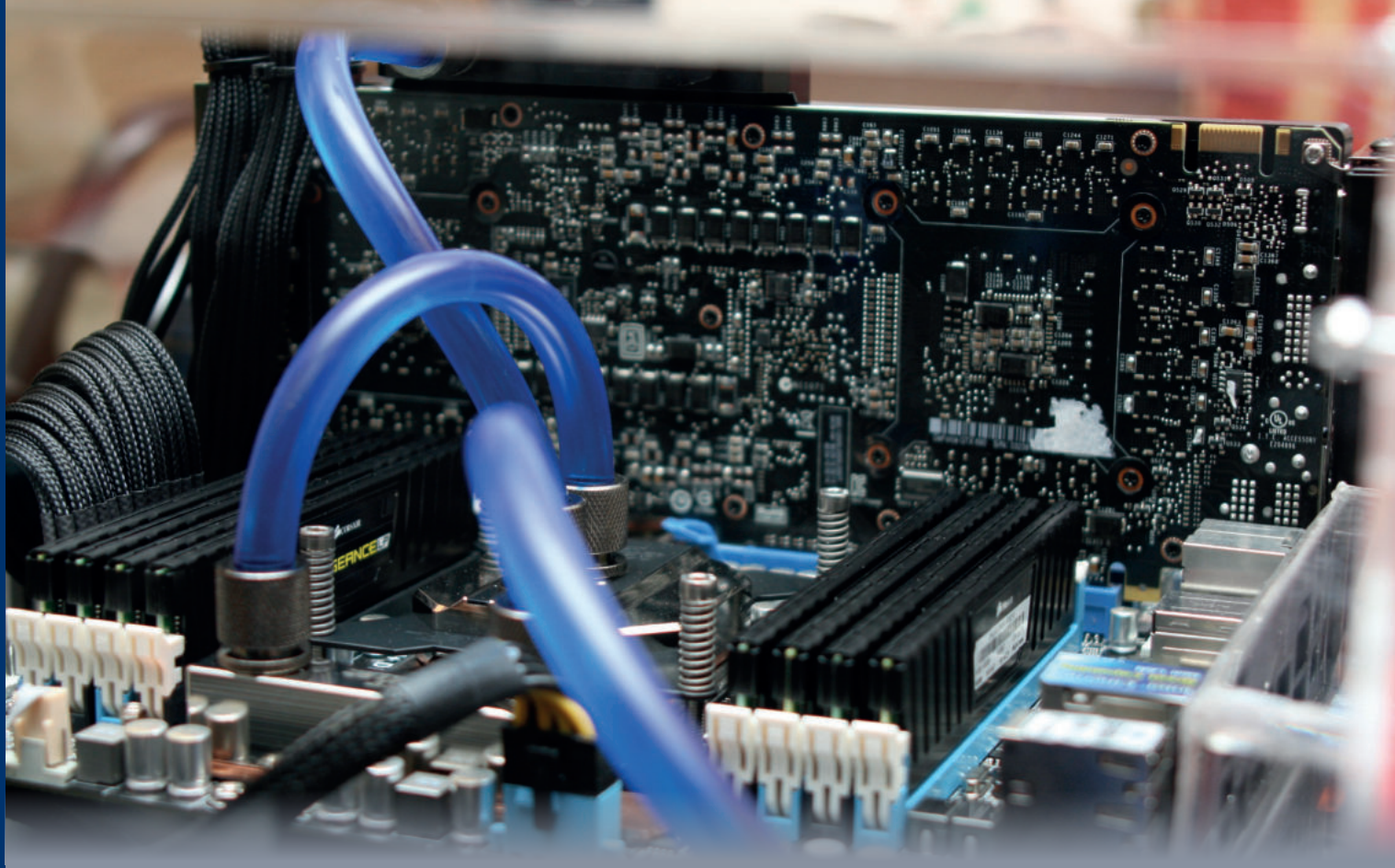


Zertifikatsprogramm - Z102

Systemnahe Programmierung

- Rechnerstrukturen und Betriebssysteme
- Assemblerprogrammierung
- Die Programmiersprache C
- Systemnahe Sicherheitsaspekte
- Programmierprojekt

Dr. rer. nat.Werner Massonne

Open C³S
Open Competence Center for Cyber Security

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Modul Z102
Systemnahe Programmierung

Studienbrief 1: Rechnerstrukturen und Betriebssysteme

Studienbrief 2: Assemblerprogrammierung

Studienbrief 3: Die Programmiersprache C

Studienbrief 4: Systemnahe Sicherheitsaspekte

Studienbrief 5: Programmierprojekt

Autor:
Dr. rer. nat. Werner Massonne

---

2. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

2. Auflage (6. August 2018)

Didaktische und redaktionelle Bearbeitung:
Romy Rahnfeld, Patrick Eisoldt

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Inhaltsverzeichnis

Einleitung zu den Studienbriefen

I. Abkürzungen der Randsymbole und Farbkodierungen

Definition	D
Exkurs	E
Übung	Ü

II. Zu den Autoren

Werner Massonne erwarb sein Diplom in Informatik an der Universität des Saarlandes in Saarbrücken. Er promovierte anschließend im Bereich Rechnerarchitektur mit dem Thema „Leistung und Güte von Datenflussrechnern“. Nach einem längeren Abstecher in die freie Wirtschaft arbeitet er inzwischen als Postdoktorand bei Professor Freiling an der Friedrich-Alexander-Universität.

III. Modullehrziele

Dieses Modul beschäftigt sich mit Programmiertechniken, die auf einem tiefen Level auf die Gegebenheiten eines Rechnersystems Bezug nehmen. Wir sprechen deswegen von systemnaher Programmierung. Systemnahe Programmierung ist nicht mit Systemprogrammierung zu verwechseln. Systemprogrammierung benutzt die systemnahe Programmierung insbesondere zur Implementierung von Betriebssystemen oder ganz eng an Betriebssystem und Hardware angelehnter Softwarekomponenten wie Treiber, Prozesskommunikation usw.

Das wesentliche Merkmal der systemnahen Programmierung ist die direkte Kommunikation mit der Hardware und dem Betriebssystem eines Rechners. Diese erfordert fundamentale Kenntnisse über Architektur und Betriebssystem des Zielrechners. Umfangreiche Hochsprachen wie C++ oder Java abstrahieren von der Hardware eines Rechners, eine direkte Interaktion ist hier gerade nicht gewollt. Daher wird die systemnahe Programmierung meist in vergleichsweise minimalistischen Sprachen durchgeführt, entweder direkt in der Maschinensprache eines Rechners, die wir hier als Assembler bezeichnen, oder in C. C ist zwar ebenfalls eine Hochsprache, jedoch beinhaltet das Sprachkonzept von C viele Komponenten und Merkmale, die einen direkten Bezug zur Hardware haben. Große Teile der heute weit verbreiteten Betriebssysteme Windows und Linux sind in C programmiert.

Im ersten Studienbrief werden die aus Sicht eines Programmierers wesentlichen Prinzipien heutiger Rechnerarchitekturen und die allgemeinen Aufgaben von Betriebssystemen vorgestellt. Die dabei vorgestellten Prinzipien sind allgemeingültig und beziehen sich nicht auf eine reale Architektur.

Im zweiten Studienbrief beschäftigen wir uns mit einer ganz realen Rechnerarchitektur, auf der heute ein Großteil der Arbeitsplatzrechner aufbaut, nämlich mit IA-32. Diese wird detailliert vorgestellt, und die Assemblerprogrammierung dieser Architektur wird intensiv erlernt. Das hierbei erworbene Wissen kann relativ leicht auf andere Rechnerarchitekturen übertragen werden.

Der dritte Studienbrief beschäftigt sich mit der Programmiersprache C. Es handelt sich um einen kompletten Programmierkurs, in dem der gesamte Sprachumfang von C behandelt wird. C ist aufgrund gewisser Eigenheiten keineswegs eine leicht zu erlernende Sprache. Die Programmierung in C erfordert viel Übung, demzufolge ist gerade dieser Studienbrief mit sehr vielen praktischen Übungen angereichert.

Studienbrief 4 beschäftigt sich mit Sicherheitsaspekten und Sicherheitsproblemen. Gerade die Verwendung einer systemnahen Programmiersprache wie C gibt dem Programmierer zwar viele Freiheiten, birgt aber auch viele Risiken. Es wird gezeigt, wie Sicherheitslücken in Programmen entstehen und welche Auswirkungen sie haben können. Mit den erworbenen Kenntnissen in systemnaher Programmierung können diese Sicherheitsaspekte verstanden und beurteilt werden. Die Kenntnis darüber ist wichtig, um selbst Sicherheitsprobleme bei der eigenen Software-Entwicklung umgehen zu können.

Der abschließende Studienbrief beinhaltet ein kleineres Programmierprojekt in Assembler und ein größeres Programmierprojekt in C. Damit die Programmierprojekte einen wirklich praktischen Bezug haben, geben wir in diesem Studienbrief einen kleinen Einblick in relevante Algorithmen, die in der Informatik eine große Bedeutung haben. Bei beiden Programmierprojekten werden solche Algorithmen implementiert, um den Umgang mit Assembler und C zu festigen.

Dieses Modul kann einerseits als unabhängiges Grundlagenmodul betrachtet werden. Die erworbenen Kompetenzen sind vielfach verwendbar und nützlich. Gleichzeitig ist dieses Modul aber auch Basis des fortgeschrittenen Moduls „Reverse Engineering“. Reverse Engineering ist ein wichtiger Teilbereich der digitalen Forensik und beschäftigt sich mit der Analyse unbekannter Software. Die zu untersuchende Software liegt dabei in der Regel nur in Form von Maschinenprogrammen vor, weswegen tiefe Kenntnisse im Bereich systemnaher Programmierung für Reverse Engineering unabdingbar sind. Um das Modul „Reverse Engineering“ nicht mit Grundlagen überladen zu müssen, werden diese hier geschaffen, um sich dort gezielter und eingehender den fortgeschrittenen Techniken widmen zu können.

Viel Spaß und viel Erfolg!

Studienbrief 1 Rechnerstrukturen und Betriebssysteme

1.1 Lernergebnisse

Sie sind imstande, die allgemeinen Mechanismen bei der Abarbeitung von Programmen auf einer Von-Neumann-Rechnerarchitektur zu beschreiben. Sie können beschreiben, wie aktuelle Rechner prinzipiell aufgebaut sind. Sie können die grundlegenden Funktionseinheiten benennen und erklären. Sie können die fundamentalen Befehlsklassen darstellen, die ein Prozessor beherrschen muss, um universelle Programme ausführen zu können. Auch einige wichtige Architekturvarianten können Sie benennen. Schließlich sind Sie in der Lage, kleine Assemblerprogramme zu entwickeln.

