t.

TRAUNER VERLAG
UNIVERSITÄT

Leseprobe

Schriftenreihe
Informatik

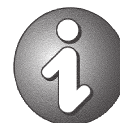
35

2. erweiterte und überarbeitete Auflage 2011

Jörg R. Mühlbacher

Betriebssysteme

Grundlagen



Impressum

Schriftenreihe Informatik

Jörg R. Mühlbacher
Betriebssysteme
Grundlagen
2. erweiterte und überarbeitete Auflage 2011

Leseprobe

© 2011
Das Werk und seine Teile sind
urheberrechtlich geschützt.

Reihe Informatik und Wirtschafts-
informatik herausgegeben von
den Fachbereichen Informatik und
Wirtschaftsinformatik der
Johannes Kepler Universität Linz
4040 Linz, Österreich/Austria

Herstellung:
TRAUNER Druck GmbH & Co KG,
4020 Linz, Köglstraße 14,
Österreich/Austria

ISBN 978-3-85499-843-3
www.trauner.at

Jörg R. Mühlbacher

Betriebssysteme

Grundlagen

Leseprobe

Universitätsverlag Rudolf Trauner

Leseprobe

Hinweise:

Im vorliegenden Text wird aus Gründen der Lesbarkeit – z.B. anstatt *die Benutzerin und der Benutzer* oder *der Programmierer und die Programmiererin* – häufig nur die männliche Form benutzt. Es ist aber ein Anliegen, ausdrücklich zu betonen, dass damit keinerlei geschlechtsspezifische Absicht verbunden ist.

Abbildungen in diesem Buch können von der Originaldarstellung abweichen, da aus Platzgründen häufig Verkleinerungen durchgeführt wurden.

VORWORT

Dieses Buch behandelt den Stoff der Vorlesung „Betriebssysteme“, die für Studierende der Informatik (2. Semester, Bakkalaureat) an der TNF der Johannes Kepler Universität (JKU) abgehalten wird und ebenso von Studierenden der Wirtschaftsinformatik der JKU besucht wird.

Es stehen daher Grundlagen von Betriebssystemen im Vordergrund, im Einzelfall werden Beispiele (meist) anhand der Windows-Familie herangezogen.

Um bei der Beschreibung von Algorithmen von einer konkreten Programmiersprache unabhängig zu bleiben, wurde ein Pseudocode gewählt, der in seiner Syntax allerdings eng an JAVA bzw. C angelehnt ist.

Neben Überarbeitungen insbesondere der Abschnitte über Benutzerschnittstellen, Sicherheit und Zugangskontrollen, wurde das Buch um eine Einführung in Virtualisierung und deren Anwendungen erweitert. Im Kapitel über Netzwerkkommunikation mit dem Schwerpunkt IPv4 wurde eine kurze Beschreibung von IPv6 hinzugefügt.

Das Thema Parallelität wird weitgehend unabhängig vom Thema Betriebssysteme dargestellt, sodass der Einfluss der Entwicklung von Betriebssystemen auf Methoden im Softwareengineering sichtbar wird, denn die Prinzipien des Zusammenspiels von Prozessen in einem Betriebssystem gelten allgemein, wenn ein paralleler Zugriff auf gemeinsame Ressourcen notwendig wird.

Das Kapitel über Sicherheit geht über den Vorlesungsstoff hinaus und findet in einer entsprechenden weiterführenden Lehrveranstaltung seinen Platz. Seine Hereinnahme in dieses einführende Buch soll auch unterstreichen, dass Sicherheitsfragen in heutigen Systemen immer wichtiger werden.

Der Text ist so ausgelegt, dass er auch als Basis im Rahmen eines „Blended Learning“-Ansatzes bei E-Learning verwendet werden kann. Daher enthalten einige Kapitel kurze Begriffswiederholungen aus voran gegangenen Abschnitten, um sie bei Bedarf auch unabhängig von anderen durcharbeiten zu können.

Leseprobe

Besonders danken möchte ich Herrn Dr. Rudolf Hörmanseder für die intensive Diskussion und für viele Vorschläge. Mit ihm wurde das Kapitel über Netzwerkkommunikation abgestimmt. Dieses Thema wird in seinem zwischenzeitlich im selben Verlag erschienenen Folgebuch über Computernetzwerke ausführlich behandelt. Daher beschränkt sich dieser Abschnitt auf das Wichtigste, um die Netzwerkschichten eines Betriebssystems verstehen zu können.

Herzlicher Dank gebührt Frau Inge Naderer für das Tippen des Manuskriptes, sowie Frau Sabine Link und Herrn Dipl.-Ing. Christian Praher für Korrekturen und Layout in der Endphase vor der Drucklegung.

Linz, im Februar 2011 Jörg R. Mühlbacher

INHALTSVERZEICHNIS

Vorwort ... 3
Inhaltsverzeichnis ... 4
A Betriebssysteme erste Grundlagen ... 9
A.1 Was ist ein Betriebssystem? ... 9
A.2 Zweck und Aufgaben eines Betriebssystems ... 10
A.2.1 Prozessmanagement ... 10
A.2.2 Hauptspeicherverwaltung ... 11
A.2.3 Dateiverwaltung ... 12
A.2.4 Benutzerschnittstelle ... 12
A.2.5 Netzwerkanbindung ... 15
A.2.6 Betriebssystemaufbau ... 15
A.2.7 Betriebsmittel (Ressourcen) ... 16
B Hardware Mechanismen ... 17
B.1 Allgemeines ... 17
B.2 Von-Neumann-Zyklus ... 18
B.3 Laden des Betriebssystems, Reset-Vektor ... 19
B.4 Ein-/Ausgabe ... 20
B.5 Polling ... 20
B.6 Interrupts ... 21
B.6.1 Grundlagen von Interrupts ... 21
B.6.2 Von-Neumann-Zyklus im Detail ... 22
B.6.3 Aufbau einer Interrupt-Service-Routine (ISR) ... 24
B.6.4 Software Interrupts (SWI) ... 24
B.7 Prozeduren und Prozeduraufrufe ... 25
B.7.1 Vergleich Interrupts, Software-Interrupts, Prozeduren ... 25
B.7.2 Prozeduren und ihre Parameterversorgung ... 26
B.7.2.1 Prozeduraufrufe generell ... 26
B.7.2.2 Rückgabe eines Funktionswertes ... 27
B.7.2.3 Parameter über Call by Value ... 28
B.7.2.4 Parameter über Call by Reference ... 29
B.7.2.5 Reale Implementierungen ... 29
C Betriebsarten ... 31
C.1 Stapelverarbeitung (Batchprocessing) ... 31
C.2 Mehrprogrammbetrieb (Multiprogramming, Multitasking) ... 32
C.3 Teilnehmerbetrieb (Timesharing) ... 33
C.4 Echtzeitverarbeitung (Real-Time Processing) ... 33
C.5 Gridcomputing ... 34
D Klassifikation von Betriebssystemen ... 37
D.1 Betriebsarten ... 37
D.2 CPU-Scheduling ... 37
D.3 Technischer Aufbau (Architektur) ... 39
D.3.1 Monolithische Betriebssysteme ... 39
D.3.2 Geschichtete Betriebssysteme (Layered Operating Systems) ... 40
D.3.3 Client Server Modell ... 41
D.4 Speicherverwaltung ... 43
D.5 Benutzersicht ... 43
D.6 Virtuelle Maschinen ... 44
D.6.1 Grundprinzip ... 44

