Informatik (II)
Vorlesung
Sommersemester 2008

Johannes Waldmann, HTWK Leipzig

8. April 2008

Einleitung/Überblick

Inhalt

- Technische Grundlagen
  - Rechnerarchitektur
  - Prozessoren
  - Bus und Schnittstellen
  - Speichertechniken
- Betriebssysteme
  - Aufgaben, Ziele, Konzepte, Strukturen
  - Ein/Mehr-Prozeß/Nutzer-Systeme
  - Vergabe von Speicher, Geräten, Rechenzeit
  - Scheduling, Threads
- Netze, Sicherheit
  - Netze: Schichten-Modell, Adressierung, Dienste
  - Kompression, Fehlererkennung, Verschlüsselung
  - Datenschutz (Recht) und Datensicherheit (Technik)
  - Kryptographie mit öffentlichen Schlüsseln

Organisation

Vorlesung:

- ungerader Dienstag, 13:45–15:15, G126
- gerader Dienstag, 7:30–9:00, Li110

Übungen:

- *entweder* Do(u) 13:45 + Do(g) 7:30
- *oder* Do(u) 15:15 + Do(g) 9:30

Prüfungszulassung: Übungsaufgaben,
Prüfung: Klausur.

Literatur

- Christian Horn, Immo Kerner, Peter Forbrig (Hrsg.): *Lehr- und Übungsbuch Informatik, Grundlagen und Überblick*, 3. Auflage, Fachbuchverlag Leipzig

  ergänzend:
- Christian Horn, Immo Kerner: *Lehr- und Übungsbuch Informatik (Band 4), Technische Informatik und Systemgestaltung*
- William Stallings: *Betriebsysteme*, Addison-Wesely/Pearson, 2003

Betriebssystem: Beispiel GNU/Linux

- Übungen mit Linux-Live-CD Knoppix
- im MM-Pool und zuhause
- Version 5.3, sobald verfügbar, siehe `http://www.knopper.net/knoppix/knoppix53.html`
- Download: im MM-Pool `Y:/Shareware/Knoppix`
- CD/DVDs bitte selbst vervielfältigen.

Rechner-Architektur

Rechnerarchitektur

(HKF S. 56 ff)
die Bestandteile:

- CPU
- Arbeitsspeicher
- Festplatte
- E/A-Module

das Bus-System:

- Adressbus,
- Datenbus,
- Steuerbus

Begründung für Architektur

implementiert von-Neumann-Modell:

- Programmausführung besteht aus Folge von Schritten
- ein Schritt: Anwenden einer Operation (in CPU) auf Operanden (aus Speicher)
- Programme sind Daten (d. h. beide stehen im Hauptspeicher)

Die CPU (central processing unit)

Steuerwerk:

- Befehlszähler
- Befehlsregister
- Adressregister

Rechenwerk:

- Datenregister (mehrere)
- ALU (arithmetical and logical unit)
- Zusatzregister (Flags)

Befehls-Abarbeitung in CPU

- (Adresse des aktuellen Befehls steht in Befehlszähler) Holen des Befehls in Befehlsregister
- Dekodieren und Ausführen des Befehls

  (dabei evtl. Speicher lesen/schreiben, rechnen)
- Bestimmung der Folgeadresse (in Befehlszähler)

  (Nachfolger bzw. Sprung)
- Behandeln von Unterbrechungen

  (ausgelöst durch Hard- oder Software)

Befehlssätze

- Verarbeitungsbefehle (ALU-Operationen auf Datenregistern)
- Transportbefehle (Datenregister ↔ Hauptstspeicher)
- Ein/Ausgabe-Befehle
- Sprungbefehle:
  - unbedingter Sprung
  - bedingter Sprung (abh. von ALU-Flagregister)
  - Sprung (zu Unterprogramm) mit Rückkehrabsicht
  - Rückkehr (aus Unterprogramm)

Übung KW 11

Suchbilder: Innenleben eines PCs (2005):
`http://dfa.imn.htwk-leipzig.de/~waldmann/edu/ss05/informatik/bilder/pc/`

- Suchen Sie die wesentlichen Bestandteile!
- Welche Einzelteile (außer Schaltkreisen) gibt es?
- Welche Kapazität hat die Festplatte?
- Wie heißt die Grafikkarte?
- Suchen Sie Informationen zu einem der sichtbaren Schaltkreise! (google nach Aufdruck)

- Der Prozessor ist ein AMD Athlon XP 2500, wieviele Register besitzt er, welche Rechenoperationen kann er ausführen?

Unterschiede zu 2008?
`http://dfa.imn.htwk-leipzig.de/~waldmann/edu/ss08/informatik/bilder/pc/`

Beispiel: IA32

(Intel-architektur für Prozessoren mit 32-Bit-Adressierung, enthält x86 ... Pentium)
siehe `http://developer.intel.com/design/pentium4/manuals/index_new.htm`
Register:

- acht Register (je 32 bit): EAX, EBX, ECX, EDX, EBP, ES
  EDI, ESP; sechs spezielle Adress-Register (16 bit); Flag-Register EFLAGS, Befehlszähler EIP
- Gleitkomma-Rechenwerk: acht Register (je 80 bit)
- MMX-Register usw.

Befehle (nicht vollständig):

- Transport: MOV
- Arithmetik: ADD, SUB, MUL, DIV, INC, DEC, NEG
- Logik: AND, OR, XOR, NOT
- Bit-Operationen: SAR, ...
- Ablaufsteuerung: JMP, JNZ, CALL, RET

IA32-Beispiel

eine Funktion aus einem C-Programm:
`int f (int x) { return 3*x + 1; }`
übersetzt mit `gcc -S` ergibt sich:

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %edx // das ist x
movl    %edx, %eax
addl    %eax, %eax
addl    %edx, %eax
incl    %eax // das ist Resultat
popl    %ebp
ret
```

CISC/RISC

Man kann CPUs nach der Komplexität des Befehlssatzes unterscheiden:

- CISC: *complex* instruction set computer
- RISC: *reduced* instruction set computer

CISC: *complex* instruction set

- alle Operanden sind frei adressierbar, direkt und indirekt

```
mem[123] := mem[mem[400] + 10] * 18
```

- Programme sind kürzer (weniger Befehle drücken mehr aus)
- Ausführung eines Befehls dauert (evtl.) lange,
- Entwurf und Herstellung der CPU ist kompliziert

RISC: *reduced* instruction set

- nur einfache Transportbefehle (Register ↔ Hauptspeicher)
- Operanden für Rechenbefehle nur in Registern
- Programme länger (mehr Befehle benötigt)
- Befehlsausführung schneller
- CPU-Entwurf und -Herstellung einfacher

Historische Entwicklung

- Speicher war teuer → Programme sollen kurz sein → komplexe Befehle (CISC), Beispiel VAX, Pentium II
- CISC-CPU-Herstellung war teuer, Speicher wurde billiger → einfache Befehle (RISC), Beispiel: Sparc
- heute: Herstellen komplizierter Schaltkreise ist möglich → Mischformen (CISC/RISC)

Siehe *RISC vs. CISC, the Post-RISC-Era*, ars technica 4/99, `http://arstechnica.com/cpu/4q99/risc-cisc/rvc-1.html`

Maschinenzahlen

Darstellung und Verarbeitung von Zahlen

Information wird binär repräsentiert, kleinste Einheit: 1 Bit (Strom oder kein Strom, 0 V oder 5 V, Ladung oder keine Ladung)
alles weitere daraus zusammengesetzt, z. B. ganze Zahlen im Binärsystem:
$110101_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32 + 16 + 4 + 1 = (53)_{10}$
jede natürliche Zahl besitzt genau eine solche Darstellung.

Von Zahlen zu Bitfolgen

```
static int width = 8; // zum Beispiel
static boolean [] encode (int x) {
    boolean result [] = new boolean [width];
    for (int i=0; i<width; i++) {
        if ( 0 == x % 2 ) {
            result [i] = false;
        } else {
            result [i] = true;
        }
        x = x / 2; // wird abgerundet
    }
    return result;
}
```

Von Bitfolgen zu Zahlen

```
static int width = 8; // zum Beispiel
static int decode (boolean [] x) {
    int accu = 0;
    for (int i=width-1; i>=0; i--) {
        accu *= 2;
        if (x[i]) {
           accu ++;
        }
    }
    return accu;
}
```

Binäre Addition

Addition von zwei Binärzahlen (mit Übertrag) wie üblich.

