1 Einführung

Bei der Arbeit am Computer machen sich viele Anwender erst beim Auftreten rätselhafter Fehlerneldungen Gedanken darüber, wer oder was diese verursachen könnte – liegt es nicht an einem Programm, fällt schnell der Verdacht auf die Schicht unterhalb der Anwendungen: auf das Betriebssystem.

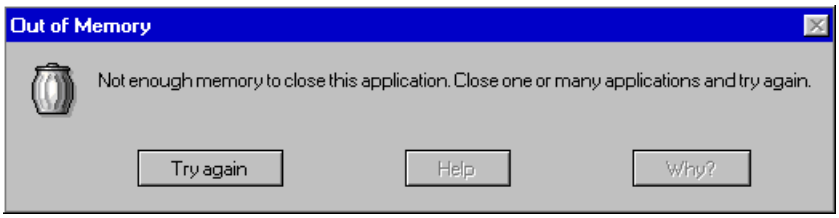

Abbildung 1.1: Solche Fehlermeldungen wollen Betriebssystem-Entwickler vermeiden. (Bild: Error Message Generator, `http://atom.smasher.org/error/`

Mit welchen Techniken moderne Betriebssysteme (wie Windows, Linux, Mac OS, BSD ...) versuchen, Problemen aus dem Weg zu gehen, ist Gegenstand der Vorlesungen Betriebssysteme I und II an der FH München.

1.1 Motivation: Aufgaben eines Betriebssystems

Das Betriebssystem ist das erste „Programm“, das ein Computer beim Booten lädt[1]. Nachdem der Rechner es in den Hauptspeicher geladen und ihm die Kontrolle übergeben hat, beginnt es mit der Initialisierung wichtiger Kernkomponenten des Computers, wie z. B. des Hauptspeichers.

[1] Ob die im ROM des Rechners liegende Software, die das Betriebssystem startet (beim PC das BIOS), das erste Programm und das Betriebssystem damit das zweite ist, ist eine Auslegungsfrage.

1.1.1 Hardware-Abstraktion

Die Hauptaufgabe ist dabei, eine Abstraktionsschicht zu schaffen, die es Anwendungsprogrammen erlaubt, die Computerressourcen zu nutzen, ohne selbst genau zu wissen, wie der Computer beschaffen ist; welche spezifischen Einzelkomponenten vorhanden sind.

Beispiel 1.1 Unter MS-DOS gab es in den 80er und frühen 90er Jahren verschiedene Programme, die eine grafische Ausgabe besaßen, beispielsweise das Textverarbeitungsprogramm Microsoft Word (nicht Word für Windows) und die grafische Arbeitsoberfläche GeoWorks. Beide Programme brachten Grafikkartentreiber für zahlreiche damals üblichen Grafikkarten mit; neue Grafikkarten wurden von beiden Programmen erst in neueren Programmversionen unterstützt – weil MS-DOS keine Abstraktionsschicht für die Benutzung grafischer Kartenbetriebsarten enthielt, musste jedes grafische Programm seine eigenen Kartentreiber enthalten. ■

Die Abstraktion verschiedener Hardwarekomponenten durch das Betriebssystem erleichtert nicht nur das Programmieren von Anwendungsprogrammen, es verringert auch das Risiko eines Systemabsturzes, der durch fehlerhafte Ansteuerung der Hardware verursacht wird. Erlaubt man Anwendungsprogrammen den direkten Zugriff auf Hardware (im Beispiel 1.1 auf die Grafikkarte), stellt jedes neue Programm ein potenzielles Risiko dar, weil es möglicherweise fehlerhaft oder gar zerstörerisch auf die Hardware zugreift.

Moderne Betriebssysteme verhindern direkte Zugriffe auf die Hardware und bieten stattdessen Betriebssystemschnittstellen an, über die die Anwendungen auf die Hardware zugreifen. Ist in dieser Situation das Anwendungsprogramm fehlerhaft programmiert und ruft diese Schnittstellen mit unsinnigen Parametern auf, kann das Betriebssystem dies erkennen und den Zugriff ablehnen.

Beispiel 1.2 Ist am Computer ein Drucker angeschlossen und wollen zwei Anwendungsprogramme gleichzeitig eine Seite Papier ausdrucken, käme es zu einem chaotischen Ergebnis, wenn beide Programme zur selben Zeit mit dem Ausdruck beginnen würden: Mit den abwechselnd von Programm A und B beim Drucker eintreffenden Datenfragmenten könnte das Gerät nichts anfangen.

