2 bereit für den kritischen Abschnitt ist. Er kann diesen aber nur betreten, wenn Prozess 1 nicht ebenfalls diesen Zustand schon besitzt.

1.4.14.1 Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);
   /* critical section */

   prepare1 = false;
}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);
   /* critical section */

   prepare2 = false;
}
while (true);
```

Prozess2

In dieser Implementation wird ein wechselseitiger Ausschluss erreicht und die Prozesse können unabhängig von der vorhergehenden Ausführung auch mehrfach abgearbeitet werden. Ein großes Problem dieser Lösung ist aber die Möglichkeit der sogenannten Prozess Verklemmung (Deadlock).

1.4.14.2 Prozess Verklemmung (Deadlock)

Die Prozess Verklemmung (Deadlock) erfolgt durch ein wechselseitiges Setzen der Zustandsvariablen.

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);
   /* critical section */

   prepare1 = false;
}
while (true);
```

Prozess1

Verklemmung (Deadlock)

```
do {
   prepare2 = true;
   while (prepare1);
   /* critical section */

   prepare2 = false;
}
while (true);
```

Prozess2

Die Lösung für dieses Problem ist die Implementation des Algorithmus von Peterson (1981).

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```
Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);
   /* critical section */
   prepare1 = false;
}
while (true);
```
Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);
   /* critical section */
   prepare2 = false;
}
while (true);
```
Prozess2

Der Algorithmus von Peterson implementiert einen sicheren wechselseitigen Ausschluss, indem mit Hilfe der Zustandsvariable *turn* entschieden wird, welcher Prozess nun wirklich den kritischen Abschnitt betreten darf. Entgegen der ersten Implementierung mit *turn* ist es für einen Prozess jedoch nicht notwendig, dass der vorherige Prozess schon einen Durchlauf hatte, da hier zusätzlich die Thread Flags mit ausgewertet werden. Das Problem dieser Lösung ist die Implementation für mehr als zwei Prozesse. Diese gestaltet sich dann wiederrum sehr aufwendig.

1.4.14.3 Spinlocks

Das Problem der vorgestellten Lösungen ist jeweils das Problem des aktiven Wartens in dem Prozess, der blockiert wird. Dies verbraucht unter anderem Ressourcen wie Rechenleistung. Das Problem des aktiven Wartens bezeichnet man auch als Spinlock.

```
bool prepare1 = false;
bool prepare2 = false;
```
Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```
Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```
Prozess2

1.4.15 Semaphore

Semaphor (von altgriechisch ‚Zeichen' + ‚tragen' – „Zeichenträger") ist eine Datenstruktur mit einer Initialisierungsoperation und zwei Nutzungsoperationen. Die Datenstruktur besteht aus einem Zähler und einer Warteschlange für die Aufnahme blockierter Prozesse. Der Zähler sowie die Warteschlange sind geschützt und können nur über die Semaphoroperationen verändert werden.

Die Funktion der Semaphore wird in zwei Bestandteile aufgeteilt. Zum einen regeln Semaphore die Wechselwirkungssituationen von Prozessen durch Zähler. Zum anderen realisieren sie ein passives Warten der Prozesse, wenn eine Weiterausführung nicht gestattet werden kann.

Die Semaphor-Implementierung im Betriebssystem bringt einige Vorteile mit sich. Der Scheduler des Betriebssystems wird in die Semaphor-Operationen mit eingebunden. Im Gegensatz zum Lock bzw. Mutex brauchen die Threads, die „reservieren“ und „freigeben“, nicht identisch zu sein. Weiterhin beheben Semaphore den Nachteil des aktiven Wartens anderer Synchronisationslösungen (Spinlocks). Ein Prozess, der blockiert ist, wird bis der Blockadegrund entfallen ist, in eine Warteschlange geschoben. Die Blockierzeit der Prozesse der Warteschlage kann durch andere Prozesse genutzt werden.

Semaphore als Mechanismus für die Prozesssynchronisation wurden von Edsger W. Dijkstra konzipiert und 1965 in „Cooperating sequential processes“ vorgestellt. Die Nutzungsoperationen wurden von Dijkstra mit P und V bezeichnet:

- P-Operation = Wait (warten)
- V-Operation = Release (freigeben)

Die P-Operation wird wie folgt aufgerufen: Der Zähler wird dekrementiert. Wenn der Zähler größer gleich 0 ist, setzt der Prozess seine Aktionen fort. Wenn der Zähler kleiner als 0 ist wird der Prozess blockiert und in die Warteschlange eingereiht.

Die V-Operation wird wie folgt aufgerufen: Der Zähler wird inkrementiert. Der Prozess wird aus der Warteschlange entnommen, entblockiert und fortgesetzt.

1.4.15.1 Szenario I - Recht auf Betreten eines kritischen Abschnitts

Gemeinsam von A und B genutzter Semaphor: `mutex`
Initialisierung: `init (mutex, 1)`

```
Prozess A
...

P(mutex)
… /* kritischer_Abschnitt_A */
V(mutex)

...
```

```
Prozess B
...

P(mutex)
… /* kritischer_Abschnitt_A */
V(mutex)

...
```

1.4.15.2 Szenario II - Semaphor zur Betriebsmittelverwaltung

Zur Betriebsmittelverwaltung genutzter Semaphor: `s_available`
Initialisierung: `init (s_available, n)`

```
Prozess

...

P (s_available)

… /* Nutzung des Betriebsmittels */

V (s_available)

...
```

1.4.15.3 Szenario III – Semaphor in Signalisierungsfunktion

Gemeinsam von A und B genutzter Semaphor: `s_inform`
Initialisierung: `init (s_inform, 0)`

```
Prozess A

...

C_I

V (s_inform)

...
```

```
Prozess B

...

P (s_inform)

C_V

...
```

Prozess B erst nach Prozess A ausführbar

1.5 Speicherverwaltung und Speicherzugriffe

1.5.12 Speicherverwaltung

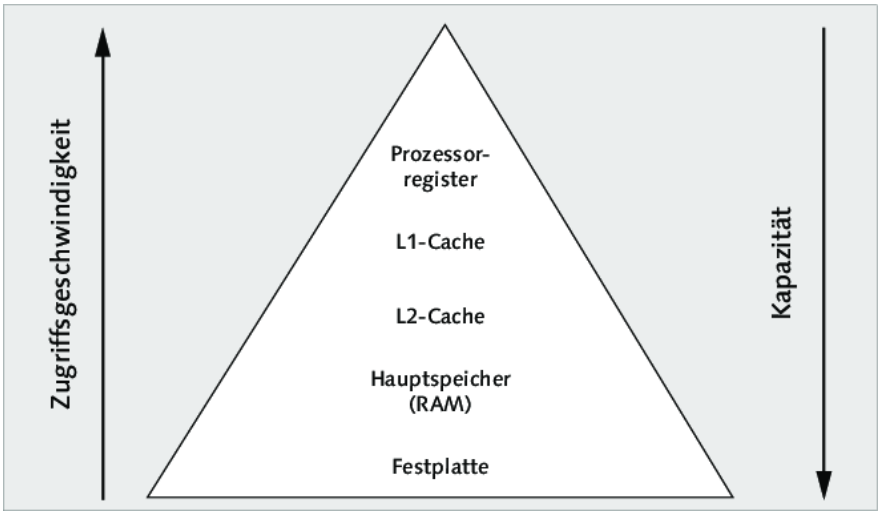
1.5.12.1Hauptspeicher – Random Access Memory RAM

Die Größe des Hauptspeichers auf einem 32 Bit System beträgt 2^32 Byte was insgesamt 4GiB ausmacht, wohingegen er auf einem 64 Bit System 2^64 Byte groß ist, was insgesamt 18EiB ausmacht. Die verschiedenen Maschinenbefehle können auf diese Adressen zugreifen und die dort gespeicherten Bytes lesen oder schreiben. Während der Prozessabarbeitung referenziert das Befehlsregister den nächsten auszuführenden Befehl, indem es dessen Hauptspeicheradresse speichert.

