1 Motivation und Einführung

Streitfragen um die „richtige“ Architektur von Betriebssystemen und Rechnern bestimmen vielfach die (ver-) öffentlichte Meinungsbildung und die Werbeaussagen großer Hersteller.
Jenseits der plakativen und vermeindlichen Vergleiche und Entwicklungssprünge haben sich jedoch die der Rechnerarchitektur und dem Aufbau von Betriebssystem zugrundeliegenden Konzepte in den letzten Jahren nicht umwälzend geändert. Vielmehr existiert eine stabile Basis gut verstandener und in der Praxis erprobter Grundlagen, die auch in modernen Rechnerarchitekturen und Betriebssystemen Anwendung finden. Hierzu zählt der Bestand an Wissen über Aufbau und interne Organisation von Prozessoren und Betriebssystemen, hierbei insbesondere Aspekte der Ablaufsteuerung, Speicherverwaltung und des Zugriffs auf Hintergrundspeicher. Gleichzeitig eroberten sich verteilte Systeme in der jüngeren Vergangenheit einen solch zentralen Stellenwert in der Anwendung, daß Kenntnisse über deren Basismechanismen inzwischen auch als Grundlagenwissen anzusehen sind.

1.1 Rechner- und Betriebssystementwicklung

Betriebssysteme und die Architektur der sie ausführenden Hardware sind einem stetigen Wandel unterworfen und haben sich im Verlauf der Entwicklung elektronischer Datenverarbeitung stark verändert. Im Kern bilden beide zusammen Triebfeder und Schrittmacher der verschiedenen Generationen und wirkten mit ihren Konzepten vielfach stilbildend für die jeweils aktuelle Form der Datenverarbeitung.

Technisch gesehen ist ein Betriebssystem lediglich ein Programm, welches durch die zugrundeliegende Hardware zur Ausführung gebracht wird. Jedoch nimmt dieser Programmtyp eine solche zentrale Stellung ein, daß vielfach Hardware und Betriebssystem (immernoch) als Einheit betrachtet werden, obwohl sich inzwischen beide voneinander entkoppelt -- jedoch durch starke Wechselwirkungen geprägt -- entwickeln.

Aufgrund dieser zentralen Stellung definiert DIN 44300 den Begriff *Betriebssystem* daher als: „Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen“.

Nachfolgend wird ein kurzer Überblick von Hardware-, Rechner- und Betriebssystemgenerationen gegeben, welche die historische Entwicklung kurz umreißt und einen Einblick in die Fortschritte der zugrundeliegenden Konzepte bietet. Gleichzeitig finden typische Interaktionsformen mit der aus Hardware und Betriebssystem gebildeten Recheneinheit Berücksichtigung.

Zeitraum	Schaltungsrealisierung	Verbreitung	Typische Betriebssysteme	Typische Programmiersprachen	Geschwindigkeit	Beispiele
1945-1955	Relais, später Röhren	wenige Exemplare (Univac: 46 Stück)	*Keine*	Steckkarten, später Maschinencode	10^3Ops./Sec.	Univac
1955-1965	Transistoren und Dioden	Hunderte	Hardwarespezifisch (z.B. Multics,)	Assembler, später FORTRAN, Algol60 und COBOL	10^4Ops./Sec.	IBM 1401, 7094, DEC PDP
1965-1980	Integrierte Schaltungen	Tausende	Hardwarespezifisch (z.B. OS/360, ...)	höhere Programmierspachen, erste Programmiermethoden	10^6Ops./Sec.	IBM 360
ab 1980	Hoch integrierte Schaltungen (VLSI)	Millionen	Prozessorspezifisch (z.B. MS-DOS, Windows, Minix, Linux)	Höhere Programmiersprachen (C++, Java, ...)	min. 10^7Ops./Sec.	Intel Prozessoren, AMD, Motorola ...

Handelte es sich bei der Univac noch um ein eher experimentelles und nur in wenigen Stückzahlen gefertiges Gerät, die sich durch hohe Anschaffungs- und Betriebskosten sowie immensen Platzbedarf und geringe Zuverlässigkeit auszeicheten, so beanspruchen die die ab 1955 entwickelten Maschinen ihre Rolle als erste kommerziell erfolgreiche Systeme. Für sie hat sich die Gattungsbezeichnung *Mainframe* eingebürgert. Die Arbeit mit den Rechnern dieser Generation läuft ausschließlich offline im *Stapelverarbeitungsbetrieb* (auch: *batch-Betrieb*), d.h. die zu berechnenden Aufgaben (*jobs*) werden nicht interaktiv abgearbeitet, sondern nach ihrer Zeitdauer manuell durch einen menschlichen Operator zusammengestellt und der Maschine durch Lochkarten vorgelegt. Zumeist sind die Maschinen für genau ein konkretes Anwendungsgebiet (z.B. numerische Berechnungen) konzipiert.
Später entwickelt sich daraus die *Stapelverarbeitung*, welche die Eingaben in Form von Magnetbändern -- jedoch immernoch seriell und nicht interaktiv -- akzeptiert.
Ab 1965 ist der Versuch einer Konsolidierung der verschiedenen Hardwarestränge seitens der Hersteller zu spüren. So fertigt IBM mit dem System 360 ein Serie kompatibler Rechner (d.h. die auf einer Systemvariante ablauffähige Software kann auch auf einem anderen Mitglied der Rechnerfamilie zur Ausführung gebracht werden). In dieser Entwicklungsgeneration steht bereits sowiel Rechenkapazität zur Verfügung, daß ein aktuell bearbeiteter Auftrag die Maschinen nicht mehr vollständig auszulasten vermag. Daher wird mit dem Konzept des *Multiprogramming* die Möglichkeit zur Bearbeitung einer anderen anstehenden Aufgabe geschaffen, während sich die aktuell in Bearbeitung befindliche im Wartezustand (etwa auf Ende einer Ein-/Ausgabeoperation) befindet. Diese Ausführungsform bedingt jedoch eine zusätzliche Komplexität im Aufbau der Verwaltungsfunktionen, da diese nun die gleichzeitige Präsenz verschiedener Programme im verwalteten Speicher zu berücksichtigen haben. Darüberhinaus ist der Programmablauf so zu organisieren, daß sich die verschiedenen Programme geschützt voneinander abgearbeitet werden und sich nicht

gegenseitig beinträchtigen.
In Erweiterung des Multiprogrammbetriebes wurde mit *Timesharing* eine Variante dieser Betriebsform eingeführt, die es erstmals mehreren Benutzern gestattete gleichzeitig online am System zu Arbeiten. Durch schnelles Umschalten zwischen den Aufgaben der Einzelnutzer entsteht so die Illusion der alleinigen Arbeit am System.
Mit wachsenden Hauptspeichergrößen führt die dritte Entwicklungsgeneration (die sog. *Minicomputer*) auch Techniken zur Lösung der Abhängigkeit von langsamen Speichertypen ein. So wurde mit *Simultaneous Peripheral Operation On Line* -- kurz *Spooling* -- ein Verfahren eingeführt alle übergebenen Aufgaben zunächst von Lochkarten einzulesen und auf Platten zwischenzuspeichern.
Aus einer Weiterentwicklung des Timesharing-Systems *MULTIC* (MULTIplexed Information and Computing System) entwickelt sich ab den frühen 1970er Jahren das System *UNIX*, welches zunächst auf der PDP-7-Hardware entwickelt, jedoch später auch für andere Plattformen portiert, wurde.

Die Hardware- und Betriebssystementwicklung ab 1980 ist begleitet durch einen massiven Siegeszug an einem immer größer werdenen Massenmarkt. Die Fortschritte der Fertigungstechnik erlauben eine immer weiter voranschreitende Packungsdichte der integrierten Schaltungen und verbilligen gleichzeitig deren Herstellung. Die so gefertigten Hardwaren (als *Personal Computer* (*PC*) oder auch *Mikrocomputer* bezeichnet) entsprechen in ihren Strukturprinzipien weitestgehend den Minicomputern der Vorgängergeneration, jedoch zu einem sehr viel niedrigeren Verkaufspreis.

Als eines der ersten Betriebssysteme für dies neuen Rechnergeneration stand *CP/M* (*Control Program for Microcomputers*) zur Verfügung, welches den 1974 durch Intel vorgestellten 8-Bit Mikroprozessor 8080 unterstützte. Seit den frühen 1980er Jahren finden Prozessoren dieses Typs Verwendung in der, zunächst ausschließlich durch IBM vorangetriebenen, PC-Line. Nachdem Verhandlungen zwischen den Autoren des CP/M-Systems und IBM über die Nutzung des Betriebssystems für den IBM PC scheiterten lieferte Microsoft mit *MS-DOS* (*Microsoft Disk Operating System*) das erste kommerziell vertriebene Betriebssystem für diesen Rechnertyp.

Gegen Ende der 1980er Jahre geriet die Benutzbarkeit der -- damals schon in hohen Stückzahlen verkauften PCs -- immer stärker ins Zentrum des Interesses. Als Vorreiter führte der Hersteller Apple auf seiner nicht IBM-kompatiblen Hardware eine graphische Oberfläche (*GUI* -- *Graphical User Interface*) zur Bedienung des Rechners ein. Deren, ursprünglich am Xerox PARC-Forschungszentrum entwickeltes Konzept, gewann in der Folge immer größere Bedeutng und wurde letztlich durch das MS-DOS-Programm *Windows* popularisiert.

Seit 1995 vertreibt Microsoft ein -- ebenfalls *Windows* genannte -- Familie eigenständiger PC-Betriebssysteme, die den kommerziellen Massenmarkt beherrscht.
Gleichzeitig findet vielfach -- insbesondere auf Netzwerkrechnern (Servern) -- die ursprünglich durch den finnischen Studenten Linus Torvalds entwickelte -- Unix-Variante *Linux* breiten Einsatz.

Hinzu tritt in den 1990er Jahren der Trend zur verteilten Verarbeitung auf Basis durch Netzwerke gekoppelter Rechner.

1.2 Peripherie

Parallel zur Entwicklung der Kernhardware und der darauf ablaufenden Betriebssysteme entwickelt sich die Landschaft der mit der Berechnugnseinheit kommunizierenden Peripheriegeräte. Hierbei ist der Begriff des Peripheriegerätes jedoch aus Sicht des Zentralprozessors interpretiert und daher entsprechend weiter gefaßt als seine landläufige Definition, welche ausschließlich an der Zentraleinheit (bestehend aus CPU, Speicher und Bussen) anschließbare Geräte berücksichtigt.

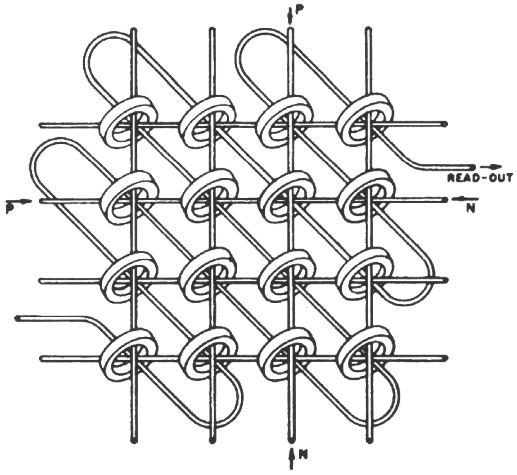
Die erste Rechnergeneration verfügte noch über keine dedizierten Peripheriegeräte. Eingaben wurden direkt, durch Schalttafeln und Erstellung von Hardwareverbindungen, an der Maschine vorgenommen; Ausgaben ebendort -- von Lampen oder anderen Signalgebern -- abgelesen.

Ab den 1955er Jahren finden sich zunehmen externe Geräte zur Abwicklung und dauerhaften Speicherung der verarbeiteten Daten. Urvater des Ein- und Ausgabegerätes waren zunächst Lochstreifen, welche zur Steuerung automatisierter Webstühle bereits im 19 Jahrhundert und später zur Darstellung von Morsezeichen Verwendung fanden, die später durch Lochkarten abgelöst wurden.
Die Lochkarte bot, neben der vergleichsweise einfachen manuellen Handhabung die Möglichkeit bestehende -- zuvor durch Tabelliermaschinen verarbeitete -- Datenbestände weiter einsetzen zu können.
Im Deutschen war damals der Begriff *Hollerithmaschine -- nach H. Hollerith , der bereits zur Abwicklung der US-Volkszählung von 1890 Lochkarten einsetzte -- für lochkartenbetriebene Rechner gängig.*
Zur Senkung der Fehlerrate (etwa durch Vertauschung zweier Lochkarten) und Erhöhung der Verarbeitungsgeschwindigkeit wurden die Lochkarten zunehmend durch Magnetbänder ersetzt.
Das Aufkommen von Ferritkernspeicher n markiert den Übergang zu Speicherorganisationsformen mit wahlfreiem Zugriff, da jeder einzelne Speicherring (Bit) separat angesprochen werden konnte.

Die Grundidee der Ferritkernspeicher (auch als *Ringkernspeicher* bekannt) hat sich bis heute erhalten, wenngleich ihre technische Realisierung im Lauf der Zeit natürlich mit den modernisierten Fertigungsmethoden Schritt hielt.
In einem Ringkernspeicher sind parallel zum Rahmen dünne Metalldrähte -- die Steuerdrähte -- gespannt, die mit Strom beschickt werden können. Zusätzlich verläuft ein Diagonaldraht (Lesedraht), der alle alle magnetisierbaren Ringkerne verbindet.

Jeder Ring kann separat durch gerichteten Stromfluß in den Steuerdrähten auf zwei verschiedene Weisen magnetisiert werden. Fliest der horizontale Steuerstrom transversal (in der Abbildung in *P* -Richtung) und der vertikale nach unten (ebenfalls *P*), so entsteht eine Magnetisierung die diagonal nach unten gerichtet ist. Im umgekehrten Falle (Stromrichtungen: aufwärts und linksgerichteter Strom (jeweils durch *N*) entsteht ein entgegengesetztes, d.h. diagonal nach links-oben gerichtetes Magnetfeld.

Abbildung 1: *Aufbau eines Ringkernspeichers*



(click on image to enlarge!)

Quelle: J. A. Rajchmann: A Myriabit Magnetic Core Matrix Memory. IRE Proceedings 1408, Okt. 1953, IEEE 1953.

Zum Auslesen des Speicherzustandes wird die Hystereseeigenschaft ausgenutzt und testweise ein beliebiger Beschreibungsvorgang durchgeführt. Ändert sich durch diesen die zuvor präsente Magnetisierung, so wird im Lesedraht ein Stromstoß induziert, andernfalls nicht.

Als Weiterentwicklung der Ringkernspeicher bürgerten sich *Trommelspeicher* ein, welche die Datenspeicherung auf einer magnetisierten rotierenden Trommel vornimmt. Das Auslesen geschieht durch am Gehäuse fest angebrachte Magnetköpfe.
Seit den 1960er Jahren (erster„Masseneinsatz“in IBM 305) kommen zunehmend magnetische Speicherplatten mit wahlfreiem als externe Speicher in den Masseneinsatz. Ihre technischen Basiskonzepte unterscheiden sich kaum von aktuellen Festplatten.

Mit dem Aufkommen des interaktiven Timesharing-Betriebes wurden auch neue Formen der Ein- und Ausgabeverarbeitung notwendig, da nicht mehr jeder Nutzer direkten Zugriff auf das physische Speichermedium haben konnte. Daher entstanden visuelle Endgeräte zur Präsentation der Daten, sowie Tastaturen zu ihrer Erfassung.

Der Siegeszug der graphischen Benutzeroberflächen in den 1980er Jahren führte die *Maus* als neues Eingabemedium als festen Bestandteil der Rechnerperipherie ein.

Aktuelle Trends schließen Sprachein- und Ausgabe, sowie tastaturbasierte Eingabeformen mit reduzierter Tastenzahl (etwa auf Mobiltelephonen und PDAs) ein.

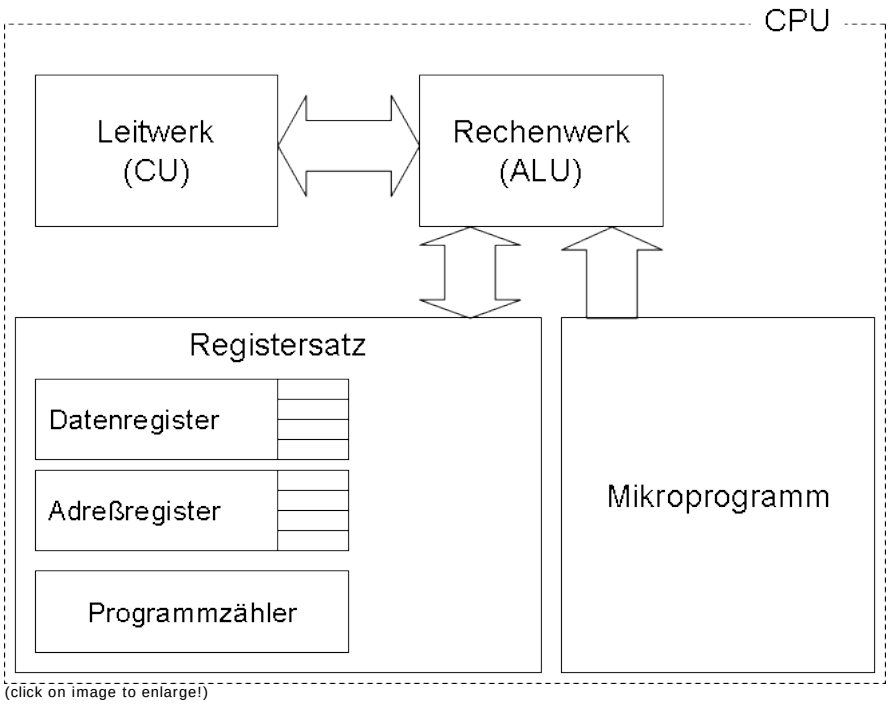
1.3 Zentralprozessoren

Wesentliches Element einer Rechnerarchitektur ist der *Zentralprozessor* (engl. *central processing unit*, kurz:CPU), welcher die Hauptteile der Verarbeitung übernimmt. Die CPU, deren stetiger Leistungszuwachs sich in der Vergangenheit nach dem *Moore'schen Gesetz (es sagt auf Basis empirischer Daten eine Verdopplung der Transistorenanzahl pro Chip (und damit der Leistungsfähigkeit) alle 18 Monate voraus) entwickelte, ist in jüngerer Zeit selbst zum Marketingbestandteil avanciert.*
Gleichzeitig rücken die internen Realisierungsdetails stark in das Bewußtsein des Anwenders, da sie die Leistungsfähigkeit und wirtschaftlich einer gegebenen Hardware stark beeinflußen.

Historisch gesehen determiniert die verwendete CPU typischerweise die auf einer Hardware ausführbaren Programme. So sind übersetze Anwendungen im Normalfall nur auf einem bestimmten Prozessortyp ausführbar, außer die verfügbare Hardware oder eine darauf angebotene Softwarekomponente bieten eine simulierende Unterstützung der ursprünglichen Zielhardware an. Unter Verwendung dieser simulierenden Unterstützung verhält sich ein Programm, welches für einen anderen Prozessortyp übersetzt wurde unter Eingabe derselben Daten so als würde es auf der ursprünglichen Hardware ausgeführt werden. Es handelt sich mithin um eine programmgesteuerte Imitation des Orginalprozessors.
Diese imitierende Simulation kann an Leistungsfähigkeit die ursprüngliche Zielhardware durchaus signifikant übertreffen. Aus diesem Grunde hat IBM für das System 360 den Begriff der *Emulation* (wohl auch um den negativen Beigeschmack der Begriffe *Simulation* und *Imitat* zu vermeiden) geprägt, deren Prozessor Programme für den damals kommerziell erfolgreichen Großrechnertyps IBM 7070 ausführen konnte. ([Cer03, S. 188])

Aktuelle Prozessoren folgen dem in Abbildung 5 dargestellten schematischen Aufbau, welche die zentralen Architekturkomponenten *Leitwerk* (engl. *control unit* (CU)), *Rechenwerk* (engl. *arithmetic and logic unit* (ALU), Registersatz und typischerweise den Mikroprogrammspeicher umfaßt.

Abbildung 2: *Schematischer Aufbau eines Zentralprozessors*



(click on image to enlarge!)

Diese, nach dem Initiator als von-Neumann -Architektur, bezeichnete Organisationsweise prägt den Aufbau von Rechenanlagen seit den späten 1940er Jahren und ist im Kern bis heute gültig; wenngleich sich in der Gegenwart verschiedene Engpässe dieses Ansatzes zeigen.

Die Kerngedanken dervon-Neumann-Architektursind:

- **Einteilung**der Verarbeitungseinheit in Leit- und Rechenwerk, Hauptspeicher, Ein- und Ausgabeeinheiten sowie Verbindungen (Busse) dazwischen.
  Dabei bilden Leit- und Rechenwerk die CPU.
- Die verarbeiteten Daten werden binär repräsentiert und taktgesteuert in Worten fester Bitlängen verarbeitet.
- Programme und Daten werden in einem gemeinsamen Hauptspeicher unsepariert und gekennzeichnet abgelegt. Der Hauptspeicher ist fortlaufend organisiert und wird durch eineindeutige Adressen angesprochen.
- Die Verarbeitung von Programmen und Daten erfolgt sequentiell typischerweise in der Reihenfolge der abgespeicherten Programmbefehle. Verzweigungen, Schleifen und Sprungbefehle können Abweichungen von der sequentiellen Abarbeitungsreihenfolge bedingen.

Die Festlegungen der von-Neumann-Architektur haben sich zunehmend als Gründe verschiedener Leistungsengpässe erweisen, die durch partielle Aufweichungen dieser Architekturprinzipien behoben werden können.

Einige bekannte Engpässe der von-Neumann-Architektur:

- Gleichbehandlung von Daten und darauf operierenden Programmen.
  Die Zugriffe auf die im Hauptspeicher identisch abgelegten Daten und Programme können sich gegenseitig behindern, da keine separaten Ein-/Ausgabekanäle für diese verarbeiten Inhalte vorgesehen sind.
  Zusätzlich eröffnet die nicht erfolgte Separierung von Daten und Programmen eine besondere Klasse von Bedrohungsszenarien. So können geeignet gestaltete Datenblöcke sog. Pufferüberläufe (engl. *buffer overflows*) provozieren, bei denen über Datenbereiche ausführbarer (Angriffs-)Code in den Programmbereich eingeschleußt wird. Beiden Problemen kann durch Einführung getrennter Speicher und Busse für Daten und Programme (insbesondere im Umfeld schneller Zwischenspeicher sog. *Caches*) begegnet werden (Havard-Architektur). Ein neuerer Ansatz zur Beibehaltung der Ablage im selben Speicherbereich wird durch Einführung einer expliziten Kennzeichung nicht ausführbarer Datenbereiche im Hauptspeicher realisiert.
- Sequentielle Verarbeitung.
  Die Leistungsfähigkeit einer Hardware läßt sich durch Schaffung von Möglichkeiten zur gleichzeitigen (parallelen) Verarbeitung einzelner Befehle steigern. Ansätze hierzu bilden:
  - Instruction Level Parallelism
  - Multithreading
  - Multiprozessorsysteme
- Festgelegte Befehlsabfolge.
  Die Verarbeitungsreichenfolge der Einzelbefehle eines Programmes kann von der Reihenfolge ihres Auftretens im Programm abweichen. In Verbindung mit Parallelitätsansätzen lassen sich so zeitaufwendige Befehlsfolgen„im Voraus“berechnen und auf diesem Wege die Verarbeitungszeit senken.
  Voraussetzung hierfür ist der flexible Umgang mit den belegbaren Registern, d.h. im Idealfalle die Existenz mehrerer Registersätze.

Die in Abbildung 5 zusammengestellten Kernkomponenten einer CPU erfüllen festgelegte Aufgaben. Ihre Arbeitsteilung bildete sich im Laufe der Entwicklungsgeschichte heraus und ist bis heute aktuellen Prozessorarchitekturen nahezu unverändert präsent.

Rechenwerk (ALU)

Das Rechenwerk führt die Rechenoperationen auf den speicherresidenten Operatoren durch.

Aktuelle CPU-Architekturen (wie Intel Pentium, AMD Athlon und Power PC) verfügen innerhalb einer CPU über mehrere Rechenwerke, insbesondere solche für Fest- und Gleitkommaoperationen.

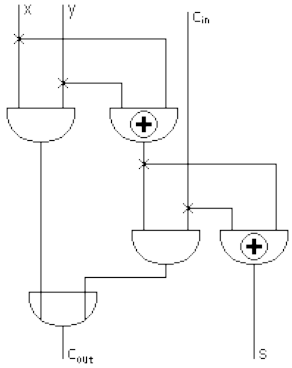
Typischerweise bietet ein Rechenwerk die folgenden grundlegenen Operationstypen an:

- Arithmetische Operationen: Addition, Subtraktion, Multiplikation und Division.
- Logische Operationen: AND, OR, NOT, NOR, XOR, ...
- Bitmanipulationen: Rotation, Verschiebung und Vergleiche.

Als Beispiel einer ALU-Komponente zur Addition n-stelliger Binärzahlen mit Übertrag sei der *Ripple-Carry-Addierer* betrachtet:

Grundkomponente eines solchen Addierers sind eine Reihe verschalteter Volladdierer (engl. *full adder* (FA)), wie er schematisch in Abbildung 1 abgebildet ist.

Abbildung 3: *Aufbau eines Volladdierers*

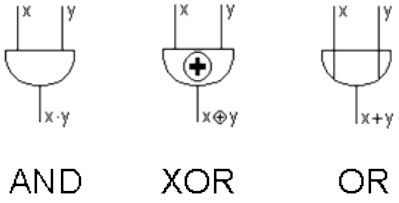


(click on image to enlarge!)

Abbildung aus: Wikipedia, Volladdierer

Anmerkung: Bedeutung der Schaltsymbole

Abbildung 4: *Schaltsymbole*

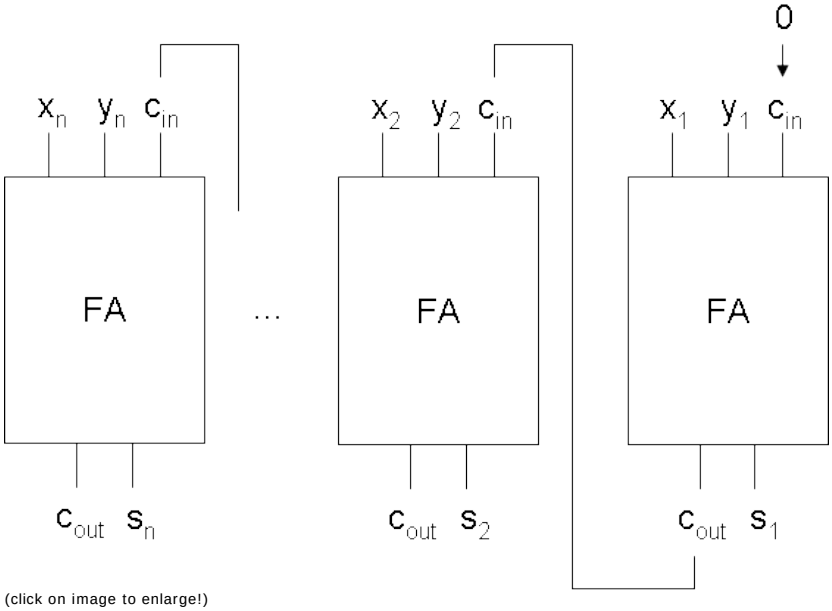


(click on image to enlarge!)

Im Unterschied zum Halbaddierer liefert der Volladdierer nicht nur einen Übertrag im Rahmen der Ergebnisberechnung, sondern akzeptiert diesen auch als Eingabe (einer möglicherweise vorhergehenden Additionsoperation).

Durch die Hintereinanderschaltung meherer Volladdierer und Verknüpfung des c_{in}-Einganges des *n* -ten Addierers mit dem c_{out}-Ausgang des vorhergehenden *n-1* -tenVolladdierers entsteht die in Abbildung 1 dargestellte Rechenwerkkomponente zur Berechnung n-stelliger Binärzahlen.
Zur Initialisierung der Übertragsbehandlung wird der Eingangsübertrag für den ersten Volladdierer mit 0 vorbelegt.

Abbildung 5: *Ripple-Carry-Adder*



(click on image to enlarge!)

Es ist keine zwingende Voraussetzung, daß alle Rechenoperationen, die von einer ALU bereitgestellt werden vollständig in Hardware realisiert sind. Durch die Verwendung von Mikroprogrammen können diese selbst durch

Programme implementiert sein.

Leitwerk (CU)

Dem Leitwerk obliegt die Steuerung des gesamten Verarbeitungsflusses. Hierzu zählen insbesondere die Versorgung der ALU mit Berechnungsaufgaben durch Bereitstellung der benötigen Befehle und Operanden.

Zur Regelung der Verarbeitung speichert das Leitwerk prozessorinterne Zustände (Status von Berechnungen (wie Überläufe, Ergebnis Null) und Fehlerindikatoren) in CPU-internen Speicherbereichen, den sog. Registern.

Registersatz

Der Registersatz wird durch eine Reihe von Speicherzellen, die direkt in der CPU untergebracht sind realisiert. Diese Speicherform stellt die schnellste (der Zugriff geschieht im Rahmen eines Prozessortaktzyklus) und gleichzeitig teuerste (daher stehen in der Regel in Summe nur wenige Byte zur Verfügung) dar.

Register besitzen typischerweise eine feste Bitlänge, die mit der verarbeitbaren Wortlänge der CPU in engem Zusammenhang steht. Üblich sind 8-, 16-, 32- und 64-Bit Register sowie 40- und 80-Bit Register für Spezialanwendungen (Fließkommaberechung).

Häufig werden die zur Verfügung stehenden Register bezüglich ihrer Funktionsweise, d.h. ihres Einsatzgebietes unterschieden, da oftmals nicht in jedem Register Operanden für beliebige Maschineninstruktionen zur Verfügung gestellt werden können.

Minimalanforderung an einen Registersatz ist die Bereitstellung eines Befehlszeigerregisters (engl. *Instruction Pointer Register*), welches die Adresse des nächsten abzuarbeitenden Befehls enthält.
Darüberhinaus sind Register zur Aufnahme der Speicheradressen der zentralen Datenstrukturen (z.B. Stack) üblich.

Beispielsweise partitioniert die Intel-Pentium-Architektur (IA-32) den verfügbaren Registersatz in vier Bereiche:

- Allzweck (engl. *General-purpose*) Register:
  Acht Register zur Aufnahme von Operanden und Speicheradressen.
  - EAX: Akkumulator zur Speicherung von Operanden und Ergebnisdaten.
  - EBX: Zeiger auf Daten im DS-Segment.
  - ECX: Zählregister für Schleifen und Zeichenkettenmanipulationen.
  - EDX: Zeiger auf Ein-/Ausgabedaten.
  - ESI: Zeiger in das durch DS bezeichnete Datensegment. Quellzeiger für Zeichenkettenoperationen.
  - EDI: Zeiger in das durch ES bezeichnete Datensegment zur Aufnahme von Zieldaten (typischerweise für Zeichenkettenoperationen verwendet.
  - ESP: Zeiger auf das oberste Element des Stacks, der durch das SS-Register bezeichnet wird.
  - EBP: Zeiger auf das unterste Element des Stacks, der durch das SS-Register bezeichnet wird.
- Segment Register:
  Identifizieren Speicherbereiche.
  - DS, ES, FS, GS: Datensegmente, die Ein-/Ausgabedaten enthalten.
  - SS: Stack Segment.
- Status- und Kontrollregister:
  Speichern Aussagen über verschiedene CPU interne Status.
  - CF (Carry Flag): Gesetzt falls bei einer arithmetischen Operation ein Über- oder Unterlauf auftritt.
  - PF (Parity Flag): Gesetzt falls das niedersignifikante Byte eines Resultats eine geradzahlige Anzahl 1-en enthält.
  - AF (Adjust Flag): Gesetzt falls bei einer Dezimalzahloperation ein Über- oder Unterlauf auftritt.
  - ZF (Zero Flag): Gesetzt falls das Ergebnis einer Berechnung Null ist.
  - SF (Sign Flag): Identisch zum höchstsignifikanten Bit eines Berechnungsergebnisses und damit identisch zum Vorzeichen.
  - OF (Overflow Flag): Gesetzt falls Berechnungsergebnis zu groß oder zu klein für Speicherung ist.
  - DF (Direction Flag): Steuert die Richtung der Stringverarbeitung.
  - IF (Interrupt Enable Flag): Steuert die Reaktion des Prozessors auf maskierbare Unterbrechungen.
  - TF (Trap Flag): Steuert den Eintritt in die Einzelschrittverarbeitung (zur Fehlersuche genutzt).
  - IOPL (IO Privilege Level): Bitfeld, welches die Ein-/Ausgabeprivilegien eines Prozesses festlegen.
  - NT (Nested Task Flag): Gesetzt falls aktuell verarbeitete Task mir vorhegend verarbeiteter in Verbindung steht.
  - RF (Resume Flag): Steuert Reaktion der CPU auf Debug-Ausnahmen.
  - VM (Virtual Mode): Steuert Eintritt in den virtuellen 8086-Modus.
  - AC (Alignment Flag): Aktiviert (in Verbindung mit dem `AM`-Bit des Kontrollregisters `CR0`) die Bereichsprüfung für Speicherreferenzen.
  - ID (Identification Flag): Weist auf Unterstützung des `CPUID`-Befehls hin.
- Befehlszeiger .
  Das `EIP`-Register enthält einen 32-Bit langen Verweis auf die nächste auszuführende Instruktion.

