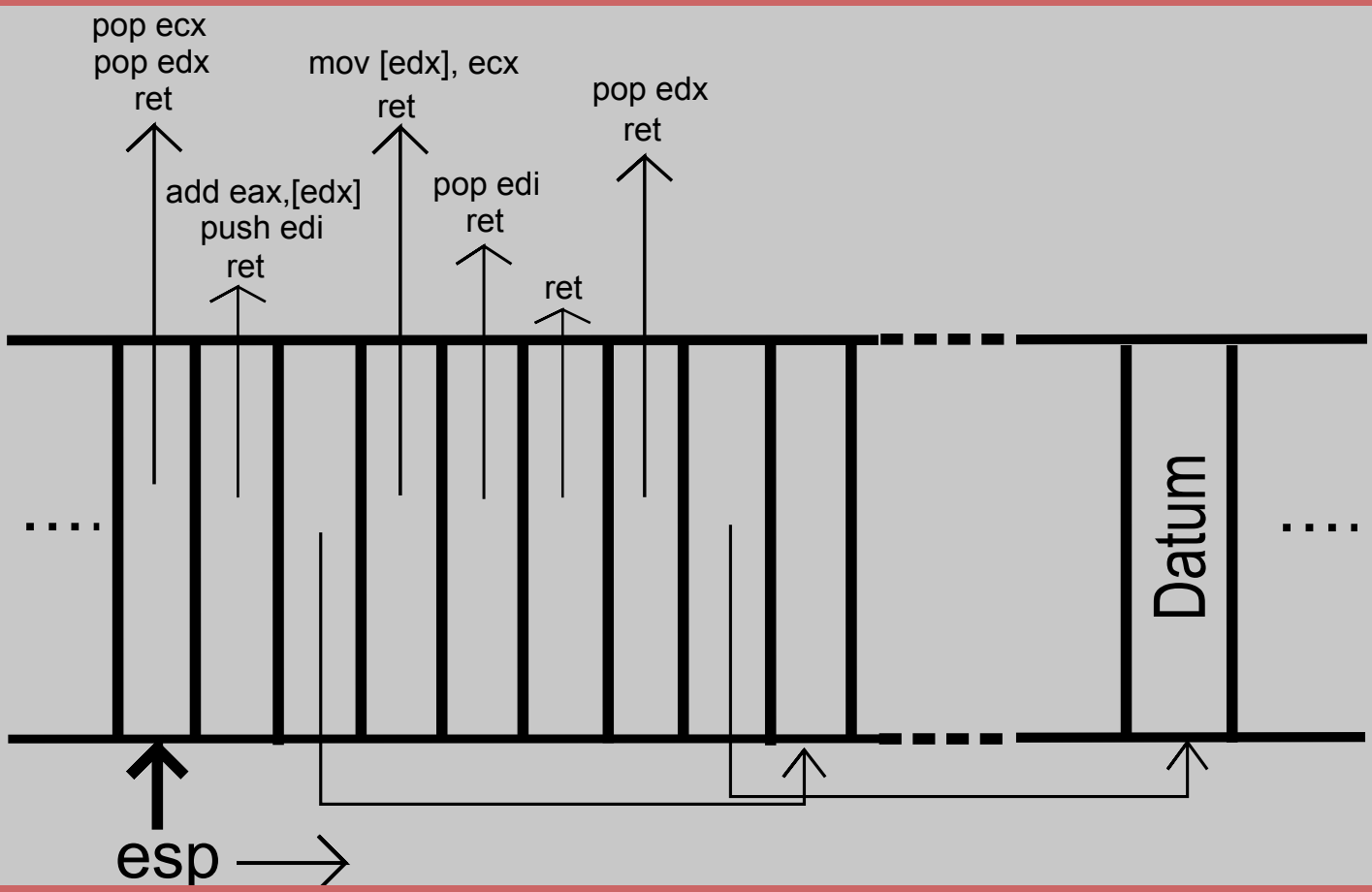
FAU
Friedrich-Alexander-Universität

Open C³S
Open Competence Center for Cyber Security



Bachelorstudiengang Informatik/IT-Sicherheit

Systemnahe Programmierung / Reverse Engineering [Sysprog]

Autoren:

Dr. rer. nat. Werner Massonne

Friedrich-Alexander-Universität Erlangen-Nürnberg

Systemnahe Programmierung / Reverse Engineering [Sysprog]

Studienbrief 1: Rechnerstrukturen und Betriebssysteme

Studienbrief 2: Assemblerprogrammierung

Studienbrief 3: Maschinenprogrammanalyse

Studienbrief 4: Systemnahe Sicherheitsaspekte

Studienbrief 5: Malware – Techniken und Analyse

Autor:

Dr. rer. nat. Werner Massonne

---

9. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

© 2024 Felix Freiling
Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

9. Auflage (15. November 2024)

Didaktische und redaktionelle Bearbeitung:
Romy Rahnfeld

Das Werk einschließlich seiner Teile ist urheberrechtlich geschützt. Jede Verwendung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung der Verfasser unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

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
Quelltext	Q
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 Modul „Algorithmen und Datenstrukturen“ haben Sie bereits die Programmiersprache C intensiv kennengelernt. In diesem Modul wenden wir uns mehr der Assemblerprogrammierung zu. Die systemnahe Programmierung ist nicht plattformunabhängig, sondern bezieht sich in ihrer konkreten Ausprägung immer auf ein festes Zielsystem. Als Zielsystem wählen wir in diesem Modul das Architekturmodell IA-32 von Intel mit dem Betriebssystem Microsoft Windows.

Kenntnisse in systemnaher Programmierung ermöglichen es uns, Programme und Mechanismen ihrer Implementierung auf einem sehr tiefen Level zu verstehen. Diese Kenntnisse wenden wir im Bereich der Programmanalyse an. Mit Programmanalyse im weitesten Sinne beschäftigt sich der zweite Teil dieses Moduls.

Im ersten Studienbrief werden die aus Sicht eines Programmierers wesentlichen Prinzipien heutiger Rechnerarchitekturen und die allgemeinen Aufgaben von Betriebssystemen vorgestellt, ohne auf eine reale Architektur Bezug zu nehmen. Viele Inhalte aus dem ersten Teil dieses Studienbriefs kennen Sie bereits aus den Modulen „Rechnerstrukturen“ und „Systemsicherheit 1“, sie werden hier zusammengefasst wiederholt. Im zweiten Teil des ersten Studienbriefs beschäftigen wir uns genauer mit Microsoft Windows und seinen Programmierschnittstellen.

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.

Im Anschluss wenden wir uns der Programmanalyse zu. In erster Linie geht es dabei um das Verstehen von Maschinenprogrammen. Das Ziel der Analysen ist die Repräsentation eines Maschinenprogramms auf einem höheren Abstraktionslevel, also auf Hochsprachenniveau. Diesen Vorgang bezeichnet man als Dekompilierung oder im weitesten Sinne auch als Reverse Engineering. Um eine Dekompilierung durchführen zu können, müssen wir zunächst den umgekehrten Weg verstehen, also die Codeerzeugung aus einem Hochsprachenprogramm. Mit der Codeerzeugung durch Compiler und dabei angewandten Optimierungsverfahren beschäftigen wir uns in Studienbrief 3.

Studienbrief 4 beschäftigt sich mit Sicherheitsaspekten und Sicherheitsproblemen. Es wird gezeigt, wie Sicherheitslücken in Software 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.

Studienbrief 5 beschäftigt sich schließlich mit einer konkreten Anwendung systemnaher Programmierkenntnisse, der Malware-Analyse. Malware ist selbst systemnah programmiert und liegt in der Regel nur als Maschinenprogramm vor, also helfen auch die Kenntnisse in systemnaher Programmierung bei ihrer Analyse.

Des Weiteren werden Typen, Merkmale und Implementierungs- und Verschleierungstechniken von Malware vorgestellt. Anhand des Analyse-Tools IDA wird das Vorgehen bei einer praktischen Malware-Analyse gezeigt.

Nach dem erfolgreichen Abschluss dieses Moduls werden Sie in der Lage sein, systemnahe Programme zu verfassen und zu verstehen. Die Grundprinzipien der Programmanalyse werden Ihnen vertraut sein. Sie verstehen die Mechanismen, die Sicherheitslücken in Programmen entstehen lassen, und Sie kennen auch einige Gegenmaßnahmen. Schließlich werden Sie ein Gefühl für die Funktionsweise von Malware entwickelt haben, und einfache Beispiele können Sie sogar selbst analysieren.

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. Darüber hinaus können Sie 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.

Des Weiteren können Sie die Kernfunktionen und Aufgaben eines Betriebssystems nennen und erklären. Die Begriffe Prozess, Adressraum und Programmierschnittstelle können Sie erläutern.

Sie können den grundsätzlichen Aufbau von Windows beschreiben. Sie können erklären, wie einige wichtige Betriebssystemfunktionalitäten in Windows realisiert sind. Sie haben einen Überblick über den Aufbau des Windows-PE-Formats gewonnen und können den Aufbau grob beschreiben.

Darüber hinaus können Sie die Windows-Programmierschnittstelle (API) und den Mechanismus der Systemaufrufe erklären. Ebenso können Sie den Aufbau von Windows-Prozessen und der zugehörigen Datenstrukturen sowohl beschreiben als auch analysieren. Die Verfahren der Exception-Behandlung können Sie benennen und erklären.

1.2 Advance Organizer

Wenn Sie einen Rechner systemnah, d. h. 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.

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 zunächst 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. Danach werden einige Besonderheiten des Betriebssystems Microsoft Windows herausgearbeitet, die für die systemnahe Programmierung Windows-basierter Rechner wesentlich sind.