1.5.12.2Cachespeicher

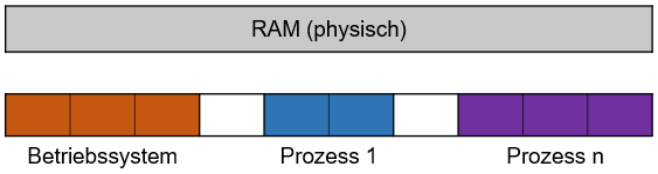
Der Hauptspeicher ist langsamer als die Abarbeitung von Befehlen im Prozessor. Aufgrund dessen erfolgte eine Einführung von Pufferspeichern, sogenannten Caches, um Zugriffszeit auf häufig benutzte Datensätze und Variablen des Hauptspeichers zu verkürzen. Diese werden in verschiedenen Leveln (L1 Cache, L2 Cache und L3 Cache) aufgeteilt. Caches befinden sich entweder direkt auf dem Prozessor-Chip (L1-Cache) oder „direkt daneben“. Sie sind zwischen ein paar Kilobytes und wenigen Megabytes groß. Gepufferte Hauptspeicherwerte sind transparent zwischengespeichert, sprich der Prozess greift auf Adressen im Hauptspeicher zu, ohne dass ein Cache für ihn ersichtlich ist.

1.5.12.3Speicherpyramide



Quelle: Linux, Plötner et.al., Rheinwerk Computing

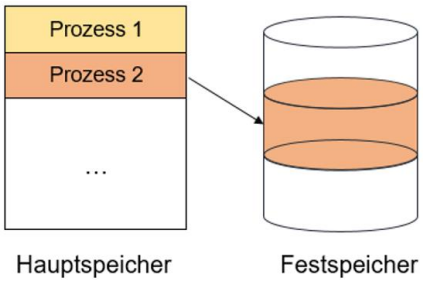
Der Hauptspeicher ist in mehrere Bereiche aufgeteilt. Zum einen in den Speicherbereich für das Betriebssystem (Programm, Puffer, Variablen) und zum anderen in den Speicherbereich für Prozesse (Programm und Daten).



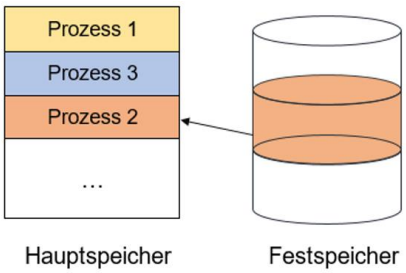
Die Speicherverwaltung stellt die statische oder dynamische Aufteilung des Speichers für aktuell gestartete Prozesse bereit. Jeder Prozess bekommt einen eigenen Adressraum (virtueller Speicher). Dabei stellt ein Adressraum die Abstraktionsebene des physikalischen Speichers dar. Die Speicherzellen im Hauptspeicher haben dabei eine eindeutige Speicheradresse.

1.5.12.4Relokation / Swapping

Die Relokation bezeichnet die Verlagerung und das Swapping das Austauschen. Zweck ist es, mehrere Prozesse gleichzeitig im System zu halten.

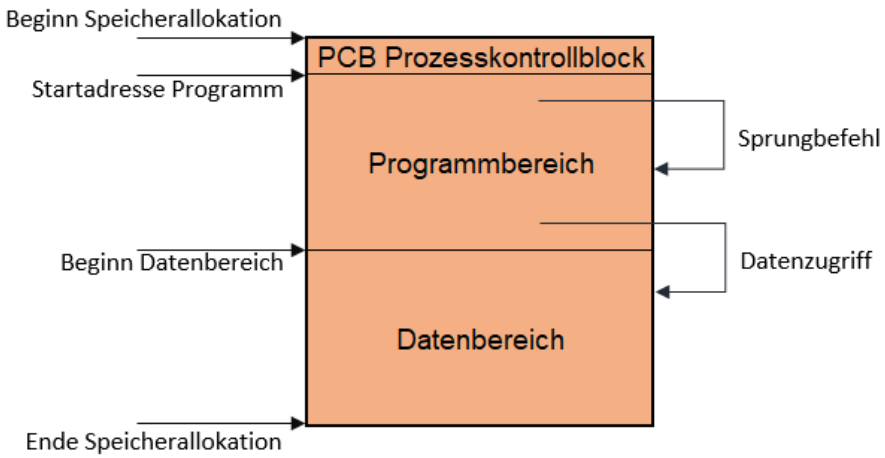


Auslagern eines Prozesses aus dem Hauptspeicher in den Festspeicher



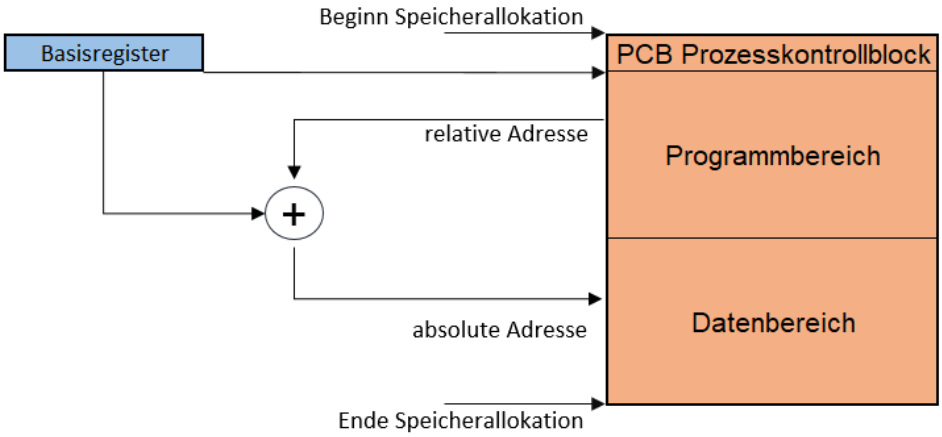
Wiedereinlagern eines Prozesses aus dem Festspeicher in den Hauptspeicher

Probleme stellen hierbei aber absolute Sprungbefehle in den Prozessorregistern der einzelnen Prozesse dar. Nicht jeder Prozess beginnt an Hauptspeicher Adresse 0x0000 0000. Deshalb ist ein Konzept zur Verwendung von relativen Speicheradressen notwendig.