Die durch eine CPU angebotenen Befehle können (wie im Falle des Ripple-Carry-Addierers gezeigt) festverdrahtet sein, oder selbst durch kleine in der CPU abgelegte Programme -- sog. *Mikroprogramme* -- realisiert sein.

Eine ausführliche Fallstudie zur IA-32- und IA-64-Architektur findet sich in [Mär01, S. 174ff.]

Mikroprogramm

Das Konzept der Mikroprogrammierung, d.h. Einzelbefehle in eine Reihe vonMikrooperationenaufzuteilen, wurde kommerziell erstmals im IBM-System 360 verwirklicht. Seine Wurzeln reichen jedoch noch in die Zeit der Vorgängermaschine 1620 zurück, die bereits durch geschickte Speicheroperationen eine Änderung des Befehlssatzes ermöglichte ([Cer03, S. 107]).

Die Existenz von Mikrooperationen ist ein zentrales Kennzeichen einer CISC-Architektur.

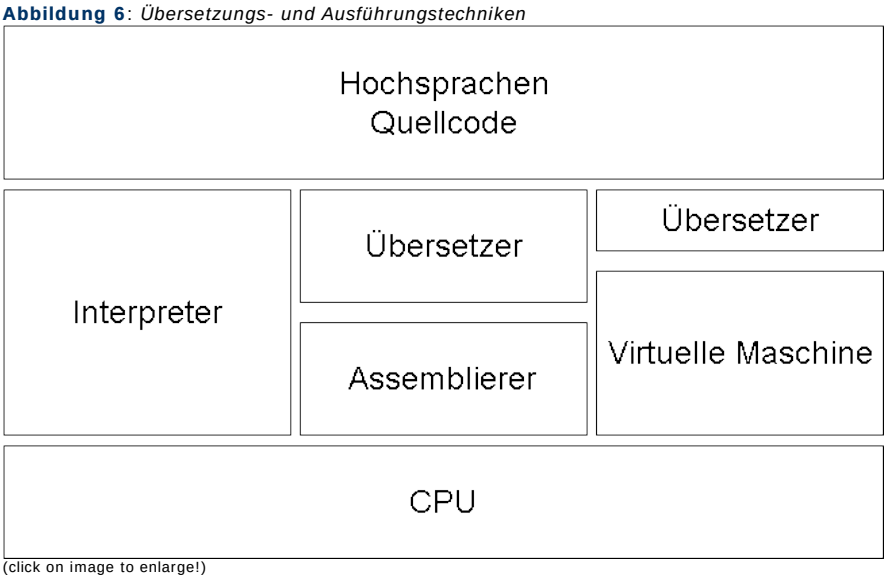
2 Rechnerarchitekturen

2.1 Befehlsarchitekturen

Die Architektur des Befehlssatzes, d.h. sein Aufbau und Umfang, bestimmt die Schnittstelle zur direkten Interaktion mit der CPU-Hardware welche typischerweise durch maschinell übersetzte (compilierte) Betriebssysteme und Anwendungsprogramme direkt bedient wird.

Die maschinelle Bedienung dieser Schnittstelle kann durch vielfälltige Softwarekomponenten erfolgen. Abbildung 6 stellt drei in der Praxis bedeutsame Ansätze gegenüber.

Abbildung 6: *Übersetzungs- und Ausführungstechniken*



(click on image to enlarge!)

Im linken Bildbereich ist das Zusammenspiel einer ausschließlich interpretierenden Ausführungsumgebung dargestellt. Eine solche akzeptiert ein Quellprogramm als Eingabe und führt dieses„direkt“, d.h. ohne expliziten Übersetzerlauf, aus. Die notwendige Transformation der Hochspracheninstruktionen in die von der CPU unterstützten Befehle wird dynamisch zur Laufzeit vorgenommen.

Der mittlere Bereich stellt die klassischen Entwicklungsschritte unter Verwendung eines CPU-spezifischen (d.h. plattformspezifischen) Übersetzers dar, der maschinenspezifischen Assemblercode erzeugt, der vermöge eines Assemblierers in direkt (nativ) ausführbaren Binärcode übersetzt wird.

Als Beispiel seien die im Verlauf der Übersetzung eines C-Programmes entstehenden Artefakte angeführt.

Quellcode:

```
#include <stdio.h>
                int main(int argc, char** argv) {
                    printf("hello world");
                    return 0;
                }
```

Erzeugter Assembler-Code für die Intel-Architektur

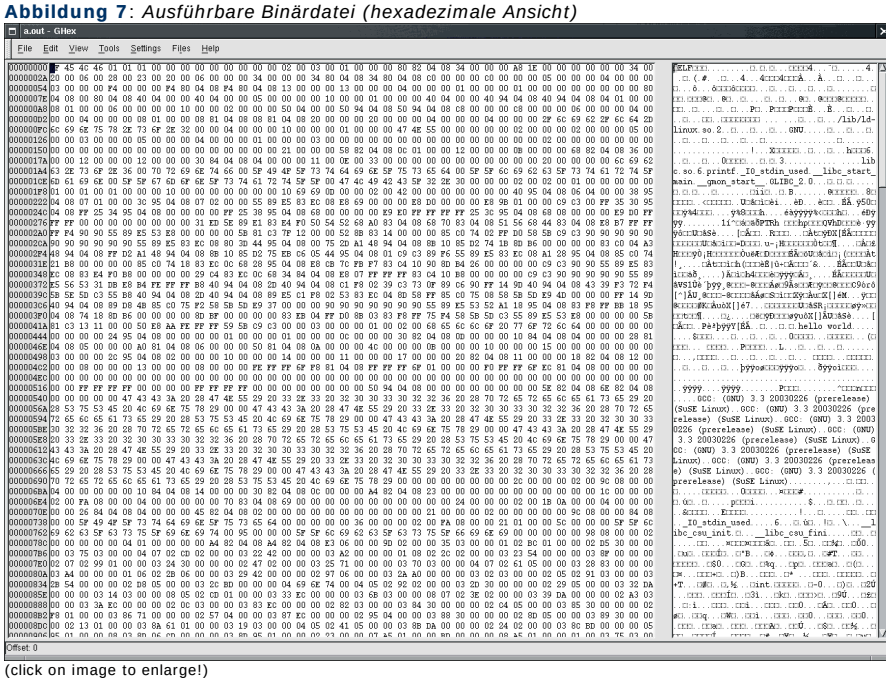
```
section .data
hello:     db 'hello world',10
helloLen:  equ $-hello
section .text
global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,hello
    mov edx,helloLen
    int 80h
    mov eax,1
    mov ebx,0
    int 80h
```

Der Quellcode eines Assemblerprogrammes kann vergleichsweise einfach in binären Maschinencode übersetzt werden, da jede dort auftretende mnemonische Anweisung eineindeutig auf eine maschinenspezifische Instruktion

abgebildet werden kann.

Ausführbare Binärdatei:

Abbildung 7: *Ausführbare Binärdatei (hexadezimale Ansicht)*



(click on image to enlarge!)

Darüberhinaus hat der in der Graphik der Abbildung 6 rechts dargestellte Ansatz der hybriden Übersetzung durch die Popularität von Programmiersprachen wie Java oder C# in jüngerer Zeit eine große Anhängerschaft erworben.

Bei dieser Vorgehensweise wird der Quellcode in ein Zwischenformat, den sog. Intermediärcode oder Bytecode, übersetzt, der dann durch einen als *Virtuelle Maschine* bezeichneten Interpreter zur Laufzeit in reale Maschineninstruktionen der Zielhardware umgesetzt wird.

Grundsätzliches Unterscheidungsmerkmal der Abarbeitung des entstehenden Maschinencodes ist die Realisierung des Instruktionssatzes. Hierbei wird zwischen CISC-, RISC- und VLIW-Varianten unterschieden, die jeweils in der Realisierung des Befehlsvorrates und der Umsetzung des Befehlsformates differieren..

CISC-Befehlssatzarchitekturen

Architekturen, die viele und hochdifferenzierte Instruktionen für vielfältige Problemstellungen anbieten werden als *Complex Instruction Set Computing* (kurz: CISC) bezeichnet.

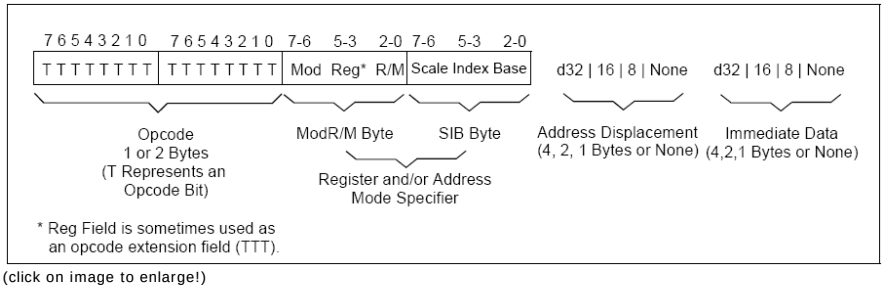
Die Ursprünge der CISC-Architekturen reichen bis in die Zeit der Großrechner in den 1960er Jahren zurück. Damals erwies sich die Mikroprogrammierung als probates Mittel der Komplexitätsreduktion bei der Planung und Umsetzung der verarbeitenden CPUs. Überdies bargen die im ROM des Zentralprozessors abgelegten Umsetzungen mächtiger Befehle einen Geschwindigkeitsvorteil, da auf die CPU-internen Speicherbereiche in deutlich performanterer Zugriff realisiert werden konnte, als dies für die damals üblichen Ferritkern-Hintergrundspeicher erzielbar gewesen wäre.

Ziel der Schaffung von Instruktionssätzen mit meheren Hundert verschiedenen (300-400 verschiedene Instruktionstypen sind keine Seltenheit) Befehlen war es die semantische Lücke zwischen den zur Applikationsprogrammierung eingesetzten Hochsprachen und den realen Maschineninstruktionen möglichst schmal zu halten. ([Mär01, S. 156f.])

Gleichzeitig verfügen CISC-Architekturen über eine Reihe verschiedender Befehlsformate, die sich in ihrem internen Aufbau unterscheiden. Im Kern bestehen Befehlsworte immer aus der Spezifikation der auszuführenden Maschineninstruktion, den benötigen Operanden und der Angabe des Speicherplatzes für das Berechnungsergebnis. Teilweise können die Eingangsoperanden direkt im Befehl untergebracht werden, statt sie aus einem Register oder einer Speicherzelle zu laden. Operanden dieses Typs werden als *Immediate Operanden* bezeichnet.
Im Falle von Verzweigungsbefehlen enthält der Operand die Adresse der nächsten auszuführenden Instruktion.

Als Konsequenz des komplexen Befehlsaufbaus und der mächtigen angebotenen Funktionalität kann die Abarbeitung eines CISC-Befehls mehere Taktzyklen in Anspruch nehmen.

Abbildung 8: *Instruktionsformat der Intel IA-32-Architektur*



(click on image to enlarge!)

Abbildung: Intel Architecture Software Developer's Manual Vol. 2, S. B-1

Abbildung 8 illustriert beispielhaft das Instruktionsformat der Intel IA-32-Architektur. Es stellt den binären Opcode eines Befehls wahlweise durch ein (z.B. `ADD`, `XOR`, `CMP`) oder zwei Byte (z.B. `MOVZX`, `CMPXCHG`, `XADD`) dar, die von einigen Befehlen (z.B. `MUL`, `DIV`, `NEG`) durch die in den Bits 3-5 des ModR/M-Bytes untergebrachten Erweiterungsfelder zusätzlich ergänzt werden.
Grundsätzlich legt das ModR/M-Byte den Adressierungsmodus eines Befehls fest, d.h. welche Register angesprochen werden (Reg-Feld) oder ob eine vorzeichenbehafte Verarbeitung vorgenommen wird.

Als Beispiel eines Mikroprogrammes sei die Realisierung der Operation `CMPXHG8Bauf` der Intel-Pentium-Hardware angeführt.
Die genannte Operation vergleicht den Inhalt einer 64-Bit Speicherzelle mit dem der Kombination der Register EDX und EAX. Sind die beiden Inhalte gleich, so wird der Registerinhalt aus ECX:EBX in die Speicherzelle transferiert, andernfalls der Speicherzelleninhalt in die Register geladen.
Das hierfür nötige Mikroprogramm lautet (Pseudocode, Pfeile deuten einzelne Speichertransferoperationen an):

```
IF(EDX:EAX = DEST)
                ZF <-- 1
                DEST <-- ECX:EBX
                ELSE
                ZF <-- 0
                EDX:EAX <-- DEST
```

Bedingt durch die breite Verfügbarkeit schneller Hintergrundspeicher und die zunehmende Komplexität in der Erstellung der benötigten Mikroprogramme, sowie die Zunahme der zur Speicherung benötigten Chipfläche bedingte den zunehmenden Wechsel zu Befehlssätzen, deren Mächtigkeit und Umfang gegenüber dem CISC-Ansatz deutlich reduziert ausfällt.

Exkurs: Komplexitätsprobleme CISC-Architekturen: der Pentium FDIV-BUG

- Empirische Untersuchung der Häufigkeit des Fehlerauftretens
- Statistical Analysis of Floating Point Flaw

Daher integrieren vormals ausschließlich als CISC ausgelegte CPU-Familien (z.B. Intel x86, DEC-VAX) zunehmen Elemente alternativer Befehlssatzarchitekturen.

RISC-Technik

Grundidee des *Reduced Instruction Set Computings* (RISC) ist es eine Komplexitätsreduktion der CPU zur Minimierung des Umfanges des angebotenen Befehlssatzes und gleichzeitig dessen Befehlsformaten herbeizuführen.

Typische RISC-Prozessoren (wie MIPS R10000, Sun-SPARC, DEC-Alpha, Power PC) bieten daher oftmals nur ca. 50 verschiedene Befehle an, deren interne Realisierungskomplexität so stark reduziert ist, daß sie vollständig in Hardware und damit genau einem Prozessortaktyklus abgearbeitet werden können. Eine Nebenbedingung hierfür bildet die Beschränkung der in einer Maschineninstruktion zugelassenen Speicherzugriffe. So gestatten RISC-Architekturen üblicherweise lediglich Operationen auf registerresidenten Operanden, um aufwendige Speicherzugriffe zu umgehen.

Alle Elemente des Befehlssatzes eines RISC-Prozessors besitzen, unabhängig von der Wortbreite des verarbeitenden Prozessors, dieselbe Wortlänge und einen einheitlichen Aufbau.
Ausgehend hiervon vereinfacht sich der interne Prozessoraufbau und die Befehlsverarbeitung dramatisch, da keine Mikroprogramm-Unterstützung und flexible Decodierung der Befehlsworte benötigt wird. Überdies bieten RISC-Architekturen lediglich eine stark reduzierte Befehlszahl an, die nur basale Instruktionen umfaßt.

Als Konsequenz dieses Reduktionsvorganges umfassen für RISC-Architekturen ausgelegte Maschinenprogramme typischerweise eine signifikant größere Anzahl Instruktionen als deren CISC-Pendant, da die RISC-Architektur die Explizierung komplexer Anweisungen auf der Maschinensprachenebene bedingt.
Aufgrund des vereinfachten Prozessoraufbaus können RISC-Rechner jedoch gleichzeitig eine geringer Zykluszeit (d.h. höhere Taktfrequenz) realieren, weshalb die in der Regel in einem Prozessorzyklus abgearbeiteten umfangreicheren Befehlsfolgen mit gegenüber der CISC-Architektur höherem Durchsatz umgesetzt werden können.

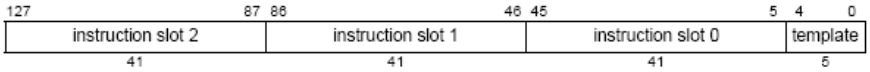
Die durch RISC-Technik erzielbare Steigerung der Verarbeitungsgeschwindigkeit wird durch die erreichbare Zykluszeit physikalisch begrenzt. Um dennoch eine weitere Skalierung zu ermöglichen werden seit den 1980er Jahren instruktionsparallele Ansätze verstärkt untersucht und auch in kommerziellen CPUs angeboten.

Expliziter Parallelismus und EPIC/VLIW

Architekturen mit explizitem Parallelismus auf Maschinenwortebene, sog. *Instruktionsebenenparallelität*, erlauben die simulatane Verarbeitung mehr als eines Maschinenbefehls zu einem Zeitpunkt.
Voraussetzung dieser Architekturform ist die Präsenz mehrer eigenständiger Rechenwerke, beispielsweise separater ALU-Einheiten für Integer- und Gleitkommaberechnung.

Grundlage dieser Befehlssatzarchitekturen ist die massive Erweiterung des verarbeiteten Maschinenwortes zum *Very Large Instruction Word* (VLIW), das je nach Rechnertyp bis zu 128-Bit lange Instruktionsworte (z.B. Intels IA-64-Architektur (Abbildung 9)) anbieten kann.
Begründet durch die bereits auf maschineninstruktionsebene explizierte Parallelität der Einzelinstruktionen findet sich verschiedentlich auch der durch den Hersteller Intel eingeführte Begriff des *Explicit Parallel Instruction Computings* (EPIC).

Abbildung 9: *Instruktionsformat der Intel IA-64-Architektur*



(click on image to enlarge!)

Der in Abbildung 9 dargestellte Aufbau der IA-64-Befehlsworte zeigt die Realisierung einer typischen VLIW-Instruktion. Jedes Befehlswort besteht aus höchstens drei eigenständigen Instruktionen, die jeweils durch 41-Bit lange (Sub-)Instruktionsworte codiert werden. Das als *template* bezeichnete Bitfeld regelt für jedes VLIW-Instruktionswort separat die Zuordnung der Subinstruktionsworte zu den verfügbaren Recheneinheiten bzw. spezifiziert deren Verarbeitungsnatur näher. Folgende Instruktionstypen werden unterschieden.

- ALU-Operationen
- Integer-Operationen
- Memory-Zugriffe
- Fließkommaberechnungen
- Branches (Sprunganweisungen)

Grundsätzlich sind nicht alle kominatorisch möglichen Zusammensetzungen zugelassen. So darf beispielsweise das *slot 2* Instruktionswort spezifikationsgemäß nicht mit einem Speicherzugriff (M) bestückt werden.
Die Realisierung des Parallelismus, durch geeignete Zusammensetzung der VLIW-Worte, obliegt generell dem Übersetzer, d.h. sie wird bereits vor der tatsächlichen Ausführungszeit fixiert. Der Programmübersetzungsprozeß nimmt daher -- aufgrund des zu leistenden gesteigerten Optimierungsaufwandes -- zusätzliche Zeit in Anspruch und erfordert speziell auf die Zielhardware angepaßte Übersetzer.

VLIW/EPIC-basierte Ansätze können, bei geeigneten -- d.h. gut parallelisierbaren -- Problemstellungen, große Geschwindigkeitszuwäche entfalten. Ist ein Instruktionsebenenparallelismus jedoch aufgrund von starken Datenabhängigkeiten (beispielsweise wenn jede Instruktion ein durch die jeweilige Vorgängerinstruktion berechnetes (Teil-)Ergebnis als Eingabe erwartet) nicht möglich, so werden müssen viele Subinstruktionsworte durch Leeroperationen (`NOP`) aufgefüllt werden und die durch sie ansprechbaren Funktionseinheiten bleiben ungenutzt.

Grundsätzlich lassen sich Befehlsfolgen ohne Abhängigkeiten , wie nebenläufig auszuführende Threads sehr effizient in VLIW-Instruktionsworte.

Web-Referenzen 1: Weiterführende Links

- Univac History
- The History of the UNIVAC Computer
- UNIVAC Memories
- Hello-World-Programm auf IBM 1401
- MULTICS
- Computer Model Katalog
- Rechner_ und Betriebssystemgenerationen und deren Auswirkungen auf betriebliche Anwendungssysteme
- Moore s Gesetz
- Two Approaches to 64-Bit Computing

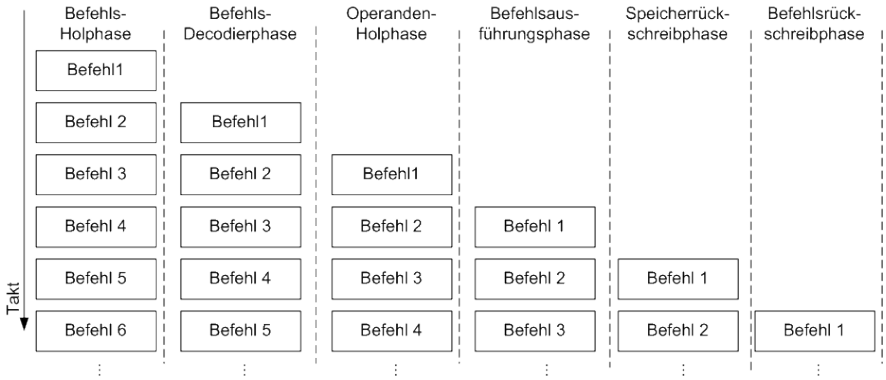
2.2 Mikroarchitekturen

Parallel zum Übergang von CISC- zu RISC-basierten Befehlsarchitekturen und der zunehmenden Verbreitung von auf dem VLIW-Ansatz aufbauenden Prozessoren werden neue Leistungspotentiale auch immer mehr durch neue Konzepte auf der Mikroebene erschlossen. Diese Ebene ist hierbei unterhalb der Abarbeitung der einzelnen Instruktionen der Befehlsarchitektur angesiedelt und widmet sich ausschließlich der internen Realisierung der Verarbeitung einer einzelnen Maschineninstruktion.

Phasen-Pipelining

Insbesondere bei RISC-Architekturen, die sich zum Ziel setzen jede Maschineninstruktion in Schnitt in genau einem Taktzyklus abarbeiten, definieren die physikalischen Signallaufzeiten eine Obergrenze des Prozessortaktes, da in einem Takt nicht beliebig viele logische Operationen ausgeführt werden können.
Um dennoch die Verarbeitungsgeschwindigkeit einerseits und gleichzeitig die Auslastung der verschiedenen Prozessorkomponenten andererseits zu erhöhen etablierten sich *Phasen-Pipeline-Architekturen*. Grundansatz dieses Miroarchitekturtyps ist es, jeden Einzelbefehl in eine Reihe fein granulierter Mikroinstruktionen aufzuspalten und diese durch jeweils durch separate CPU-Komponenten abzuarbeiten. Als Resultat können gleichzeitig Befehle verschiedener Ausführungsphasen verarbeitet werden. Die hierbei abzuarbeiten Einzelverarbeitungsschritte sind in jedem Falle von deutlich geringerer Schaltungskomplexität als vollständige Befehlsverarbeitungen.

Abbildung 10: *Schema einer sechsstufigen Phasen-Pipeline*



(click on image to enlarge!)

AbbildungAbbildung 10 illustriert den Aufbau einer sechsstufigen Phasen-Pipeline. Ihre Nutzung setzt voraus, daß die Ausführung jedes Maschinenbefehls in die sechs dargestellten Phasen eingeteilt wird:

- *Befehlsholphase* (IF)
  Der nächste auszuführende Maschinenbefehl wird durch das Leitwerk in die ALU geladen. Anschließend wird der Befehlszähler erhöht.
- *Befehlsdecodierphase* (DE)
  Die für den geladenen Maschinenbefehl auszuführenden Mikroprogramm-Operationen werden ermittelt.
- *Operandenholphase* (OF)
  Die zur Befehlsausführung benötigten Eingangsdaten (Operanden) werden geladen.
- *Befehlsausführungsphase* (EX)
  Durch Befehlsausführung wird das Ergebnis berechnet.
- *Speicherrückschreibphase* (MEM)
  Berechnungsergebnisse werden in Speicher zurückgeschrieben.
- *Befehlsrückschreibphase* (WB)
  Das Berechnungsergebnise werden in Register zurückgeschrieben.

Wird der Füllungsgrad der Phasen-Pipleline betrachtet, so fällt auf, daß jede Pipeline erst nach einer gewissen Anzahl Takten Ergebnisse liefert. Wird für jede Phase eine Ausführungszeit von genau einem Taktzyklus unterstellt, so ist die *Einschwingphase* identisch zur Pipelinelänge. Daher liefert die Pipleline der Abbildung 10 ihr erstes Ergebnis erst nach sechs Takten.
Alle darauffolgenden Takte wird jedoch im Idealfalle ein weiteres Ergebnis produziert.

Der maximale Auslastungsfall, der in jedem Taktzyklus ein Ergebnis berechnet, kann jedoch nur erreicht werden, wenn ein gleichmäßiger Füllungsgrad der Pipleline vorliegt, d.h. alle Pipelinestufen befüllt sind. Dies setzt jedoch voraus, daß keine Datenabhängigkeiten zwischen den verarbeiteten Einzelbefehlen bestehen, da zu Auslastungslücken führen können.

Solche Auslastungslücken (sog. *pipeline bubbles*) treten dann auf, wenn bestimmte Abhängigkeitssituationen (sog. *pipeline hazards*) gegeben sind, die eine effiziente Nutzung der Pipeline verhindern.

Typischerweise werden drei Typen von Pipeline Hazards unterschieden:

- **Ressourcen Abhängigkeit**treten dann auf, wenn nicht alle Phasen beliebig überlappend ausgeführt werden können.
- **Datenabhängigkeit**treten dann auf, wenn eine Instruktion das Ergebnis der direkt vorhergehenden als Eingangsoperand benötigt.
- **Kontrollflußabhängigkeit**treten dann auf, wenn die Verarbeitung einer Instruktion durch Modifikation des Befehlszeigers einen anderen als den bereits in der Pipeline befindlichen als nächste auszuführende Instruktion wählt.

Beispiel 1: Ressourcen Abhängigkeit

Instruktion	Taktzyklus								
	1	2	3	4	5	6	7	8	9
I_1	IF	DE	OF	EX	MEM	WB			
I_2		IF	DE	OF	EX	MEM	WB		
I_3			IF	DE	OF	EX	MEM	WB	
I_4				IF	DE	OF	EX	MEM	WB
I_5					IF	DE	OF	EX	MEM

Beispiel 1 zeigt als Beispiel einer Ressourcenabhängigkeit den Zustand der sechsstufigen Beispielpipeline, der im Falle nur genau eines Busses (oder nur genau einer Kontrolleinheit) zum Speicherzugriff eintritt. Hierbei kann die *Befehlsholphase* nicht überlappend mit der *Speicherrückschreibphase* ausgeführt werden.

Zur Lösung dieses Problems werden Warteanweisungen (NOP) in die Pipeline eingesteuert, während der die Verarbeitung in der betreffenden Stufe ruht.

Das Ergebnis ist in Abbildung2 dargestellt. Auffallend ist hierbei, daß selbst nach Einfügen einer NOP-Operation der

Befehlsholvorgang für die Instruktion I_5nicht durchgeführt werden kann, da sich dieser mit dem Speicherzugriff des 2. in der Berarbeitung befindlichen Befehls überschneiden würde. Daher wird erneut ein Wartezyklus eingefügt und die Verarbeitung der 5. Instruktion verzögert.
Dieser Vorgang wiederholt sich bis im 9. Verarbeitungstakt keine Überlappung zwischen Befehlsholphase und Speicherrückschreibphase mehr auftritt.

Beispiel 2: Auflösung der Ressourcen-Abhängigkeit

Instruktion	Taktzyklus								
	1	2	3	4	5	6	7	8	9
I_1	IF	DE	OF	EX	MEM	WB			
I_2		IF	DE	OF	EX	MEM	WB		
I_3			IF	DE	OF	EX	MEM	WB	
I_4				IF	DE	OF	EX	MEM	WB
(I_5-I)					NOP	IF	DE	OF	EX
(I_5-II)					NOP	NOP	IF	DE	OF
(I_5-III)					NOP	NOP	NOP	IF	DE
I_5					NOP	NOP	NOP	NOP	IF

Beispiel 3 zeigt den Fall einer Datenabhängigkeit, die durch den Verarbietungsversuch des Ergebnisses der Addition durch die nachfolgende Subtraktion entsteht.

Beispiel 3: Datenabhängigkeit

Instruktion	Taktzyklus						
	1	2	3	4	5	6	7
ADD R1,R2,R3	IF	DE	OF	EX	MEM	WB	
SUB R4,R5,R1		IF	DE	OF			

Durch Einstreuung von NOP-Befehlen kann die Verarbeitung der Subtraktion solange verzögert werden, bis das Berechnungsergebnis der vorangehenden Addition zur Verfügung steht. So kann die Verarbeitung erst nach drei Wartezyklen mit der Operandenholphase fortgesetzt werden.

Beispiel 4: Lösung der Datenabhängigkeit

Instruktion	Taktzyklus						
	1	2	3	4	5	6	7
ADD R1,R2,R3	IF	DE	OF	EX	MEM	WB	
(SUB R4,R5,R1)		IF	DE	NOP			
(SUB R4,R5,R1)		IF	DE	NOP	NOP		
(SUB R4,R5,R1)		IF	DE	NOP	NOP	NOP	
SUB R4,R5,R1		IF	DE	NOP	NOP	NOP	OF

Das Schema des Beispiels5 zeigt eine Datenflußabhängigkeit, die durch den bedingten Sprung JNE(Verzweigung bei Ungleichheit) entsteht. Nach Ausführung des Sprungbefehls muß bei l1fortgesetzt werden. Dies geschieht durch Umsetzen des Befehlszeigers unter Überspringen der auf die JNE-Instruktion folgenden Anweisungen.
Diese sind jedoch bereits in die Phasen-Pipeline geladen und liefern daher die nicht mehr benötigten (rot dargestellten) Ergebnisse.

Beispiel 5: Kontrollflußabhängigkeit

Instruktion	Taktzyklus								
	1	2	3	4	5	6	7	8	9
CMP R1,0	IF	DE	OF	EX	MEM	WB			
JNE I1		IF	DE	OF	EX	MEM	WB		
…			IF	DE	OF	EX	MEM	WB	
…				IF	DE	OF	EX	MEM	WB
…					IF	DE	OF	EX	MEM
I1: …								IF	DE

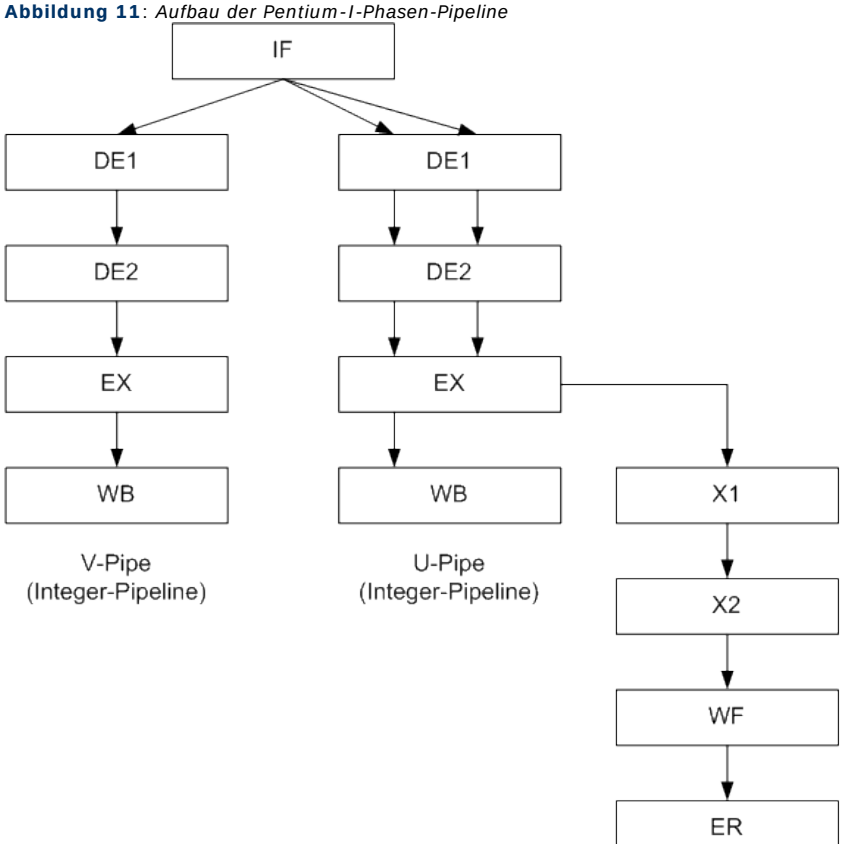
Die Einsteuerung von NOP-Anweisungen, bzw. das Verwerfen nicht mehr benötigter Ergenisse, garantiert zwar die Korrektheit der berechneten Resultate, setzt jedoch die Leistungsfähigkeit einer Phasen-Pipeline herab.
So erzwingt die restriktive Umsetzung des Speicherzugriffes aus Beispiel1 zum Einschub von vier Wartezyklen nach jeweils vier Berechnungsphasen.
Ähnliches gilt für die durch Beispiel3 betrachteten Datenabhängigkeiten. Auch sie können nur durch Einbringung entsprechender Verzögerungen, während der die ALU nicht genutzt wird, aufgelöst werden.
Im Falle der Kontrolfußabhängigkeit werden zwar keine Wartezyklen benötigt, jedoch werden im Verlauf der Verarbeitung durch die Pipeline Ergebnisse berechnet, die nicht benötigt werden, was effektiv einer Zeitverzögerung in der Bereitstellung der tatsächlich gewünschen Resultate gleichkommt.