Leseprobe

D.6.2 Systemaufrufe (system calls) über Unterprogramme (Prozeduren) ... 46
D.6.3 Systemaufrufe mittels Software-Interrupts (SWI) ... 46
D.6.4 Hypervisor ... 47
D.6.4.1 Aufgabenstellung und Anwendung ... 47
D.6.4.2 Hosted Hypervisor („Typ 2“) ... 48
D.6.4.3 Bare Metal Hypervisor („Typ 1“) ... 49
D.6.5 Paravirtualisierung ... 50
D.6.6 Kernelbasierte Virtuelle Maschinen ... 50
E Prozesse ... 53
E.1 Zustände ... 53
E.2 Prozessbeschreibungsblock ... 54
E.3 Zustandsdiagramm ... 55
F CPU-Steuerung (CPU-Scheduling) ... 59
F.1 Einleitung und Ausgangssituation ... 59
F.2 Long-Term-Scheduling ... 59
F.3 Short-Term-Scheduling ... 60
F.3.1 Klassifikation von Short-Term-Scheduling ... 61
F.3.2 Ziele von Short-Term-Verfahren ... 61
F.3.2.1 CPU-Auslastung ... 62
F.3.2.2 Durchsatz ... 62
F.3.2.3 Verweilzeit ... 62
F.3.2.4 Antwortzeit ... 62
F.3.2.5 Wartezeit ... 62
F.3.3 Nonpreemptive Algorithmen ... 63
F.3.3.1 First Come First Served (FCFS) ... 63
F.3.3.2 Shortest Job First (SJF) ... 63
F.3.3.3 Priority Scheduling (HPF) ... 64
F.3.3.4 Priority Scheduling mit Aging ... 65
F.3.4 Preemptive Algorithmen ... 65
F.3.4.1 Preemptive Priority Scheduling ... 66
F.3.4.2 Round Robin (RR) ... 67
F.3.4.3 Mehrere Warteschlangen mit Prioritäten ... 68
F.3.4.4 Fair Share Scheduling ... 68
G Parallelität ... 71
G.1 Prozesse allgemein ... 71
G.2 Gründe für parallele Prozesse ... 74
G.2.1 Information Sharing ... 74
G.2.2 Beschleunigung (Computation Speedup) ... 74
G.2.3 Modularität ... 74
G.2.4 Zweckmäßigkeit (Annehmlichkeit, Convenience) ... 75
G.3 Threads und Context Switching ... 75
G.3.1 Threads im Single User Multitasking Betriebssystem SYMBIAN ... 76
G.3.2 Threads bei der Java Virtual Machine ... 76
G.3.3 Thread Prioritäten in Windows ... 78
G.4 Klassische Problemstellungen ... 80
G.4.1 Grundgedanke ... 80
G.4.2 Producer-Consumer-Problem ... 80
G.4.2.1 Problemstellung ... 80
G.4.2.2 Busy Waiting ... 81
G.4.2.3 Race Condition ... 83
G.4.3 Wechselseitiger Ausschluss und Kritische Regionen ... 83
G.4.3.1 Anforderungen an eine zulässige Lösung ... 84
G.4.3.2 Falsche Lösungsversuche ... 84
G.4.4 Hardwarehilfen ... 87
G.4.4.1 Einprozessorsysteme ... 87
G.4.4.2 Mehrprozessorsysteme ... 87
G.4.5 TestAndSet als Methode einer Java Klasse ... 89
Leseprobe

G.4.6 Spin Lock 89
G.4.7 Semaphore 91
G.4.7.1 Motivation 91
G.4.7.2 Beschreibung der Semaphor-Operationen 91
G.4.7.3 Beispiel Code 92
G.4.7.4 Anwendungen von Semaphoren 95
G.4.8 Kritische Region in Java 97
G.4.9 Monitore 97
G.4.9.1 Allgemeines 97
G.4.9.2 Monitore in Java 97
H Systemverklemmungen (Deadlocks) 99
H.1 Betriebsmittelverwaltung 99
H.2 Synchronisation von Prozessen 99
H.2.1 Petrinetze 99
H.3 Deadlocks 101
H.3.1 Ausgangssituation 101
H.3.2 Modellierung mittels Petrinetz 103
H.3.3 Bewältigung von Deadlocks 104
H.3.3.1 Allgemeine Konzepte 104
H.3.3.2 Deadlock Vorbeugung 106
H.3.3.3 Deadlock Vermeiden 107
H.3.3.4 Erkennen und Wiederherstellen 109
I Speicherverwaltung 111
I.1 Allgemeine Problemstellung 111
I.2 Adressbindung 111
I.2.1 Absolute Adressen 112
I.2.2 Relative Adressen 113
I.3 (Programm-)Lader 114
I.3.1 Lader und Adressbindung 115
I.4 Linker 115
I.4.1 Aufgabe 115
I.4.2 Statisches und dynamisches Linken 116
I.4.2.1 Prinzip 116
I.4.2.2 Nachteil von statischem Linken 116
I.4.2.3 Vorteil von dynamischem Linken 117
I.5 Logische und physische Adressen 118
I.6 Methoden für die Speicherverwaltung 119
I.6.1 Kontinuierlicher Speicherbereich 119
I.6.2 Dynamische Speicherverwaltung 121
I.6.3 Speicherfragmentierung 122
I.6.3.1 Interne Fragmentierung 122
I.6.3.2 Externe Fragmentierung 122
I.6.3.3 Maßnahmen gegen die externe Fragmentierung 123
I.6.4 Seitenverwaltung 124
I.6.4.1 Virtueller Adressraum 124
I.6.4.2 Zuordnung über Seitentabelle 124
I.6.5 Eigener Adressraum für jeden Prozess 126
I.6.6 Translation Look-Aside Buffer 127
I.6.7 Seitenverwaltung und Schutzmaßnahmen 128
I.6.7.1 Read-Only-Bit 128
I.6.7.2 Valid-Bit 128
I.6.7.3 Gemeinsam benutzte Seiten (Page Sharing) 128
J Virtueller Speicher 131
J.1 Zielsetzung 131
J.2 Paging zur Verwaltung von Virtuellem Speicher 133
J.2.1 Grundlagen 133
J.2.2 Fallstudie Windows 133
Leseprobe

J.3 Strategien für Paging Systeme ... 135
J.3.1 Platzierungsstrategie (placement strategy) ... 135
J.3.2 Ladestrategie (fetch policy) ... 135
J.3.3 Ersetzungsregel (replacement strategy) ... 136
J.3.4 Aufwand bei einem Seitenfehler ... 137
J.3.5 Wichtige Ersetzungsstrategien ... 138
J.3.5.1 LRU ... 138
J.3.5.2 NUR (Clock Algorithm) ... 139
J.4 Lokalität und Arbeitsmenge ... 141
K Files, Filesysteme ... 147
K.1 Aufgaben und Schichtenmodell ... 147
K.2 Operationen auf Files ... 148
K.3 File Typen – Metadaten ... 149
K.3.1 Filetyp auf Basis des Namens ... 149
K.3.2 Filetyp auf Basis des erzeugenden Programms ... 150
K.4 Verzeichnisse ... 150
K.5 File-Allokation auf Massenspeichern ... 151
K.5.1 Grundlegendes ... 151
K.5.2 Datenstrukturen ... 153
K.5.2.1 Fragmentierung ... 153
K.5.2.2 Files als verkettete Listen ... 153
K.5.2.3 Clusterzuordnungstabelle (FAT) ... 154
K.5.2.4 Indexknoten ... 155
K.5.2.5 Data-Runs ... 155
K.5.2.6 Inodes ... 156
K.5.2.7 Verwaltung über eine Tabelle ... 157
K.6 Disk Quota ... 160
K.7 Mehrere Filesysteme gleichzeitig ... 160
K.8 RAID ... 161
K.8.1 Stufen von RAID ... 161
K.8.2 RAID_0: Striping ... 161
K.8.3 RAID_1: Spiegelung ... 162
K.8.4 RAID_5: Stripes mit Prüfbits ... 162
L Sicherheit in Betriebssystemen ... 163
L.1 Zum Begriff Sicherheit ... 163
L.2 Identifizieren und Authentifizieren ... 165
L.2.1 Authentizität und Authentifizierung ... 165
L.2.2 Identifizieren ... 166
L.2.3 Zugangskontrollen ... 167
L.2.4 Betriebssysteminterne Identitäten ... 168
L.3 Speichern von Passwörtern ... 169
L.3.1 Passwortsicherheit ... 169
L.3.2 Kryptographische Hashfunktionen ... 170
L.3.3 Salzen ... 173
L.4 Schutz von Objekten ... 174
L.4.1 Aufgabenstellung ... 174
L.4.2 Modelle ... 175
L.4.3 Implementierung der Zugriffsmatrix ... 177
L.4.3.1 Zugriffskontrolllisten (ACL) ... 177
L.4.3.2 Fähigkeiten (Capabilities) ... 178
L.4.4 Vergabe von Berechtigungen in Windows ... 178
L.4.4.1 Beschreibung des Szenarios ... 178
L.4.4.2 Sicht von Windows ... 179
L.4.4.3 Erzeugen von neuen Objekten ... 181
L.4.4.4 Erzeugen von neuen Subjekten ... 183
L.4.4.5 Gruppe für viele/mehrere Subjekte ... 184
L.4.4.6 Erweiterung auf mehrere Gruppen ... 186
Leseprobe