```
   13      1 1 0 1
+  25    1 1 0 0 1

------------------
```

Realisierung in Hardware (ALU) durch Folge von Addierschaltkreisen.
diese bestehen aus (jeweils wenigen) Transistoren

Negative Zahlen, binäre Subtraktion

Zur Darstellung negativer ganzer Zahlen bei fixierter Bitbreite benutzt man das *Zweierkomplement*:

Beispiel: 8 bit

$C(13) = 2^8 - 13 = 243 = (11110011)_2$

Man erhält das Zweierkomplement $C(x)$ einer Binärzahl x, indem man das Einerkomplement bestimmt
(für alle Bits $0 \leftrightarrow 1$) und dann um 1 erhöht. (Beispiel $C(13)$)

Zahl $< 0 \iff$ höchstes Bit $= 1$
Darstellbarer Zahlhbereich:
$-128 = -2^7 \le x \le 2^7 - 1 = +127$

Subtraktion: $x - y = x + C(y)$ (Überlauf ignorieren)
(Beispiel: $23 - 13$)

Kleine ganze Zahlen sind zuwenig

exakte Binärdarstellung ist geeignet für (kleine) ganze Zahlen, aber *nicht* für

- *sehr große* Zahlen
- nicht ganze (gebrochene oder reelle) Zahlen

Diese kommen praktisch vor, und man möchte mit ihnen wenigstens näherungsweise rechnen.

Gleitkomma-Zahlen

Teile Zahl x in Mantisse m und Exponent e, so daß $x = m \cdot 2^e$.

Normierung: $1 \le m < 2$

$10^6 = 2^{19} + 2^{18} + 2^{17} + 2^{16} + 2^{14} + 2^9 + 2^6 =$
$(11110100001001000000)_2 \approx (1.11101)_2 \cdot 2^{19}$

$(13.5)_{10} = 27/2 = 27/16 \cdot 8 = (1.1011)_2 \cdot 2^3$

- Mantisse und Exponent haben je ein Vorzeichen (Darstellung im Zweierkomplement)
- Trick: wegen der Normierung ist das höchste Bit der Mantisse $= 1$, könnte also weggelassen werden (? 0)

Gleitkomma-Normen

genormt in IEEE 754, verfügbar (u. a.) in Java

- 24 bit für Mantisse, 8 bit für Exponent (*float*),
- 53 bit für Mantisse, 11 bit für Exponent (*double*)

Welches ist größte darstellbare Zahl? zweitgrößte? kleinst
positive darstellbare Zahl? zweitkleinste?
(Wo sind die Bits für die Vorzeichen?)
David Goldberg: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, `http://docs.sun.com/source/806-3568/ncg_goldberg.html`

Gleitkomma-Zahlen in Java

```
float  x = 7.9;
double y = 2.4e17; // bedeutet 2.4 * 10^17
```

- Typen sind `float` und `double`
- Mantisse hat einen Dezimalpunkt
- oder es folgt ein Exponent (mit `e` oder `E`)
- Mantisse und Exponent können Vorzeichen `+`, `-` besitzen

Darstellbare Zahlen

- ein Format kann eine endliche Menge rationaler Zahlen darstellen
- es gibt rationale Zahlen, die nicht exakt darstellbar sind,

  Beispiele: $1/3, 0.2$
- sehr große und sehr kleine Zahlen sind nicht darstellbar

  (Überlauf, *overflow*, Exponent zu groß)

  Beispiele: $10^{10^{10}}, -10^{10^{10}}$
- Zahlen sehr nahe bei 0 sind nicht darstellbar

  („Unterlauf", *underflow*, Exponent zu klein)

  Beispiele: $10^{-1234567}, -10^{-1234567}$

Rechenfehler

Da Nachkommastellen (in der Mantisse) abgeschnitten werden, treten Rundungs-Fehler auf.
IEEE-Standard verlangt:

- erst exakt rechnen (notfalls mit mehr internen Stellen)
- dann auf korrekt runden (auf nächste darstellbare Zahl).

Rundungsfehler pflanzen sich fort.

```
1.0 / 2.0 + 1.0 / 3.0 + 1.0 / 6.0
  ==> 0.9999999999999999
1e40 - 1e5 * 1e35
  ==> 1.2089258196146292e24
```

Abhängigkeit vom Rechenweg

geschickte Umformungen vermeiden evtl. gefährlich Rundungen

```
double x = 1e+10;
double y = 1e-10;
System.out.println ( (x+y)*(x-y) - x*x );
  ==> 0.0
System.out.println ( x*x - y*y   - x*x );
  ==> 0.0
System.out.println ( x*x - x*x   - y*y );
  ==> -1.0000000000000001E-20
```

Automatische Typ-Anpassungen

für arithmetische Operationen gilt:
wenn *wenigstens einer* der Operanden einen Gleitkomma-Typ hat, dann *ist* es eine Gleitkomma-Operation.
ggf. wird der andere Operand in Gleitkommazahl verwandelt.

```
System.out.println (2   + 1   / 7);
System.out.println (2.0 + 1   / 7);
System.out.println (2.0 + 1.0 / 7);
```

Manuelle Typ-Verwandlungen

durch Vorsetzen des Zieltyps in Klammern:

```
System.out.println ( (double) 2 );
   ==> 2.0
System.out.println ( (int) 2.7 );
   ==> 2
```

Vor Abschneiden evtl. Runden:

```
System.out.println ( (int) Math.round (2.7)
   ==> 3
```

Logische Schaltungen

Wahrheitswerte und Funktionen

$\mathbb{B}$ = Menge der Wahrheitswerte (Java: `boolean`)
Java: `false`, `true`, mathematisch oft: $\{0, 1\}$
jede Funktion $f : \mathbb{B}^k \to \mathbb{B}$ heißt *aussagenlogische* oder *Boole*sche Funktion.
George Boole (1815–1864) veröffentlichte 1854: An investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities.
`http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Boole.html`

Wertetabellen

Für jedes $f : \mathbb{B}^k \to \mathbb{B}$ ist der Definitionsbereich endlich. (Wi
groß genau?)
Die Funktion f kann deswegen durch eine Wertetabelle gegeben werden. Beispiele:

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$g(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$h(x,y)$
0	0	1
0	1	0
1	0	0
1	1	1

Für jedes k gibt es nur endlich viele verschiedene k-stellig
Boolesche Funktionen. (Wieviele genau?)

Wichtige Boolesche Funktionen

- nullstellig: die Konstanten `false` und `true`
- einstellig: identische Funktion, Verneinung (Nicht, $\neg$)

x	id(x)
0	0
1	1

x	$\neg x$
0	1
1	0

- zweistellig: Disjunktion (Alternative, Oder, $\vee$), Konjunktion (Und, $\wedge$), Implikation (Folgerung, $\to$)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \to y$
0	0	1
0	1	1
1	0	0
1	1	1

Notation in Java

- Negation: `!x`

  (vgl. auch Ausrufezeichen in `a != b`)
- Disjunktion: `x || y`
- Konjunktion: `x && y`
- Implikation: (mathematisch $x \to y$)

  In Java wäre `x <= y` (kleiner-oder-gleich bzgl. `false < true`) denkbar, aber das gibt es tatsächlich nicht.

Rechenregeln

Für

$$(\texttt{false}, \texttt{true}, \vee, \wedge)$$

gelten die gleichen Rechenregeln wie für

$$(0 \quad , 1 \quad , +, \cdot)$$

(für Zahlen). (Nennen Sie einige.)
... und noch weitere, z. B. das De-Morgansche Gesetz:

$$\neg x \wedge \neg y \;=\; \neg(x \vee y)$$

Augustus De Morgan (1806–1871),
`http://www-history.mcs.st-andrews.ac.uk/`
`Mathematicians/De_Morgan.html`

Basis-Funktionen

Die Funktionen

- Nicht (einstellig)
- Oder (beliebig viele Stellen)

  $\texttt{Oder}(x_1, \ldots, x_n) = 1$ gdw. wenigstens ein $x_i = 1$
- Und (beliebig viele Stellen)

  $\texttt{Und}(x_1, \ldots, x_n) = 1$ gdw. alle $x_i = 1$

bilden eine *Basis*: man kann jede andere Boolesche Funktion durch Kombination von Nicht, Oder, Und darstellen (evtl. Autotool-Aufgabe dazu).

Beispiele: $(x \to y) \;=\; \text{Oder}(\text{Nicht}(x), y) = \neg x \vee y$
$(x{=}{=}y) \;=\; (\neg x \wedge \neg y) \vee (x \wedge y).$

Basis-Satz (Beweis-Idee)

Man stellt f als *Disjunktion* von Funktionen $f_1, f_2, \dots$ dar (je eine Funktion für jede 1 in der Wertetabelle von f)

x	y	$f(x,y)$	$f_1(x,y)$	$f_2(x,y)$
0	0	1	1	0
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Jedes solche f_i hat dann genau eine 1 in seiner Wertetabelle, kann also selbst als *Konjunktion* dargestellt werden.

$$f_1(x,y) = \neg x \wedge \neg y, \quad f_2(x,y) = x \wedge y$$

Eine Basis mit einem Element

Die Funktion $\mathrm{NAND} : (x,y) \mapsto \neg(x \wedge y)$ bildet eine Basis.

x	y	$x \wedge y$	$\mathrm{NAND}(x,y)$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Beweis: wir können diese Funktionen darstellen:

- Nicht: $\neg x = \mathrm{NAND}(x,x)$
- Und: $x \wedge y = \neg\,\mathrm{NAND}(x,y)$
- Oder: $x \vee y = \mathrm{NAND}(\neg x, \neg y)$.

…und mit dieser Basis auch alle anderen.

nand in Hardware

Die Funktion NAND kann leicht mit drei Transistoren realisiert werden: `http://www.allaboutcircuits.com/vol_4/chpt_3/6.html`. man packt 4 davon in einen Schaltkreis (74F00, je nach Hersteller, z. B. `http://www.philipslogic.com/products/gates/nand/`)
…und daraus kann man (Basis-Satz!) CPUs bauen.
Historisches zu Rechner-Prozessor-Aufbau:

- aus Einzel-Elementen (1934 Relais, 1945 Elektronenröhre, 1955 Transistor)
- (1960) aus Standard-Schaltkreisen (nand, …)
- (ab 1970) aus Spezial-Schaltkreisen

Übung Logik

Übung zu Booleschen Funktionen

Stellen Sie die Funktionen durch die angegebenen Operatoren dar.

- `x == (y == z)` durch false, true, nicht, und, oder
- `x || y && !z` durch false, true, <= (Implikation)
- `(p == q) && (q != r) || (p != s)` durch false, true, nicht, und, oder

Speicher

Hauptspeicher

Mit zwei nand-Bausteinen kann man ein Bit speichern: `http://www.play-hookey.com/digital/rs_nand_latch.html`
Das ist ein Schalt-*Werk* (= Schalt-*Netz* mit Rückführungen), es heißt Flip-Flop.

Hauptspeicher (II)

Hauptspeicher aus Flip-Flops:

- Nachteil:
  - mehrere Transistoren pro Bit,
  - Stromfluß auch ohne Zugriff

grundsätzlich andere Technik: Ladungsspeicherung (wie i Kondensatoren).

- Vorteil:
  - nur ein Transistor pro Bit,
  - Stromfluß nur bei Zugriff,
- Nachteil:
  - wg. Ladungsverlusten regelmäßiges Lesen/Überschreiben nötig.

Physikalische Grundlagen für Speicher

- Halbleiter-Speicher
  - statisch (Flip-Flops)
  - dynamisch (Transistor als Kondensator)
  - irreversibel (ROM)
- magnetische Speicher (Magnetband, Diskette, Festplatte)
- optische Speicher (CD, DVD)

„Historische“ Speicherprinzipien

- mechanische Speicher:

  (Zahnräder, Hebel: Pascal 1650, Babbage 1823, Zuse (Z1) 193?)

  (Lochkarte: Jaquard 1805; Lochband)
- elektromechanisch (Relais), Zuse (Z2/Z3) 1936;

  elektronisch (Röhren)
- magnetisch:

  fest: Ferritkerne,

  rotierend: Magnettrommeln

Disketten und Festplatten

Speichermedium: magnetisch beschichtete Scheibe(n) adressiert durch

- Nummer des Kopfes (der Scheibe)
- Nummer der Spur (des Zylinders)
- Nummer des Sektors

Aufgaben:
Mit welcher Geschwindigkeit bewegt sich der Kopf eine 3.5-Zoll-Platte bei 7200 U/min relativ zur Magnetschicht? Wie dicht liegen die Spuren nebeneinander? Die Bits?

CD und DVD

Prinzip: entlang einer Spur gibt es Vertiefungen (pits) und Erhebungen (lands), bei Abtasten wird ein Laserstrahl reflektiert (land) oder nicht (pit).
Vertiefungen sind „oben“, darauf ist Schutzschicht (Lack), abgetastet wird von unten.
CD: pit-Länge: 0.8 μm, Spur-Abstand: 1.6 μm
Aufgaben: wie lang ist die Spirale? wie schnell dreht sich die CD? wieviel Bit (pits) passen drauf?
DVD: pits und Abstand $\approx$ halb so groß wie bei CD
DVD-9 verwendet zwei Schichten (Laser muß anders fokussiert werden, erste Schicht: innen nach außen, zweite: außen nach innen).

Die Speicher-Hierarchie

Ein Computer benutzt Speicher dieser Art:

1. CPU- und ALU-Register
2. Hauptspeicher
3. Festplatte (im Gerät)
4. Archiv-Speicher (extern)

dabei gilt (von oben nach unten):

- Zugriffszeit steigt,
- Größe steigt,
- Kosten (pro Bit) sinken.

Cache

Idee: langsamen (umfangreichen, preiswerten) Speicher S beschleunigen durch (wenig) zusätzlichen, schnelleren (teureren) Speicher C

- C enthält Kopien von einigen Bereichen von S
- bei jedem Zugriff auf S wird stattdessen die (schnellere) Kopie in C benutzt — falls sie vorhanden ist.
- …falls nicht, wird ein länger nicht benutzter C-Bereich aufgegeben und mit dem jetzt aktuellen Bereich (= Date aus der Nähe des aktuellen Zugriffs) gefüllt.

Bemerkung: obiges gilt für Lese-Zugriffe. Geschrieben wird meist auf C *und* S (write-through)

Zugriffsformen

- inhaltsbezogene (assoziative) Speicher:

  in Hardware derzeit selten (Cache)

- orts-adressierter Speicher
  - wahlfreier Zugriff (random access memory, RAM)
  - sequentieller Zugriff (Magnetband, Lochband)
  - index-sequentieller Zugriff (wahlfreier Zugriff auf Blöcke mit sequentiellem Inhalt) (Festplatten)

Verwaltung eines Dateisystems (Abbildung von Datei-Namen auf Block-Folge) erfolgt durch Betriebssystem.

Speicher und Bus

Wie kommt die CPU an die Daten? Einfachster Fall:

- alle Speicherbausteine (RAM, Platte, CD) sind mit Bus verbunden
- CPU schreibt gewünschte Adresse auf Adressbus (und wartet)
- Speicher erkennt Adressbereich, holt gewünschte Daten und schreibt sie auf Datenbus
- CPU wacht auf, liest Daten vom Datenbus

OK für RAM, ansonsten nicht effizient: CPU muß auf (langsame) Speicher warten.

Asynchroner Datentransport

CPU könnte in der Wartezeit weiterrechnen,
muß aber bemerken, wenn die Daten tatsächlich ankommen.
(die Speicher-Einheit signalisiert das auf Steuer-Bus)
zwei Möglichkeiten:

- CPU fragt Steuerbus regelmäßig ab (polling)
- Steuersignal unterbricht Rechnung der CPU (interrupt)

interrupts sind die bessere Lösung (keine „sinnlosen“ Abfragen, Trennung von Rechnung und Datentransport), müssen aber richtig verwaltet werden (Rechnung muß korrekt fortgesetzt werden → Register retten)
(interrupt während einer interrupt-Behandlung?)

Direkter Speicherzugriff (DMA)

im einfachen Modell erfolgt jeder Datentransport über (Bus und) CPU,
z. B. DVD → CPU → RAM

falls nur (große) Datenblöcke kopiert werden sollen, braucht man die CPU eigentlich nicht
z. B. DVD → RAM

mit direct memory access (DMA) können Geräte zum Datenaustausch unabhängig von CPU den Bus benutzen

Andere Ein/Ausgabe-Geräte

Alle Ausführungen zu Speichern
(Adressierung, Bus, Interrupts, DMA)
gelten sinngemäß auch für die anderen (unmittelbar oder mittelbar) am Bus angeschlossenen Geräte
(Grafikkarte, Tastatur, Maus, Modem usw.)

Betriebssysteme

Was ist ein Betriebssystem?

Literatur: HKF, Kapitel 3, S. 143 ff + ergänzend:

- William Stallings: *Betriebssysteme*, Addison-Wesley 200
- Andrew Tanenbaum: *Moderne Betriebssysteme*, AW 200?
- Linus Torvalds, David Diamond: Just for Fun, DTV 2002

Das Betriebssystem ist die Schnittstelle zwischen Nutzer(programmen) und Computer-Hardware.

Leistungen eines Betriebssystems

- definiert *virtuelle Maschine*,

  d. h. systematisieren und vereinfachen den Zugriff auf Hardware
- organisiert *parallelen Zugriff* auf einzelne Betriebsmittel

  (Gerechtigkeit, Effizienz, Datenschutz)
- Steuerung, Protokollierung, Abrechnung der Rechnernutzung

Zugriffe auf das Betriebssystem

- Kommando-Interpreter (shell) → System-Prozeduren

  z. B. `ls -l, /sbin/ifconfig`
- Anwendungsprogramme → Programmbibliothek → System-Prozeduren

  z. B. `lame -h foo.wav foo.mp3` liest und schreibt Dateien
- grafische Oberfläche (z. B. `kde`, `gnome`)

  verwaltet Anwendungsprogramme (z. B. `konsole, konqueror`)

Geschichte: Einprogramm-BS

(ca. 1955–1962)
eigentlich: Rechner *ohne* Betriebssystem
Bediener startet *ein* Programm,
das direkt auf Hardware zugreift.

- unflexibel,
- Bediener erforderlich
- schlechte Rechnerauslastung (durch Warten auf E/A)

Geschichte: Stapel-BS

(ca. 1960–1968)
eigentlich: Geburtsstunde der Betriebssysteme
mehrere Programme werde unter Steuerung eines *Monitorprogramms* ausgeführt

- Monitor muß höhere Rechte als Nutzerprogramme habe
- gerechte Verteilung der Ressourcen muß organisiert (programmiert) werden
- kein Bediener erforderlich
- bessere Auslastung (während E/A anderes Programm ausführen)

Time-Sharing-BS

(ab 1968)
Rechenzeit wird in schnellem Wechsel an mehrere Prozesse verteilt
dadurch interaktiver Betrieb möglich
üblich war

- ein Rechner (Server)
- mehrere Nutzer (an Text-, später Grafik-)Terminals
- mehrere Prozesse pro Nutzer

Geschichte - Zusammenfassung

- Einprogramm-Betrieb
- Stapelbetrieb
- Timesharing, Dialogbetrieb
- verteilte Systeme (mehrere Prozessoren, mehrere Rechner)

Ziele bei BS-Entwicklung

- Erweiterbarkeit

  (erreicht durch Schichten, Module)
- Portabilität

  (erreicht durch weitgehend maschinenunabhängige Programmierung, meist in C)
- Zuverlässigkeit, Robustheit (gegen Fehlbedienung, Hardwarefehler)

  (erreicht durch gestaffelte Zugriffs- und Ausführungsrechte)
- Kompatibilität

  (erreicht durch Implementierung von genormten Schnittstellen, z. B. POSIX - portable operating system standard)
- Leistung (global und interaktiv)

  (erreicht durch clevere Programmierung)

Welt- und Individualgeschichte

historische Entwicklung der Betriebssysteme wird beim Linux-Start „wiederholt“:

- Start im Einprogramm-Betrieb
- Textmodus (Boot mit`knoppix 2 lang=de`

  verschiedene Konsolen `ALT-F1, ALT-F2, ..`
- grafische Oberfläche (bei knoppix: KDE und GNOME wählbar)

  die Textkonsolen sind trotzdem noch da (`CTRL-ALT-F1` usw, zurück mit `ALT-F6`)

  graf. Oberfläche beenden durch `CTRL-ALT-DEL`, wieder starten durch `xinit`

Netzkonfiguration MM-Pool

- IP-Adresse: 141.57.118.Arbeitsplatz
- Netzmaske: 255.255.255.192
- Broadcast: 141.57.118.255
- Gateway: 141.57.118.3
- Nameserver (DNS): 141.57.1.1

Prüfen mit `ifconfig`, muß `eth0` (= Ethernetkarte Nr. 0) anzeigen
Auf Rechnern 31–33 vorher manuelles Laden des Kartentreibers (= Kernel-Modul) nötig:
`sudo modprobe sk98lin`
(`sudo` bedeutet: mit Superuser-Rechten)

Übung (KW 16)

Rechnen wie in grauer Vorzeit

Booten mit Option `knoppix 2` ergibt Textmodus, mit Kommandointerpreter (Shell) `bash`
mit `ALT-F1, ALT-F2` usw. zwischen virtuellen Konsolen umschalten.
typische Befehle:

- Verzeichnis anzeigen: `ls`, `ls foo`, wechseln: `cd foo`, erzeugen: `mkdir foo`, löschen: `rmdir foo` (muß leer sein)
- Datei erzeugen/editieren: `emacs foo.text` (beenden mit `C-x C-c`)
- Datei-Art anzeigen: `file foo.bar`, Datei-Inhalt ausgeben: `cat foo.bar`, kopieren: `cp`, verschieben (umbennen) `mv`, löschen: `rm`

  Aufgabe: in welcher Reihenfolge stehen die beiden Argumente bei `cp` und `mv`? Ausprobieren!
- Die meisten Programm gestatten weitere Optionen. Es gibt Hilfetexte, z. B. `man date`, `info emacs`, `ls --help`

  Aufgabe: Wie kann man Datei-Größen anzeigen lassen? (Suche Antwort in der Hilfe/manpage zu `ls`.)

Grafische Oberflächen

- aus Textkonsole starten: `startx` (KDE ist voreingestellt)
- Aufgabe: virtuelle Bildschirme ausprobieren (auf jedem andere Anwendungen starten)
- Auflösung umschalten: `Ctrl-Alt-NumPlus`
- auf Textkonsole zurückschalten: `Ctrl-Alt-Fi`, von dort wieder zur Grafik mit: `Alt-F5`
- beenden mit `Ctrl-Alt-Del`

Aufgabe: weitere Oberflächen ausprobieren.
Beim Booten `knoppix desktop=icewm`, weiter

Wo sind jeweils die virtuellen Bildschirme?

Einblicke in die Arbeit des Systems

- Kommandofenster (Konsole) öffnen,
- Hardware-Info mit `cat /proc/cpuinfo`, `lspci`, `usbview`, `gscanbus`
- System-Name und -Version: `uname -a`

  geladene Module (Treiber): `/sbin/lsmod`

  Logmeldungen des Systems: `dmesg`

  Aufgabe: einige Zeilen aus `dmesg` verstehen.
- aktuelle Prozesse anschauen durch `ps`, `ps -ax`, `pstree` (statisch)
- oder `top` (dynamisch - verlassen mit q)

  Aufgabe: `top` starten und dann einige Anwendungsprogramme ausführen, dabei Prozeßliste betrachten.

  Welche Prozesseigenschaften werden angezeigt?

Die Herkunft von Linux

Wiederholung der Betriebssystem-Entwicklung auch bei der Linux-Entwicklung (vgl. Torvalds: Just for Fun, S. 69) wollte (Jan. 1991) von PC über Modem auf Uni-Rechner zugreifen. benötigt zwei gleichzeitig laufende Prozesse:

- vom Modem lesen und auf Bildschirm schreiben
- von Tastatur lesen und ins Modem schreiben

Verwaltung dafür *ist* bereits eine Art Betriebssystem.

Linux wird zu einem „echten“ BS

sobald diese (POSIX-)Funktionen implementiert waren:

- Prozeßverwaltung (Speicherzuteilung, Kommunikation)
- Gerätezugriff (z. B. Modem, Bildschirm, Tastatur)
- Dateisystem (zunächst übernommen von Minix),

(Oktober 1991, Version 0.02) konnte man UNIX-Anwendungs-Programme auf Linux (kompilieren und) ausführen:
Shell (`bash`), Editor (`vi, emacs`), Compiler (`gcc`).

kurze Geschichte von UNIX

- Ken Thompson 1968, mit Dennis Ritchie: C

  verbessertes, vereinfachtes MULTICS für pdp7/11
- Richard Stallman ca. 1984: GNU (GNU's not UNIX)

  freie Re-Implementierung/Verbesserung von UNIX
- Andrew Tanenbaum 198?: Minix (für PCs)

  Ergänzung seiner Lehrbücher über Betriebssysteme
- Linus Torvalds ab 1991: (GNU/)Linux

  Verbesserung von Minix durch Ausnutzung von neuen Prozessor-Eigenschaften (Intel 386)

Verbreitete Betriebssysteme

- Großrechner
  - VMS (DEC/Digital)
  - IBM: OS/360 (Fred Brooks: *The Mythical Man-Month*, 1964), VM/370, OS/390, MVS
- „Klein“rechner, Personal-Computer:
  - Apple
  - DOS, Windows(3, 95, 98, NT, XP), OS/2
  - UNIX-Varianten ((Free-)BSD, GNU-Mach, NeXT, Darwin, GNU/Linux)

Vorsicht: jedes hat seinen Sinn (historisch und aktuell), es gibt hier kein *Gut* und *Böse*.

Dateisysteme

Dateisysteme

Wdhlg: auf der Festplatte stehen Bits, *physisch* gruppiert in Spuren, Sektoren, Schichten.
man möchte die Bits aber *logisch* gruppieren, sie sollen eine Sammlung von *Daten* darstellen = eine *Datei*.
Eine Sammlung von Dateien wird durch ein *Dateisystem* verwaltet.

Designziele (allgemein)

(nach Stallings: Betriebssysteme, 12. 1. 2.)

- soll Arbeit des Nutzers mit Dateien ermöglichen
- soll Gültigkeit der Datei-Inhalte gewährleisten
- soll Leistung optimieren:
  - aus Systemsicht: hoher Gesamtdurchsatz
  - aus Nutzersicht: kurze Antwortzeiten
- soll verschiedene Speichermedien unterstützen
- soll standardisierte Schnittstellen besitzen
- soll in Mehrnutzer-Systemen die Datei-Operationen mehrere Nutzer koordinieren

Designziele (aus Nutzersicht)

(nach Stallings: Betriebssysteme, 12. 1. 2.)
Jeder Benutzer muß/sollte können:

- Dateien erstellen, löschen, lesen, ändern
- kontrolliert auf fremde Dateien zugreifen
- Zugriffsrechte der eigenen Dateien (gegenüber Fremden) festlegen
- Form und Struktur der Dateien frei wählen
- Dateien sichern und wiederherstellen
- auf Dateien über (selbst gewählte, symbolische) Namen zugreifen

UNIX: index-sequentielle Dateien

- die kleinste *logische* Datengruppe ist ein *Block*

  (z. B. 1 kByte, die hintereinander auf einer Spur stehen — sequentielle Anordnung)
- Eine Datei besteht aus mehreren Blöcken,

  diese bilden eine Baumstruktur

  (innere Knoten: Indexblöcke, Blätter: Datenblöcke)
- für kurze Dateien soll wenig Zusatz-Verwaltung stattfinden:

  benutzt abgestufte Index-Höhe (0 bis 3)

Freispeicher-Verwaltung

- bei Bedarf muß das System sehr schnell einen freien Block finden:
- anstatt dann erst einen zu suchen, werden freie Blöcke vorher zu verketteter Liste verbunden

  (beim Formatieren des Datenträgers)
- diese Freiliste ist immer aktuell zu halten

  (Blöcke aus gelöschten Dateien wieder dort einfügen)

Datei-Informationen (inodes)

zu jeder Datei (= Gruppe von Blöcken) gibt es genau eine *inode* (Informations-Knoten). Dort steht:

- Dateimodus
  - Art: gewöhnlich/Verzeichnis/Speziell/Pipe
  - Rechte: Lesen/Schreiben/Ausführen für Besitzer/Gruppe/Welt erlaubt?
- Besitzer-ID (Nutzer, Gruppe)
- Dateigröße
- Block- bzw. Indexblock-Adressen
- Datum des letzten Zugriffs/der letzen Änderung der Datei/der Inode

Namen, Verzeichnisse

- Jede Datei gehört zu einem *Verzeichnis* (directory).
- Ein Verzeichnis ist selbst eine Datei, die eine Listen von Datei-Verweisen (Paare von Dateiname und Inode-Adresse) enthält.
- Jedes Verzeichnis enthält wenigstens diese zwei Verweise:
  - Name `.` auf sich selbst
  - Name `..` auf übergeordnetes Verzeichnis (parent)
- Das oberste Verzeichnis im gesamten System ist `/` (root).

Absolute und relative Pfade

- Eine Folge von jeweils direkt untergeordneten Verzeichnisnamen heißt *Pfad*
- Verzeichnisnamen werden voneinander und vom Dateinamen durch `/` (slash) getrennt.

  `/home/waldmann/edu/ss04/informatik/folien/`
- Ein Pfad, der mit `/` beginnt, heißt *absoluter* Pfad

  (beginnt bei root)
- alle anderen *relative* Pfade

  (relativ zu `.` = aktuelles Verzeichnis)

  Beispiel: `../../programme/dining/Philo.java`

Ausführung von Programmen

ausführbare Programmdateien enthalten

- Maschinencode
- oder Skripte (Shell-Befehle)

Wenn man in der Shell `prog arg1 arg2 ..` eingibt, dann wird die erste ausführbare Datei mit Name `prog` in den Verzeichnissen des aktuellen `PATH` benutzt