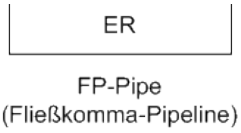
Phasenpipelines in der Praxis
Die Verwendung von Phasenpipelines zur Steigerung des Durchsatzes bei gleichzeitiger Erhöhung der Taktfrequenz hat sich inzwischen breit bei kommerziellen Mikroprozessoren durchgesetzt.
Intel verwendet seit der Pentium-Architektur Piplelining-Techniken, die zuerst eher zaghaft durch die maximal achtstufige Umsetzung in den Pentium-I-CPUs Einzug hielt, jedoch inzwischen mit einer extrem großen 30-stufigen Umsetzung in den Pentium-III-Chips bzw. 20 Stufen beim Pentium-4 ihre Entfaltung findet. Diese
AMD verwendet im 32-Bit Athlon-Prozessor eine 10-stufige Pipeline und bietet in der 64-Bit CPU-Variante eine um zwei zustätzliche Stufen erweiterte Umsetzung an.

Die ungewöhnlich lange Pipeline der Pentium-III- und -4-Realisierungen zeigt, daß auch große Phasenpipelines effizient eingesetzt werden können und erlauben es die genannten CPUs bei sehr hohen Taktraten zu betreiben.

Die Abbildung 1 zeigt die Realisierung der ersten Generation der Pentium-Mikroarchitektur, welche über eine zweifach gegabelte Pipeline verfügt. Die Architektur verfügt über drei parallel angeordnete Pipelines, wovon die als„U-“und„V-Pipeline“bezeichneten jeweils zur Verarbeitung von ganzzahligen Ausdrücken dienen. Zusätzlich zweigt die Ausführungsphase (EX) der U-Pipeline in die FP-Pipeline ab, welche vier zusätzliche Stufen zur Verarbeitung von Fließkommainstruktionen bereitstellt.
Neben den bereits bekannten Pipelinephasen zeigt die Graphik die zusätzlichen Ausführungsphasen X1und X2, sowie die gesonderte Rückschreib- (*Write Float* (WF)) und Fehlerbehandlungsphase (*Error Handling* (ER)) für Fließkommawerte.

Abbildung 11: *Aufbau der Pentium-I-Phasen-Pipeline*





(click on image to enlarge!)

Durch die extrem langen Pipelines gängiger Prozessoren werden sehr hohe Anforderung an die adäquate Bereitstellung der zu verarbeitenden Instruktionen gestellt, da sich Pipelinekonflikte mit gravierenden Performanceeinbußen auswirken können.
Aus diesem Grunde führen *superskalare* Architekturen eine Reihe von Maßnahmen zur Verbesserung der Pipelineauslastung ein.

Web-Referenzen 2: Weiterführende Links

- Pipeline Hazards
- Pipelinling 101
- Pentium-4-Hyper-Pipeline
- The perils of deep pipelining
- The Anatomy of Modern Processors: Pipelines
- Pipeline Streaming
- Pipelines of Superscalar (Pentium Family) and Dynamic Execution (P6-Family) Architectures

---

Superskalarität

Superskalare CPUs versuchen den Durchsatz durch CPU-interne Parallelität zu erhöhen. Hierzu werden Funktionseinheiten derselben Funktionseinheiten physisch mehrfach realisiert und angeboten. Zur Versorgung der Einzeleinheiten können entweder explizite VLIW-basierte Ansätze bereits zur Übersetzungszeit oder dynamische Pipeline-basierte zur Ausführungszeit Einsatz finden.

Nachfolgend wird ein Überblick der Strategien zur Ansteuerung von Funktionseinheiten in superskalaren CPUs gegeben. Gleichzeitig werden Strategien und Techniken diskutiert, die eingesetzt werden können um die Anzahl der Wartezyklen innerhalb einer Pipeline zu verringern und so ihre Auslastung zu steigern.

- **Hardwaremaßnahmen**
  Zur Behebung struktureller Ressourcen-Konflikte können zusätzliche Hardwarebausteine wie parallel Speicherzugriffskanäle und -controller eingesetzt werden um die zwangsweise Serialisierung der Instruktionen an diesen„Engstellen“(engl. *bottle neck*) zu verhindern.
- **Kontrollfluß-Modifikationen**
  Zur Erhöhung der Auslastung einer gegebenen CPU kann die Verarbeitungsreihenfolge der Instruktionen eines Programms von der Reihenfolge ihres Auftretens im Programm abweichen. Hierbei werden zwei Verarbeitungstypen unterschieden:
  - *In-order-Execution*
    Hierbei wird die Reihenfolge der im Programm codierten Befehle zunächst nicht verändert, sondern durch gegeabelte Pipelines auf verschiedene Ausführungseinheiten verteilt und so parallelisiert. Bekanntestes Beispiel für diesen Ansatz ist die Pipeline des Pentium .
  - *Out-of-Order-Execution*
    Instruktionen können gleichsam„auf Vorrat“ausgeführt werden um den Leerlauf, der durch NOP-Operationen entstünde, produktiv zu nutzen.
- **Registersatz-Modifikationen**
  Die während der Out-of-order-Ausfürhung entstehenden (noch) nicht benötigten Berechnungsergebnisse werden -- um störende Wechselwirkungen mit den regulär verarbeiteten Instruktionen auszuschließen -- in separaten Registern (sog. *Schattenregistern*) abgelegt.
- **Sprungvorhersage**
  Um die Produktion nicht mehr benötigter Resulate möglichst auszuschließen wird durch zusätzlichen Aufwand eine Sprungvorhersage betrieben, die es ermöglicht (mit einer gewissen Wahrscheinlichkeit) das Sprungziel vorherzubestimmen und die (vermutlich) benötigten Befehle„auf Vorrat“in die Pipeline zu laden.
- **Compilermaßnahmen**
  Ist der verwendete Übersetzer auf die später verwendete Zielarchitektur abgestimmt, so können gewisse vorausschauende Optimierungen bereits vor der Laufzeit vorgenommen werden.

Neben dem Einsatz superskalarer Techniken zur Steierung des Durchsatzes der CPU hat sich in jüngerer Zeit *simulatanes Multithreading* zur weiteren Leistungssteierung am Markt etabliert.

Multithreading-CPUs

Multithreading-CPUs führen die Unterstützung nebenläufig ausführbarer Programmteile noch einen Schritt weiter über die Möglichkeiten des expliziten Parallelismus auf Instruktionsebene hinaus und nehmen Anleihen bei den symmetrischen Multiprozessorsystemen, welche in einem System mehrere eigenständige CPUs vereinen.

Um jedoch der Kostensituation und verschiedenen Einsatzproblemen von Multiprozessorsystemen zu begegnen bieten Multithreading-CPUs nur simultane Instruktionsverarbeitung an ohne alle Ausführungsressourcen (wie Registersätze) mehrfach zu realisieren.

Grundsätzlich steuert in einer Multithreading-CPU genau ein Leitwerk mehrere getrennt ansprechbare Recheneinheiten an. Auf diese Weise können sich in einem Taktzyklus mehrere Instruktionen in der Befehlsausführungsphase befinden und Ergebnisse liefern.

Bei Multithreading-CPUs vergrößert sich tendenziell der Grad der Konkurrenz um die verfügbaren Ressourcen wie Register, Zwischenspeicher (Caches) und andere prozessorinterne Verwaltungseinheiten, so daß erhöhter Aufwand für

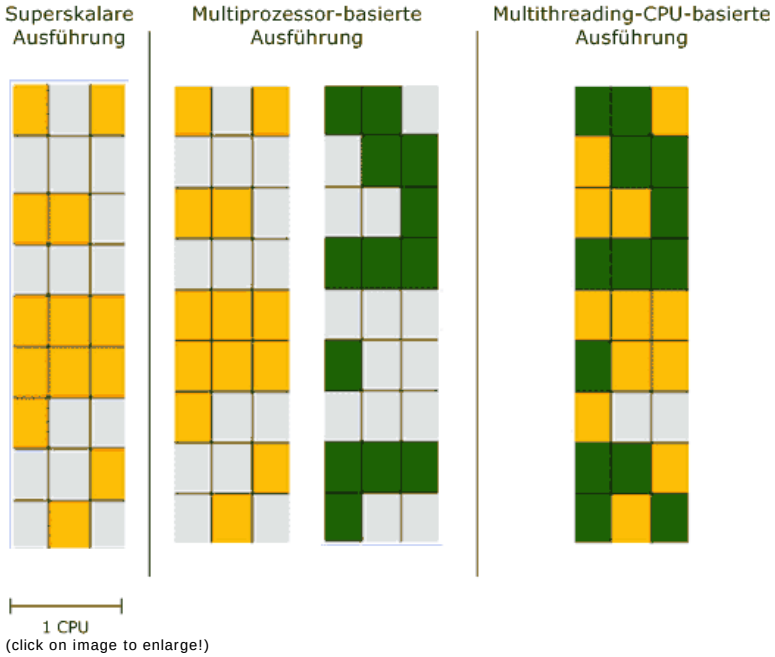
die Synchronisation und Behebung der auftretenden Zugriffskonflikte anzusetzen ist.

Grundsätzlich werden (nach [Mär01, S. 250] mit Verweis auf Culler) drei Varianten der Multithreading-Realisierung in einer CPU unterschieden:

- **Blocked Multithreading**:
  Eine begrenzte Anzahl zusätzlicher Register wird bereitgestellt, jedoch keine zusätzlichen Verarbeitungsressourcen. Als Konsquenz muß ein rechenbereiter Thread waren, bis ihm eine freie Rechenkomponente zugeteilt wird.
- **Interleaved Multithreading**:
  In jedem Taktzyklus wird erneut ein Thread zur Ausfürhung ausgewählt. Ein gegebener Befehl muß auf seine Ausführung daher so lange waren, bis er zur Abarbeitung selektiert wird.
- **Simulataneous Multithreading**(SMT):
  Kombiniert interleaved Multithreading und superskalare Ansätze um die gleichzeitige Ausführung von Instruktionen in einem Taktzyklus ermöglichen zu können.

Abbildung 12 zeigt die Ausführung nebenläufiger Befehlsfolgen auf einer superskalaren, einer Multiprozessor- und einer Multithreading-CPU.
Während die superskalare CPU die vorhandenen Ressourcen durch die auftretenden Wartezyklen nicht vollständig ausnutzen (48% ungenutzter Anteil im Beispiel) kann kann das Multiprozssorsystem zwar den Durchsatz des Gesamtsystems durch die parallele Ausführung separater Befehlsfolgen erhöhen bietet jedoch prozentual nur in etwa dieselbe Auslastung der vorhandenen Ressourcen.
Im Idealfalle kann eine Multithreading-basierte CPU, durch geeignete Zuteilung der Berechnungsaufgaben an die vorhandenen Berechnungsressourcen einen höheren Gesamtdurchsatz der CPU erreichen.

Abbildung 12: *Ausführung nebenläufiger Befehlsfolgen*



(click on image to enlarge!)

Gegenwärtig wird SMT durch Intel in seiner Pentium-4-Architektur popularisiert und für den Massenmarkt angeboten. Erste Erfahrungen mit diesem CPU-Typ stimmen durchaus positiv, wenngleich der Geschwindigkeitszugewinn sich lediglich im Bereich von 30% bewegt, was unter anderem auf die fortbestehende Konkurrenz um verschiedene Prozessor-Ressourcen zurückzuführen ist.

Web-Referenzen 3: Weiterführende Links
- Hyperthreading
- Hyper-Threading Technology
- Hyperthreading @ Intel
- SMT vs. SMP

2.3 Parallelverarbeitung

Rechnerklassifikation

Aspekte der Parallelverarbeitung werden in aktuellen Rechnerarchitekturen auf verschiedenste Weise und unterschiedlichen Ebenen genutzt, so daß eine trennscharfe Unterscheidung für gegenwärtige Prozessoren kaum mehr möglich ist, da auch diese bereits Parallelitätstechniken einsetzen.

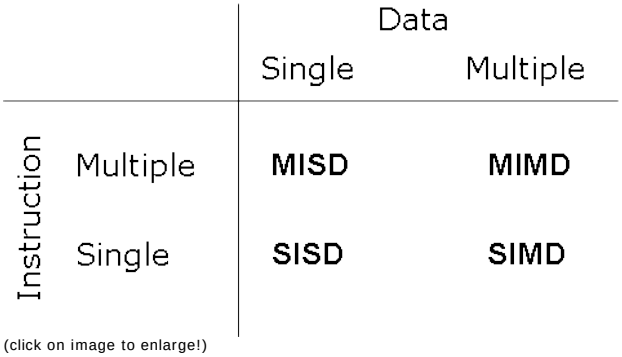
Klassischerweise werden nach Flynn die vier in Abbildung 13 dargestellten Verarbeitungsformen unterschieden, die jedoch heute nicht mehr in ihrer ursprünglichen Reinform auftreten:

- **SISD-Rechner**:
  In diese Klasse fallen die mit genau einem Leiterwerk und einem Rechenwerk ausgestatteten klassischen Universalrechner.
  SISD-Rechner führen zu einem gegebenen Zeitpunkt daher genau einen Befehl auf einem verarbeiteten Datum aus. Die gegebene Klassifikation läßt jedoch überlappende Ausführung, wie sie beispielsweise durch Phasen-Piplelines entstehen ebenso außer Acht, wie Parallelität auf Ebene der Rechenwerke wie sie durch Multithreading-CPUs

eingeführt wird.

- **SIMD-Rechner**:
  In diese Klasse fallen alle mit genau einem Leitwerk und mehreren Rechenwerken ausgestatteten Feldrechner oder Array-Prozessoren.
  Durch die Duplizität der Rechenwerke gestatten SIMD-Rechner die parallele Ausführung jeder Einzelinstruktion auf verschiedenen Eingangsdaten zu einem Zeitpunkt. Jedoch muß auf allen verarbeiteten Daten jeweils dieselbe Instruktion zur Ausführung gebracht werden.
  Typische Anwendungsfelder dieses Rechnertyps umfassen die Verarbeitung multimedialer Bild- und Tondaten.
  Auch Vektorprozessoren, die Operanden, welche aus einer festen Anzahl von Skalarwerten bestehen, verarbeiten können gemäß Giloi der Klasse der SIMD-Rechner zugeordnet werden.
- **MISD-Rechner**:
  Dieser Rechnertypus -- der mehrere Leitweitwerke, aber nur ein Rechenwerk vorsehen würde -- ist in der Praxis bisher nicht anzutreffen. Eine mögliche Realisierung dieser Rechnerklasse müßte ein „fließbandartig“ betriebenes Rechenwerk mit Daten, die durch unterschiedliche Leitwerke bereitgestellt werden, beschicken.
- **MIMD-Rechner**:
  In diese Klasse fallen alle Rechner, welche über mehrere Leit- und mehrere unabhängige Rechenwerke verfügen.
  Diesem Rechnertypus sind alle Formen von Parallelrechnern mit mehreren eigenständigen CPUs sowie verschaltete Rechner zuzuordnen, sofern sich diese als genau ein virtuelles System präsentieren.

Abbildung 13: *Rechnerklassifikation nach Flynn*



(click on image to enlarge!)

Das Flynn sche Schema stößt bereits bei der Rubrizierung aktuell verfügbarer Prozessoren, wie den mit mehreren separaten Funktionseinheiten ausgestatteten Intel- oder AMD-CPUs an seine Grenzen. Zwar kann jede einzelne Funktionseinheit der Klasse SIMD zugeordnet werden, jedoch ist eine eindeutige Klassifikation der Gesamt-CPU nicht mehr unstrittig möglich.

Speichergekoppelte-Umgebungen

In speichergekoppelten Umgebungen steht allen physischen Prozessoren ein gemeinsamer Hauptspeicher zur Verfügung. Anhand der physischen Realisierung dieses Speichers wird zwischen Architekturen des Typs *Unified Memory Access* (UMA) und *Non-Unified Memory Access (NUMA) unterschieden.*

Die enge Kopplung zwischen den Einzelprozessoren der UMA-Realisierungen setzt voraus, daß alle Prozessoren durch leistungsfähige Busse Zugang zum gemeinsam genutzten Speicher erhalten. Daher ist der physische Abstand zwischen jedem Einzelprozessor und dem Hauptspeicher in der Regel identisch groß.
Aufgrund der sich daraus inhärent ergebenden Sternarchitektur wird dieser Parallrechnertyp auch als *Symmetrisches Multiprocessing* bezeichnet.

Dieser Rechnertyp ist in der Praxis -- mit Ausbaustufen von bis zu acht verschalteten Prozessoren -- häufig anzutreffen. Die Integration größerer CPU-Anzahlen sind in Ihrer Realisierung als problematisch anzusehen, da die Konkurrenz um den Hauptspeicher bei steigender CPU-Anzahl zu überproportionalen Leistungseinbußen führt. Zusätzlich kann der Zugewinn an Busbandbreite nicht mit der Leistungssteigerung der CPUs Schritt halten, weshalb sich jeder Speicherzugriff als Performance aufzehrende Engstelle erweist.

Durch lose Kopplung in NUMA-Umgebungen, welche inhärent die unterschiedliche Realisierung der Speicherzugriffe für alle angebundenen CPUs zulassen können diese Engpässe vermieden werden. Hierzu kann an jedem physischen Konten eigener Hauptspeicher angesiedelt werden. Die einzelnen „Hauptspeicherinseln“ werden durch zusätzliche Hardware oder das das verwaltetende Betriebssystem zu einem virtuellen Gesamtspeicher verschaltet.

Nachrichtengekoppelte-Umgebungen

Wird die Forderung nach gemeinsamem Speicher zwischen den Einzelknoten aufgegeben, so lassen sich flexiblere Umsetzungen finden, die auch deutlich höhere CPU-Anzahlen umfassen können.
Jedoch entfällt dann der gemeinsame Speicher als Medium des Datenaustausches zwischen den Einzelprozessoren und muß durch eine explizite Kommunikationinfrastruktur ersetzt werden.

Hierbei kommen typischerweise Ansätze der Botschaftenvermittlung (*message passing*) zum Einsatz, welche Datenpakete zwischen den beteiligten Prozessorknoten versenden.

Eine bekannte Umsetzung in diesem Umfeld stellt das *Message Passing Interface* dar, welches eine Programmiersprachen-Bibliothek zur Verfügung stellt, welche die schnelle Botschaftenvermittlung zwischen verteilt ausgeführten Fortran- oder C-basierten Applikationen ermöglicht.

Leistungsabschätzung

Die zu erwartende Leistungssteigerung (*speedup*) eines Multiprozessorsystems kann durch verschiedene Gesetze abgeschätzt werden. Bekanntester Ansatz hierzu ist das Amdahlsche Gesetz.

In seiner in Abbildung 14 abgebildeten Form setzt die parallel ($p_{parallel}$) und seriell ($p_{seriell}$) auszuführenden Programmteile, sowie die Anzahl der verfügbaren CPUs (n_{CPU}) für eine feste Problemgröße in Beziehung und ermittelt daraus statisch die zu erwartetende Leistungssteigerung (S).
Für den Zusammehang zwischem seriell und parallel ausführbarem Code gilt dabei: $p_{seriell} + p_{parallel} = 1$.

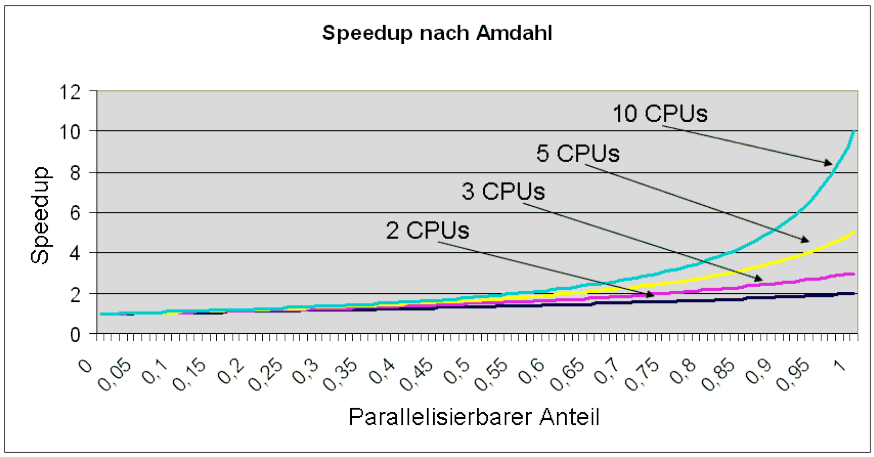
Abbildung 14: *Amdahlsches Gesetz*

$$S := \frac{1}{p_{seriell} + \frac{p_{parallel}}{n_{CPU}}}$$

(click on image to enlarge!)

Abbildung 15 zeigt den abgeschätzen Geschwindigkeitsgewinn relativ zu den betrachteten Frieheitsgraden.

Abbildung 15: *Abschätzung des Speedups nach Amdahl*



(click on image to enlarge!)

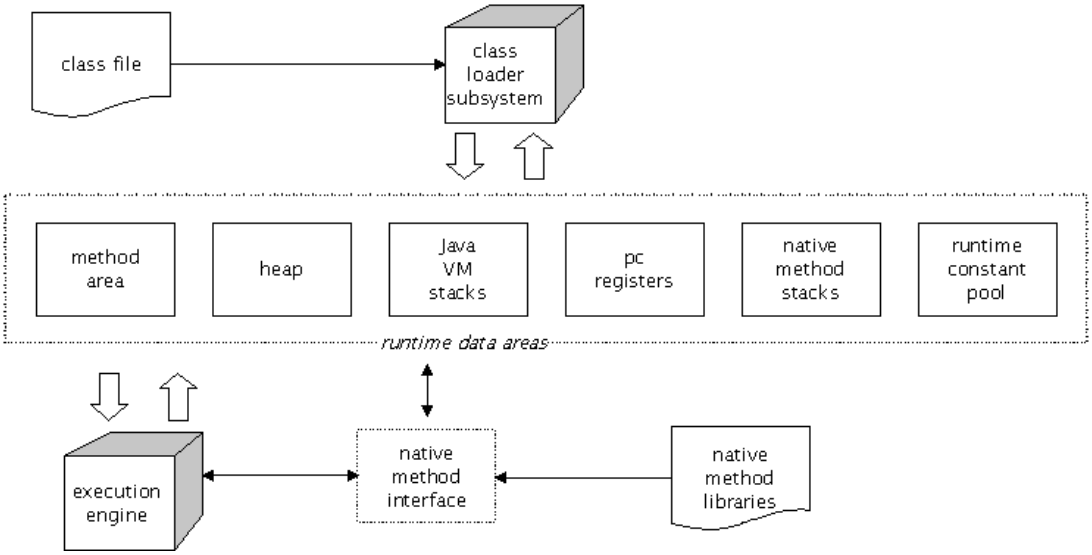
2.4 Fallstudie: Die virtuelle Rechnerarchitektur der Java-Maschine

Kern der oft apostrophierten Plattformunabhängigkeit der Programmiersprache Java ist die Generierung eines generischen Zwischenformates -- des Byte-Codes. Dieser wird von einer plattformabhängig implementierten Programmeinheit, der *Java Virtual Machine* (Abk. JVM) interpretativ zur Ausführung gebracht.
Jede Java-Applikation wird auf einer eigenen virtuellen Maschine zur Ausführung gebracht. Dies garantiert eine größtmögliche Abschottung, mit dem Ziel maximierter Sicherheit, der möglicherweise gleichzeitig auf einer realen Maschine ausgeführten Java-Programme voneinander.
Das Konzept der virtuellen Maschine, die als Programm auf einer realen Hardware abläuft, ermöglicht eine vergleichsweise einfache und schnelle Portierbarkeit auf neue Zielumgebungen, da lediglich die virtuelle Maschine an die veränderte reale Maschine angepaßt werden muß.

Der Gedanke virtueller Maschinen, die generischen Zwischencode -- oftmals auch als *P-Code* bezeichnet -- ausführen, ist nicht neu. Bereits USCD Pascal, E-BASIC und die verschiedenen SmallTalk-Implementierungen, setzt diesen praktisch um.
Die Realisierung der Befehlsfolgen (Opcodes) innerhalb der virtuellen Maschine von Java ähnelt teilweise frappant der Architektur der für die Züricher Pascal-Implementierung entwickelten (abstrakten) P-Maschine.

Inzwischen steht mit der zAAP (zSeries Application Assist Processor)-Hardware für die IBM-Mainframemaschinen *z890* und *z990* sogar eine vollständige Hardwareimplementierung der JVM zur Verfügung, welche den Charakter der *virtuellen* zu einer *realen* Maschine weiterentwickelt.

Ein Beispiel einer vollständig als Softwar realisierten „klassischen“ virtuellen Maschine ist `java`, die Bestandteil des Java-Development Toolkits von SUN ist.
Bekannte andere virtuelle Maschinen sind: Kaffe oder auch IBMs *Jikes*-Implementierung
Wie bereits bekannt wird eine Java-Applikation durch den Aufruf `java`, gefolgt vom Namen der Startklasse und etwaiger Kommandozeilenparameter ausgeführt. Technisch gesehen bewirkt der Aufruf zunächt die Erzeugung einer neuen Instanz der virtuellen Maschine, auf welcher die Programm-Abarbeitbung mit der `main`-Methode der Startklasse begonnen wird.
Eine Instanz einer virtuellen Maschine existiert, solange Programmfäden (engl. *Threads*) (genaugenommen: non-deamon Threads) ausgeführt werden, bzw. die virtuelle Maschine explizit beendet wird (mit dem API-Aufruf `System.exit()`) oder ein Fehler auftritt.



Die wesentlichen Bestandteile der JVM sind:

- *method area*
  Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält klassenspezifische Daten (Typinformation).
- *heap*
  Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält die dynamisch erzeugten Objekte.
- *Java VM stacks*
  Threadspezifisch. Enthält Zustand von nicht-nativen Methodenaufrufen, deren lokale Variablen, Übergabeparameter, den möglichen Rückgabewert sowie Methoden-interne Berechnungsergebnisse. (vgl. JVM-Spezifikation)
- *pc registers*
  Threadspezifisch; enthält für jeden Zeitpunkt der Ausführung einer nicht-nativen Methode die Adresse der derzeit ausgeführten Instruktion. (vgl. JVM-Spezifikation). Während der Abarbeitung nativer Methoden ist der Wert auf `undefined` gesetzt. Konkrete Größe des virtuellen pc-Registers hängt von der Adresslänge der realen Plattform ab.
- *native method stack*
  Implementierungsspezifischer Speicherbereich zur Behandlung von nativen Methodenaufrufen.
- *runtime constant pool*
  Klassen- oder Schnittstellenspezifisch. Enthält verschiedene Konstanten, die zur Übersetzungszeit feststehen und in der `constant_pool` Tabelle der class-Datei abgelegt sind. Die Funktion dieses Bereichs ähnelt dem einer konventionellen Symboltabelle. (vgl. JVM-Spezifikation)

Die Java-Stacks sind in *stack frames* organisert. Jedem Methodenaufruf ist ein Stack-Frame zugeordnet, der beim Aufruf erzeugt (push), und beim Verlassen (pop) vom Stack entnommen wird.
Innerhalb eines Frames befindet sich

- Array der lokalen Variablen
- Operanden-Stack
  Wichtigste Datenstruktur innerhalb der virtuellen Maschine. Der Operanden-Stack enthält alle Eingangs- und Ausgangsdaten der verschiedenen Berechnungen. Seine Einträge sind typisierten in 32-Bit Worten organisiert; die korrekte Typisierung der Eingangoperanden wird von jedem Opcode geprüft.
- Referenz auf runtime constant pool
- implementierungsspezifische- und debugging-Information

Den für den Anwendungsentwickler offensichtlichsten Bestandteil der virtuellen Maschine, bilden jedoch die JVM-Instruktionen -- die Maschinensprache der JVM.
Der Befehlssatz der JVM umfaßt ausschließlich genau ein Byte lange Opcodes.
Die JVM ist generell *stack-orientiert*. Dies bedeutet, daß Quell- und Zieloperanden der meisten Operationen werden vom Stack entnommen, und das Ergebnis dort abgelegt. Insbesondere existieren, abgesehen von vier Verwaltungsspeicherplätzen je Ausführungs-Thread, keine virtuellen Prozessorregister, um die Implementierungsanforderungen an die reale Plattform zu minimieren.

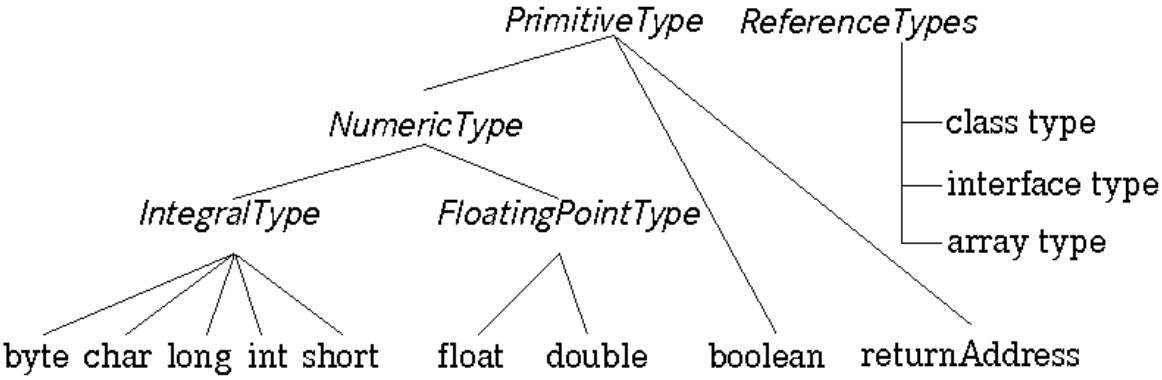
Als threadlokale Register stehen zur Verfügung:

- pc -- program counter
  Enthält die Adresse der gerade ausgeführten Instruktion.
  Handelt es sich um eine native-Methode, die in einer anderen Programmiersprache ausgeführt wird, so ist der Wert *undefined*.
- `optop`
  Verweist auf die Spitze des Operanden Stacks.
- `frame`
  Verweist auf die Ausführungsumgebung der derzeit aktiven Methode.
- `vars`
  Ein Zeiger auf die erste lokale Variable der aktuellen Methode.

Die Adresslänge innerhalb der JVM ist auf vier Byte (32 Bit) fixiert. Hieraus ergibt sich ein (theoretischer) Adressraum von 4 GB.
Die initiale und maximale Ausdehnung des Heaps kann durch die Kommandozeilenschalter `Xms` bzw. `Xmx` gesteuert werden (Beispiel: `java -Xms350M -Xmx500M HelloWorld` führt ein einfaches Hello-World-Beispiel mit einer

anfänglichen Speicherausstattung von 350 MB aus, die im Verlaufe des Programmablaufs auf höchstens 500 MB anwachsen kann.)

Das **Typsystem der JVM** lehnt sich eng an das der Hochsprache an. (Zur Erinnerung: primitive Typen in Java)
Zusätzlich erweitert es die Primitivtypen um einen Adresstypen `returnAddress` und führt explizite Referenztypen auf die verschiedenen high-level Typen (Klassen, Schnittstellen, Arrays) ein.



