Technische Informatik

anonyme Zusammenfassung einer PM-Gruppe

Sommersemester 2004

1 Vorwort

Diese Volltext-Zusammenfassung ersetzt sicher nicht die Lektüre des Buches „Einführung in die Technische Informatik“ von G.H. Schildt, A. Redlein, D. Kahn. Die Zusammenfassung ist nur für Studienzwecke gedacht und basiert vor allem auf oben genannten Buch sowie eigenen Mitschriften.

2 Das Mealy Schaltwerk

Das Mealy Schaltwerk unterscheidet sich vom Moore Schaltwerk hauptsächlich (bzw. eigentlich nur) dadurch, dass das Ergebnis nicht nur vom derzeitigen Zustand, sondern auch von den Eingabesignalen abhängt. Daher hat das Mealy Schaltwerk die Möglichkeit sofort auf Änderungen der Eingänge reagieren, es “kann mehr”. Die Eingänge können also die Ausgänge direkt beeinflußen.

Beim Moore Schaltwerk hat man hingegen die Einschränkung, dass sich Ausgangsänderungen immer erst durch eine Zustandsänderung realisieren lassen, was aber auch den Vorteil hat, dass das Moore Schaltwerk immer schön mit dem Takt synchronisiert ist.

Beim Moore Schaltwerk benötigt man im allgemeinen mehr Zustände zur Lösung eines Problems als beim Mealy-Schaltwerk. Das Moore-Schaltwerk ist jedoch trotzdem oft einfacher zu durchschauen.

Zur grafischen Darstellen des Mealy-Schaltwerks im Vergleich zum Moore Schaltwerk, siehe Buch Seite 110, Abbildung 4.36

Realisieren kann man das Mealy-Schaltwerk wie beim Moore-Schaltwerk entweder durch eine dichte oder durch eine 1:n Zustandscodierung.

2.1 Mealy-Moore-Transformation

Das Moore-Schaltwerk braucht nicht in ein Mealy-Schaltwerk umgewandelt werden, da es einen Spezialfall des Mealy-Schaltwerks darstellt.

Umgekehrt ist eine Umwandlung nicht gerade einfach, und ausserdem gar nicht 1:1 möglich. Die sofortige Reaktion der Ausgänge auf die Eingänge, wie dies beim Mealy-SW möglich ist, kann beim Moore-SW nämlich gar nicht erreicht werden. Man kann jedoch eine Umwandlung durchführen, wenn man sich damit begnügt, dass das gleiche Ausgangsbitmuster erzeugt wird - man also auf das genaue Zeitverhalten verzichtet.

Bei der Umwandlung müssen dabei neue Zustände eingeführt werden, die die zusätzlichen Möglichkeiten darstellen. Dies kann man sehr gut am Beispiel im Buch auf Seite 112 und der Abbildung 4.39 auf der selben Seite nachvollziehen.

2.2 Taktfrequenz des Mealy-SW

Wie beim Moore-SW ergibt sich die maximale Taktfrequenz aus dem Reziprokwert der Summe aus Flip-Flop-Durchlaufzeit, Durchlaufzeit der Übergangsfunktion und der Flip-Flop-Vorbereitungszeit. Die Durchlaufzeiten der Ausgangsfunktion verändern die Taktfrequenz nicht, allerdings hinken dadurch die Ausgänge etwas nach. (Sie sollten natürlich nicht allzusehr nachhinken, wodurch man die Durchlaufzeit der Ausgangsfunktion natürlich nicht ganz ignorieren kann).

3 Prozessoren

Dieser Abschnitt soll einen kurzen Überblick über den Aufbau von Prozessoren geben.

3.1 Bestandteile eines Prozessors

Die Erstellung von eigenen Hardware zur Erstellung diverser Aufgaben erfordert viel Aufwand und bringt wenig Flexibilität. Daher ist es wünschenswert ein System zu haben, das mehr Funktionalität besitzt.

Ein moderner Prozessor ist zumindest aufgebaut aus:

- Arithmetical Logic Unit (ALU)
- Register File und Busverbindungen
- Datenspeicher
- Control Unit

3.1.1 Arithemtical Logic Unit

Eine ALU dient zum verarbeiten von Binärdaten in der Größe eines Speicherwortes und hat im Allgemeinen folgende Funktionalitäten

- addieren
- boolesche Operationen durchführen
- Komplement bilden
- (unverändertes Durschalten eines Datenwortes)

Eine ALU benötigt weiters eine Steuerlogik, die es ermöglicht die Funktion auszuwählen. Die "einfache ALU" die im Buch Verwendung findet unterstützt folgende 4 Funktionen

- parallele Addition von 2 Datenworten mit je 16 bit
- bitweise UND Verknüpfung von 2 16 bit Datenworten

- bitweise Komplementbildung eines Datenwortes (Einerkomplement)
- unverändertes Durchschalten eines Datenwortes

und benötigt daher 2 Steuerleitungen F0 und F1, da mit diesen die 4 Funktionen codiert werden können

Microinstruction	Bedeutung	Symbolisch
00	A unverändert durchschalten	R←A
01	A und B addieren	R←A+B
10	A und B bitweise &-Verknüpfen	R←A∧B
11	A negieren	R←¬A

Unsere ALU besitzt folgende Eingänge

Register A 16 bit langes Register, in dem sich ein Datenwort befindet, das von der ALU als nächstes bearbeitet wird.

Register B das zweite 16 bit lange Register, in dem sich das zweite Datenwort befindet.

Unsere ALU besitzt außerdem folgende Ausgänge

Ausgang R hier wird das Ergebnis der Operation ausgegeben

Ausgang N Vorzeichenanzeige, dient Ausgang Z Nullanzeige. Ausgang hat den Wert 1, wenn im Ergebnis alle Bits null sind.

Dadurch, dass es man also die Operation auswählen kann, steht uns also eine programmierbare Hardware zur Verfügung. Die Information, die sich im Register (F0F1) befindet, wählt die Operation aus - wir nennen sie Mikro-Instruktion.

Mikro-Instruktion ist also die Information, die sich im Register (F0F1) befindet

Mikro-Operation ist die Ausführung eines Befehls

Mikro-Code (des Prozessors) ist die Summe aller Mikro-Instruktionen

Die Verarbeitungsdauer einer Mikro-Instruktion beträgt einen Maschinenzyklus. (Das ist die Zeit von einem positiven Clock-Signal zum nächsten)

3.1.2 Verwendung der Operationen

Die Verwendung der + und ¬Operation funktioniert wie erwartet, die Funktion (00) werden wir beim Speicherzugriff benötigen, und die bitweise∧-Verknüpfung dient zum Ausblenden eines Teils des Registers, während der anderen Teil unverändert erhalten bleiben soll.

A= 10101010 01010101
B= 00000000 11111111

C= 00000000 01010101

In moderneren ALUs sind mehr Funktionen realisert, für eine einfache funktionstüchtige ALU genügen aber diese 4 Funktionen.

3.1.3 Der Shifter

Zum Speichern des Ergebnisses dient ein Schieberegister, der sogenannte Shifter. Der Shifter kann folgende Operationen durchführen

(S0S1)	Beschreibung	symbolisch
(00)	keine Veränderung	SH←R
(01)	shift left	SH←lsh(R)
(10)	shift right	SH ←rsh(R)
(11)	nicht gültig	-

Um diese Funktionen auswählen zu können, muss der Mikro-Code erweitert werden. Man benötigt die 2 Steuerleitungen S0 und S1.

3.1.4 Register File und Busverbindungen

Die Verwendung von nur 2 Registern führt zu einigen Einschränkungen. Es ist daher wünschenswert mehrere Register zur Verfügung zu haben. Am besten eigenen sich dafür sogenannte Busverbindungen. Das sind parallele Leitungen (soviele, wie ein Register Bits hat) an die, die einzelnen Register angeschlossen werden.

An einem solchen 16 bit breiten Bus können wir zum Beispiel einige unserer Register anschließen. Um nun zu verhindern, dass mehr als ein Register auf den Bus zugreift, benötigt man eine Steuerung, die dafür sorgt, dass jeweils nur Speicher den Bus benutzen darf. Diese Steuereinheit nennt man Arbitration Logic oder Bus Arbiter.

Weiters wollen wir das Ergebnis auch im Speicher ablegen können, wodurch wir eine Busverbindung vom Shifter zum Speicher benötigen. In unserem Beispiel gibt es daher die folgenden Bus-Verbindungen:

- A- und B-Bus verbinden die Speicher mit dem A- bzw. B-Register
- Das Ergebnis kann vom Shifter über den S-Bus an den Speicher übertragen werden.

Im Beispiel gibt es insgesamt 16 Register. Um nun die jeweiligen Register auswählen zu können, benötigt man pro Bus (A-, B- und S-Bus) jeweils für jedes Register eine Enable-Leitung. Diese kann man durch die Verwendung eines Decoders mit 4 Steuerleitungen je Bus Ansprechen, der Mikro-Code muss also entsprechend um 12 Bit erweitert werden und besteht somit aus 16 Bit:

F0F1 sind die Bits die benötigt werden, um einen Operation auszuwählen (Durchschalten, $+$, $\wedge$und $\neg$)

S0S1 sind die Bits, die benötigt werden um den Shifter zu bedienen

A0A1A2A3 benötigen wir zur Auswahl, welches Register auf den A-Bus gehört, und sich somit im A-Register der ALU befindet. Das A (bzw. B) Register dient als Zwischenspeicher für die ALU.

B0B1B2B3 selbiges nur für den B-Bus

S0S1S2S3 gibt das Register an, in dem das Ergebnis gespeichert wird.

Drei der Register sind für einen speziellen Zweck reserviert, sie beinhalten die Konstanten +1,-1,0. Von diesen Registern kann man nur lesen.

- Register (0)10 beinhaltet 0
- Register (1)10 beinhaltet +1
- Register (2)10 beinhaltet -1
- und das Register (15)10 nennen wir AC (Accumulator)

Eine derartige Anordnung von Registern nennen wir Scratchpad oder Register File, und verwenden in Zukunft dafür ein einziges Schaltsymbol.

3.1.5 Steuerleitungen, Control-Unit

Um die korrekte Befehlsausführung zu gewährleisten, ist es notwendig, den zeitlichen Ablauf zu kontrollieren. Beispielsweise darf der S-Bus erst dann freigegeben werden, wenn das Ergebnis im SH stabil vorliegt.

Dazu verwendet man 3 Steuerleitungen C1, C2 und C4. C3 gibt es, wird aber bis jetzt nicht verwendet.

Einen Teil der Control-Unit, auf die später noch eingegangen wird, stellt eine Schaltung dar, die diese Trigger so steuert, dass sich folgender Ablauf der Operationen ergibt

Trigger 1 aktuelle Mikro-Instruktion laden, Daten auf A- und B-Bus legen, Auswahl der ALU-Funktion und des Shift-Registers.

Trigger 2 Versorgung der Register A und B mit den Daten vom Bus

Trigger 3 bis jetzt nicht genutzt (Zeit zur Durchführung der Operation durch ALU und Shifter)

Trigger 4 Ergebnis vom S-Bus in das Zielregister laden

3.1.6 Speicheranbindung

Um die vorhandenen Speichermöglichkeiten zu erhöhen, solle eine Anbindung an ein RAM- bzw. ROM geschaffen werden.

Wir benötigen dazu 2 Register, das MAR und das MBR

MAR (Memory Adress Register) in diesem Register befindet sich die Adresse des Speichers auf den zugegriffen werden soll. Es ist mit dem Adress-Bus des Speichers verbunden. Am Adress-Bus kann ständig die Information des MAR abgelesen werden.

MBR (Memory Buffer Register) hier wird der Inhalt der vom Speicher gelesen wird bzw. in den Speicher geschrieben werden soll gespeichert. Dieses Register ist über den Datenbus des Speichers mit diesem verbunden.

Die Übernahme der Daten wird durch ein Load-Signal geregelt. Weiters benötigt man noch folgende Steuerleitungen:

- read/write zur Festlegung der Datenrichtung
- memory select zur Festlegung des Zeitpunktes des Transfers

3.1.7 Lesen der Daten aus dem Speicher

1. schreiben der Adresse der gewünschten Speicherzelle ins MAR, Aktivierung von read und memory select
2. Abwarten der Speicherzugriffszeit (jene Zeit, die der Speicher braucht, bis Daten am Datenbus anliegen)
3. Das MBR kann das Datenwort einlesen

3.1.8 Schreiben von Daten in den Speicher

1. Ablegen der Adresse im MAR, Speichern des Datums im MBR, Aktivieren von write und memory select.
2. Während der Speicherzugriffszeit müssen MAR und MBR die korrekten Daten enthalten

3.2 Control Unit

MIR Micro Instrcution Register. Das ist das Register, das die Mikro-Instruktion enthält. Man benötigt jetzt noch eine Einheit, die

- den nächsten Mikro-Befehl einliest
- den Befehl decodiert
- die Steuersignale in zeitlich richtiger Reihenfolge setzt

Das macht die sogenannte Control-Unit. Gelesen werden die Micro-Befehle aus einem speziellen Speicher, dem Micro-Code-ROM, der in unserem Fall 256 Codewörter enthalten kann. Zur Adressierung der Befehle verwendet man den Micro-Instruction Counter, kurz MIC.