Kontrolliert das Betriebssystem den Drucker, übergeben die Anwendungen ihre Druckaufträge dem Betriebssystem, und dieses sorgt dann dafür, dass die Druckdaten in einer vernünftigen Reihenfolge an den Drucker weiter geleitet werden. Dazu kommt der bereits bekannte Nutzen, dass nur an einer Stelle Wissen über die Ansteuerung der verschiedenen Druckermodelle gesammelt werden muss: im Betriebssystem. Die Anwendungen schicken ihre Druckdaten in einem standardisierten Format, und das Betriebssystem setzt sie mit einem Druckertreiber in die Sprache um, die das Gerät versteht, z. B. PostScript. ■

Beispiel 1.2 zeigt einen zweiten Vorteil der Hardware-Abstraktion: Sie schützt Geräte vor gleichzeitigem Zugriff durch mehrere Anwendungen – denn manche Dinge brauchen exklusiven Zugriff, wie z. B. der Ausdruck zweier Dokumente aus zwei Programmen.

1.1.2 Speicherschutz und Virtual Memory

Laufen auf einem Rechner gleichzeitig mehrere Prozesse (von eventuell verschiedenen Anwendern), hat das Betriebssystem auch die Aufgabe, die im Hauptspeicher liegenden Datenbereiche dieser Prozesse voneinander zu trennen, also den Zugriff eines Programms auf den Datenbereich eines anderen zu trennen. Zur gleichen Thematik gehört auch die Möglichkeit, das Überschreiben (Abändern) von Teilen des Programmcodes im Speicher zu verhindern.

Diese Aufgaben nehmen Betriebssysteme im Rahmen der Speicherverwaltung (Memory Management) wahr.

Überhaupt mehrere Programme und deren Daten im Speicher zu halten, wirft recht schnell die Frage auf, wie man mit knappem Hauptspeicher umgehen kann. Hier kommt *virtual memory* ins Spiel: Wie auch bei der übrigen Hardware, virtualisieren einige Betriebssysteme zudem den Speicher. Das heißt, sie stellen den Programmen virtuellen Speicher zur Verfügung. Die Prozesse greifen dabei auf Speicherbereiche zu, die nichts mit dem physikalisch vorhandenen Speicher zu tun haben, und das Betriebssystem (mit Unterstützung durch die Memory Management Unit, eine Komponente der CPU) biegt alle Zugriffe auf diesen virtuellen Speicher passend um – selbst dann, wenn der Speicher gerade aus Platzmangel auf die Festplatte ausgelagert wurde (Swap).

1.2 Einsatzgebiete

Das bekannteste Einsatzgebiet von Betriebssystemen, nämlich der Betrieb von Desktop- und Server-PCs, ist nur eines von vielen. Dieser Abschnitt stellt noch einige alternative Verwendungsmöglichkeiten vor.

Je nach Zweck muss ein Betriebssystem ganz unterschiedliche Eigenschaften haben. So sind z. B. Echtzeitfähigkeit und hohe Performance zwei Eigenschaften, die völlig unabhängig voneinander sind:

- Unter Echtzeitfähigkeit versteht man (vereinfacht gesagt) die Eigenschaft, dass bestimmte Aufgaben in einer garantierten Maximalzeit erledigt werden, dass also beispielsweise das System auf einen Hardware-Interrupt in einer garantierten Zeit reagiert.

- Hohe Performance bedeutet hingegen, dass sich das Gesamtsystem „schnell" verhält (wie auch immer man das definieren mag) – dabei kann es durchaus vorkommen, dass einzelne Aufgaben auch länger brauchen, bis das System sie erledigt hat; nur der Gesamteindruck ist der eines schnellen Systems.

Aus keiner der beiden Eigenschaften folgt die jeweils andere, und sie schließen sich auch nicht aus.

1.2.1 Desktop-Computer

Die zahlreichen Einsatzgebiete eines normalen PCs, der auf dem privaten Schreibtisch, an der Hochschule oder am Arbeitsplatz steht, sind bekannt. Damit ein Betriebssystem die Anforderungen an solche Systeme erfüllen kann, muss es multitasking-fähig sein, gut mit den diversen Peripheriegeräten umgehen können und die heute üblichen Netzwerk- und Internetanbindungen (Modem, ISDN, DSL, Kabel etc.) unterstützen. Vor allem die Mainstream-Betriebssysteme Windows, Linux und Mac OS sind hier anzutreffen, die Hardware basiert meist auf Intel- oder kompatiblen Prozessoren.[2]

Eine zentrale Eigenschaft ist hier die Interaktivität: Das Betriebssystem darf nicht unerträglich viel Zeit vergehen lassen, bevor es Anwenderinteraktion (wie Drücken einer Taste, Bewegen der Maus oder Klicken) an den zuständigen Prozess weiter leitet – schon Verzögerungen im Bereich von Sekundenbruchteilen können ein System ungeeignet für interaktive Aufgaben machen.

1.2.2 Server-PCs

Bei Servern ist das Hardwarespektum breiter, aber die zahlreichen Blade-Systeme in den diversen Webhoster-Rechenzentren unterscheiden sich (vom Aufbau abgesehen) nicht wesentlich von normalen Desktop-PCs.