Sie können die Kernfunktionen und Aufgaben eines Betriebssystems nennen und erklären. Die Begriffe Prozess, Adressraum und Programmierschnittstelle sind Ihnen vertraut.

1.2 Advance Organizer

Wenn Sie einen Rechner sehr nahe an der Hardware-Ebene programmieren wollen, dann benötigen Sie dazu detaillierte Kenntnisse über seinen inneren Aufbau, die Architektur des Rechners. Dies gilt ebenso, wenn Sie Binärcode, also Programme in der Maschinensprache eines Rechners verstehen und analysieren wollen. Die Notwendigkeit ein Maschinenprogramm zu analysieren tritt häufig bei der Wartung uralter Programme auf, aber auch bei aktueller Schad-Software (Malware), die einem Analysten aus nachvollziehbaren Gründen von ihren Autoren nicht als Programm einer höheren Programmiersprache zur Verfügung gestellt wird.

Kenntnisse über die Programmierung eines Rechners auf Maschinenebene, was auch Assemblerprogrammierung genannt wird, reichen allerdings oft auch noch nicht aus, um Maschinenprogramme erstellen und verstehen zu können. Das Bindeglied zwischen Hardware und Programm bildet das Betriebssystem, das einem Programm seine Ausführungsumgebung bereitstellt. Kenntnisse über das Betriebssystem eines Rechners sind demnach ebenfalls erforderlich.

In diesem Studienbrief erlernen Sie die allgemeinen, von einer konkreten Architektur unabhängigen Grundlagen einer maschinennahen Programmierung. Diese Grundlagen umfassen allgemeine Kenntnisse über den grundsätzlichen Aufbau heutiger Rechner, deren Arbeitsweise und die Grundfunktionen von Betriebssystemen. Diese Kenntnisse werden in den folgenden Studienbriefen und im auf diesem Modul aufbauenden Modul „Reverse Engineering“ für real existierende Rechner weiter vertieft.

1.3 Rechnerstrukturen und Assembler

Assembler ist im Gegensatz zu höheren Programmiersprachen aufwendig zu programmieren, zudem ist Assembler architektur- bzw. maschinenabhängig (prozessorspezifisch). Die Befehle eines Assemblerprogramms werden direkt auf der zugrunde liegenden Architektur ausgeführt und haben damit direkten Einfluss auf diese.

Der Begriff Assembler wird in zwei unterschiedlichen Bedeutungen benutzt:

D

Definition 1.1: **Assembler als Programmiersprache**

Assembler ist eine Programmiersprache, in der die Befehle eines Prozessors in symbolischer Form angegeben werden.

D

Definition 1.2: **Assembler als Übersetzer**

Assembler ist ein Übersetzer, der die symbolische Form der Maschinenbefehle eines Prozessors in binären Maschinencode umsetzt.

In diesem Modul wird der Begriff Assembler in der Regel im Sinne von Definition 1.1 benutzt, also als Programmiersprache.

Ein Rechner verarbeitet Programme in seinem spezifischen Maschinencode. Man sagt auch, dass solche Programme in der Maschinensprache des Rechners formuliert sind.

E

Exkurs 1.1: **Formale Sprachen**

Wie eine natürliche Sprache besteht eine formale Sprache wie z. B. die Maschinensprache eines Rechners aus ihren Sprachelementen und den Regeln, wie diese zusammengestellt werden dürfen. Ein in diesem Sinne gültiges Maschinenprogramm ist damit eine Folge von Maschinenbefehlen, deren Zusammenstellung den Regeln der Maschinensprache entspricht. Oder etwas formaler: Ein Maschinenprogramm ist ein Element der Menge aller Maschinenbefehlsfolgen, die mit den Regeln der zugrunde liegenden Maschinensprache erzeugbar sind.

Ein Maschinenprogramm besteht aus einer Folge von elementaren Maschinenbefehlen. Jeder Rechner verfügt über einen Satz unterschiedlicher Maschinenbefehle, die man zusammengefasst als Befehlssatz des Rechners bezeichnet.

Maschinencode

Ein Rechner kann ausschließlich Daten in binärer Form verarbeiten. Demnach ist ein Maschinenprogramm auf dieser tiefsten Ebene nichts anderes als eine Folge von Bitmustern. Jeder Befehl hat sein eigenes Bitmuster, das man auch als *Opcode* bezeichnet. Die Darstellung eines Maschinenprogramms als Bitmuster nennen wir Maschinencode.

Mnemonic

Maschinencode ist für den menschlichen Betrachter weitgehend unlesbar. Daher bedient man sich einer symbolischen Darstellung der Bitmuster. Jeder Befehl hat dabei ein eindeutiges, alphanumerisches Symbol, an das man sich erinnern kann. Deswegen werden diese Symbole auch als *Mnemonics* (altgriechisch: „mnemonika“ = Gedächtnis) bezeichnet. Der Befehl `mov` (Beispiel 1.1) symbolisiert bspw. eine Verschiebung (*to move*) von Daten.

Ein Assemblerprogramm ist die Darstellung eines Maschinenprogramms in dieser symbolischen Schreibweise. Häufig werden die Begriffe Assemblerprogramm, Maschinenprogramm und Maschinencode als Synonyme verwendet. Aus dem Kontext wird aber immer klar ersichtlich sein, in welchem Sinne diese eigentlich unterschiedlichen Begriffe gemeint sind.

Wir entwickeln in diesem und im folgenden vertiefenden Studienbrief Programme in Maschinensprache und bedienen uns dabei der Assemblerschreibweise. Daher sagen wir, wir schreiben die Programme in Assembler. Die entsprechenden Bitmuster des Maschinencodes interessieren uns dabei in der Regel nicht. Diese Umsetzung überlassen wir einem Assembler im Sinne von Definition 1.2.

Um bspw. fremden Maschinencode zu verstehen und um das Verhalten eines Programms zu analysieren, werden sogenannte *Disassembler* eingesetzt. Dies sind Tools, also Programmwerkzeuge, die binären Maschinencode in ihre symbolische Assemblerdarstellung umwandeln.

Disassembler

B

Beispiel 1.1

Der obere Teil ist ein kleines Assemblerprogramm. Unter der gestrichelten Linie ist der entsprechende Maschinencode in Hexadezimalschreibweise angefügt. Die Funktion des Programms spielt an dieser Stelle keine Rolle.

```
org 100h
start:
mov dx,eingabe    ; Aufforderung zum Zahl eingeben
mov ah,9h
int 021h          ; Ausgabe der Meldung

mov ah,07h        ; Wert ueber die Tastatur
int 021h          ; auf al einlesen

mov cx,1
mov bl,5          ; Farbe
mov ah,09h
int 010h

mov ah,4Ch        ; Ende
int 021h

section .data
eingabe: db 'Geben Sie ein Zeichen ein.', 13, 10, '\$'

--------------------------------
Maschinencode (hexadezimale Sequenz):
BA 18 01 B4 09 CD 21 B4 07 CD 21 B9 01 00 B3 05 B4 09 CD
10 B4 4C CD 21 47 65 62 65 6E 20 53 69 65 20 65 69 6E 20
5A 65 69 63 68 65 6E 20 65 69 6E 2E 0D 0A 24
```

Anm.: Die Textabschnitte hinter den Semikolons (;) sind Kommentare des Programmierers und gehören nicht zum eigentlichen Programm.

1.4 Rechnerarchitektur

In der realen Welt findet man Rechner der unterschiedlichsten Art, z. B. PCs, Mac, Großrechner, Workstations, Smartphones usw. Den Kern dieser Rechner bildet die

CPU[1] (*Central Processing Unit*) oder kurz Prozessor. Bekannte CPU-Hersteller sind bspw. Intel und AMD. Jede dieser CPUs verwendet eine andere Maschinensprache mit unterschiedlichen Befehlen. Demzufolge ist die Programmierung auf Maschinenebene CPU- bzw. prozessorspezifisch. Ein Assemblerprogramm für Prozessor *A* ist i. Allg. nicht auf Prozessor *B* ausführbar.

E

Exkurs 1.2: **Kompatibilität**

Die Realität ist manchmal noch verwirrender. So gibt es bspw. kompatible CPUs unterschiedlicher Hersteller, die in gleichartigen Rechnersystemen verbaut sind. In diesem Fall können durchaus Maschinenprogramme auf beiden Systemen lauffähig sein. Es seien hier Windows-Systeme genannt, in denen wahlweise CPUs der Hersteller Intel und AMD eingesetzt werden können.

Umgekehrt kann ein Windows-PC und ein Rechner der Firma Apple durchaus die gleiche CPU beinhalten, was aber keineswegs bedeutet, dass die Programme austauschbar sind. Neben anderen Unterschieden im Rechneraufbau ist hierfür in erster Linie das Betriebssystem verantwortlich.

Um Assemblerprogramme erstellen und verstehen zu können, muss die zugehörige Rechnerarchitektur, also der Aufbau des Rechners, prinzipiell bekannt sein. Prinzipiell meint, dass der Programmierer nicht alle technischen Details und Implementierungen der CPU kennen muss, sondern nur einige wichtige Grundgegebenheiten, die aus seiner Sicht wesentlich sind.

Trotz aller Unterschiede zwischen unterschiedlichen CPUs folgen die allermeisten CPUs heutzutage einem einheitlichen Architekturprinzip. Hat man dieses verstanden und gelernt, eine beliebige CPU auf Maschinenebene zu programmieren, so ist es meist mit mäßigem Zeitaufwand möglich, auch die Programmierung einer anderen CPU zu erlernen.

Es folgt nun ein kurzer, allgemeiner Einblick in die Architekturprinzipien heutiger CPUs mit Hinblick auf die für die Assemblerprogrammierung wichtigsten Komponenten.

1.4.1 Von-Neumann-Architektur

Die meisten der heutigen Rechner basieren auf dem Von-Neumann-Modell (Abb. 1.1). Demnach besteht ein Rechner aus der zentralen Recheneinheit (CPU), dem Speicherwerk (RAM/ROM/Festplatten) und dem Ein-/Ausgabewerk (Peripheriegeräte wie z. B. DVD-ROM, Tastatur, Maus, Bildschirm). Der Datenaustausch findet mittels eines bidirektionalen („bidirektional“ = senden und empfangen) Bussystems statt.