L.4.4.7 Überprüfung der Berechtigungen durch das Betriebsystem 188
L.4.5 Rollenbasierende Zugriffsrechte 188
L.4.5.1 Basis RBAC (Core RBAC) 189
L.4.5.2 Hierarchisches RBAC (Hierarchical RBAC) 192
L.4.5.3 Beschränktes RBAC (Constrained RBAC) – Statische und Dynamische Trennung von Zuständigkeiten 193
M Netzwerkkommunikation 195
M.1 Netzwerkkommunikation ist Aufgabe des Betriebssystems 195
M.2 TCP/IP ist die dominante Protokoll-Familie 195
M.3 Begriff: Protokoll 195
M.4 Zum Namen TCP/IP 196
M.4.1 Grundfunktionalität von IP 196
M.4.2 Grundfunktionalität von TCP 197
M.5 Schichtenmodell 197
M.6 IP Version 4 (IPv4) 198
M.6.1 Adressierung 198
M.6.1.1 IP-Adressen 198
M.6.1.2 IP-Adressen und Subnetting 198
M.6.1.3 Verfeinerung der Adressierung von Host → Prozess 202
M.6.2 IP Client-Konfiguration 203
M.6.2.1 Wesentliche Konfigurationsinformationen 203
M.6.2.2 Typen der Konfiguration eines IP-Hosts 204
M.6.3 Testen einer IP-Konfiguration 205
M.6.3.1 Auslesen der Konfiguration (ifconfig / ipconfig / GUI) 205
M.6.3.2 Testen der Kommunikation (ping, arp) 205
M.6.3.3 Testen der Namensauflösung (nslookup) 206
M.6.3.4 Feststellen einer Route (traceroute) 207
M.7 IP Version 6 (IPv6) 208
M.7.1 Adressierung 209
M.7.1.1 IPv6-Adressen 209
M.7.1.2 IPv6-Adressen und Subnetting 209
M.7.1.3 Verfeinerung der Adressierung von IPv6-Host → Prozess 209
M.7.2 IPv6 Client-Konfiguration 210
M.7.2.1 Wesentliche Konfigurationsinformationen 210
M.7.2.2 Typen der Konfiguration eines IPv6-Hosts 210
M.7.3 Testen einer IPv6-Konfiguration 211
M.7.4 Auswahl zwischen IPv4 und IPv6 212
M.7.5 IPv6-Adressvergabe 213
Abbildungsverzeichnis 215
Verzeichnis der Pseudocode Beispiele 219
Tabellenverzeichnis 220
Literatur 221
Index 224
Leseprobe

Großrechnersysteme als Mehr-Benutzersysteme gehen davon aus, dass sie viele Anwender gleichzeitig bedienen müssen. Dabei bleibt offen, ob diese Anwender mit einem einfachen Terminal an das System gekoppelt sind oder ob sie auf ihrer Workstation einen zusätzlichen Prozess als Terminal-Emulation gestartet haben oder das Terminal dort simuliert wird. Entscheidend ist, dass die Applikation auf dem zentralen System abläuft und das Terminal (mit zugehöriger Peripherie wie z.B. Drucker) nur dem Prozessstart und der E/A von Daten dient.

Ein weiterer Gesichtspunkt ist die Gestaltung der Schnittstelle zum Benutzer. Sie kann befehlszeilenorientiert sein oder eine komfortable graphische Oberfläche bieten. Letztere ist bei allen Workstations vorgesehen. Darüber wurde in der Einleitung schon gesprochen.

D.6 Virtuelle Maschinen

D.6.1 Grundprinzip

Zunächst klären wir kurz, was man unter einer virtuellen Maschine versteht und erläutern dann, wie unter Anlehnung an das Schichtenmodell eine spezielle Architektur entsteht.

Aus der Sicht eines Anwenders ist eine virtuelle Maschine ein Computer, der (teilweise) durch ein Programm simuliert wird.

Das Konzept kommt aus dem Bereich Softwareengineering und wir skizzieren es kurz an einem Beispiel:

```
public class Keyboard {
    ...
    public boolean keyPressed(){
        ...
    }
    public char readKey() {
        ...
    }
    ...
}
```

Code D-1 *Schnittstellen virtuelle Tastatur*

Leseprobe

Stellt man einer Java-Programmiererin diese Klasse zur Verfügung, so kann sie die Schnittstelle (hier exemplarisch: die Methoden `keyPressed()` und `readKey()`) verwenden. Die angesprochene Tastatur ist *virtuell*, das entstehende Programm wird im Prinzip unabhängig von der Hardware. Bei Bedarf sind bloß die Methoden an eine geänderte Tastatur anzupassen.

Ähnlich kann man auch bei Betriebssystemen vorgehen. Das Betriebssystem verbirgt die darunter liegende Hardware und bietet den Anwenderprogrammen an seiner Schnittstelle zu diesen eine Sammlung von Prozeduraufrufen. Diese heißt *API (Application Programming Interface)*. So betrachtet ist das Betriebssystem eine Virtuelle Maschine, deren „Befehlsvorrat" durch sein API definiert ist.

Zur Umsetzung dieses Konzeptes bedient man sich insbesondere folgender Möglichkeiten und Kombinationen davon: Systemaufrufe über Unterprogramme und Systemaufrufe über Software-Interrupts.

Aus Sicherheitsgründen wird bei einem Betriebssystem zwischen zwei Betriebsarten unterschieden: *User-Mode* und *Kernel-Mode* (auch: *System-Mode*). Prozesse, die im Kernel-Mode ablaufen, haben höhere Rechte und Privilegien, die eigentlich nur wenigen Betriebssystemfunktionen vorbehalten sein sollten. (Siehe dazu auch die Ausführungen beim Client Server Modell Punkt D.3.3.)

Daher versucht man, selbst bei Systemaufrufen möglichst große Teile des Codes im User-Mode ablaufen zu lassen und in den Kernel-Mode nur dann umzuschalten, wenn dies unbedingt notwendig ist. Gleichzeitig soll dies dem Programmierer verborgen bleiben.

In der Windows Familie setzt man diese Idee dadurch um, dass zwischen API und den „eigentlichen" Systemaufrufen unterschieden wird, wobei auf Programmiersprachenebene die involvierten Systemaufrufe nicht offen gelegt werden, das API also eine reine Prozedur- bzw. Klassenbibliothek ist.

Durch das standardisierte API Interface können Modifikationen des Codes von Systemaufrufen – man denke an Security Patches – jederzeit durchgeführt werden, ohne dass die im Anwenderprogramm verwendeten API-Calls betroffen werden, solange die Schnittstelle (Parameter usw.) zwischen beiden Ebenen nicht verändert wird.

Die Abbildung zeigt die Situation, dass auf einem Host-Betriebssystem als Basis nicht nur die Anwendungen $App0_1$- $App0_x$ laufen, sondern speziell zwei weitere Gast-Betriebssysteme (Betriebssystem 1, Betriebssystem 2) aufgesetzt sind. Auf diesen wieder laufen die Anwendungen $App1_i$ i=1, 2,…, y und $App2_j$ j=1, 2, …, z.

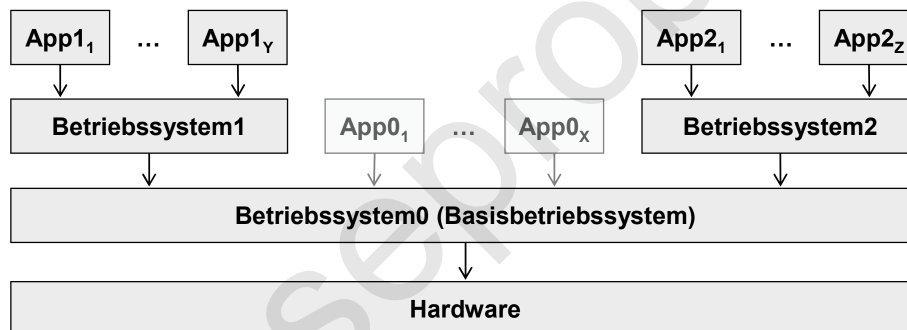


Abb. D-8 Virtualisierung für mehrere Betriebssysteme auf einer Hardware

Leseprobe

Der erste wirklich kommerzielle Einsatz des Architekturansatzes auf Basis virtueller Maschinen findet sich bei IBM bei dem System VM/370 Mitte der 1970er Jahre. Damit wurde es möglich, auf der damals neuen 370er Architektur nicht nur das weit verbreitete Betriebssystem OS/360 einzusetzen und damit alle bisherigen Anwender sozusagen bei der Stange zu halten, sondern auch ein Timesharing Konzept einzuführen, das als eigenes System parallel zum OS/360 auf dem VM/370 lief.