Der Micro-Code-ROM ist über parallele Leitungen mit dem MIR verbunden, die der Befehl der an der Stelle des MIC steht, wird über diese in das MIR übertragen.

Durch inkrementieren des MIC um jeweils 1 nach dem Instruction Fetch kann man dadurch ein sequentielles Programm abarbeiten, jedoch keine Sprünge oder dergleichen.

Dafür führen wir die Parallel-Load-Funktion des MIC ein und erweitern weiters den MIC sowie den Mikro-Befehl um 8 bit, wobei das neue Feld die Zieladresse des Sprungbefehles enthalten soll. Da wir weiters nicht nur unbedinge, sondern auch bedingte Sprünge haben wollen, benötigt man noch eine logische Schaltung (Micro Sequence Logic), die zwei weitere Bits (COND, von condition) sowie die Steuerleitungen N und Z der ALU verarbeiten kann.

- Cond=(00) kein Sprung
- Cond=(01) Sprung, wenn N=1, if N goto ADR
- Cond=(10) Sprung, wenn Z=1, if Z goto ADR
- Cond=(11) unbedinger Sprung

ENS Enable-S Bus, dadurch ist es möglich eine Verknüpfung zweier Register auszuwerten, ohne das Ergebnis zu speichern.

4 Computersysteme

4.1 Prozessoren

4.1.1 Maschinencode

Unter dem Maschinencode versteht man eine auch dem Anwender zugängliche Programmiersprache des Prozessors. Der Maschinencode ermöglicht es durch eine Sequenz von Mikro-Anweisungen ein aus mächtigeren Befehlen bestehendes Programm auszuführen. Maschinenbefehle befinden sich im Gegensatz zum Mikro-Code (Micro-Code-ROM) in einem Speicher (RAM oder ROM), eine derartige Software kann daher viel umfangreicher sein.

Maschinen-Code die Menge aller definierten Bit-Kombinationen, die es erlaubt in einer vorgegebenen Hardware bestimmte Funktionen zu programmieren

Maschinenbefehl eine einzelne derartige Bitkombination

Nicht nur die Kommandos, auch die Daten werden im RAM gespeichert. Weiters benutzen beide Arten von Informationen die gleichen Busse.

Hauptspeicher die Speicherbausteine, die direkt mit dem Prozessor verbunden sind.

Interpreter damit der Prozessor die Maschinen-Befehle ausführen kann, muss eine Software (in Mikro-Code) existieren. Das ist der Interpreter.

Aufgaben, die der Interpreter bewähltigen muss:

1. Den nächsten Maschinenbefehl aus dem Speicher des IR (Instructino Register) einlesen
2. Befehl decodieren, welche Operationen müssen auf welche Operanden angewandt werden.
3. Durchführen der Operation und Abspeichern des Ergebnisses entsprechend der Anweisung
4. Fortsetzen bei Punkt 1

Es handelt sich also um eine Endlosschleife, die beim Einschalten des Prozessors beginnt, und mit dem Ausschalten endet.

Program Counter "A register in the central processing unit that contains the addresss of the next instruction to be executed. The PC is automatically incremented after each instruction is fetched to point to the following instruction."

Nach der Ausführung aller Micro-Befehle, die zum aktuellen Maschinenbefehl gehören, wird der PC inkremientiert und der nächste Befehl wird vom Interpreter eingelesen.

Die verschiedenen Maschinenbefehle kann man in verschiedene Kategorien einteilen:

- Transfer-Operationen
- Input/Output Operationen

- Arithmetische Operationen
- Logische Operationen
- Shift-Operationen
- Flow-Control
- Sprünge
- Subroutine-Calls
- Interrupts
- Stack-Operationen

Transferoperationen (Move-Operationen)

Daten werden von einer Quelle zu einem Ziel übertragen ohne verändert zu werden. Eigentlich handelt es sich um Kopien. Transferoperationen die Daten im Hauptspeicher bearbeiten, nennt man auch oft Load- (Daten ins Register laden) bzw. Store-Operationen (Inhalt des Registers speichern)

Input/Output Operationen

Ähnlich wie Transferoperationen, jedoch Austausch mit Peripheriegeräten.

Port ein Port ist ein Register, nur befindet es sich ausserhalb des Prozessors. Jeder Rechner besitzt Ports zur Durchführung von I/O-Befehlen.

Die Steuerung der Peripheriegeräte erfolgt durch Eintragen der Steuerbefehle in einen oder mehrere Ports. Zur Auswahl des richtigen Ports benötigt die I/O-Operation die Adresse, diese kann man auf 2 Arten angeben:

- independent I/O-System
  - Hauptspeicher und Ports besitzen völlig unabhängige Adressen.
  - Es werden daher für Transfer- und I/O-Operationen auch unterschiedliche Befehle verwendet
  - Vorteil: Adressraum steht im vollen Umfang zur Verfügung
  - daher auch isolated I/O
- memory-mapped I/O-System
  - Ports werden behandelt als wären sie gewöhnliche Speicherzellen
  - Wohin die Adresse gehört, ist nur durch die Verdrahtung festgelegt
  - es finden die selben Befehle für Transfer- und I/O-Operationen Anwendung

Arithmetische Operationen

Die meisten Prozessoren beherrschen zumindest die 4 Grundrechnungsarten. Als spezielle Operationen gibt es meist auch das increment und das decrement sowie die bitweise Inversion sowie ein compare-Befehl. Dieser subtrahiert die Operanden und setzt dann die Status-Flags.

Bei der Ausführung arithmetischer Operationen können auch Fehler auftreten, z.B. ein "Overflow". Dafür gibt es zur Dokumentation ein PSW bei den meisten Prozessoren.

PSW Program Status Word, die einzelnen Bits des PSW besitzen eine bestimmte Bedeutung, und zeigen z.B. overflows an. Der Anwender kann sie lesen, und etwaige Fehler dadurch erkennen.

Das Ergebnis dienst oft zur Steuerung des Programmflusses (z.B. Sprünge). Weiters erlauben einige Befehle das Bearbeiten dieser Bits, näheres dazu unter Flow-Control.

Logische Operationen

Dazu zählen die unären Befehle die unabhängig vom Vorzustand eines Wertes auf logisch 1 oder logisch 0 setzen, oder das Komplement bilden. Weiters die boolschen Funktionen AND, OR, XOR.

Shift-Operationen

Flow-Control

Diese Befehle unterbruchen den sequentiellen Verlauf eines Programms. Neben den aus dem Mikro-Code bekannten Sprüngen, gibt es vorallem Subroutine-Calls und Interrupts.

Sprünge

Jump-Operation unbedingter Spurng im Maschinencode, analog zum unbedingten Sprung in der Register-Transfer-Ebene. der PC wird mit einem neuen Wert geladen, statt einfach nur erhöht zu werden.

Branch-Operations bedingte Sprünge, hier gibt es mehrere Varianten (z.B. Ergebnis abhängig von einem Bit-Test, Wort-Test oder auch Vergleich von 2 Worten)

Subroutine Calls

Teile von Programmen, die öfter verwendet werden, werden als Subroutinen öfters aufgerufen. Sie werden dabei vom Befehl "Subroutine-Call" aufgerufen. Am Ende wird der Befehl "Return-from-Subroutine" aufgerufen, das Programm setzt hinter dem zuletzt ausgeführten Call-Befehl fort. Ein Problem stellt das Speichern der Adresse dar, zu der das Programm zurückspringen soll, da z.B. durch das verschachtelte, oder gar rekursive Aufrufen einer Funktion, diese Adresse nicht in einem Register gespeichert werden kann. Als Lösung bietet sich ein Stack an. Weiters besteht die Gefahr, dass Registerwerte ungewollt verändert werden. Diese werden daher auch über Stacks "gerettet".

einfügen: Aufgaben von Call-Subroutine, Return-from-Subroutine

Interrupts

Interrupts unterbrechen den normalen Programmablauf und springen in bestimmte Serviceroutinen, die Interrupt Service Routine (ISR). Die ISR ist mit einer normalen Subroutine vergleichbar, wird aber nicht durch ein Call-Subroutine ausgelöst, sondern durch ein externes Ereignis oder einen prozessorientierten Ausnahmefall (z.B. Division durch null). Interrupts die durch interne Ausnahmefälle hervorgerufen werden, bezeichnet man auch als Traps.

Wenn nun ein Interrupt auftritt, übernimmt eine externe Hardware, die Interrupt-Logik die Steuerung der CPU. Sie überprüft ob es sich um einen internen oder externen IR handelt und unterbricht wenn es sinnvoll ist, den Programmablauf - dabei wird der Programmstatus (i.e. alle Register, der PC und das Statusregister) gerettet, damit nach dem Abarbeiten des ISR wieder normal fortgesetzt werden. Manche Prozessoren besitzen eigene Registerbänke und schalten bei einem Interrupt einfach um.

Danach wird die IR-Quelle festgestellt, dafür gibt es mehrere Konzepte

1. Es wird stets die gleiche Startadresse der ISR im Speicher angesprungen, die ISR stellt selbstständig fest, welche Quelle den IR hervorgerufen hat.
2. Der Prozessor hat hardwaremäßig ur viele IR-Eingänge und weiß, welche Quelle den IR ausgelöst hat.
3. Jede Quelle schickt mit dem Signal eine Identifikationsnummer mit.

Die Adresse des ISR kann logischerweise nicht Teile eines Befehles sein, sondern muss durch die Interrupt-Logik auf Grund der auslösenden Quelle bestimmt werden. Die Zuordnung der IR-Quelle zur Adresse des ISR geht natürlich auch auf mehr als nur eine Art:

1. Fixe Zuordnung
2. Interruptvektor

Zum zurückkehren, gibt es den Befehl Return-from-Interrupt.

Manchmal ist es auch nötig, bestimmte IR zu sperren. Das geht entweder im PSW oder es gibt ein eigenes Register (IR Control Register) dafür. Manchmal gibt es auch ein Bit, mit dem man alle IRs ausschalten kann (Interrupts disabled / Interrupts enabled). Durch eine Interrupt-Mask können bei aktiviertem Interrupts-enabled gewisse IRs ausgeblendet (maskiert) werden. Manche IRs kann man jedoch nicht maskieren (Non-maskable IRs).

Stack Operationen

Der Stack ist ein besonderer Speicherbereich der sich normal im Arbeitsspeicher befindet. Es gibt fix im Arbeitsspeicher befindliche, oder auch von Programmen angelegte Stacks, weiters besitzen manche CPUs einen Stack am Chip (hardware stack).

Verwendung finden Stacks z.B. zum retten des PSW vor einem Interrupt aufruf.

Die Funktionsweise ist einfach, die Push- und Put-Optionen sollten jedem Informatiker ein Begriff sein.

Der Hauptvorteil des Konzepts liegt darin, dass die Adressverwaltung vom System übernommen wird, und der Benutzer die Adressen nicht kennen muss.

Das System selbst muss wissen, an welcher Stelle im Speicher sich das oberste Element befindet, es benutzt dafür ein Register, den sogenannten Stackpointer (SP). Der Wert dieses Registers wird nur durch den Push bzw. den Pop Befehl verändert.

Buch Seite 147 - Beispiel für einen Code für Stack-Operationen.

Durch einen Stack ist es auch möglich die Probleme die im Zusammenhang mit ineinandergeschachtelten Subroutinen auftauchen (speichern des PSW, Rücksprungadressen...) zu bewältigen, da diese immer oben auf den Stack gelegt werden, und daher wieder als erste entnommen werden. (LIFO, der Stack wird daher auch LIFO-Speicher genannt).

4.1.2 Adressierungsarten

Die Adressierungsarten werden im Buch sehr detailiert beschrieben. In dieser Version soll nur eine kurze Auflistung der vorhanden Adressierungsarten erfolgen.

- Implied Mode
- Register Mode
- Immediate Mode
- Direct Addressing Mode
- Register-Indirect Mode
- Program-Counter-Relative-Addressing Mode
- Indirect Adressing Mode

4.1.3 Architekturen

Der geneigte Student möge sich derzeit diese informativen 1,5 Seiten über Architekturen auf den Seiten 152/153 durchlesen ;-)

4.1.4 Parallelverarbeitung innerhalb eines Rechners

Ein Maschinenbefehl kann im Gegensatz zu einem Micro-Code-Befehl nicht in einem Taktzyklus abgearbeitet werden. Es gibt nun verschiedene Möglichkeiten die Performance zu steigern.

Vektorverarbeitung Wenn hintereinander die selben Befehle auf verschiedene Datensätze angewendet werden, kann man diese Operanden gleichzeitig in mehrere Register laden, und die Operation parallel darauf ausführen.

Man braucht dafür genug Register und ALUs.

Superskalare Verarbeitung Hier werden die einzelnen Bestandteile, die zur Abarbeitung eines Befehls notwendig sind weitgehend als eigene Funktionen zur Verfügung gestellt und können daher entsprechend parallel ausgeführt werden. (Entsprechende Verarbeitungseinheiten vorausgesetzt)

Welche Kommandos parallel verarbeitet werden können, muss speziell analysiert werden. Zur Realisierung dieser Analyse verwendet man Markierungen, auch scores genannt.