1.3 Rechnerstrukturen und Assembler

Systemnahe Programmierung wird überwiegend in C oder direkt in Assembler durchgeführt. Die Programmiersprache C wurde bereits im Modul „Algorithmen-

und Datenstrukturen“ eingehend behandelt. Ihre Eignung zur systemnahen Programmierung ist durch die Möglichkeiten eines direkten Zugriffs auf die Speicherstrukturen eines Rechners offensichtlich. In diesem Modul wenden wir uns der Assemblerprogrammierung zu, also der Programmierung einer Architektur auf dem untersten Hardware-Level. Assemblerprogramme nehmen direkten Bezug auf die Hardware-Gegebenheiten eines Rechners, nämlich die Recheneinheiten, Register, Speicheranbindungen und den sich daraus ergebenden Befehlssatz.

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.

Mnemonics

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.

Disassembler

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.

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.[1]

```
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
```

[1] 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[2] (*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.

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

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).

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

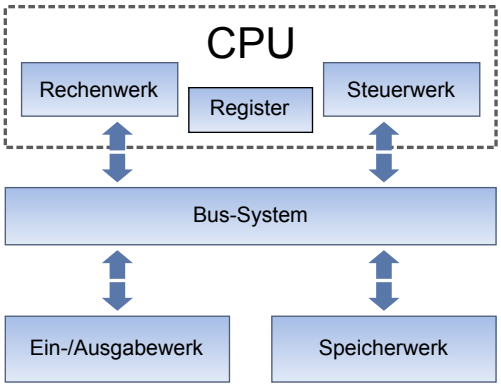


Abb. 1.1: Von-Neumann-Architektur

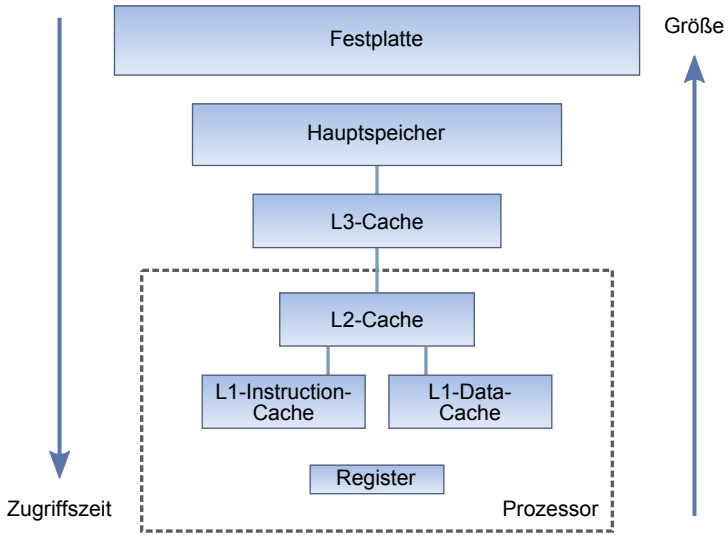


Abb. 1.2: Speicherhierarchie

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

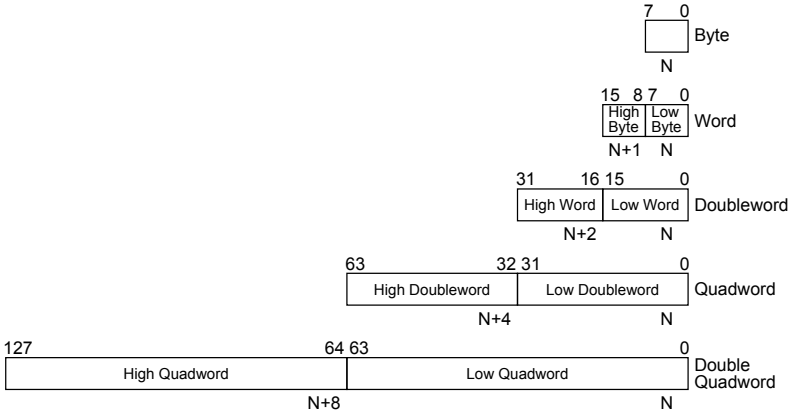
Kontrollaufgabe 1.1

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 aus-

führen 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.

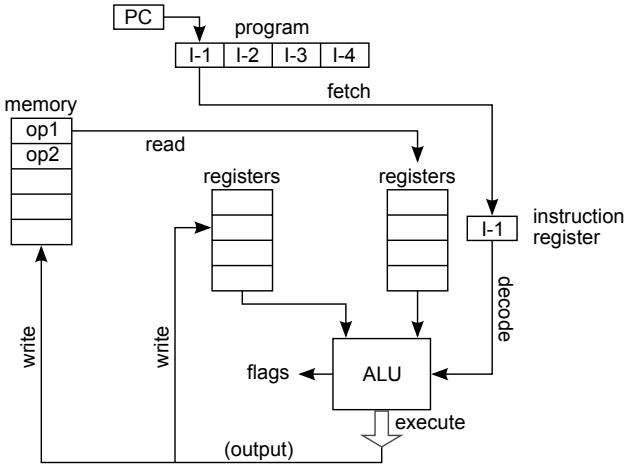


Abb. 1.4: Befehlsausführung bei der Von-Neumann-Architektur [Wisman, 2012]

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:[3]

```
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:

```
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
```

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

`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[4] 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 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.

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

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 1.2: **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[5] und IBM PowerPC[6]. 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.

B

Beispiel 1.3

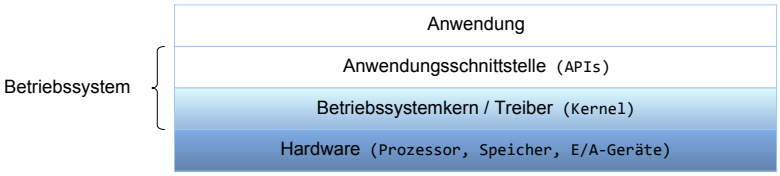
Einen Anwender interessiert es i. Allg. nicht, welche Grafikkarte konkret in einem Rechner eingebaut ist und wie sie genau funktioniert. Auch für die meisten Anwendungen spielt dies keine Rolle. Das Betriebssystem stellt daher nur eine Anzahl von Funktionen zur Verfügung, die es erlauben, die Grafikkarte 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.

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

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 Architektur IA-32 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:

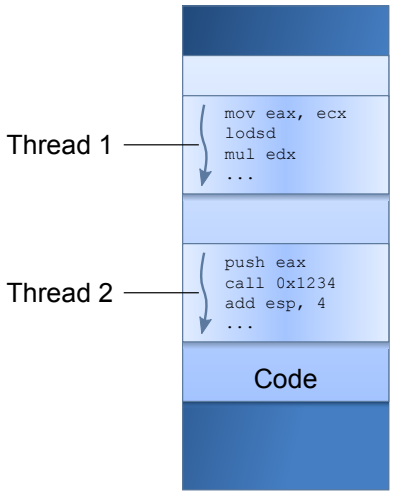
1. ausführbarer Programmcode
2. eigener Adressraum im Speicher
3. zumindest ein Thread
4. Verwaltungsstruktur

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.

Abb. 1.6: 2 Gleichzeitig ablaufende Threads



Jeder Thread besitzt einen eigenen *Stack*. Ein Stack[7] ist dabei ein Teil des Hauptspeichers, in dem lokale Daten eines Thread gehalten werden. Gäbe es nur einen Stack, so würden sich unabhängig voneinander laufende Threads gegenseitig den Stack zerstören.

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

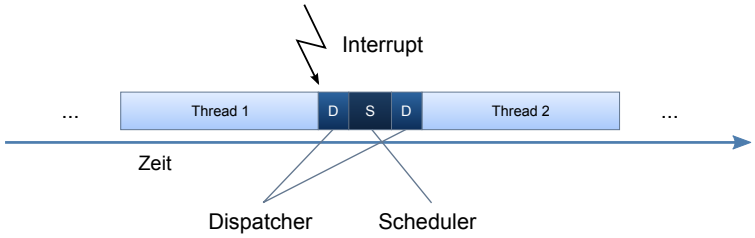
Scheduler

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.

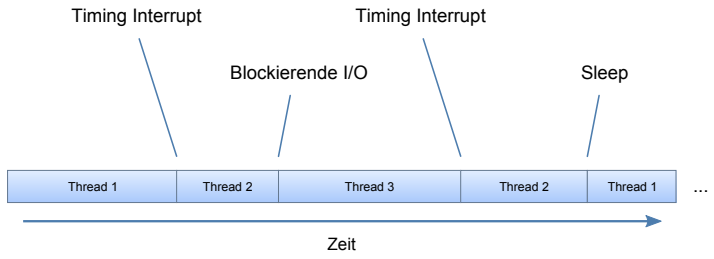
Abb. 1.7: Scheduler und Dispatcher



Interrupt

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.[8]

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 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.

[8] 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.

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.

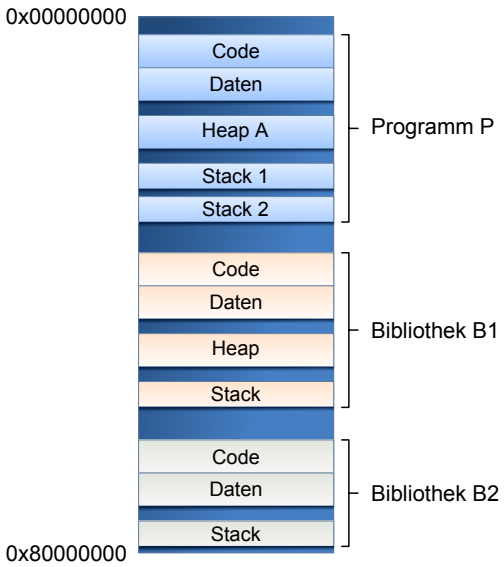


Abb. 1.9: Bereiche im Userspace

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.

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.

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.

1.6 Windows

Die systemnahe Programmierung basiert auf der Architektur und dem Betriebssystem eines Rechners. Bisher haben wir das grundsätzliche Arbeitsprinzip heute

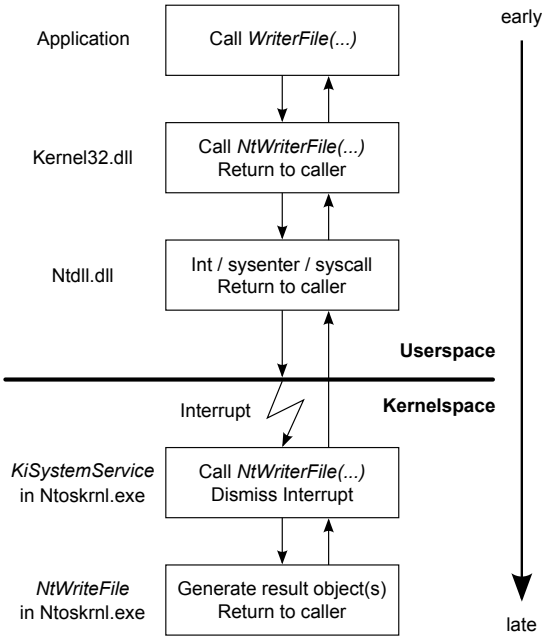


Abb. 1.10: Beispiel Syscall [Russinovich and Solomon, 2012]

üblicher Rechnerarchitekturen und die Grundprinzipien moderner Betriebssysteme kennengelernt. Im weiteren Verlauf dieses Moduls beschäftigen wir uns mit der konkreten Rechnerarchitektur IA-32 und mit dem konkreten Betriebssystem Microsoft Windows. Die dabei erlernten Prinzipien der systemnahen Programmierung sind zum großen Teil auf andere Kombinationen von Architektur und/oder Betriebssystem übertragbar.

Wir beginnen mit der näheren Betrachtung des Betriebssystems Microsoft Windows. Im nächsten Studienbrief werden wir uns eingehend mit der Architektur IA-32 und ihrer Programmierung auf Assemblerebene beschäftigen.

1.6.1 Einführung

Windows-Betriebssystemfamilie

In diesem Abschnitt werden die allgemeinen Strukturen eines Betriebssystems am konkreten Beispiel von Microsoft Windows näher untersucht. Innerhalb der Windows-Betriebssystemfamilie betrachten wir die 32-Bit-NT-Linie, die mit Windows NT ihren Anfang nahm und über Windows 2000, XP, Vista und Windows 7 bis zum aktuellen Windows 10 bis heute weitergeführt wurde. Der überwiegende Teil der heute im Einsatz befindlichen Arbeitsplatzrechner arbeitet mit diesen Betriebssystemen. Auch die neueren 64-Bit-Varianten unterscheiden sich im prinzipiellen internen Aufbau nicht wesentlich von den 32-Bit-Systemen. Wegen ihres bedeutenden Marktanteils sind Windows-Systeme für Reverse Engineering besonders interessant. Im Hinblick auf eine Programmanalyse interessieren uns hierbei insbesondere die Programmierschnittstellen sowie der Aufbau von Windows-Prozessen und der Prozessdatenstrukturen.

Abb. 1.11 zeigt den logischen Aufbau des Windows-Betriebssystems. Prozesse, die unter dem Betriebssystem Windows laufen, gliedern sich in zwei Kategorien, nämlich in Kernel- und User-Prozesse. Die Bestandteile von Windows, die im Kernelmode laufen, sind unterhalb des schwarzen Querbalkens dargestellt.

Schichtenmodell von Windows

Im Kernelmode werden die Prozesse des eigentlichen Betriebssystemkerns, die Gerätetreiber, HAL (*Hardware Abstraction Layer*) und die elementaren internen Verwaltungsdienste ausgeführt. Zu den internen Verwaltungsdiensten zählen bspw.

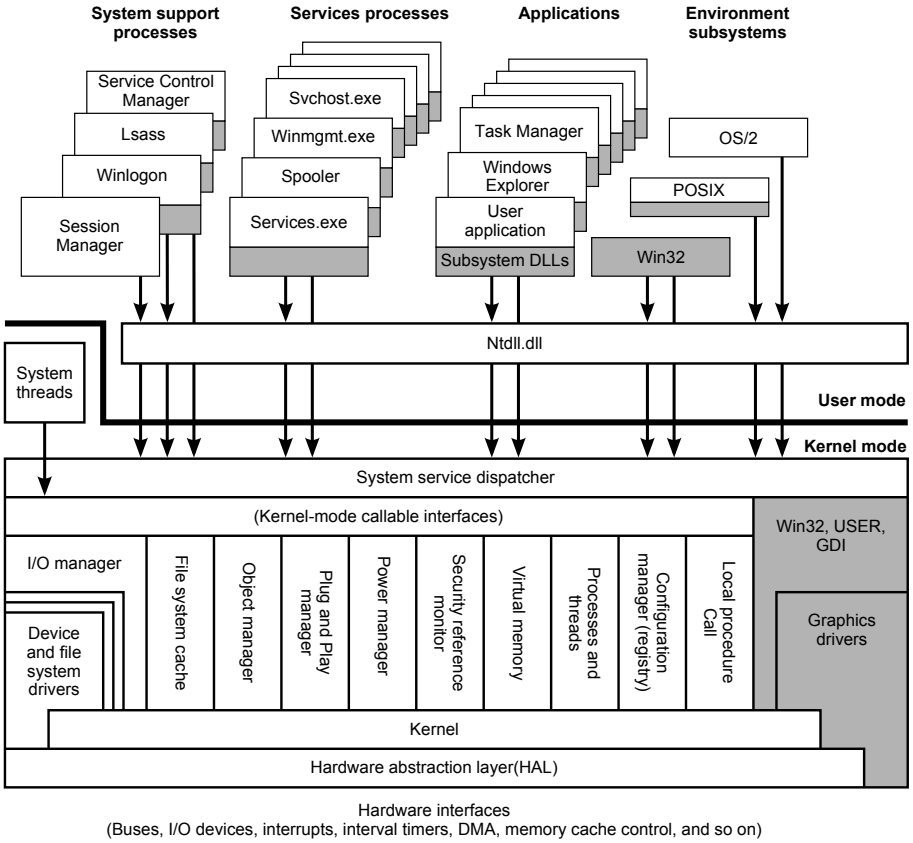


Abb. 1.11: Logischer Aufbau von Windows [Russinovich and Solomon, 2012]

die Prozess- und Speicherverwaltung, das I/O-Management und das *Graphical Device Interface* (GDI). HAL (*Hardware Abstraction Layer*) ist die unterste Schicht von Windows zur Hardware hin, die direkt auf den verwendeten Prozessor und bestimmte Besonderheiten der Hardware Bezug nimmt. Diese Zwischenschicht vereinfacht die Portierung des Betriebssystems und schirmt die hardwarespezifischen Eigenschaften der Zielplattform vom Rest des Betriebssystems ab.

Systemprozesse

Im Usermode laufen zunächst einmal verschiedene Systemprozesse wie z. B. LSASS (*Local Security Authority Subsystem*). LSASS ist der lokale Sicherheitsauthentifizierungsserver. Er überprüft die Gültigkeit der Benutzeranmeldung. LSASS ist eng mit dem Winlogon-Dienst und dem Session Manager verbunden, die für das An- und Abmelden der Benutzer und die Verwaltung der benutzerspezifischen Einstellungen zuständig sind.

Dienste

Weiterhin existieren Prozesse, die Steuerungsfunktionen im System übernehmen. Sie werden deshalb Dienste genannt. Diese Dienste arbeiten im Hintergrund und kommunizieren dabei nicht direkt mit dem Anwender. Häufig gibt es zur Konfiguration und Steuerung eines Dienstes separate Programme, in Windows sind die meisten dieser Programme in der Systemsteuerung zusammengefasst. Der Dienst *spoolsv.exe* ermöglicht es bspw., Druck- oder Faxaufträge durchzuführen, ohne andere Arbeiten am Computer zu unterbrechen. Der übergeordnete Dienst *services.exe* verwaltet das Anhalten und Starten von Diensten.

Anwendungen

Weitere Prozesse, die im Benutzermodus laufen, sind die Anwendungen. Sie werden meist vom Benutzer selbst gestartet, sofern nicht eingestellt wurde, dass sie automatisch bei jedem Systemstart anlaufen. Explorer und Task Manager sind

bekannte Beispiele für solche Anwendungen. Auch Programme, die nachträglich auf das System installiert werden, zählen dazu, z. B. Word oder PowerPoint.

Subsysteme

Schließlich existieren Subsysteme, die Funktionen zur Verwaltung eines bestimmten Programm- bzw. Prozesstyps zur Verfügung stellen. Beispiele hierfür sind die Subsysteme OS/2, POSIX und MS-DOS, die die Ausführung von Programmen dieser Betriebssysteme gestatten. Jedes Subsystem stellt eine eigene Programmierschnittstelle (API) zur Verfügung. Win32 ist mit seiner API aus dieser Sicht selbst ein Subsystem, auch innerhalb der eigenen 32-Bit-Windows-Umgebung.