CPU

Die CPU bestimmt den Befehlssatz eines Rechners und besteht aus den folgenden Hauptkomponenten:

- Steuerwerk
- Rechenwerk
- Register

[1] Die Begriffe „CPU“ und „Prozessor“ werden meist synonym verwendet. Dagegen versteht man unter einem Mikroprozessor einen physikalischen Chip, der durchaus mehrere Prozessoren beinhalten kann.

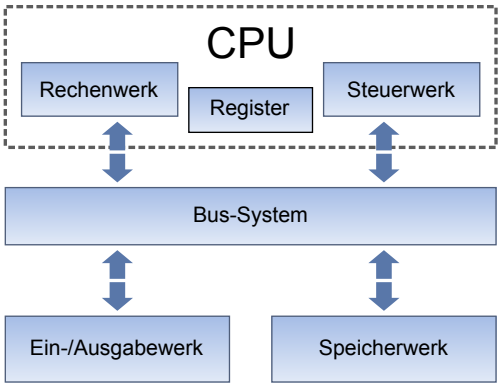


Abb. 1.1: Von-Neumann-Architektur

Kern der CPU ist das Steuerwerk. Es steuert die Abarbeitung und die Ausführung von Befehlen. Das Rechenwerk (ALU für *Arithmetic and Logic Unit*) führt die Elementaroperationen wie z. B. Addition oder Subtraktion aus. Register schließlich sind Speicherplätze, die sich im Prozessor befinden und direkt mit dem Steuerwerk und der ALU verbunden sind.

Speicher

Der gesamte Speicher eines modernen Rechners besteht aus einer Hierarchie unterschiedlicher Speicher, deren Geschwindigkeit mit der Nähe zur CPU zunimmt, während gleichzeitig die Größe abnimmt (Abb. 1.2).

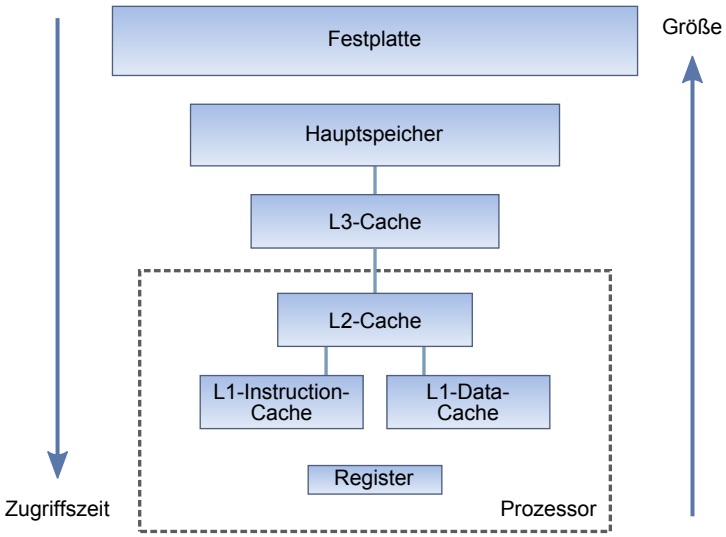


Abb. 1.2: Speicherhierarchie

Aus Sicht der Assemblerprogrammierung sind die Register und der Hauptspeicher am wichtigsten, da diese durch Prozessorbefehle direkt angesprochen werden können. In den Registern werden bspw. Operanden und Ergebnisse von arithmetischen Operationen abgelegt. Der Hauptspeicher dient zur Speicherung größerer Datenmengen und beinhaltet das auszuführende Programm.

Caches unterschiedlicher Level sind schnelle Zwischenspeicher (Puffer) für Teile des relativ langsamen Hauptspeichers. Sie sind für uns nicht von Interesse, da ihre Funktion transparent ist; die Cache-Verwaltung obliegt dem Rechner selbst. Physikalisch können Teile des Cache-Systems sowohl innerhalb als auch außerhalb der CPU liegen.

Bussystem

Die CPU kommuniziert über das Bussystem mit dem Speicherwerk und dem Ein-

/Ausgabewerk. Wesentlich sind hierbei Datenbus und Adressbus. Die Breite des Datenbusses bestimmt, wie viele Daten parallel in einem Schritt über den Datenbus transportiert werden können. Übliche Angaben für diese Breite sind 32 oder 64 Bit, was letztendlich der Anzahl der parallel verlaufenden „Kabel“ entspricht.

Die Breite des Adressbusses bestimmt, wie viele unterschiedliche Ziele von der CPU aus adressiert und damit selektiert werden können. Ziele sind bspw. Speicherzellen (in der Regel Bytes) des Hauptspeichers. Ein 32 Bit breiter Adressbus kann 4 GB = 2^{32} unterschiedliche Bytes adressieren. Man spricht auch vom physikalischen Adressraum der CPU.

K

Kontrollaufgabe 1.1

Wie groß ist der physikalische Adressraum eines 42 Bit breiten Adressbusses?

x-Bit-Prozessor

Man stößt häufig auf den Begriff *x*-Bit-Prozessor, wobei *x* heute typischerweise für 16, 32 oder 64 steht. Diese Zahl gibt an, wie „breit“ die Verarbeitungseinheiten der CPU sind, also aus wie vielen Bits die Daten, die die ALU in einem Befehl verarbeiten kann, maximal bestehen können. Eng verknüpft ist damit die Breite der Register. Zumindest die Universalregister entsprechen in ihrer Breite der Breite der ALU. Meistens entspricht die Breite des Datenbusses auch diesem Wert, es kann aber durchaus vorkommen, dass der interne und der externe Datenbus einer CPU unterschiedlich breit sind. Da die Adressierung des Hauptspeichers indirekt über Registerinhalte erfolgen kann, ist es naheliegend, dass auch der Adressbus eine Breite von *x* Bit haben sollte, dies ist aber nicht zwingend so. Eine durchgängige 32-Bit-Architektur ist die Intel 80386-CPU, mit der wir uns in Studienbrief 2 eingehend beschäftigen werden. Abb. 1.3 zeigt zusammengefasst die bei Prozessoren üblichen Datengrößen in Bits mit ihren Bezeichnungen.

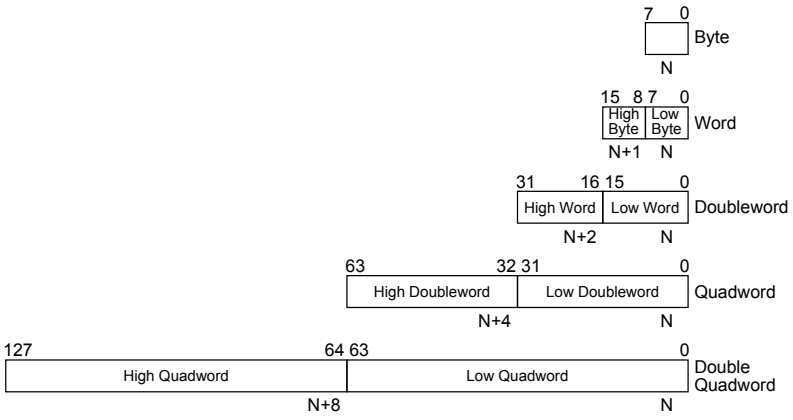


Abb. 1.3: Übliche Bezeichnungen für Datengrößen

Auf das Ein-/Ausgabewerk wird hier nicht gesondert eingegangen. Art und Ansprechart sind sehr unterschiedlich und auch weitgehend CPU-unabhängig. Zudem unterliegt der Zugriff meist dem Betriebssystem, sodass der Programmierer darauf nur indirekten Zugriff hat.

1.4.2 Programmausführung auf einer Von-Neumann-Architektur

Zu Beginn der Programmausführung liegen sowohl Programmcode als auch die Daten im Hauptspeicher. Bei der Von-Neumann-Architektur wird prinzipiell nicht

zwischen Programm- und Datenspeicher unterschieden, die Unterscheidung ergibt sich während der Programmausführung.

Instruction Pointer

Die CPU verfügt über ein spezielles Register, den sogenannte Befehlszähler (auch *Instruction Pointer* oder *Program Counter* genannt). Dieses beinhaltet die Adresse des nächsten auszuführenden Befehls im Hauptspeicher. Der Befehl wird aus dem Hauptspeicher ausgelesen und in einem Befehlsregister (*Instruction Register*) abgelegt.

Opcode

Ein CPU-Befehl besteht prinzipiell aus zwei Teilen, dem *Operation Code* oder kurz *Opcode* und dem Operandenteil. Der Opcode gibt an, welchen Befehl die CPU ausführen soll, z. B. eine Addition; im Operandenteil stehen die ggf. dazu benötigten Operanden.

B

Beispiel 1.2

Ein Additionsbefehl in Mnemonic-Darstellung könnte wie folgt aussehen:

```
ADD R3 R1,R2
```

Die Semantik dieses Befehls könnte die folgende sein:

Addiere die Inhalte von Register `R1` und Register `R2` und speichere das Ergebnis in Register `R3`.

Abb. 1.4 zeigt die wesentlichen, bei einer Befehlsausführung benötigten Komponenten einer Von-Neumann-CPU und den Datenfluss zwischen den einzelnen Komponenten.

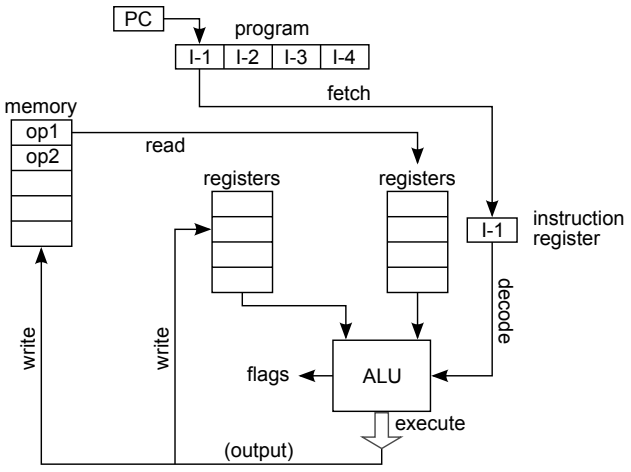


Abb. 1.4: Befehlsausführung bei der Von-Neumann-Architektur (Wisman [2012])

fetch-decode-execute-Schema

Nach dem Laden des Befehls in das Befehlsregister wird dieser von der CPU decodiert und ausgeführt. Diese Dreiteilung bei der Befehlsausführung wird auch als *fetch-decode-execute*-Schema bezeichnet. Danach wird i. Allg. der Befehlszähler inkrementiert, also in Abhängigkeit von der Befehlslänge im Speicher so erhöht, dass er auf den nächsten auszuführenden Befehl zeigt, und der Ablauf beginnt von vorne. Nur im Fall von Sprungbefehlen wird dieser sequentielle Ablauf unterbrochen.