Problem: Abhängigkeiten, besonders bei komplexen Abhängigkeiten wird es komplizierter.

Weiters kann eine effektive Ausnutzung trotz des mehrfachen Schaltungsaufwandes nicht garantiert werden.

Ein weiters Feature ist die Verwendung von speziellen Co-Prozessoren für gewisse Funktionen. Die Anbindung eines solchen Co-Prozessors ist auf verschiedene Arten möglich:

1. Vollständig sichtbare Anbindung: CPU Instruktionssatz enthält auch Instruktionen für die Co-Prozessoren
2. Partiell sichtbare Anbindung: Wie im ersten fall Instruktionen für Co-Prozessoren im CPU-Instruktionssatz, CPU und Co-Prozessor arbeiten aber partiell unabhängig
3. Transparente Anbindung: CPU sieht Co-Prozessor nicht. Der Co-Prozessor interpretiert dabei bestimmte Speicheradressen der CPU als Co-Prozessor-Instruktionen. Die CoP arbeiten unabhängig von der CPU

Instruction Piplining Beim Instruction Piplining wird der Hardware-Aufwand unter Umständen nicht im vollen Umfang verfliefacht, sondern nur Teile bestehender Hardware besser ausgenutzt. Vergleichen läßt es sich am besten mit Fließbandarbeit, da hier die Arbeit in Teile zerlegt wird, die verschiedenen Arbeitern zugeteilt wird. Die dauer, bis sich das Fließband weiterbewegt, wird jeweils von dem Prozess bestimmt, der am längsten benötigt. Die schnelleren werden also zu einer Pause gezwungen. Daher ist es beim Piplining wichtig, dass die Prozesse ähnlich lange dauern, damit keine Leerlaufzeiten entstehen.

Um dies bei unserem Micro16 Prozessor zu realiseren, unterteilen wir den Arbeitsvorgang "Maschinenbefehl ausführen" noch weiter

1. Instruction Fetch
2. Instruction Decoding
3. Address Generation
4. Operand Fetch
5. Execute
6. Store Result
7. Update Program Counter

Für jeden dieser Schritte wird eine eigene Verarbeitungseinheit reailisiert. Diese Stufen sind über Latches, Daten- und Steuerpfade so verbunden, dass sie parallel arbeiten können. (siehe Abbildung auf Seite 157).

Nachdem sich die Pipeline gefüllt hat, wird in jedem Maschinenzyklus ein Befehl erzeugt. Es ändert sich zwar nicht die Zeit, in der der Befehl verarbeitet wird, diese steigt sogar etwas an, es werden aber mehrere Befehle in der gleichen Zeit erledigt.

Im Beispiel sieht man auch, dass ab T6 alle Stufen arbeiten, jede aber mit einem anderen Befehl beschäftigt ist.

Folgende Voraussetzungen müssen für die Realisierung von Pipelining erfüllt sein:

- Falls die einzelnen Befehlsteile keine einzelnen Hardware-Betriebsmittel verwenden, kann man sie einfach ausführen

- Damit sie sich wenigst möglich beeinflußen, braucht jede Einheit eigene Latches zur Aufnahme des aktuellen Datenwortes
- Um die Teilergebnisse von einer Stufe zur nächsten zu bringen ist ein Controller notwendig, dieser erhöht die Komplexität der Hardware.
- Anpassung des Maschinencodes muss erfolgen um die effiziente Nutzung zu garantieren
- Befehlsformat mit fixer Länge erleichtert sequentielles Laden

Harvard-Architektur (siehe Abbildung 5.6, Seite 158). Bei vollständig gefüllter Pipeline versuchen mehrere Stufen gleichzeitig auf den Hauptspeicher zuzugreifen, was die klassische v. Neumann-Architektur vor ein Problem stellt. Eine Lösung dafür, stellt diese Architektur dar, die einen komplett parallelen Zugriff ermöglicht.

Probleme die im Programmablauf selbst entstehen:

- Wenn man das Resultet das Befehls der gerade exekutiert wird, im darauffolgenden braucht, kann es zu Konflikten kommen. In diesem Fall werden die Daten gleich übergeben (data forwarding)
- Wenn das Ergebnis ein Register zerstört, welches von der nächsten Instruktion schon eingelesen wurde. Hier kann unter Umständen ein geschicktes Umstellen des Befehls helfen. Diese ersten beiden Arten von Kommandos nennt man auch interferring instructions.
- Nach Sprüngen muss die Pipeline für ungültig erklärt (flush-pipe) und neu geladen werden.

Der letzte Fall kommt erstens sehr häufig vor, und führt zu einem wesentlich geringeren Durchsatz des Systems. Daher sind bessere, aber auch kompliziertere Techniken entwickelt worden:

Unbedingte Sprünge: diese lassen sich frühzeitig erkennen, so dass die Instruction-Fetch-Unit an der neuen Stelle im Programm fortsetzen kann, noch bevor die Stufe ST7 den PC ändert.

Bedingte Sprünge: Hier ergeben sich größere Schwierigkeiten. Da Ihr Ziel erst in der Executing-Unit durch Auswertung der Bedingung ermittelt wird, kann es geschehen, dass der gesamte Inhalt der Pipe ungültig ist.

Die einfachste Lösung ist es, den Pipelining-Mechanismus zu stoppen, wenn die Decoding-Unit einen solchen Sprung entdeckt. Freigabe erfolgt erst, wenn die Zieladresse des Sprunges erkannt wurde. Diesen Vorgang nennt man Interlocking. Die Performance dieser Methode ist jedoch immer noch schlecht.

Ein weiterer Ansatz ist es, die dem Sprungbefehl sequientell folgende Instruktion noch auszuführen. Diese Methode geht jedoch nur, wenn dadurch die Sprungbedingung nicht verändert wird. Man nennt das Verfahren Delayed Branch.

Andere Verfahren versuchen, sobald ein Sprungbefehl erkannt wurde, seine Zieladresse vorauszusagen. Das nennt man Predicted Branch, und es soll am Beispiel eines Jump-Befehls bei dem die Adresse des Sprungziels relativ zum PC angegeben ist erläutert werden:

Aus einem negativen Displacement (Rückwärtssprung) schließt man, dass es sich um ein Schleifenende handelt. Da eine Schleife normal öfters durchlaufen wird, lässt sich daraus folgern, dass die Bedingung mit hoher Wahrscheinlichkeit erfüllt ist. Daher wird der Sprung unter dieser Annahme ausgeführt und die Instruction-Fetch-Unit veranlasst, die sequentielle Reihenfolge zu durchbrechen.

Eine weitere Verbesserung kann durch einen Sprungziel-Cache erreicht werden. Das ist eine Tabelle, die Sprungziele bereits ausgeführter Sprünge enthält. Man nennt das Branche History.

4.1.5 CISC versus RISC

Die ersten Erweiterungen der Von-Neumann-Architektur zielten darauf ab, immer mehr Befehle (manchmal über 1000) und komplexere Konstrukte schon auf der Maschinenbefehlsebene zur Verfügung zu stellen. Dies scheint im ersten Augenblick Vorteile mit sich zu bringen. Sie wurden aber durch eine aufwändigere Control Unit und eine komplexe Micro-Programmierung erkauft. In der heutigen Terminologie bezeichnet man solche Produkte als CISC (*complex instruction set computer*).
Die rasante Weiterentwicklung der Software ermöglicht jedoch heutzutage, dass die meisten Programme nicht mehr in komplizierter Maschinensprache erstellt werden müssen, sondern auf benutzerfreundliche Weise in höheren Programmiersprachen erschaffen werden können und durch Compiler automatisch vom menschenlesbaren Source-Code in Maschinen-Code übersetzt werden. Da allerdings die Entwicklung der Compiler mit den Neuerungen der Prozessor-Hardware nicht Schritt halten konnte, und dadurch leistungsstarke Befehle überhaupt nicht verwendet wurden, entwickelten sich neue Ansätze zur Optimierung:

- Beschränkung auf wenige(ungefähr 100 bis 200) Instruktionen, Emulation der restlichen
- Optimale Implementierung dieser geringen Anzahl von Befehlen auf dem Prozessor

Architekturen, die sich aus diesen Aspekten ableiten, nennt man RISC (reduced instruction set computer). RISCs zeichen sich vor allem dadurch aus, dass sie, während ein CISC nur einen Befehl abarbeitet, gleich mehrere Befehle durchführen können.

Eigenschaften von RISCs

- Aufgrund der einfachen Befehle wird praktisch kein Micro-Code benötigt. Die Ablaufsteuerung ist fest verdrahtet.
- Der kleinere Befehlssatz ermöglicht geringere interne Kontrollmechanismen
- Höhere Taktraten durch einfachere Hardware
- Befehlsverarbeitung nach dem Pipeline-Prinzip
- Einheitliche Länge der Befehle führt zu wirkungsvoller Pipeline und effizienter Organisation
- Nur die Load/Store-Befehle kommunizieren mit dem Speicher, alle anderen ausschließlich mit Registern
- Große Anzahl von allgemein benutzbaren Registern zum Speichern der Operanden

- Registerbänke erleichtern Subroutine Calls und ISRs.
- Mehrere Pipelines für unterschiedliche Befehlsgattungen
- On-Chip-Cache zur Reduktion der Dauer von Lade- und Speicherbefehlen
- Entschärfung der Daten- und Befehlsabhängigkeit durch Umordnen der Befehlsfolge
- Spezielle Compiler zur Vermeidung von Pipelinekonflikten

SPARC - Scaleable Processor ARChitecture Ist eine Entwicklung der Berkely Universität bestehend aus einem RISC-Prozessor mit ein 4-stufigen Instruction-Pipeline (Instruction Fetch, Instruction Decoding, Execute, Store Result) und auffallend ausgefeilter Registertechnik, die Programmumschaltungen sehr gut unterstützt.

MIPS-Architektur Steht für Microprocessor without Interlocking Pipelining Stages und wurde von der Stanford Universität entwickelt. Besondere Merkmale sind die feinstufige Befehlspipeline und die realisierte Speicherhierarchie.

4.2 Speicher

Zugriffe auf den Hauptspeicher sind immer sehr langsam im Vergleich zu den restlichen Teilen der Befehlsausführung. Zusätzlich sind in den klassischen Architekturen sowohl die Daten als auch die Programme im Hauptspeicher abgelegt, was auch als "von Neumannscher Flaschenhals"bezeichnet wird. Eine Verbesserung der Zugriffszeit bedeutet auch immer eine Vergrößerung der Bandbreite.

Bandbreite Anzahl der Bits, auf die man pro Sekunde zugreifen kann (Zugriffszeit des Bausteins mal Breite des Datenbusses)
Grundsätzliche Möglichkeiten zur Verbesserung der Bandbreite:

- Schnellere Speicherbausteine Die Verwendung von schnellen SRAMs statt langsameren DRAMs stellt eine denkbare, aber äußerst teure Möglichkeit dar
- Breitere Datenwörter

Eine Verbreiterung der Datenwörter führt oftmals zur Übertragung von unbrauchbarer Information und stellt daher keinen sinnvollen Lösungsansatz dar
Performance-Strategien:

- geschickte Anordnung der einzelnen Speicherbausteine
- Verwendung von Caches
- Verwendung der Harvard-Architektur

4.2.1 Interleaved Memory

Das Konzept des Interleaved Memory bezieht sich, wie so viele andere Strategien der Informatik, auf das alternativ interpretierte Prinzip des „Divide and Conquer“.
Grundsätzlich wird davon ausgegangen, dass hauptsächlich sequenziell auf den Speicher zugegriffen wird, die Befehle also auf hintereinander liegenden Adressen zu finden sind. Man könnte also den nachfolgenden Befehl immer erst nach Beendigung des ersten auslesen. Wenn man aber den Speicher in mehrere, gleich große Speicherbereiche teilt (sogenannte Speicherbänke) und die Adressen gleichmäßig (modulo Anzahl der Speicherbänke) verteilt, so greifen sequenzielle Befehlsfolgen nie auf dieselbe Speicherbank des Vorgängers zu. Eine wesentliche Voraussetzung für Interleaved Memory ist natürlich, dass die einzelnen Speicherbänke auch eine separate Anbindung an das Bussystem (Datenbus, Adressbus, Kontrollbus) haben.