Die Idee der virtuellen Maschine finden wir heute z.B. unter Windows, wo in der „DOS-Box" das Betriebssystem DOS der 16-Bit Generation *emuliert* wird. Man beachte: bei einer softwaremäßigen Emulation eines Systems wird dieses im Detail funktionsgleich abgebildet. Bei einer Simulation wird nur das nach Außen wirksame oder das für den Betrachter relevante Verhalten nachvollzogen, die inneren Abläufe können aber durchaus völlig anders als beim Original umgesetzt sein. Ein Flugsimulator fliegt nicht wirklich, nur die darin sitzende Person hat diesen Eindruck.

Für das Apple-Betriebssystem OS X.y gibt es eine virtuelle Maschine, die dem Windows XP entspricht.

Ein weit verbreitetes Produkt ist VMware, das z.B. auf Windows oder Linux aufgesetzt werden kann und dann seinerseits verschiedene andere Systeme als Gast-Systeme unterstützt. (Ein besonderes Beispiel dazu ist das Betriebssystem MINIX3 von A. S. Tanenbaum, welches auf allen Rechnern installiert werden kann, auf denen VMware läuft.) Eine effiziente Variante von VMware ist

VMware ESX, das eine *Bare-Metal (Hypervisor)* Architektur verwendet. Bei diesem Produkt setzt die Virtualisierungsschicht nicht wie in der Standardversion als Applikation auf einem bereits vorinstallierten Host-Betriebssystem auf, sondern in Form eines eigenen Kernels direkt auf der Gerätehardware. Durch diese schlanke Architektur und der Möglichkeit des direkten Hardware Zugriffes ist VMware ESX derzeit die bevorzugte Virtualisierungsvariante für den Realbetrieb etwa in Rechenzentren oder Serverfarmen.

Microsoft wieder stellt mit VirtualPC eine virtuelle Maschine zur Verfügung, die einen Standard-PC (mit typischer Hardwareausstattung) emuliert, sodass auf diesem z.B. wieder Windows als Gast-Betriebssystem eingesetzt werden kann. Ebenso wie VMware bietet auch Microsoft ein eigenes Virtualisierungsprodukt für den professionellen Bereich namens *Hyper-V.*

Ergänzende Details werden in den Abschnitten D.6.4 bis D.6.6 besprochen.

D.6.2 Systemaufrufe (system calls) über Unterprogramme (Prozeduren)

Das API als Sammlung von Prozeduren und damit als Bibliothek von Systemaufrufen zu realisieren, folgt einem konventionellen aber durchaus zweckmäßigen Ansatz. Damit braucht ein Programmierer auch bei systemnahen Aufgaben die gewohnte Entwicklungsumgebung nicht verlassen. In einer objektorientierten Umgebung ist das API als spezielle Klassenbibliothek ausgelegt. Das folgende Beispiel liest mit Hilfe der API-Funktion `void GetSystemInfo(LPSYSTEM_INFO lpSysteminfo)` von Windows verschiedene hardwarebezogene Systemdaten aus, darunter die Anzahl der vorhandenen Prozessoren, und weist sie einer Variablen mit vordefinierter Datenstruktur `SYSTEM_INFO` zu, auf die mit der Adresse `&siSysInfo` verwiesen wird:

```
#include <windows.h>
#include <stdio.h>
void main()
{
    SYSTEM_INFO siSysInfo;
    GetSystemInfo(&siSysInfo);
    printf("Number of processors: %u\n", siSysInfo.dwNumberOfProcessors);
}
```

Code D-2 *Auslesen hardwarebezogener Systemdaten in Windows*

D.6.3 Systemaufrufe mittels Software-Interrupts (SWI)

Dieses Konzept entspricht einem indirekten Prozeduraufruf, denn anstatt die benötigte Betriebssystemfunktion so wie oben direkt aufzurufen, wird durch SWI(i) die i-te Position in einer zugehörigen *Interruptvektor-Tabelle* angesteuert, in der die Startadresse des gewünschten Systemaufrufes steht. Dieser Ansatz wird vor allem bei kleineren Betriebssystemen gewählt.

So wird z.B. im früheren PC-Betriebssystem DOS über SWI(33) (d.h.: SWI(21H)) im Wesentlichen die gesamte E/A erledigt, wobei Parameter in Prozessorregistern abzulegen sind. Das DOS selbst wieder bedient sich des darunter liegenden BIOS ebenfalls über Software-Interrupts. Beispielsweise sorgt SWI(5) für einen Ausdruck des Bildschirminhaltes auf dem angeschlossenen Drucker.

Am Rande sei vermerkt, dass ein Software-Interrupt, der zugleich einen Wechsel vom User-Mode zum Kernel-Mode erzeugt, oft auch als *TRAP* bezeichnet wird.

Leseprobe

D.6.4 Hypervisor

D.6.4.1 Aufgabenstellung und Anwendung

Für Anwenderinnen und Anwender ist es sehr zweckmäßig, wenn auf ihrem Arbeitsplatzrechner bei Bedarf neben dem oder innerhalb des aktuell verwendeten Betriebssystems ein weiteres Betriebssystem gestartet werden kann, mit dem man früher schon gearbeitet hat. Damit wird man unter Umständen von lästigen Problemen einer mangelhaften Rückwärtskompatibilität befreit. Durchaus können auch Lizenzierungsfragen eine gewichtige Rolle spielen.

Eines von mehreren möglichen Beispielen für dieses Szenario ist der XP-Mode unter Windows-7, wenn man einerseits die Vorgabe hat, ein neueres Betriebssystem zu installieren, anderseits, z.B. aus einer Flottenpolitik vorgesehen, bei ein und demselben Hersteller verbleiben will oder muss.

Der Hauptgrund für die steigende Verwendung von Virtualisierungskonzepten liegt aber bei Einsparungsmöglichkeiten bei Servern, insbesondere, wenn diese typischerweise im Normalbetrieb wenig ausgelastet sind.

In diesem Zusammenhang, der häufigsten Realität entsprechend, gehen wir von einem Mehrprozessorsystem aus, das über eine symmetrische Multiprozessorarchitektur (SMP) verfügt. Hier sind alle CPUs im Hostsystem (Host-Hardware) ident und vor allem gleichberechtigt bei der Ressourcenzuteilung, insbesondere bei der Zuteilung von Prozessen.

Bei Servern in einer Firma ist das Ziel folgendes: Anstatt mehrere Server für unterschiedlichste Aufgaben zu installieren und zu betreiben, werden diese Aufgaben an virtuelle Maschinen übertragen und diese parallel auf einem (z.B.) SMP Server ausgeführt. Damit kann man die Gesamtauslastung optimieren und eine oder mehrere CPUs bzw.- CPU-Leistung von momentan nicht besonders beschäftigten nunmehr virtuellen Servern an solche Server zuteilen, die temporär einer besonders hohen Auslastung unterliegen.

Studien haben ergeben, dass damit auch Energiekosten, Wartungskosten und vor allem auch die Kosten für den Betrieb einer räumlichen Infrastruktur eingespart werden. Bei Virtualisierung steht damit insbesondere die Servervirtualisierung durch das hohe Einsparungspotential im Vordergrund.

Das in Abb. D-8 gezeigte Schema lässt offen, ob das *Basisbetriebssystem (Basis-Hostbetriebssystem* auch kurz: *Host System*) ein beliebiges Betriebssystem sein kann und/oder ob spezielle Anforderungen an dieses gestellt werden müssen. Ebenso erhebt sich die Frage, ob die Gastbetriebssysteme mit jenem des Hostsystems mehr oder weniger „verträglich" sind (z.B. bei ihren API- Schnittstellen) oder völlig andere Betriebssysteme mit durchaus unterschiedlicher Architektur zum Einsatz kommen.

Ebenso kann die Variante überlegt werden, dass das Host-System und die (darüber?) laufenden Gastbetriebssysteme zwar unterschiedlich (konfiguriert) sind, sich aber einen gemeinsamen Kernel teilen.