Kernbefehlssatz

Um beliebige Programme auf einer Von-Neumann-Architektur des gezeigten Schemas ausführen zu können, ergeben sich mehr oder weniger zwangsläufig einige Befehlsarten, die den Kern des Befehlssatzes ausmachen. Diese Kernbefehlsarten finden sich bei jedem Prozessor des Von-Neumann-Typs wieder. Daher werden sie bereits an dieser Stelle kurz vorgestellt und mit Beispielen veranschaulicht.

1.4.2.0.1 Datentransferbefehle

Datentransferbefehle bewegen Daten innerhalb der CPU bzw. vom Hauptspeicher in die CPU oder von der CPU in den Hauptspeicher. Einige typische Beispiele mit Erklärungen sind hier aufgelistet:[2]

```
mov  R1, R2     ; kopiert Register R2 nach Register R1
mov  R1, [1234] ; kopiert den Inhalt von Hauptspeicheradresse
                  1234 nach Register R1
mov  [1234], R2 ; kopiert Register R2 in den Hauptspeicher
                  an Adresse 1234
mov  R1, 4711   ; laedt die Konstante 4711 in Register R1
```

Beim Datentransfer zwischen Registern bzw. Register und Hauptspeicher werden die Daten kopiert, das Original bleibt also erhalten.

1.4.2.0.2 Arithmetische und logische Operationen

Dies ist die Klasse der eigentlichen Rechenoperationen der CPU. Die ausführbaren Operationen entsprechen weitgehend den in der ALU hardwaremäßig vorhandenen Schaltungen wie Addierwerk, Multiplizierwerk, Shifter (verschiebt einen Registerinhalt um eine Anzahl Bits nach links oder rechts) usw. Typische Beispiele für solche Befehle sind:

```
add  R1, R2     ; addiere die Register R1 und R2
                  und ueberschreibe R1 mit dem Ergebnis
mul  R1, R2     ; multipliziere R1 und R2 in gleicher Weise
shl  R1, 1      ; shifte den Inhalt von R1 um 1 Bit nach
                  links
and  R1, R2     ; verknuepfe R1 und R2 bitweise mit AND
                  und ueberschreibe R1 mit dem Ergebnis
```

1.4.2.0.3 Kontrollflussbefehle

Von einem Wechsel des Kontrollflusses spricht man, wenn der Befehlszähler nach der Befehlsausführung nicht einfach inkrementiert wird, sondern einen anderen Wert annimmt, also auf eine andere Stelle im Programmcode zeigt. Dies bezeichnet man als Sprung (*Jump*).

bedingte und unbedingte Sprünge

Grundsätzlich werden bedingte und unbedingte Sprünge unterschieden. Unbedingte Sprünge werden immer ausgeführt, bedingte Sprünge nur dann, wenn irgendein Ereignis eintritt. Im engen Zusammenhang zu bedingten Sprüngen stehen Vergleichsbefehle.

Typische Kontrollflussbefehle sehen wie folgt aus:

[2] Die Schreibweise dieser Befehle orientiert sich an der Konvention bei Intel x86-Assembler: Der Zieloperand steht links, Speicheradressen stehen in eckigen Klammern.

```
jmp  1000       ; setze das Programm an Adresse 1000 fort
;
cmp  R1, R2     ; vergleiche Register R1 und R2
jge  2000       ; setze das Programm an Adresse 2000 fort,
                  falls R1 >= R2 ist (jump if greater or equal)
                  ansonsten fuehre Folgebefehl aus
```

`jmp 1000` ist ein unbedingter Sprung. `jge 2000` hingegen ist ein bedingter Sprung. Ihm geht der Vergleichsbefehl `cmp` voraus. Register `R1` und `R2` werden miteinander verglichen. Der bedingte Sprung bezieht sich auf das Ergebnis dieses Vergleichs.

1.4.3 Architekturvarianten

Innerhalb der Von-Neumann-Architektur existieren einige Entwurfsphilosophien, die zwar am grundlegenden Prinzip nichts ändern, aber ein fundamentales Unterscheidungsmerkmal bei CPUs darstellen. Zwei wichtige Varianten sollen hier vorgestellt werden.

1.4.3.1 Load-Store-Architekturen

Bei Load-Store-Architekturen gibt es dedizierte Lade- und Schreibbefehle für den Hauptspeicherzugriff. Alle anderen Befehle arbeiten ausschließlich auf Registern.

Bei einer Nicht-Load-Store-Architektur wäre bspw. ein solcher Befehl denkbar:

```
add R1, [1234]  ; addiere Register R1 und den Inhalt vom
                  Hauptspeicher an Adresse 1234
```

Bei einer Load-Store-Architektur wären hierfür zwei Befehle notwendig:

```
mov R2, [1234]  ; lade R2 aus Speicheradresse 1234
add R1, R2      ; addiere Register R1 und R2
```

Pipelining

Neben dem offensichtlichen Nachteil eines längeren Programmcodes haben Load-Store-Architekturen den Vorteil, dass sie den Hauptspeicherzugriff von den CPU-internen Vorgängen abkoppeln. Dies kann zu einer höheren Verarbeitungsgeschwindigkeit führen, insbesondere im Zusammenhang mit sogenanntem *Pipelining*, der überlappenden Befehlsausführung. Speicheroperanden können bzw. müssen aus dem relativ langsamen Hauptspeicher frühzeitig geladen werden. Bei der Abarbeitung der CPU-internen (Register-)Befehle entfällt die Wartezeit auf diese Operanden. Die Prozessorarchitektur ARM[3] ist ein Beispiel für eine Load-Store-Architektur.

1.4.3.2 CISC und RISC

CISC

CISC steht für *Complex Instruction Set Computer*. CISC-Architekturen zeichnen sich dadurch aus, dass sie wenige universelle Register und viele Spezialregister haben. Dies ist eine Folge des Befehlssatzes, der bei CISC aus vielen verschiedenen und

[3] Details sind bspw. unter
http://arm.com/products/processors/instruction-set-architectures/index.php
zu finden.

komplexen Befehlen besteht. Ein komplexer Befehl wäre bspw. ein Kopierbefehl, der einen kompletten Speicherbereich kopieren kann und dabei ein spezielles Zählregister verwendet.

Der Vorteil von CISC-Befehlen liegt auf der Hand: Der Maschinencode wird kompakt und für den Menschen eventuell leichter lesbar. Der Nachteil ist die schwerere Erlernbarkeit des Befehlsumfangs. Zudem gibt es bei CISC-Prozessoren meist keine einheitliche Länge der codierten Maschinenbefehle.

RISC RISC-Prozessoren gehen einen anderen Weg, nämlich den der Beschränkung auf das Wesentliche: Es existieren nur die notwendigsten Befehle, dafür aber viele Universalregister, und die Länge der Befehlscodierung ist einheitlich (z. B. 32 Bit). RISC steht für *Reduced Instruction Set Computer*.

Es lässt sich kein Urteil fällen, welche der beiden Architekturvarianten die bessere ist. In der Geschichte der Prozessortechnik schlug das Pendel schon mehrfach in die eine oder andere Richtung aus. Viele moderne Prozessoren kombinieren ohnehin beide Konzepte.

E

Exkurs 1.3: **Kurzhistorie von CISC und RISC**

Etwas lapidar könnte man behaupten, dass RISC näher am physikalischen Aufbau einer CPU orientiert ist. Die Historie zeichnet allerdings ein etwas anderes Bild.

Die ersten Mikroprozessoren (z. B. 8080, Z80, 8085, 8086) waren eher CISC-Prozessoren. Den Anstoß für den RISC-Gedanken gaben zunächst die Compilerbauer, denn es konnte beobachtet werden, dass die von Compilern erzeugten Maschinenprogramme nur eine (kleine) Teilmenge des Befehlssatzes einer CPU nutzten. Zu diesem Zeitpunkt war die Entwicklung großer Assemblerprogramme bereits in den Hintergrund getreten, die Hochsprachen dominierten das Feld der Software-Entwicklung.

Der Grundgedanke von RISC ist leicht nachzuvollziehen: Wenn ohnehin nur wenige und auch eher die nicht komplexen Maschinenbefehle genutzt werden, dann ist es naheliegend, die CPU danach zu optimieren. Die einfache und einheitliche Befehlsstruktur ermöglicht es, einfachere und damit kleinere CPUs zu bauen, da weniger Komplexität auf die Chips integriert werden muss. Komplexität bedeutet hierbei letztendlich die Anzahl der Transistoren auf dem Chip.

Klassische Vertreter des RISC-Gedankens sind ARM, Sun SPARC[4] und IBM PowerPC[5]. Aber auch in klassische Vertreter der CISC-Kultur, wie der Intel x86-Prozessorfamilie, floss der RISC-Gedanke ein, indem um einen eigentlichen RISC-Kern die komplexeren CISC-Befehle, die aus Kompatibilitätsgründen erhalten bleiben mussten, in einer Kombination aus Preprocessing und Mikrocode in eine Folge aus einfachen CPU-Befehlen aufgelöst wurden.

Da RISC-Prozessoren bzw. die RISC-Kerne hybrider Prozessoren klein und kompakt sind und eine einheitliche Befehlsstruktur aufweisen, die zu einer schnellen Befehlsdecodierung führt, können diese schneller getaktet werden als komplexe CISC-Prozessoren. Auf den ersten Blick ist das ein enormer Vorteil, denn schnellere Taktung bedeutet schnellere Programmausführung.

Intel brachte im Jahr 2000 den Pentium 4 auf Basis einer RISC-Architektur namens NetBurst auf den Markt. Bis etwa 2005 benutzten die Intel-Prozessoren diese Architektur, und es wurden mit dem sogenannten Prescott-Kern Taktraten von bis zu 4GHz erreicht. Was die Entwickler wohl nicht so genau bedacht hatten: Schnellere Taktung führt halbleitertechnisch bedingt zu einer immer höheren Verlustleistung. Diese Prozessoren waren kaum noch zu kühlen. Nie jaulten die PCs lauter als im Jahr 2005, weil immer leistungsfähigere Lüfter zur CPU-Kühlung verbaut werden mussten.