4.2.2 Caches

Um nicht vollständig zwischen den teuren SRAM-Bauteilen und den kostengünstigeren DRAM-Bauteilen entscheiden zu müssen, wird mit sogenannten Caches ein Kompromiss geschlossen. Dadurch entsteht eine Speicherhierarchie, wobei die langsameren Speicher natürlich in größeren Kapazitäten vorhanden sind:
langsamer Hauptspeicher →
schneller Zwischenspeicher (Cache) →
Prozessorregister
Caches bestehen bestehen also aus schnellen SRAMs und verfügen zusätzlich über eine eigene Hardware für das Adressieren, Laden und Auslagern von Speicherinhalten. Sie arbeiten für den Benutzer unsichtbar (transparent) und haben zur Aufgabe dem Prozessor während der Laufzeit die benötigen Daten zu liefern. Dabei gibt es zwei Möglichkeiten:

Cache Hit: Daten wurden im Cache gefunden → schneller Zugriff

Cache Miss: Daten wurden im Cache nicht gefunden und aus dem Arbeitsspeicher geholt → langsamer Zugriff

Die Wirksamkeit eines Caches ist nun abhängig von seiner Trefferrate (*hit rate*) und der sich daraus ergebenden durchschnittlichen (effektiven) Speicher-Zugriffszeit t_{eff}.
hit rate h = (Anzahl Speicheraufrufe (Cache oder HSP)) / (Anzahl Cache Hits)
$t_{eff} = h * t_{cache} + (1 - h) * t_{main}$
Wobei t_{cache} die Zugriffszeit auf den Cache-Speicher und analog t_{main} die Zugriffszeit auf den Hauptspeicher darstellt. Da t_{cache} und t_{main} durch die Speicher-Chip-Technologie bestimmt werden, hängt die effektive Zugriffszeit nur von der Trefferate h ab.
Der Aufbau eines Cache-Memory ist so organisiert, dass jeder Speicherzugriff von der Cache-Logik verarbeitet wird. Dabei wird ein Tag-RAM verwendet, welches ein Verzeichnis von den Adressen der Speicherplätze hat, die sich derzeit im Cache befinden. Der Comparator vergleicht nun bei jeder Schreib- oder Leseoperation die Informationen mit den Werten im Tag-RAM. Bei einer Übereinstimmung wird ein Hit-Signal gesetzt, welche von der Logic-Einheit erkannt wird. Über den Data-RAM wird die benötigte Information auf den Datenbus gelegt. Bei Cache Misses wird das neue”Datum auch im Data-RAM abgelegt um künftige Misses zu vermeiden.
Der Speicherraum im Data-RAM ist jedoch sehr begrenzt, weshalb sich folgende Replacement-Strategien anbieten:

LRU-Methode (Least Recently Used): Es wird jenes Datum gelöscht, wessen letzter Aufruf am längsten zurückliegt

LFU-Methode (Least Frequently Used): In diesem Fall wird jenes Datum gelöscht, das in letzter Zeit am seltensten genutzt wurde

RANDOM: Es wird zufällig ein Datum zur Eliminierung ausgewählt

Eine andere Realisierungsmöglichkeit von Caches ist Direct Mapping. Dabei werden bestimmte Adressen nur in einer vorgeschriebenen Cache-Zeile gespeichert. Das bedeutet, dass ein Wort abhängig von den niedrigsten Stellen seiner Adresse obligatorisch einem genau festgelegten Fach zugewiesen wird.
Ein Problem ergibt sich aber, wenn beispielsweise in einer Schleife sowohl die Adresse, als auch das Datum benötigt werden. Diese Wörter könnten nicht gleichzeitig im Cache enthalten sein, so dass die Trefferrate durch das dauernde Auswechseln der Speicherstellen sehr leiden würde. Daher stehen bei der Methode des assoziativen Zweiwege-Caches zwei Möglichkeiten zur Verfügung, um ein Wort einzuordnen. Dieses Verfahren stellt eine Verbesserung des Direct Mapping dar, weil für jedes Wort zwei Speicherstellen reserviert sind, d.h. es existieren zwei Möglichkeiten der Einordnung. Auf dieselbe Art lassen sich auch Vierwege-Caches realisieren.
Abgesehen von Leseoperationen, sind selbstverständlich auch Schreiboperationen von großer Wichtigkeit. Eine Möglichkeit besteht darin, die Schreiboperationen durch den Cache durchzuschreiben, also jede Information sowohl im Cache, also auch im Hauptspeicher abzulegen. Dieses Write Through-Verfahren bietet nämlich die äußerst wichtige Datenkohärenz; die Daten im Cache und im Hauptspeicher sind zu jedem Zeitpunkt identisch. Das Problem dabei ist jedoch, dass der Cache-Speicher dadurch zu keiner Beschleunigung des Systems führt.
Eine Alternative findet sich im Copy-Back-Verfahren, wo die Daten vorerst nur in den Cache-Speicher geschrieben werden und erst, wenn entweder die Daten durch ein Cache Miss and den Hauptspeicher zurückgegeben werden oder ein anderer Prozessor auf diese Daten zugreifen möchte.
Die Zusammenführung dieser beiden Vorteile, nämlich Datenkohärenz und Beschleunigung der Schreiboperationen, findet sich in der Buffer-Write-Through-Methode. Dabei wird das zu schreibende Datum gleichzeitig in den Cache und in einen zweiten schnellen Zwischenspeicher geschrieben (Pufferspeicher, engl. Buffer). Während der Prozessor schon weiterarbeitet, werden nun die Daten von zweiten Speicher in den langsamen Hauptspeicher übertragen.
Des weiteren seien noch Split-Caches erwähnt, welche getrennte Daten- und Instruktionscaches besitzen.
On-Chip-Caches sind direkt auf dem Prozessorchip integriert, haben aber aus Platzgründen nur sehr beschränkte Kapazitäten (typischerweise 32 KB). Die Zugriffszeiten sind allerdings ähnlich schnell wie die der Prozessorregister. Oft wird dies dadurch entschärft, dass ein On-Chip-Caches aus einem first level cache und einem second level cache besteht, wobei letzterer zwischen FLC und HSP geschalten ist.

4.2.3 Direct Memory Access (DMA)

Dieses Konzept ermöglicht eine Beschleunigung der Kommunikation der Prozessors mit den meist wesentlich langsameren peripheren Geräten. Dabei wird ein DMA-Controller (DMAC) eingesetzt, der dem Prozessor solange den Datenbus entzieht, bis das jeweilige Gerät den Datentransfer abgeschlossen hat. Dieser besteht aus folgenden Schritten:

1. Der Prozessor gibt dem DMAC das Ziel und die Größe der zu übertragenden Daten an
2. Der DMAC fordert vom entsprechenden Gerät die Daten an und wartet auf den Transfer
3. Nach dem Ende der Übertragung meldet der DMAC dies dem Prozessor

4.2.4 Controller und Co-Prozessoren

Controller sind Prozessoren, die den Prozessor, hauptsächliche bezüglich der Kommunikation mit I/O-Geräten, entlasten sollen. Sie treten in verschieden Aufgabenbreichen auf:

Externspeicher: Zur Kommunikation mit Hardisks, Floppys, Tapes oder optischen Platten

Graphik I/O: Zur Generierung von Signalen zur Bildschirmkontrolle und zum Beispiel eines Signals zur Steuerung des Cursors

Serial I/O: Zur Anbindung der CPU an die Umwelt über die standardisierte V.24-Schnittstelle (RS232-Standard)

Netzwerke: Auch hier existieren Spezialprozessoren zur Entlastung der CPU

Co-Prozessoren übernehmen diverse Spezialfunktionen der CPU und führen dadurch ihrerseits zu einer Entlastung selbiger.

Mathematik-Co-Prozessoren: Übernimmt zum Beispiel die Fließkommaberechnungen

Graphik-Co-Prozessoren: Entlasten die CPU von komplexen graphischen Berechnungen

Signalprozessoren: Da eine CPU nur ein Signal von wenigen Kilohertz verarbeiten kann sind Signalprozessoren grundlegend für die exorbitant rechenintensive Verarbeitung und Analyse von analogen Signalen

Mutlimediaprozessoren: Diese Form der Co-Prozessoren stellen eine Weiterentwicklung der Graphik-Controller dar. Sie besitzen zumeist einen Display-Processor sowie einen Pixel-Processor. Letzterer übernimmt Aufgaben der Komprimierung und Dekomprimierung verschiedenster Dateiformate.

4.2.5 Interconnection

Zur Interconnection stehen derzeit ausschließlich Bussysteme zur Verfügung. Hierbei lassen sich zwei Arten unterscheiden:

Paralleler Bus: Jedes Bit erhält eine eigene Leitung für die Übertragung

Serieller Bus: Die Bits werden Stück für Stück nacheinander übertragen

Folgende Standards für Bussystem fanden Verbreitung:

- ISA (Industrial Standard Architecture)
- EISA (Extended ISA)
- MIRCOCHANNEL (IBM Standard)

- Local Bus
- PCI Bus (Peripheral Component Interconnect Bus)
- SCSI (Small Computer System Interface)
- PCMCIA (PC Memory Card Int'l Association)

4.3 Periphere Geräte

Obwohl die Grundelemente eines PCs in den bisherigen Abschnitten bereits aufgebaut wurden, so fehlen dem Gerät noch die offensichtlichsten Bestandteile, wie zB Monitor, Tastatur, Maus, Drucker, etc. Diese fallen in das Gebiet der peripheren Geräte.

4.3.1 Externspeicher

Externspeicher gliedern sich in unterster (bzw. langsamster) Ebene in die Hierarchie der Speicher Bauteile. Sie finden sich in verschiedene Ausführungen, als magnetische Speicher, als optische Speicher und als holographische Speicher.

Magnetische Speicher Zur Speicherung von Information werden die beiden Magnetisierungsrichtungen magnetischer Werkstoffe benutzt. Ein Schreib-Lesekopf erzeugt auf dem Medium ein permanentes Magnetfeld gewisse Ausrichtung, welches im Lesevorgang, ohne Veränderung am Datenbestand, wieder ausgelesen werden kann. Zu ihnen zählen:

- Magnetbandspeicher
- Magentplattenspeicher (zB Festplatten)
- Disketten (zB Floppydisk)

Optische Speicher Optische Speicher werden entweder durch ein rein optisches Verfahren oder durch ein Verbundverfahren beschrieben, welches die optische und magnetische Aufzeichnung verknüpft. Die wichtigsten Vertreter sind:

- CD-ROM (Compact Disk - Read Only Memory)
- DVD (Digital Versatile Disk bzw. Digital Video Disk)
- Löschbar optische Platten (Abwandlungen von CDs bzw. DVDs)

4.3.2 Dialoggeräte

Dialoggeräte dienen zu Kommunikation zwischen Anwender und Maschine, sowie zur Dauerhaften Ausgabe von Daten und werden auch als Mensch-Maschine-Schnittstelle (Man Machine Interface, MMI) bezeichnet. Die Ergonomie beschäftigt sich mit der sinnvollen Ausgestaltung eben dieser. Handelt es sich bei der Maschine speziell um einen Computer und kommuniziert der Mensch interaktiv mit dem Rechner, so spricht man von einem Human Interface (HI). Die wichtigsten Geräte hierbei sind: Tastatur, Touchscreen, Joystick, Maus, Trackball, Touchpad, Digitizer (hand-held puck kann auf Tablett direkt Koordinaten ansprechen), Belegleser (zum

Lesen von zB Strichcodes, vgl. EAN), Klarschriftleser (OCR-A, OCR-B-Schriften), Markierungsleser (erkennt bist zu 1000 Markierungen auf speziellen Formblättern), Scanner, Bildschirme (Röhrenmonitor, Liquid Crystal Display, Thin-Film-Transistor-Bildschirm), Nadeldrucker, Tintenstrahldrucker, Laserdrucker, Plotter, Vernetzung

5 Prozesse

Früher wurden administrative Aufgaben eines Betriebssystems, was die Ausführung anbelangt, stark von den Programmen der Benutzer abgegrenzt. Heutzutage wird das alles in einen Topf geworfen und per Multitasking scheinbar gleichzeitig ausgeführt.
Heute schafft das Betriebssystem eine Virtuelle Maschine. „Klienten" des OS sind nicht mehr die physischen Benutzer selbst sondern die Prozesse, die die OS-Funktionen durch Betriebssystemaufrufe (system calls) nutzen.
Prozess = Programm in Ausführung
Programm = Folge von Anweisungen, die auf der vom OS realisierten VM ausgeführt werden
Um den Zustand eines Prozesses zu beschreiben, müssen der aktuelle Befehl und das ganze Umfeld (Variablen, benutzte Resourcen) mit angegeben werden. Zusätzlich sind noch Infos des OS über die Prozessorregister und Ausführungspriorität vonnöten. Dies alles zusammen wird auch *Prozess-Image* genannt. (Abb.6.1)

5.1 Threads

Mehrere voneinander getrennte Prozesse bedeuten einen großen Aufwand für das OS, weshalb man sogenannte Threads (*Lightweight Process*) eingeführt hat. Threads bestehen nur aus dem Registersatz des Prozessors, den zugehörigen lokalen Variablen und dem zugehörigen Program Counter. Threads sind soquasi eine logische Unterteilung von Prozessen (siehe Abb. 6.2), die Prozesse sind nur mehr Container für mehrere Threads. Die Aufteilung eines Prozesses/Programms in Threads wird nicht vom OS, sondern vom Programmierer vorgenommen. Man kann Threads mit Subroutinenn eines Programms, die gleichzeitig ausgeführt werden vergleichen.
Einige Eigenschaften von Threads:

- Threads laufen parallel ab (auf mehreren Prozessoren)
- Threads agieren im Adressraum *eines* Prozesses → Vollzugriff auf sämtliche Ressourcen des Prozesses
- Kommunikation zwischen den Threads verläuft durch globale Variablen des Prozesses
- Die Funktionen eines Threads werden über ein Thread-Interface angeboten

Es existieren keine Schutzmechanismen zwischen den Threads, was durch möglichen gleichzeitigen Zugriff auf Resourcen zu einer erhöhten Absturzgefahr führt. Auch die Verwedung nicht reentranter (für parallelen Zugriff ausgelegter) Bibliotheken führt zu Problemen.
Es gibt zwei Ansätze der Thread-Integration in das OS:

Thread-Funktionen außerhalb des OS: Sammlung von Funktionen/Unterprogrammen außerhalb des OS

Threads als integraler Bestandteil des OS: Die für den Einsatz der Threads notwendigen Funktionen sind auf system calls des OS abgebildet.