1.6.2 Anwendungen und Bibliotheken

Portable-Executable-Format

Alle Dateitypen, die auf einem Windows-System zur Ausführung kommen, haben eine Eigenschaft gemein. Sie alle verwenden das PE(*Portable Executable*)-Format. Anwendungen haben die Endungen *.exe, .com* oder *.scr* (Screensaver). Dynamische Bibliotheken haben die Endungen *.dll* und *.ocx*. Kernel-Programme oder Treiber enden mit *.sys, .vxd, .exe* oder *.dll*. Darüber hinaus gibt es noch ausführbare Skriptdateien (z. B. mit den Endungen *.bat* oder *.cmd*), die uns hier allerdings nicht weiter interessieren.

Die Windows-Anwendungen lassen sich in drei Gruppen unterteilen:

1. Windows-Systemprogramme (z. B. *winlogon.exe* für die Benutzeranmeldung und der Programmmanager *explorer.exe*)
2. Windows-Hilfsprogramme (z. B. der Taschenrechner *calc.exe* und das Textbearbeitungsprogramm *notepad.exe*)
3. Programme, die nicht zum Betriebssystem selbst gehören.

Alle Windows-Anwendungen bestehen aus drei grundlegenden Teilen: Code, Daten und optionalen Debug-Informationen.

Libraries

Anwendungen können vorgefertigte Programmbibliotheken (*Libraries*) verwenden. Auch diese Bibliotheken bestehen aus Code und Daten, sind aber nicht alleine ausführbar. Windows stellt viele Standardbibliotheken zur Verfügung, diese bilden die API (*Application Programming Interface*). Beispiele hierfür sind die Bibliotheken *user32.dll* und *kernel32.dll*. Des Weiteren gibt es Compiler-Bibliotheken, aus denen sich der Programmierer bei der Erstellung eines Hochsprachenprogramms bedienen kann. Darüber hinaus gibt es unzählige Bibliotheken von irgendwelchen Herstellern für irgendwelche benutzerspezifischen Spezialanwendungen.

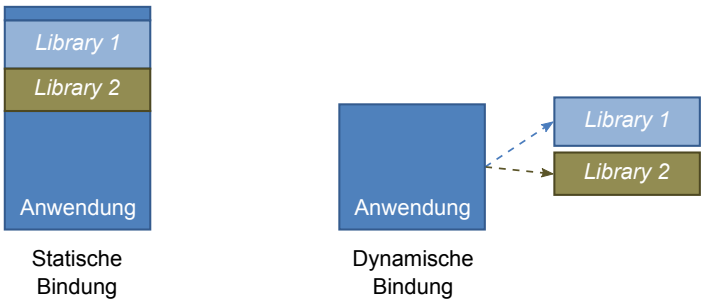
Import und Export

Bibliotheken stellen für Anwendungen Funktionen und Daten zur Verfügung. Man sagt auch, Bibliotheken **exportieren** Funktionen und Daten. Diese können von Anwendungen ihrerseits **importiert** werden. Bibliotheken exportieren Funktionen und Daten via Namen (z. B. *CreateFileA*) oder Ordinalzahl (eine Nummer, z. B. `0x30`), die von den Anwendungen ihrerseits via Namen oder Ordinalzahl importiert werden.

Der Import von Bibliotheken bzw. einzelner Bestandteile davon kann grundsätzlich statisch oder dynamisch erfolgen (s. Abb. 1.12). Bei der statischen Bindung werden die Bibliotheken der Anwendung einverleibt. Dies geschieht bereits bei der Kompilierung der Anwendung. Bei der dynamischen Bindung werden die Bibliotheken erst zur Laufzeit der Anwendung in den Speicher geladen. Beim Kompilieren werden lediglich Verweise auf die für die Ausführung notwendigen Bibliotheken sowie darin enthaltenen Funktionen und Daten in der Anwendung

hinterlegt. Beim Start der Anwendung wird geprüft, ob die in den Verweisen stehenden Bibliotheken schon geladen wurden. Falls nicht, lädt der Loader fehlende Bibliotheken in den Speicher.

Abb. 1.12: Statische und dynamische Bindung



Dynamic Link Library

Windows-Bibliotheken sind für die dynamische Bindung vorgesehen. Das erklärt auch die Dateiendung *.dll* für *Dynamic Link Library* (DLL). Bibliotheken werden entweder zusammen mit der Anwendung auf einem Rechner installiert, oder ihr Vorhandensein wird auf dem Zielrechner der Anwendung vorausgesetzt.

Beide Bindungsarten haben offensichtliche Vor- und Nachteile. Die Vorteile einer statischen Bindung sind das schnelle Laden und Starten einer Anwendung und die Unabhängigkeit vom Vorhandensein einzelner Bibliotheken. Die Nachteile bestehen neben der logischerweise ausgeschlossenen Mehrfachverwendung von Bibliotheken zum einen in einer größeren, aufgeblähten Anwendungsdatei und zum anderen in der Notwendigkeit, die Anwendungen bei einem Update der Bibliothek neu zu kompilieren.

K

Kontrollaufgabe 1.3

Formulieren Sie die Vor- und Nachteile der dynamischen Bindung!

1.6.3 Windows API

Das Windows *Application Programming Interface* (API) bildet die Schnittstelle zwischen Betriebssystem und Anwenderprogrammen. Die API besteht aus vielen DLLs, die sich im *system32*-Ordner eines Windows-Systems befinden. Implementiert sind diese DLLs in C oder Assembler. Die Ordnung der DLLs ist nicht flach, sondern hierarchisch. Viele DLLs benutzen ihrerseits andere DLLs, wie dies in Abb. 1.13 dargestellt ist.

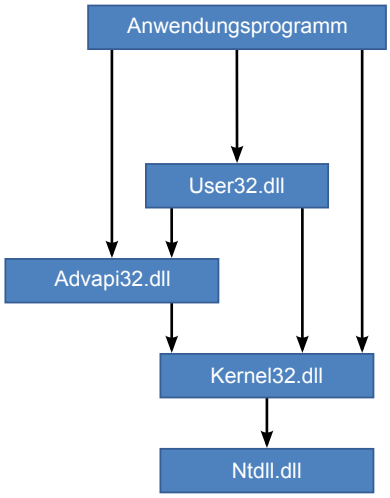
Einige wichtige DLLs sind die folgenden:

- Native API: *ntdll.dll*
- Kernelfunktionen: *kernel32.dll*
- Basisfunktionen: *advapi32.dll*
- Grafik/Fenster: *user32.dll*, *gdi32.dll*
- TCP/IP: *Winsock ws2_32.dll*

Systemaufruf

Die native API, die aus der Bibliothek *ntdll.dll* besteht, befindet sich am äußersten Ende der DLL-Hierarchie und ist nicht dazu vorgesehen, von Anwenderprogrammen direkt angesprochen zu werden. Einem Aufruf einer Funktion dieser Bibliothek folgt ein Systemaufruf, also die Kommunikation mit einem Kernel-Prozess. Alle übergeordneten Bibliotheksaufrufe, die letztendlich eine Kommunikation

Abb. 1.13: DLL-Hierarchie



mit dem Kernel benötigen, landen bei der *ntdll.dll*. Die Bibliothek *ntdll.dll* kommuniziert als einzige Bibliothek mit dem Kernel. Ein Interrupt, der den Wechsel vom Benutzermodus in den Kernel-Modus einleitet, wird in der *ntdll.dll* ausgelöst. Folgerichtig wird nach dem Bearbeiten der Anfrage im Kernel-Modus ein Ergebnis erst an *ntdll.dll* übergeben, bevor es an andere DLLs weitergereicht wird. Offiziell ist *ntdll.dll* undokumentiert, weil sie nicht zur Benutzung durch direkte Aufrufe vorgesehen ist. Microsoft ändert auch gelegentlich die Schnittstellen, um die Aktivitäten von Reverse Engineering an dieser DLL zu untergraben.

Eine Ebene oberhalb befindet sich *kernel32.dll*, die Funktionsaufrufe an *ntdll.dll* weiterleitet. Basisfunktionen wie die Registry-Verwaltung bietet *advapi32.dll* an. Sie sendet ihrerseits Informationen an die Bibliothek *kernel32.dll* weiter. Bibliotheken zur Steuerung von Grafiken und Fenstern sind die *user32.dll* und die *gdi32.dll*. Sie leiten den Informationsfluss an *advapi32.dll* oder *kernel32.dll* weiter.

Ein Aufruf einer Funktion einer DLL hat allgemein die Form *DLL!Funktion*(A/W), bspw. *Kernel32!CreateFileA*. Dabei steht *Kernel32* für die aufzurufende DLL *kernel32.dll*. Ein Ausrufezeichen oder ein Punkt trennen den Namen der DLL von der Funktion, die innerhalb dieser DLL aufgerufen werden soll. *CreateFile* ist der Name einer Funktion zum Erstellen von Dateien. Viele API-Funktionen haben hinter dem eigentlichen Funktionsnamen noch einen Buchstaben (A oder W) angehängt. Diese Unterscheidung wird dann gemacht, wenn die Übergabeparameter einen oder mehrere Strings enthalten. Hierbei steht A für ASCII-Format und W für Unicode-Format.

B

Beispiel 1.5

Die Deklaration der API-Funktion *CreateFile* sieht in C-Notation wie folgt aus:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,           // pointer to name of the file
  DWORD dwDesiredAccess,        // access (read-write) mode
  DWORD dwShareMode,            // share mode
  LPSECURITY_ATTRIBUTES         // pointer to
  lpSecurityAttributes,         // security attributes
  DWORD dwCreationDistribution, // how to create
  DWORD dwFlagsAndAttributes,   // file attributes
  HANDLE hTemplateFile          // handle to file with
                                // attributes to copy
  );
```