Die immer weitere Steigerung der Taktrate war eine technologische Sackgasse. Die (Intel-)Prozessoren gingen nun einen anderen Weg: SIMD und Multi-Core.

SIMD (*Single Instruction, Multiple Data*) bedeutet, dass in einem Befehl auf einem Prozessor mehrere, auch gleichartige Rechenoperationen ausgeführt werden können, wenn z. B. mehrere Addierwerke vorhanden sind.

Multi-Core bedeutet, dass auf einem physikalischen Prozessorchip mehrere Prozessorkerne vorhanden sind, die gleichzeitig unterschiedliche Programme abarbeiten können.

Die Multi-Core-Prozessoren mit SIMD-Eigenschaften dominieren heute den PC-Markt. Aber sind diese Prozessoren nun RISC oder CISC? Sie sind beides!

1.5 Betriebssysteme

Anwenderprogramme werden auf modernen Rechnern nicht unmittelbar „auf der Hardware“ ausgeführt. Vielmehr existiert eine Instanz, die dem Benutzer die Bedienung seines Rechners und die Ausführung von Programmen ermöglicht. Diese Instanz bezeichnet man als Betriebssystem. Das Betriebssystem bietet also die Umgebung an, in der Anwendungen ausgeführt werden können.

Das Betriebssystem sorgt einerseits für eine Ordnung bei der Ausführung von Programmen, und andererseits entkoppelt es die Anwendungen von der konkreten Hardware.

Ordnung

Unter den Begriff „Ordnung“ fallen hierbei verschiedene Aspekte wie die Ablage und Verwaltung von Programmen und Daten auf der Festplatte, die Ausführungsreihenfolge von Programmen bei Multiuser-/Multitasking-Rechnern (*Scheduling*), die Benutzerschnittstelle usw. Auf diese Aspekte eines Betriebssystems wollen wir in diesem Studienbrief allerdings nur am Rande eingehen.

Abstraktion

Die zweite wesentliche Aufgabe eines Betriebssystems ist die Abstraktion. Die konkrete Implementierung des Rechners auf Hardware-Ebene sowie der direkte Zugriff auf diese Ebene sollen i. Allg. vor dem Anwender und den Anwendungen verborgen bleiben. Die Benutzbarkeit dieser Komponenten ohne Hardware-Zugriff und Hardware-Verständnis sind Ziel der Abstraktion.

[4] http://de.wikipedia.org/wiki/Sun_SPARC
[5] http://de.wikipedia.org/wiki/PowerPC

B

Beispiel 1.3

Einen Anwender interessiert es i. Allg. nicht, welche Grafikkarte konkret in einem Rechner eingebaut ist und wie sie genau funktioniert. Auch für die meisten Anwendungen spielt dies keine Rolle. Das Betriebssystem stellt daher nur eine Anzahl von Funktionen zur Verfügung, die es erlauben, die Grafikkarte zu benutzen. Solchen Funktionen schalten bspw. einzelne Pixel auf dem Bildschirm an oder zeichnen Geraden.

Logisch bildet das Betriebssystem verschiedene Schichten zwischen Anwendungen und der zugrunde liegenden Hardware. Abbildung 1.5 verdeutlicht die Rolle des Betriebssystems und zeigt schematisch eine Aufteilung in vier Schichten. Diese Schichten kommunizieren über genau definierte Schnittstellen miteinander. Das grundsätzliche Verständnis dieses Schichtenmodells und der Kommunikationsschnittstellen ist für das Verständnis eines Programmverhaltens unbedingt erforderlich.

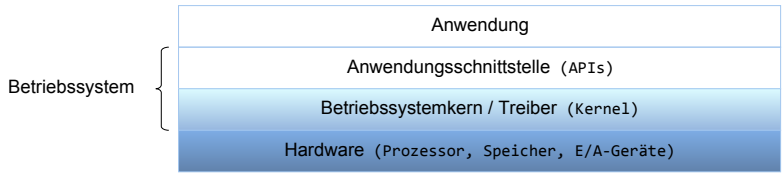


Abb. 1.5: Schichtenmodell eines Betriebssystems

Hardware Die unterste Schicht im Modell bildet die Hardware. Es ist in der Regel nicht erwünscht, dass Anwendungen direkt mit dieser Ebene kommunizieren können.

Betriebssystemkern Auf der Hardware baut der Betriebssystemkern, auch *Kernel* genannt, auf. Hier sind die elementaren Steuerungsfunktionen des Betriebssystems und die sogenannten Treiber implementiert. Als Treiber bezeichnet man die Programme, die die Interaktion mit den Hardware-Komponenten steuern (Grafikkartentreiber usw.).

Anwendungsschnittstelle Die Vermittlung zwischen dem Betriebssystemkern und den Anwendungen übernimmt die Anwendungsschnittstelle , auch API (*Application Programming Interface*) genannt. Über die API kommunizieren die Anwendungen mit dem Betriebssystemkern. Diese Schnittstelle muss klar und eindeutig definiert sein. Anwendungen rufen von der API bereitgestellte Funktionen auf, z. B. zur Ansteuerung der Grafikkarte. Ändert sich die Grafikkarte oder der Treiber für diese, so sollte das für die Anwendung transparent sein.

Privilegierung & Speicherverwaltung Weitere wesentliche Aufgaben eines Betriebssystems sind die Privilegierung und die Speicherverwaltung . Es soll nicht jedem Programm erlaubt sein, alle Ressourcen eines Rechners zu nutzen, also uneingeschränkten Zugriff auf alle Komponenten eines Rechners zu erhalten. Dazu werden Privilegienstufen, sogenannte Schichten oder Ringe, definiert, die den einzelnen Programmen zugeordnet werden. Normale Anwenderprogramme laufen hierbei in einer niedrigen Privilegienstufe, Betriebssystemprogramme in einer hohen. Insbesondere soll der Speicherbereich hochprivilegierter Programme vor Anwenderprogrammen geschützt werden, aber auch die Speicherbereiche unterschiedlicher Anwenderprogramme vor gegenseitigem Zugriff. Bei modernen Prozessoren werden diese Privilegienstufen hardwareseitig unterstützt. In der Intel-x86-Prozessorfamilie existieren ab dem 80286 vier Ringe, wobei Ring 0 die höchste und Ring 3 die niedrigste Privilegienstufe darstellen.

Zusammengefasst lassen sich die wesentlichen Aufgaben eines Betriebssystems wie folgt beschrieben:

- *Einschränkung* des Zugriffs auf die Hardware
- Definition und Einhaltung von Sicherheitsrichtlinien (*Schutz*)
- Aufrechterhaltung der Systemstabilität bei Programmierfehlern (*Stabilität*)
- Verhinderung der Monopolisierung von Ressourcen (*Fairness*)
- Schaffung weniger, einheitlicher Hardware-Schnittstellen (*Abstraktion*)

Dazu kommen die Verwaltungsfunktionen sowie die Benutzerschnittstellen und Bedienelemente, die die Nutzbarkeit des Systems sicherstellen.

1.5.1 Grundbegriffe

Zumindest Teile des Betriebssystems müssen mit vollen Privilegien ausgeführt werden, damit z. B. die Register für die Speicherverwaltung bearbeitet werden können. Beim Ringsystem der erwähnten Intel-Prozessoren läuft dann der Betriebssystemkern in Ring 0, weniger kritische Betriebssystemteile und die eigentlichen Anwendungen in Ringen mit niedrigeren Privilegien, um die Adressbereiche des Kerns zu schützen. Moderne Betriebssysteme wie Windows oder Linux verwenden nur zwei der vier möglichen Ringe, Ring 0 und Ring 3. Damit ist die Gesamtheit aller Programme in zwei Bereiche aufgeteilt: User- und Kernel-Bereich.

Bem.: *Alle Programme, die im Rahmen dieses Moduls entwickelt und analysiert werden, liegen im Userbereich.*

Etwas genauer werden die Oberbegriffe User-Bereich und Kernel-Bereich wie folgt unterteilt und definiert:

D

Definition 1.3: **Userland, Usermode, Userspace**

- **Userland**
  Als Userland bezeichnet man die Menge aller ausführbaren Dateien, die mit eingeschränkten Rechten laufen. Dazu zählen die Anwendungen, bestimmte (unkritische) Dienste und Bibliotheken. (Das Windows Userland ist bspw. die Menge aller *exe*- und *dll*-Dateien.)
- **Usermode**
  Der Usermode ist eine Privilegienstufe der CPU mit eingeschränkten Rechten. Jeder Codeausführung ist ein Privilegienring zugeordnet. (Ausblick: Bei der in Studienbrief 2 behandelten Intel IA-32-Architektur wird in den unteren zwei Bits der Segmentregister angegeben, welchem Ring die Segmente zugeordnet sind. Sind diese Bits auf `11` gesetzt, so steht das für Usermode (Ring 3), `00` steht für Kernelmode (Ring 0).)
- **Userspace**
  Der Userspace gibt den Adressbereich an, auf den unterprivilegierte Programme zugreifen dürfen.

K

Kontrollaufgabe 1.2

Definieren Sie in gleicher Weise die Begriffe **Kernelland**, **Kernelmode** und **Kernelspace**.

1.5.2 Prozesse, Threads und Loader

1.5.2.1 Prozesse

Bisher sprachen wir meist von Programmen und Anwendungen, wobei eine Anwendung die konkrete Ausführung eines Programms darstellt. Im Zusammenhang mit Betriebssystemen ist der Begriff *Prozess* üblich. Ein Prozess ist die konkrete Ausführung eines Programms auf einem Prozessor in einem vom Betriebssystem bereitgestellten Umfeld. Dieses Umfeld wird auch als Prozesskontext bezeichnet.

Prozesskontext

Der Prozesskontext wird in Hardware-Kontext und Software-Kontext unterteilt. Der Hardware-Kontext besteht aus dem Prozessor (bzw. den Prozessoren), der dem Prozess zur Ausführung des Programmcodes bereitgestellt wird und dem Adressraum, in dem das Programm ausgeführt und Daten geschrieben bzw. gelesen werden können. Diesen Adressraum erhält der Prozess exklusiv, was ihn gegenüber anderen Prozessen abschottet. Der Software-Kontext enthält alle Verwaltungsinformationen des Prozesses, z. B. seine Identifikationsnummer, Adressrauminformationen usw.