Besitzt ein OS einen Multi-Threaded-Kernel, so ist das OS an sich in mehrere Threads aufgeteilt.

5.2 Objekte

In modernen OS wird nahezu jede Resource eines Computersystems als Objekt betrachtet (Code, Daten, Tastaturen, Threads, etc.). Die Verwendung der Objekte erfolgt nur über spezielle Zugriffsoperationen, die oft in Form eines Managers für eine ganze Klasse gleichartiger Objekte zusammengefasst sind. So ist eine genaue Überwachung und notfalls auch Verhinderung der Zugriffe auf Objekte möglich.
Ein Beispiel für Objekte sind Dateien:
Dateien (Files) sind eine von ihrer Darstellung unabhängige Folge von Daten (Elementen). Es gibt:

unstrukturierte Files: eine Lose Folge von Elementen einfacher Datentypen

strukturierte Files: die Elemente sind Records, die wiederum aus mehreren Datentypen zusammengesetzt sind

Man kann eine Datei wie ein Array sehen: per Index wird auf die Records zugegeriffen.
Weiters kann man Dateien über den Zugriff in Random Acess Files und sequentielle files unterteilen. Zur Manipulation von Dateien gibt es entsprechende System Calls: read, write, seek, execute, close, curpos
Auch der Drucker, die Tastatur und andere Devices werden als spezielles File-Objekt angesprochen.
Jeder Prozess hat ein abstraktes Standard Input- und Output-File namens stdin und stdout, an das zur Laufzeit echte Files gebunden werden. Unter der *Device-Unabhängigkeit* versteht man, dass die Devices auf eine einheitliche Art und Weise angesprochen werden, beispielsweise über stdin und stdout.

5.3 Parallelität

Das OS muss virtuelles n-Prozessoren-System schaffen, auf dem mehrere Prozesse quasi gleichzeitig ausgeführt werden. Die Zuteilung der Prozesse zu den endlich vielen Prozessoren wird *Prozess Scheduling* (bzw. untergeordnet: Thread Scheduling) genannt. Task ist gleichbedeutend mit Prozess. Der Grund für die Ausnutzung echter Parallelität liegt in der gewünschten Leistungssteigerung.
Unter einem *Deadlock* versteht man die Blockierung des Systems. Veranschaulichen kann man einen Deadlock durch Dijktras *Dining Philosophers Problem*: fünf Philosophen wollen mit fünf Gabeln essen, wobei jeder Philosoph zwei Gabeln benötigt um zu essen. Nimmt am Anfang jeder eine Gabel, so verhungern alle, da ja keiner eine zweite Gabel zur Verfügung hat (siehe Abb).
Unter *Pipelining* versteht man die überlappende Ausführung mehrerer Befehle.
Die parallele Auführung von Teilen eines Gesamtsystems heißt *explizit*, wenn sie bei der Programmierung vorgesehen werden muss, und *implizit*, wenn sie nicht extra vorgesehen werden muss sondern automatisch gefunden und ausgenutzt wird.

5.4 Prozesshierarchien

Eine Möglichkeit ist die Darstellung als Prozessbaum: ein Prozess erzeugt Unterprozesse (Kinder), etc. Logisch zusammenhängende Prozesse werden als *Job* (ein Ausschnitt in der Prozesshierarchie) bezeichnet. Letzterer kann einheitlich durch seinen Parent-Prozess gesteuert, z.B. durch ein Signal terminiert werden. Ein Parent-Prozess „vererbt“ seine Umgebung an seine Kinder: manche Teile der Umgebung (z.B: Speicherbereiche) werden kopiert, manche (z.B: Datei) direkt weiterverwendet. Die Beziehungen der einzelnen Prozesse werden, ungleich Threads, direkt vom OS verwaltet. Das *Prozess-Management* eines OS stellt Mechanismen zur Erzeugung, Kontrolle und Termination von Prozessen bereit.

5.5 Prozesszustände

Jeder Prozess befindet sich zu einem bestimmten Zeitpunkt in einem bestimmten Zustand (Abb. 6.8)

created: das Prozess-Image wird zusammengestellt

ready: der Prozess ist bereit einem Prozessor zur Ausführung zugeteilt zu werden

running: der Prozess befindet sich in Ausführung

blocked: der Eintritt eines externen Ereignisses wird erwartet - der Prozess kann sich nur selbst in diesen Zustand versetzen und wird durch das externe Ereignis „aufgeweckt“

suspended-ready: das OS versetzt einen Prozess von running oder ready in einen Tiefschlaf

suspended-blocked: das OS versetzt einen Prozess von blocked in einen Tiefschlaf

dead: der Prozess ist kurz vor der Termination, sein Prozess-Image kann noch ausgewertet werden

Prozesse können entweder durch Eigeninitiative oder „von außen“ beendet werden, wobei letztere Variante Probleme aufwerfen kann. *Atomic Actions* lassen sich nicht vor Beendigung ihrer Codesequenz killen.
Zur Verwaltung der Prozesse legt ein OS für jeden Prozess einen *Prozess-Deskriptor* folgenden Inhalts an:

- Identifikation durch Prozess-ID
- aktueller Prozesszustand
- Priorität des Prozesses
- Inhalte der relevanten Prozessor-Register (Context)
- Besitzer des Prozesses
- Zugriffsrechte
- Verweise auf zugehörige Daten / Programm / Objekte
- sofern im Zustand blocked: das deblockierende Ereignis

Das Prozess-Management verwaltet nun Listen der möglichen Zustände, in die die Prozesse eingetragen werden.
Unter *Context Switch* versteht man das Wechseln des akutell auf einem Prozessor laufenden Prozesses (genaugenommen des Context), die *Context Switch Time* ist ergo die dazu benötigte Zeit (zwischen 10 und 100 Millisekunden). Das Aktivieren eines neuen Prozesses auf einem Prozessor wird *Dispatching* genannt.
In Thread-basierten OS gelten für Threads praktisch die gleichen Eigenschaften wie für Prozesse.

5.6 Scheduling

Generell wird die Verteilung der Aufgaben auf die tatsächlich vorhandenen Resourcen als Scheduling bezeichnet. Scheduling läuft oft in mehreren Ebenen ab (Job-Scheduling → Prozess-Scheduling → Thread-Scheduling).

5.6.1 Prozess-Scheduling

Jeder Prozess der ready-Liste muss für kurze Zeit running werden. Erfolgt der Wechsel in sehr kurzen Abständen, so entsteht die Illusion einer parallelen Abarbeitung aller Prozesse.
An das Scheduling werden eine Reihe von Forderungen gestellt:

- **Fairness**
- **Effizienz** (optimale Auslastung des Prozessors)
- hoher **Durchsatz**
- geringe **Anwortzeiten**
- ungefähr 80 % **CPU-Auslastung** (=Zeit der laufenden Prozesse / CPU-Betriebszeit)

Nun einige Entscheidungsverfahren für das Scheduling:

First Come First Serve (FCFS): Wer zuerst kommt, mahlt zuerst. Der Prozessor wird einem Prozess nur entzogen, wenn letzterer terminiert oder blocked wird.

Round Robin Scheduling (RRS): Jeder ready-Prozess bekommt für ein kurzes Zeitintervall einen Prozessor zugeteilt. Ist sein Zeitintervall abgelaufen, so wird er zuunterst in die ready-Liste eingetragen.

Static Priority Scheduling (SPS): Es werden immer die Prozesse höchster Priorität running gesetzt.

Dynamic Priority Scheduling (DPS): DPS funktioniert wie SPS, abgesehen davon, dass die Prioritäten der Prozesse dynamisch während der Laufzeit durch das OS geändert werden können.

Shortest Job First: Die Prozesse werden nach ihrer Abfertigungsdauer in eine Queue gereiht.

FCFS & Priority Scheduling: Für jede Prioritätsstufe gibt es eine eigene Queue, in der FCFS gilt.

5.6.2 Thread-Scheduling

Das Thread Scheduling ist dem Prozess-Scheduling hierarchisch untergeordnet (siehe Abb.6.11). Muss der Scheduler zeitkritische Aufgaben erledigen, so agiert das Thread-Scheduling prozessübergreifend, d.h. Threads mit höherer Priorität aus einem anderen Prozess werden running gesetzt, obwohl der aktuelle Prozess noch über Threads im ready-Zustand verfügt.

5.6.3 Job-Scheduling

Ziel des Job-Scheduling ist es, dem Prozess-Scheduler eine gut bewältigbare Arbeit zu überlassen. Es gilt das Gleiche wie für das Prozess-Scheduling, nun sind lediglich ganze Jobs betroffen.

6 Speicherverwaltung

Code-Segmente sind Speicherabschnitte, die ausführbaren Maschinencode enthalten. Daten-Segmente sind Speicherabschnitte, die beispielsweise Programmvariablen beinhalten. Weiters wird zwischen virtuellen und physikalischen Speicheradressen unterschieden. Die Abbildung von virtuellen auf physikalische Adressen wird *Binding* genannt.
Aufgabe der Speicherverwaltung ist neben der Adresszuordnung auch die Memory-Protection: der Speicherbereich der einzelnen Prozesse darf nicht von „außen“ (durch andere Prozesse) verletzt werden. Schutz vor „Selbstzerstörung“ der Prozesse ist nur schwer zu realisieren.

6.1 Virtuelle Adresszuordnung

Die Zuordnung virtueller Adressen erfolgt beim Compilieren eines Programms: den einzelnen Subroutinen und Variablen werden virtuelle Adressen zugeordnet, an denen sie anzutreffen sind. Wird ein Programm modularisiert (also in mehrere Teile zerlegt) compiliert, so wird zuerst ein Zwischencode (*Relocateable Object Code*) erzeugt, in dem die Zuordnung der Adressen zu den Variablen und Befehlen noch veränderbar bzw. noch gar nicht erfolgt ist. Die einzelnen Teile wissen ja voneinander noch nicht, wie die Adressen in einem anderen Modul zugeordnet sind. Das Zusammenfügen der einzelnen vorcompilierten Module und das endgültige Zuweisen der Adressen erledigt dann der sogenannte *Linker*. Gelinkt werden kann entweder *statisch* gleich nach dem compilieren oder *dynamisch* zur Laufzeit (*Runtime Linker*). Statisches Linken mag Laufzeit sparen, doch muss nach der Änderung eines der Module das gesamte Programm neu gelinkt werden. Ein weiterer Vorteil des dynamischen Linkens ist die Vermeidung von Speicherplatzverschwendung häufig (in mehreren Programmen) oder selten gebrauchter Module.
Bisher wurde stets ein linearer, eindimensionaler Adressraum betrachtet. Blähen sich allerdings allerdings Segmente zur Laufzeit auf, so sind Überschneidungen mit dahinter liegenden Segmenten unvermeidlich. Abhilfe schaffen zweidimensionale Adressräume, bei denen ein Segment (von nun an „äußeres“ Segment genannt) in mehrere virtuelle Adressräume (von nun an „innere“ Segmente genannt) unterteilt wird. Eine zweidimensionale Adresse besteht dann aus der Segment-Nummer und einer Adresse, genauer gesagt eines Offsets, innerhalb des Segmentes.

6.2 Physikalische Adresszuordnung

Aufgrund der eingeschränkten Größe des Hauptspeichers werden die Prozess-Images größtenteils auf externen Speichern abgelegt und erst beim Dispatching in den Hauptspeicher verschoben (*speicherresident*).

6.2.1 Swapping (im Buch durchlesen)

Beim Swapping werden die Prozess-Images als Ganzes zwischen dem externen und dem internen Speicher bewegt. Wird ein nicht residenter Prozess einem Prozessor zugewiesen, so erfolgt das *Roll-In* (Laden des Prozessimages) in eine freie Partition des Hauptspeichers. Man unterscheidet zwischen einer fixen und einer variablen Partitionierung des Hauptspeichers bezüglich Anzahl und Größe der Partitionen. Das Auslagern eines Images auf den externen Speicher wird *Roll-Out* genannt.
Die Memory Protection kann einerseits durch ein Upper und ein Lower Bound Register, außerhalb derer keine Speicherzugriffe möglich sind erfolgen. Eine andere Lösung sind *Storage Keys*, bei denen Speicherbereiche durch Schlüssel ihren Prozessen zugeordnet werden und der Prozessor vor jedem Speicherzugriff prüft, ob der laufende Prozess in diesen Speicherbereich schreiben darf.
Vorteile des Swappings ist die Einfachheit der Implementierung und der geringe Hardwarebedarf. Nachteile sind die Speicherverschwendung durch Fragmention bei der Partitionierung, die Einschränkung der maximalen Größe der virtuellen Adressräume und das langwierige vollständige Herumkopieren der Prozessimages.