```
echo $PATH
  /usr/local/bin:/usr/bin:/usr/X11R6/bin:/bi
which emacs
  /usr/bin/emacs
emacs foo.java # ==> /usr/bin/emacs foo.java
```

Rechte-Verwaltung

- In einem UNIX-System gibt es
  - verschiedene *Nutzer* (siehe `/etc/passwd`)
  - und verschiedene *Gruppen* (siehe `/etc/group`)
- Jeder Nutzer gehört zu einer (oder mehreren) Gruppen. Jede Datei gehört zu genau einem Nutzer und genau einer Gruppe.
- Jede Datei hat neun verschiedene Zugriffsrechte:
  - Lesen/Schreiben/Ausführen ($\boxed{\text{r}}$ead/$\boxed{\text{w}}$rite/e$\boxed{\text{x}}$ecute)
  - für Besitzer/Gruppenmitglieder/andere ($\boxed{\text{u}}$ser/$\boxed{\text{g}}$roup/$\boxed{\text{o}}$ther)

Rechte (Dateien und Prozesse)

- Jeder Prozeß, der von einem Nutzer gestartet wird, erbt dessen Rechte für Datei-Operationen.
- Die meisten Dateien sind für alle lesbar, aber nur für Besitzer schreibbar.
- Die Systemdateien sind nur für den Administrator (root) schreibbar.
- Damit kann ein korrekt installiertes und laufendes System allein durch Nutzereinwirkung eigentlich nicht kaputtgehen.
- Man sollte nur in Ausnahmefällen mit root-Rechten arbeiten (zum Konfigurieren des Systems, Installieren neuer Pakete).

Partitionen, mounts

- Bereiche von Festplatten heißen *Partitionen* und können in ein Dateisystem *eingehängt* (gemountet) werden