Auf Details soll hier nicht eingegangen werden. Der Funktion werden Namen und diverse Zugriffsvorgaben für die zu erstellende Datei übergeben. Rückgabewert ist ein sogenannter *Handle*; dies ist ein eindeutiger Referenzwert zu einer vom Betriebssystem verwalteten Systemressource, z. B. einem Bildschirmobjekt oder einer Datei auf einer Festplatte.

1.6.4 Systemaufrufe

Ob ein Programm im User- oder im Kernelmode arbeitet, wird durch den *Current Privilege Level* (CPL) angezeigt.[9] Hardware-Zugriffe und privilegierte CPU-Befehle, die direkt auf die Hardware wirken, sind nur im Kernelmode möglich, ebenso wie Zugriffe auf den Kernelspace. Bei Anwendungen sind daher unter Umständen oft Wechsel in den Kernelmode notwendig.

In Abb. 1.14 ist ein Aufruf der Funktion *CreateFileA* abgebildet, der durch mehrere DLLs schließlich den Sprung in den Kernelmode schafft, um dort die nötige Funktion auszuführen. Der Wechsel in den Kernelmode geschieht über einen Systemaufruf (Syscall). In der Regel wird eine Anwendung den Syscall durch den Umweg über die Windows API ausführen. Einen Syscall direkt aus einer Anwendung heraus aufzurufen ist durchaus auch möglich, allerdings wohl ausschließlich auf der Assemblerebene. Darüber hinaus sind die Schnittstellen zum Kernel nicht offiziell dokumentiert, was einem „normalen“ Programmierer den Gebrauch stark erschwert.

E

Exkurs 1.3: **Syscall bei x86-Prozessoren**

Die Implementierung von Syscalls auf Assemblerebene sei bereits an dieser Stelle gezeigt, obwohl die Grundlagen zum Verständnis erst in Studienbrief 2 geschaffen werden. Im Moment reicht ein intuitives Verständnis allerdings aus, die Details können später nachgeschlagen werden.

Übergang in den Kernelmode

Der Übergang in den Kernelmode erfolgt bei älteren Systemen durch Auslösen des Interrupt 3 mit dem `int`-Befehl. Ab dem Pentium II gibt es den Befehl `sysenter`, bei AMD-Prozessoren heißt er `syscall`. Durch `sysenter`

[9] Bei IA-32 ist dieser für Windows-Programme in Bit 0 und 1 des Segmentregisters `cs` hinterlegt.

werden die Register `cs, ss, eip` und `esp` aus CPU-modellspezifischen Systemregistern geladen, was den Befehl unter die Kontrolle des Kernelmode stellt, obwohl er im Usermode aufgerufen werden kann. Der Rücksprung aus dem Kernelmode in den Usermode erfolgt (bei Aufruf über `sysenter`) mit dem Befehl `sysexit`. Die vier genannten Register werden dabei wieder aus CPU-modellspezifischen Registern geladen.

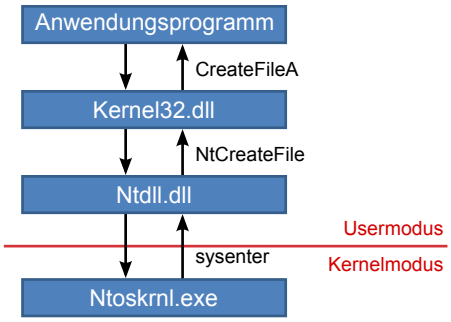


Abb. 1.14: Übergang in den Kernelmode

1.6.5 Programmierung der Windows API

Aufruf von API-Funktionen

API-Funktionen werden von einer Hochsprache wie C vielfach indirekt über die Funktionen der Standardbibliothek aufgerufen, können aber auch direkt von der Hochsprache aus oder von einem Assemblerprogramm benutzt werden. Als Beispiel ist nachfolgend ein Windows-Programm in C aufgelistet, das die Funktion *MessageBox* der Windows-DLL *user32.dll* benutzt, die ein in Windows übliches Meldungsfeld erzeugt.