6.2.2 Paging

Um jedem Prozess seinen gesamten virtuellen Adressraum auch physikalisch zur Verfügung zu stellen, werden sowohl der virtuelle Adressraum, als auch der physikalische Speicher in gleich große Blöcke, genannt *Pages* und *Page Frames* unterteilt. Speicherresident werden allerdings immer nur jene Pages gehalten, die zu einem gewissen Zeitpunkt auch tatsächlich zur Ausführung benötigt werden (auch *Working Set* genannt). Ausgenutzt wird dabei die Tatsache der zeitlichen und örtlichen Lokalität: eine erst kürzlich referenzierte Adresse wird mit hoher Wahrscheinlichkeit bald erneut und eine benachbarte Adresse ebenfalls sehr wahrscheinlich bald aufgerufen.
Ist der Hauptspeicher für alle Working Sets der Prozesse, so müssen Pages ständig vom externen Speicher geladen werden (*Trashing*). Versucht ein Prozess auf eine nicht residente Page zuzugreifen, so ist ein *Page Fault* die Folge und das OS muss eine Page nachladen. Die *Page Fault Frequency* lässt auch die ungefähre Größe des Working Sets berechnen. Soll nun eine Page nachgeladen werden, obwohl alle Page-Frames besetzt sind, muss ein Page-Frame „entladen“ werden. Zur Bestimmung welcher Page-Frame hierbei herangezogen wird, gibt es mehrere Verfahren:

First In First Out: Jede Page bekommt zum Zeitpunkt des Ladens einen Zeitstempel, bei einem Page-Fault wird immer die „älteste“ Page ersetzt.

Last Recently Used: Es wird jene Page ersetzt, die am längsten nicht benutzt wurde.

Least Frequently Used: Jene Page, die am seltensten benutzt wurde wird ersetzt.

Not Used Recently: Jede Page wird gekennzeichnet, sofern einmal referenziert, diese Kennzeichnung erlischt nach einem gewissen Zeitintervall wieder. Ungekennzeichnete Pages werden ersetzt.

Es gibt zwei Bereiche, aus denen die zu ersetzenden Pages-Frames ausgewählt werden: lokal aus dem Speicherbereich eines Prozesses, oder global. Weiters wird zwischen dem bisher besprochenen Demand-Paging - Pages werden bei einem Page-Fault auf Bedarf nachgeladen - und dem Anticipate Paging - hier wird versucht zu erahnen, welche Pages bald benötigt werden.
, ist die Memory-Protection bereits implizit vorhanden.
Als Vorteile des Paging sind die großen virtuellen Adressräume und die Memory Protection - da ein Prozess nur Pages referenzieren kann, die seinem Prozess-Image angehören ist sie bereits implizit vorhanden - zu nennen. Nachteil sind etwaige Timing-Probleme durch das Paging.

6.2.3 Segmentierung

Segmentierung ist ein Verfahren, das sich auch für zweidimensionale virtuelle Adressräume eignet. Unterschieden werden zwei Arten: die reine Segmentierung, die segmentweisem Swapping entspricht, und die Segmentierung mit Paging.
Bei der reinen Segmentierung wird in einer Segment Table jeder logischen Segmentnummer (der erste Teil der virtuellen Adresse) eine physikalische Speicheradresse und Länge des Speicherabschnittes zugewiesen. Um die genaue physikalische Adresse eines inneren Segments zu erfahren, wird der zweite Teil der virtuellen Adresse verarbeitet: der Offset wird lediglich zu der physikalischen Adresse des äußeren Segments hinzuaddiert.
Bei der Segmentierung mit Paging verweist jede Segmentnummer der Segment Table auf eine weitere Paging Table. Um die physikalische Adresse eines inneren Segments zu erfahren, wird der Offset aus der virtuellen Adresse ausgewertet: er gibt nun den genauen Eintrag in der Page Table sowie den Offset in der Page selbst an (siehe Abb. 7.7).
Wird jedem Segment ein einzelnes Objekt, im diesem Falle eine Variable, zugewiesen, so kann das OS die Typsicherheit der Variablen dynamisch überprüfen (einer Integer-Variablen kann kein String zugewiesen werden). Diese Technik wird *Capability Based Adressing* genannt.
Segmentierung mit Paging ist das eleganteste Speicherverwaltungskonzept.

7 Interprozess-Kommunikation

Wird eine Aufgabe in mehrere Prozesse unterteilt, so müssen diese Prozesse miteinander kommunizieren, sprich Nachrichten austauschen können.

7.1 Server-Prozesse

Server-Prozesse stellen Diensleistungen zur Verfügung, die durch andere Prozesse durch Anfragen (*Service Requests*) und Rückmeldungen genutzt werden können. Trifft ein neuer Service Request ein, bevor eine alte Anfrage abgearbeitet werden konnte, so wird er in eine Queue eingeordnet. Eine andere Möglichkeit wäre es den Prozess zu unterteilen: ein *Dispatcher*-Thread weist mehreren *Worker*-Threads die Arbeit zu.
Race-Conditions liegen vor, wenn das Ergebnis der parallelen Abarbeitung eines Programms

von der relativen „Geschwindigkeit“ der beteiligten Prozesse abhängig ist: wird beispielsweise ein running-Prozess vor Beendigung seiner Arbeit ready gesetzt und der nun ausgeführte Prozess „pfuscht“ ihm in seine noch nicht fertiggestellte Arbeit (schreibt beispielsweise in eine noch offene Datei), so führt dies klarerweise zu Problemen.

7.2 Synchrone Methoden

Bei Synchronen Methoden muss sich der Empfänger-Prozess seine Post (die Nachrichten) selbst vom Sender holen.

7.2.1 Semaphore

Eine Semaphor ist ein vom OS anforderbares Objekt, bestehend aus einem auf 0 initialisierten Zähler, einer zunächst leeren Liste für Prozess-IDs und zwei Methoden namens wait (P) und signal (V)[1]. Der Zähler des Semaphor legt fest, wieviele kritische Prozesse existieren dürfen. Die Funktionsweise erklärt sich aus der Arbeit der Methoden: wird P aufgerufen, so wird der Zähler dekrementiert und, sofern der Zähler unter Null fällt, die Prozess-ID des aufrufenden Prozesses in die Liste eingetragen sowie der aufrufende Prozess blocked gesetzt. Wird V aufgerufen, so wird der Zähler inkrementiert und die Prozesse aus der Queue, sofern der Zähler wieder auf größer gleich Null steigt, wieder auf running gesetzt.
Semaphore eignen sich zur Verhinderung von Race Conditions.

7.2.2 Message Passing

Über einen *Exchange* genanntes Objekt werden Daten zwischen Prozessen ausgetauscht. Ein Exchange besteht aus zwei Listen, einer für Messages und einer für Prozess-IDs. An das Exchange-Objekt gesendete Nachrichten werden in der FIFO-Message-Liste gespeichert und können von dort auch wieder abgeholt werden. Ist die Message-Liste gerade leer, obwohl ein Prozess eine Nachricht holen wollte, so wird dieser Prozess in die FIFO-Prozess-Liste eingetragen und blocked gesetzt. Trifft nun eine Nachricht ein, so wird ein Prozess aus der Prozess-Liste entfernt und die Nachricht an ihn übergeben.

7.2.3 Höhere Mechanismen

In Ada wird beispielsweise das *Rendezvous-Konzept* verwendet, bei dem Empfänger und Sender aufeinander warten müssen.
Unter einem *Monitor* versteht man einen Programmabschnitt, der nur von einem Prozess verwendet werden kann. Es empfiehlt sich nun, kritische Abschnitte in so einen Monitor zu stellen. Dieser Ansatz wird auch von Ada in Form von *Protected Objects* verwendet.

7.3 Asynchrone Methoden

Die Ankunft einer asynchronen Nachricht unterbricht den Empfänger in seiner aktuellen Tätigkeit. Die asynchronen Signale sind direkt vergleichbar mit Interrupts.

[1] Die Bezeichnungen P und V stehen für „Proberen“ und „Verhogen“, der Erfinder der Semaphore, Dijkstra war nämlich Holländer

8 Netzwerke

Durch die Vernetzung der Computer ist es unter anderem möglich, auf geographisch weit entfernte Daten schnell zuzugreifen bzw. mit weit entfernten Personen zu kommunizieren, Resourcen gemeinsam zu nutzen und ausfallsichere Systeme zu schaffen (andere Computer im Netzwerk übernehmen die Funktion eines ausgefallenen).

8.1 Struktur

Ein Netzwerk besteht aus mehreren Computern (*Hosts*), die durch ein *Communication Subnet* verbunden sind. Man unterscheidet zwischen:

- Local Area Network (bis etwa 1km)
- Metropolitian Area Networks (bis 50km)
- Wide Area Networks (z.B. das Internet)

Ein Subnet besteht aus Übertragungsstrecken (*Circuits*) und Schaltstellen (*Interface Message Procerors*), die Übertragungsstrecken verbinden.
Es gibt zwei Techniken der Informationsübertragung:

Circuit Switching: Vor der Übertragung wir ein Übertragungskanal fix reserviert, über den anschließend kommuniziert wird.

Packet Switching: Die zu übertragenden Daten werden in Pakete aufgeteilt, die dann von IMP zu IMP weitergereicht werden. Übertragungsstrecken werden somit nur dann reserviert, wenn tatsächlich Daten übertragen werden.

Ein Subnet kann auf zwei verschiedene Arten aufgebaut werden:

Point to Point: Eine Überttragungsstrecke verbindet zwei IMPs, jeder IMP kann Anfangs- und Endpunkt mehrerer Übertragungsstrecken sein. Bei weitläufigen Übertragungen über mehrere IMPs leiten die Zwische-IMPs die Pakete weiter.

Broadcast: Eine einzelne Übertragungsstrecke verbindet alle IMPs, es sind keine Zwischen-IMPs notwendig und Hosts werden mit IMPs gleichbedeutend. Diese Technik setzt auch das alte 10Base-2 Ethernet (das mit den Koaxialkabeln) ein. Die Kapazität eines Broadcast-Mediums kann durch Frequency Divison Multiplexing und Time Divison Multiplexing auf viele, logisch getrennte Übertragungskanäle aufgeteilt werden.

Sind zwei Netze nicht miteinander kompatibel, so werden sie durch *Gateways* verbunden.

8.2 Architekturen

Die notwendigen Mechanismen über die eine Maschine verfügen muss, um an einem Netzwerk teilhaben zu können, werden durch mehrere Schichten (*Layer*) implementiert. Kommunikation zwischen den Maschinen erfolgt dabei immer nur auf dem gleichen Layer über ein genau festgelegtes Protokoll. Innerhalb einer Maschine bieten Layer ihnen übergeordneten Layern Kommunikation-Services an und nutzen gleichzeitig die Services der ihnen untergeordneten Layer (siehe Abb. 9.2). Die tatsächliche Datenübertragung läuft dabei natürlich nur über den untersten Layer ab.

8.2.1 OSI Reference Model

Es folgt die Beschreibung der Layer des OSI Referenz Modells (siehe Abb.9.3) für Computer-Netzwerke.

Layer 1 - Physical Layer: Hier erfolgt die bitweise Übertragung auf physikalischen Übertragungsstrecken.

Layer 2 - Data Link Layer: Dieser Layer hat die Aufgabe, eine fehlerfreie Datenübertragung zu realisieren: die aus dem übergeordnete Network Layer kommenden Daten werden portioniert und mit einem Header, einem Trailer und einer CRC-Checksumme zur Fehlererkennung versehen. Das Ganze heißt dann *Frame*. Die korrekte Übertragung eines Frames wird mittels eines *Acknowledge-Frames* bestätigt. Die Aufgabe verfälschte oder verlorengegangene Frames unter Auswertung des Ack-Frames nochmals zu Senden wird *Error Control* genannt. Unter *Flow Control* versteht man Mechanismen, die ein „überfüttern“ eines langsamen Empfängers mit zu schnell gesendeten Frames verhindern.

Layer 3 - Network Layer: Dieser Layer ist für den eigentlichen Betrieb des Communication Subnets zuständig. Auf den IMPs gibt es nur die untersten drei Layer, dem nächsthöheren Transport Layer ermöglicht der Network Layer also bereits echte End-zu-End-Verbindungen. Hauptaufgabe des Network Layer ist das *Routing*, die Lösung des Problems, über welche IMPs das Datenpaket am effizientesten zu seinem Ziel gelangt. Auf dieser Ebene findet auch das *Accounting* statt - die Erfassung der von einem Benutzer „verbrauchten“ Übertragungskapazität. Protokoll ist z.B: IP.

Layer 4 - Transport Layer: Aufgabe dieses Layers ist die „Abschirmung“ der höheren Layer von den unteren, da diese den Netzbetreibern und nicht dem Benutzer unterliegen. Der Transport Layer bietet auch nochmals Mechanismen zur sicheren Datenübertragung. Im Falle hohen Datenaufkommens teilt er die Daten auf mehrere Verbindugnen auf. Protokoll ist z.B: TCP.

Layer 5 - Session Layer: Laut Professor Schildt weiß keiner so genau, was dieser Layer tun soll. Protokolle sind jedenfalls HTTP, FTP, etc.

Layer 6 - Presentation Layer: Dieser Layer befasst sich im Gegensatz zu den unteren Layern auch mit der Struktur der übertragenen Daten. Hier wird zumeist auch etwaige Verschlüsselung realisiert.