Datentypen der JVM:

- `byte`
  Vorzeichenbehaftete 8-Bit Zahl in Zweierkomplementdarstellung.
- `short`
  Vorzeichenbehaftete 16-Bit Zahl in Zweierkomplementdarstellung.
- `int`
  Vorzeichenbehaftete 32-Bit Zahl in Zweierkomplementdarstellung.
- `long`
  Vorzeichenbehaftete 64-Bit Zahl in Zweierkomplementdarstellung.
- `char`
  Vorzeichenbehaftete 16-Bit Zahl zur Darstellung von Unicode-Zeichen.
- `float`
  32-Bit Binärdarstellung einer Fließkommazahl gemäß IEEE 754-1985.
- `double`
  64-Bit Binärdarstellung einer Fließkommazahl gemäß IEEE 754-1985.
- `boolean`
  Wird zwar durch die JVM definiert, jedoch existieren keine Opcodes, die Parameter dieses Typs erwarten. Der Compiler setzt Anweisungen die `boolean`-Typen enthalten als Operationen auf `int`-Typen um.
- `returnAddress`
  Für den Programmierer nicht zugänglicher Typ, der Sprungadressen aufnimmt.
- Referenztypen
  Verweisen auf Klassen, Arrays oder Schnittstellen.
  Der Wert kann auch `null` sein, wobei die JVM keine konkrete Darstellung dieses Wertes unterstellt.

Jede Bytecode-Instruktion besteht zunächst aus ihrem Opcode, optional gefolgt von den benötigten Operanden. Diese stehen jedoch nicht für sich, sondern sind eingebettet in den organisatorischen Rahmen der `class`-Datei, deren Format im Anschluß vorgestellt wird.

Die verschiedenen Maschineninstruktionen lassen sich in Klassen einteilen:

Instruktionen zum Zugriff auf lokale Variablen:

Instruktion	Funktion
`ret`	Rücksprung aus Subroutine, wird in der Implementierung von `finally` benutzt
`aload`	Lädt Referenz von spezifisch indizierter Position auf den Operanden-Stack
`aload <n>`	Lädt Referenz auf Operanden-Stack (Index im Opcode explizit angegeben)
`astore`	Speichert auf Operanden Stack liegenden Wert in lokale Variable
`astore <n>`	Legt Referenz in lokaler Variable auf Operanden-Stack ab (Index wird im Opcode explizit angegeben)
`dload`	Lädt `double`-Wert aus lokaler Variable auf den Operanden-Stack
`dstore`	Lädt `double`-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
`fload`	Lädt `float`-Wert aus lokaler Variable auf den Operanden-Stack
`fstore`	Lädt `float`-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
`iload`	Lädt `int` auf Operanden-Stack (Index im Opcode explizit angegeben)
`iload <n>`	Lädt `int`-Wert aus lokaler Variable auf den Operanden-Stack
`istore`	Legt `int`-Wert von spezifisch indizierter Position in lokaler Variable auf Operanden-Stack ab
`istore <n>`	Lädt `int`-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab

lload	Lädt long-Wert aus lokaler Variable auf den Operanden-Stack
lstore	Lädt long-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
iinc	Inkrementiert lokale Variable um fixe int-Zahl

Instruktionen zur expliziten Modifikation des Operanden-Stacks:

Instruktion	Funktion
bipush	Ablegen eines byte-Wertes auf dem Operanden-Stack
sipush	Ablegen eines short-Wertes auf dem Operanden-Stack
pop	Entnimmt und verwirft (de facto: löscht) obersten Operanden-Stack-Eintrag.
pop2	Entnimmt (de facto: löscht) obersten beiden Operanden-Stack-Einträge.
swap	Tauscht die beiden obersten Operanden-Stack-Einträge aus.
ldc	Ablegen eines Elements (referenziert über 16-Bit Index) des runtime constant pools auf dem Operanden-Stack
ldc_w	Ablegen eines Elements (referenziert über 32-Bit Index) des runtime constant pools auf dem Operanden-Stack
aconst_null	Legt null auf dem Operanden-Stack ab.
dconst_<*d*>	Legt double Konstante auf dem Operanden-Stack ab. dconst_0 die Konstante 0.0, bzw. dconst_1 den Wert 1.0.
fconst_<*f*>	Legt float Konstante auf dem Operanden-Stack ab. fconst_0 die Konstante 0.0, bzw. fconst_1 den Wert 1.0.
iconst_<*i*>	Legt int-Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_m1 -- -1; iconst_0 -- 0; iconst_1 -- 1; iconst_2 -- 2; iconst_3 -- 3; iconst_4 -- 4; iconst_5 --5.
dup	Dupliziert oberstes Element des Operanden-Stacks. dup_x1 legt das neue Element als zweitunterstes auf dem Stack ab. dup_x2 legt das neue Element, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes auf dem Stack ab.
dup2	Dupliziert die beiden obersten Elemente des Operanden-Stacks. dup2_x1 legt die neuen Elemente als zweitunterstes und folgendes auf dem Stack ab. dup2_x2 legt die neuen Elemente, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes und folgendes auf dem Stack ab.
lconst_<*l*>	Legt long Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_0 -- 0; iconst_1 -- 1.

Instruktion zur Steuerung des Kontrollflußes:

Instruktion	Funktion
goto	Bedingungsloser Sprung innerhalb derselben Methode. Der anzugebende *branch offset* ist auf 16-Bit fixiert, woraus sich ableiten läßt, daß das Opcodesegment einer Methode niemals (in der JVM-Version 1.3) die Größe von 64KByte überschreiten darf. Näheres zu den Einschränkungen der virtuellen Maschine findet sich in im Abschnitt 4.10 der JVM-Spezifikation, sowie in der Diskussion des Class-File Formats.
goto_w	Bedingungsloser Sprung innerhalb derselben Methode (mit 32-Bit Offset)
if_acmp<*cond*>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als Referenzen übergeben. Als Bedingungen stehen Gleichheit (eq) und Ungleichheit (ne) zur Verfügung.
if_icmp<*cond*>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als int-Werte übergeben. Als Bedinungen stehen zur Verfügung: Gleichheit (eq), Ungleichheit (ne), Kleiner (lt), Kleiner oder Gleich (le), Größer (gt) und Größer oder Gleich (ge).
if<*cond*>	Bedingter Sprung, nach Vergleich des obersten Elements des Operanden-Stacks mit Null Für spezifische Vergleiche stehen folgende Opcodes zur Verfügung: Geleichheit (ifeq), Ungleichheit (ifne), Kleiner (iflt), Kleiner oder Gleich (ifle), Größer (ifgt), Größer oder Gleich (ifge).
ifnonnull	Bedingter Sprung -- im Falle der Ungleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
ifnull	Bedingter Sprung -- im Falle der Gleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
jsr	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 16-Bit Adresse.
jsr_w	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 32-Bit Adresse.
lookupswitch	Zugriff auf Sprungtabelle per Schlüssel und anschließende Verzweigung. Benutzt zur Implementierung des switch-Konstrukts
tableswitch	Indexbasierter Zugriff auf Sprungtabelle und anschließende Verzweigung.

Instruktionen zur Operation auf Klassen und Objekten:

Instruktion	Funktion
anewarray	Array-Erzeugung an definierter Stelle im Laufzeit-Konstanten-Pool.
checkcast	Typkompatibilitätsprüfung (siehe Beispiel)
instanceof	Prüft ob Objekt gegebenen Typ hat (d.h. Ausprägung der Klasse -- oder einer Subklasse -- ist; das Interface implementiert; Array-Kompatibel ist)
new	Erzeugt neues Objekt *Hinweis*: Die Punktnotation zur Trennung der Pakethierarchien wird hier durch Slash „/“ ersetzt.

Instruktionen zur Methodenausführung:

Instruktion	Funktion
invokespecial	Ruft Instanzenmethode auf; mit besonderer Behandlung bestimmter Umstände. *Hinweis*: Diese Instruktion wurde umbenannt, frühere JDK-Versionen benutzen `invokeonvirtual`.
invokestatic	Ruft statische Klassenmethode auf.
invokevirtual	Ruft Instanzenmethode auf.
invokeinterface	Ruft Schnittstellenmethode auf.
areturn	Retourniert Referenz auf Speicherobjekt nach Methodenausführung.
dreturn	Retourniert `double`-Wert nach Methodenausführung.
freturn	Retourniert `float`-Wert nach Methodenausführung.
ireturn	Retourniert `int`-Wert nach Methodenausführung.
lreturn	Retourniert `long`-Wert nach Methodenausführung.
return	Retourniert nach Methodenausführung ohne Rückgabewert (`void`-Methode).

Instruktionen zum Zugriff auf Attribute:

Instruktion	Funktion
getfield	Legt Attributinhalt eines Objekts auf Operanden-Stack ab. Die Klasse des Objekts, auf das der Zugriff erfolgen soll, wird über einen 16-Bit Offset auf dem runtime constant pool addressiert. Auch hier wird wieder die Limitierung der virtuellen Maschine auf 2^{16} Klassen deutlich.
getstatic	Legt Attributinhalt eines statischen Klassenattributes auf dem Operanden-Stack ab.
putfield	Setzt Wert eines Attributs.
putstatic	Setzt Wert eines statischen Klassenattributes.

Instruktionen zur Operation auf Arrays:

Instruktion	Funktion
newarray	Erzeugt einen neuen Array, und definiert die Komponententypen als einen der Primitvtypen. Die Implementierung von SUN verwendet für den Wahrheitswert je acht Bit. Anderen Umsetzungen ist es explizit Freigestellt hier mit optimierteren Speicherstrukturen zu operieren.
anewarray	Erzeugt einen neuen Array von Referenztypen zur Aufnahme beliebiger Objekte.
multianewarray	Erzeugt einen mehrdimensionalen Array.
aaload	Lädt Referenz von Arrayposition.
aastore	Speicher Objekt an spezifischer Arrayposition.
arraylength	Liefert Elementanzahl (Kardinalität) eines Arrays.
baload	Lädt `byte` oder `boolean` aus Arrayposition.
bastore	Legt `byte` oder `boolean` an Arrayposition ab.
saload	Lädt `short` aus Arrayposition.
sastore	Legt `short` an Arrayposition ab.
caload	Lädt `char` aus Arrayposition.
castore	Legt `char` an Arrayposition ab.
daload	Lädt `double` aus Arrayposition.

dastore	Legt double an Arrayposition ab.
faload	Lädt float aus Arrayposition.
fastore	Legt float an Arrayposition ab.
iaload	Lädt int aus Arrayposition.
iastore	Legt int an Arrayposition ab.
laload	Lädt long aus Arrayposition.
lastore	Legt long an Arrayposition ab.

Instruktionen zur Typkonversion:

Instruktion	Funktion
i2b	Konvertiert int zu byte.
i2c	Konvertiert int zu char.
i2d	Konvertiert int zu double.
i2f	Konvertiert int zu float.
i2l	Konvertiert int zu long.
i2s	Konvertiert int zu short.
d2f	Konvertiert double zu float.
l2d	Konvertiert long zu double.
l2f	Konvertiert long zu float.
l2i	Konvertiert long zu int.
d2f	Konvertiert double zu float.
f2d	Konvertiert float zu double.
f2i	Konvertiert float zu int.
f2l	Konvertiert float zu long.
d2f	Konvertiert double zu float.
d2f	Konvertiert double zu float.
d2i	Konvertiert double zu int.
d2l	Konvertiert double zu long.

Instruktionen zur Durchführung arithmetischer Operationen:
Eingangsperanden werden vom Stack entnommen und das Berechnungsergebnis ebenda abgelegt.

Instruktion	Funktion
dadd	Addiert zwei double-Werte.
dsub	Subtrahiert zwei double-Werte.
ddiv	Dividiert zwei double-Werte.
dmul	Multipliziert zwei double-Werte.
dneg	Negiert double-Wert durch Zweierkomplementbildung.
drem	Divisionsrest bei Division zweier double-Werte.
dcmp<*op*>	Vergleicht zwei double Werte. Ist einer der beiden Operanden NaN, so legt dcmpg1, dcmpl hingegen -1 als Ergebnis auf dem Stack ab.
fadd	Addiert zwei float-Werte.
fsub	Subtrahiert zwei float-Werte.
fmul	Multipliziert zwei float-Werte.
fdiv	Dividiert zwei float-Werte.
iadd	Addiert zwei int-Werte.
isub	Subtrahiert zwei int-Werte.
imul	Multipliziert zwei int-Werte.

idiv	Dividiert zwei int-Werte.
iand	Boole'sche UND-Verknüpfung zweier int-Werte.
ior	Boole'sche ODER-Verknüpfung zweier int-Werte.
ixor	Exklusive Boole'sche ODER-Verknüpfung zweier int-Werte.
ineg	Negiert int-Wert durch Zweierkomplementbildung.
irem	Divisionsrest bei Division zweier int-Werte.
ishl	Linksshift eines int-Wertes.
ishr	Rechtsshift eines int-Wertes.
iushr	Rechtsshift eines int-Wertes unter Nulleinfügung und Vorzeichenlösung.
fneg	Negiert float-Wert durch Zweierkomplementbildung.
frem	Divisionsrest bei Division zweier float-Werte.
fcmp<*op*>	Vergleicht zwei floatWerte. Ist einer der beiden Operanden NaN, so legt fcmpg1, fcmpl hingegen -1 als Ergebnis auf dem Stack ab.
ladd	Addiert zwei long-Werte.
land	Boole'sche UND-Verknüpfung zweier long-Werte.
ldcmp	Vergleicht zwei longWerte.
ldiv	Dividiert zwei long-Werte.
lmul	Multipliziert zwei long-Werte.
lneg	Negiert long-Wert durch Zweierkomplementbildung.
lrem	Divisionsrest bei Division zweier long-Werte.
lor	Boole'sche ODER-Verknüpfung zweier long-Werte.
lshl	Linksshift eines long-Wertes.
lshr	Rechtsshift eines long-Wertes.
lsub	Subtrahiert zwei long-Werte.
lushr	Rechtsshift eines long-Wertes unter Nulleinfügung und Vorzeichenlösung.
lxor	Exklusive Boole'sche ODER-Verknüpfung zweier long-Werte.

Sonstige Instruktionen:

Instruktion	Funktion	
Ausnahmebehandlung	athrow	Wirft eine Exception.
Synchronisation -- Monitoroperationen	monitorenter	Der Zugriff auf jedes Objekt wird durch einen Monitor synchronisiert. Die Anweisung sperrt ein Objekt.
	monitorexit	Gibt ein gesperrtes Objekt frei.
Sonstige Opcodes	nop	Buchstäblich: *no operation*. Bewirkt nichts; keine Operatoren, keine Änderungen am Operanden-Stack.

Die beiden Opcodes mit den Ordnungsnummern 254 und 255 (0xfe und 0xff, mnemonic impdep1 und impdep2) sind durch SUN als reserviert gekennzeichnet. Sie können von durch den Hersteller der virtuellen Maschine mit eigendefinierter Funktionalität implementiert werden.
Darüberhinaus ist mit dem Opcode 202 (mnemonic breakpoint) ein Einstiegspunkt für Debugger definiert.

Alle Opcodes sind mit einem Byte codiert. Hieraus ergibt 256 als maximaler Befehlsumfang der virtuellen Maschine.
Zur Verringerung der notwendigen verschiedenen Befehle sind nicht alle Opcodes für alle Typen der JVM implementiert. Üblicherweise existieren nur Opcodes für int, float, long und double sowie die Referenzen. Für alle anderen Typen stehen Konvertierungsmöglichkeiten in die genannten zur Verfügung.

Opcode	byte	short	int	long	float	double	char	Referenz
...ipush	bipush	sipush						
...const			iconst	lconst	fconst	dconst		aconst
...load			iload	lload	fload	dload		aload
...store			istore	lstore	fstore	dstore		astore
...inc			iinc					

...aload	baload	saload	iaload	laload	faload	daload	caload	aaload
...astore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
...add			iadd	ladd	fadd	dadd		
...sub			isubb	lsub	fsub	dsub		
...mul			imul	lmul	fmul	dmul		
...div			idiv	ldiv	fdiv	ddiv		
...rem			irem	lrem	frem	drem		
...neg			ineg	lneg	fneg	dneg		
...shl			ishl	lshl				
...shr			ishr	lshr				
...ushr			iushr	lushr				
...and			iand	land				
...or			ior	lor				
...xor			ixor	lxor				
i2...	i2b	i2s		i2l	i2f	i2d		
l2...			l2i		l2f	l2d		
f2...			f2i	f2l	f2d			
...cmp				lcmp				
...cmpl				fcmpl	dcmpl			
...cmpg				fcmpg	dcmpg			
if_...cmp*cond*			if_icmp*cond*					if_acmp*cond*
...return			ireturn	lreturn	freturn	dreturn		areturn

Treten bei der Ausführung der Opcodes Ausnahmen auf, so werden durch die virtuelle Maschine Laufzeit-Ausnahmeereignisse (engl. *runtime exception*) generiert. Wie bekannt werden Ausnahmen dieser Kategorie (üblicherweise) nicht aufgefangen und behandelt.
So wird die `ClassCastException` im Beispiel aus Kapitel 2 durch die versuchte explizite Typumwandlung ausgelöst. Der erzeugte Bytecode für diese Anweisung lautet:

```
aload_3
checkcast 2
```

`aload_3` lädt die Referenz auf eine lokale Variable auf den Operanden-Stack. Die lokale Variable 3 entspricht im Beispiel `myC11`.
`checkcast` testet ob die auf dem Operanden-Stack befindliche Referenz kompatibel zum übergebenen Typen (hier die 2 als Referenz auf die zweite geladene Klasse; benannt mit `C2`) ist. Im Falle der Nichtkompatibilität wird durch die virtuelle Maschine eine `ClassCastException` erzeugt.

Beispiel 6: Einfache arithmetische und Ein-/Ausgabeoperationen

```
(1).class public examples/BC1
(2).super java/lang/Object
(3)
(4).method public <init>()V
(5)   aload_0
(6)   invokenonvirtual java/lang/Object/<init>()V
(7)   return
(8).end method
(9)
(10).method public static main([Ljava/lang/String;)V
(11)    .limit locals 5
(12)   .limit stack 10
(13)
(14)    iconst_2
(15)    istore_0
(16)
(17)    bipush 101
(18)    istore_1
(19)
(20)    bipush 99
(21)    istore_2
(22)
(23)    ;we will need this twice
(24)    getstatic java/lang/System/out Ljava/io/PrintStream;
(25)    astore_3
(26)
(27)    iload_1
```

```
(28)    iload_2
(29)    iadd
(30)    istore_1
(31)
(32)    ;convert int to string
(33)    iload_1
(34)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(35)    astore 4
(36)
(37)    ;Print a string
(38)    aload_3
(39)    aload 4
(40)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(41)
(42)    iload_1
(43)    iload_0
(44)    idiv
(45)
(46)    ;convert int to string
(47)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(48)    astore 4
(49)
(50)    ;Print a string
(51)    aload_3
(52)    aload 4
(53)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(54)
(55)    ;print a fixed string
(56)    aload_3
(57)    ldc "The End"
(58)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(59)   return
(60).end method
(61)
```

Download des Beispiels

Der Java-Assemblercode des Beispiels 6 zeigt die Verwendung einiger einfacher arithmetischer und Ein-/Ausgabeoperationen.
Zunächst zeigt das Beispiel den Aufbau einer Java-Assemblerdatei, wie sie vom Übersetzer *Jasmin* akzeptiert und in ausführbaren Java-Bytecode umgewandelt wird.
So legt die `.class`-Deklaration zunächst fest, daß es sich um die öffentlich zugängliche (d.h. als `public` deklarierte) Klasse `BC1`, im Paket `examples` handelt.
Die darauf folgende `.super`-Definition legt die Elternklasse der betrachteten Klasse fest. Im Falle keiner explizit definierten Elternklasse ist dies vorgabegemäß die Standardklasse `Class`.
Nach den einführenden Deklarationen definiert die Quellcodedatei den statischen Initialisierter.

Die Methode `main` stellt den Beginn der aktiven Verarbeitung dar. Ihre Signaturdefinition (`([Ljava/lang/String;)V`) läßt die JVM-interne Kurzschreibweise des Typsystems erkennen. So deutet die in den Klammern der Übergabewerte eingeschlossene eckige Klammern an, daß ein Array gleichtypisierter Ausprägungen übergeben wird. Diese Ausprägungen sind alle vom Standardtyp `java.lang.String` (die paket-separierenden Punkte werden JVM-intern zu Pfadseparatoren aufgelöst). Zusätzlich ist dem Klassenname ein einleitendes `L` vorangestellt, um auszudrücken, daß es sich nicht um einen Primitivtyp, sondern um eine Sprachkomponente (das „L“ deutet hierbei auf den Begriff *language* hin) handelt.
Nach der Klammer ist der Rückgabetyp --- im Falle von `main` vorgabegemäß `void` --- angegeben. Auch er wird unter Verwendung derselben Abkürzungskonvention dargestellt.

Zu Eingangs der Methode `main` allozieren die beiden Direktiven `.limit` zunächst Speicher für die lokalen Variablen (`.limit locals`) und die Tiefe des methodenintern verwendeten Operandenstacks (`.limit stack`).

Die (aktive durch den Programmierer gesteuerte) Verarbeitung beginnt im Beispiel mit dem Anweisung `iconst_2` welche den ganzzahligen Wert 2 auf dem Operandenstack ablegt. Anschließend wird dieser Wert, mittels der Anweisung `istore_0` vom Stack entnommen und in die erste lokale Variable gespeichert.
Mit `iconst_2` findet ein besonderer Befehl zur Ablage einer ganzzahligen Konstante auf dem Operanden Stack Verwendung, der es gestattet bestimmte (häufig benötigte) Konstantenablagen in nur genau einem Byte auszudrücken. Durch die JVM-Spezifikation vorgesehen sind hierbei Instruktionen für die Konstanten `-1, 0, 1, 2, 3, 4` oder `5`. Im Ergebnis ist die Nutzung der abkürzenden Befehlsschweibweise äquivalent zum Einsatz der Instruktion `bipush` unter Explizierung der abzulegenden Konstante.

Diese äquivalente Form der Belegung einer lokalen Variable zeigt der zweite Anweisungsblock, der die numerische Konstante `101`, für die keine abkürzende Schreiweise angeboten wird, auf dem Operandenstack ablegt um sie der zweiten lokalen Variable (mit der Indexnummer 1) zuzuweisen.
In derselben Weise wird für die Initialisierung der dritten lokalen Variablen mit dem Wert `99` verfahren.

Anschließend wird durch `getstatic` der Dateideskriptor der Standardausgabe (d.h. desjenigen Streams mit dem Wert `System/out`) gelesen und die zurückgelieferte Adresse in der vierten lokalen Variable (Indexnummer 3) abgelegt.

Der darauffolgende Anweisungsblock zeigt die Umsetzung einer einfachen Ganzzahladdition, die zunächst die beiden zu verknüpfenden Operanden (die Inhalte der lokalen Variablen mit den Indexnummern 1 und 2) auf dem Stack

ablegt und anschließend mittels der Ganzzahladdition (iadd) verknüpft.
Das auf dem Stack abgelegte Berechnungsergebnis wird durch istore_1 er zweiten lokalen Variablen zugewiesen.

Der nächste Anweisungsblock bereitet die Ausgabe des Berechnungsergebnisses auf der Standardausgabe vor.
Hierzu plaziert er zunächst den Inhalt der lokalen Variable mit der Indexnummer 1 (d.h. das Berechnungsergebnis des direkt vorhergehenden Schrittes) auf dem Stack.
Anschließend wird eine Standard-API-Methode (die Methode valueOf) aufgerufen, welche den auf dem Stack übergebenen int-Parameter in eine Zeichenkette wandelt und die Referenz darauf als Rückgabewert auf dem Stack plaziert.
Dieser Rückgabewert wird in der fünften lokalen Variable (Indexnummer 4) abgelegt.

Anschließend werden die in zwischenzeitlich den vierten und fünften lokalen Variablen abgelegten Adressen des Ausgabe-Streams und der auszugebenden Zeichenkette geladen und auf dem Operanden-Stack abgelegt.
Durch Aufruf der Standard-Ausgabemethode println mittels invokevirtual wird die referenzierte Zeichenkette auf der Standardausgabe dargestellt.

Der folgende Anweisungsblock demonstriert eine Ganzzahldivision mittels idiv welche Divisior und Dividenden als Operadnen auf dem Stack erwartet und das Berechnungsergebnis ebenda plaziert.

Anschließend wird das (noch auf dem Stack liegende) Berechnungsergebnis direkt weiterverarbeitet und in eine Zeichenkette gewandelt. Hierbei kommt die bereits bekannte Funktion zum Einsatz.
Danach erfolgt wiederum die Ausgabe in der bekannten Form.

Abschließend wird eine fixe Zeichenkette ausgegeben, deren Zeichenkettendarstellung nicht berechnet zu werden braucht. Ihr Wert kann daher direkt aus dem Laufzeitkonstantenpool per ldc geladen werden.
Die übrigen Schritte zur Erzeugung der Ausgabe bleiben indes unverändert.

Nutzung von Methoden:
Bereits bei den einfachen Operationen aus Beispiel 6 zeigt sich, daß die wiederholte Angabe von sehr ähnlichen Instruktionsfolgen nicht zu vermeiden ist. Insbesondere die beiden Konversionen des int-Datentyps als Voraussetzung der zeichenbasierten Ausgabe ist vollständig identisch.
Zur Strukturierung stehen daher auf der Java-Assemblerebene die bereits aus der Java-Hochsprache bekannten *Methoden* zur Verfügung, wie Beispiel 7 zeigt.

Beispiel 7: Nutzun von Methoden

```
(1).class public examples/BC2
(2).super java/lang/Object
(3)
(4).method public <init>()V
(5)   aload_0
(6)   invokenonvirtual java/lang/Object/<init>()V
(7)   return
(8).end method
(9)
(10).method public static printInt(I)V
(11)    .limit locals 2
(12)    .limit stack 2
(13)
(14)    iload_0
(15)    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
(16)    astore_1
(17)
(18)    getstatic java/lang/System/out Ljava/io/PrintStream;
(19)    aload_1
(20)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(21)    return
(22).end method
(23)
(24).method public static main([Ljava/lang/String;)V
(25)    .limit locals 50
(26)   .limit stack 40
(27)
(28)    iconst_2
(29)    istore_0
(30)
(31)    bipush 101
(32)    istore_1
(33)
(34)    bipush 99
(35)    istore_2
(36)
(37)    ;we will need this twice
(38)    getstatic java/lang/System/out Ljava/io/PrintStream;
(39)    astore_3
(40)
(41)    iload_1
(42)    iload_2
(43)    iadd
(44)    istore_1
(45)
(46)    iload_1
```

```
(47)    invokestatic examples/BC2/printInt(I)V
(48)
(49)    iload_1
(50)    iload_0
(51)    idiv
(52)
(53)    invokestatic examples/BC2/printInt(I)V
(54)
(55)    ;print a fixed string
(56)    aload_3
(57)    ldc "The End"
(58)    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
(59)   return
(60).end method
```

Download des Beispiels

Die Funktionalität des Beispieles ist mit der der vorhergend vorgestellten Codesequenz identisch. Jedoch finden sich jetzt die Instruktionsfolgen zur Berechnung der Zeichenkettenrepräsentation einer Ganzzahl und ihrer anschließenden Ausgabe in die Methode `printInt` ausgelagert.

Diese Methode akzeptiert eine Ausprägung des Primitivtyps `int` als Übergabe und liefert keinen Rückgabewert.
Die Signatur ist daher dahingehend vereinbart, daß genau eine `int`-konforme Zahl als Parameter auf dem Stack erwartet wird, d.h. der Aufrufer hat diese vor dem Aufruf dort abzulegen.

Zusätzlich benötigt die Methode selbst zu ihrer Ausführung einige lokale Variablen, die auf dem methodenspezifischen Stack abgelegt werden. Dieser stellt eine Erweiterung des bereits durch den Aufruf verwendeten Operandenstacks dar.

Mit Java steht jedoch keineswegs die einzige Hochsprache zur Erzeugung von Byte-Code-Dateien zur Verfügung.
Diese Seite listet eine Vielzahl verschiedener Alternativen.
Beispielsweise erzeugt der Oberon-Compiler von Canterbury für alle Oberon-Module, einschließlich der Systemmodule, Java-Klassen.

Beispiel 8: Die Hello World Applikation als Oberon Programm

```
MODULE helloworld;

IMPORT Out;

BEGIN
  Out.String( "Hello World" );
  Out.Ln;
END helloworld.
```

Download des Beispiels

Die erzeugten `class`-Dateien -- SYSTEM.class, helloworld.class, Out.class, Sys.class -- können auf jeder JVM zur Ausführung gebracht werden. `java helloworld` liefert das erwartete Ergebnis.

Das **Class-File-Format**

Ausgangspunkt jeder Programmausführung innerhalb der JVM ist die `class`-Datei als Eingabe. Sie wird üblicherweise durch durch den Java-Compiler (im JDK: `javac` erzeugt).

Einige Eigenschaften jedes `class`-Files:

- Es enthält die Definition genau einer Klasse oder einer Schnittstelle, unabhängig von deren Zugriffs- und Sichtbarkeitseigenschaften.
- Es wird als Bytestrom aufgefaßt, daher werden alle größeren Informationseinheiten durch serielles Lesen und aneinanderfügen gebildet.
  Unabhängig von der physischen Realisierung der tatsächlich zugrundeliegenden Hardware werden alle Multibyte-Daten im *big endian*-Format, d.h. größere Einheiten werden durch Linksshift und Boole'sches ODER gebildet, abgelegt.
  Die Schnittstellen `java.io.DataInput`, `java.io.DataOutput`, `java.io.DataInputStream` und `java.io.DataOutputStream` unterstützen dieses Format. (Beispiel)
- Die Strukturen innerhalb der `class`-Datei werden nicht zusätzlich optimiert abgelegt, daher erfolgt weder ein Auffüllen auf spezifische Wortgrenzen, noch ein Alignment an solchen.

Die JVM-Spezifikation legt zur Definition der Struktur des `class`-Files eigene Datentypen fest: `u1`, `u2` und `u4` zur Definition vorzeichenloser ein-, zwei- und drei-Bytetypen. Für diese (von der Java-üblichen vorzeichenbehafteten Mimik (abgesehen von char) abweichenden) Datentypen stehen mit `readUnsignedByte()`, `readUnsignedShort()` und `readInt()` entsprechende Lesemethoden zur Verfügung.

The `class` File Format @ Java Virtual Machine Specification

```
ClassFile {
      u4 magic;
      u2 minor_version;
      u2 major_version;
```

```
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Constant Pool @ Java Virtual Machine Specification

```
cp_info {
  u1 tag;
  u1 info[];
}
```

field_info @ Java Virtual Machine Specification

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
       }
```

method_info @ Java Virtual Machine Specification

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

attribute_info @ Java Virtual Machine Specification

```
 attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
  }
```

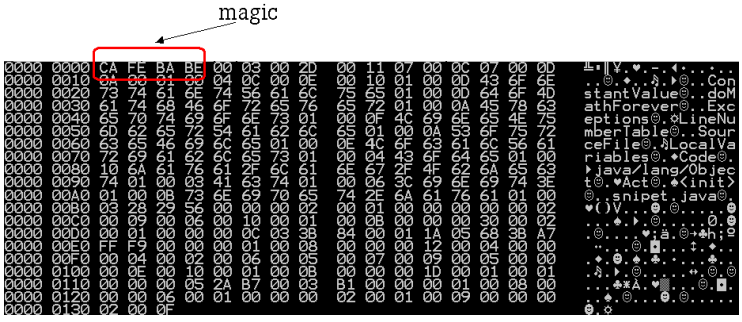
Aufbau einer class-Datei verdeutlicht an nachfolgendem Beispielquellcode.

Hinweis: Mit *Classeditor* existiert ein freies Werkzeug zur Inspektion und Modifikation übersetzter Class-Dateien.