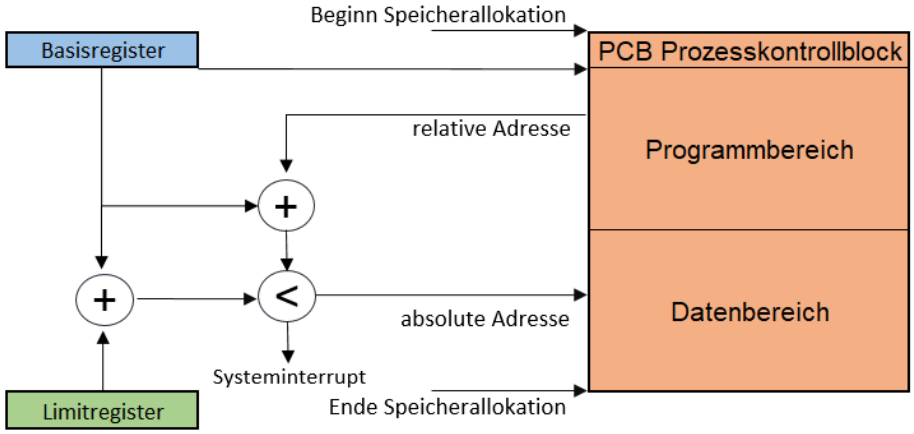
1.5.12.5Lokationskonzept

Die absolute Adresse wird über ein Basisregister realisiert:



1.5.12.6Schutzkonzept

Mit Hilfe des Limitregisters ist ein Schutzmechanismus der einzelnen Prozesse realisierbar:

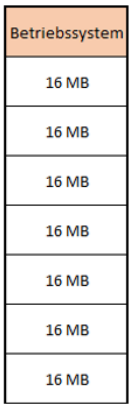


1.5.12.7Partitionierung

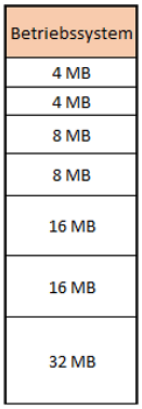
Die Aufteilung des Speichers in einzelne Bereiche mit definierten Grenzen nennt man Partitionierung. Die Partitionierung ist bereits aus dem Bereich der Dateisysteme bekannt. Hierbei wird ein zusammenhängender Teil des Hauptspeichers fest für das Betriebssystem reserviert. Ein weiterer zusammenhängender Teil des Speichers wird für jeden einzelnen Prozess genutzt. Realisiert wird dies entweder mit der Variante der statischen Partitionierung oder mit der dynamischen Partitionierung.

Statische Partitionierung

Bei der statischen Partitionierung geht es um die Aufteilung des Speichers in eine feste Anzahl von Fragmenten mit zwei möglichen Varianten:



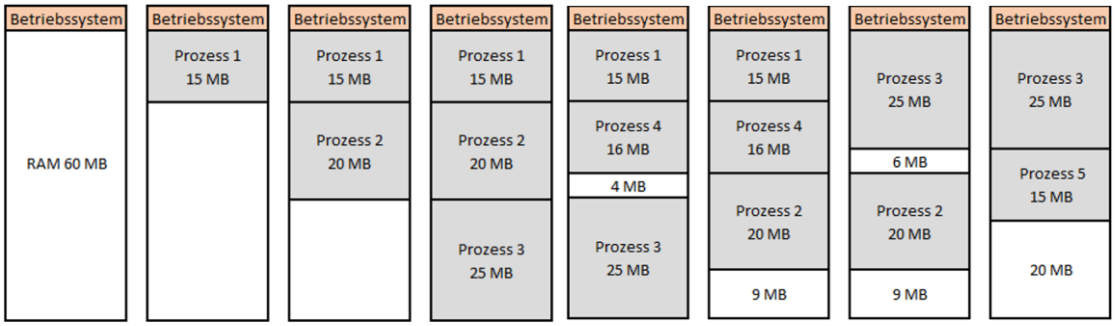
Partitionen mit fester Fragmentgröße



Partitionen mit unterschiedlicher Fragmentgröße

Dynamische Partitionierung

Es gibt verschiedene Strategien für die Auswahl des freien Speicherblocks bei der dynamischen Partitionierung: Zum einen die Variante *Best Fit*. Hierbei erfolgt die Suche nach dem kleinsten Block, der den Prozess fasst. Bei *First Fit* geht es um die Suche nach dem ersten Block der groß genug ist den Prozess aufzunehmen. *Next Fit* befasst sich mit der Suche nach dem ersten Block der groß genug ist den Prozess aufzunehmen beginnend ab letzter Speicherzuweisung.



1.5.12.8Das virtuelle Speicherkonzept

Die Probleme der bereits vorgestellten Speicherkonzepte sind, dass die physikalische Grenze des Hauptspeichers die Rahmenbedingungen für die Allokation des RAM vorgibt. Außerdem können Prozesse nur den Teil des Hauptspeichers beanspruchen, der nach dem Laden des Betriebssystems noch zur Verfügung steht. Auch die Ausführung von Prozessen mit mehr Speicherbedarf ist nur auf einem System mit mehr RAM oder besonderen Speicherumgebungen möglich (MS-DOS EMM 386 / HIGHMEM).

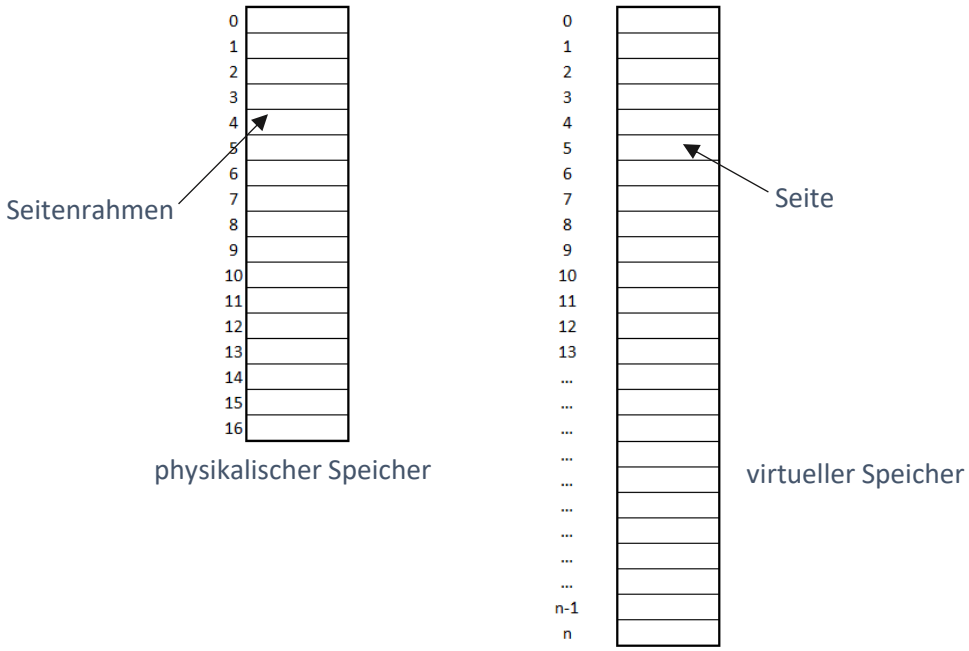
Die Lösung für diese Probleme ist die virtuelle Speicherverwaltung pro Prozess. Eine virtuelle Speicherverwaltung pro Prozess bedeutet, dass die Aufteilung des zugestandenen Speichers pro Prozess virtualisiert wird. Das Management erfolgt durch das Betriebssystem mit oder ohne Hardwareunterstützung (MMU). Die Größe des virtuell zugewiesenen Speichers ist einzig abhängig von der Bitbreite der Betriebssystemarchitektur (16 Bit = 64 kiB / 32 Bit = 4 GiB / 64 Bit = 18 EiB). Ein Prozess kann mehr virtuellen Speicher zugewiesen bekommen als real vorhanden ist (*Hauptspeicher + Hintergrundspeicher = virtueller Speicher*). Problematisch ist jedoch, dass die Einlagerung ganzer Prozesse nicht möglich ist.