```
/dev/hda6 on / type ext3 (rw)
/dev/hda1 on /media/c type ntfs (rw,noexec,nosuid,n
/dev/hda5 on /media/e type vfat (rw,noexec,nosuid,n
proc on /proc type proc (rw)
```

- Dabei können auf einzelnen Partitionen unterschiedliche Dateisysteme verwendet werden (siehe Beispiel)
- Das funktioniert auch für Netzwerk-Dateisysteme, RAM-Disks, CD-ROMs (siehe knoppix).

In Unix gilt: *alles ist eine Datei.*

UNIX-Dateisystem (Übung)

File-Operationen

- Erstellen: z. B. durch Editieren: `emacs` *name*

  Textdatei ausgeben: `cat` *name*

  Löschen (remove): `rm` *name*

  Kopieren (copy): `cp` *quelle ziel*

  Verschieben (move): `mv` *quelle ziel*
- Verzeichnisinhalt (list): `ls`, `ls` *name*,

  auch versteckte Dateien anzeigen (all): `ls -a`

  ausprobieren: `ls ../knoppix/./Desktop/..`

Nutzer, Gruppen, Datei-Rechte

- Welche Nutzer gibt es? `less /etc/passwd`

  (blättern vorwärts: space, rückwärts: b, verlassen: q)
- Welche Gruppen gibt es? `less /etc/group`
- Zu welcher Gruppe gehören Sie? `groups`

Standard-Verzeichnisse

```
/etc      # Config-Dateien

/usr      # Standard-System
/usr/bin  # Programme
/usr/man  # Man-Pages (Dokumentation)