Der entscheidende Punkt für alle Lösungsansätze ist, dass Gastbetriebssysteme letztlich auf die nur ein Mal verfügbare gemeinsame Hardware (shared hardware) zugreifen müssen.

Ein besonderes Problem dabei ist folgendes: Ein Gastbetriebssystem führt aus seiner Sicht auch Kernel-Mode Prozesse aus, die privilegierte CPU-Instruktionen verwenden, kann also aus Sicht des Basis-Hostbetriebssystems nicht so ohne weiteres (vollständig) im User-Mode ablaufen.

Somit stellt sich die Frage, wie solche Instruktionen eines Gastbetriebssystems ermöglicht werden können. Dazu ist eine zusätzliche Software-, allenfalls Hardwareunterstützung erforderlich, die solche Ressourcen-Zugriffe verwaltet und auch überwacht. Diese Software wird als *Virtual Machine Monitor (VMM)* oder *Hypervisor* bezeichnet.

Leseprobe

Der entscheidende Unterschied gängiger Implementierungen eines solchen Hypervisors (oder VMM) besteht darin, auf welcher Ebene/Schicht dieser eingebettet wird:

- oberhalb des Host-Betriebssystems
  (Hosted Virtualisation, Hypervsior Typ 2)

  oder

- direkt auf der gemeinsamen Hardware
  (Bare Metal Virtualisation, Hypervsior Typ 1).

Zugleich muss man sich offen lassen, ob ein Gastbetriebssystem letztlich für dieselbe Hardware (insbesondere: CPU) ausgelegt ist und damit ausschließlich Instruktionen ausführt, die auch auf dem Host-System ausgeführt werden können oder, ob das Gastsystem für einen völlig anderen CPU-Typ und damit unterschiedlichen Instruktionsvorrat ausgelegt ist.

Daher sprechen wir im Folgenden auch oft von einer *virtuellen Maschine (VM)* anstatt – eingeschränkt – von einem Gastbetriebssystem.

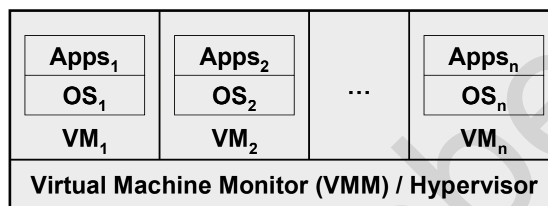


Abb. D-9 *Hypervisor – Allgemeine Architektur*

Vergleicht man die Abb. D-9 mit der früheren Darstellung in Abb. D-8, so kann ein Hypervisor auch als ganz spezielles „Betriebssystem“ verstanden werden, dessen Aufgabe es ist, den parallelen Betrieb von virtuellen Maschinen VM_i zu steuern und zu überwachen.

D.6.4.2 Hosted Hypervisor („Typ 2“)

Hier wird der Hypervisor (VMM) direkt auf das Host-Betriebssystem (Basis Betriebssystem) aufgesetzt. Dieser Ansatz ist nahe liegend und konform zu Abb. D-8 und ist in Abbildung Abb. D-10 dargestellt.

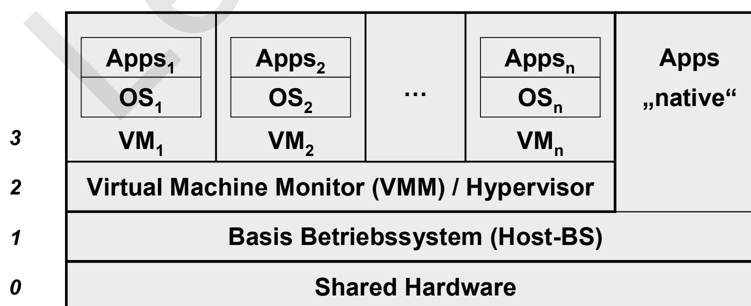


Abb. D-10 *Hosted Hypervisor („Typ 2“)*

Vertreter des Konzeptes „Hosted Hypervisor" sind VMware Server, Microsoft Virtual Server und diverse Workstation-Lösungen wie VMware Workstation, Microsoft Virtual PC. Ebenso ist Oracle Virtual Box dieser Klasse zuzurechnen.

Wir halten fest: Bei der Hosted Hypervisor Methode läuft der Hypervisor auf einem vorinstallierten Basisbetriebssystem und damit sind die VM_i auf der dritten Ebene aufgesetzt (Abb. D-10).

Zum Zugriff auf die Hardware wird das Basisbetriebssystem verwendet. Damit müssen insbesondere Instruktionen einer VM_i im Kernel-Mode vom Hypervisor durch einen Trap (siehe D.6.3) aufge-

Leseprobe

fangen, emuliert und dann an das darunter liegende Basisbetriebssystem zur Bearbeitung weitergereicht werden. Gleiches gilt für I/O-Anforderungen. Periphere Geräte und Anschlüsse, für die über den Hypervisor keine Emulationssoftware verfügbar gemacht wird, können dann nicht gemeinsam angesprochen werden.

Ein besonderer Nachteil der Hosted Methode ist auch, dass sie für Real Time VM_i nicht geeignet ist, denn dem Basis Betriebssystem obliegt letztlich das gesamt CPU-Scheduling und ebenso das Scheduling für den allgemeinen Ressourcenzugriff auf der gemeinsamen Hardware: In der Praxis ist das Host-Betriebssystem nämlich nicht für Real Time ausgelegt.

Ein Vorteil besteht in einer einfachen Installation ohne besondere Modifikation des Gastbetriebssystems bzw. der VM_i, sofern sie vom Hypervisor unterstützt werden.

Außerdem kann innerhalb einer Herstellerfamilie der Hypervisor naturgemäß auf das bekannte Gastbetriebssystem maßgeschneidert und optimiert werden.

D.6.4.3 Bare Metal Hypervisor („Typ 1")

In diesem Fall sitzt der Hypervisor (Virtual Machine Monitor) direkt auf der Hardware und setzt daher kein vorinstalliertes Betriebssystem voraus (Abb. D-11).

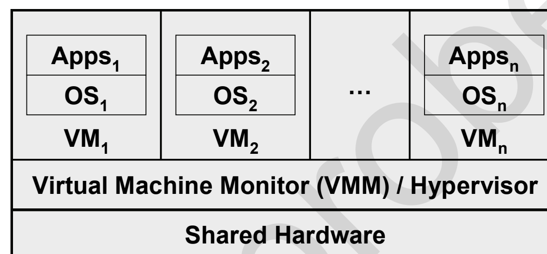


Abb. D-11 *Bare Metal Hypervisor („Typ 1")*

Leseprobe

Demnach erinnert diese Situation an die eingangs getroffene Bemerkung, den Hypervisor als spezialisiertes Betriebssystem sehen zu können, dessen primäre Aufgabe darin besteht, virtuelle Maschinen VM_i, i = 1,2,…,n, auf einer gemeinsamen Hardware auszuführen.

Vertreter dieser Architektur sind z/VM von IBM, VMware ESX-Server, Xen Hypervisor und Microsoft Hyper-V.

Ein besonderer Vorteil betreffend Effizienz besteht bei der Bare Metal Lösung darin, dass hier bei I/O Anforderungen der Hypervisor ohne den Umweg über ein Basis Betriebssystem kommunizieren kann. Allerdings werden dazu zwei besondere Schnittstellen (sie heißen hier auch: *Treiber*) benötigt.

- Zunächst ist eine explizite Schnittstelle nach unten zur gegebenen Hardware erforderlich, da dieser Zugriff nicht mehr über das („fehlende") Basisbetriebssystem bereitgestellt wird.
- Nach oben hin, zu den virtuellen Maschinen VM_i ist analog zu Hosted Version ebenso Emulations-Software für die I/O Anforderungen (I/O Requests) erforderlich. Gleiches gilt für privilegierte Instruktionen, wenn eines der VM_i ein Gastbetriebssystem ist, da eine Kernel-Mode Instruktion einer Gast VM_i aus der Sicht des Host-Systems für dieses einerseits keine privilegierte Instruktion ist, anderseits aber einer speziellen Prüfung und Behandlung bedarf.

Da ein Bare Metal Hypervisor speziell für gegebene Hardware entwickelt oder angepasst ist, kann er bei neueren Prozessoren von Architekturerweiterungen, z.B.: von *Intel-VT („Vanderpool Technology")* oder *AMD-V („Pacifica")* unmittelbar und ohne Umwege Gebrauch machen. Dies ist ein attraktiver Vorteil des Verfahrens. Daher wird ihm besondere Zukunftsträchtigkeit zugemessen, da

man davon ausgehen kann, dass diese Architekturerweiterungen auf allen gängigen Prozessortypen zu finden sind.