Beispiel 9: Java-Quellcode der untersuchten Klassendatei

```
(1)class Act {
(2)     public static void doMathForever() {
(3)             int i=0;
(4)             while (true) {
(5)                     i += 1;
(6)                     i *= 2;
(7)             } //while
(8)     } //doMathForever()
(9)} //class Act
```

Download des Beispiels

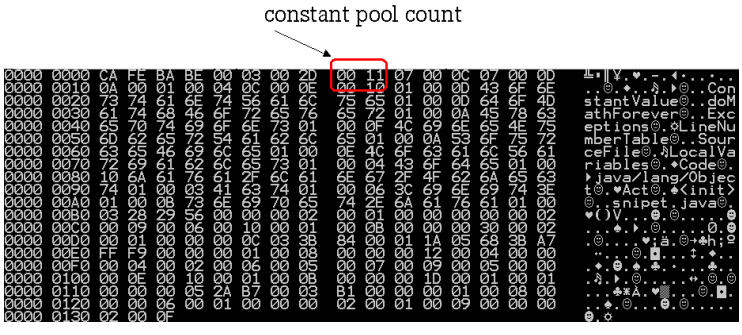


Der `magic`-Identifier ist auf die Bytekombination (in hexadezimaler Darstellung) `CA FE BA BE` fixiert. Anhand dieser erkennt der Kassenlader der Laufzeitsystems die Datei als ausführbare Java-Bytecode-Datei an.
Ist diese gegenüber dem Vorgabewert modifiziert wird eine `java.lang.ClassFormatError` Ausnahme im Hauptthread generiert (`Bad magic number` wird als zusätzliche Nachricht der Ausnahme ausgegeben). siehe JVM-Spezifikation

Die beiden Versionskennungen `minor` und `major` bilden gemeinsam den Versionsschlüssel der `class`-Datei in der gängigen Punktnotation. Hierbei gilt: `major.minor`
Die Klassendatei des Beispiels trägt den Versionsschlüssel `45.3`.
Der im SUN JDK enthaltene Java-Compiler erlaubt per Kommandozeilenparameter (`target`) die JVM-spezifische Steuerung der Codegenerierung. Die den einzelnen Sprachversionen zugeordneten Bytecodeversionen sind in der nachfolgenden Tabelle zusammengestellt.

Java-Version	Bytecode-Version
1.1	45.3
1.2	46.0
1.3	47.0
1.4 (sowie 1.4.1, 1.4.2 und der Prototyp des 1.5-Compilers)	48.0
Zweite Vorabversion des 1.5-Compilers	50.0
1.5 (beta1)	49.0
1.5 (beta2)	49.29

Vorgabegemäß wird durch die Compilerversion 1.5 (ab Beta-Version 2) `49.29` erzeugt.
Trägt eine `class`-Datei eine durch die JVM nicht unterstützte Versionsnummer, so wird eine `java.lang.UnsupportedClassVersionError`-Ausnahme generiert.
Jede JVM kann verschiedene `class`-Datei-Versionen unterstützen, die letzendlich Festlegung welche Versionen durch einzelne Java-Plattform-Releases zu unterstützten sind obliegt jedoch SUN. So unterstützt SUNs JDK v1.0.2 `class`-Dateien der Versionen 45.0 bis einschließlich 45.3. Die JDK-Generation v1.1.x ab Version 45.0 bis einschließlich 45.65535 und Implementierungen der Java 2 Plattform, Version 1.2, bis einschließlich 46.0. JDK v1.3.0 verarbeitet Klassendateien bis hin zur Versionsnummer `47.0`. Zur Verarbeitung von Klassen, welche die in 1.5 eingeführten Generizitätsmechanismen verwenden nicht zwingend eine Ausführungsumgebung dieser Versionsnummer benötigt, da das erzeugte Klassenformat (bisher, da diese Aussagen auf dem Informationsstand der verfügbaren Betaversion basieren) nicht verändert wurde. Die Nutzung des dynamischen Boxing/Unboxings benötigt jedoch eine Ausführungsumgebung mindestens der Version 1.5.
siehe JVM-Spezifikation

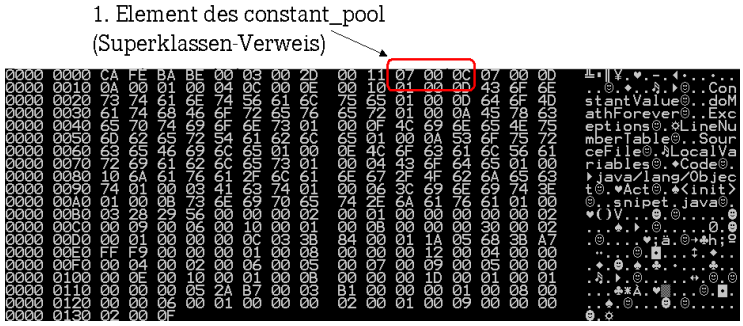


Die Bytefolge `constant_pool_count` enthält die um eins erhöhte Anzahl der Einträge der `constant_pool` Tabelle.
Im Beispiel ist dies: 17.
siehe JVM-Spezifikation

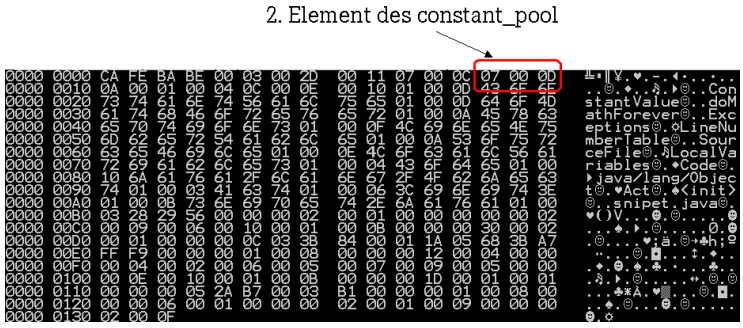
Der `constant pool` enthält die symbolische Information über Klassen, Schnittstellen, Objekte und Arrays. Die Elemente des dieser Datenstruktur sind vom Typ `cp_info` und variieren je nach `tag` in ihrer Länge. Als Konstantentypen (=Inhalt des Tag-Bytes) sind zugelassen:

Konstantentyp	Wert
CONSTANT_Utf8	1
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_NameAndType	12
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_CLASS	7
CONSTANT_String	8
CONSTANT_Fieldref	9

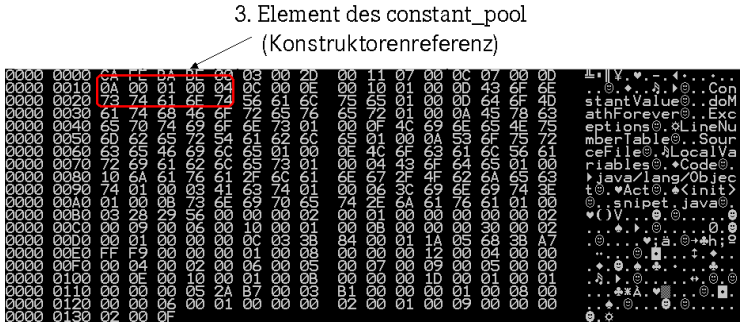
siehe JVM-Spezifikation



Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das zwölfte Element des Konstantenpools (0x0C) enthält. Zum Lesezeitpunkt der ersten Konstante kann diese Referenz noch nicht aufgelöst und auf Gültigkeit geprüft werden. (Später sehen wir, daß es sich um eine Referenz auf `java.lang.Object` handelt).
Hierbei handelt es sich immer um die Referenz auf die Superklasse. Auch für die API-Klasse `Object` selbst findet sich diese Refenz in der Klassendatei, auch wenn Diassemblierungswerkzeuge wie `javap` diese nicht ausgeben.

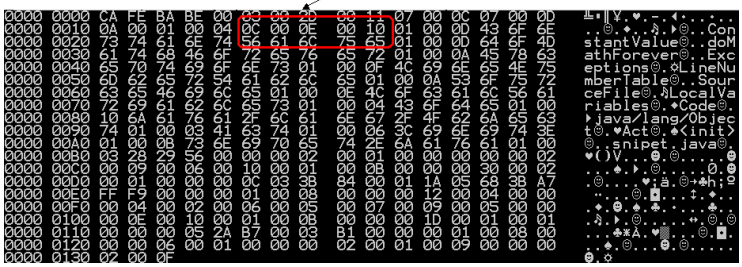


Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das 13. Element des Konstantenpools (0x0D) enthält. An dieser Stelle findet sich der String `Act`, also der Klassenname der zur Klassendatei gehörigen Klasse selbst. Auch hierbei handelt es sich zunächst um eine nicht auflösbare Vorwärtsreferenz.
Technisch gesehen realisiert sie den `this`-Verweis.



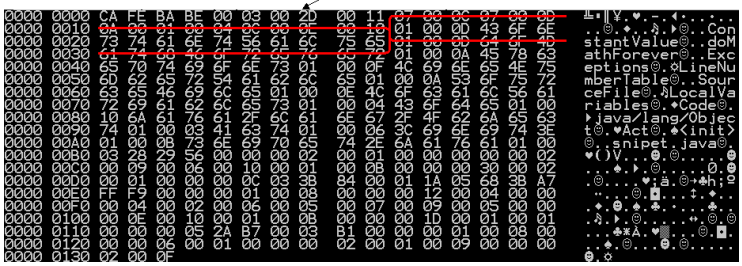
Konstante vom Typ `CONSTANT_Methodref`. Die folgenden zwei Bytes (im Beispiel: 0x00 01) bezeichnen das referenzierte Objekt, gefolgt vom Methodenindex (0x00 04). Im Beispiel handelt es sich um das Objekt mit der Referenznummer 1 (=erstes Element des Konstantenpools, die Superklasse `Object`). Unter der Referenznummer 0x04 wird auf die Methode `<init>()` verweisen. Da die betrachtete Klasse `Act` keinen eigenen Konstruktor definiert, wird der der Superklasse aufgerufen.

4. Element des constant_pool
(Rückgabetyp des 14. Elements)



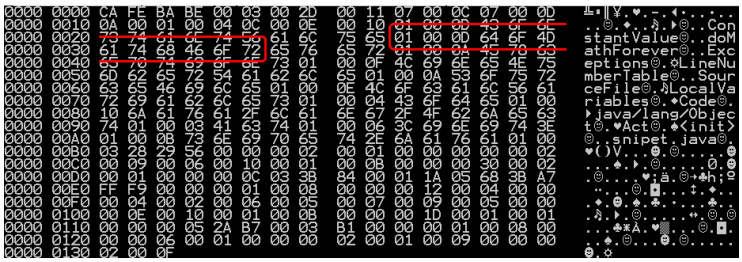
Konstante vom Typ `CONSTANT_NameAndType`. Die folgenden beiden Bytes (`0x00 0E`) verweisen auf die zu beschreibende Position innerhalb des Konstantenpools (im Beispiel: `init`). Dieser Position wird der and Position `0x10` spezifizierte Typ als Rückgabetyp zugeordnet -- im Beispiel `()V`; also `void`.

5. Element des constant_pool
(konstante Zeichenkette)



Konstante vom Typ `CONSTANT_Utf8` leitet einen konstenten Zeichenketten-Ausdruck ein.
Zeichenketten werden generell im Unicode UTF-8 Format abgelegt, wobei jedoch aus Speicherplatzeffizienzgründen für die Zeichen zwischen `\u0001` und `\u007F` nur jeweils ein Byte benötigt werden. Für alle anderen Unicode-Symbole wird der entsprechende 2- bzw. 3-Byte Speicherplatz zur Verfügung gestellt.
Die Konstante wird von der Länge der Zeichenkette (im Beispiel: 13) gefolgt, daher kann auf eine terminierende Null verzichtet werden.
Die Bedeutung dieser -- im Java-Quellcode nicht enthaltenen -- Zeichenkette wird im Kontext des Klassenladevorgangs deutlich.

6. Element des constant_pool
(Methodenname)



Konstante vom Typ `CONSTANT_Utf8`. Sie leitet den in der Klasse `Act` spezifizierten Methodennamen `doMathForever` ein.

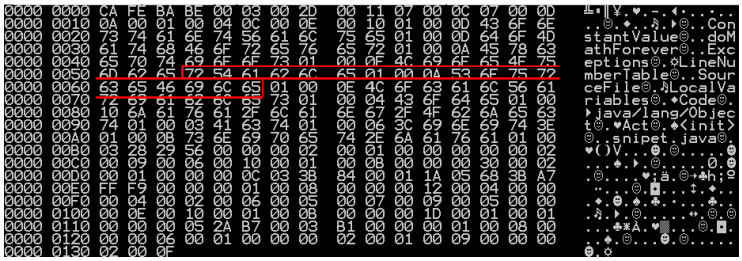
7. Element des constant_pool
(Zeichenkette "Exceptions")

Konstante vom Typ `CONSTANT_Utf8`, die den fixen String *Exceptions* einleitet.

8. Element des constant_pool
(Zeichenkette "LineNumberTable")

Konstante vom Typ CONSTANT_Utf8, die den fixen String *LineNumberTable* einleitet.

9. Element des constant_pool
(Zeichenkette "SourceFile")

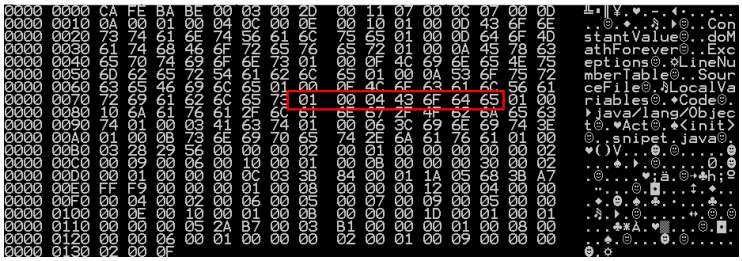


Konstante vom Typ CONSTANT_Utf8, die den fixen String *SourceFile* einleitet.

10. Element des constant_pool
(Zeichenkette "LocalVariable")

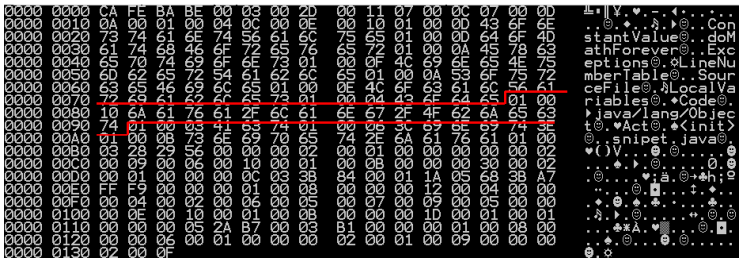
Konstante vom Typ CONSTANT_Utf8, die den fixen String *LocalVariables* einleitet.

11. Element des constant_pool
(Zeichenkette "Code")



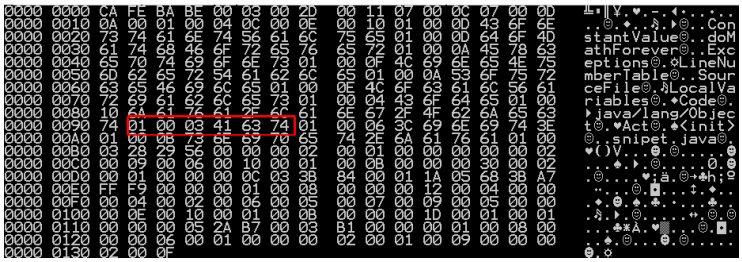
Konstante vom Typ CONSTANT_Utf8, die den fixen String *Code* einleitet.

12. Element des constant_pool
(Zeichenkette "java/lang/Object")



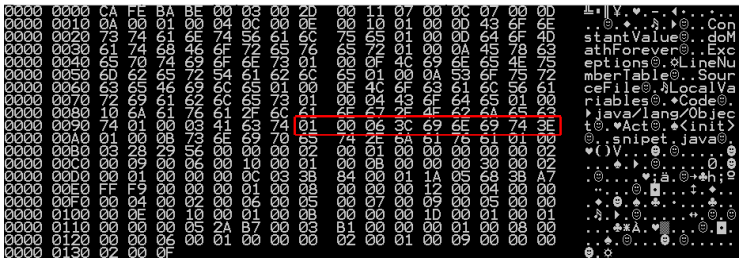
Konstante vom Typ CONSTANT_Utf8, die den String *java/lang/Object* einleitet. Vom ersten Element des Konstantenpools referenziertes Element. Die in der Java-Hochsprache üblichen Punkte zur Trennung der Pakte, Subpakete und Klassennamen werden in der JVM konsequent (aus historischen Gründen) durch Querstriche ersetzt.

13. Element des constant_pool
(Zeichenkette "Act")



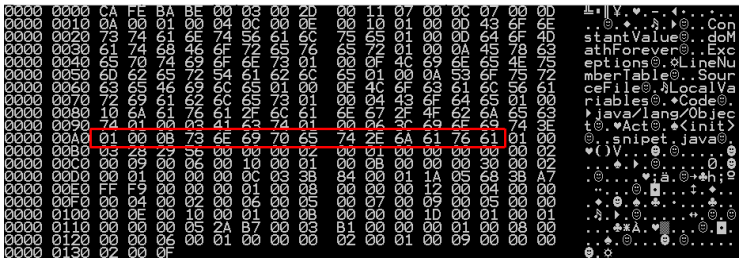
Konstante vom Typ `CONSTANT_Utf8`, die den String *Act* einleitet. Vom zweiten Element des Konstantenpools referenziertes Element. Name der Klasse.

14. Element des constant_pool
(Zeichenkette "<init>")



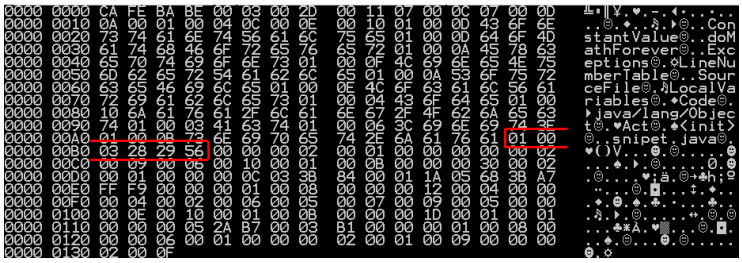
Konstante vom Typ `CONSTANT_Utf8`, die den String *<init>* einleitet. Methodenname der innerhalb der Superklasse `Object`. Der Rückgabewert dieser Methode ist im vierten Element des Konstantenpools abgelegt.

15. Element des constant_pool
(Zeichenkette "snipet.java")



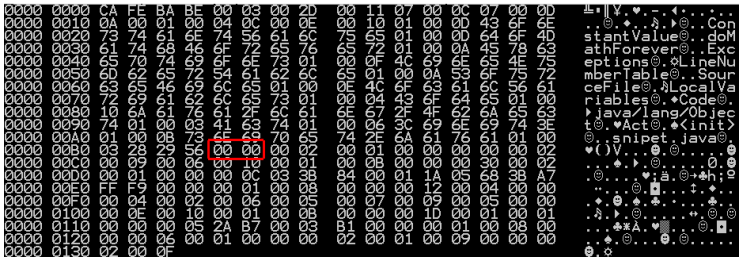
Konstante vom Typ `CONSTANT_Utf8`, die den String *snipet.java* -- den Namen der Quellcodedatei in der sich die Definition der Klasse `Act` befindet -- einleitet.

16. Element des constant_pool
(Zeichenkette "()V")



Konstante vom Typ `CONSTANT_Utf8`, die den String *()V* einleitet. Methodendeskriptor, der weder Übergabeargumente noch Rückgabetyp besitzt.

Zugriffsflags



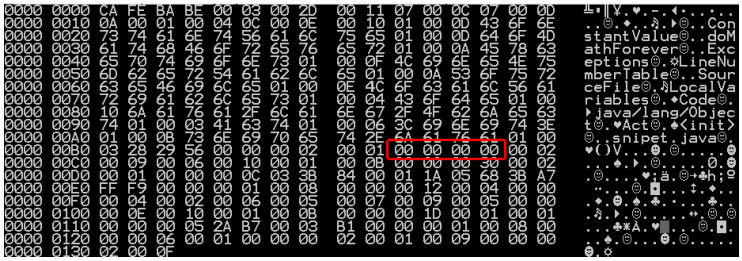
Zugriffsflags für die Klasse `Act`. Der konkrete Code ergibt sich aus der binären ODER-Verknüpfung verschiedener Zugriffsflaggen, die in untenstehender Tabelle wiedergegeben sind.

Flag	Wert	Beschreibung
ACC_PUBLIC	0x0001	`public`-Deklaration; Zugriffbar von allen anderen Klassen, auch außerhalb des eigenen Pakets
ACC_FINAL	0x0010	`final`-Deklaration; Verbot der Vererbung
ACC_SUPER	0x0020	Besondere Behandlung der Superklasseninstruktionen bei Aufruf über Opcode `invokespecial`
ACC_INTERFACE	0x0200	Struktur ist Schnittstelle, keine Klasse
ACC_ABSTRACT	0x0400	Abstrakte Struktur, von der keine Ausprägungen erzeugt werden können

Referenz auf this und super

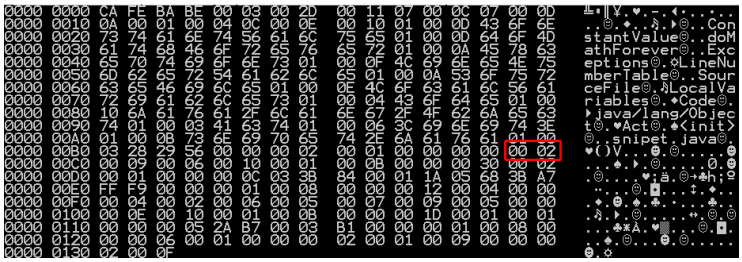
Referenz in den Konstantenpool, auf die Klasse selbst (`this`) und die Superklasse (`super`).

interface_count und
fields_count



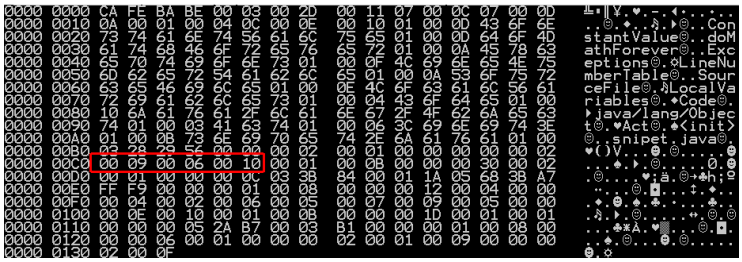
`interface_count`: Anzahl der durch die Klasse implementierten Schnittstellen.
`fields_count`: Anzahl der Klassen- oder Instanzvariablen der Klasse.

methods_count



Anzahl der durch die Klasse implementierten Methoden.
Die Zahl ergibt sich aus der tatsächlich durch den Anwender definierten Anzahl und den impliziten, d.h. durch den Compiler hinzugefügten (z.B. Konstruktor), Methoden.

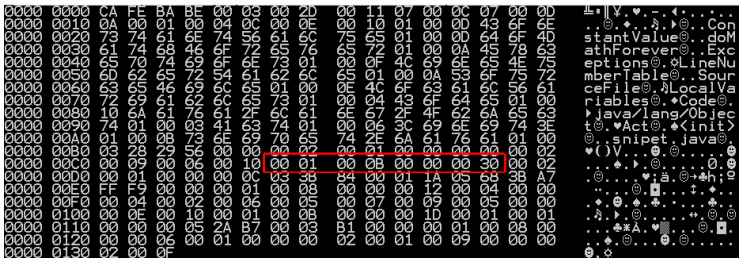
methods_info
(doMathForever())



Informationen über die in der Klasse `Act` definierten Methoden.
Die Zugriffsflaggen (0x00 09) weisen `doMathForever()` als `static` und `public` aus. (Die konkrete Wertebelegung kann untenstehender Aufstellung entnommen werden. Auch in diesem Falle wird der tatsächliche Wert durch Boole'sche ODER-Verknüpfung der Einzelwerte gebildet.)
Der Verweis (0x06) auf das sechste Element des Konstantenpools referenziert den Methodennamen im Klartext. Der zweite Verweis (0x10) auf den Konstantenpool kennzeichnet `doMathForever()` als parameterlose Methode ohne Rückgabetypen.

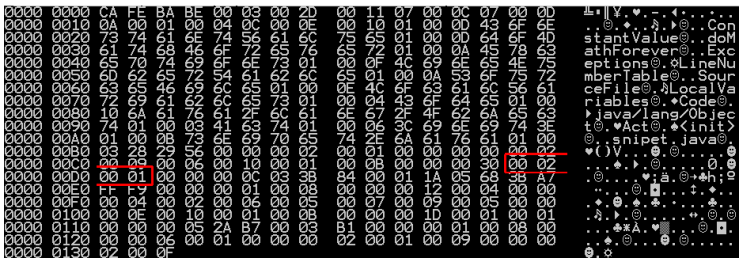
Flag	Wert	Beschreibung
ACC_PUBLIC	0x0001	`public`-Deklaration; Zugriffbar von allen anderen Klassen, auch außerhalb des eigenen Pakets
ACC_PRIVATE	0x0002	Als `private` deklariert, daher nur innerhalb der definierenden Klasse verwendbar
ACC_PROTECTED	0x0004	`protected`-Deklaration, Zugriff nur in Subklassen möglich
ACC_STATIC	0x0008	`static`-Deklaration, keine Auswirkungen auf Sichtbarkeit und Zugriffsrechte
ACC_FINAL	0x0010	`final`-Deklaration; nach initialer Zuweisung keine Wertänderung möglich
ACC_VOLATILE	0x0040	`volatile`-Deklaration; keine Berücksichtigung in Optimierungsmaßnahmen
ACC_TRANSIENT	0x0080	`transient`-Deklaration; keine Berücksichtigung durch Perisistenzmanager

Attribute der Methode doMathForever()



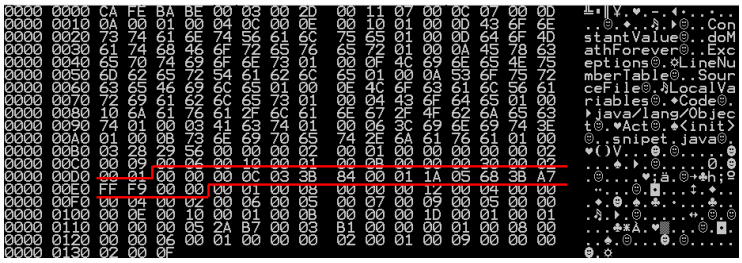
Die Methode `doMathForever()` verfügt nur über genau ein Attribut, daher ist der `attribute count` zu Beginn der Bytesequenz auf 0x00 01 gesetzt. Dieses eine Attribut wird durch Index 11 innerhalb des Konstantenpools referenziert. Dort ist die Zeichenkette `Code` lokalisiert. Dadurch wird angezeigt, daß die folgenden Bytes die Implementierung dieser Methode beinhalten.
Der abschließende vier-Byte Indikator enthält die Länge der Methodenimplementierung (im Beispiel: 0x30).

maxStack und maxLocals der Methode doMathForever()



`maxStack`: Maximalhöhe des Operandenstacks die während der Methodenausführung erreicht werden kann. (Im Beispiel: 2; die Opcode-Implementierung der beiden verwendeten arithmetischen Operationen benötigen niemals mehr als zwei Stackpositionen.)
`maxLocals`: Anzahl der lokalen Variablen. (die verwendete Variable `i`)

Bytecode und Ausnahmentablle der Methode doMathForever()



Die `code length` legt die Anzahl der folgenden Bytecode-Instruktionen fest (im Beispiel: 12), darauf folgen die tatsächlichen Opcodes.

```
pc    instruction    mnemonic
0     03             iconst_0
1     3B             istore_0
2     840001         iinc 0 1
5     1A             iload_0
6     05             iconst_2
7     68             imul
8     3B             istore_0
9     A7FFF9         goto 2
```

Die Ausnahmentabelle (`Exception Table`) enthält die Anzahl der durch die Methode aufgefangenen Ausnahmeereignisse; im Beispiel: 0.

Eigenschaften des Code-Bereichs der Methode doMathForever()

In diesem Bereich werden zusätzliche Charakteristika des bereits definierten Codebereichs hinterlegt, z.B. Debugginginformation.
Im betrachteten Falle ist nur eine Eigenschaft angegeben (`attribute_count = 0x01`). Diese referenziert das achte Element des Konstantenpools -- die Zeichenkette LineNumberTable. Die beiden abschließenden Attribute bezeichnen die Länge dieser Tabelle (0x12) und die Anzahl der Einträge (0x4).
Die LineNumberTable des Beispiels:

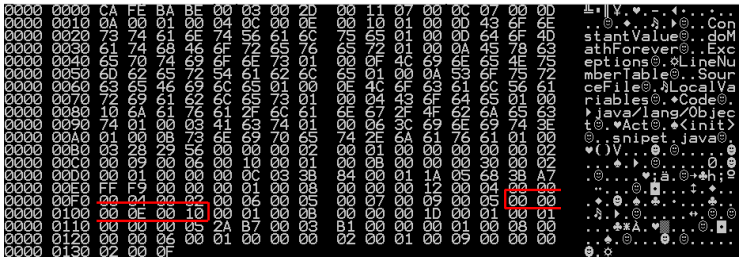
```
line 4: i = 0;
line 5: while(true) {
line 6: i += 1;
line 7: i *= 2;
```

Zuordnung zwischen LineNumberTable und der Methode doMathForever()

Diese Datenstruktur stellt die Zuordnung zwischen den Quellcodezeilen und den resultierenden Opcodes her.

```
LineNumberTable[0]:  iconst_0    istore_0
LineNumberTable[1]:  iinc 0 1
LineNumberTable[2]:  iload_0     iconst_2    imul  istore_0
LineNumberTable[3]:  goto 2
```

Zugriffsflaggen und Indizes der Klasse Act()



Diese zweite methodenbezogene Struktur gibt Auskunft über den Konstruktor der Klasse `Act`.
Im ersten Doppelbyte sind die Zugriffsrechte spezifiziert; in diesem Falle sind keine gesonderten Festlegungen getroffen -- es handelt sich um eine *einfache* Methode.
Die Referenz in den Konstantenpool verweist auf die implementierende Methode (im Beispiel: Position 0x0E, dort findet sich die Methode `<init>`).
Durch die letzten beiden Bytes wird der Typ des Konstruktors referenziert, im betrachteten Beispiel die Position 0x10 im Konstantenpool, mithin ein parameterloser Konstruktor.

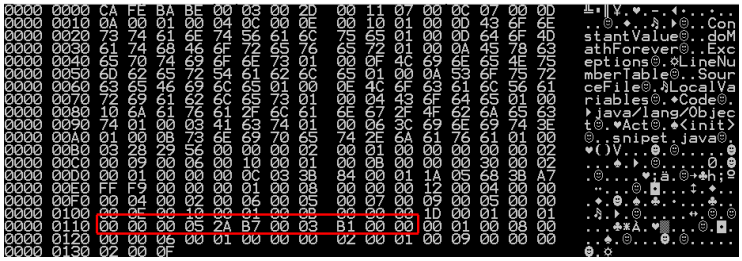
Attribute der Klasse Act()

Der Zähler (ersten beiden Bytes) zu beginn der Struktur zeigt an, daß nur ein Attribut der Klasse `Act()` folgt. Im Beispielfall handelt es sich dabei um das über den Index 11 (0x0B) angesprochene Element des Konstantenpools, die Zeichenkette `Code`.
Der abschließend angegebene Längenzähler fixiert die Anzahl der folgenden Bytes.

maxStack und maxLocals der Klasse Act()

Analog der Definition für Methoden, die maximale Höhe des Operandenstacks und die Anzahl der lokalen Variablen.

Bytecode und Ausnahmetabelle der Klasse Act()



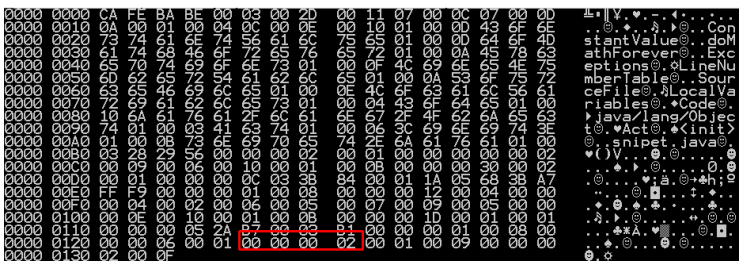
Opcode-Implementierung des Konstruktors, sowie die Aufzählung der durch ihn potentiell ausgelösten Ausnahmeereignisse (im keine, daher Anzahl gleich Null).
Die Implementierung in Java-Bytecode:

```
pc    instruction    mnemonic
0     2A             aload_0
1     B70003         invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4     B1             return
```

Eigenschaften des Code-Bereichs der Klasse Act()

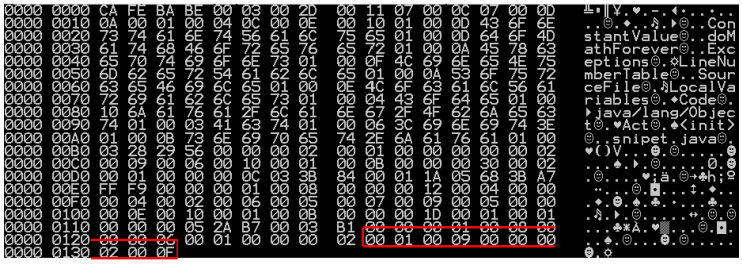
Anzahl der Eigenschaften im ersten Doppelbyte (im Beispiel: 1). Die spezifische Eigenschaft wird durch Index acht im Konstantenpool (=`LineNumberTable`) näher definiert.
Diese Tabelle hat die Länge 0x06, mit einem einzigen Eintrag.