Quelltext 1.1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <windows.h>
4  const char szText[] = "Haben Sie d. Programm verstanden?";
5
6  int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE
       hPrevInstance, PSTR szCmdLine, int iCmdShow) {
7       int iAntwort = MessageBox(NULL, szText, "Eine kleine Box",
            MB_ICONINFORMATION | MB_OKCANCEL | MB_DEFBUTTON1);
8       if (IDOK == iAntwort)
9         MessageBox(NULL, "Das ist prima!","",
              MB_ICONINFORMATION | MB_OK | MB_DEFBUTTON1);
10      else if (IDCANCEL == iAntwort)
11        MessageBox(NULL, "Schade, dann schauen Sie es sich
              bitte nochmal an!","",MB_ICONINFORMATION | MB_OK |
               MB_DEFBUTTON1);
12      return 0;
13 }
```

Q

Die Deklarationen der Windows API werden durch Einfügen der Datei `windows.h` eingebunden. Windows-Programme in C verwenden statt der üblichen Hauptfunktion `main` die Funktion `WinMain` mit diversen Standardparametern. Die Funktion

MessageBox erhält einige Textparameter und Parameter für die zu generierenden interaktiven Bedienknöpfe.

Wer sich eingehend mit der Programmierung der Windows API, insbesondere zur Erzeugung grafischer Oberflächen, beschäftigen will, kann sich bspw. bei diesen (einführenden) Webseiten bedienen. Details würden den Rahmen dieses Moduls sprengen:

- http://pronix.linuxdelta.de/C/win32/win32_1.shtml
- http://www.zetcode.com/gui/winapi/main/
- https://riptutorial.com/de/ebook/winapi
- http://www.winprog.org/tutorial/

Als allgemeines Nachschlagewerk ist die umfassende Dokumentation von Mikrosoft selbst zu empfehlen:

https://docs.microsoft.com/en-us/windows/win32/api/

In den Übungen zu diesem Studienbrief können Sie sich ein wenig näher mit der Windows-API-Programmierung in C beschäftigen. Die Programmierung der Windows API in Assembler wird Gegenstand der Übungen des folgenden Studienbriefs sein.

1.6.6 Das PE-Format

Das PE-Format (*Portable Executable*) ist das Binärformat für ausführbare Dateien in Windows-Systemen und basiert auf dem *Common Object File Format* (COFF).[10] *Portable* suggeriert eine Austauschbarkeit der Dateien zwischen unterschiedlichen Betriebssystemen. In Wirklichkeit sind PE-Dateien allenfalls von anderen Betriebssystemen ladbar, aber noch lange nicht ausführbar. Dies gilt auch innerhalb der Windows-Familie.

Das PE-Format spezifiziert die Struktur unter Windows ausführbarer Dateien, also von Anwendungen, Bibliotheken und Gerätetreibern. Insbesondere legt die Spezifikation fest, wo und wie in einer PE-Datei die Informationen über benötigte Bibliotheken stehen, und wo die Code- und Datenblöcke zu finden sind. Der Windows Loader wird durch das PE-Format direkt bei der Erstellung eines Prozesses unterstützt.

Alignment

Eine PE-Datei wird nahezu 1:1 in den Speicher übertragen. Allerdings ist eine PE-Datei kompakter als das Speicherabbild. Dies ist in Abb. 1.15 dargestellt. Die einzelnen Blöcke der PE-Datei (Header und Sections[11]) beginnen in der Datei häufig bei Mehrfachen von `0x200` Byte, also bei `0x0, 0x200, 0x400` usw. Dies wird als Datei-Alignment bezeichnet. Diese Blöcke werden in den Speicher ab den Mehrfachen von `0x1000` Byte geladen, also ab `0x0, 0x1000, 0x2000` usw. Dies ist das Speicher-Alignment der Blöcke. Zwangsläufig entstehen so „Lücken“ im Speicher, die größer sind als die Lücken in der Datei.

Eine PE-Datei wird in den linearen oder virtuellen Adressraum geladen und belegt dort i. Allg. nicht den Platz ab Adresse `0x0`. Abbildung 1.16 zeigt dies nochmals anhand des vorher gezeigten Beispiels. Es wird angenommen, dass das Speicherabbild der PE-Datei ab der virtuellen Adresse (VA) `0x400000` im Speicher platziert

[10] Nähere Informationen zum COFF-Format sind bspw. unter en.wikipedia.org/wiki/COFF zu finden.
[11] Dies sind insbesondere Code- und Datenblöcke, die meist als .text- und .data Sections bezeichnet werden.

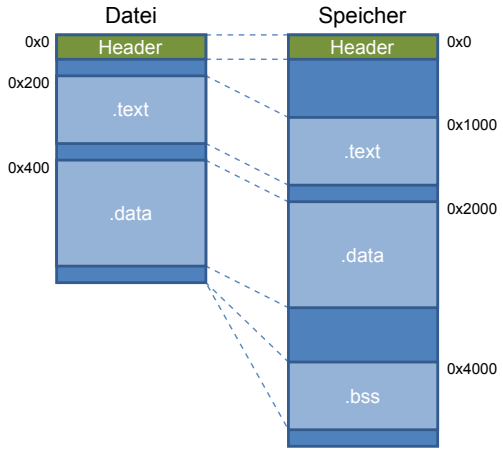


Abb. 1.15: Alignment in Datei und Speicher

wird. Die .text Section beginnt damit bspw. an der virtuellen Adresse `0x401000`. Der Offset zur relativen Startadresse des Speicherabbilds wird als relative virtuelle Adresse (RVA) bezeichnet. Die .text Section beginnt also ab der RVA `0x1000` zur VA `0x400000` des Speicherabbilds der PE-Datei.

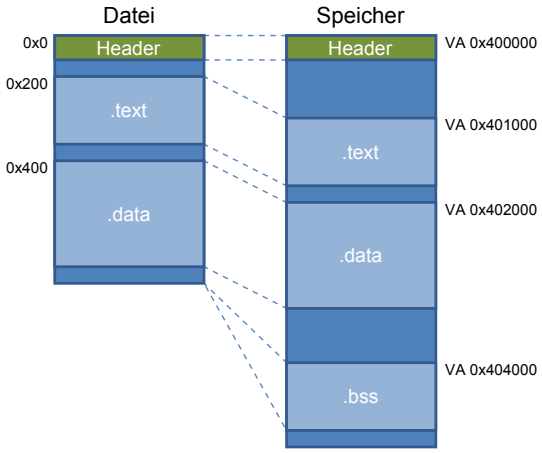


Abb. 1.16: Offset, virtuelle Adresse und relative virtuelle Adresse

Alle Dateien im PE-Format (s. Abb. 1.17) setzen sich aus zwei Bestandteilen zusammen, einem **Header** und ihm nachfolgenden **Sections**. In den Sections sind die eigentlichen Inhalte einer PE-Datei enthalten. Der Header besteht aus drei Teilen:

DOS Header: Der DOS Header[12] existierte schon zur DOS-Zeit und gewährleistet eine Art Aufwärtskompatibilität. Der DOS Header erlaubt es theoretisch, eine für Windows kompilierte Datei unter einer alten DOS-Version auszuführen. Allerdings wird dies in der Praxis kaum genutzt, und die meisten Windows-Programme brechen mit der Meldung „This program cannot be run in DOS Mode“ (oder ähnlich) ab, wenn sie unter einem inkompatiblen Betriebssystem ausgeführt werden.

PE Header: Im PE Header sind die wichtigsten Verwaltungs- und Strukturinformationen der PE-Datei enthalten, die insbesondere der Loader bei der

[12] Der DOS Header wird oft vereinfacht auch DOS Stub genannt, auch wenn der eigentliche DOS Stub nur ein Teil des DOS Header ist.

Prozesserzeugung benutzt. Auf einige Details werden wir im Folgenden noch näher eingehen.

Section Table: Die Größe der Section Table ist variabel, weil die Anzahl der Sections beliebig sein kann. Die Section Table enthält die Metadaten der Sections, also alle Informationen über die auf den Header folgenden, aneinandergereihten Sections. Pro Section enthält die Tabelle einen Eintrag. Neben dem Namen der Section sind hier seine Größe und Lage sowohl in der Datei als auch im Speicher und die Art der Section angegeben. Der Name der Section ist frei wählbar, wobei .text für Code und .data für Daten sehr häufig vorkommen. Der Name .bss deutet auf uninitialisierte Daten hin; das bedeutet in den meisten Fällen, dass diese Section zunächst nur Nullen enthält. Der Name .rdata wird für Sections mit Read-Only-Daten verwendet. Des Weiteren ist noch der Name .rsrc für Sections mit Programmressourcen (z. B. Windows Icons) gebräuchlich, und .idata deutet auf die Import Address Table hin (s.u.).

Abb. 1.17: PE-Format

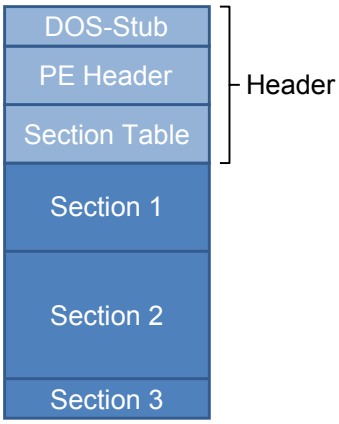
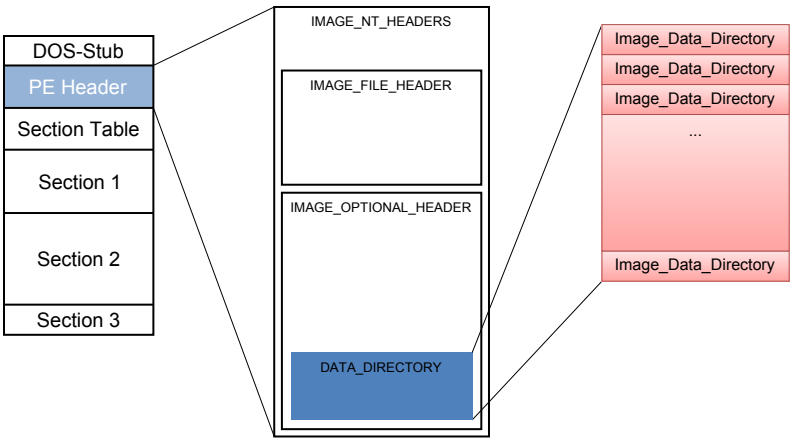


Abb. 1.18 zeigt etwas detaillierter den Aufbau des PE-Header. Der PE-Header ist aus zwei *Sub Headern* zusammengesetzt, dem *File Header* und den *Optional Header*.

Abb. 1.18: Aufbau PE-Header [Holz, 2012]



File Header Der File Header enthält unter anderem Informationen über die Anzahl der Sections, den erforderlichen CPU-Typ, die Größe des Optional Header sowie ein Flag, das

angibt, ob es sich bei der PE-Datei um eine Anwendung oder eine Bibliothek (DLL) handelt.

Optional Header

Der Optional Header enthält Informationen zu Umfang von Code und Daten, den Programmeinstiegspunkt (*Entry Point*) und die benötigten Alignment-Daten, um die Expansion der PE-Datei in den Speicher vorzunehmen. Des Weiteren findet sich hier die virtuelle Startdresse, ab der die PE-Datei normalerweise in den Speicher abgebildet werden soll, sofern diese nicht belegt ist.[13]

Data Directory

Das *Data Directory*, ein Array mit 16 Einträgen, ist ebenfalls im Optional Header zu finden. Es verweist auf Tabellen, die für das Laden und Ausführen des in der PE-Datei enthaltenen Programms unabdingbar sind. Von besonderer Wichtigkeit sind hierbei das *Export Directory*, das *Import Directory* und die *Import Address Table*. Import und Export Directory geben die zur Programmausführung zu importierenden Funktionen bzw. die vom Programm selbst exportierten Funktionen an.

Export Directory

Bibliotheken exportieren Funktionen und/oder Daten, die zusammenfassend als Symbole bezeichnet werden. Die exportierten Symbole werden über das Export Directory veröffentlicht. Neben mehreren Statusinformationen sind im Export Directory der Name der exportierenden Bibliothek, die (gewünschte) virtuelle Adresse im Speicher, die Anzahl der Exporte und Verweise auf die Tabellen mit den eigentlichen Exporten enthalten.