Der Programmablauf selbst kann sich in voneinander unabhängige Teile aufspalten, die dynamisch während eines Prozesses entstehen. Diese sogenannten *Threads* sind dann parallele Teilprogrammausführungen, die hardwareseitig tatsächlich parallel, d. h. gleichzeitig, auf unterschiedlichen Prozessoren laufen können.

Ein Prozess besteht zusammengefasst aus den folgenden Komponenten:

1. ausführbarer Programmcode
2. eigener Adressraum im Speicher
3. zumindest ein Thread
4. Verwaltungsstruktur

1.5.2.2 Threads

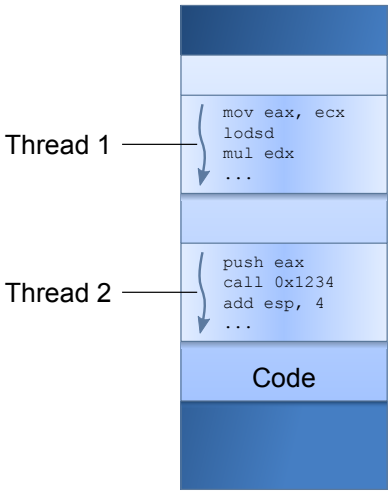
Ein Thread ist ein Ausführungsstrang von Maschinenbefehlen. Threads entstehen innerhalb von Prozessen, und jeder Thread ist immer einem Prozess zugeordnet. Ein Prozess kann mehrere Threads besitzen. Über bestimmte Funktionen kann ein Prozess neue Threads initialisieren oder bestehende Threads beenden. Am Anfang hat jeder Prozess jedoch genau einen Thread.

Moderne Rechner haben in der Regel mehrere CPUs bzw. CPU-Kerne, wodurch nicht nur Prozesse, sondern auch Threads gleichzeitig und parallel ausgeführt werden können (Abb. 1.6). Eine CPU arbeitet Thread 1 ab, während gleichzeitig dazu eine andere CPU Thread 2 abarbeitet.

Jeder Thread besitzt einen eigenen *Stack*. Ein Stack[6] ist dabei ein Teil des Hauptspeichers, in dem lokale Daten eines Thread gehalten werden. Gäbe es nur einen

[6] Aufbau und Verwaltung von Stacks werden in Studienbrief 2 eingehend betrachtet.

Abb. 1.6: 2 Gleichzeitig ablaufende Threads



Stack, so würden sich unabhängig voneinander laufende Threads gegenseitig den Stack zerstören.

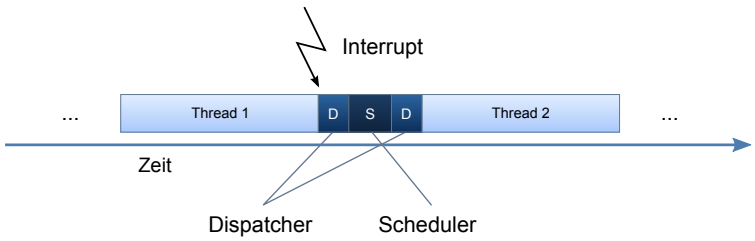
Scheduler

Hat ein Prozess mehr Threads als physikalisch Prozessoren vorhanden sind, dann muss es eine Instanz geben, die entscheidet, welcher Thread wann auf welcher CPU ausgeführt wird. Diese Entscheidung trifft der *Scheduler*, der eine zentrale Rolle innerhalb des Betriebssystems spielt.

Dispatcher

Den Thread-Wechsel, auch Kontextwechsel genannt, erledigt der *Dispatcher*. Der Dispatcher ist für das Laden/Entladen von Threads zuständig, insbesondere für die Sicherung/Wiederherstellung von Registerinhalten (Thread-Kontext). Abb 1.7 zeigt den zeitlichen Ablauf bei einem Thread-Wechsel und die Interaktion von Scheduler und Dispatcher.

Abb. 1.7: Scheduler und Dispatcher



Der Scheduler wird in gewissen Situationen aufgerufen, z. B. bei synchronen (langsamen) Schreib-/Leseoperationen auf die Festplatte (synchroner *Interrupt*) oder freiwilligen Unterbrechungen (*Sleep*) oder bei Unterbrechungen, die nicht aus dem aktuellem Programmfluss hervorgehen, sondern von extern erzwungen werden (asynchrone Interrupts). Zu letzteren zählen auch die Timing Interrupts, die periodisch generiert werden, um die Monopolisierung der CPU durch einen einzelnen Thread zu verhindern. Die Abfolge der Threads bei periodischen Interrupts werden durch das Scheduling-Verfahren bestimmt, das eine möglichst faire Verteilung der Rechenzeit auf die einzelnen Threads gewährleisten soll. Abb. 1.8 zeigt beispielhaft die stückweise Abarbeitung von Threads bei verschiedenen synchronen und asynchronen Interrupts.

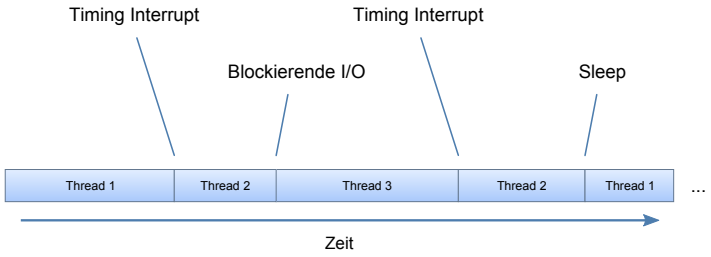


Abb. 1.8: Scheduling dreier Threads auf einer CPU

1.5.2.3 Loader

Beim Start eines Programms muss ein neuer Prozess initiiert werden. Diese Aufgabe übernimmt der *Loader*. Der Loader erstellt einen Adressraum für den Prozess und lädt den eigentlichen Programmcode und die Programmdaten in den Speicher. Des Weiteren erstellt der Loader die Verwaltungs- oder Datenstrukturen des Prozesses und registriert diesen bei anderen Betriebssystemkomponenten, wie z. B. dem Scheduler. Schließlich erstellt der Loader den initialen Thread entsprechend des Einstiegspunktes des Programms. Danach ist der Prozess einsatzbereit, und die eigentliche Programmausführung kann beginnen.

1.5.2.4 Datenstrukturen

Die Verwaltungsinformationen zu aktiven Prozessen und Threads werden in speziellen Datenstrukturen hinterlegt. Diese nennt man PEB (*Process Environment Block*) und TEB (*Thread Environment Block*).

Im PEB werden prozessspezifische Informationen abgelegt. Dazu gehört die Prozess-ID, über die der Prozess eindeutig identifizierbar ist. Des Weiteren sind im PEB die Adressrauminformationen abgelegt.[7]

Der TEB beinhaltet unter anderem die Thread-ID, über die der Thread identifiziert und angesprochen/beendet werden kann. Daneben sind im TEB Stack-Informationen (z. B. die Größe und die Startadresse) und sonstige Verwaltungsinformationen hinterlegt.

Thread-Kontext

Vom TEB ist der eigentliche Thread-Kontext zu unterscheiden. Der Thread-Kontext ist dynamisch und besteht aus den aktuellen Registerwerten eines Thread. Wird ein Thread unterbrochen, so muss der Thread-Kontext gesichert werden, um ihn bei der Reaktivierung des Thread wiederherstellen zu können.

1.5.3 Adressräume

Die beiden grundlegenden Ressourcen eines Rechners sind Prozessorzeit und Speicher. Die wichtigste Randbedingung bei der Verwendung der Ressource Speicher ist es, eine Isolation der Anwendungen untereinander zu gewährleisten, um unerwünschte wechselseitige Beeinflussungen auszuschließen. Deshalb besitzt jeder Prozess seinen eigenen Adressraum. Gewollte Interaktionen zwischen Anwendungen müssen in kontrollierter Weise erfolgen.

Virtualisierung

Die Virtualisierung des Adressraums erlaubt es, dass jeder Prozesse virtuell den kompletten Adressbereich der Architektur verwenden kann. Aus Prozesssicht ist sein Adressraum homogen, d. h. zusammenhängend. Für die Trennung der

[7] Dazu gehören insbesondere bei IA-32-basierten Betriebssystemen auch Verweise auf Page Table und Page Directory, die in Abs. 2.3.2.7 vorgestellt werden.

den Prozessen tatsächlich physikalisch zugeteilten Speicherbereiche ist die Speicherverwaltung zuständig, die für den Prozess „unsichtbar“ – man sagt auch „transparent“ – im Hintergrund arbeitet. Dieses Verfahren sorgt auch dafür, dass unerlaubte Zugriffe auf die Speicherbereiche anderer Prozesse oder auf privilegierte Speicherbereiche automatisch geblockt werden. Insbesondere wirken sich so Programmierfehler innerhalb eines Prozesses nur auf den Prozess selbst aus und gefährden nicht die Stabilität des Gesamtsystems. Der virtuelle Adressraum wird in der Regel vom Betriebssystem in zwei Teile geteilt, den Userspace und den Kernelspace. Bei 32-Bit-Windows belegt der Userspace die beiden unteren GB des virtuellen Adressraums, der Kernelspace die beiden oberen. Prozessbezogene Daten (Code, Daten, Heap, Stack etc.) befinden sich im Userspace, Kernel und Treiber arbeiten im Kernelspace, d. h. im Kernelspace liegen Kernel-Code, Kernel-Daten, Kernel-Stack sowie diverse kritische Datenstrukturen.

Kernelspace

Aus einer Anwendung heraus können keine beliebigen Adressen im Kernelspace angesprochen werden. Dadurch werden alle Prozesse im Kernelspace vor Manipulationen durch Anwendungen geschützt. Allerdings kann das Betriebssystem gezielt Teile des Kernelspace in den Adressraum einer Anwendung „einblenden“, um bspw. eine schnelle Datenkommunikation zu ermöglichen. Solche Einblendungen stehen aber unter der alleinigen Kontrolle des Betriebssystemkerns.

Die von Kernel-Prozessen benutzten Adressräume sind keineswegs so strikt voneinander getrennt wie die von Anwenderprozessen. Es existieren vielmehr Datenstrukturen im Kernelspace, die von vielen Kernel-Prozessen gemeinsam genutzt werden. Es ist daher leicht vorstellbar, dass unkontrollierte Zugriffe auf den Kernelspace verheerende Auswirkungen auf das Gesamtsystem haben können.