/usr/local     # Erweiterungen
/usr/local/bin # Programme

/var      # Logdateien
/tmp      # temporär
/home     # Nutzerbereiche
```

In einem stabilen System brauchen nur die letzten drei (`/var, /tmp, /home`) schreibbar zu sein (andere können auf Read-Only-Partitionen stehen)

Spezielle Dateien

- ... alles ist eine Datei: auch Hardware!

  Geräte(treiber) werden auch über Dateien angesprochen

  (Festplatte `/dev/hda`, Brenner `/dev/dvdrecorder`, Soundkarte `/dev/dsp`, Maus `/dev/usb/mouse0`)
- ... auch Software: System-Informationen stehen im virtuellen `/proc`-Filesystem
- ergibt gewaltige Vereinfachungen und Einsparungen bei der Benutzung und Programmierung (seit 1970!)
- unter anderem darauf bezieht sich der Spruch:

  Those who do not know UNIX are doomed to re-invent it — poorly.

Verzeichnis-Operationen

- aktuelles Verzeichnis anzeigen

  (print working directory): `pwd`
- aktuelles Verzeichnis wechseln

  (change directory): `cd` *pfad*
- neues Verzeichnis erstellen

  (make directory): `mkdir` *pfad*
- Verzeichnis löschen (muß leer sein)

  (remove directory): `rmdir` *pfad*

- Zu welchem Nutzer/Gruppe gehören die Dateien?

  ausführliche Datei-Angaben (long, inode): `ls -li`

  welche Bedeutung haben die Ausgaben?

  Vergleichen Sie mit den inode-Bestandteilen (Vorlesung)

```
ls -li .
ls -li /cdrom
ls -li /
```

Spezielle Dateien: Links (Verknüpfungen)

- hard link:

  `ln` *quelle ziel*

  erzeugt einen neuen Verzeichnis-Eintrag *ziel*, der auf die gleiche inode wie *quelle* zeigt.

  überprüfen (inodes anzeigen) mit: `ls -il`

- soft link:

  `ln -s` *quelle ziel*

  erzeugt Verzeichniseintrag *ziel* und neue inode, die als „link"markiert ist und den Namen *quelle* enthält.

  überprüfen mit `ls -il`

Dateiarbeit unter Java

Die Klasse `File`

liefert Abstraktion für Datei/Verzeichnis-Namen:
Pfad = [Präfix] (Dir Sep)* Name

- Unix: Präfix = `/`, Sep = `/`
- Windows: Präfix = `A:` usw., Sep = `\`

```
class File {
    File (String path);
    File (File parent, String child);
    String getName ();
    File getParentFile ();
}
```

System-unabhängige Pfade

Schnittstelle:

```
interface FileSystem {
    char getSeparator();
    String normalize(String path);
}
```

Implementierung:

```
class UnixFileSystem implements FileSystem {
    char getSeparator () { return '/'; }
    String normalize(String path) { .. }
}
```

File-Eigenschaften

```
class File { ..
    boolean exists ();
    boolean canWrite ();
    boolean canRead ();

    boolean isFile ();
    boolean isDirectory ();

    long length ();
}
```

Datei-Inhalt lesen

Dateiinhalt ausgeben (wie Programm `cat`):

```
File f = new File ("foo.bar");
FileReader r = new FileReader (f);
int c;
while ((c = r.read ()) != -1) {
  System.out.print ((char) c);
}
```

- Zuweisung ist als Ausdruck erlaubt — hat welchen Wert?
- Datei- (Stream-)Ende durch „Zeichen" `-1`

  (u. a. deswegen) werden ints gelesen (nicht chars)
- Umwandlung in char durch *typecast*

ASCII-Kodierung

american standard code for information interchange
orig. 7 bit: Buchstaben, Ziffern, Zeichen.

```
waldmann@dfa:$ od -cb ascii.tex
0000000   \   b   e   g   i   n   {   s   l   i   d
        134 142 145 147 151 156 173 163 154 151 144
0000020   C   I   I   -   K   o   d   i   e   r   u
        103 111 111 055 113 157 144 151 145 162 165
0000040   (   a   m   e   r   i   c   a   n       s
        050 141 155 145 162 151 143 141 156 040 163
```

nur für lateinisches Alphabet geeignet.
andere Alphabete erfordern

- andere Codes (französisch, kyrillisch)
- Unicode (mehrere Bytes pro Zeichen, chinesisch)

Zeilenweises Arbeiten

```
File f = new File ("foo.bar");
BufferedReader r =
    new BufferedReader (new FileReader (f));
int c = 0;
for (String s;(s = r.readLine ()) != null;){
    c++;
}
System.out.println (c + " lines");
```

beachte: Ende durch `null`
Aufgabe: zusätzlich alle Zeichen zählen

Java-Exceptions (try-catch, throws)

Ausnahme-Bedingungen, deren Behandlung woanders stattfindet (in Exception-Handler)

```
try {
 .. // geschützter Block
} catch (IOException e) {
 .. // Handler
}
```

Falls Exception nicht behandelt wird,
muß ihre Weitergabe deklariert werden:

```
void foo ()
   throws IOException
{ .. // ungeschützter Block
}
```

IOExceptions

- Datei existiert nicht
- Datei ist speziell
- Datei ist nicht lesbar
- Datei ist nicht schreibbar

Ein selbst-druckendes Programm

Wie funktioniert das? (Was steht in `int [] [] a`?)

```
class Self {
   public static void main (String [] args) {
int [] [] a = { { 9, 83, 121, 115, ... }, ... };
        System.out.println ("class Self {");
        System.out.println ("    public static void
        System.out.print ("int [] [] a = ");
        for (int i = 0; i < a.length; i++) {
            String isep = (0 == i) ? "{ " : ", ";
            System.out.print (isep);
            for (int j = 0; j < a[i].length; j++) {
                String jsep = (0 == j) ? "{ " : ", "
                System.out.print (jsep + a[i][j]);
            }
            System.out.println (" }");
        }
```

```
        System.out.println ("};");
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a[i].length; j++) {
                System.out.print ((char) a[i][j]);
            }
            System.out.println ();
        }
    }
}
```

kompletter Quelltext hier: `http://www.imn.htwk-leipzig.de/~waldmann/edu/current/informatik/programme/self/Self.java`

Finden Sie kürzere selbstdruckende Programme? (google: self printing program)

Speicher, Prozesse

Verwaltung des Hauptspeichers

(siehe HKF 3.2.2, S. 155ff)

Problem: im ausführbaren Programm stehen Adressen von Daten (Variablen) und (Unter-)Programmen.

- *absolute* Adressen (bereits bei Herstellung (Kompilation) des Programms)

  nur Einprogramm-Modus, unflexibel, Speicherverwaltung ist Handarbeit

- *relative* Adressen in kompilierten Files, bei Verbinden (link) und Laden ersetzen durch absolute Adressen

  spart Re-Kompilation, trotzdem unflexibel (geladene Programme sind nicht verschieblich)

Speicher-Verwaltung (II)

- benutze relative Adressierung *in der CPU selbst*:

  jeder Speicherzugriff $m[i]$ wird übersetzt in $m[B+i]$, wobei B ein CPU-Register (für Basis-Adresse) ist.

  Für jedes Programm beginnt (scheinbarer) Adressbereich bei 0, ($\Rightarrow$ Programme sind verschieblich)

dabei beachten:

- Zugriff/Addition von B muß sehr schnell gehen,
- Prozeß darf B nicht ändern

(Def: *Prozeß* $=$ laufende Instanz eines *Programms*.)

Aufteilung des Speichers

Speicheranforderung der Prozesse ist nicht vorhersehbar, muß aber trotzdem sinnvoll und schnell benatwortet werden.

- Vergabe von zusammenhängenden Speicherbereichen (einer je Prozeß)

  unflexibel, ungerecht, Fragmentierung

- Vergabe von Folgen von Teilstücken (pages)

  (z. B. in Intel-CPU (ab 80386) verwaltet über page directory, page table)

(das war der Anlaß für Linus Torvalds, einen Unix-Kernel für diese Prozessoren zu schreiben)

Kernel- und User-Mode

User-Prozesse sollen von relativer und virtueller Adressierung gar nichts merken.
Dazu muß es aber einen Steuerprozeß (den *Kernel*) geben, der mit absoluten Adressen rechnet.
scharfe Trennung zwischen User-Mode und Kernel-Mode:

- User-Prozeß darf nur eigene Speicher-pages benutzen, und nicht in Kernel-Mode umschalten

- nur durch Interrupt (page fault, timer) geht CPU in Kernel-Mode

Virtueller Speicher

- Prozesse müssen nicht immer komplett im Hauptspeicher wohnen.
  Selten genutzte pages kann man bei Bedarf auslagern (in swap-file).
- Bedarf entsteht, wenn ein anderer Prozeß eine page benutzen will, die *nicht* mehr in der Speicher paßt.
  (Anforderung einer ausgelagerten page löst Interrupt aus.)
- Kandidaten zum Auslagern: pages, die lange nicht benutzt wurden.
  (Betriebssystem benutzt geeignete Markierungen der pages.)

Prozesse

- werden erzeugt (aus parent-Prozeß heraus)

  (vgl. `pstree`)
- laufen oder warten

  (worauf könnte ein Prozeß warten?)
- enden normal oder werden abgebrochen

  (wodurch?) (vgl. `kill`)

vgl. Grafik HKF S. 160

Scheduling

- Das Betriebssystem verteil die CPU-Zeit auf die rechenbereiten Prozesse (d. h. die nicht auf E/A warten).
- benötigt Algorithmen zu *gerechten* Zuordnung.
- benutzen Prioritäten, die sich evtl. dynamisch ändern (im Laufe der Zeit von groß nach klein).

Aktives und passives Warten

Prozeß, der auf Eintreten einer Bedingung wartet, kann

- immer wieder nachfragen (aktives Warten, *polling*)
- *schlafen*, bis er benachrichtigt wird (passives Warten)

passives Warten ist besser, das Betriebssystem verwaltet für jede Ressource eine Liste aller Prozesse, die darauf warten.
das ist wenig zusätzliche Arbeit, denn das Freiwerden der Ressource geschieht ja selbst auch durch einen Betriebssystem-Aufruf

Prozeß-Kommunikation

Prozeße kommunizieren (tauschen Daten)

- mit der Außenwelt (E/A)
- untereinander

Datenaustausche erfolgt

- in gemeinsamen Speicherbereich (unsicher)
- oder (sicher) über festgelegten Kanal (pipe)

  vgl. Pipe-Symbol in der Shell

typisches Problem: ausschließlicher Zugriff auf ein Betriebsmittel (z. B. Drucker, Festplatte)

Ein- und Ausgabe-Umleitung

- im einfachsten Fall liest ein Prozeß die Eingabe von der Konsole (Eingabe-Ende: Control-D) und schreibt die Ausgabe auf die Konsole.