Import Directory

Das Import Directory enthält eine Liste der verwendeten Bibliotheken und der zu importierenden Symbole. Einträge in dieser Liste sind letztendlich entweder Namen oder Zahlen des jeweils korrespondierenden Exports. Es ist durchaus üblich, dass importierte Bibliotheken ihrerseits weitere Bibliotheken aufrufen. Beim Laden einer PE-Datei analysiert daher der Windows Loader *rekursiv* das Import Directory, lädt alle dabei referenzierten Bibliotheken und ermittelt die Adressen aller referenzierten Symbole. Diese Adressen landen in der *Import Address Table* (IAT). Der Aufruf importierter Funktionen erfolgt über die in der IAT hinterlegten Adressen.

1.6.7 Windows-Prozesse

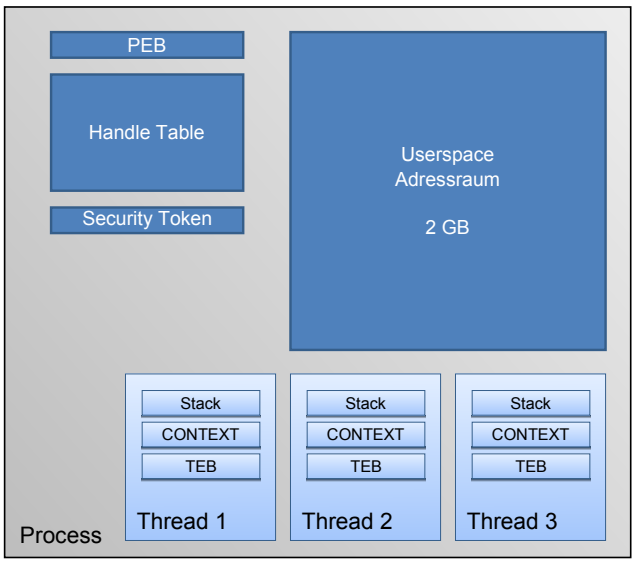


Abb. 1.19: Prozesskomponenten (Prozessrahmen)

[13] Wünschen sich z. B. 2 DLLs dieselbe Startadresse, so ist eine Relocation erforderlich.

Ein Windows-Prozess hat den in Abb. 1.19 gezeigten logischen Rahmenaufbau. Er beinhaltet einen eigenen Adressraum von 2 GB Größe im Userspace, die Verwaltungsdaten im PEB (*Process Environment Block*), die *Handle Table*, das *Security Token* und einen oder mehrere Threads. Die Threads wiederum bestehen aus Verwaltungsdaten im TEB (*Thread Environment Block*), einem eigenen Stack und dem Kontext (Instruction Pointer und Registerinhalte, die bei einem Thread(Prozess)-Wechsel aus bzw. in den physikalischen Prozessor entladen bzw. daraus geladen werden müssen). PEB und TEB werden in Abs. 1.6.7.1 detaillierter betrachtet.

Security Token

Das Security Token bzw. Access Token enthält Informationen über die Benutzerrechte, die sich aus der Art der Benutzeranmeldung bei Windows ergeben. Grob unterscheidet Windows bei der Programmausführung zwischen Administratoren mit vollen Rechten und Benutzern mit eingeschränkten Rechten. Das Security Token enthält einen Security Descriptor, der die Rechte eines Benutzers beschreibt. Damit kann bspw. die Ausführung bestimmter Teile einer Anwendung für normale User gesperrt werden.

Handle Table

Die Handle Table dient zur Kommunikation mit dem Kernel. Ein Handle ist ein eindeutiger Referenzwert zu einer vom Betriebssystem verwalteten System-Ressource, wie z. B. Bildschirmobjekte oder einzelnen Dateien auf Festplatten. Wenn ein Anwendungsprogramm eine solche Ressource verwenden will, erhält es durch den Aufruf einer geeigneten Systemfunktion (zum Beispiel zum Öffnen oder Erzeugen von Dateien) als Rückgabewert die Referenz, die zur weiteren Verwendung der Ressource durch Systemfunktionen anzugeben ist (etwa zum Lesen aus einer Datei). Die Handles eines Prozesses werden in der Handle Table verwaltet.

Es existieren zahlreiche API-Funktionen zur Prozesserstellung, wie z. B. *ShellExecuteA/W, WinExec, CreateProcessA/W, CreateProcessUserA/W* und *CreateProcessWithTokenA/W*, die letztendlich alle bei der API-Funktion *CreateProcessInternalW* landen.

Prozesserstellung

Bei der Prozesserstellung wird der gezeigte Prozessrahmen aufgebaut: Insbesondere wird der neue Prozessadressraum erstellt, Programmcode und Daten werden geladen, die System-DLL *ntdll.dll* wird eingebunden, der PEB aufgebaut und der Haupt-Thread wird erstellt (Stack und Kontext werden initialisiert und der TEB wird erstellt). Anschließend wird der Prozess beim Betriebssystem registriert und der Haupt-Thread gestartet.

Bis zu diesem Zeitpunkt wurde lediglich *ntdll.dll* in den Speicher geladen. Innerhalb des Haupt-Thread werden die zusätzlich benötigten DLLs (ggf. rekursiv) importiert, die Symbole aufgelöst und die IAT gefüllt. Schließlich wird der eigentliche *Entry Point* (EP) des Programms aufgerufen. In einem C-Programm wäre das die *main*-Funktion.

Ein Prozess läuft so lange, bis alle seine Threads beendet sind, oder bis er vorzeitig über die API-Funktionen *TerminateProcess* oder *ExitProcess* abgebrochen wird.

Abb. 1.20 zeigt das Speicher-Mapping des Prozesses aus Abb. 1.19. Im Userspace befinden sich alle Datenstrukturen, Code, globale Daten usw. Im Kernelspace liegen die Handle Table und die Exception Handler des Betriebssystems (s. Abs. 1.6.8). Die Handle Table liegt zwar im Kernelspace, wird aber in den Adressraum des Prozesses als *Read Only* eingeblendet. Eine Modifikation der Handle Table kann somit nur der Kernel vornehmen.

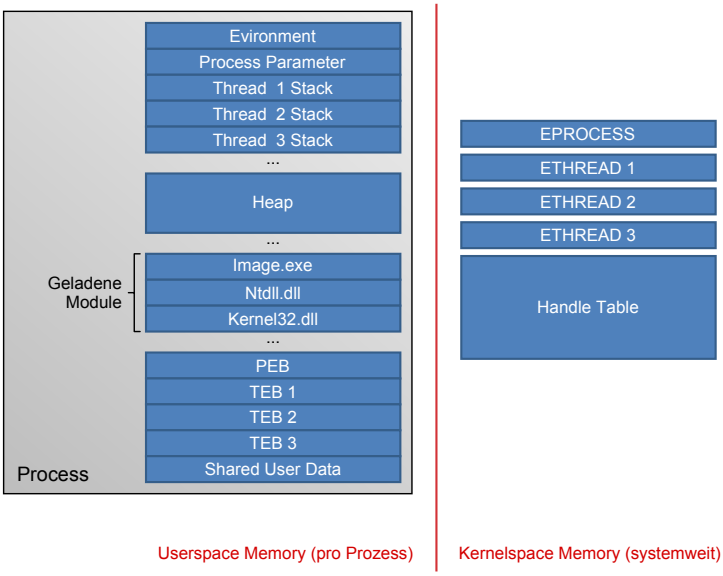


Abb. 1.20: Prozess im Speicher

1.6.7.1 PEB und TEB

Der Windows Loader ist in der *ntdll.dll* implementiert. Er wird immer dann aktiv, wenn ein neuer Prozess initialisiert wird, also insbesondere beim Starten eines Anwenderprogramms. Der Windows Loader initialisiert den Prozess, lädt (mittels der API-Funktion *LoadLibrary*) die benötigten DLLs in den Speicher und füllt die IAT. Die Prozessinitialisierung beinhaltet den Aufbau der Verwaltungsstrukturen des Prozesses und des initialen Thread.

Process Environment Block (PEB)

Die Verwaltungsstrukturen eines Prozesses sind im PEB (*Process Environment Block*) zusammengefasst, der alle zur Ausführung benötigten Prozess- und Loader-Informationen enthält.

Thread Environment Block (TEB)

Jeder Windows-Prozess bestimmt **einen** initialen Thread. Während der Prozessausführung können weitere Threads dynamisch erzeugt werden. Die Verwaltungsinformationen der einzelnen Threads sind im jeweiligen TEB (*Thread Environment Block*) zusammengefasst. Die TEBs verweisen auf den übergeordneten PEB (Abb. 1.21).

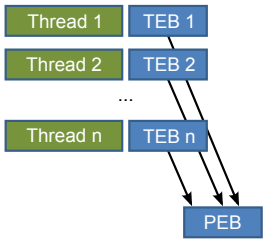


Abb. 1.21: TEB/PEB

Im TEB sind bspw. Informationen über Adresse und Größe des Stack, den TLS[14] (*Thread Local Storage*) und die Exception Handler (s. Abs. 1.6.8) abgelegt. PEB und TEB werden in der Regel nicht von den Anwendungen selbst, sondern nur vom Betriebssystem benutzt, das CPU-Register `fs`[15] zeigt auf den TEB, `fs:[0x30]` zeigt auf den PEB.

[14] TLS ist ein Speicherbereich für private Daten eines Thread. Speicher im TLS kann über API-Funktionen oder Konstrukte mancher Hochsprachen allokiert, benutzt und freigegeben werden.
[15] `fs` ist ein IA-32-Segmentregister und wird im nächsten Studienbrief erklärt.

1.6.8 Exceptions

Eine Exception ist ein Ereignis während eines Programmablaufs, das die Ausführung von Code außerhalb des normalen Kontrollflusses erfordert bzw. erzwingt. Es existieren zwei Arten von Exceptions: Hardware Exceptions und Software Exceptions.

- Hardware Exceptions werden von der CPU selbst initiiert, bspw. bei einer Division durch 0 oder durch den Zugriff auf eine ungültige Speicheradresse.
- Software Exceptions werden explizit durch die Anwendung selbst oder das Betriebssystem ausgelöst. Beispiele hierfür sind Breakpoints beim Debugging eines Programms oder ungültige Parameter bei einem Funktionsaufruf, die nicht zur Kompilierungszeit entdeckt werden konnten.

Exceptions erfordern eine Exception-Behandlung (*Exception Handling*), also die Ausführung einer Ausnahme- bzw. Fehlerbehandlungsroutine. Diese unterscheidet sich prinzipiell nicht für Software und Hardware Exceptions.

Aus der Sicht des Exception Handling sind drei Arten von Exceptions zu unterscheiden:

1. **Faults**
   Faults sind Exceptions **bei** der Ausführung einer Operation. Der „Fehler" kann durch den Exception Handler behoben werden, und die Operation wird danach wiederholt.
2. **Traps**
   Traps sind Exceptions **nach** der Ausführung einer Operation. Der „Fehler" kann durch den Exception Handler behoben werden, und die nächste Operation wird danach ausgeführt.
3. **Aborts**
   Aborts sind kritische Fehler, die zu einem Prozessabbruch führen.

Exception Handler

Exceptions werden von einem *Exception Handler* abgewickelt. Dies sind allgemein Routinen, die auf eine Ausnahmesituation oder einen Fehlerfall in einem Prozess angemessen reagieren sollen.

Structured Exception Handling

Windows bietet das sogenannte *Structured Exception Handling* (SEH) an. Da der Ort (im Programmcode), an dem eine Exception ausgelöst wird, nicht voraussagbar ist, sind Anwendungscode und Exception Handler voneinander getrennt. Ohne weitere Vorkehrungen sind Exception Handler dann Routinen im Kernelmode, die die Standardbehandlung von Exceptions beinhalten[16]. Die Exception selbst wird durch einen Exceptioncode (Art der Exception) und Kontextinformationen (Instruction Pointer und andere CPU-Register) spezifiziert.