Userspace

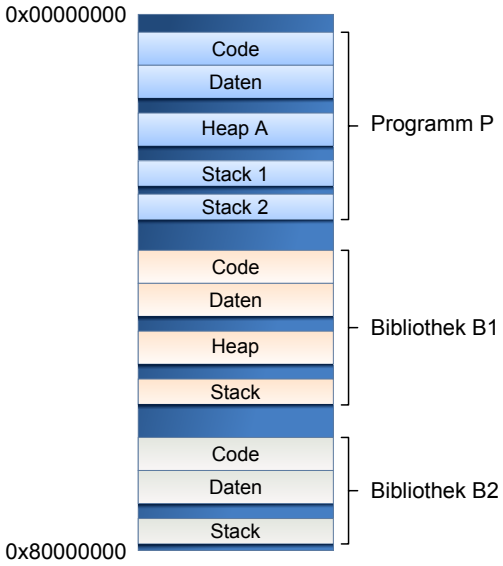
Beim Starten eines Programms im Userspace, also bei der Prozesserzeugung, werden das eigentliche Programm und seine Abhängigkeiten in den Speicher geladen. Die Abhängigkeiten sind insbesondere Bibliotheksfunktionen, die das Programm benutzen will. Die Bereiche, die dadurch im Userspace belegt werden, können in vier Gruppen untergliedert werden:

1. Die *Codebereiche* enthalten den eigentlichen Maschinencode des Programms bzw. von Bibliotheksfunktionen.
2. Die *Datenbereiche* enthalten die statischen Daten, also z. B. globale Variablen und Konstanten, die bereits zur Kompilierungszeit bekannt sind.
3. Die *Stack-Bereiche* enthalten die Prozeduraufrufdaten und lokale Variablen. Jedem Thread ist genau ein Stack-Bereich zugeordnet.
4. Die *Heapbereiche* enthalten globale Daten, die allerdings erst dynamisch während der Programmausführung entstehen und deren Umfang zur Kompilierungszeit nicht bekannt ist.

Abb. 1.9 zeigt beispielhaft die Lage dieser vier Bereiche im Speicher während einer Prozessverarbeitung. Der Prozess führt Programm P aus. P benutzt zwei Bibliotheken $B1$ und $B2$. P, $B1$ und $B2$ erzeugen eigene Code- und Datenbereiche, P und $B1$ verwenden einen Heap, $B2$ nicht. P benutzt zwei Stack-Bereiche, also existieren zu diesem Zeitpunkt zwei Threads, die Bibliotheken bestehen jeweils nur aus einem Thread.

Stack- und Heap-Bereiche sind dynamisch, sie wachsen also zur Laufzeit. Aus diesem Grund sollten sie im Adressraum eines Prozesses möglichst weit voneinander entfernt platziert werden und aufeinander zuwachsen.

Abb. 1.9: Bereiche im Userspace



1.5.4 Programmierschnittstellen

Anwendungen aus dem Userspace können und dürfen nicht auf den Kernelspace zugreifen. Andererseits bieten Prozesse im Kernelspace Funktionen an, die für das Ausführen von Anwendungen auf dem Rechner notwendig sind, z. B. Ein-/Ausgabefunktionen für Tastatur und Bildschirm.

API

Um sowohl den Anforderungen nach Stabilität der Kernelspace-Prozesse als auch den Bedürfnissen der Userspace-Prozesse nach Interaktion mit Betriebssystemprozessen gerecht zu werden, erfolgt die Kommunikation dieser beiden Ebenen über wohldefinierte Programmierschnittstellen, die APIs (*Application Programming Interfaces*). Dahinter verbergen sich oftmals mehrere Schichten interner Betriebssystemschnittstellen, von denen eine Anwendung keine Kenntnis haben muss. Diese internen Schnittstellen sind auch oftmals undokumentiert.

Nicht jeder Aufruf der API führt zwangsläufig zu einer Kommunikation mit dem privilegierten Kernel. Je nachdem, welche Privilegienstufe zum Abarbeiten einer API-Anfrage benötigt wird, wird die Anfrage außerhalb oder innerhalb des Kernel bearbeitet.

B

Beispiel 1.4

Man stelle sich eine fiktive API-Funktion *GetTime* vor, deren Aufruf die aktuelle Systemzeit liefern soll. Es ist vermutlich nicht erforderlich, das reine Auslesen der Systemuhr an einen privilegierten Prozess zu binden. Demzufolge könnte *GetTime* auch im Usermode laufen.

Funktionen, die im weitesten Sinne das System manipulieren, werden sicherlich im Kernelmode ausgeführt. Der Übergang vom Usermode in den Kernelmode erfolgt durch Systemaufrufe, sogenannte *Syscalls*.

Abb. 1.10 zeigt schematisch den Ablauf beim Aufrufen einer API-Funktion *WriteFile*, die schreibenden Zugriff auf einen Datenträger gewährt. Das Beispiel ist der Windows-Welt entnommen. Zur Bedienung des Aufrufs werden - wie oben erwähnt - mehrere interne Schnittstellen zwischen den Standard-Windows-

Bibliotheken durchlaufen. Die Bibliotheken *kernel32.dll* und *ntdll.dll* arbeiten dabei noch im Userspace. Der Übergang in den Kernelspace erfolgt durch den internen Aufruf des Systemprozesses *ntoskrnl.exe* mittels eines Interrupts. Nach Ausführung des eigentlichen Schreibbefehls erfolgt die Rückgabe eines Ergebnisstatus in umgekehrter Richtung hin zum Anwendungsprogramm. Der Kernelspace wird mit dem Übergang nach *ntdll.dll* verlassen, was die Interrupt-Behandlung und damit den Syscall beendet.

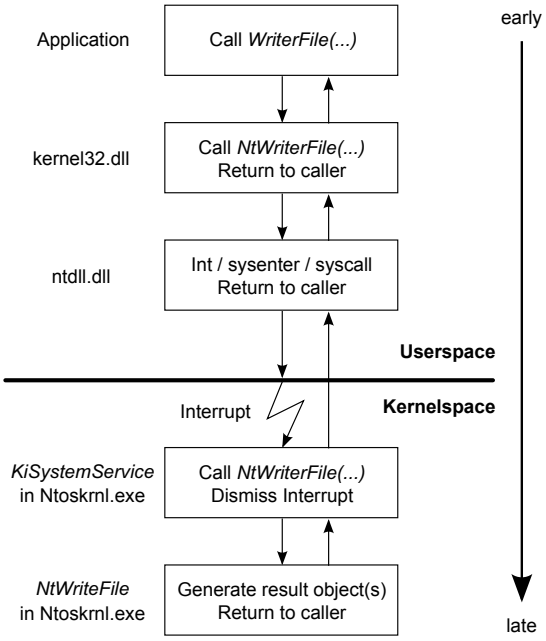


Abb. 1.10: Beispiel Syscall (Russinovich and Solomon [2012])

APIs sind vor allem Schnittstellen für Programmierer, die die verschiedensten vorgefertigten Funktionen zur Verfügung stellen. Die genaue Implementierung dieser Funktionen muss den Programmierer nicht interessieren, sondern nur ihre Funktionalität. Wegen dieser Transparenz und der genau definierten Schnittstellen können die Interna von API-Funktionen auch ausgetauscht werden, bspw. bei einem Betriebssystem-Update, ohne dass das Anwendungsprogramm deswegen geändert werden muss.

In der Windows-Welt bekannte Beispiele für APIs sind *win32.dll* für die Benutzung der grafischen Oberfläche und *OpenGL* und *DirectX* zur Erstellung von Computergrafiken.

1.6 Zusammenfassung

Der überwiegende Teil der heute üblichen Rechner basiert auf dem Von-Neumann-Modell. Der Aufbau und die Funktionsweise dieser Rechner ähneln sich sehr stark, woraus sich übereinstimmende Prinzipien bei der Programmierung dieser Rechner auf Maschinenebene ergeben. Einige grundsätzliche Befehlstypen finden sich in allen Maschinensprachen wieder. Aus diesen fundamentalen Befehlstypen, die auf Register und Hauptspeicher wirken, lassen sich bereits unabhängig von einer konkreten Architektur die Grundprinzipien der Assemblerprogrammierung ableiten. Im folgenden Studienbrief werden die erworbenen Kenntnisse auf eine reale Architektur, die IA-32 von Intel, angewandt.

Betriebssysteme machen aus der reinen Hardware benutzbare Rechner. Sie stellen die Software-Umgebung zur Verfügung, in der Anwenderprogramme, also auch Assemblerprogramme, ausgeführt werden können. Neben der Erledigung

diverser Verwaltungsaufgaben abstrahieren Betriebssysteme von der vorhandenen Hardware, verteilen die vorhandenen Ressourcen und sorgen gleichzeitig für eine möglichst hohe Betriebssicherheit. Zur Erlangung dieser Betriebssicherheit, die insbesondere durch Privilegierungen und Speicherverwaltung erreicht werden, werden die Betriebssysteme durch Hardware-Mechanismen direkt unterstützt. Diese werden wir im folgenden Studienbrief ebenfalls bei IA-32 kennenlernen.

1.7 Übungen

Nehmen Sie eine fiktive Rechnerarchitektur an, die über acht Register `R1` bis `R8` verfügt, und die mittels einer Pseudo-Assemblersprache, wie sie in Abs. 1.4.2 vorgestellt wurde, programmiert werden kann.

Ü

Übung 1.1

Beschreiben Sie detailliert die Ausführung folgender Befehlssequenz auf der Architektur:

```
mov R1,[1000]
mov R2,[1010]
add R1, R2
add R1, R1
mov [1000], R1
```

Ü

Übung 1.2

Was macht das folgende Programm? Welcher Wert steht nach Programmende in Register `R4`?
(Es wird angenommen, dass kein Registerüberlauf stattfindet.)

```
 mov R1,[1000]
 mov R2,[1010]
 cmp R1, R2
 jge label1
 mov R3, R2
 jmp label2
label1:
 mov R3, R1
label2:
 mov R4, 10
 add R4, R3
end
```

(*Anmerkung:* `label1` und `label2` sind Sprungmarken. Das sind symbolische Namen für Adressen im Programmspeicher, wie sie bei der Assemblerprogrammierung üblich sind. Die Ersetzung dieser Sprungmarken durch konkrete Speicheradressen geschieht automatisch bei der Umwandlung des Assemblerprogramms in den Maschinencode.)