Zuordnung zwischen LineNumberTable und der Klasse Act()



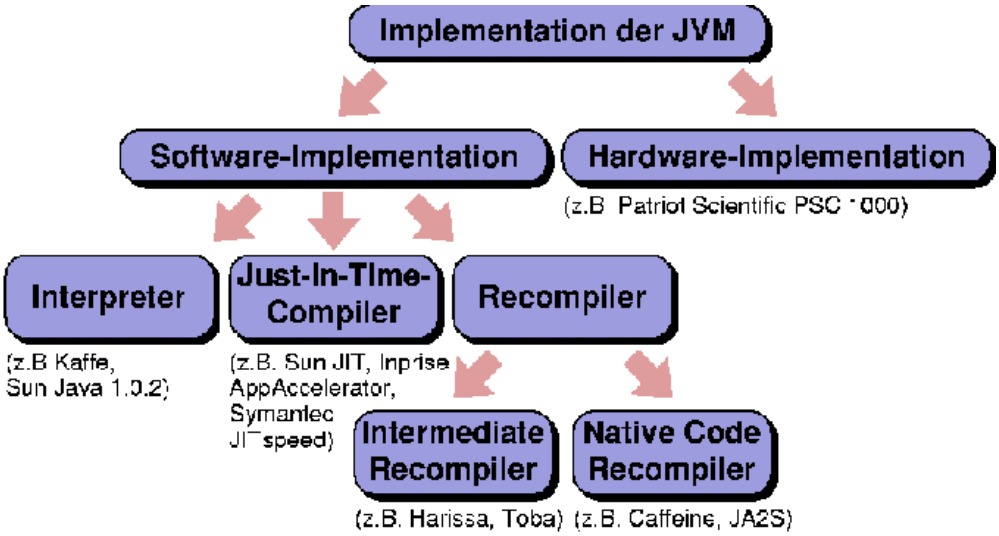
Zuordnung der Quellcodezeilennummern zu den resultierenden Opcodes.

Allgemeine Eigenschaften



Am Ende einer Klassendatei kann eine beliebige Menge allgemeiner Attribute angeben werden.
für das Beispiel wurde durch den Compiler genau ein Attribut erzeugt, ein Verweis auf die Quelldatei (Konstantenpool-Index 0x09). Auf diese Information folgt ein Verweis auf den Namen der Quellcodedatei (Konstantenpool-Index 0x15).

Schlußbemerkungen:



Graphik aus: Raner, M.: Blick unter die Motorehaube

Die interpretative Ausführung einer `class`-Datei mit der virtuellen Java-Maschine ist jedoch keineswegs zwingend, auch wenn sie das derzeit am häufigsten anzutreffende Vorgehen verkörpert. Bereits in der Standardedition des Java Development Toolkits von SUN wird seit Version 1.2 ein *just in time compiler* mitgeliefert, der transparent in die Ausführungsumgebung integriert ist. Er durchbricht die befehlweise interpretative Abarbeitung und greift den bei der Abarbeitung dynamisch durch den Interpretationsprozeß entstehenden plattformspezifischen Maschinencode ab und puffert ihn zwischen. Bei jeder erneuten Ausführung derselben Bytecodesequenz wird nun dieser bereits übersetzte Code ausgeführt. Dieses Vorgehen ist in den verbreiteten JVM-Implementierungen der Internet-Browser von Netscape und Microsoft verwirklicht. Ebenso bieten fast alle verfügbaren Java-Entwicklungsumgebungen diese Laufzeit-optimierende Komponente an. Der dadurch erzielte Geschwindigkeitsvorteil bewegt sich, je nach Struktur der Anwendung, zwischen zehn und 50 Prozent.
Den größten Gewschwindigkeitszuwachs verspricht man sich jedoch von der vollständigen Realisierung der virtuellen Maschine in Hardware; damit wird sie de facto zur realen Maschine. Hierzu liegen jedoch noch keine Ergebnisse vor, welche die der derzeitigen Implementierung auf handelsüblichen Prozessoren signifikant überträfen.
Eine andere Sichweise nutzt das Bytecodeformat welches eine Zwischenrepräsentation darstellt nicht zur Interpretation mit dem Ziele der direkten Ausführung, sondern als Eingabeformat eines weiteren Übersetzungsschrittes, der üblicherweise plattformabhängigen nativen Code erzeugt. Für C++ existieren bereits Umsetzungen, die Bytecode in übersetzungsfähigen C++-Quellcode transformieren. Eine Spielart hiervon bildet das diskutierte Werkzeug `javap` dessen Ausgabeformat, nach einigen Umformumgen, direkt als Eingabe weiterer Übersetzer akzeptiert wird.

Web-Referenzen 4: Weiterführende Links

- Inside the Java Virtual Machine

---

3 Betriebssysteme

Mit der zunehmenden Verlagerung der Steuerungsaufgaben -- wie Bandwechsel, Auswahl des nächsten zu berechnenden Aufgabe sowie Überwachtung des Maschinenstatus -- eines Rechners weg vom (menschlichen) Operateur hin zu dafür entwickelten Softwaresystemen entstand bereits in der Ära der IBM 7094 der Begriff des *Betriebssystem*s für Programme der genannten Funktionalität. (Vgl. [Cer03, S. 105])

3.1 Ablaufsteuerung

Eine der zentralen Aufgaben jedes Betriebssystems ist die Organisation und Verwaltung der aus einem Rechnersystem ausgeführten Programme. Die zuständige Betriebssystemkomponente wird daher zumeist mit *Ablaufsteuerung* bezeichnet.
Die Ablaufsteuerung organisiert die auszuführenden Aufgaben in Prozessen und organisiert deren Zuordnung zur CPU um Berechnungsergebnisse zu erhalten.

Prozesse und Threads

Aktuelle Rechnersysteme erlauben es verschiedene Aufgaben (scheinbar) zu einem Zeitpunkt auszuführen. So können gleichzeitig Dateien auf die Festplatte geschrieben werden und mit verschiedenen Applikationen (etwa Textverarbeitung, Web-Browser und MP3-Player) gearbeitet werden.

Voraussetzung dieses Verhaltens ist die Übertragung eines auf einem Speichermedium (Festplatte, CD-ROM, DVD oder Internet-Server) zur Verfügung stehenden Programms in den Hauptspeicher der Maschine und dessen Ausführung durch den (oder die) Prozessor(en).

Das Betriebssystem betrachtet alle auszuführenden Programme und Aufgaben einheitlich als *Prozeß* (oder *Task*), der über bestimmte festgelegte Eigenschaften verfügt und im selben Schema wie alle anderen Prozesse behandelt wird.

Gemäß der Anzahl der gleichzeitig verarbeitbaren Prozesse wird zwischen *Single Tasking*-Systemen, welche zu einem gegebenen Zeitpunkt nur genau einen einzigen Prozeß abarbeiten können, und *Multitasking* unterschieden. Letztere setzen durch schnelle Umschaltung zwischen den Einzelprozessen die sog. *Multiprogrammierung* um.

Solange das ausführende System jedoch nur eine gleichzeitig aktive Verarbeitungseinheit bietet, ist ausschließlichlich *nebenläufige Verarbeitung* (auch: *Pseudoparallelität*) realisierbar, welche durch schnelle Prozeßumschaltungen den Eindruck der simultanen Abarbeitung verschiedener Prozesse erweckt.
Reale Parallelität ist ausschließlich durch geeignete Hardwareunterstützung, d.h. durch die Bereitstellung duplizierter Funktionseinheiten erzielbar, die zu einem Zeitpunkt mehr als einen Prozeß ausführen.

Prozeßerzeugung
Zur Erzeugung eines neuen Prozesses müssen die auszuführenden Instruktionen (d.h. das Programm) sowie die zur Programmverarbeitung benötigten Hauptspeicherbereiche (Stack und Heap) verfügbar sein. Programmdaten und darauf operierender Code bilden zusammengenommen den *Prozeßkontext*.
Zusätzlich müssen betriebssystemseitig benötigte Verwaltungsdaten zur Kontrolle verschiedener Aspekte der laufenden Prozesse bereitgestellt werden. Dieser Datenbereich wird mit dem Sammelbegriff *Prozeßkontrollblock* oder auch *Prozeßtabelleneintrag* bezeichnet.

Jeder Eintrag in der Prozeßtabelle enthält diejenigen Daten, die benötigt werden um einen Prozeß verwalten können. Hierzu zählen insbesondere alle Angaben über den Zustand des Prozesses, die benötigt werden um ihn nach einem Anhalten (etwa durch Taskumschaltung) wieder lauffähig restaurieren zu können.
Daher umfaßt der Eintrag in der Prozeßtabelle alle prozessorspezifischen Daten, wie Registerbelegungen sowie den aktuellen Stand des Befehls- und Stackzeigerts. Zusätzlich werden Daten über alle geöffneten Dateien sowie weitere betriebssystemspezifische Verwaltungsdaten (wie Benutzer- und Prozeß-ID) gespeichert.

Nach [Tan02, S. 95] umfaßt jeder Eintrag der Prozeßtabelle typischerweise die nachfolgend zusammenstellten Eigenschaften.

Prozeßverwaltung	Speicherverwaltung	Dateiverwaltung
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Stacksegment	Dateideskriptor
Stackzeiger		Benutzer-ID
Prozeßzustand		Gruppen-ID
Priorität		
Scheduling-Parameter		
Prozeß-ID		
Elternprozeß		
Prozeßfamilie		
Signale		
Startzeit des Prozesses		
Verbrauchte CPU-Zeit		
CPU-Zeit der Kinder		
Zeitpunkt des nächsten Alarms		

Zur Erzeugung von Prozessen durch den Anwender stellen die Betriebssysteme entsprechende API-Funktionen zur

Verfügung, welche den auszuführenden Code in den Speicher laden und das Betriebssystem zur korrekten Initialisierung des Prozeßkontrollblockes veranlassen.

UNIX-artige Betriebssysteme wickeln die Prozeßerzeugung über die Funktion fork ab. Wird diese in einem Programm aufgerufen, so wird automatisch ein neuer Prozeß -- der *Kindprozeß* -- erzeugt. Dieser wird zunächst als Kopie des aktuell laufenden -- des *Elternprozeß* -- (d.h. desjenigen der fork aufruft) im Speicher angelegt, welche zwar über eigene lokale Variablen, sowie einen eigenen Stack verfügt, aber Zugriff auf dieselben Dateien besitzt wie der Elternprozeß.
Die Funktion fork regelt dabei über ihren Rückgabewert welche der beiden identischen Kopien als Eltern- und welche als Kindprozeß ausgeführt wird. So wird im Elternprozeß die Prozeßnummer des Kindprozesses zurückgeliefert, während die Funktionskopie des Kindprozesses als Rückgabewert 0 liefert.
Zunächst unterscheiden sich beide Prozeßinstanzen ausschließlich durch die Belegung des Rückgabewertes von fork.

Beispiel 10: Prozeßerzeugung unter UNIX

```
(1)int main (int argc, char** argv) {
(2)     printf("parent\n");
(3)     int i, pid;
(4)
(5)     if ((pid=fork()) == -1) {
(6)             printf("error creating process\n");
(7)             exit(1);
(8)     }
(9)
(10)    if (pid == 0) {
(11)            for (i=0; i<10; i++) {
(12)                    sleep(1);
(13)                    printf("child %d\n",i);
(14)            }
(15)    }
(16)    if (pid > 0) {
(17)            printf("child's pid=%d\n",pid);
(18)            for (i=0; i<10; i++) {
(19)                    sleep(1);
(20)                    printf("parent again %d\n",i);
(21)            }
(22)    }
(23)    return 0;
(24)}
```

Download des Beispiels
Download der Ergebnisdatei

Beispiel 10 zeigt die Anwendung der Systemfunktion fork.
Die lokalen Variablen i und pid stehen nach dem Aufruf von fork sowohl im Eltern- als auch im Kindprozeß zur Verfügung, jedoch verweisen sie auf unterschiedliche Speicherplätze um beiden Prozessen die unabhängige Belegung zu ermöglichen.
Insbesondere besitzt pid nach Aufruf und Duplikation der Codebereiche der beiden Prozesse im Elternprozeß den Wert der von 0 verschiedenen Prozeßnummer des Kindprozesses und setzt daher die Ausführung normal fort.
Im Kindprozeß hingegen ist pid mit 0 belegt und setzt daher mit der Abarbeitung in Zeile 11 fort und wird nach Abarbeitung der Schleife die ausschließlich im Elternprozeß ausgeführte Codesequenz der Zeilen 16 bis 22 überspringen.
Schlägt die Prozeßerzeugung fehlt, so wird im erzeugenden Prozeß die Prozeßnummer -1 zurückgegeben; Ein Kindprozeß existiert dann naturgemäß nicht.

Solange der erzeugende Elternprozeß im System existiert, ist jeder Kindprozeß diesem eindeutig zugeordnet. Hierduch entsteht eine hierarchische Struktur aller Prozesse, die sich ausgehend vom Startprozeß init aufspannt. Dieser Startprozeß nimmt eine Sonderrolle im System ein, da er als erster erzeugter Prozeß immer mit der feststehenden Prozeßnummer 1 versehen ist. Zusätzlich wird er durch die Abwesenheit eines Elternprozesses charakterisiert. Aus diesem Grunde ist init der nicht existente Elternprozeß mit der Prozeßnummer 0 zugeordnet.

Terminiert der Elternprozeß im Verlauf der Programmausführung vor dem Kindprozeß, so wird die Elternprozeßnummer im Kind auf die des init-Prozesses gesetzt. Kindprozesse, deren Elternprozeß vor ihnen terminiert, werden als *Waisen* (engl. *Orphan*) bezeichnet, die von init„adoptiert“ werden.

Diese Vorgehensweise wurde gewählt, da der erzeugende Elternprozeß über den Terminierungsstatus des Kindprozesses informiert wird. Dieser wird als int-Wert durch die im Kindprozeß aufgerufene Funktion exit übermittelt. Hierbei gilt unter UNIX die Besonderheit, daß die Binärrepräsentation des tatsächlich zurückgegebenen Wertes durch logisches Und mit 377 verknüpft wird.

Beendet der Kindprozeß seine Ausführung, ohne das der Elternprozeß den Rückgabewert übernimmt, so verbleibt der Kindprozeß als sog. *Zombie* bis zur Terminierung des Elternprozesses im Hauptspeicher.
Daher sollte ein erzeugender Prozeß unbedingt wait oder waitpid aufrufen um den Terminierungsstatus des Kindprozesses abzufragen und diesem somit die vollständige Terminierung zu ermöglichen.

Im Windows-System stehen die Standard-API-Funktionen WaitForSingleObject zum Warten auf das terminieren des erzeugten Kindprozesses bzw. GetExitCodeProcess zur Abfrage des Rückgabewertes zur Verfügung. Anders als unter UNIX retourniert Windows den numerischen Terminierungsstatus unmodifiziert an den erzeugenden Prozeß.

Oftmals ist es jedoch nicht gewünscht den vollständigen Code des Kindprozesses auch ungenutzt im Elternprozes zur

Verfügung zu stellen, und gleichzeitig umgekehrt im Kindprozess den dort ungenutzen Code des Elternprozesses, den dieser nach der Abspaltung mittels `fork` ausführt, vorzuhalten.
Daher tritt ein Aufruf von `fork` zumeist in Gesellschaft mit einem Aufruf einer Funktion aus der exec-Familie auf.

Die Vertreter (typischerweise existieren mindestens fünf spezialisierte Varianten des exec-Aufrufs) dieser Funktionsgruppe ersetzen den gegenwärtig laufenden Prozeß vollständig durch einen anderen. Dessen Code muß zum Aufrufzeitpunkt noch nicht speicherresident sein, sondern kann durch die exec-Funktion nachgeladen werden.
Beispiel 11 zeigt die Verwendung der Funktion `execlp` im Kindprozeß. Dessen Code wird durch den Funktionsaufruf in Zeile 14 vollständig durch den des Beispiels 12 überlagert.

Der Überlagerungsprozeß überschreibt den Prozeßkontext vollständig, so daß alle vom Elternprozeß duplizierten Daten und das aktuell ausgeführte Programm danach nicht mehr im Hauptspeicher zugreifbar sind. Aus diesem Grund wird typischerweise nicht der Aufruf `fork` zur Prozeßerzeugung, sondern `vfork` eingesetzt sofern direkt auf den `fork`-Vorgang ein exec-Aufruf folgt. Die `vfork`-Funktion besitzt den Vorteil, daß sie keine Duplikation der Datenbereiche des Elternprozesses vornimmt und daher deutlich schneller abgearbeitet werden kann als `fork`. In der Konsequenz darf der erzeugte Kindprozeß keine Daten verändern, da diese physisch mit dem Elternprozeß geteilt werden.

Beispiel 11: Prozeßerzeugung und Überlagerung mittels exec unter UNIX

```
(1)#include <stdlib.h>
(2)
(3)int main (int argc, char** argv) {
(4)     printf("parent\n");
(5)     int i, pid;
(6)     int execResult;
(7)
(8)     if ((pid=vfork()) == -1) {
(9)             printf("error creating process\n");
(10)            exit(1);
(11)    }
(12)
(13)    if (pid == 0) {
(14)            execResult = execlp("./child.exe", "child", NULL);
(15)            //only gets here in case of an error
(16)            printf("execlp returned: %d\n",execResult);
(17)    }
(18)    if (pid > 0) {
(19)            printf("child's pid=%d\n",pid);
(20)            for (i=0; i<10; i++) {
(21)                    sleep(1);
(22)                    printf("parent again %d\n",i);
(23)            }
(24)    }
(25)    return 0;
(26)}
```

Download des Beispiels
Download der Ergebnisdatei

Der unabhängig übersetzte Code, welcher durch den Kindprozeß nachgeladen wird ist in Beispiel 12 dargestellt:

Beispiel 12: Nachgeladener Code des Kindprozesses

```
(1)int main(int argc, char** argv) {
(2)     int i;
(3)
(4)     for (i=0; i<10; i++) {
(5)             printf("child %d\n",i);
(6)             sleep(1);
(7)     }
(8)}
```

Download des Beispiels

Zur Beschleunigung des `fork`-Aufrufes setzen Linux-Systeme die sog. *Copy-on-Write*-Technik ein. Hierbei werden die für den Kindprozeß bereitzustellenden Datenressourcen nicht direkt bei der Ausführung von `fork` dupliziert, sondern solange auf die Daten des Elternprozesses lesend zugegriffen, bis der Kind- oder Elternprozeß diese zu verändern wünscht. Erst dann (im Falle des schreibenden Zugriffes) müssen die Daten für den Kindprozeß dupliziert und separat bereitgestellt werden.
Hintergrund dieser Optimierungstechnik, die zu deutlichen Geschwindigkeitsgewinnen führen kann, ist die Erkenntnis, daß Kindprozesse nur in seltenen Fällen alle vom Elternprozeß übernommenen Daten verändern. In der überwiegenden Mehrzahl werden die duplizierten Daten hingegen ausschließlich lesend verarbeitet und nur ein kleiner Teil modifiziert.

Die Windows-Betriebssystemfamilie nimmt die Unterscheidung in Kindprozeßerzeugung und separatem Nachladen des benötigen Codes nicht vor, sondern bündelt beide Vorgänge in der Systemfunktion `CreateProcess`.
Beispiel 13 zeigt den Aufruf der genannten Systemfunktion, welche den in Beispiel 14 dargestellten Code als Kindprozeß lädt.

Beispiel 13: Prozeßerzeugung unter Windows

```
(1)#include <Windows.h>
(2)
(3)int main( VOID ) {
(4)   STARTUPINFO si;
(5)   PROCESS_INFORMATION pi;
(6)   LPDWORD result;
(7)   int i;
(8)
(9)   ZeroMemory( &si, sizeof(si) );
(10)   si.cb = sizeof(si);
(11)   ZeroMemory( &pi, sizeof(pi) );
(12)
(13)   // Start the child process.
(14)   if( !CreateProcess( NULL, // No module name (use command line).
(15)       "childWin.exe", // Command line.
(16)       NULL,             // Process handle not inheritable.
(17)       NULL,             // Thread handle not inheritable.
(18)       FALSE,            // Set handle inheritance to FALSE.
(19)       0,                // No creation flags.
(20)       NULL,             // Use parent's environment block.
(21)       NULL,             // Use parent's starting directory.
(22)       &si,              // Pointer to STARTUPINFO structure.
(23)       &pi )             // Pointer to PROCESS_INFORMATION structure.
(24)   ) {
(25)       return 1;
(26)   }
(27)    for (i=0; i<10;i++) {
(28)            printf("parent %d\n",i);
(29)            Sleep(1000);
(30)    }
(31)    //wait for child to exit
(32)    WaitForSingleObject(pi.hProcess, INFINITE);
(33)    GetExitCodeProcess(pi.hProcess, result);
(34)    printf("child's exit code was: %i\n", *result);
(35)
(36)   return 0;
(37)}
```

Download des Beispiels
Download der Ergebnisdatei

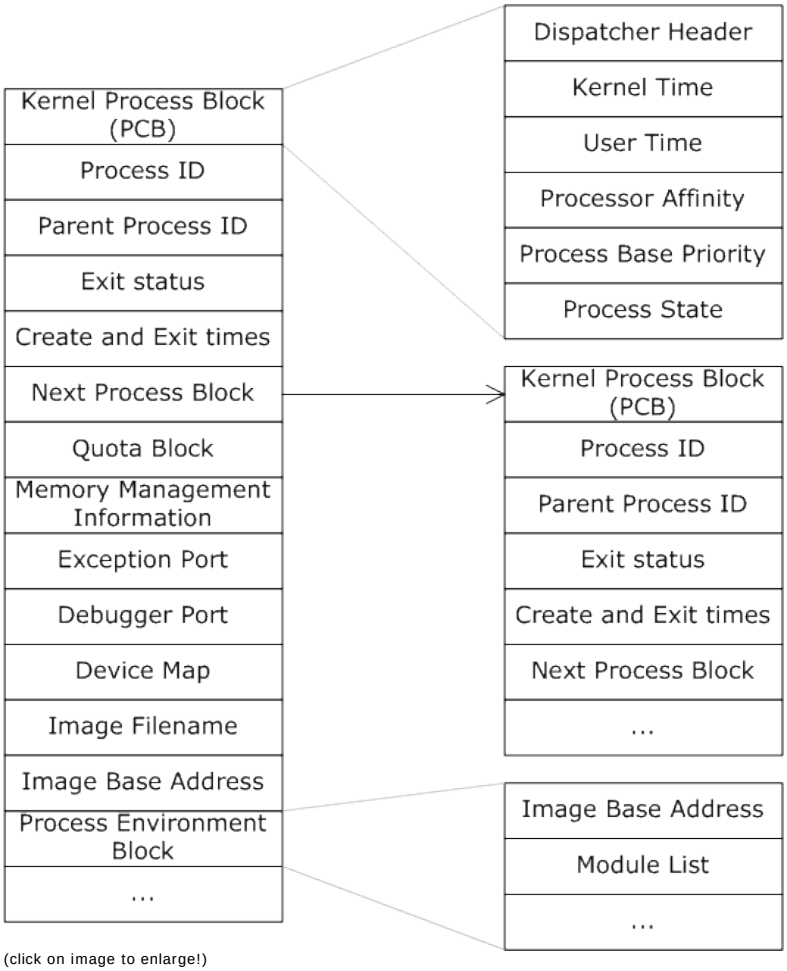
Beispiel 14: Quellcode des Kindprozesses

```
(1)#include <Windows.h>
(2)
(3)int main(int argc, char** argv) {
(4)     int i;
(5)     for (i=0; i<10; i++) {
(6)             printf("child %d\n",i);
(7)             Sleep(1000);
(8)     }
(9)     exit(2);
(10)}
```

Download des Beispiels

Die Struktur des Prozeßkontrollblocks (PCB), der auch als *Kernel Process Block* bezeichnet wird, ist in Abbildung 16 dargestellt.

Abbildung 16: *Aufbau des Windows-Prozeßkontrollblocks*



(click on image to enlarge!)

Grundsätzlich enthält auch der unter Windows benutzte Prozeßkontrollblock die im vorhergehenden dargestellten Eigenschaften. Im Detail (beispielsweise an der Speicherung der geladenen DLL-Bibliotheken) lassen sich jedoch die Plattformbesonderheiten gut erkennen. (Vgl. hierzu auch: [Sol00, 279 u. 291])

Element	Beschreibung
Dispatcher Header	Verwaltungsinformation der Taskumschaltung
Kernel Time	Verbrauchte priviligierte Rechenzeit
User Time	Verbrauchte Anwenderrechenzeit
Processor Affinity	Zuweisung an genau eine CPU eines Multiprozessorsystems
Process Base Priority	Prozeßpriorität
Process State	Gegenwärtiger Prozeßzustand
Process ID	Eineindeutige Prozeßnummer
Parent Process ID	Prozeßnummer des Elternprozesses
Exit status	Abbruchgrund
Create and Exit times	Erzeugungs- und Terminierungszeitpunkt
Next Process Block	Verweis auf den nächsten Eintrag der Prozeßtabelle
Quota Block	Speichernutzungsbeschränkung
Memmory Management Information	Verschiedene Aussagen über Nutzung des virtuellen Speichers
Exception Port	Kanal der Interprozeßkommunikation. Prozeßmanager sendet, bei Auftreten eines Ausnahmeereignisses, Nachricht dorthin.
Debugger Port	Kanal der Interprozeßkommunikation. Prozeßmanager sendet, bei Auftreten eines Debug-Ereignisses, Nachricht dorthin.
Device Map	Zuordnung der logischen Gerätenamen zur jeweiligen Hardware
Image Filename	Name der ausführbaren Binärdatei
Image Base Address	Startadresse der speicherresidenten ausführbaren Binärdatei
Module List	Verzeichnis der geladenen dynamischen Module (DLL-Dateien)

Die Tabelle zeigt im Kern wesentliche Gemeinsamkeiten mit der Auflistung der Prozeßkontrolfelder des UNIX-Systems

auf. So werden auch unter Windows alle im System verwalteten Prozeße einheitlich durch eine numerische Prozeßnummer identifiert. Ebenso besitzt jeder Prozeß (mit Ausnahme des Startprozesses `init` unter UNIX bzw. `Idle` unter Windows) einen eindeutigen Elternprozeß.
Zusätzlich zeigt die Tabelle einige systemspezifische Besonderheiten, wie beispielsweise die Verwaltung der geladenen DLL-Bibliotheksdateien.

Neben der direkten Möglichkeit Prozesse durch Betriebssystemfunktionen zur erzeugen, bieten hierfür auch Hochsprachenunterstützung an, wenngleich diese im engeren Sinne selbst keine Prozeßerzeugung vornehmen. Typischerweise bieten Hochsprachen lediglich einfach benutzbare Programmierschnittstellen als die nativen des Betriebssystems zur Prozeßerzeugung an. Die Aufrufe dieser Schnittstellen müssen jedoch zur Laufzeit auf die betriebssystemseitig verfügbaren Schnittstellen abgebildet werden. Dies kann bereits zur Übersetzungszeit durch den Compiler, oder zur Laufzeit durch den Interpreter, geschehen.

Im Falle der Java-Standard-API ist letzterer Ansatz verwirklicht, d.h. die virtuelle Java Maschine interpretiert zur Laufzeit den Hochsprachenaufruf und bildet ihn auf die entsprechende Betriebssystemfunktionalität ab. Innerhalb Javas stellt die Klasse `Runtime` verschiedene Varianten einer exec-Funktion zur Verfügung. Ausprägungen der Klasse `Runtime` fungieren hierbei zur Laufzeit als Schnittstelle zur betriebssystemseitigen Ausführungsumgebung, welche die Java Virtual Machine beherbergt. Daher ist zwar die direkt durch den Programmierer aufgerufene `exec`-Methode ebenso wie die durch sie aufgerufenen Methoden der Klasse `ProcessBuilder`, welche die betriebssystemseitige Prozeßerzeugung kapselt, in Java realisiert, die Implementierung des `ProcessBuilders` durch die nicht als Bestandteil der Standard-API aufgefaßte Klasse `ProcessBuilderImpl` hingegen als plattformspezifische native Methode vorgenommen.

Daher verhält sich die durch exec-Javamethode so, wie die Kombination der UNIX-Systemaufrufe `fork` und exec, da die Javaplattform implizit die Erzeugung eines neuen Prozesses vornimmt und es damit durch exec zu keiner Überlagerung des ausgeführten Programmcodes kommt.
Ähnlich wie die gleichnamige UNIX-Systemfunktion gestatten Javas exec-Varianten ebenfalls den freien Aufruf externer ausführbarer Dateien, wie 15 zeigt.

Beispiel 15: Prozeßerzeugung unter Java

```
(1)import java.io.BufferedInputStream;
(2)import java.io.IOException;
(3)public class JavaExec {
(4)     public static void main(String args[]) {
(5)             Process p = null;
(6)             System.out.println("creating child");
(7)             Runtime rt = Runtime.getRuntime();
(8)             String command[] = new String[2];
(9)             command[0] = "java";
(10)            command[1] = "Child";
(11)            try {
(12)                    p = rt.exec(command);
(13)            } catch (IOException ioe) {
(14)                    System.out.println("IOException occured");
(15)            }
(16)            try {
(17)                    p.waitFor();
(18)            } catch (InterruptedException ie) {
(19)                    System.out.println("InterruptedException occured");
(20)            }
(21)            System.out.println("child has finished execution");
(22)            BufferedInputStream pOut = new BufferedInputStream(p.getInputStream());
(23)            byte output[] = null;
(24)            try {
(25)                    output = new byte[pOut.available()];
(26)                    pOut.read(output);
(27)            } catch (IOException ioe) {
(28)                    System.out.println("IOException occured");
(29)            }
(30)            System.out.println("output of the child process was: "
(31)                            + new String(output));
(32)    }
(33)}
```

Download des Beispiels
Download der Ergebnisdatei

Der Code zeigt den Aufruf eines externen Programmes, am Beispiel der Erzeugung einer zweiten Java-Virtual-Machine-Instanz welche die Klasse `Child` zur Ausführung bringt.
Zur Erzeugung einer zusätzlichen Prozeßinstanz muß zunächst mittels der Methode `getRuntime` dasjenige Objekt ermittelt werden, weches die Verbindung zur betriebssystemseitigen Ausführungsumgebung repräsentiert.
Die auf Objekten dieses Typs ausführbare Methode `exec` gestattet dann die Übergabe einer Kommandozeile an die Betriebssystemumgebung. Im Beispiel wird die Zeichenkette `java Child` (aufgrund des separiertenden Leerzeichens in zwei `String`-Objekte aufgespalten) übergeben.
Der betriebssystemseitig erzeugte Prozeß wird unmittelbar nach seiner Erzeugung automatisch gestartet und abgearbeitet. Die Java-Methode liefert ein Objekt des Typs `Process` zurück, welches innerhalb der Hochsprache zur Kontrolle des erzeugten Prozesses eingesetzt werden kann.
Insbesondere kann auf die Terminierung durch Aufruf der Methode `waitFor` seitens des Erzeugers gewartet werden.

Als Besonderheit der Java-Plattform besitzen die erzeugten Kindprozesse keinen Zugriff auf die Standardkanäle für Standard-Ein- und -Ausgabe sowie Fehlerausgabe. Stattdessen werden alle Ausgaben des Kindprozesses an den erzeugenden Prozeß umgeleitet, von dem auch alle über die Standardeingabe vermittelten Eingaben erwartet werden. Aus diesem Grunde leitet der Elternprozeß die Ausgabe des Kindprozesses (sie ist Eingabe im erzeugenden Prozeß) mittels `getInputStream` um und gibt sie anschließend selbst aus.

Threads

Für viele Aufgaben, welche kurzfristig Nebenläufigkeitstechniken nutzen möchten, stellt der für die Prozeßerzeugung einzuplanende Zeitaufwand eine große Effizienzhürde dar. Insbesondere die Duplizierung des Elternprozeßkontextes nimmt, durch die notwendigen Kopieraufwände zur Bereitstellung der Variableninhalte auch für die Kindprozesse einen großen Zeitaufwand ein.
Darüber hinaus ist oftmals -- wie am Beispiel des Aufrufpaares `fork-exec` bzw. `CreateProcess` gezeigt -- ausschließlich die Bereitstellung zusätzlicher Ausführungsressourcen gewünscht, ohne die Datenressourcen gleichzeitig zu duplizieren.