Eine andere Möglichkeit für den I/O-Zugriff aus einer VM_i besteht darin, dass die angesprochenen Einheiten (Devices) vom Hypervisor aufgeteilt werden. Der Hypervisor weist ein Device einer einzelnen VM exklusiv zu. Dieser VM wird damit ein „direkter" Zugriff auf das Betriebsmittel gestattet.

Ein weiterer Vorteil der Bare Metal Version liegt auch darin, dass als VM auch ein Real Time Betriebssystem möglich ist, da der Hypervisor auf das dort notwendige spezielle CPU-Scheduling (z.B.) durch Prioritätensetzung eingehen kann.

Die erforderliche Vielfalt an Treibern bei einer Bare Metal Hypervisor Version ist natürlich ein gewisser Nachteil, da der Hypervisor mit jedem Treiber umgehen können muss. Daher unterstützen solche Systeme nur ein klar umgrenztes (zertifiziertes) Set von Hardware.

D.6.5 Paravirtualisierung

Die vorhin angesprochene Virtualisierungsmethoden fallen unter die Rubrik *„vollständige Virtualisierung"* . Es gibt noch eine spezielle Variante davon, die als *Paravirtualisierung* bezeichnet wird.

Die Idee dazu basiert auf Performance-Überlegungen, denn insbesondere privilegierte Instruktionen einer virtuellen Maschine müssen, wie schon erwähnt, vom Hypervisor (besten Falls) mit Hardwareunterstützung abgefangen, emuliert und das Ergebnis an das Host-Betriebssystem weitergereicht werden.

Um diese Umwege zu vermeiden oder zu reduzieren, kann man, sofern der Code des Betriebssystems oder Code einer VM_i offen gelegt ist bzw. weil dieser dem jeweiligen Hersteller bekannt ist, das Gastbetriebssystem (die VM_i) entsprechend verändern. Dann sind seitens einer derart modifizierten VM_i „nur" verhältnismäßig einfache Aufrufe von API-Funktionen des Hypervisors erforderlich, was zu einem Performance-Gewinn führt.

Aus der Gliederungssicht für Hypervisor-Varianten trifft man bei Paravirtualisierung typischerweise eine Bare Metal („Typ 1") Lösung an.

Vertreter dieser Variante sind PowerHypervisor, Xen, Microsoft Hyper-V, VMware ESX-Server.

Der evidente Nachteil der Paravirtualisierung ist, dass die Gast-VM_i (bzw. die Gastbetriebssysteme) zu modifizieren sind und damit wegen der notwendigen Kenntnis über den Source Code oder durch die Übernahme von vorgefertigten Varianten Herstellerabhängigkeiten in Kauf genommen werden müssen.

D.6.6 Kernelbasierte Virtuelle Maschinen

Am Anfang des Abschnitts D.6.4 ist in einem Nebensatz die Variante angedacht, dass ein Hostsystem und die darüber laufenden Gastbetriebssysteme sich einen gemeinsamen Kernel teilen können. Die von der Linux- Gemeinde entwickelte *„Kernel-Based Virtual Maschine" (KVM)* geht in etwa in diese Richtung.

Sie setzt dabei eine CPU-Unterstützung durch Intel-VT oder AMD-V voraus.

Die KVM (entwickelt von der Firma Qumranet) wird in den Kernel des Linux-Betriebssystems integriert und wirkt dann zusammen mit diesem Kernel als Hypervisor.

Insofern ist eine Klassifikation nach dem „Typ 2" (Hosted Hypervisor) gerechtfertigt, da der Linux-Kernel der Host ist und zentrale Aufgaben wie CPU-Steuerung oder Speicherverwaltung übernimmt. Anderseits liegt die KVM als Erweiterung des Linux-Kernel direkt auf der Hardware, sodass auch eine Klassifikation als „Typ 1" argumentierbar wäre.

Leseprobe

Die KVM stellt hinsichtlich der Virtualisierung für die Gastsysteme selbst keine virtuelle Hardware zur Verfügung, sondern setzt eine Emulationssoftware namens QEMU voraus. Die Bereitstellung von virtualisierten Geräten wird von QEMU für den jeweiligen Gast übernommen.

Schematisch erhält man einen Aufbau, der in Abbildung Abb. D-12 gezeigt ist.

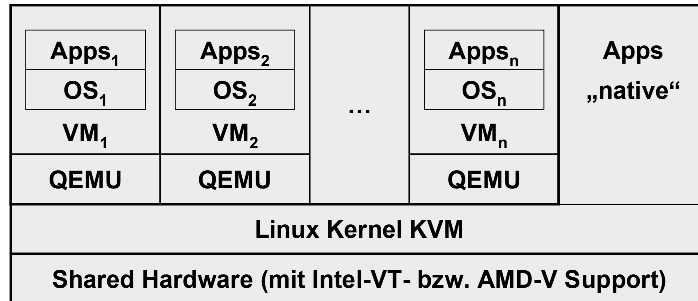


Abb. D-12 Kernelbasierte Virtuelle Maschine (KVM)

Leseprobe

G.4.7 Semaphore

G.4.7.1 Motivation

Wir erweitern das Spin Lock Konzept, indem eine Warteschlange damit assoziiert wird und kommen so zu *Semaphoren*. Dabei sollen Prozesse in diese Warteschlange Q gestellt werden, sobald sie warten müssen. Ist dies der Fall, wechselt ein Prozess vom Zustand *running* auf *waiting*. Beim Herausnehmen aus der Warteschlange Q kommt der Prozess vom Zustand *waiting* in den Zustand *ready* bzw. *running*.

Mit diesem Ansatz erreicht man, dass wegen des wechselseitigen Ausschlusses anzuhaltende Prozesse während des Wartens in der Warteschlange verweilen und so keine Rechenzeit benötigen. Damit muss zur Realisierung einer Semaphore das Betriebssystem mit seiner Scheduling Strategie involviert werden.

Ob nach dem Entnehmen eines Prozesses oder Threads P aus der Warteschlange Q sofort *running* wird oder bloß - wie es i.Allg. der Fall ist - in den Zustand *ready* für späteres Scheduling versetzt wird, ist Implementierungssache und im Semaphorkonzept nicht weiter spezifiziert. Wichtig ist auch für den Fall, dass mehrere Prozesse in Q warten, Starvation durch eine *faire* Entnahmestrategie verhindert wird.

Die Bezeichnung Semaphore kommt von den griechischen Wörtern *sema = Zeichen* und *pherein = tragen*. Semaphoren werden, dem Namen entsprechend, für den Austausch von Signalen zwischen Prozessen verwendet. Von E. W. Dijkstra stammen die Bezeichner p (proberen = versuchen) und v (verhogen = erhöhen) in seiner Abhandlung über parallele Prozesse und dort eingeführte Semaphore [Dij85, Nachdruck von 1965].

G.4.7.2 Beschreibung der Semaphor-Operationen

Eine Semaphore S besteht im Wesentlichen aus einer Integer Variable mit einer Initialisierung `init(S,n)` und zwei *atomaren* Operationen:

- `wait`:
  p-Operation
- `signal`:
  v-Operation.

Weiters ist einer Semaphore eine *Warteschlange Q von Prozessen* zugeordnet.

Die Wirkung der Operationen ist wie folgt:

- `init(S,n)`:
  Die Semaphore S wird mit einem Wert `n` initialisiert. Falls S verwendet wird, um eine Critical Region zu implementieren, nimmt man `init(S,1)`. Ein Spezialfall ist eine binäre Semaphore `mutex`, deren Initialisierung man mit `init(mutex,true)` bzw. `init(mutex,false)` durchführt.