Typische Anwendungsgebiete für Server sind Web-, FTP-, Mail-Server im Internet oder Intranet und Datenbank-Server. Zudem gibt es das weite Feld des *High Performance Computing*, bei dem besonders leistungsstarke Rechner (oder aus zahlreichen PCs zusammengesetzte Cluster) zur Lösung mathematisch/numerischer Probleme eingesetzt werden.

[2] Apple steht kurz davor, seinen Migrationsprozess von der PowerPC-Architektur auf Intel-Prozessoren abzuschließen; Mac OS ist zur Zeit noch für beide Architekturen erhältlich.

1.2.3 Industrieanwendungen

Hier verlassen wir die Sphäre der „normalen" PCs. Industrierechner benutzt man beispielsweise für die Robotersteuerung (Abbildung 1.2), in automatischen Navigationssystemen und bei der Temperaturregelung. Die komplexen elektrischen Anlagen in heutigen Autos haben oft eine Motorenkontrolle, und auch Herzschrittmacher sind Computer.

Abbildung 1.2: Auch Rechner für Robotersteuerungen arbeiten mit einem Betriebssystem. (Bild: Wikipedia, KUKA Schweißanlagen)

Wichtig sind hier in den meisten Fällen Echtzeitfähigkeiten. Die genauso benannten Betriebssysteme (englisch: *real time operating system*) versprechen, Deadlines möglichst häufig einzuhalten.

1.2.4 Embedded-Systeme

Das Wort *embedded* bedeutet „eingebettet": Das deutet darauf hin, dass man Embedded-Systeme meist gar nicht als Computer wahrnimmt. Wer zum Handy greift und die Wahlwiederholungstaste drückt oder einer Nummer aus dem Adressbuch abruft, denkt nicht darüber nach, dass das Handy vielleicht einen Telefonbuchprozess startet, der Datensätze aus einer Datei im Dateisystem liest, das sich auf einem Chip befindet ...

Beispiele für Einsatzgebiete von Embedded-Systemen sind neben den Mobiltelefonen andere *gadgets* wie PDAs, mobile MP3-/Video-Player, Fernseher, Videorekor-

der, DVD-Player, Netzwerk-, DSL- und WLAN-Router, Taschenrechner, Videospiel-Konsolen, Geldautomaten usw. Viele dieser Geräte gab es irgendwann in simplen Varianten mit fest verdrahterer Steuerung, etwa Taschenrechner, welche die wenigen Rechenfunktionen fest auf einem Chip eingebaut hatten. Mit zunehmender Komplexität vieler Geräte ist dieser Ansatz aber nicht mehr geeignet, so dass der Einsatz von Betriebssystemen nötig ist.

1.3 Software-Entwicklung

Sich mit den Grundlagen von Betriebssystemen zu beschäftigen, ist aus verschiedenen Gründen sinnvoll. Zum Einen ist die Thematik selbst spannend, zum Anderen hat ein Verständnis der Zusammenhänge aber auch Auswirkungen auf die Arbeit von Software-Entwicklern.

Beim Programmieren kämpfen Entwickler regelmäßig mit zwei Problembereichen: Zuverlässigkeit und Sicherheit.

- Bei der Zuverlässigkeit stellt man an Software die Anforderung, genau das zu tun, was der Entwickler vorgesehen hat – also kein unerwartetes Verhalten zu zeigen. Dazu gehört zum Beispiel die Toleranz von Benutzerfehlern.
- Sicherheit bedeutet vereinfacht, dass sich das Programm (durch böswillige Anwender) nicht zu anderen als den vorgesehen Zwecken (miss-) brauchen lässt.

Treten Fehler auf, können diese schon im Betriebssystem liegen (und müssen dann zur Not umgangen werden) oder sie können darin begründet sein, dass der Programmierer Betriebssystemfunktionen auf eine falsche Weise verwendet.

Typische Programmfehler sind *race conditions* und *buffer overflows*:

- Bei einer *race condition* greifen mehrere Programme (Prozesse) quasi-gleichzeitig auf einen Datenbereich zu, und das Ergebnis einer Berechnung oder eine Entscheidung hängt davon ab, in welcher konkreten Reihenfolge der Zugriff erfolgt. Durch richtige Programmierung kann man sicherstellen, dass die Ausführreihenfolgen keine Rolle spielen – dann ist ein Programm frei von *race conditions*.
- Tritt ein *buffer overflow* (Pufferüberlauf) auf, hat der Programmierer zugelassen, dass beim Einlesen von Daten (aus einer Datei oder als direkte Eingabe durch den Benutzer) eine Array-Variable über ihre Feldgrenzen hinaus beschrieben wird – hinter diesem Array liegen aber andere Daten, und die werden damit überschrieben.