SEH bietet die Möglichkeit, bestimmte Exceptions direkt im Programm abzufangen. Mit entsprechenden Hochsprachenkonstrukten können Code Blocks durch einen eigenen, zugeordneten Exception Handler „geschützt" werden. Geschützt ist der Code Block dann in dem Sinne, dass der Standard-Exception-Handler zunächst außen vor bleibt.

[16] und die meist nichtssagenden Windows-Fehlermeldungen erzeugen.

B

Beispiel 1.6

Nehmen wir den Fall einer Division durch 0 an. Was soll bei einem solchen Fehler passieren? Der Standard-Exception-Handler von Windows würde vermutlich den Prozess in dem Fall beenden. Das Programm sollte die Möglichkeit haben, diese Ausnahme selbst zu bedienen und eventuell „milder“ zu reagieren. In C++ bspw. existiert das **try-catch Statement**, mit dessen Hilfe ein eigener Handler für diesen Fall programmiert werden kann.

Abb. 1.22 zeigt eine solche Situation mit zwei Code Blocks und den zugeordneten Exception Handlern.

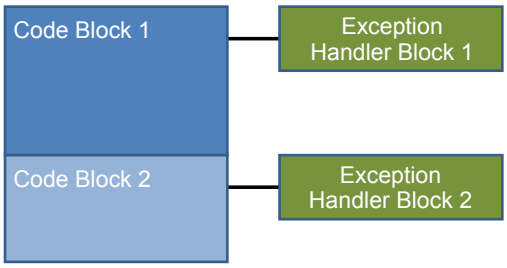


Abb. 1.22: Structured Exception Handling (SEH)

SEH ist in Form einer verketteten Liste implementiert. Ein Eintrag im TEB zeigt auf das erste Listenelement, wie dies in Abb. 1.23 dargestellt ist. Die Liste besteht aus *Exception Registration Records*. Diese Records verweisen auf den nächsten Record und den eigentlichen Exception Handler. `0xffffffff` zeigt das Ende der Liste an.

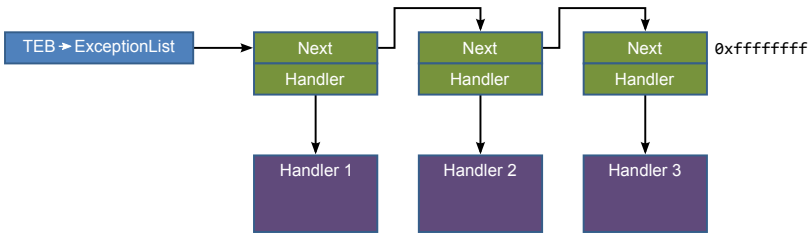


Abb. 1.23: Implementierung SEH

Frame Based Exception Handling

Eine Erweiterung von SEH ist das *Frame Based Exception Handling* (FBEH). Wird bspw. eine Exception innerhalb einer verschachtelt aufgerufenen Funktion ausgelöst, so kann der zugeordnete Exception Handler in diesem Kontext nicht immer die Exception angemessen bedienen. Er gibt die Kontrolle daher an den Exception Handler der aufrufenden Funktion weiter. Dieses Verfahren wird ggf. rekursiv fortgesetzt. Die Realisierung erfolgt über einen *Exception Handler Stack*. Dies ist in Abb. 1.24 anhand eines Beispiels dargestellt. Wenn Code Block 3 eine Exception auslöst, dann versucht zunächst Handler 3, diese zu bedienen. Gelingt dies nicht, so wird die Exception-Behandlung an Handler 2 des aufrufenden Code Block 2 weitergereicht, dann eventuell an Handler 1. Erst wenn alle Handler des Stack die Exception nicht bedienen können, wird der Standard-Exception-Handler aktiviert, also der von Windows.

Vectored Exception Handling

Eine andere Erweiterung von SEH ist das *Vectored Exception Handling* (VEH). Hierbei beziehen sich die Exception Handler nicht auf Code Blocks, sondern auf den gesamten Thread. Realisiert wird VEH durch eine beliebig lange Liste von Handlern, die bei einer Exception der Reihe nach aufgerufen werden. Werden VEH

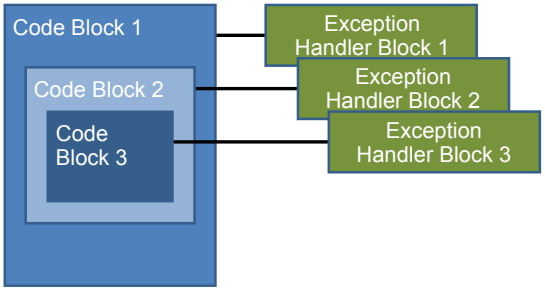


Abb. 1.24: Frame Based Exception Handling (FBEH)

und FBEH/SEH gleichzeitig implementiert, so werden zunächst die VEH Handler aufgerufen, dann die von FBEH/SEH.

1.7 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.

Einige Aspekte einer konkreten Implementierung von Betriebssystemen haben wir am Beispiel von Microsoft Windows kennengelernt. Für die systemnahe Programmierung von Windows-Rechnern sind die folgenden Punkte sehr wichtig:

- Programme, die auf Windows-Rechnern funktionieren sollen, müssen mit dem Betriebssystem interagieren. Dies geschieht durch den Aufruf von Bibliotheksfunktionen über eine klar definierte Schnittstelle (API). Systemaufrufe (Syscalls) bilden dabei den Übergang in den eigentlichen Betriebssystemkern.
- Windows-Programme benutzen ein einheitliches Format (PE-Format). Darin sind in einer streng strukturierten Weise neben dem Programm selbst alle Informationen enthalten, um aus einem Programm einen Prozess zu erzeugen.
- Der Windows Loader macht aus einer Datei im PE-Format einen Prozess. Die dabei entstehenden Speicher- und Datenstrukturen sind für die Dynamik eines Programms, also seines Laufzeitverhaltens, wesentlich.
- Windows benutzt mit SEH eine besondere Form der Ausnahmebehandlung.

1.8 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 Maschinencode. Wichtig: Ein Programm bleibt nicht „stehen“, wenn es eine Sprungmarke erreicht. Sprungmarken erzeugen keinen Programmcode.)

Ü

Übung 1.3

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.)

Ü

Übung 1.4

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.

Die folgenden Übungen sollen Sie mit der API-Schnittstelle von Windows besser vertraut machen.

Ü

Übung 1.5

Schreiben Sie ein Programm in C, das eine beliebige Datei einliest und deren Inhalt in eine andere Datei ausgibt. Die Länge der Eingabedatei ist hierbei nicht beschränkt. Verwenden sie dafür ausschließlich die API-Funktionen von Windows (*CreateFile, ReadFile*, etc.). API-Funktionen liefern meist Fehlercodes, wenn z.B. der Zugriff auf eine Datei nicht möglich ist. Berücksichtigen Sie solche Fehlercodes in Ihrem Programm.

Hinweis: Wie im Beispielprogramm in Abs. 1.6.3 beschrieben muss `windows.h` eingebunden werden. Deklarationen und Erklärungen zu den benötigten API-Funktionen sind leicht im Internet zu finden.

Ü

Übung 1.6

Schreiben Sie ein Programm in C, das die API-Funktion *IsDebuggerPresent* verwendet. Diese Funktion überprüft, ob ein Programm unter einem Debugger – möglicherweise zu Analysezwecken – ausgeführt wird. Wenn das Programm einen Debugger entdeckt, soll es irreführende Dinge tun, ansonsten soll es irgendwelche „vernünftigen“ Ergebnisse produzieren. Was das Programm jeweils produziert, bleibt Ihrer Fantasie überlassen.

Liste der Lösungen zu den Kontrollaufgaben

Lösung zu Kontrollaufgabe 1.1 auf Seite 16

Über einen 42 Bit breiten Adressbus können $2^{42} = 4398046511104$ Byte adressiert werden.

Lösung zu Kontrollaufgabe 1.2 auf Seite 23

- **Kernelland**
  Das Kernelland ist die Menge aller ausführbaren Dateien, welche mit uneingeschränkten Rechten laufen. Dazu zählen insbesondere die elementaren Betriebssystemdienste und die Hardware-Treiber.
- **Kernelmode**
  Der Kernelmode ist die Privilegienstufe der CPU mit vollen Rechten im Ring 0.
- **Kernelspace**
  Der Kernelspace gibt den Adressbereich an, auf den voll privilegierte Programme zugreifen dürfen.

Lösung zu Kontrollaufgabe 1.3 auf Seite 32

Die Nachteile der statischen Bindung sind die Vorteile der dynamischen Bindung und umgekehrt:

Die Nachteile einer dynamischen Bindung sind das langsame Laden und Starten einer Anwendung und die Abhängigkeit vom Vorhandensein einzelner Bibliotheken. Die Vorteile bestehen neben der möglichen Mehrfachverwendung von Bibliotheken zum einen in einer kleineren, kompakteren Anwendungsdatei und zum anderen in der entfallenden Notwendigkeit, die Anwendungen bei einem Update der Bibliothek neu zu kompilieren.

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.

Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. *16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

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, 2012.

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.

Tobias Klein. *Aus dem Tagebuch eines Bughunters. Wie man Softwareschwachstellen aufspürt und behebt*. dpunkt Verlag, 2010.

Microsoft. Microsoft pe and coff specification.
http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx, 2010.

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

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

Matt Pietrek. An in-depth look into the win32 portable executable file format. *February 2002 issue of MSDN Magazine*, 2002.
http://msdn.microsoft.com/en-us/magazine/bb985992.aspx.

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.

Rolf Rolles. Binary literacy – optimizations.
http://www.openrce.org/repositories/users/RolfRolles/Binary%20Literacy%20-%20Static%20-%206%20-%20Optimizations.ppt, 2007.

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

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

Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

Sebastian Strohhäcker. Malicious code: Code obfuscation.
www.reverse-engineering.info/OBF/strohhaecker.pdf, 2004.

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

Stichwörter