Im Detail geschieht durch diese atomaren Operationen folgendes:

- `wait(S)`:
  Der Prozess P überprüft, ob $S > 0$. Ist dies der Fall, wird unteilbar (atomar) $S = S - 1$ ausgeführt und P fährt ohne Verzögerung fort. Andernfalls wird P in den Zustand waiting versetzt und in die dazu gehörige Warteschlange Q eingefügt. (Das Einhängen in die Warteschlange Q erfolgt durch Einfügen des TCBs oder durch Einhängen eines Verweises (Pointer) auf den TCB des betreffenden Prozesses.)
- `signal(S)`:
  Der Prozess P überprüft mittels `P.signal(S)`, ob gegenwärtig ein oder mehrere Pro-

Leseprobe

zesse in Q gereiht sind. In diesem Fall wird ein Prozess Pi aus Q ausgewählt und in den Zustand ready (bzw. running, s.o.) versetzt. Die Auswahl aus Q muss fair sein, damit Starvation verhindert wird. Falls die Schlange Q leer war, wird atomar S = S + 1 ausgeführt, sodass die Wirkung von `signal` nicht verloren geht.

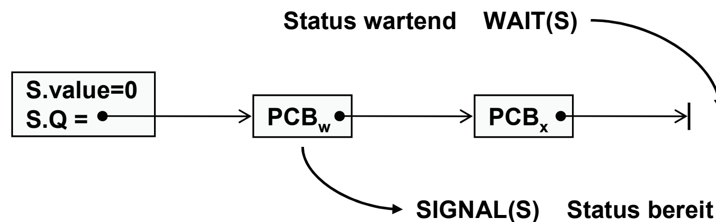


Abb. G-13 *Semaphore mit zwei wartenden Prozessen W und X*

Angenommen, S sei mit `init(S,3)` auf einen Anfangswert gesetzt worden. Dann können mit `P1.wait(S)`, `P2.wait(S)` und `P3.wait(S)` maximal drei Prozesse die Semaphore ohne Verzögerung passieren. Erst ein Prozess P4 wird nach `P4.wait(s)` in die Warteschlange gestellt und verbleibt dort ohne CPU-Verbrauch, bis ein Pi ein `Pi.signal(s)` sendet.

Die erforderliche Unteilbarkeit der Semaphor-Operationen kann wie folgt erreicht werden:

- bei *Nonpreemptive Scheduling:*
  Leicht zu erzielen, da keine Scheduling-Interrupts erlaubt sind, wenn eine Folge von Instruktionen ausgeführt wird.
- bei *Preemptive Scheduling mit nur einer CPU:*
  Hier kann das Interrupt Register maskiert werden, um Interrupts für eine gewisse Zeit zu verhindern.
- im *allgemeinen Fall* (preemptive Scheduling und Mehrprozessorsystem):
  Spin Lock Operationen Lock und Unlock können verwendet werden. Das bedeutet, dass man die Semaphore selbst als kurze Kritische Region (busy waiting!) betrachtet, mit der man lange Kritische Regionen (Prozess wird schlafen gelegt, Zustandswechsel zu waiting) implementieren kann.

Leseprobe

G.4.7.3 Beispiel Code

Alle Operationen auf einer Semaphore sind atomar zu realisieren. Wir betrachten jede Semaphor-Operation für sich als kurze Critical Region und können dazu Spin Locks einsetzen.

Im Folgenden verwenden wir die Notation `lock` und `unlock`, um zu garantieren, dass die dazwischen liegenden Befehle atomar sind:

```
lock;
      // Mit Spin Lock auszuführende Befehle;
unlock;
```

Code G-18 *lock / unlock für atomare Code Abschnitte*

Ebenso verwenden wir einen hier nicht näher definierten Datentyp für eine Warteschlange Q (bzw.: S.Q als die der Semaphore zugeordnete Warteschlange) als Liste von Prozessbeschreibungsblöcken PCB vom Typ `process`. Auf Q seien als Operationen `enqueue` und `dequeue` erklärt.

Die gewählte Notation ist an eine prozedurale Schreibweise angelehnt:

```
enqueue(S.Q, PCB)
...
dequeue(S.Q, PCB);
```

Code G-19 Hinzufügen/Entfernen eines Prozesses in/von der Warteschlange der Semaphore

Dabei repräsentiere PCB den laufenden Prozess P und `dequeue` soll im schon früher erklärten Sinn fair einen Prozess entnehmen, trivialer weise durch FIFO.

Ferner unterstellen wir seitens des Betriebssystems die Verfügbarkeit der API-Calls.

```
block(PCB)
wakeup(PCB)
```

Code G-20 Prozess Zustandsüberführungsfunktionen block und wakeup

Entsprechend dem Task-Zustandsdiagramm wird durch `block(PCB)` bzw. `wakeup(PCB)` ein durch seinen PCB repräsentierter Prozess vom Zustand *running* in den Zustand *waiting* bzw. von *waiting* nach *ready* gebracht.

Wir erinnern daran, dass der Übergang von *waiting* nach *ready* den allgemeinen im System vorgegeben Scheduling Prinzipien folgt, dies aber nicht unbedingt so sein muss. Man könnte den betroffenen Prozess hier sofort in den Zustand *running* versetzen, würde damit aber einen Eingriff in die globale Scheduling Strategie nur um eines „lokalen Vorteiles" wegen provozieren.

Eine Semaphore kann damit wie folgt vereinbart werden:

```
typedef struct {
      int value;              /* Zaehler, der Semaphore zugeordnet */
      struct process *Q;      /* verweist auf die Warteschlange Q */
} semaphore;

VAR S:semaphore;
lock; S.value = n; unlock;   /* init(S, n) */
lock; wait(S); unlock;       /* wait(S)*/
lock; signal(S); unlock;     /* signal(S)*/
```

Code G-21 Beschreibung Semaphore in Pseudocode

Die Befehle (hier: procedure calls) `wait` und `signal` selbst beschreiben wir als an die Programmiersprache C angelehnte Prozeduren vom Typ `void`, da sie keinen Rückgabewert liefern. Ferner sei wieder ein Call by Reference unterstellt. Wäre `semaphore` als Klasse in Java definiert, wäre mit `semaphore s = new semaphore();` das angelegte `s` ohnedies ein Zeiger auf das angelegte Klassenobjekt. Dies ist bei einer Umsetzung des Pseudocodes von Code G-21 zu beachten.

Leseprobe

```
lock;
      void wait(semaphore S) {
            S.value--;
            if (S.value < 0) {
                  enqueue(S.Q, PCB);
                  block(PCB);
            }
      }
unlock;

lock;
      void signal(semaphore S) {
            S.value++;
            if (S.value <= 0) {
                  dequeue(S.Q, PCB);
                  wakeup(PCB 1);
            }
      }
unlock;
```

Code G-22 *Operationen signal und wait auf eine Semaphore*

Leseprobe

Es soll vorab auffallen, dass der Code nicht streng der klassischen Beschreibung von Semaphoren folgt. Man beachte nämlich, dass in dieser Implementierung `S.value` negativ ist, sobald Prozesse auf ein `signal` warten.

Es repräsentiert `S.value` < 0 mit seinem Absolutwert die Anzahl der wartenden Prozesse. Diese Beobachtung kann man nachvollziehen, wenn man etwa `init(S,1)` annimmt, einen Prozess bei wait vorbeilässt und dann überlegt, dass alle folgenden sich um das Passieren der Semaphore bewerbenden Prozesse schlafen gelegt werden müssen, bis mittels `signal` ein Prozess aus der Warteschlange geholt wird.

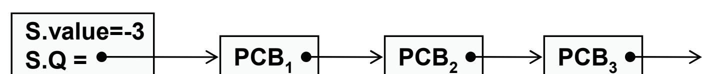


Abb. G-14 *Semaphore zeigt durch S.value = -3 drei wartende Prozesse*

Daher muss bei der vorliegenden Implementierung in jedem Fall bei `wait` bzw. `signal` der Zähler herunter (`S.value--;`) bzw. hinauf (`S.value++;`) gezählt werden.

In der „klassischen“ Definition zählt ein `signal` den Zähler nur dann hinauf, wenn kein Prozess aus der Warteschlange zu entnehmen ist.