Mandl 2013 fasst daher das Grundkonzept der virtuellen Speicherverwaltung etwas anders auf. Der Speicherbedarf eines Prozesses sollte größer sein können als der physikalisch vorhandene Hauptspeicher. Ein Programmierer sollte nur einen kontinuierlichen (linearen) Speicherbereich beginnend mit Adresse 0 sehen und sich nicht um die Fragmentierung des Hauptspeichers kümmern müssen. Weiterhin sollte ein Prozess auch dann noch ablaufen können, wenn er nur teilweise im Hauptspeicher ist. Wichtig ist, dass die Teile des Prozesses (Daten und Code) die gerade benötigt werden im physikalischen Speicher sind.

Für die Realisierung dieses Grundkonzeptes sind komplexere Methoden der Speicherverwaltung notwendig. Methoden wie Paging, Segmentierung und Paging mit Segmentierung werden eingesetzt.

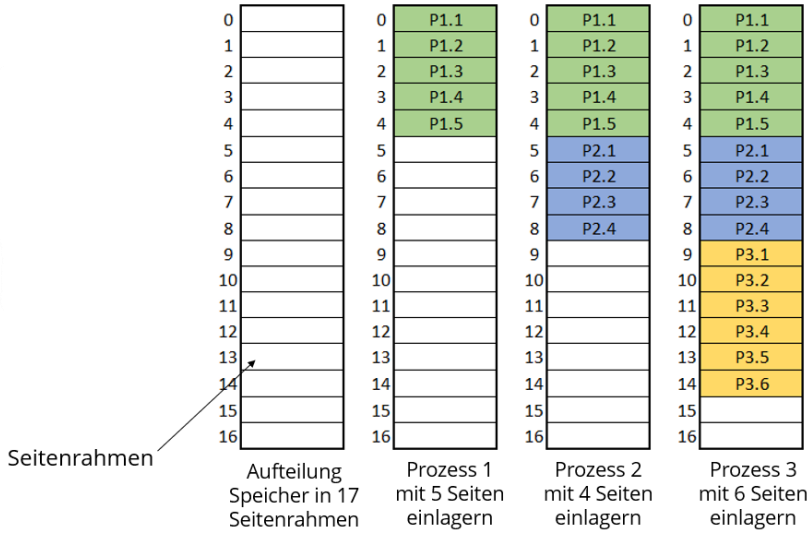
1.5.12.9Paging

Das virtuelle Speicherkonzept mit Paging ist ähnlich dem statischen Partitionieren. Der physikalische Speicher wird in gleich große Fragmente aufgeteilt, wobei die Fragmente die sogenannten Seitenrahmen (Page Frames) darstellen. Hierbei muss ein Prozess nicht vollständig in ein solchen Seitenrahmen passen. Ein Prozess erhält einen virtuellen Speicherbereich mit max. 2 ^ Bitbreite Architektur. Die Aufteilung des virtuellen Speichers eines Prozesses erfolgt dynamisch in einzelne Seiten (Pages) entsprechend der Größe eine Seitenrahmens.

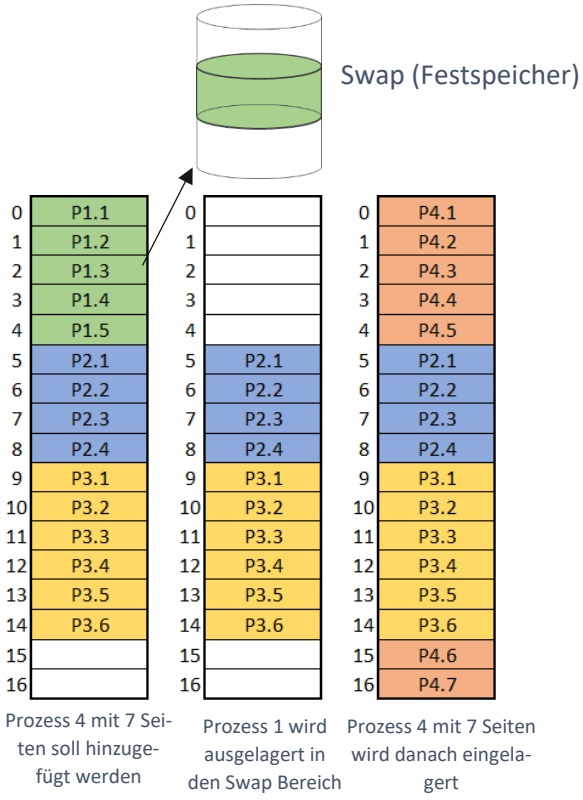


Beispiele

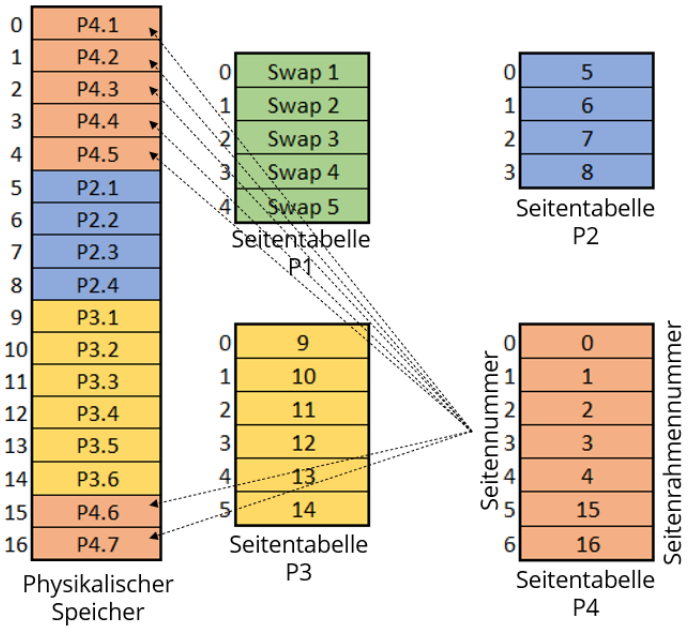
Hier im Beispiel ist dargestellt, wie die Einlagerung von Prozessen im Hauptspeicher sequenziell erfolgt, sofern der dafür notwendige Platz vorhanden ist. Die Prozesse werden vollständig eingelagert.



Hier im Beispiel wird dargestellt, wie die Einlagerung von Prozessen im Hauptspeicher fragmentiert erfolgt, wenn der dafür notwendige Platz nicht vorhanden ist. Die Prozesse müssen vorher einer Relokation / Swapping unterzogen werden.

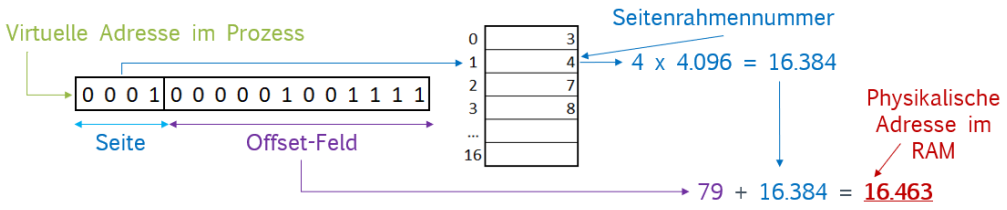


Hier im Beispiel wird dargestellt, wie die Zuordnung von Prozessfragmenten, also den Pages im Hauptspeicher, mittels Seitentabellen (Page Tables) zum jeweiligen physischen Seitenrahmen erfolgt. Jeder Prozess benötigt dazu eine eigene Seitentabelle.



Bei der Berechnung von Adressen wird zwischen virtueller Adresse und physischer Adresse unterschieden. Hierbei gilt, dass die Seitenanzahl und Seitenrahmenanzahl eine Zweierpotenz sind. Die virtuelle Adresse im Programmablauf besteht aus der Seitennummer und dem Offset und die physikalische Adresse besteht aus der Seitenrahmennummer und dem Offset.