Layer 7 - Application Layer: Dieser Layer enthält die eigentlichen Anwendungen, wie Email-Services oder File-Server.

8.2.2 Fallbeispiele

Als einziges Beispiel sei Carrier Sense Multiple Access with Collision Detection (CMSA/CD), auch bekannt unter 10Base-2 Ethernet, genannt. Eine Station, die Nachrichten senden will, verfährt dabei nach folgenden Regeln: Wird eine Kollision erkannt, so sendet die erkennende Station ein Störsignal aus, um alle übrigen Stationen darüber zu informieren und bricht daraufhin die eigene Sendung ab. Dann wird für eine zufällige Zeitspannende pausiert und nach deren Ablauf die Sendung wiederholt.

9 Betriebsystem-Struktur

Dieses Kapitel widmet sich der generellen Frage, wie ein Betriebsystem eigentlich aufgebaut ist.

Eine allgemeine, einfache Darstellung für den schemenhaften Aufbau des Betriebssystems liefert das traditionelle Schichtmodell (s. Abb. 10.1), bestehend zumindest aus: *Interrupt Handling, Prozess-Management, Interprozess-Kommunikation, Speicherverwaltung, Resource-Management* und *Prozess-Interface*. Als sog. *Kernel* oder *Nucleus* werden die untersten Schichten bezeichnet, die über *Maschineninstruktionen* als Interface direkt die Hardware ansteuern.

Technisch gesehen ist ein Betriebssystem lediglich eine Kollektion von Programmen im Speicher, welche durch die zugrunde liegende Hardware zur Ausführung gebracht wird – über *System Calls* von laufenden Prozessen oder über *Hardware-Interrupts*. Diese Exekutionen laufen jeweils in der für sie zuständigen Betriebssystem-Schicht ab; wenn der Aufruf von übergeordneten Ebenen kommt, wird er entsprechend weitergeleitet und eventuelle Rückmeldungen gehen denselben Weg zum Auslöser wieder zurück.

Eine wichtige Aufgabe des Betriebssystems ist es jetzt, alle diese unter Umständen gleichzeitig auftretenden Ereignisse kontrolliert ablaufen zu lassen; einen etwaigen gegenseitigen Ausschluss (*Mutual Exclusion*) gilt es zu vermeiden.

9.1 System-Calls

Zugriff auf die Hardware gelingt das nur durch den Aufruf von entsprechenden System-Calls im *Prozess-Interface* des Betriebssystems. Dieses Prozess-Interface muss klarerweise *reentrant* sein (Prozesse laufen parallel und System-Calls treten daher auch gleichzeitig auf). Kommt es zu einem derartigen Aufruf, wird durch einen *Software-Interrupt*-Befehl (*Trap*) die Exekution des Prozessors unterbrochen und ähnlich wie bei den bekannten Interrupts der Hardware die wichtigsten Register „gerettet“ und das Programm an der Startadresse, welche im korrespondierenden *Trap-Vektor* enthalten ist, ausgeführt. Anstatt einen Software-Interrupt auszulösen könnte man auch gewöhnliche Sprungbefehle einsetzen, deren Verwendung aber für den jeweiligen Prozess einen erhöhten Verwaltungsaufwand bewirkt: Kenntnis der Anfangsadressen der System-Calls (der Maschinencode des Prozess-Interface muss im Programm eingebunden sein) und selbstständige Sicherung sämtlicher Registerinhalte. Erstere Variante bietet außerdem große Flexibilität auf Grund der Trap-Vektoren; ein aktueller Prozessor bietet dem Programmierer eine Vielzahl verschiedener Traps an, denen jeweils eigene Vektoren zugeordnet sind.

Dass die parallele Ausführung von System-Calls Probleme verursachen kann, soll exemplarisch anhand der Aufrufe `S_V` und `S_P` von Semaphoren verdeutlicht werden: Kommt es während der Ausführung eines `S_P`, nachdem der Counter dekrementiert und die Abfrage Counter ≥ 0 durchgeführt wurden, unmittelbar zu einem `S_V`-Call, wird der Counter inkrementiert und in weiterer Folge nun wegen Counter = 0 der (noch) nicht existierende Prozess der Prozess-Queue gelöscht und ihm ein WAKEUP-Signal geschickt; die Fortführung des ersteren `S_P`-Calls bewirkt, dass sich jetzt ein wartender Prozess in der Queue befindet, der Counter jedoch fälschlicherweise wieder bereits 0 ist.

Offensichtlich darf während dieser kritischen System-Calls keine Unterbrechung stattfinden, d.h. die Befehle müssen *atomic* (unteilbar) sein. Realisiert wird das entweder

1. durch ein Verbot der parallelen Ausführung während der kritischen Phase; parallele Befehle landen in einer Warteschlange (man spricht dann von *Serialized Actions*) oder

2. durch Einsatz spezieller Software- und/oder Hardwaremechanismen (z.B: die Semaphoren-Aktionen hardwareseitig als Maschinenbefehl ausführen oder verwenden reiner Software-Lösungen von *T. Dekker* oder *G. L. Peterson*).

9.2 Netzwerkintegration

Damit die Prozesse oder die InterProzess-Kommunikation netzwerktauglich werden, benötigt man Implementierungen für den Zugriff auf die im Kapitel 9 erwähnten Kommunikations-Protokolle und -Kanäle. Die Schwierigkeiten dabei sind u.A:

- das leider von keinen Systemprogrammierern entworfene *OSI Reference Model*, ist eher „telefonlastig“ anstatt „computergerecht“.

- eine konsequente Umsetzung und Einbindung der vielen Layer in das Betriebssystem ist durch den entstehenden Overhead sehr ineffizient; üblicherweise genügt es, die netzwerkrelevanten Funktionen im Transport Layer (Layer 4) anzusiedeln.

- die Netzwerk-Nutzung von Prozessen aus ist wegen der längeren Übertragungzeiten in WANs (z.B: 64 KBit/s) eher nur für LANs sinnvoll einzusetzen.

Diese Mechanismen sollten natürlich so abstrakt sein, dass der Anwender (Programmierer) keinen Unterschied sieht, ob sich die Prozesse nun auf dem gleichen oder verschiedenen Rechnern befinden. Die wichtigsten zwei Realisierungen stellen Erweiterungen zu den Konzepten aus Kapitel 8.2 *Semaphore* und *Exchanges* dar:

- Das Prinzip des so genannten *Message Passings* über Exchanges beruht darauf, dass zuerst über spezielle System-Calls eine Verbindung zu zwei auf verschiedenen Hosts befindlichen Exchanges hergestellt wird. `E_RECEIVE` arbeitet wie gewohnt und `E_SEND` liefert die Message direkt in die Queue des verbundenen Remote-Exchanges. Diese Methode hat sich ohne Zweifel stark am Transport Layer orientiert. Betriebssysteme wie das auf *Sockets* (entspricht den Exchanges) aufbauende *Unix-BSD* oder das sich ganz allgemein auf das *Transport Layer Interface* stützende *Unix System V* nützen diese Netzwerkanbindung.

- Eine viel flexiblere Variante des Informationsaustauschs zwischen Prozessen wird realisiert, wenn Objekte wie das Semaphore oder Exchange völlig *netzwerkglobal* gemacht werden, d.h. ein `S_P`-Call beispielsweise ist dadurch auch auf eine Remote-Semaphore möglich. Es werden dabei die *verteilte* und die *zentralisierte* Variante unterschieden. Für erstere sind ausgeklügelte Algorithmen (z.B: *Lamport's Algorithm* oder *Maekawa's Square Root Algorithm*) für den Abgleich der lokalen „Kopie“ des Hosts mit einer globalen Semaphore notwendig. Bei der zweiten Variante liegen die globalen Objekte lokal auf einem Host. Je nach dem wo sich das zu einer Semaphore gehörenden Objekt befindet, kommt es entweder zur lokalen Ausführung oder im anderen Fall zur Erstellung eines Remote-Prozesses (auch *Client Agents* genannt) auf dem spezifischen Host. Dieser Prozess exekutiert dann dort wie gewöhnlich den gewünschten Service-Call, schickt eventuelle Rückmeldungen zurück und terminiert anschließend.

Dieses Verfahren des *Remote Procedure Calls (RPC)* ist heute für verteilte Applikationen und vor allem für Client/Server-Anwendungen das häufigste gebrauchte Mittel. Im Programm sieht es einem normalen Funktionsaufruf sehr ähnlich. In der Ausführung hingegen werden die Funktionsparameter in eine Message verpackt und an den eventuell entfernten Server über eine Transport Connection geschickt. Dieser packt sie aus und ruft damit bei ihm lokale Funktion (Procedure) auf. Das Resultat dieser Funktion wird wiederum in eine Meldung verpackt und an den aufrufenden Prozess zurückgeschickt, dort ausgepackt und als Funktionswert (oder veränderte Parameter) zurückgegeben. Der aufrufende Prozess hat also eine entfernte Prozedur aufgerufen, woraus der Remote Procedure Call seinen Namen erhielt.
Eine nicht zu vernachlässigende Eigenschaft von RPC-Aufrufen ist die, dass ein aufrufender Client im `S_P`-Prozess solange die Programmausführung anhält, bis er eine Antwort der Prozedur vom Server erhalten hat.
Sinnvoll erweisen sich *Remote Procedure Calls* auch für eine „Auslagerung“ anderer System-Calls, z.B: die des Filesystems: damit lässt sich praktisch kein Unterschied im Zugriff zwischen Remote-Files oder lokalen Files feststellen. Stellt ein Computer sein Filesystem mehreren Clients zur Verfügung, dient dieser als so genannter *File-Server*, wobei zwei Arten unterschieden werden:

Stateful-File-Server führen über jedes offene File Buch, d.h. ein durch `F_OPEN` erzeugter Client Agent bleibt temporär aktiv bis die Datei explizit mit `F_CLOSE` geschlossen wird. Ein Nachteil stellen die langwierigen Wiederherstellungsmaßnahmen dar, falls es zu einem Absturz des Server kommt

Stateless-File-Server führen immer die Sequenz von `F_OPEN` und `F_CLOSE` aus. Zwischen diesen zwei Befehlen kann der Client Agent andere (File-)Operationen durchführen. Da pro RPC-Aufruf alle Files vorerst immer geöffnet werden müssen, ist als einziger Minuspunkt eine schlechtere Performance festzustellen.

9.3 Sicherheitsaspekte

Die Sicherheit im Betriebssystem und dessen Effizienz (Geschwindigkeit) stehen in einem indirekten Verhältnis zueinannder. Es muss ein vernünftiger Mittelweg gefunden werden, wo der auftretende *Overhead* noch vertretbar und ausreichend Sicherheit gewährleistet ist. Es gibt hierbei gewisse Grundregeln:

- Einen wirksamen Schutz stellen diverse Schichtenmodelle dar, die ringförmige Bereiche von höchst-priviligiertenr Prozessen (innen) bis hin zu niedrig-priviligierten (außen) definieren, wobei ein Übergang von äußeren Schichten in restriktivere innere durch Schutzmaßnahmen und Kontrollen gesichert sein muss.

- Ein (User-)Prozess darf von vorherein keine Möglichkeiten besitzen, Teile oder das gesamte System zu gefährden und im Falle von zu niedriger Berechtigungen die Funktionalität der Maschineninstruktionen dementsprechend einzuschränken.

- Wird direkter Zugriff auf Hardware mittels eines System-Calls benötigt, wird kurzfristig ein Ausnahmezustand mittels der *Trap-Befehle* aktiviert, um kontrolliert in die inneren Bereiche zu gelangen.

- Das Betriebssystem *Multics* war eines der ersten Systeme, welches von Grund auf dieses Schema konsequent einsetzte.

Für das Schichtenmodell besitzen aktuelle Prozessoren auch eigene Hardwareunterstützung: Prozessormodes von User-Mode zu System-Mode, in die ein Wechsel von seiten des Programms nur durch Traps möglich ist. moderne Betriebsysteme gehen dazu über, einen klein gehaltenen, stark abgeschotteten primären Kernel mit den elementaren Funktionen zum generellen Ansteuern der Hardware – wie Interrupt Handling, Prozess Management – bereitzustellen, der im Gegensatz zu den restlichen Programmen nur Mechanismen, aber keine selbstständigen Aufgaben besitzt (→ *Policy/Mechanism-Splitting)*. Dieser primäre Kernel stellt eine Art Server dar, den wichtige Betriebssystemkomponenten im sekundären Kernel oder noch weiter ausgelagerte Prozesse (z.B. das Filesystem) als Client-Prozess benutzen dürfen. Aktuelle Trends gehen außerdem dahin, dass die Hardware immer mehr Aufgaben des Kernels übernimmt, wodurch sehr kleine, aber flotte Kernels entstehen. Auch ein verstärkter Einsatz von RISC-Prozessoren ist in dieser Hinsicht praktisch: damit wird die Geschwindigkeit erhöht und doch die Portierbarkeit von Code beibehalten.

10 Resource-Management

... ist eine der umfangreichsten Aufgaben von Betriebssystemen. Es gestattet den (User-)Prozessen die verschiedensten Betriebsmittel komfortabel zu benützen.