Übung 1.3

Ü

Es sei angenommen, dass in den Registern `R1` bis `R3` drei vorzeichenlose ganze Zahlen stehen.
Schreiben Sie ein Assemblerprogramm, das in `R4` die Summe der Zahlen ablegt, falls diese größer 10 ist. Ist die Summe kleiner oder gleich 10, soll die kleinste der drei Zahlen in `R4` abgelegt werden. „Erfinden“ Sie hierzu ggf. neue bedingte Sprungbefehle wie bspw. `jle` (jump if less or equal).
(Wieder wird angenommen, dass kein Registerüberlauf stattfindet.)

Übung 1.4

Ü

Es sei angenommen, dass in den Registern `R1` bis `R5` fünf unsortierte vorzeichenlose ganze Zahlen stehen.
Beim Programmende sollen die fünf Zahlen aufsteigend sortiert in den Registern `R1` bis `R5` stehen.

Verzeichnisse

I. Abbildungen

II. Definitionen

III. Exkurse

IV. Literatur

Parvez Anwar. Buffer overflows in the microsoft windows environment. *Technical Report, RHUL-MA2009-06, University of London*, 2009.

O. Bittel. *Algorithmen und Datenstrukturen*. Vorlesungsskript, FH-Konstanz, 2007.

Volker Claus and Andreas Schwill. *Duden Informatik*. Bibliographisches Institut, 2003.

Solar Designer. Getting around non-executable stack (and fix). *Bugtraq*, 1997.

Gustavo Duarte. *Motherboard Chipsets and the Memory Map*. http://duartes.org/gustavo/blog/post/motherboard-chipsets-memory-map, 2008.

Felix Freiling, Ralf Hund, and Carsten Willems. *Software Reverse Engineering*. Vorlesung, Universität Mannheim, 2010.

Thorsten Holz. *Program Analysis*. Vorlesung, Ruhr-Universität Buchum, 2011.

Intel. *Intel 80386, Programmers Reference Manual*. http://css.csail.mit.edu/6.858/2011/readings/i386.pdf, 1987.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2012.

Kip R. Irvine. *Assembly Language for Intel-Based Computers (5th Edition)*. Prentice Hall, 2006.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Hanser Fachbuch, 1990.

Kurt Mehlhorn. *Datenstrukturen und Algorithmen 1: Sortieren und Suchen*. Teubner Verlag, 1986.

Tilo Müller. *Software reverse engineering*. Vorlesung, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2015.

Gary Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, 2000.

Ryan Roemer, Erik Buchanen, Hovav Shacham, and Steve Savage. Return-oriented programming: Exploitation without code injection. *University of California, San Diego*, 2008.

Ryan Roemer, Erik Buchanen, Hovav Shacham, and Steve Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & Sys. Security 15(1):2*, 2012.

Joachim Rohde. *Assembler GE-PACKT, 2. Auflage*. Redline GmbH, Heidelberg, 2007.

M. E. Russinovich and D. A. Solomon. *Windows Internals*. Microsoft Press Corp, 2012.

Uwe Schöning. *Algortithmen - kurz gefasst*. Spektrum Hochschultaschenbuch, 1997.

Hovav Shacham and andere. On the effectiveness of addressspace randomization. *ACM Conference on Computer and Communications Security (CCS)*, 2004.

Axel Stutz and Peter Klingebiel. *Übersicht über die C Standard-Bibliothek*. http://www2.hs-fulda.de/~klingebiel/c-stdlib/index.htm, 1999.

Ronald Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2010.

Ray Wisman. *Processor Architecture*. http://homepages.ius.edu/RWISMAN/C335/HTML/Chapter2.htm, 2012.

Anhang

Prioritäten und Assoziativitäten der Operatoren in C

Operatoren	Assoziativität
() [] -> .	von links
! ~ ++ -- + - * & (type) sizeof	von rechts
* / %	von links
+ -	von links
<< >>	von links
< <= > >=	von links
== !=	von links
&	von links
∧	von links
|	von links
&&	von links
||	von links
?:	von rechts
= += -= *= /= %= &= ∧= |= <<= >>=	von rechts
,	von links

Open C³S
Open Competence Center for Cyber Security

Fort- und Weiterbildung

Neue Bedrohungszenarien stellen Sicherheitsexperten und IT-Verantwortliche in Unternehmen und einschlägigen Behörden vor immer größere Herausforderungen. Neue Technologien und Anwendungen erfordern zusätzliches Know-how und personelle Ressourcen.

Zur Erhöhung des Fachkräftepools und um neues Forschungswissen schnell in die Praxis zu integrieren, haben sich die im Bereich lehrenden und forschenden Verbundpartner zum Ziel gesetzt, ein hochschuloffenes transdisziplinäres Weiterbildungsprogramm im Sektor Cyber Security zu entwickeln. Auf der Grundlage kooperativer Strukturen werden wissenschaftliche Weiterbildungsmodule im Verbund zu hochschulübergreifenden Modulpaketen und abschlussorientierten Ausbildungslinien konzipiert und im laufenden Studienbetrieb empirisch getestet.

Die Initiative soll High Potentials mit und ohne formale Hochschulzugangsberechtigung über innovative Weiterbildungsangebote (vom Zertifikat bis zum Masterprogramm) zu Sicherheitsexperten aus- und fortbilden. Hierzu werden innovative sektorale Lösungen zur Optimierung der Durchlässigkeit von berulicher und hoch- schulischer Bildung entwickelt und für eine erfolgreiche Implementierung vorbereitet. Unter prominenter Beteiligung einschlägiger Verbände, der Industrie sowie Sicherheits und Ermittlungsbehörden verfolgt die Initiative das Ziel, im deutschsprachigen Raum eine Generation von Fachkräften wissenschaftlich aus- und weiterzubilden, die unser Internet schützen kann.

Open Competence Center for Cyber Security

Open C³S ist aus dem Verbundvorhabens Open Competence Center for Cyber Security entstanden. Das Gesamtziel des Programms war die Entwicklung eines hochschuloffenen transdisziplinären Programms wissenschaftlicher Weiterbildung im Sektor Cyber Security. Das Bundesministerium für Bildung und Forschung (BMBF) fördert das Großprojekt im Rahmen des Wettbewerbs „Aufstieg durch Bildung: offene Hochschulen“, der aus BMBF-Mitteln und dem Europäischen Sozialfonds finanziert wird.

Neun in Forschung und Lehre renommierte Hochschulen und Universitäten aus dem gesamten Bundesgebiet haben sich zum Ziel gesetzt, Online-Studiengänge auf dem Gebiet der Cybersicherheit zu entwickeln. Dieses Konzept soll den Studierenden ermöglichen, sich berufsbegleitend auf hohem Niveau wissenschaftliche Qualifikationen anzueignen und akademische Abschlüsse zu erlangen. Beruflich erworbene Kompetenzen können eingebracht werden. Die Bezeichnung „Open“ steht auch für die Öffnung des Zugangs zu akademischer Bildung ohne klassischen Hochschulzugang.

Mission der Initiative ist es, dringend benötigte Sicherheitsexperten aus- und fortzubilden, um mit einer sicheren IT-Infrastruktur die Informationsgesellschaft in Deutschland und darüber hinaus zu stärken.

Umsetzungsnahes Wissen ist ein wesentlicher Schlüssel um der wachsenden digitalen Bedrohung zu begegnen. Solange wir nicht in der Lage sind, Systeme hinreichend zu härten, Netzwerke sicher zu designen und Software sicher zu entwickeln, bleiben wir anfällig für kriminelle Aktivitäten. Unser Ziel ist es, die Mitarbeiter von heute zu Sicherheitsexperten und Führungskräften von morgen auszubilden und dafür zu sorgen, dass sich die Zahl und die Fertigkeiten dieser Experten nachhaltig erhöht.

Weitere Informationen zu Open C³S: https://open-c3s.de

Z102 Systemnahe Programmierung

Dieses Modul beschäftigt sich mit Programmiertechniken, die auf einem tiefen Level auf die Gegebenheiten eines Rechnersystems Bezug nehmen.Wir sprechen deswegen von systemnaher Programmierung. Systemnahe Programmierung ist nicht mit Systemprogrammierung zu verwechseln. Systemprogrammierung benutzt die systemnahe Programmierung insbesondere zur Implementierung von Betriebssystemen oder ganz eng an Betriebssystem und Hardware angelehnter Softwarekomponenten wie Treiber, Prozesskommunikation usw.

Das wesentliche Merkmal der systemnahen Programmierung ist die direkte Kommunikation mit der Hardware und dem Betriebssystem eines Rechners. Diese erfordert fundamentale Kenntnisse über Architektur und Betriebssystem des Zielrechners. Umfangreiche Hochsprachen wie C++ oder Java abstrahieren von der Hardware eines Rechners, eine direkte Interaktion ist hier gerade nicht gewollt. Daher wird die systemnahe Programmierung meist in vergleichsweise minimalistischen Sprachen durchgeführt, entweder direkt in der Maschinensprache eines Rechners, die wir hier als Assembler bezeichnen, oder in C. C ist zwar ebenfalls eine Hochsprache, jedoch beinhaltet das Sprachkonzept von C viele Komponenten und Merkmale, die einen direkten Bezug zur Hardware haben. Große Teile der heute weit verbreiteten Betriebssysteme Windows und Linux sind in C programmiert.

Dieses Modul kann einerseits als unabhängiges Grundlagenmodul betrachtet werden. Die erworbenen Kompetenzen sind vielfach verwendbar und nützlich. Gleichzeitig ist dieses Modul aber auch Basis des fortgeschrittenen Moduls „Reverse Engineering“. Reverse Engineering ist ein wichtiger Teilbereich der digitalen Forensik und beschäftigt sich mit der Analyse unbekannter Software. Die zu untersuchende Software liegt dabei in der Regel nur in Form von Maschinenprogrammen vor, weswegen tiefe Kenntnisse im Bereich systemnaher Programmierung für Reverse Engineering unabdingbar sind. Um das Modul „Reverse Engineering“ nicht mit Grundlagen überladen zu müssen, werden diese hier geschaffen, um sich dort gezielter und eingehender den fortgeschrittenen Techniken widmen zu können.

Zertifikatsprogramm

Die Zertifikatsmodule auf wissenschaftlichem Niveau und mit hohem Praxisbezug bilden ein passgenaues Angebot an Qualifikation und Spezialisierung in der nebenberuflichen Weiterbildung. Damit können einzelne Module nebenberuflich studiert werden. Durch die Vergabe von ECTS-Punkten können sie auf ein Studium angerechnet werden.

https://zertifikatsprogramm.de