Bei einer Bitbreite von 16 Bit kann die Aufteilung mit einer Seitengröße von 4 KiB = 2^{12} = 4.096 Bytes wie im folgenden Beispiel realisiert werden. Ein Offset-Feld von 12 Bit wird benötigt, um alle 4.096 Bytes referenzieren zu können. Ein einzelner Prozess kann an Hand der 4 Bit Seitennummer bis zu 2^4 = 16 verschiedene Seiten referenzieren, die über die Seitentabelle des Prozesses auf Seitenrahmen im Hauptspeicher verweisen.

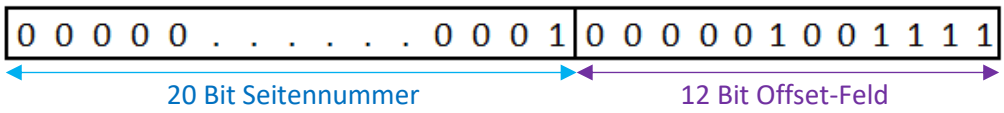


Die bisherigen Betrachtungen beliefen sich alle auf Prozesse die vollständig in den Hauptspeicher geladen werden können. Wenn jedoch weniger Hauptspeicher zur Verfügung steht als virtueller Speicher für einen Prozess zugeteilt wird, ist eine Erweiterung des Konzeptes notwendig. Dabei gibt es mehrere Voraussetzungen. Die Einlagerung von Prozessen erfolgt nur teilweise in einzelnen benötigten Seiten. Seiten werden nachgeladen, sofern die Seite nicht im Speicher eingelagert ist. Außerdem bleibt die Programmabarbeitung unbeeinträchtigt von der Seitenauslagerung. Dies führt zu einem On Demand Paging.

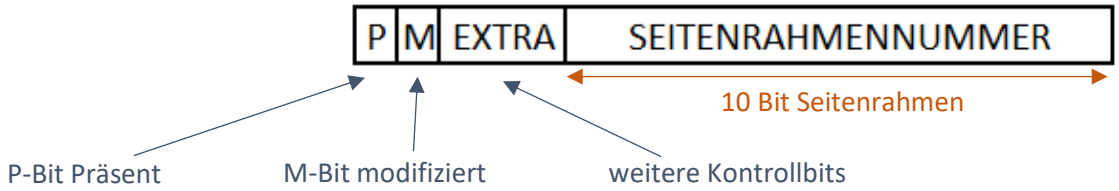
1.5.12.10 On Demand Paging

On Demand Paging bildet die Grundlage für die Umsetzung einer teilweisen Seiteneinlagerung. Dabei erfolgt eine Überwachung der Nutzung von einzelnen Seiten und deren aktueller Einlagerungszustand. Dafür gibt es zusätzliche Angaben in der Seitentabelle eines Prozesses. Weiterhin gibt es Strategien zur Ersetzung von Seitenrahmeninhalten und eine effiziente Überwachung der Lokalität bei Seitenzugriffen.

Durch die Aufteilung der Anzahl der Bits für Seiteneintrag und Offset kann eine Seitenanzahl beispielsweise von 20 Bits und ein Offset von 12 Bit für 32 Bit Systeme genutzt werden:



Der dazugehörige Eintrag in der Seitentabelle wird um Status Bits für eingelagerte Seiten (P Präsent) und geänderte Seiten (M Modifiziert) ergänzt:



Funktionsweise

Der Prozess möchte Speicherbereich der Seitentabelle aufrufen, der nicht eingelagert ist. Er wird durch einen Seitenfehler (Page Fault) Interrupt unterbrochen, sodass der Prozess eingelagert wird. Kommt es zu einer fehlenden Speicherseite im Festspeicher gibt es zwei Varianten damit umzugehen. Entweder wird eine Seite im Hauptspeicher angelegt oder es wird bei fehlender Speicherseite ein Page Fault Error ausgegeben und der Prozess gestoppt.

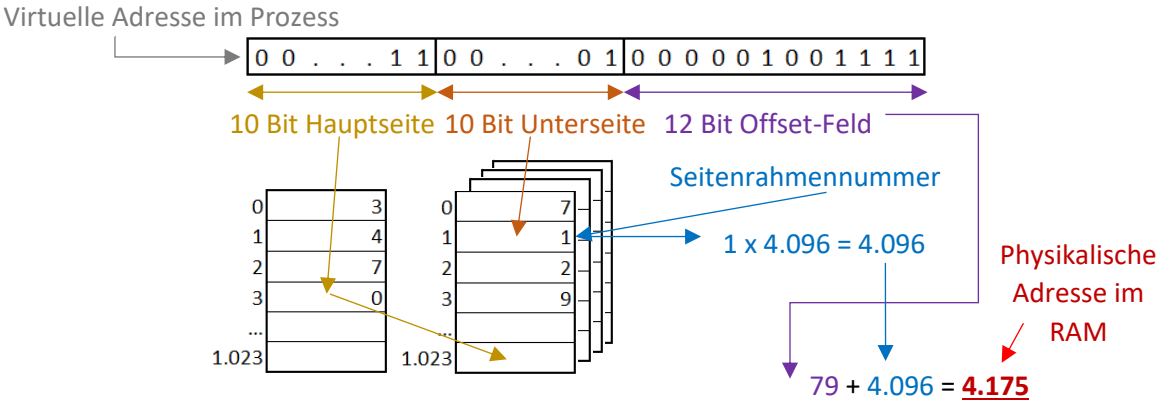
1.5.12.11 Seitenersetzungsstrategien

Wenn kein freier Seitenrahmen verfügbar ist, werden Seitenrahmen aus dem Hauptspeicher ausgelagert. Falls das M-Bit gesetzt ist, wird dieser Seitenrahmen darauf in den Festspeicher ausgelagert und der Seitentabelleneintrag entfernt. Eine Seitenrahmenverdrängung ist durch verschiedene Techniken möglich. Es kann das First in First Out (FiFo) Prinzip, das Least Recently Used (LRU) Prinzip oder das Second Chance (Clock/Uhranzeiger) Prinzip verwendet werden. Im Anschluss wird die Seite aus dem Festspeicher eingelagert und die Seitentabelle aktualisiert.

1.5.12.12 Mehrstufige Seitentabelle

Mehrstufige Seitentabellen wenden die Physical Address Extension (PAE) an. Dabei werden auch die Seitentabellen in Pages unterteilt und im virtuellen Speicher implementiert. Untertabellen sind als Pages entweder eingelagert oder ausgelagert abgelegt, wodurch keine einzelne große Seitentabelle wie im Beispiel mit 2^{20} = 1 GiB = 1.048.576 Zeilen also 256 Seitenrahmen zu je 4KiB mehr verwaltet werden muss. Damit wird die Ablage der kompletten Seitentabelle im Hauptspeicher möglich, da nur zwei Tabellen zu 2^{10} = 1 KiB = 1.024 Einträgen benötigt werden.

Das folgende Beispiel enthält 4 KiB = 4.096 große Seitenrahmen und eine Seitentabelle von zweimal 2^{10} = 1 KiB = 1.024 Einträgen.