G.4.7.4 Anwendungen von Semaphoren

G.4.7.4.1 Prozess-Synchronisation

Man betrachte zwei parallele Prozesse P_1, P_2 und verwende eine Semaphore `sync` wir folgt:

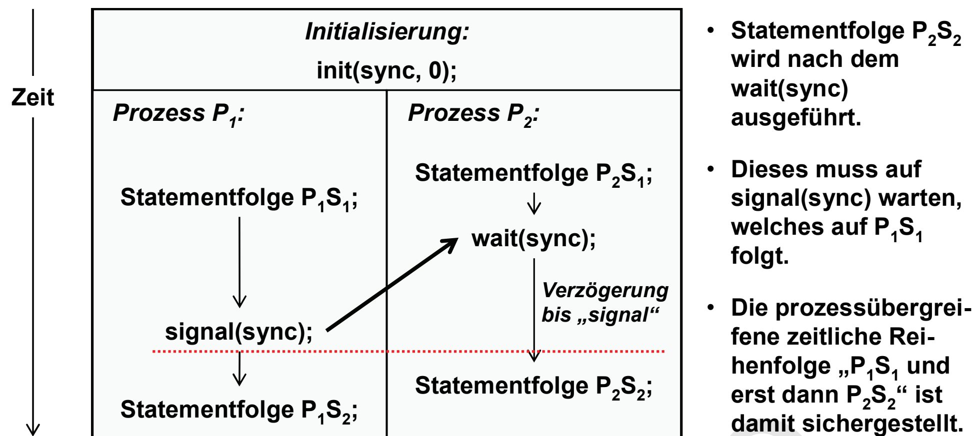


Abb. G-15 *Synchronisation über Semaphore mit wait und signal*

P_2 kann die Anweisungsfolge S_2 erst ausführen, wenn P_1 `signal(sync)` ausgeführt hat: P_2 muss auf P_1 warten, beide Prozesse werden synchronisiert. Dieses Beispiel, erweitert auf 2 Semaphore, werden wir im Abschnitt über Systemverklemmungen (Deadlocks) als Ausgangspunkt der Diskussion wieder aufgreifen.

G.4.7.4.2 Critical Region (CR)

Eine Critical Region CR kann über eine Semaphore `mutex` (die Wahl des Namens ist an mutual exclusion angelehnt) realisiert werden, die mit 1 bzw. `true` zu initialisieren ist. Vor dem Betreten der Critical Region muss ein Prozess ein `wait(mutex)` ausführen. Unmittelbar nach dem Verlassen der CR ist ein `signal(mutex)` erforderlich, damit ein anderer Prozess an die Reihe kommen kann. Das Signal `signal(mutex)` teilt also mit, dass die CR nunmehr wieder frei geworden ist.

Im Pseudocode verwenden wir wieder die Typvereinbarung `semaphore`.

```
semaphore mutex;
init(mutex, 1);
...
wait(mutex);
     { betrete CR; verlasse CR;}
signal(mutex);
```

Code G-23 *Mutex Semaphore in Pseudocode zur Realisierung einer CR*

G.4.7.4.3 Producer-Consumer-Problem mit Semaphoren

Nun sind wir gerüstet, das am Anfang des Kapitels vorgestellte Producer-Consumer-Problem mit allen erforderlichen Nebenbedingen zu lösen:

- Zugriff auf den Puffer: unter mutual exclusion
- Reihenfolge: Erzeugte Daten (items) müssen in der Reihenfolge aus dem Puffer genommen werden, in der sie dort abgelegt worden sind

Leseprobe

- Kein busy waiting vor einem leeren Puffer für den Consumer
- Kein busy waiting vor einem vollen Puffer für den Producer

Wir verwenden dazu drei Semaphore:

- `mutex`:
  Sichert mutual exclusion für den Pufferzugriff.
- `empty`:
  Repräsentiert die Anzahl der freien Plätze und wird mit der Pufferkapazität n durch `init(emtpy, n)` initialisiert. Zählen ist atomar!
- `full`:
  Repräsentiert die Anzahl der belegten Plätze und wird mit `init(full,0)` initialisiert. Zählen ist atomar!

Der Produzent sieht dann schematisch wie folgt aus.

```
do {
      Item erzeugen;
      wait(empty);
            wait(mutex);
                  /* Critical Region */
                  Item in Puffer einfügen;
            signal(mutex);
      signal(full);
} while (1);
```

Code G-24 *Producer-Consumer: Produzent in Pseudocode realisiert mit Semaphore*

Der Konsument ist im Prinzip symmetrisch dazu:

```
do {
      wait(full);
            wait(mutex);
                  /* Critical Region */
                  Item aus Puffer entnehmen
            signal(mutex);
      signal(empty);
} while (1);
```

Code G-25 *Producer-Consumer: Konsument in Pseudocode realisiert mit Semaphore*

Durch `wait(mutex)` (siehe Code G-24 und Code G-25) ist mutual exclusion beim Zugriff auf den Puffer gesichert.

Der Konsument (siehe Code G-25) überprüft mit `wait(full)`, ob ein Element im Puffer zum Entnehmen ist, ansonsten schläft er. Die Operation `signal(empty)` wird verwendet, um die Anzahl der leeren Positionen im Puffer zu zählen (also ein `inc(empty)`), sowie um einen schlafenden Produzenten zu wecken, falls dies notwendig ist.

Leseprobe

Die Lösung ist symmetrisch. Im umgekehrten Fall erhöht der Produzent (siehe Code G-24) via `signal(full)` die Anzahl der belegten Pufferpositionen, nachdem ein neues Element in den Puffer gestellt wurde (also ein `inc(full)`). Falls `full` vorher gleich Null war, wird ein schlafender Konsument geweckt.

G.4.8 Kritische Region in Java

Java stellt das `synchronized` Statement zur Verfügung, um eine CR zu implementieren. Das `synchronized` Statement versucht eine exklusive Sperre für ein Objekt zu erhalten, das durch einen Ausdruck spezifiziert wird. Die CR wird nicht ausgeführt, bevor die Sperre nicht im Besitz ist.

```
public static void SortIntArray(int[ ] a) {
	/* sortiere Array a, sodass kein anderer Thread es manipulieren kann,
	während es sortiert wird */
	synchronized (a) {
		Algorithmus zum Sortieren von a;
	} /* synchronized */
} /* SortIntArray */
```

Code G-26 *Critical Region realisiert in Java durch synchronized*

Leseprobe

G.4.9 Monitore

G.4.9.1 Allgemeines

Monitore stellen ein weiteres Synchronisationskonzept zur Verfügung. Sie sind ähnlich zu abstrakten Datentypen und übertragen das Prinzip einer Datenkapsel auf den Zugriff mittels paralleler Prozesse. Monitore wurden von P. Brinch Hansen und A. Hoare, voneinander unabhängig, eingeführt (1973,1974).

Es gibt eine Reihe von Zugriffsfunktionen, die vom Anwender definiert werden, und eine Menge von Variablen, auf die damit zugegriffen wird. In objektorientierter Programmierung heißen solche Zugriffsfunktionen Methoden, die in der Deklaration einer Klasse definiert und auf Objekte dieser Klasse angewandt werden.

Die Zugriffsfunktionen, bzw. Methoden arbeiten also in gewohnter Weise, jedoch wird Mutual Exclusion dabei beachtet. Auf anstehende, wartende Zugriffe (Methodenaufrufe) wird ein Scheduling ausgeführt. Wir gehen darauf hier nicht weiter ein.

G.4.9.2 Monitore in Java

Das vorhin genannte Monitor-Konzept wurde auf Java übertragen. Wie bei den Kritischen Regionen wird wieder das Schlüsselwort `synchronized` verwendet. Jede Klassenmethode muss wie aus Code G-27 ersichtlich deklariert werden. Dies bewirkt:

- Jedes erzeugte Objekt ist implizit mit einem Monitor verbunden.
- Eine `synchronized` Methode sperrt das Objekt, bevor sie ausgeführt werden kann. Jeder andere Thread, der versucht ein gesperrtes Objekt anzusprechen, kommt in eine zugehörige Warteschlange.

```
public class BankKonto {
	private double kontostand;

	public synchronized void geldabheben(double betrag) {
		kontostand = kontostand - betrag;
	} /* Abheben */

	public synchronized void geldeinzahlen(double betrag) {
		kontostand = kontostand + betrag;
	} /* Einzahlen */
} /* BankKonto */
```

Code G-27 *Monitor in Java, Sperre auf Objekte*

Das `synchronized` Schlüsselwort zeigt an, dass die Methode als Ganzes eine Critical Section ist. Die Sperre erfolgt nur beim instanzierten Objekt. (Statt „Instanz“ sollte man besser „Exemplar“ sagen, denn eine Instanz ist im Deutschen etwas Anderes als „instance“ auf Englisch).

Falls `static` hinzugefügt wird, erfolgt die Sperre bei der Klasse:

```
	public static synchronized void geldabheben(double betrag) {
		...
	}
```

Code G-28 *Monitor in Java, Sperre auf Klasse*

Leseprobe