```
sort
abc
xxxx
lkhlkj
aabkj
Control-D
```

- durch < die *Eingabe* eines Prozesses von einer Datei holen (anstatt von der Konsole)

```
sort < foo.text
```

- durch > die *Ausgabe* eines Prozesses in eine Datei umleiten (anstatt auf die Konsole)

```
ls -l > dir.text
```

überprüfen mit `ls` und `cat`

Prozeßkommunikation durch Pipes

alle Leerzeichen durch Zeilenschaltung ersetzen (translate

```
man cat | tr " "  "\\n"
```

. . . und Zeilen sortieren:

```
man cat | tr " "  "\\n" | sort
```

. . . dabei gleiche Zeilen nur einmal ausgeben (sort, unique

```
man cat | tr " "  "\\n" | sort -u
```

. . . die Ausgabezeilen zählen (word count, lines)

```
man cat | tr " "  "\\n" | sort -u | wc -l
```

ergibt Anzahl verschiedener Wörter in manpage von `cat`

Spezielle Dateien: Benannte Pipes

Eine Pipe (= Röhre, durch die Daten fließen)
ist ein Mittel zur Prozeß-Kommunikation unter UNIX.

Ein Prozeß schreibt in die Pipe, der andere Prozeß liest aus der Pipe. Beide Prozesse sind dadurch *synchronisiert*.

Die bei `|` benutzten pipes sind *temporär*.
Es gibt auch *permanente* Pipes, das sind spezielle *Dateien* (d. h. sie haben eine Inode, einen Namen, usw.)

- Anlegen mit `mkfifo` *name*
- überprüfen mit `ls -li`

- Schreiben Sie in die pipe: `man mkfifo >` *name*
  (Vorsicht: scheint zu verklemmen. Ist richtig so.)
- Öffnen Sie eine zweites Befehlsfenster (konsole).
  Lesen Sie aus der pipe: `cat` *name*
- Lesen Sie erneut aus der pipe
  (Zweite Konsole, Vorsicht, ...)
- Schreiben Sie erneut in die pipe. (Erste Konsole.)

Threads (KW 18)

Threads

moderne Programmiersprachen gestatten „innere“ Prozesse:

- Programm enthält mehrere threads (Fäden)
- aber Betriebssystem sieht nur einen Prozeß
- threads haben gemeinsame Umgebung (Klassen, Variablen)

⇒ Multi-Threading in Java

Java-Threads (I)

Java-Threads (II)

Java-Exceptions (try-catch, throws)

Ausnahme-Bedingungen, deren Behandlung woanders stattfindet (in Exception-Handler)

```
try {
 .. // geschützter Block
} catch (IOException e) {
 .. // Handler
}
```

Falls Exception nicht behandelt wird,
muß ihre Weitergabe deklariert werden:

```
void foo ()
   throws IOException
{ .. // ungeschützter Block
}
```

Threads in Applets

ein Applet kann für seine Bestandteile jeweils eigene Threads verwenden. Bsp:

- ein Zähler, der von selbst nach unten zählt (= ein Thread)
- aber durch einen Click hochgesetzt werden kann
- Applet = mehrere solche Zähler/Buttons, die gleichzeitig laufen

Aufgabe: machen Sie daraus ein interessantes Spiel!
(Ziel/Wertungspunkte festlegen, ausrechnen, anzeigen)
Quelltexte (mit Lücken):
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/informatik/programme/threads/`

Beispiel: Clients/Server

- ein Server (backt und verkauft Brötchen Nr. $0, 1, 2, \ldots$)
- mehrere Kunden (kaufen das jeweils nächste),
  jeder Kunde ist ein selbständiger `Thread`

Aufgabe: Ablauf so organisieren, daß

- jedes Brötchen auch verkauft wird
- keines mehrfach verkauft wird

Client-Objekte (I)

```
class Customer implements Runnable {
    private String name;
    private Clerk clerk;
    private int items;

    Customer (String n, Clerk c, int i) {
        name = n; clerk = c; items = i;
    }
    ...
}
```

Client-Objekte (II)

```
class Customer implements Runnable {
  private String name;
  private Clerk clerk;
  private int items;  ...
  public void run () {
    for (int k=0; k<items; k++) {
      Util.nap ();
      int x = clerk.get ();
      System.out.println
        ("Customer " + name + " got item "
    }
  }
}
```

Abwarten ...

```
class Util {
  static void nap () {
    int d = (int) (Math.random () * 1000);
    try { Thread.sleep (d); }
    catch ( Exception e ) {
      System.out.println
          ("got exception: " + e);
    };
  }
}
```

Hauptprogramm

```
class Shop {
  public static void main (String [] argv)
    Clerk cl = new Clerk ();
    Customer [] customer = new Customer [5]
    for (int i=0; i<customer.length; i++) {
      customer[i] =
          new Customer ("Cust" + i, cl, 5);
      new Thread (customer [i]).start ();
    }
  }
}
```

fehlt natürlich noch der Server (Verkäufer)!

Server-Objekt

so einfach geht es aber *nicht*:

```
class Clerk {
    private int clock = 0;
    int get () {
        System.out.println ("get request");
        int item = clock;
        Util.nap ();
        clock ++;
        System.out.println ("answer request"
        return item;
    }
}
```

(ausprobieren)

Synchronisierte Methoden

Die Server-Methode `get()` ist nicht *thread-safe*: Sie liefer bei gleichzeitiger Ausführung durch mehrere Threads falsche Ergebnisse.
Abhilfe (ausprobieren):