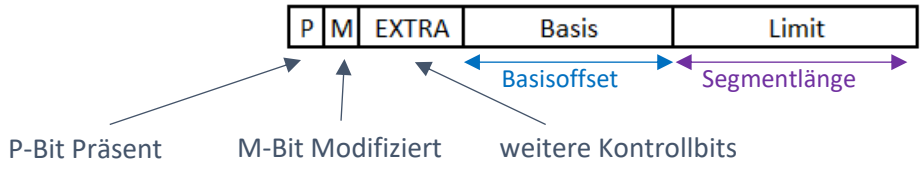


1.5.12.13 Translation Lookaside Buffer (TLB)

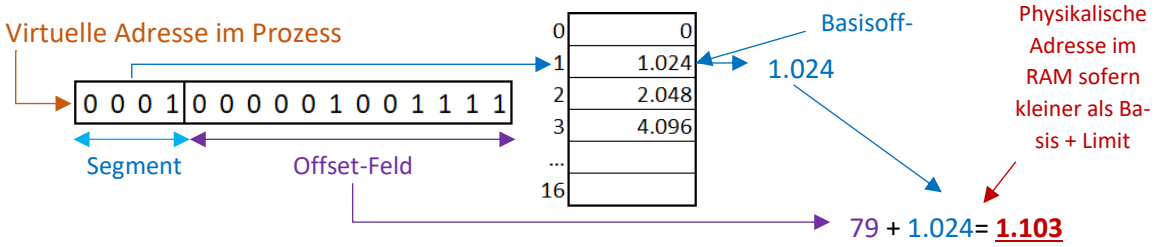
Beim Paging werden zusätzliche Speicherzugriffe auf die Seitentabellen, die ebenfalls im Speicher abgelegt werden, notwendig. Diese Speicherzugriffe erfordern Zugriffszeit und Bremsen das System aus. Eine Beschleunigung kann nur durch einen zusätzlichen Cache für Adressübersetzung (Adressumsetzungspuffer, TLB) erreicht werden. Der TLB enthält Einträge der Seitentabellen, auf die zuletzt gemäß dem örtlichen bzw. räumlichen Lokalitätsprinzip zugegriffen wurde. Bei Zugriffen auf eine nicht im TLB enthaltene Seite wird die entsprechende Seite in den TLB eingetragen.

1.5.12.14 Segmentierung

Eine weitere Möglichkeit der Verwaltung des virtuellen Speichers bietet die Segmentierung. Dabei wird der virtuelle Adressraum für den Prozess in einzelne Segmente (Programmsegment, Datensegment, Stacksegment, etc…) aufgeteilt, wobei die Segmentgröße differenzieren kann. Die Speicheranforderung pro Prozess ist nicht zwingend zusammenhängend und nicht ständig eingelagert. Ein Schutzmechanismus mittels Basis- und Limitregister ist pro Segment umsetzbar und die Speicheranpassung leicht pro Segment realisierbar.
Für jeden Prozess ist eine Segmenttabelle notwendig mit folgenden Eigenschaften eines Segmenttabelleneintrags:

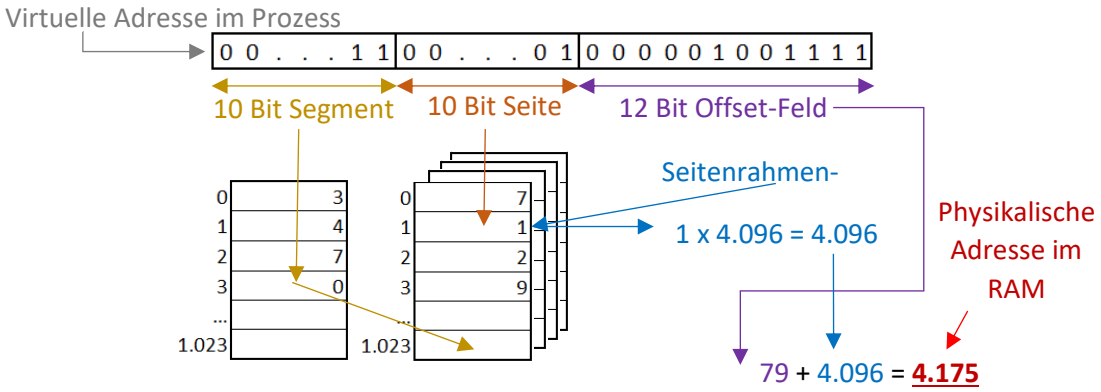


Die Umrechnung der virtuellen Adresse in die physikalische Adresse erfolgt folgendermaßen:



Um große Segmente effizient zu verwalten ist eine Kombination von Segmentierung und Paging on Demand notwendig. Dafür existieren zwei Realisierungsmöglichkeiten. Zum einen nutzt man eine Segmenttabelle pro Prozess und pro Segment eine eigene Seitentabelle (Unix). Zum anderen gibt es eine Segmenttabelle pro Prozess und eine einzelne Seitentabelle für alle Segmente (Windows).

Kombination von Segmentierung und Paging on Demand im Unix:



1.5.12.15 Besondere Techniken

PrePaging und Trashing

Beim PrePaging und Trashing geht es um die Überwachung der Lokalität bei Seitenzugriffen. Die Überwachung erfolgt zum einen zeitlich, sprich der Zugriff auf häufig genutzte Seiten von Prozessen wird überwacht. Dazu ist eine zusätzliche Überwachung des Nutzungsverhaltens von Seiten notwendig. Eine weitere Möglichkeit ist die räumliche Überwachung, wobei es um Datenzugriffe im unmittelbaren Umfeld der Daten (sequentiell in die Zukunft gerichtet) geht. Dabei ist eine vorausschauende Einlagerung von Seiten in die Seitenrahmen nötig.

Weiterhin werden zu viele Seiteneinlagerungen, Umlagerungen, Auslagerungen beim Kontextwechsel einzelner Prozesse, dem sogenannten *Trashing (Flattern)* durch zu wenig Speicher oder zu viele speicherintensive Prozesse verhindert.

Copy on write

Copy on write wird vorrangig beim Forking (Duplizieren) von Prozessen in Linux Umgebungen eingesetzt. Dabei geht es um das Verfügbar machen einer Speicherseite im Adressraum eines anderen Prozesses. Bei lesendem Zugriff ist es nicht nötig, die Daten tatsächlich zu kopieren und ein weiteres Mal

im Hauptspeicher anzulegen, sondern es genügt, wenn die beiden Prozesse auf ein und dieselbe Speicherseite zugreifen. Wenn einer der beiden Prozesse die Daten zu ändern versucht, müssen diese tatsächlich (und dann auch nur teilweise) kopiert werden.

Shared Memory

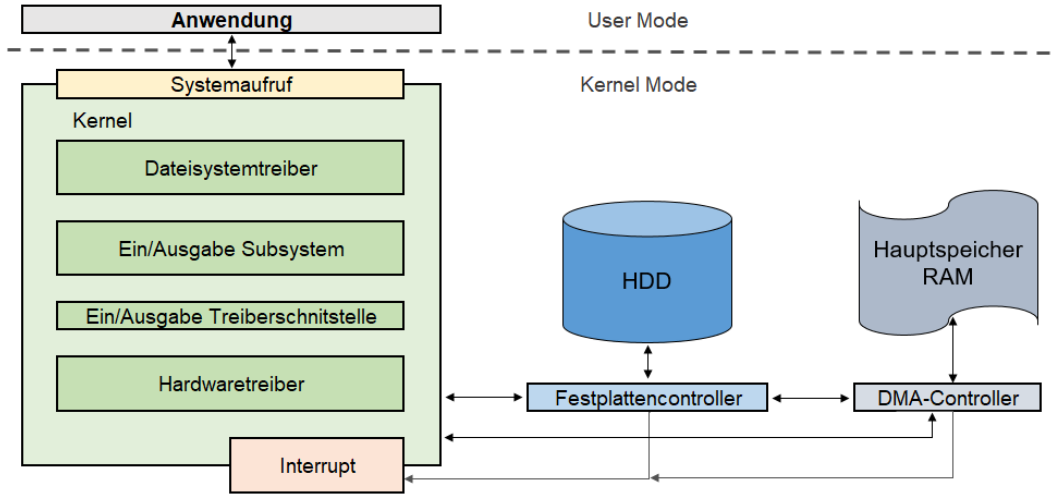
Mithilfe von Shared Memory erfolgt eine Optimierung der Speicherauslastung und des Kontextwechsels durch gemeinsam genutzte Speicherbereiche („Shared Memory“). Programmsegmente müssen nicht mehrfach im Hauptspeicher gehalten werden. Außerdem gibt es bei Anwendungen mit mehreren Daten separate Datensegmente. Auch die Einbindung von Shared Librarys (Dynamic Link – DLL Windows / Shared Library - Shared Libs Linux) ist möglich. Gemeinsam genutzte Speichersegmente befinden sich in mehreren Seitentabellen unterschiedlicher Prozesse. Zusätzlich gibt es hier ein notwendiges weiteres Steuerbit zur Erkennung, sofern ein Prozess Speicher freigibt.