Daher gilt grundsätzlich folgende Aufteilung (vgl. hierzu: [Tan02, S.98]):

Prozeßlokale Elemente	Threadlokale Elemente
Adressraum	Befehlszähler
Globale Variablen	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Signale	
Signalebehandlungsroutinen	
Abrechnungsdaten	

Aus diesen Gründen haben sich inzwischen separate Ausführungsfäden (sog. *Threads*) als leichtgewichtige Prozesse breit etabliert und den Begriff des *Multithreadings* in Erweiterung des klassischen Multiprogrammings geprägt.
Dieser Prozeßtyp wird innerhalb genau eines Prozeßkontextes zur Ausführung gebracht. Daher kann jeder Thread auf die globalen Variablen des Elternprozesses zugreifen, besitzt jedoch einen eigenen Befehlszähler sowie Speicherplatz für den threadlokalen Stack und seine lokalen Variablen.
Durch die Beschränkung der zur Ausführung benötigten Ressourcen auf das absolute Minimum können Threads sehr schnell erzeugt werden und belasten daher die Gesamtleistungsfähigkeit des Systems nur marginal.

Im UNIX-Betriebssystem Linux steht für C und C++ die *POSIX Thread Library* (kurz: PThreads) zur Verfügung. Ihre durch den 1995 verabschiedeten Standard IEEE 1003.1c festgelegten Funktionen gestattet es dem Programmierer auf Hochsprachenebene eigenständig Threads zu erzeugen. Beispiel 16 zeigt die Umsetzung eines Thread-basierten C-Programmes.

Beispiel 16: Threads unter Linux mit PThreads

```
(1)#include <stdio.h>
(2)#include <pthread.h>
(3)
(4)int counter=0;
(5)
(6)void threadedFunction(void *ptr){
(7)     int i;
(8)     for (i=0; i<10; i++) {
(9)             printf("hello from %s (counter=%d)\n",(char*) ptr,counter++);
(10)            sleep(1);
(11)    }
(12)}
(13)
(14)int main(int argc, char** argv) {
(15)    pthread_t thread1, thread2;
(16)    int  iret1, iret2;
(17)
(18)    iret1 = pthread_create( &thread1, NULL, (void*)&threadedFunction, (void*) "thread 1");
(19)   iret2 = pthread_create( &thread2, NULL, (void*)&threadedFunction, (void*) "thread 2");
(20)
(21)    pthread_join( thread1, NULL);
(22)   pthread_join( thread2, NULL);
(23)
(24)   printf("thread 1 returns: %d\n",iret1);
(25)   printf("thread 2 returns: %d\n",iret2);
(26)   exit(0);
(27)}
(28)
```

Download des Beispiels
Download der Ergebnisdatei

Die zur Umsetzung des Beispiels genutzte Bibliothek gestattet es, Nebenläufigkeit mit Funktionsgranularität zu realisieren. So wird im Beispiel die Funktion `threadedFunction` mittels des Aufrufes `pthread_create` als Programmfaden deklariert und ausgeführt.
Der Aufruf `pthread_join` sorgt dafür, daß mit der Weiterverarbeitung solange angehalten wird, bis der durch den übergebenen Zeiger identifizierte Thread seine Ausführung beendet hat.
Durch Nutzung der global deklarierten Variable `counter` zeigt der Beispielcode zusätzlich den gemeinsamen Zugriff auf Speicherbereiche durch die beiden erzeugten Threads.

Der Geschwindigkeitsvergleich zwischen zwei funktional äquivalenten Umsetzungen (`forkBench.c` und `pthreadBench.c`) zeigt deutlich, die Geschwindigkeitssteigerung beim Einsatz von Threads gegenüber schwergewichtigten herkömmlichen Prozessen.
So benötigt die Erzeugung und Terminierung von zehn Prozessen mittels `fork` 0.3 sec. während die selbe Anzahl Threads in 0.01 sec. angelegt und beendet werden kann. (Meßumgebung: Dual Athlon 1200, Linux 2.6.5)

Innerhalb der Windows-Betriebssystemumgebung wird zur Threaderzeugung die Standard-API-Funktion `CreateThread` angeboten. Sie operiert, wie der zuvor eingeführte POSIX-Aufruf, ebenfalls auf Funktionsebene. Der Code aus Beispiel 18 setzt die aus dem POSIX-Beispiel bekannte Funktionalität unter Nutzung der Windows-API um.

Beispiel 17: Threads unter Windows mit der Standard-API

```
(1)#include <windows.h>
(2)
(3)int counter=0;
(4)
(5)DWORD WINAPI ThreadFunc(LPVOID lpParam) {
(6)     int i;
(7)
(8)     for (i=0;i<10;i++)
(9)             printf("hello from thread %d (counter=%d)\n",*(DWORD*)lpParam, counter++);
(10)   return 0;
(11)}
(12)
(13)int main(VOID) {
(14)    DWORD dwThreadId1, dwThreadId2;
(15)    DWORD dwThrdParam1=1, dwThrdParam2=2;
(16)    HANDLE hThread1, hThread2;
(17)
(18)    hThread1 = CreateThread(
(19)        NULL,                       // default security attributes
(20)        0,                          // use default stack size
(21)        ThreadFunc,                 // thread function
(22)        &dwThrdParam1,              // argument to thread function
(23)        0,                          // use default creation flags
(24)        &dwThreadId1);              // returns the thread identifier
(25)
(26)   hThread2 = CreateThread(NULL,0,ThreadFunc,&dwThrdParam2,0,&dwThreadId2);
(27)
(28)   if (hThread1 == NULL || hThread2 == NULL) {
(29)      printf("CreateThread failed");
(30)   } else {
(31)      CloseHandle(hThread1);
(32)      CloseHandle(hThread2);
(33)   }
(34)    Sleep(100);
(35)   exit(0);
(36)}
```

Download des Beispiels
Download der Ergebnisdatei

Konventionsgemäß kann einer als Thread abzuarbeitenden Funktion in Windows lediglich genau ein Parameter des Typs `LPVOID` übergeben werden. Ein Verweis auf den erzeugten Thread wird innerhalb der Ausführung der Funktion `CreateThread` erzeugt und als Adresse an den Aufrufer zurückgeliefert.

Innerhalb Microsofts .NET-Umgebung vereinfacht sich der Umgang mit Betriebssystem-Threads deutlich. So kapselt, wie in Beispiel 1 dargestellt, die .NET-API durch die Klasse `ThreadPool` die Erzeugung eines Threads vollständig.
Die aus der Standard-API bekannte Einschränkung bei der Erzeugung nur höchstens einen Parameter übergeben zu können, ist jedoch auch in der .NET-Umsetzung unverändert präsent. Beispiel 1 setzt die aus den Beispielen 16 und 18 bekannte Funktionalität unter Verwendung des .NET-Frameworks um.

Beispiel 18: Threads unter Windows mit dem .NET-Framework

```
(1)using System;
(2)using System.Threading;
(3)
(4)class netThread {
(5)     private static int counter=0;
(6)     [STAThread]
(7)     static void Main(string[] args) {
(8)             ThreadPool.QueueUserWorkItem(new WaitCallback(myThread),"thread 1");
(9)             ThreadPool.QueueUserWorkItem(new WaitCallback(myThread),"thread 2");
(10)            Thread.Sleep(15000);
(11)    }
(12)
(13)    static void myThread(object threadName) {
(14)            for (int i=0;i<10;i++) {
(15)                    Console.WriteLine("hello from "+threadName+ " (counter="+counter.ToString()
+")");
(16)                    counter++;
(17)                    Thread.Sleep(1000);
(18)            }
(19)    }
(20)}
(21)
```

Download des Beispiels
Download der Ergebnisdatei

Ähnlich zum herstellerseitig auf die Windows-Betriebssystemfamilie beschränkte .NET-Framework stellt auch die plattformunabhängig verfügbare Java-Umgebung eine Thread-Unterstützung auf Hochsprachenebene zur Verfügung. Ausgangspunkt der Threadunterstützung ist die Standard-API-Schnittstelle `Runnable`.

Sie definiert die Signatur der Operation `run`, welche als Startmethode eines Threads fungiert.

Anders als die Threadrealisierung durch die POSIX-Bibliothek und die in .NET umgesetzte Unterstützung gestattet Java ausschließlich die Bereitstellung von Nebenläufigkeit auf Klassenebene, da syntaktisch nur diese Schnittstellen implementieren können.
Beispiel 19 zeigt die Erzeugung und Ausführung von zwei separaten Threads, welche beide als Ausprägungen der Klasse `MyThread` realisiert sind.

Beispiel 19: Threaderzeugung unter Java

```
(1)public class JavaThreadExample {
(2)     public static int counter = 0;
(3)     public static void main(String[] args) {
(4)             Thread t1 = new Thread(new MyThread("thread 1"));
(5)             Thread t2 = new Thread(new MyThread("thread 2"));
(6)             t1.start();
(7)             t2.start();
(8)             System.out.println("threads started ...");
(9)     }
(10)}
(11)class MyThread implements Runnable {
(12)    protected String text;
(13)    public MyThread(String text) {
(14)            this.text = text;
(15)    }
(16)    public void run() {
(17)            for (int i = 0; i < 10; i++) {
(18)                    System.out.println("hello from " + text + " (counter="
(19)                                    + JavaThreadExample.counter + ")");
(20)                    JavaThreadExample.counter++;
(21)                    try {
(22)                            Thread.sleep(1000);
(23)                    } catch (InterruptedException ie) {
(24)                            System.out.println("An InterruptedException occurred\n"
(25)                                            + ie.toString() + "\n" + ie.getMessage());
(26)                    }
(27)            }
(28)    }
(29)}
```

Download des Beispiels
Download der Ergebnisdatei

Das Beispiel zeigt die Erzeugung zweiter Threads, welche zunächst als gewöhnliche Java-Objekte vom Typ Klasse, welche `Runnable` implementiert, bereitgestellt werden.
Nach der Objekterzeugung muß die Ausführung je Thread explizit durch Aufruf der Methode `start` intiiert werden. Erst der Aufruf dieser Methode veranlaßt die virtuelle Java-Maschine dazu die im Thread-Objekt präsente Methode `run` nebenläufig auszuführen.
Würde diese direkt aufgerufen werden, so erfolgt keine nebenläufige, sondern die herkömmliche synchrone, Ausführung.

Die Ausgabe des Beispiels zeigt gleichzeitig die Grenzen des Threadansatzes auf. Zwar verfügen alle Threads eines Prozeßkontextes über dieselben globalen Variablen, auf die diese unabhängig voneinander lesend und schreibend zugreifen können. Jedoch werden diese Zugriffe prinzipiell unsynchronisiert abgewickelt, wodurch es zu Datenverlust durch Überschreiben vorheriger Ergebnisse kommen kann. Abhilfe schaffen hier die im Abschnitt *Konkurrenz und Synchronisation* vorgestellten Synchronisationsmechanismen.

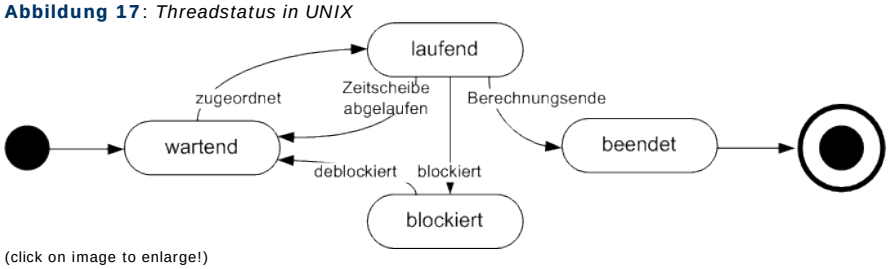
Insgesamt zeigt sich die durch Threads eingeführte Trennung von Ressourcen- und Rechenkapazitätsbündelung als sinnvoll. So kapseln aktuelle Betriebssysteme codeausführrende Einheiten generell durch Threads und konzentrieren die Ressorucenverwaltung im umgebenden Prozeß. Daher besitzt sowohl unter Linux, als auch Windows, jeder Prozeß mindestens einen Thread.

Prozessmodi

Zur Umsetzung der Parallelitätsillusion auf Rechnersystemen mit nur einer Berechnungseinheit ist die schnelle Umschaltung zwischen den rechenbereiten und -willigen Threads zwingend notwendig. Da sich je Berechnungseinheit jedoch zu einem Zeitpunkt nur genau ein Thread in Ausführung befinden kann, und die Anzahl der laufenden Rechenvorgänge in einem System typischerweise größer ist als die Anzahl der zur Verfügung stehenden Hardware-Berechnungsressourcen, müssen einzelne Threads temporär von der Ausführung suspendiert werden, um anderen die Berechnung ihrer Ergebnisse zu ermöglichen. Die Selektion eines rechenbereiten Threads und Zuweisung zur Berechhnungseinheit der Hardware obliegt dem sog. *Task Scheduler*.
Voraussetzung seiner Arbeit ist die Verwaltung von verschiedenen Threadstatus, die es ermöglichen zwischen rechnenden und von der Berechnung suspendierten Threads zu unterscheiden.

Im Betriebssystem UNIX werden generell die vier in Abbildung 17 dargestellten Threadstatus unterschieden:

Abbildung 17: *Threadstatus in UNIX*



(click on image to enlarge!)

- **warten**: Ein Thread in diesem Status ist rechenbereit und ablauffähig. Die zur Verfügung stehenden Rechenenheiten sind jedoch aktuell mit der Verarbeitung anderer Threads beschäftigt; Daher wird der Thread solange im Wartezustand gehalten, bis er einer Recheneinheit zugeteilt wird.
- **laufen**: Ein Thread rechnet aktuell, d.h. er ist genau einer Rechenheit der Hardware zugeteilt und seine Instruktionen werden ausgeführt.
- **blockiert**: Die Ausführung eines Threads wurde unterbrochen weil der Thread auf ein externes Ereignis (z.B. Ein-/Ausgabe) wartet. Dieser Thread kann keiner Recheneinheit zugeteilt werden.
- **ende**: Der Thread hat seine Berechnungsaufgabe abgeschlossen und wird aus dem Hauptspeicher entfernt. Hierbei werden alle ihm zugeteilten Daten- und Speicherbereiche freigegeben.

Die Zustandsübergänge zwischen den einzelnen Status treten jeweils druch verschiedene Bedingungen und Ereignisse ein.
So befindet sich ein neu im System erzeugter Thread zunächst im Zustand *wartend*, da er nach seiner Erzeugung zunächst auf die erstmalige Zuteilung von Berechnungsressourcen durch das Betriebssystem warten muß. Wird ein Thread zur Verarbeitung selektiert, so wird er genau einer Berechnungseinheit *zugeordnet* und sein Code dort zur Ausführung gebracht. Hierdurch erfolgt der Zustandsüberganz von *wartend* zu *laufend*.
Ein Thread verbleibt solange in Ausführung, bis entweder seine zugeteilte Berechnungszeitscheibe abgelaufen ist oder er durch Warten auf ein externes Ereignis (z.B. Bereitstellung von Ein-/Ausgabedaten, Einlagern von Speicherseiten nach Seitenfehler) selbst die Berechnung einstellt. Im Falle des Entzugs der Berechnungssressourcen durch das Betriebssystem nach Ablauf der zugeteilten Rechenzeit geht der (prinzipiell unverändert rechenwillige und -bereite) Thread wieder in den Zustand *wartend* über und verbleibt dort bis zur erneuten Zuteilung einer Berechnungsressource. Suspendiert der Thread seine Ausführung aufgrund einer externen Wartebedingung vor Ablauf der zugeteilten Zeitscheibe so wird der Thread in den Zustand *blockiert* überführt. Dort verbleibt er, bis das externe Ereignis eingetreten ist und kann bis dorthin nicht wieder einer Berechnungsressource zugeordnet werden. Das bedeutet, er wird vorübergehend aus der Liste der rechenbereiten Threads (sie befinden sich alle im Zustand *wartend*) entfernt.
Nach dem Ende der Berechnung geht ein Thread in den Zustand *beendet* über. In diesem Zustand wird sein Ende an seinen zugeordneten Elternprozeß gemeldet und die mit dem Thread verbundenen Daten- und Programmbereiche aus dem Hauptspeicher entfernt. Abschließend werden die zugeordneten Verwaltungsdaten des Betriebssystem und -- sofern es sich um den letzten Thread eines Prozesses handelt -- auch der Prozeßkontrollblock geleert.

In Abbildung 17 unberücksichtigt ist der mögliche Zustand *Zombie*, der eingenommen wird falls ein Thread zwar seine Ausführung beendet, aber sein Terminieren nicht an den Elternprozeß melden kann.

Üblicherweise wird in UNIX- und Windows-Betriebssystem der Zustandsübergang von *laufend* zu *wartend* automatisch zeitscheibengesteuert durch das Betriebssystem vollzogen (sog. *präemptives Multitasking*). In einigen Systemen (darunter die 16-Bit Windowsvarianten) ist der Prozeß selbst für die Rückgabe der Zeitscheibe und damit Kooperation mit anderen rechenwilligen Prozessen verantwortlich. Daher wird dieser Multitaskingtyp auch als *kooperatives Multitaksing* bezeichnet.
Vermöge einer dafür vorgesehenen Betriebssystem-Funktion (bei Windows ist dies der Aufruf `yield`) muß ein Prozeß die Kontrolle explizit an das Betriebssystem übergeben um anderen Prozessen die Ausführung zu ermöglichen.
Durch den Aufruf pause kann ein UNIX-Thread sich selbst von der Ausführung suspendieren, bis er durch ein Signal

geweckt wird. Er wird daher nach dem Auruf von pause nicht in den Zustand `wartend` überführt, sondern verbleibt in `blockiert` bis er durch ein Signal (=externes Ereignis) geweckt wird.

Der Code aus Beispiel 20 illustriert dies am Beispiel der Wartebedingung auf das Eintreffen des Signals `SIGUSR1`. Erst wenn dieses durch die Behandlungsroutine „`catcher`" verarbeitet wurde, wird die Ausführung unmittelbar mit der auf die pause-Anweisung folgenden Instruktion fortgesetzt.

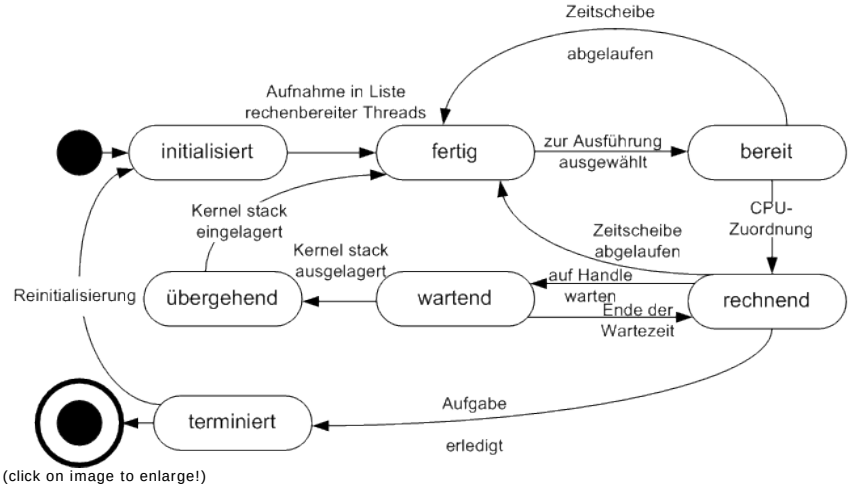
Beispiel 20: Manuelles Suspendieren der Ausführung

```
(1)#include <signal.h>
(2)
(3)void catcher() {
(4)     printf("SIGUSR1 catched\n");
(5)}
(6)
(7)int main(int argc, char** argv[]) {
(8)
(9)     signal(SIGUSR1, catcher);
(10)
(11)    printf("before\n");
(12)    pause();
(13)    printf("after\n");
(14)}
```

Download des Beispiels

Die Windows-Systemfamilie bietet in der Version *Windows 2000* ein dem UNIX-Threadzustandsmodell ähnliches Schema an. (Vgl. hierzu: [Sol00, S. 384f.]) Es unterscheidet zwischen sieben verschiedenen Status, die ein Thread einnehmen kann. Die zusätzlichen Zustände entstehen durch Aufspaltung UNIX-Zustände *wartend* und *blockiert*. Eine Übersicht der verschiedenen Zustände ist in Abbildung 18 dargestellt.

Abbildung 18: *Threadstatus in Windows 2000*



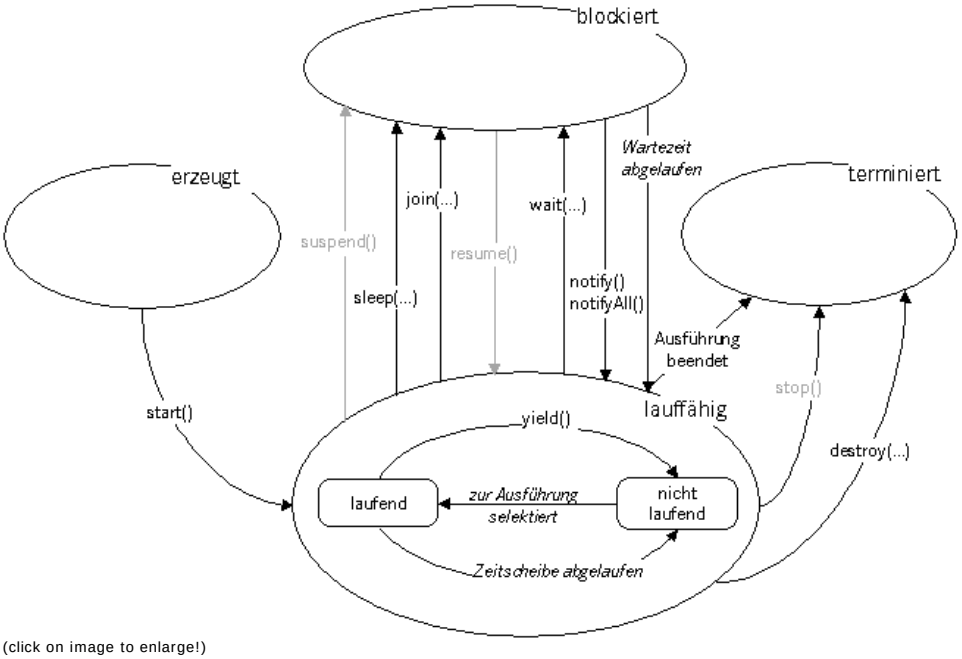
(click on image to enlarge!)

Die einzelnen Zustände sind hierbei:

- **initialisiert** (initialized): Betriebssysteminterner Übergangszustand während der Threaderzeugung.
- **fertig** (ready): Rechenbereite und vollständig speicherresidente Threads. Ausschließlich Threads in diesem Zustand können den Hardwarerecheneinheiten zugeteilt werden.
- **bereit** (standby): Ein in diesem Zustand befindlicher Thread wird der nächsten freiwerdenden CPU zugeordnet. Zu einem Zeitpunkt kann sich nur höchstens ein Thread in diesem Zustand befinden.
- **rechnend** (running): Ein Thread in diesem Zustand ist einer CPU zugeordnet und rechnet.
- **wartend** (waiting): Ein Thread kann durch Eintritt einer Wartebedingung in diesen Zustand versetzt werden. Eine solche Bedingung kann durch Warten auf das Ende einer Ein-/Ausgabeoperation oder durch freiwillige vorzeitige Rückgabe der Zeitscheibe gegeben sein.
- **übergehend** (transition): Ein Thread wird in diesen Zustand überführt, wenn er rechenfertig ist, jedoch threadspezifische Speicherbereiche des Kernels nicht hauptspeicherresident sind.
- **terminiert** (terminated): Sind alle Instruktionen eines Threads ausgeführt, so geht er in diesen Zustand über. Ein Thread muß nach Verlassen dieses Zustands nicht zwingend aus dem Speicher entfernt werden, sondern kann auch neuinitialisiert und erneut ausgeführt werden.

Auch innerhalb der Virtuellen Java Maschine werden Threads in verschiedenen Ausführungsstatus verwaltet. Im Kern sind die hierbei unterschiedenen Phasen mit *erzeugt*, *lauffähig*, *blockiert* und *terminiert* stark and die bereits von UNIX bekannten Modi an. Jedoch stehen durch die verschiedenen Java-Methoden zur Threadverwaltung eine Reihe von Eingriffspunkten zur manuellen Beeinflussung der jeweiligen Zustandsübergänge durch den Programmierer zur Verfügung.
Abbildung 19 zeigt die verschiedenen Zustandsübergänge sowie die Übergangsbedingungen.

Abbildung 19: *Zustandsmodell der Java-Threads*



(click on image to enlarge!)

Unterbrechungsverwaltung und Scheduling

Die Rechenbereitschaft und Zuordnung zu einer physischen Ausführungseinheit ist zwar notwendig, jedoch nicht hinreichend, um einem Thread dauerhaften CPU-Zugriff zu ermöglichen. Vielmehr existieren eine Reihe von Gründen, die -- obwohl die zu berechnende Aufgabe noch nicht erfüllt ist -- zum Entzug der CPU-Ressource führen können. In diesem Falle geht der Thread vom Zustand *laufend* in den Status *blockiert* über und kann zunächst nicht mehr zur Ausführung gelangen.

Zusammenfassend werden alle Ereignisse, welche zur Entziehung der CPU-Ressource eines Threads führen als *Unterbrechung* (engl. *Interrupt*) bezeichnet.
Zusätzlich werden Unterbrechungen hinsichtlich ihrer Ursachen in *synchrone Unterbrechungen* und *asynchrone Unterbrechungenen* unterschieden.
Während synchrone Unterbrechungen innerhalb des ausgeführten Threads selbst durch den Programmcode verursacht werden liegt die Ursache asynchroner Unterbrechungen jenseits des Betriebssystems innerhalb der angeschlossenen Hardware.

Synchrone Unterbrechungen können sich in einem Instruktionsfluß durch die Struktur des ausgeführten Codes (z.B. Division durch Null) ergeben oder beabsichtigt, durch einen Systemaufruf, ausgelöst worden sein. Bei identischer Konfiguration, d.h. Ausführung desselben Programmes an der selben Speicheradresse unter Rekonstruktion des übrigen Systemzustandes, treten synchrone Unterbrechungen erneut in identischer Weise auf.
Asynchrone Unterbrechungen ergeben sich aus den dynamischen Zuständen der verwalteten Hardware und können durch Ein-/Ausgabe oder Auftreten des durch einen Hardwarebaustein periodisch generierten Uhrensignals (Zeitgeber) erzeugt worden sein.
Die nachfolgende Tabelle gibt einen Überblick verschiedener Unterbrechungstypen und ihrer Ursachen.

Unterbrechung	Typ	Auslöser
Ein-/Ausgabeanforderung	asynchron	Geräte-Controller
Seitenfehler	synchron	Betriebssystem
Zeitgeber	asynchron	Uhrenbaustein
Aufruf von Systemfunktionen	synchron	Betriebssystem
Division durch Null	synchron	Betriebssystem
Arithmetik Über-/Unterlauf	synchron	Betriebssystem
Speicherschutzverletzung	synchron	Betriebssystem
Mausbewegung	asynchron	Maus
Tastendruck	asynchron	Tastatur
Netzwerkpaket	asynchron	Netzwerkkarte
USB	asynchron	USB-Controller
Stromausfall	asynchron	Stromversorgung

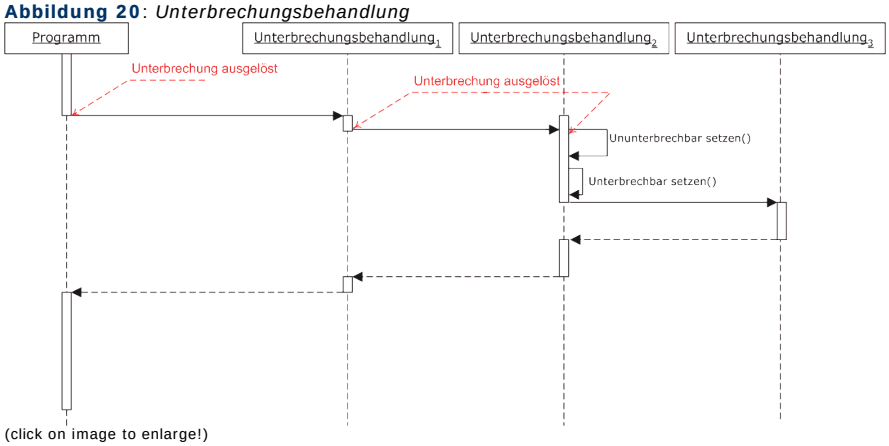
Synchrone Unterbrechungen wie *Seitenfehler*, *Aufruf von Systemfunktionen* und die Arithmetikfehler werden zwar durch die ausgeführten Programminstruktionen bedingt, Unterbrechungsauslöser ist jedoch das Betriebssystem selbst.

Die nähere Analyse der Tabelle zeigt, daß sich die darin aufgelisteten Unterbrechungen nicht nur hinsichtlich ihres Typs, sondern auch graduell im Hinblick auf die anzunehmende Dringlichkeit ihrer Behandlung unterscheiden lassen.

So werden Unterbrechungen zusätzlich zur getroffenen Einteilung in synchrone und asynchrone noch zusätzlich mit Prioritäten versehen. Beispielsweise kann die Behandlung eines Tastendrucks nicht die Verarbeitung eines Zeitgeberereignisses unterbrechen.
Zusätzlich zur Priorisierung von Unterbrechungen kann eine Unterbrechung, welche nicht durch andere Unterbrechungen in der Ausführung ihrer Behandlungsroutine unterbrochen werden darf, sich temporär als ununterbrechbar deklarieren um mit Teilen der Behandlung abzuschließen bevor sie durch weitere Unterbrechungen unterbrochen werden kann.
Im Kern stellt die Ununterbrechbarkeit lediglich eine besondere Form der Priorisierung dar, welche als so hoch einzustufen ist, daß keine anderen Unterbrechungen unterbrechend wirken können.

Im Falle des Auftretens einer Unterbrechung wird die aktuelle Programmausführung so lange suspendiert, bis die Unterbrechung behandelt wurde. Eine Ausnahme hiervon bilden lediglich niederpriorisierte Unterbrechungen, die versuchen höherpriore zu suspendieren und ununterbrechbare Unterbrechungsbehandlungen.
Abbildung 20 zeigt die Behandlung der `Unterbrechungsbehandlung`$_1$ welches einen Thread des in Ausführung befindlichen `Programmes` unterbricht. Während der Behandlung der Unterbrechung wird diese ihrerseits durch die `Unterbrechungsbehandlung`$_2$ unterbrochen und von der Ausführung suspendiert.
`Unterbrechungsbehandlung`$_2$ deklariert sich im Verlauf ihrer Ausführung als Ununterbrechbar. Daher wird das Auftreten der Unterbrechung$_3$ zunächst solange ignoriert, bis sich die laufende `Unterbrechungsbehandlung`$_2$ wieder als ununterbrechbar deklariert und durch die `Unterbrechungsbehandlung`$_3$ unterbrochen wird.
Nach Abschluß der `Unterbrechungsbehandlung`$_3$ kehrt diese Routine an diejenige Stelle zurück, in der `Unterbrechungsbehandlung`$_2$ verlassen wurde und schließt die Behandlung dieser Unterbrechung ab. Nach Ende der `Unterbrechungsbehandlung`$_2$ kehrt diese zur `Unterbrechungsbehandlung`$_1$ zurück, welche sie initial unterbrochen hat.
Ist auch diese Unterbrechungsbehandlung abgeschlossen und wurde nicht zwischenzeitlich durch andere Unterbrechungen unterbrochen, so wird zur Programmausführung zurückgekehrt.

Abbildung 20: *Unterbrechungsbehandlung*



(click on image to enlarge!)

Scheduling

Zentrale Bedeutung in der Steuerung des Systemverhaltens kommt der Zeitgeberunterbrechung zu. In Betriebssystemen mit präemptivem Multitasking legt sie die technische Basis der Parallelitätssteuerung. Konkret wird jeder zur Ausführung selektierte Thread höchstens für eine feststehende Zeitspanne (die sog. *Zeitscheibe*) ausgeführt, sofern er nicht bereits vor Ablauf der maximalen Rechenzeit blockiert wurde, und nach Ablauf der Zeitscheibe wird ihm die CPU-Ressource entzogen und automatisch dem betriebssysteminternen *Scheduler*-Thread zugeteilet. Dieser während der gesamten Laufzeit des Betriebssystems aktive Thread entscheidet dann welcher Thread aus der Menge der rechenbereiten entnommen und als nächstes der CPU zugeteilt wird.

Die konkrete Auswahl eines Threads aus der Menge der um Ausführung konkurrierenden erfolgt nach festen Kritierien, die im *Schedulingalgorithmus* hinterlegt sind.
Grundsätzlich wird in aktuellen Betriebssystemimplementierungen ausschließlich die durch einen Prozeß verursachte CPU-Belastung als Kriterium der Prozessorzuteilug herangezogen und auftretende Ein-/Ausgabeoperationen vernachlässigt. Dieser Einschätzung liegt die Auffassung zu Grunde, daß das Geschwindigkeitsverhalten im wesentlichen durch die zu berechnenden Operationen determiniert wird. Die zunehmend leistungsfähigen Prozessoren offebanren jedoch eine starke Abhängigkeit der Gesamtausführungsgeschwindigkeit von den durchzuführenden Lese- und Schreibzugriffen, da moderne Hauptprozessoren um einen Faktor 10.000 schneller Ergebnisse erzeugen, als sie Festplatten zu liefern vermögen. Daher wird auch das Ein-/Ausgabeverhalten zukünftig in die Zuteilungsstrategie der Threads eingang finden.