10.1 Objektorientierung in Betriebssystemen

Dabei werden diese abstrahiert zu *Objekten*, in bestimmte Klassen eingeteilt und betriebsintern vom *Type Manager* verwaltet. Ein Ansprechen eines Gerätes läuft aus sicherheitsrelevanten Gründen nur indirekt über eine *Objekt-ID* und den jeweils vom Type Manager bereitgestellten *Zugriffsfunktionen*, unter Einhaltung der *Access-Rights*.

10.1.1 Protection

... im Type Management wird vom Betriebssystem als *Protection-Matrix* gespeichert und in *Protection Domains* groupiert (s. Abb. 11.1). Für diese Matrix sind zwei Speichermethoden gebräuchlich:

Acess Control List (ACL) – spaltenweise alle nicht-leeren Elemente, d.h. pro Objekt seine jeweiligen Rechte in den unterschiedlichen Protection Domains. Bei einem Objektzugriff werden die notwendigen Rechte für den Aufruf in dessen ACL mit der, durch den Prozess festgelegten, Protection Domain überprüft.

Capabilities – zeilenweise, d.h. demnach eine pro Protection Domain enstehende Liste an gewährten Rechten für die einzelnen Objekte, die jedem Programmprozess als *C-List* zugeordnet ist und dadurch seine Zugriffsmöglichkeiten für diverse Ressourcen festlegt.

In beiden Füllen kann die Matrix durch geeignete *Protection Calls* modifiziert werden. Für eine gute Systemsicherheit ist eine geeignete Rechteverteilung als auch ein angebrachtes Regelwerk ausschlaggebend.

10.1.2 Device-Unabhängigkeit

... wird durch all jene Maßnahmen bewirkt, die ein einheitliches Interface zu den unterschiedlichsten Objekten unterstützen. Die allgemeine Bestrebung ist, Objekte ähnlicher Art gleich zu behandeln – in der Objektorientierung dann *virtuelle Klasse* genannt. Für die Applikationen als auch für den Benutzer ist diese Abstraktion ein wahrer Segen: Ein Programm kann allgemein z.B. für eine Vielzahl unterschiedlicher Bildschirm-Controllern geschrieben werden oder der Benutzer kann Hardware austauschen, ohne dass es für seine Applikation hinderlich ist.

10.2 Resourcen-Klassen

In den nächsten drei Unterkapiteln wird ein kleiner Einblick in wichtige Gruppen wie Externspeicher, Drucker oder dem *Human Device Interface* gegeben.

10.2.1 Externspeicher

genauer gesagt, geht es zuerst um die logische Sichtweise von *Files* als Objekte und danach um deren Implementierung in Hardware.

Directories dienen der Organisation der Files im Filesystem, die bedingt durch ihre Größe selbst als (spezielles) File am Externspeicher untergebracht sind.

Directory Trees als *hierarchische Struktur* (s. Abb. 11.3) ermöglichen schnelles Suchen nach Files;

Pfadnamen werden durch die Position (= Folge von Directory-Namen von der Wurzel bis zum File-Objekt mit Trennzeichen (*Path Name Delimiter*) festgelegt. Entweder *absolut*, durch den volständigen Pfad ab Wurzel, oder *relativ* – indem ausgehend von einem *Current Directory* andere Directories angewählt werden.

. steht für (absoluter) Pfad des jeweiligen Directory selbst

.. mein hingegen das *Parent Directory* (wenn vorhanden), sonst die Wurzel selbst.

Weitere Besonderheiten von Directory-Trees:

- (*Symbolic*) *Links*: statt einem File mit (Benutzer-)Daten beinhalten sie einen Verweis auf andere Files.
- ein globaler Directory Tree kann auch komplette *Subtrees* anderer Externspeicher gemäß den physikalischen Gegebenheiten enthalten.
- *Mount-Fähigkeit* bezeichnet einen globalen Tree in dem in beliebigen Directories Externspeicher "*montiert*" werden.
- *Network Root Directory* ermöglicht die Einbindung von Filesystemen anderer Rechner äber Angabe des *Hostnamens*. (Beim *Network File System (NFS)*: Netzwerkfunktionalität mittels *Remote Mounts*)

File-Attribute sind Zusatzinformationen außerhalb der tatsächlichen Daten zum Klassifizieren und Charakterisieren von File-Objekten. Sie können auch bestimmen, ob nur sequentiell (*Sequential Access Method, SAM)* oder beliebig (*random access* mittels *DAM*) im File zugegriffen werden darf (oder sogar indiziert über *ISAM*).

Concurrency Control bedeutet die Koordination gleichzeitiger Zugriffe auf ein und dasselbe File, die vorwiegend darin besteht, dass eine *Mutual Exclusion* im kritischen Abschnitt verwendet wird; d.h. Locken ganzer File-Objekts oder von Teilen davon (*Records*) auf Prozessebene mittels System Calls. Erfolgt dies automatisch nach einem `F_OPEN`, heißt es *implizit*, kann der Lock-Zustand willkürlich gesetzt werden, handelt es sich um das *explizite File Locking*. File-Server oder speziell Datenbanken verwenden unterstützend *Transactions*, die sogar trotz eines Computerabsturzes keine Inkonsistenz der File-Objekte entstehen lassen, weil ihnen "Unteilbarkeit" versprochen wurde (erfolgreich oder gar nicht).

Device Driver kümmern sich um die hardware-abhängigen Details im Umgang mit *Disk-Controllern* und erstellen für alle Geräte ein einheitliches Interface mit einer *Standardblockgröße* und einheitlichen *Blocknummern*.
Der Device Driver hat darüberhinaus auch die Möglichkeit zur Optimierung (oft auch in der Hardware selbst implementiert, → "*intelligente*" Controller):

- *cachen* von Lese/Schreib-Operationen in einem RAM
- Minimierung der *Seek Times* durch *Disk Scheduling*-Algorithmen – aufeinanderfolgende E/A-Befehle entsprechend des Hardware-Layouts umordnen (z.B. von innen nach außen)

Disk-Management Für die Handhabung von Files auf einem Externspeicher gibt es zwei prinzipielle Methoden:

Swapping jedes File bekommt einen fixen zusammenhängenden Bereich an Blöcken zugeteilt. Diese Art ist zwar effizient weil garantiert sequentiell, der Größe der Datei ist aber eine obere Schranke gesetzt.

Sequentierung entspricht einem erweiterten Swapping: statt einem einzigen großen Block kommen mehrere zusammenhängender Segmente als Vielfaches von *Clustern* (= bestimmte Mindest-Blockanzahl) zum Einsatz, denen auch im Nachhinein auch zusätzliche Segmente angehängt werden können. Der *File Descriptor* eines jeden Files enthält als *Records* (Blocknummer, Blockanzahl) die Auflistung aller Segmente, aus denen das File besteht. Damit lässt sich die Directory-Struktur einfach konstruieren (s. Abb. 11.7)
Durch diese Art der Speicherung kann auch das *Bad-Blocks*-Problem beseitigt werden: vom Disk-Controller eventuell gemeldete defekte Blöcke bekommen einen Eintrag als "belegt" in die Segmentliste, verweisen aber nicht auf Daten.

Möglichkeiten zum Erhöhen der Performance des Disk-Managements (oft zur lasten der *Reability*):

- Verwendung diverser Cache-Strategien; vor allem durch *Anticipatory Fetch*, d.h. vorausschauendes Hereinholen eines oder mehrer Blöcke, die voraussichtlich als nächste benötigt werden. (Bei Verzicht auf *Write-Through* gehen im Falle eines Crashes die Cache-Inhalte verloren!)

- Verteilung der Segmente auf physikalisch verschiedene Datenträger (*Striping*)

10.2.2 Hardcopies

Zum Verstehen dessen Management ist neben Kenntnis der allgemeinen Funktionsweise nicht flüchtiger Medien (v.a. Drucker, s. Kapitel 5.3) das im vorigen Abschnitt besprochene File-Managing von Bedeutung: Drucker oder vergleichbare Ausgabegeräte werden vom System abstrakt als spezielle Files gesehen (mit `F_OPEN`, `F_WRITE`, `F_CLOSE`) die auch eine Art Beschreibung des Geräts (*Device Description*) enthält, z.B. 8 Datenbits, Even Parity, 2 Stoppbits bei seriell. Schreibt man Daten in dieses File, produziert das (nahezu transparent) einen Ausdruck am Drucker. Auf Prozessebene kommt das Konzept des *Spoolings* zum Einsatz, um eine eventuell lange Blockierung des Geräts zu umgehen (Druck-Jobs kommen gereiht in eine Spool-Queue, welche beizeiten vom *Printer-Server* zum Gerät geschickt wird)

Genauer betrachtet, kümmert sich der Printer-Server auch um die Formatierung, z.B. *Bannerpage* mit Datum/Uhrzeit, eventuell Kopf/Fußzeile pro Seite). Klassische (ältere) zeichenorientierte Drucker sind im Management einfacher als Geräte mit Grafik-Output: der Schriftsatz ist bereits vom Hersteller festgelegt (Größe, Abstände, Fontfamilie) und braucht nur noch als Zusatzinformation jedem Druck-Vorgang mitgegeben werden; für die graphische Variante sind geeignete Beschreibungssprachen wie z.B. *Initial Graphics Exchange Specification, IGS* oder *PDF/PS* von nöten. Der Prozess des Printer Spoolings durchläuft dann üblicherweise eine Serie von Filtern (Sprache, Text, Grafik) bis schließlich der eigentliche Server-Prozess den standarisierten Output als *DVI*-File (*Device Independent*-File) in der Spool-Queue vorfindet und als *Postscript* gewandelt an den Drucker sendet. Diese Vorgangsweise ist idealer weise über *Message Passing* auch netzwerktauglich.

10.2.3 Dialog-Geräte

Der *Desktop Approach* als neue E/A-Organisationsform war eine glorreiche Erfindung; damit lassen sich Unordnungen am Schreibtisch

> „[...] Bücher, Zeitungen, Kaffeetasse, Kugelschreiber usw. liegen neben/übereinander am Tisch herum; [...] „*Asterix und der Kupferkessel*“ ist gerade noch unter [...] dem Manuskript der „*Informatik*“ zu erkennen [...]“ [2]

bestens kopieren; wichtige Konzepte sind:

- *(Multi-)Windows*; jeder Prozess hat sein eigene Fenster
- *Windows-Manager*, z.B. *X Windows System*; gewährleistet die Window-Funktionalität und stellt architekturunabhängige Grafikfunktionen zur Verfügung.
- *virtuelle Bildschirme*; Illusion von mehreren Ausgabegeräten
- Einbindung von Keyboard, Maus, Tablet, usw. inklusive aktueller Schreibposition (*Cursor*) und Edit-Funktionen (*Insert Mode / Overstrike Mode*)

[2] Jede Ähnlichkeit mit dem Schreibtisch des Verfassers sollte reiner Zufall sein.

10.3 Deadlocks

Eine konsequente Weiterführung und Lösung des in Kapitel 6.3 vorgestellten Dijkstra's *Dinning Philosophers Problem* ist die Einführung eines sehenden und hörenden Kellners, der die gerechte Verteilungs-Funktion übernimmt, sodass kein Denker verhungern muss (die Reihenfolge ist in diesem Fall ausschlaggebend).

Übertragen in die Computerwelt, (Philosophen=Prozesse ohne Kommunikation mit zeitweise (alleinigen) Anspruch auf zwei Gabeln=Objekte, Kellner=Type Management), sind für die Entstehung einer Deadlock-Situation vier notwendige Bedingungen festzustellen:

1. *Mutal Exclusion* (ein bestimmtes Objekt kann zu jedem Zeitpunkt von höchstens einem Programmprozess benutzt werden).
2. *Resource Waiting* (wenn ein angefordertes Objekt besetzt ist, geht der anfordernde Programmprozess in den Zustand BLOCKED über).
3. *Partial Allocation* (Programmprozesse, die bereits im Besitz von Objekten sind, können die Zuteilung weiterhin beantragen).
4. *Nonpreemption* (ein einmal zugeteiltes Objekt kann nur vom Programmprozess selbst zurückgegeben werden, nicht aber von außen dem Programmprozess weggenommen werden).

Zur formalen Darstellung dienen *Resource Allocation Graphs*: ein Graph mit gerichteten Kanten (=erfolgte bzw. nicht erfüllte Anforderungen) verbundenen Knoten (=Prozesse und deren Objekte). Mögliche Lösungsstrategien wären:

Deadlock Detection and Recovery: Regelmäßige Überprüfung auf Zyklen im Resource Allocation Graph; wird durch brutale Terminisierung eines Prozesses aufgelöst. (Nachteil: stark erhöhter System-Ouverload)

Deadlock Prevention: Durch geeignete Betriebssystemkonzeption (z.B. *Spooling* von Druckern) soll es gar nicht zum Deadlock kommen.

Deadlock Avoidance: Ähnlich wie *Deadlock Detection*: spezielle Algorithmen können jedoch bereits vorausschauend bei der Zuteilung von Resourcen spätere potentielle erkennen (zusätzliche Information der Objekte ausschlaggebend).

Anmerkend ist zu erwähnen, dass eine zu große Konzentration auf Dead-Locks sinnlos ist; ein Maschinenabsturz durch Hard- oder Softwarefehler ist eigentlich viel wahrscheinlicher.