Direct Memory Access

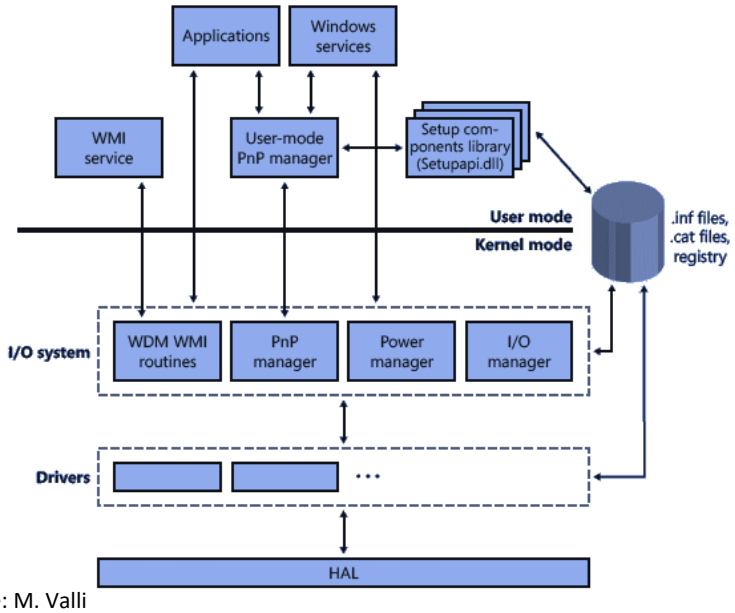
Für eine effiziente Ein- und Auslagerung von Daten aus dem Fest- in den Hauptspeicher wurden hardwareseitig Implementationen entwickelt, die einen direkten Zugriff auf den Hauptspeicher über das Bussystem ermöglichen (Direct Memory Access – DMA). Der Prozessor und damit indirekt auch Programme können nicht direkt auf die Festplatte zugreifen. Durch Programmierung des DMA-Controllers können die betreffenden Datenblöcke in vorher festgelegte Bereiche des Hauptspeichers kopiert werden. Während der DMA-Controller die Daten von der sehr langsamen Festplatte in den Hauptspeicher kopiert, kann die CPU bereits weitere Prozesse abarbeiten.

1.5.13 E/A Zugriffe auf die Festplatte

1.5.13.1Zuordnung der Festspeicher E/A Geräte



1.5.13.2Windows E/A System



Quelle: M. Valli

1.6 Paketverwaltung

Bei der Paketverwaltung geht es um die zentrale Verwaltung von Software. Diese wird in Form von Programmpaketen bereitgestellt. Weiterhin erfolgt eine Vorkonfiguration der Pakete für spezielle Betriebssysteme, OS-Versionen und Distributionen.

Die Paketverwaltung wird auf zwei Arten umgesetzt. Zum einen erfolgt die Paketverwaltung über Programme, die Pakete direkt installieren und löschen, ohne Abhängigkeiten und Kontrollprüfungen. Zum anderen kann die Paketverwaltung von Programmen durchgeführt werden, die aus anderen Quellen Daten nachladen und Abhängigkeiten und Konflikte prüfen.

Die Paktverwaltung umfasst die Installation, die Konfiguration, die Aktualisierung und die Deinstallation.

1.6.1 Umsetzung der Paketverwaltung in den jeweiligen Betriebssystemen

Die App Stores repräsentieren dabei die moderne Form der Paketverwaltung.

1.6.1.1 Umsetzung bei Windows

Bei dem Betriebssystem Windows handelt es sich um .msi-Pakete (MicroSoft Installer). Es gibt eine Laufzeitumgebung für Installationsroutinen unter Microsoft-Windows-Betriebssystemen. Die Paketverwaltung erfolgt mit dem Windows-Systemdienst, den Paketdateien und den Patchdateien. Externe Abhängigkeiten werden oftmals nicht beachtet.

1.6.1.2 Umsetzung bei macOS

Die Paktverwaltung bei macOS erfolgt mit .dmg Paketen (Apple Disk Image).

1.6.1.3 Umsetzung bei Android

Die Paketverwaltung unter Android wird mit .apk Paketen (Android Package Kit) realisiert.

1.6.1.4 Umsetzung bei iOS

Die Paktverwaltung bei iOS erfolgt mithilfe von .ipa Paketen (Apple iPhone Application).

1.6.2 Paketverwaltung unter unixioden Betriebssystemen

1.6.2.1 Debian Package Manager (dpkg)

- Paket installieren („install“)
  - `dpkg -i <Paket>.deb`
- Paket deinstallieren („remove“)
  - `dpkg -r <Paketname>`
- Pakete auflisten („list“)
  - `dpkg -l`
- Paketinhalt auflisten
  - `dpkg -L <Paketname>`

1.6.2.2 RedHat Package Manager (rpm)

- Paket installieren („install“)
  - `rpm -i <Paket>.rpm`
- Paket deinstallieren („erase“)
  - `rpm -e <Paketname>`
- Pakete auflisten („query all“)
  - `rpm -qa`
- Paketinhalt auflisten („list / look inside“)
  - `rpm -l <Paketname>`

1.6.2.3 Advanced Packaging Tool (APT)

APT stellt eine Erweiterung für dpkg dar und löst Abhängigkeiten auf.