Etliche Sicherheitslücken und *exploits* basieren auf einem dieser beiden Programmierfehler. Wer versteht, wie das Betriebssystem arbeitet, ist auch eher motiviert, diese Fehler zu vermeiden.

1.4 Voraussetzungen

Für die Teilnahme an der Vorlesung und/oder die Lektüre dieses Buchs werden Kenntnisse in den folgenden Bereichen vorausgesetzt:

- **C** – Grundlagen der Programmierung in C, so dass Sie kurze C-Programme lesen können
- **Rechnerarchitekturen** – grober Aufbau eines Computers (Prozessor, Hauptspeicher, Peripherie etc.), Zusammenwirken dieser Komponenten (Bussysteme, Interrupts etc.)

Als Teil der Vorlesung wird ein kurzes Programmierpraktikum angeboten; für die Teilnahme daran sind einige zusätzliche Kenntnisse nötig:

- **Unix-Shell** – Benutzung der Standard-Shell `bash` unter Linux (oder einem anderen Unix-System)
- **Python** – Grundlagen der Programmierung in Python, so dass Sie Python-Programme lesen können. Für das Praktikum sind auch „aktive" Programmierkenntnisse in Python notwendig. Eine kurze Einführung in Python finden Sie am Ende des Buchs in Anhang A.

1.5 Aufbau und Inhalt dieses Buchs

In diesem Buch behandeln wir, soweit es möglich und sinnvoll ist, alle Themen nach folgendem Schema:

1. Am Anfang stehen praktische Beispiele unter dem Betriebssystem Linux[3] – z. B. finden Sie im nächsten Kapitel über Prozesse zunächst eine Beschreibung, wie Sie Prozesse unter Linux starten und manipulieren (anhalten, fortsetzen, abschießen etc.).

[3] `http://www.linux.org`

2. Im zweiten Schritt lernen Sie die theoretischen Grundlagen kennen. Da es für viele Probleme unterschiedliche Lösungsansätze gibt, erfahren Sie hier einiges über die Gestaltungsmöglichkeiten, die ein Betriebssystemprogrammierer hat.
3. Danach wird es wieder praktisch: Anhand von Linux-Quellcode zeigen wir Implementierungen der vorgestellten Theorie. So schließt sich der Kreis, und Sie sehen, warum die Befehle und Vorgehensweisen aus dem ersten Teil funktionieren.
4. Jedes Kapitel endet mit einem kurzen Übungsblock: Anhand der Selbsttestfragen (deren Antworten Sie im Anhang C finden) können Sie Ihr Wissen überprüfen.

Die Themen präsentieren wir in der gleichen Reihenfolge, in der sie auch in der Vorlesung behandelt werden.

- Kapitel 2 führt in das Konzept der Prozesse und Threads ein und beschreibt neben der Theorie auch Implementierungsdetails, etwa den Aufbau von Prozesstabellen in echten Betriebssystemen.
- In Kapitel 3 geht es um Interrupts, also Unterbrechungen des gerade laufenden Prozesses.
- Scheduler behandelt Kapitel 4: Sie sorgen dafür, dass die Prozessorzeit gerecht unter den Prozessen verteilt wird.
- Kapitel 5 beschäftigt sich mit dem Thema Synchronisation.
- Interprozess-Kommunikation (englisch: *inter-process communication*, kurz IPC) ist Gegenstand des Kapitels 6.
- Generell zu vermeiden sind Deadlocks: Wie es zu Deadlocks kommt und mit welchen Vorkehrungen man ihnen aus dem Weg geht, verrät Kapitel 7.
- Alleiniger Gegenstand der Vorlesung Betriebssysteme II ist die Speicherverwaltung, die Kapitel 8 behandelt.

1.6 Über den Autor

Hans-Georg Eßer ist Diplom-Informatiker und Diplom-Mathematiker (beide Abschlüsse von der RWTH Aachen), Autor mehrerer Fachbücher[4] sowie hunderter Zeitschriftenbeiträge und seit 2000 Chefredakteur einer Computerzeitschrift. Im Wintersemester 2006/07 hält er als Lehrbeauftragter die Vorlesungen Betriebssysteme I und II an der Fachhochschule München.

1.7 Feedback, Errata

Dieses Buch ist in Teilen parallel zur Vorlesung entstanden. Die jeweils aktuellste Fassung finden Sie auf der Homepage zur Vorlesung unter `http://fhm.hgesser.de/`.

Über Hinweise zu Fehlern im Buch (oder in den Vorlesungsfolien) und kritische Anmerkungen (etwa zur Verständlichkeit einzelner Themen) würde ich mich sehr freuen – bitte per E-Mail an `fhm@hgesser.de`.

[4] `http://www.esser-books.de/`