Nachfolgend werden die aktuell präsenten Basisansätze zur Steuerung der Parallelität auf Threadebene vorgestellt und abschließend an Beispielen aus realen Betriebssystemen diskutiert.

Die Kernfrage der Planung der Threadumschaltung bildet die Fragestellung wann eine Umschaltung zwischen zwei Threads erfolgen soll. Diese Entscheidung kann beim Eintreten verschiedener Ereignisse im System imminent werden:

- Threaderzeugung.
  Nach der Erzeugung einer neuen rechenfähigen Verwaltungseinheit im Betriebssystem ist zu klären ob diese direkt ausgeführt wird, oder ob der erzeugende Thread zunächst zuende rechnet.
- Threadterminierung.
  Beendet sich ein aktuell in Ausführung befindlicher Thread oder wird er durch ein externes Ereignis terminiert, so ist ein neuer rechenwilliger Thread zur Ausführung zu selektieren. Liegt ein solcher nicht vor, so wird dem Leerlaufthread (*idle*-Thread) die CPU zugeteilt.
- Threadblockade.
  Wird ein Thread aufgrund einer Unterbrechung blockiert, so muß seine Ausführung suspendiert werden, bis die

Unterbrechung behandelt wurde.
- Ende einer Unterbrechungsbehandlung.
  Ist die Behandlung einer Unterbrechung, die zur Blockade eines rechnenden Threads geführt hat, abgeschlossen, so ist der blockierte Thread wieder der Menge der rechenbereiten und -willigen zuzuordnen.
- Zeitgeberereignis.
  Unterbrechung des aktuell laufenden Threads und Selektion eines rechenbereiten zur Ausführung.

Prinzipiell sollte sich die Auswahl eines geeigneten Schedulingalgorithmus an der Natur der Ausführungsumgebung orientieren. So muß der Theadumschaltung in batch-orientierten Umgebungen deutlich weniger Aufmerksamkeit zu Teil werden, als für hoch interaktive Systeme, da das Toleranzverhalten der wartenden Anwender bei Stapelverarbeitung deutlich ausgeprägter erscheint. Ebenso bedingen Echtzeitsysteme, welche in fixierten Zeitintervallen die Lieferung einer Antwort garantieren müssen deutlich stärkere Restriktionen an die Auswahl des nächsten rechenbereiten Prozesses als interaktive Workstation- oder Desktopsysteme.
Einen Überblick der spezifischen Zielsetzungen liefert die nachfolgende Zusammenstellung (vgl. [Tan02, S. 153]):

- Generell:
  - Fairness -- Jeder Thread erhält Rechenzeit
  - Offensichtlichkeit -- Vergabekritierien für Rechenzeit sind offengelegt und nachvollziehbar
  - Balance -- Alle Teile des Systems sind annähernd gleichmäßig ausgelastet
- Batch-Systeme:
  - Durchsatz -- Maximiere Auslastung der verfügbaren Ressourcen
  - Durchlaufzeit -- Minimiere Durchlaufzeit jedes Einzeljobs
  - CPU-Belastung -- Maximiere Auslastung der CPU
- Interaktive Systeme:
  - Antwortzeit -- Minimiere Antwortzeit für Benutzeranfragen
- Echtzeitsysteme:
  - Vorhersagbarkeit -- Garantie des Eintreffens einer Bedingung (z.B. Beendigung einer Berechungsaufgabe) zu einem fixierten Zeitpunkt

In Batch-Systemen wird die Jobverarbeitung im ununterbrechbaren Modus angeboten, d.h. rechenwillige Jobs können bereits rechnende nicht präemptiv unterbrechen. Daher obliegt dem Scheduling Algorithmus „lediglich“ die Auswahl des nächsten auszuführenden Jobs aus der Warteschlange der rechenbereiten nach Beendigung der aktuell verarbeiteten Aufgabe.

First-Come First Served: Einfachster Algorithmus, der Jobs in der Reihenfolge ihres Eintritts in die Warteschlange zur Verarbeitung selektiert. Wird ein Job während seiner Ausführung blockiert, so wird er am Ende der Warteschlange eingereiht und muß auf die erneute Zuteilung der CPU warten bis alle vor ihm befindlichen Jobs zugeteilt worden sind. Dieser, grundsätzlich auf andere Systemtypen übertragbare, Zuteilungsalgorithmus läßt sich mit relativ wenig Aufwand implementieren und ausführen. Augenscheinlich erfüllt die Einordnung am Warteschlangende auch die gestellten Kriterien der Fairness und Offenheit. Allerdings birgt die Vorgehensweise auch einen gravierenden Nachteil in sich. So benachteiligt die First-Come First-Served-Vorgehensweise Ein-/Ausgabe-intensive Prozesse tendenziell, da diese nach jeder Blockierung am Ende der Warteschlange eingereiht werden. Die hierdurch eintretende Wartezeit verlängert sich zusätzlich proportional zur Systemlast, die direkt mit der Warteschlangenlänge korreliert ist.
Beispiel 21 zeigt ein Beispiel für die Zuteilungsstrategie nach dem First-Come-First-Served-Prinzip.

Beispiel 21: Job-Scheduling nach dem First-Come-First-Served-Prinzip
Gegeben seien die Aufgaben j_1, j_2 und j_3 mit den individuellen Laufzeiten
l: $l(j_1)=3$, $l(j_2)=1$ und $l(_3)=2$.

Bei Zuteilung nach dem First-Come-First-Served-Prinzip ergeben sich daher als Wartezeiten $w(j_i)$:
$w(j_1)=3$
$w(j_2)=w(j_1)+1=3+1=4$
$w(j_3)=w(j_2)+2=w(j_1)+1+2=3+1+2=6$

Daraus ergibt sich die mittlere Wartezeit pro Job als
$(3+4+6)/3=4^1/_3$.

Shortest Job First: Diese Zuteilungsstrategie setzt voraus, daß die Laufzeit einer Aufgabe im Voraus bekannt ist und alle Jobs gleichzeitig vorliegen und sich die Menge der zu verarbeitenden Aufgaben während der Verarbeitung nicht ändert. Ist dies der Fall, so werden die Job in der aufsteigenden Reihenfolge ihrer Ausführungszeiten zur Ausführung selektiert. Diese Scheduling-Strategie liefert zuverlässig optimale Ergebnisse, ist jedoch aufgrund der Eingang aufgestellten Restriktionen nur schwer unzusetzen, da in der Praxis häufig auch während der Ausführung noch weitere Jobs hinzukommen, die bei der nächsten Zuteilungsrunde berücksichtigt werden müssen. Insbesondere kann die Ankunft eines „Kurzläufers“, d.h. Jobs mit kurzer Laufzeit, die Modifikation einer bereits erstellten Zuteilungsreihenfolge bedingen. Überdies ist die Laufzeit von Jobs vor ihrer Ausführung nur schwer zu bestimmen, da beispielsweise die Dauer von Ein-/Ausgabeoperationen nicht im allgemeinen Falle bestimmt werden kann.
Beispiel 22 zeigt ein Beispiel für die Zuteilungsstrategie nach dem Shortest-Job-First-Prinzip sowie die möglichen anderen Zuteilungsstrategien.

Beispiel 22: Job-Scheduling nach dem Shortest-Job-First-Prinzip

Gegeben seien die Aufgaben und Laufzeiten aus Beispiel 21.
Bei Zuteilung nach dem Shortest-Job-First-Prinzip ergibt sich daher die

Durchlaufreihenfolge $s_1=\{j_2, j_3, j_1\}$.

Die Wartezeiten *w* für jeden Job bei dieser

Durchlaufreihenfolge ergeben sich als:
$w(j_1)=w(j_3)+3=w(j_2)+2+3=1+2+3=6$

$w(j_2)=1$

$w(j_3)=w(j_2)+2=1+2=3$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge
s_1 liegt daher bei $(1+2+3)/3=3^1/_3$.

Bei der Alternativzuteilung nach dem First-Come-First-Served-Prinzip ergibt sich die in Beispiel 21 berechnete mittlere Durchlaufzeit
als $(3+4+6)/3=4^1/_3$.

Bei Wahl der Abarbeitungsreihenfolge $s_3=\{j_1, j_3, j_2\}$

(dies entspricht der Zuteilungsstrategie *Longest-Job-First*) ergeben sich die individuellen Wartezeiten als **$s_3=\{j_1, j_3, j_2\}$**:

$w(j_1)=3$

$w(j_2)=w(j_3)+1=w(j_1)+2+1=3+2+1=6$

$w(j_3)=w(j_1)+2=5$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge
s_3 liegt daher bei $(6+5+3)/3=4^2/_3$.

Bei Wahl der Abarbeitungsreihenfolge j_2, j_1, j_3 ergeben sich die individuellen Wartezeiten als:
$s_4=\{j_2, j_1, j_3\}$:

$w(j_1)=w(j_2)+3=1+3=4$

$w(j_2)=1$

$w(j_3)=w(j_1)+2=w(j_2)+3+2=1+3+2=6$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge
s_4 liegt daher bei $(4+1+6)/3=3^1/_3$.

Bei Wahl der Abarbeitungsreihenfolge j_3, j_1, j_2 ergeben sich die individuellen Wartezeiten als:
$s_5=\{j_3, j_1, j_2\}$:

$w(j_1)=w(j_3)+3=2+3=5$

$w(j_2)=w(j_1)+1=w(j_3)+3+1=6$

$w(j_3)=2$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge
s_5 liegt daher bei $(5+6+2)/3=4^1/_3$.

Bei Wahl der Abarbeitungsreihenfolge j_3, j_2, j_1 ergeben sich die individuellen Wartezeiten als:
$s_6=\{j_3, j_2, j_1\}$:

$w(j_1)=w(j_2)+3=w(j_3)+1+3=2+1+3=6$

$w(j_2)=w(j_3)+1=2+1=3$

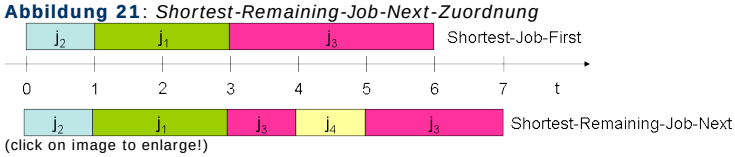
$w(j_3)=2$

Die mittlere Durchlaufzeit pro Job bei Anwendung der Abarbeitungsreihenfolge
s_5 liegt daher bei $(6+3+2)/3=3^1/_3$.

Beispiel 22 zeigt, daß die Zuteilungsstrategie nach dem Shortest-Job-First-Prinzip immer ein optimales Ergebnis liefert, das die Wartezeit je Job verkürzt. Neben der durch die vorgestellte Strategie gefundenen Abarbeitungsreihenfolge können jedoch noch weitere existieren, welche dasselbe Optimum erreichen (im Beispiel sind dies s_4 und s_6), jedoch keine Alternativreihenfolge, welche das durch Bevorzugung des kürzestlaufenden Jobs ermittelte Optimum unterbieten würde.

Durch Modifikation des Shortest-Job-First-Prinzips zum **Shortest Remaining Job Next** läßt sich eine der beiden zentralen Einschränkungen des Shortest-Job-First-Algorithmus aufheben. Zwar erfordert auch die modifizierte Variante unverändert Kenntnis über die Laufzeit eines Jobs, jedoch wird der Zuteilungsvorgang nicht mehr nur genau einmal für eine Menge von Jobs ausgeführt, sondern die Zuteilung bei Ankunft eines neuen Jobs dynamisch neu vorgenommen.
Der Zuteilungsalgorithmus unterscheidet sich daher nur unwesentlich vom Shortest-Job-First-Prinzip und legt den nächsten zu verarbeiten Job bei Ankunft eines neuen neu fest. So können neuankommende Aufgaben bereits bestehende verdrängen, sofern sie eine Gesamtlaufzeit aufweisen, die kürzer als die Restlaufzeit des aktuell verarbeiteten Jobs ist.
Die Abarbeitungszeit des unterbrochenen Jobs verängert sich dabei um die Dauer des „zwischengeschalteten“ Jobs, der ihn unterbrach.
Die Shortest-Remaining-Job-Strategie erweist sich als besonders Fair, was die Abarbeitung von (eigentlich bevorzugt zu verarbeitenden) kurzlaufenden Aufgaben betrifft und beseitigt deren -- durch verspätete Ankunft hervorgerufene -- Benachteiligung im Shortest-Job-First-Prinzip.
Abbildung 21 zeigt die Unterbrechung des aus den vorhergehenden Beispielen bekannten Jobs j_3 nach einer Zeiteinheit durch den neuankommenden Job j_4 der Dauer 1.

Abbildung 21: *Shortest-Remaining-Job-Next-Zuordnung*



(click on image to enlarge!)

Während alle bisher disktutierten Verfahren ausschließlich auf den Stapelverarbeitungsbetrieb zugeschnitten sind, läßt sich die *Shortest-Remaining-Job-Next*-Zuteilungsstrategie sowohl in Batch-orientierten als auch interaktiven Systemen einsetzen. Allerdings haben sich im Laufe der Zeit eine Reihe spezialisierter Zuteilungsalgorithmen für den Typus multitaskingfähiger interaktiver Systeme herausgebildet, die deren besondere Gegebenheiten besser berücksichtigen. Einige ausgewählte Basistechniken werden nachfolgend eingeführt.

Das älteste und daher in der Praxis auch am weitesten verbreitete Schedulingverfahren interaktiver Systeme ist unter dem Namen *Round Robin geläufig.*
Basisprinzip des Round-Robin-Verfahrens ist die Zuweisung eines fixierten Ausführungsquantums -- der Zeitscheibe -- an jeden Thread. Für die durch das Quantum festgelegte Zeitscheibe erhält jeder Thread exklusiv die CPU und wird nach Ablauf der zugeteilten Zeitscheibe durch den Scheduler unterbrochen, sofern er nicht zuvor durch eine Unterbrechung blockiert wurde.
Nach Entzug der CPU wird der Thread in eine Wartschlange eingereiht und der nächst in der Schlange befindliche Thread erhält die CPU.
Round-Robin-basierte Prozessorzuteilung setzt keinerlei Kenntnisse über die Laufzeit eines Threads voraus und kann daher leicht implementiert werden. Die technische Umsetzung erfordert lediglich die Verwaltung einer Liste der Länge der maximal verwaltbaren Threadanzahl.
Wichtigster Freiheitsgrad des Round-Robin-Verfahrens ist die Länge des CPU-Quantums, welches einem Thread zugeteilt wird. Aufgrund des zu kalkulierenden Zeitaufwandes für die Threadumschaltung (Kontextwechsel) sollte die Zeitscheibe nicht zu gering festgelegt werden, um das Verhältnis zwischen Rechen- und Verwaltungszeit positiv zu halten. Gleichzeitig sollte die Zeitscheibe nicht zu großen Umfang annehmen, da dies die Interaktivität (d.h. die Wartezeit eines Threads auf CPU-Zutteilung) negativ beeinträchtigt.

Wird die Laufzeit l mit $l(t_1)=4$, $l(t_2)=1$ und $l(t_3)=2$ angenommen sowie eine Zeitscheibengröße von genau einer Zeiteinheit fesgelegt, so ergibt sich der in Beispiel 23 dargestellte Ablauf.

Beispiel 23: Round-Robin-Scheduling

1. Rechnend: t_1

Warteschlange: {t_2, t_3}
2. Rechnend: t_2

Warteschlange: {t_3, t_1}
3. Rechnend: t_3

Warteschlange: {t_1}

Anmerkung: t_2 hat zu Ende gerechnet.
4. Rechnend: t_1

Warteschlange: {t_3}
5. Rechnend: t_3

Warteschlange: {t_1}

Anmerkung: t_3 hat zu Ende gerechnet.
6. Rechnend: t_1

Warteschlange: {}
7. Rechnend: t_1

Warteschlange: {}

Anmerkung: t_1 hat zu Ende gerechnet.

Das Beispiel illustriert die Gleichbehandlung aller im System aktiven Threads als weitere zentrale Eigenschaft des Round-Robin-Ansatzes. Dieses Verhalten ist oftmals nicht gewünscht. So sollten Threads, die wichtige Systemereignisse verarbeiten häufiger zugeordnet werden als gewöhnliche Anwenderthreads. Daher läßt sich das vorgestellte Verfahren durch die Zuordnung von Prioritäten zu jedem Thread dahingehend qualitativ verbessern, daß einzelne Threads bevorzugt zugeordnet werden.
Die Warteschlangenverarbeitung des Round-Robin-Algorithmus wird als Konsequenz von einer reinen Sortierung gemäß der Erzeugungsreihenfolge auf eine prioritätsbasierte Reihung umgestellt. Der Zuteilungsvorgang wählt unverändert das erste Element der Warteschlange aus und ordnet ihm die CPU-Ressource für den durch die Zeitscheibe vorgegebenen Zeitraum zu.

Um das „Verhungern“ niederpriorer Threads -- sie würden permanent hinter höherprioren eingeordnet und erst nach deren Ende in der CPU-Verteilung berücksichtigt werden -- zu verhindern werden in der Praxis die Threadprioritäten dynamisch berechnet. Daher wird die *dynamische Priorität* des aktuell ausgeführten Prozesses nach dem Entzug der CPU, in Abhängigkeit seiner statischen Basispriorität, neu festgelegt.
Beispiel 24 zeigt die prioritätsbasierte Zuordnungsreihenfolge dreier Threads.

Beispiel 24: Prioritätsbasiertes Round-Robin-Scheduling

Gegeben seien drei mit Prioritätsstufen versehene Threads,

so daß gelte: $p(t_1)=3$, $p(t_2)=1$ und $p(t_3)=2$.

Die CPU-Zuteilung erfolge dynamisch prioritäsgesteuert.

Initial sei jeder Thread mit einer dynamischen Priorität von 10 versehen, von der die definierte Prioritätsstufe abgezogen wird. Dem Thread mit der numerisch höchsten Priorität wird anschließend die CPU zugeteilt. Nach Ablauf der Zeitscheibe oder Blockierung des Threads wird die dynamische Priorität um die definierte Prioritätsstufe vermindert.

Stehen im System mehrere Threads gleicher dynamischer Priorität zur Verfügung,

so wird bevorzugt einer ausgewählt, der noch nicht ausgeführt wurde. Steht kein Thread höherer dynamischer Priorität zur Verfügung, so wird der Thread, dem aktuell die CPU zugeteilt ist, weiter ausgeführt.

Nach einmaliger Zuteilung aller Threads beginnt das Verfahren von neuem.

1. Rechnend: t_2 (dynamische Priorität: 10-1=9)

Wartend: t_1 (dynamische Priorität: 10-3=7; noch nie ausgeführt), t_3 (dynamische Priorität: 10-2=8; noch nie ausgeführt)

2. Rechnend: t_2 (dynamische Priorität: 9-1=8

Wartend: t_1 (dynamische Priorität: 7; noch nie ausgeführt), t_3 (dynamische Priorität: 8; noch nie ausgeführt)

Anmerkung: Verminderung der dynamischen Priorität des ausgeführten Threads, dynamische Priorität der wartenden Threads unverändert.

3. Rechnend: t_3 (dynamische Priorität: 8)

Wartend: t_1 (dynamische Priorität: 7), t_2 (dynamische Priorität: 8-1=7)

Anmerkung: t_2 bei Kontextwechsel verdrängt, da t_3 höhere dynamische Priorität besitzt.

4. Rechnend: t_1 (dynamische Priorität: 7)

Wartend: t_2 (dynamische Priorität: 7), t_3 (dynamische Priorität: 6)

Anmerkung: t_1 verdrängt t_3 trotz „nur“ gleicher dynamischer Priorität, da t_1 noch nie ausgeführt wurde.

Nach dieser Zuteilung beginnt das Verfahren von neuem, da alle Threads einmal zugeordnet wurden.

Zur vereinfachten Handhabung bieten die Implementierungen des Round-Robin-Verfahrens typischerweise gestufte Prioritätsklassen an, in die einzelne Threads eingeordnet werden. Windows 2000 bietet beispielsweise 32 verschiedene Prioritätsstufen an, von denen 16 (Stufen 16-31) Echtzeitthreads, 15 (Stufen 1-15) Anwenderthreads und genau eine (Stufe 0) dem Leerlaufprozeß vorbehalten ist. Innerhalb der Anwenderprioritätsstufen wird zwischen *hochprioren* (Klasse *high*), *höherprioren* (Klasse *above normal*), *normalen* (Klasse *normal*), *niederprioren* (Klasse *below normal*) und *niedrigprioren* (Klasse *idle*) Threads unterschieden.
Windows 2000 verwendet zur Abwicklung des Schedulings dynamische Prioritäten um eine faire Zuteilungsstrategie zu gewährleisten.

Beim Einsatz von Prioritätsklassen wird typischerweise prioritätsbasiertes Round-Robin-Scheduling zwischen den

Prioritätsklassen und Round-Robin-Verteilung innerhalb der Prioritätsklassen eingesetzt. Daher bedingt diese Umsetzungsform den Einsatz mehrer eigenständiger Warteschlangen, die jeweils genau einer Prioritätsklasse zugeordnet sind.

Der zentrale Vorteil des Round-Robing-Verfahrens ist die einfache Umsetzung und die Unabhängigkeit von, typischerweise vorab nicht verfügbaren, Aussagen über die (erwartete) Laufzeit eines Threads. Allerdings böte die Berücksichtigung von Laufzeitaspekten eine weitere Quelle von Informationen über den Thread, die im Rahmen der Selektion des nächsten auszuführenden Threads berücksichtigung finden könnte. Im Grunde handelt es sich dabei um zusätzliche Information, die wie die Aussage über die bereits erfolgte CPU-Zuordnung im Schedulingalgorithmus der durch Beispiel 24 betrachtet wurde in die Auswertung einbezogen werden kann. Ebenso wie im Beispiel geschehen, könnten Aussagen über das zurückliegende Laufzeitverhalten eines Threads gesammt und ausgewertet werden.
Einen Ansatz hierzu bildet eine Übertragung des *Shortest-Job-First*-Verfahrens auf interaktive Systeme. Hierbei werden während der Ausführung Daten über CPU-Verweildauer gesammelt. Diese werden bei der dynamischen Festlegung der Priorität geeignet einbezogen, um Threads mit kurzer Laufzeit zu bevorzugen.
Um ein realistisches Bild des veränderlichen Threadverhaltens zu erhalten bietet es sich an, nicht die gesamte bisherige Lebensdauer des Threads zur Beurteilung des Laufzeitverhaltens heranzuziehen, sondern lediglich auf jüngere Daten zurückzugreifen und diese geeignet zu gewichten. Einen Ansatz hierzu stellt die „Alterung“ (engl. *aging*) der gesammelten Daten dar, die ein gleitendes Fenster fester Größe zur Ermittlung der Einflußfaktoren auf die Prioritätsfestlegung heranziehen. Beispiel 25 zeigt die Berechnung dieses Einflußfaktors.

Beispiel 25: Aging

Gegeben seien drei mit Prioritäten versehene Threads, so daß gilt: $p(t_1)=2$, $p(t_2)=3$ und $p(t_3)=1$. Als Schedulingalgorithmus findet das in Beispiel 24 beschriebene Verfahren anwendung. Zusätzlich wird aus den während der Laufzeit ermittelten CPU-Verweildauer eine zweite Priorität dynamisch ermittelt, welcher zur ersten addiert wird.

Thread	dynamische Priorität	Status
Schedulerlauf 1:		
t_1	10-2+10-0=8+10=18	wartend
t_2	10-3+10-0=7+10=17	wartend
t_3	10-1+10-0=9+10=19	rechnend
Schedulerlauf 2:		
t_1	18	wartend
t_2	17	rechnend
t_3	9-1+10-0/2-7=8+3=11	wartend
Schedulerlauf 3:		
t_1	18	rechnend
t_2	7-3+10-0/2-2=4+8=12	wartend
t_3	11	wartend
Schedulerlauf 4:		
t_1	8-2+10-0/2+5=6+50=11	wartend
t_2	12	rechnend
t_3	11	wartend
Schedulerlauf 5:		
t_1	11	wartend
t_2	4-3+10-0/4+2/2-1=1+8=9	wartend
t_3	11	rechnend

Während des ersten Schedulerlaufes liegen noch keine Daten über das dynamische Verhalten der (neu erzeugten) Threads vor, daher ist die zugehörige Prioriätskomponente durchgängig auf den Wert 0 gesetzt.
Beim zweiten Durchlauf des Schedulers wird Thread t_2 zur Ausführung selektiert. Zusätzlich werden die gesammelten Ablaufdaten des ausgeführten Threads mit der Priorität 7 bewertet und bei der Neufestlegung der dynamischen Priorität entsprechend berücksichtigt.
Die dritte Anwendung des Scheduleralgorithmus liefert t_1 als Thread mit der höchsten dynamischen Priorität und teil diesem die CPU zu.
Während seiner Ausführung werden auch über ihn dynamische Daten gesammelt, die beim vierten Schedulerlauf entsprechend in die Neufestlegung der dynamischen Priorität Eingang finden.
Der vierte Schedulerlauf wählt bereits zum zweiten Mal den mit der niedersten statischen Thread-Priorität versehenen t_3 zur Ausführung aus, da der Zuteilung-Algorithmus -- gestützt auf den gesammelten Verhaltensdaten -- seine erneute Selektion empfiehlt.
Der fünfte Durchlauf des Schedulers selektiert wahlfrei t_3 zur Ausführung, da seine dynamische Priorität identisch zu

t_3 ist und beide Thread bisher gleichhäufig ausgeführt wurden. Thread t_2 erhält die dynamische Priorität 9 zugeteilt. Gleichzeitig zeigt die Berechung seiner Priorität die Wirkung des *agings*, bei dem mehrere historische Werte, nach ihrem Alter gewichtet, in die Berechung einbezogen werden.

Insgesamt zeigt das Beispiel auch die Wirkungsmacht des Einbezugs von Laufzeitinformation in die Prioritätsfestlegung. So wird Thread t_2 -- obwohl er über die niedrigste statische Priorität verfügt --, vermöge der positiven Laufzeitdaten, ebenso häufig ausgeführt wie t_3, der mit der höchsten statischen Priorität ausgestattet wurde.

Eine alternative Schedulingvariante ergibt sich, wenn die Zielsetzung der größtmöglichen Fairness in den Vordergrund gerückt wird. So ergibt sich durch die Verfolgung der Zielsetzung jedem Thread möglichst denselben Anteil an CPU-Zeit zuzuteilen eine **Scheduling-Garantie** auf die sich jeder Anwender verlassen kann.
Als Ziel kann hierbei verfolgt werden jedem Thread relativ zu seiner Lebenszeit dieselbe Menge Zuordnungen zuteil werden zu lassen. Beispiel 26 zeigt einen exemplarischen Ablauf:

Beispiel 26:

Thread	Alter	bisherige Zuteilungen	Status
Schedulerlauf 1:			
t_1	1	0	rechnend
Schedulerlauf 2:			
t_1	2	1	rechnend
Schedulerlauf 3:			
t_1	3	2	wartend
t_2	1	0	rechnend
Schedulerlauf 4:			
t_1	4	2	rechnend
t_2	2	1	wartend
Schedulerlauf 5:			
t_1	5	3	wartend
t_2	3	1	rechnend
Schedulerlauf 6:			
t_1	4	2	rechnend
t_2	3	2	wartend

Das Beispiel zeigt fünf aufeinanderfolgende Zurodnungen unter Anwendung garantierter Zuteilung. Zur CPU-Zuordnung ausgewählt wird jeweils der Thread, dessen Alter (d.h. Verweildauer im System) gegenüber den bisher erfolgten Zuteilungen am kleinsten ist.

Die dargestellte Schedulingvariante berücksichtigt ausschließlich die Anzahl der laufenden Threads und verspricht diesen Gleichbehandlung. Eine unter dem Namen **Fair-Share Scheduling** bekannte Variante dieses Vorgehens betrachtet nicht die Anzahl der Threads, sondern die Anzahl der sie besitzenden Anwender und garantiert die Gleichbehandlung aller Anwender statt der durch sie gestarteten Berechnungsaufgaben.
Ziel dieser Modifikation des Algorithmus ist es die inhärente Bevorzugung derjenigen Anwender, die viele Threads starten, zu eliminieren.
Eine Umsetzungsmöglichkeit wäre daher die Nutzung von garantiertem Scheduling zwischen den Anwendern und Zuordnung der Threads eines Anwenders im Round-Robin-Verfahren.

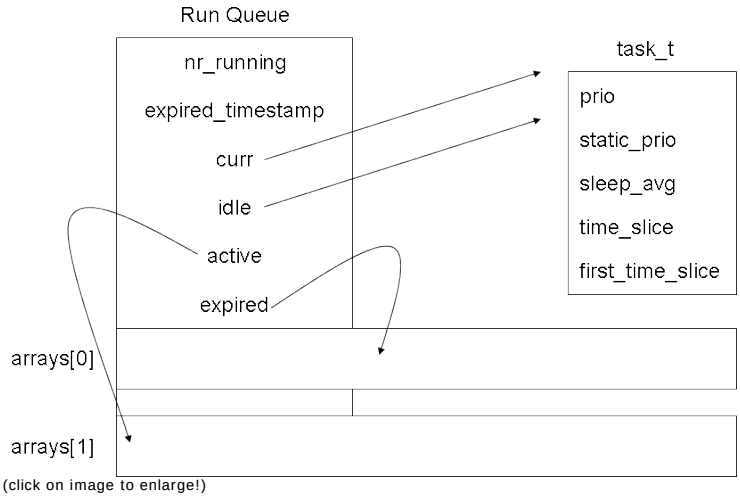
Das **Lotterie-Scheduling** greift die Idee der Fairness auf und steigert das Versprechen des Algorithmus der garantierten Zuordnung sogar noch. So werden beim Lotterie-ähnlichen Scheduling Lose an alle im System laufenden Threads verteilt. Jede zu vergebende Zeitscheibe wird unter den rechenwilligen Threads „verlost“. Gewinnt das Los eines Threads, so erhält er die CPU zugeordnet.
Der zugrundeligende Algorithmus ist vergleichsweise einfach umzusetzen und gestattet sogar, durch die Verteilung von mehr als einem Los an einen Thread, die Nachbildung von Prioriätten. Allerdings hängt die praktische Einsetzbarkeit stark von der Qualität der zur Verfügung stehenden Zufallszahlen, welche die Basis des „Glücksspiels“ bilden ab. Sind diese nicht gleichverteilt, sondern treten gehäuft oder gar periodisch auf, so kann dies zur unerwünschten Bevorzugung einzelner Threads führen.

Die Windows 2000 Systemfamilie setzt eine Variante des garantierten Schedulings ein, welche jedem im System laufenden Thread grundsätzlich denselben CPU-Anteil (Quantum) garantiert. Zusätzlich verfügt jeder Thread über eine Prioritätsebene, welche seine Zuordnungshäufigkeit steuert. Die Workstations-Editionen der Windows-Betriebssysteme sehen zusätzlich, beispielsweise zur Steigerung derjenigen Threads die einem Programm zugeordnet sind, mit welchem der Anwender aktuell arbeitet, Prioritäts-*Boosts* vor, die kurzfristig die Priorität eines Threads steigern können.

Das Linux-Betriebssystem verfügt seit der Version 2.6 über einen neuen Scheduleransatz, welcher die faire CPU-Zuteilung auch bei hohen Systemlasten garantieren soll. Da Linux im Betriebssystemkern ausschließlich Prozesse behandelt berücksichtigt der Schedulingalgorithmus auch ausschließlich diese als kleine Einheit der Rechenzeitverteilung.
Ziel des Schedulingalgorithmus ist es größtmögliche Skalierbarkeit zu verwirklichen und die CPU-Zuteilung in konstanten Zeitintervallen zu garantieren. Der Schedulingalgorithmus besitzt daher die Komplexität *O(1)*.

Grundlegende Datenstruktur des Schedulers ist die in Abbildung 22 zusammengestellt.

Abbildung 22: *Datenstruktur des Linux-Schedulers*



(click on image to enlarge!)

TODO:

Hier geht's demnächst weiter ...

Web-Referenzen 5: Weiterführende Links

- PThreads-Tutorial
- YoLinux Tutorial: POSIX thread (pthread) libraries
- Vorlesung *Java Threads*

Schlagwortverzeichnis