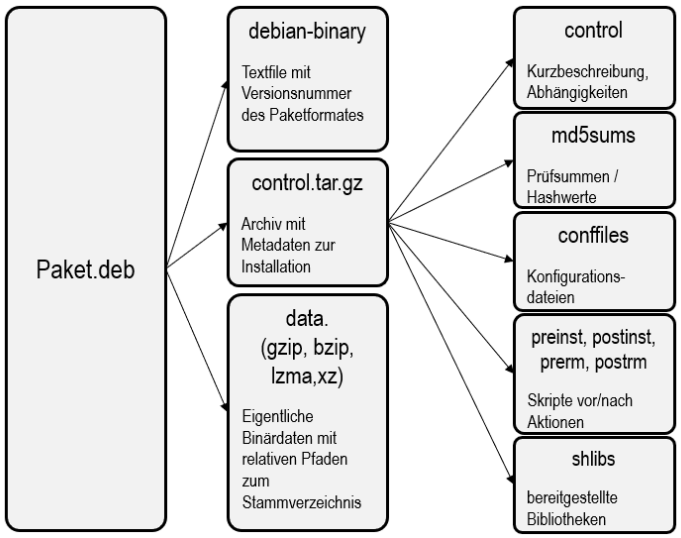
- Paketliste aktualisieren
  - `apt-get update`
- Update für alle über apt-get verwalteten/installierten Pakete
  - `apt-get upgrade`
- Upgrade der gesamten Distribution inklusive zurückgehaltener Pakete
  - `apt-get dist-upgrade`
- Paket installieren
  - `apt-get install <Paketname>`
- Paket deinstallieren

- `apt-get remove <Paketname>`

– Paket suchen
- `apt-cache search <Paketname>`

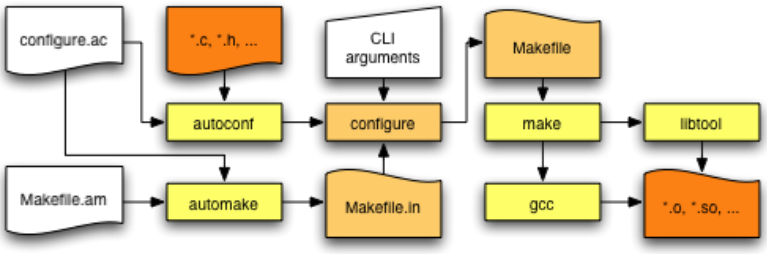
Der Nachfolger ist *apt*. Er setzt sich aus apt-get, apt-cache und Zusatzfunktionen zusammen. Der Befehlsaufbau ist gleich, jedoch bietet *apt* eine bessere Virtualisierung in Bezug auf die Farben und den Fortschrittsbalken.

1.6.3 Aufbau eines Paket-Containers am Beispiel eines .deb Paketes



1.6.4 Build-System

Build-System arbeitet nach dem Prinzip „Baue die Software spezifisch für mein Ziel-System“. Es gibt die Quelltextdateien, welche durch den Compiler (maschinenspezifisch) verarbeitet werden. Außerdem gibt es die Objektdateien, welche vom Linker verbunden bzw. eingebunden werden. Des Weiteren gibt es zusätzliche Skripte, die für die Pre- und Post-Prozesse zuständig sind. Der *make* Befehl nutzt das *Makefile* als Bauanleitung. Diese Makefiles werden meist automatisch erstellt (GNU Build System). Dazu gehören autoscan, alocal, autoheader, autoconf und automake.



Der Nutzer verwendet für die Installation von Software Quellcode anstatt bereits kompilierter Binärdaten. Dafür werden die Dateien *configure* (Skript), *Makefile.in* und *config.h.in* verwendet. Bei der Ausführung von *configure* entstehen die Dateien *config.status*, *Makefile* und *config.h.*

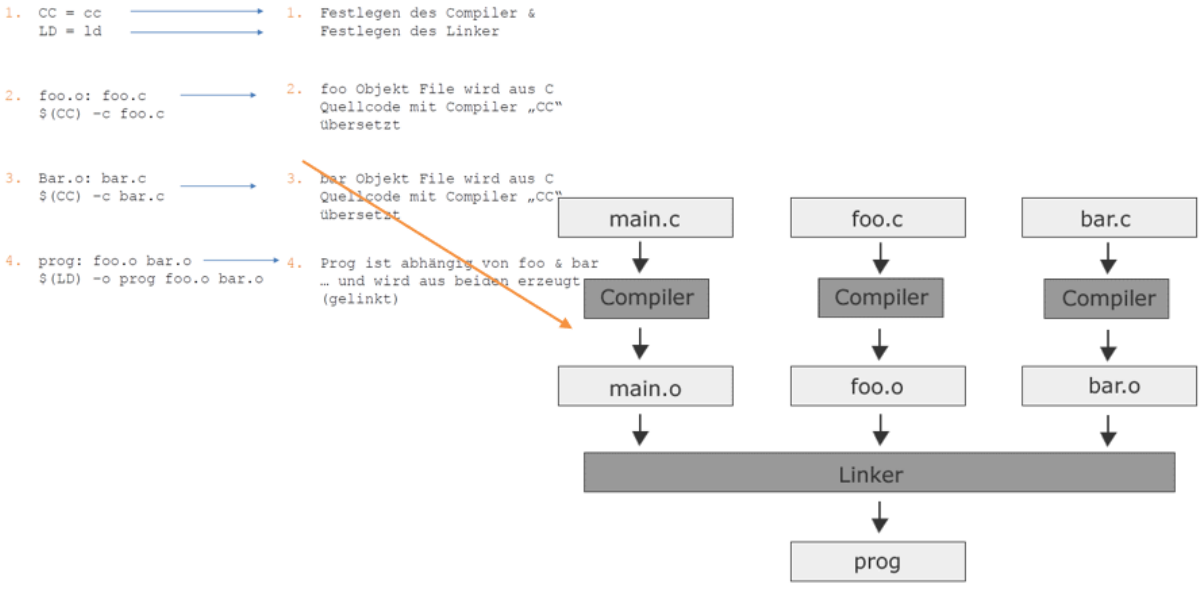
Der Standard-Ablauf für die Installation ist:

1. *./configure*
2. *make*
3. *make install*

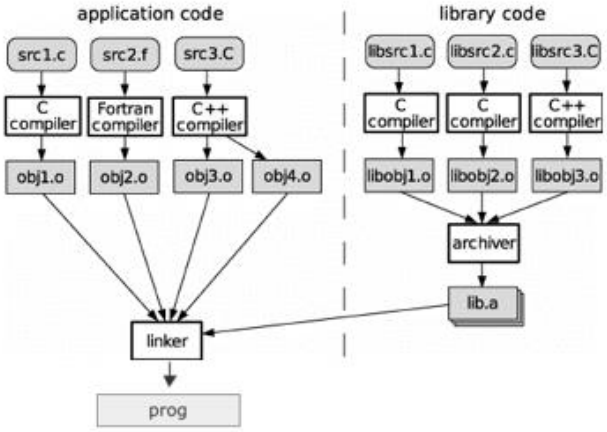
1.6.4.1 Beispiel Makefile

```
1. CC = cc                          1. Festlegen des Compiler &
   LD = ld                             Festlegen des Linker

2. foo.o: foo.c                     2. foo Objekt File wird aus C
   $(CC) -c foo.c                      Quellcode mit Compiler „CC"
                                       übersetzt

3. Bar.o: bar.c                     3. bar Objekt File wird aus C
   $(CC) -c bar.c                      Quellcode mit Compiler „CC"
                                       übersetzt

4. prog: foo.o bar.o                4. Prog ist abhängig von foo & bar
   $(LD) -o prog foo.o bar.o           … und wird aus beiden erzeugt
                                       (gelinkt)
```

Das Beispiel in graphischer Ansicht:



Makefile externe Libraries:



1.6.4.2 Vorteile von Build gegenüber der Paketverwaltung

Ein Vorteil von Build gegenüber der Paketverwaltung ist, dass die Sicherheit gegeben ist, dass die Executable auf Basis des vorhandenen Quellcodes erstellt wurde. Weiterhin ist der Sourcecode anpassbar. Generell ist Build auf eine eigene Hardwarekonfiguration angepasst.

1.6.4.3 Nachteile von Build gegenüber der Paketverwaltung

Build ist zeitintensiver und fehleranfälliger als die Paketverwaltung. Weiterhin können Abhängigkeiten zwar automatisiert geprüft, jedoch nur manuell aufgelöst werden. Automatische Updates sind ebenfalls nicht möglich, da es sich nicht um die Paketverwaltung handelt.