```
class Clerk { ..
    synchronized int get () { .. }
}
```

Durch `synchronized` kann die Methode jeweils nur einmal laufen.
In Java läuft zu jeder Zeit für jedes Objekt *höchstens eine* seiner `synchronized`-Methoden.
Damit sind Zugriffe auf Attribute abgesichert.

Fünf Philosophen

ein klassisches Beispiel für multi-threading:

- 5 Philosophen. Jeder einzelne: denkt, wird hungrig, ißt, denkt, wird hungrig, ißt, ...
- runder Tisch, in der Mitte ein Topf Spaghetti (wird nie alle)
- es gibt zwischen je zwei Tellern genau eine Gabel, zum Essen benötigt man beide.

Die Ressourcen (Gabeln) müssen den Clients (Philosophen) so zugeteilt werden, *so daß keiner verhungert*.

Gefahr 1: Deadlock (Verklemmen)

- wechselseitiger Ausschluß (jede Ressource von nur einem Prozeß benutzbar)
- besitzen und warten (Prozeß besitzt bereits Ressourcen und wartet auf weitere)
- kein Ressourcenentzug (kein anderer als der Besitzer-Prozeß selbst kann die Ressource freigeben)

  (beachte: bis hierher sind es sehr vernünftige Forderungen)
- Zyklisches Warten (jeder Prozeß besitzt wenigstens eine Ressource, auf die ein andere wartet)

Gefahr 2: Unfairness (Verhungern)

Ressourcen-Zuteilung unfair =

- es gibt einen Ablauf der Ereignisse,
- bei dem wenigstens ein Prozeß nie die gewünschten Ressourcen erhält.
- (. . . weil ihm die anderen abwechselnd alles wegnehmen)

Deadlock?

Dieser Ansatz hat Deadlock (ausprobieren):

```
class Philo {
  void run () {
    while (true) {
      System.out.println (id() + " hat Hung
      right.take (this);
      left.take (this);
      System.out.println (id() + " ist satt
      right.drop (this);
      left.drop (this);
} } }
```

Wie kann man das verhindern?

Lösungen

- ein Diener läßt immer höchstens vier Philosophen am Tisch platznehmen
- der Philosoph mit Nummer 0 ist Linkshänder (= nimmt linke Gabel zuerst)

Beweisen Sie, daß damit Verklemmen und Verhungern vermieden werden!

Archivierung, Kompression, Verschlüsselung

Überblick

Hilfsprogramme für die Arbeit mit Dateien (Datenströmen)

- Archivierung

  (mehrere Dateien ↔ eine Archiv-Datei)
- Kompression

  (eine Datei ↔ eine kürzere Datei)
- Verschlüsselung

  (eine Datei ↔ eine „unlesbare"Datei)

Kompression

berechnet effizientere Darstellung unter Ausnutzung

- der inneren Struktur der Datei (Wiederholungen,. . .)
- der „äußerer Struktur" der Anwendung (Bild oder Musik)

man unterscheidet

- verlustfreie K. (z. B. für Texte, Programme)
- verlustbehaftete K. (z. B. Bild, Musik, Film)

Kompression–Anforderungen

(teilweise widersprüchlich)

- hohe Kompression
- schnelle Kompression
- schnelle De-Kompression

Verlustfreie Kompression

- zeichenweise (Kodierungen):

  move-to-front, Huffman
- blockweise:

  Lempel-Zhiv-Welch (zip), Burrows-Wheeler (bzip2)

Beispiele:

```
zip   -9 informatik.pdf.zip  informatik.pdf
bzip2 -9 informatik.pdf -c > informatik.pdf.bz2
ls -l
 275766 informatik.pdf
 148264 informatik.pdf.zip
 143766 informatik.pdf.bz2
```

ftp://rtfm.mit.edu/pub/usenet/news.answers/compression-faq/

Kodierungen

(ASCII ist schon eine Kodierung, die aber nichts komprimiert, da jedes Zeichen 8 bit lang ist.)
Idee:

- benutze variable Code-Länge
- häufige Zeichen bekommen kurze Codes

Probleme:

- Ende der Codewörter muß erkennbar sein

  → benutze *Präfix*-Codes
- Häufigkeit der Zeichen ist vorher nicht bekannt

  → benutze *dynamische* Codes

Codes

Ein *Code* ist eine Abbildung c
von Original-Zeichen zu Folgen von Code-Zeichen (oft: Bits), die im folgenden Sinne umkehrbar ist:
es gibt *keine* Wörter $x_1x_2 \ldots x_m \neq y_1y_2 \ldots y_n$
mit $c(x_1)c(x_2)\cdots c(x_m) = c(y_1)c(y_2)\cdots c(y_n)$.
Beispiele:

- $A \to 00, B \to 01, C \to 10, D \to 11$ ist umkehrbar
- $A \to 0, B \to 10, C \to 11, D \to 110$ ist nicht umkehrbar
- $A \to 00, B \to 0011, C \to 10, D \to 01$ ist …?
- $A \to 00, B \to 001, C \to 10, D \to 01$ ist …?

Präfix-Codes

Eine Menge von Wörtern $\{w_1, w_2, \ldots\}$ heißt *präfix-frei*,
wenn es keine verschiedenen i, j gibt, so daß w_i ein *Präfix*
(Anfangsstück) von w_j ist.
Satz: Jede präfixfreie Menge ist eine Code.
Beispiele:

- $A \to 00, B \to 01, C \to 10, D \to 11$ ist ein Präfix-Code.
- $A \to 0, B \to 10, C \to 110, D \to 111$ ist ein Präfix-Code.
- $A \to 00, B \to 0011, C \to 10, D \to 01$ ist kein Präfix-Code, aber ein Code!

Huffman-Code

- Jeder Präfix-Code entspricht einem Binärbaum

  (in den Blättern stehen die Zeichen)
- welches ist (bei gegebener Zeichen-Häufigkeit) der beste Präfix-Code(-Baum)?
- Lösung: Sortiere die Zeichen nach Häufigkeit, fasse jeweils die zwei seltensten zusammen

  Dieser Code heißt Huffman-Code.

Dynamischer Huffman-Code

Nachteile des Huffman-Codes:

- keine Ein-Pass-Verarbeitung möglich (Häufigkeiten sind erst am Ende bekannt)
- Code(-Baum) muß mit übertragen werden

Ausweg:

- beginne mit angenommener Gleichverteilung

  ($\to$ alle Codewörter gleich lang, z. B. ASCII)
- berechne Huffman-Code-Baum

  nach Lesen *jedes* Zeichens neu

Move-To-Front

- ordne alle Zeichen in Liste $l = [A, B, C, D, E, F]$
- wiederhole für jedes zu kodierende Zeichen x:
  - $c(x) =$ Position von x in l
  - bewege (move) x unmittelbar nach Benutzung an das linke Ende (to front) von l
    (andere Zeichen rutschen nach rechts)

Vorteil:

- häufigste Zeichen weit links $\to$ kleine Codes sind häufig
  $\to$ fortsetzen mit Huffman
- Verfahren ist dynamisch

Lempel-Ziv-Welch-Kompression (LZW)

benutzt Wörterbuch (dictionary) für wiederholte Blöcke, ersetzt diese durch ihre Nummer.

```
Wörterbuch d = ASCII (0 .. 255);
String     w = "";
for (jedes neue Zeichen x)
   if (w x  ist in  d)
   then
      w = w x;
   else
      gib Code für  w  aus;
      schreibe  w x  in  d;
      w = x;
```

Lempel-Ziv (sliding window) LZ77

- kein extra Wörterbuch,
- sondern suche *aktuellen Block* in *aktuellem Fenster*
- falls gefunden, gibt (Position, Länge) aus

benutzt in `arj, lha, zip, zoo`

Burrwos-Wheeler-Transformation

eine *umkehrbare* Transformation, die (im Allg.) einen viel besser komprimierbaren String erzeugt.

Beispiel: `abraca`.

alle zyklischen Permutationen:	diese sortieren (lexikographisch):
`("abraca",0)`	`("aabrac",5)`
`("bracaa",1)`	`("abraca",0)`
`("racaab",2)`	`("acaabr",3)`
`("acaabr",3)`	`("bracaa",1)`
`("caabra",4)`	`("caabra",4)`
`("aabrac",5)`	`("racaab",2)`

Folge der letzten Buchstaben und Position der 0:
`("caraab", 1)`
daraus kann man die Eingabe rekonstruieren!

Burrows-Wheeler (II)

der transformierte String läßt sich besser komprimieren (z. B. mit move-to-front, Huffman)

```
t? Ihr dr .. ahn geneig    ten Menge, .. r unbekann
taeuscht, ..  Glueck ge    ten sang;  .. ch die ers
talten, D .. nkende Ges    ten steige .. iebe Schat
te Lieb u ..  Kommt ers    ten, Die f .. nde Gestal
te Widerk .. h! der ers    ten, Wie i .. gt ihr wal
teh ich n .. uehn. Da s    ten, die,  .. nnt die Gu
teigen au .. Schatten s    ten, halbv .. h einer al
teigt; Me ..  um mich s    ten? Fuehl ..  festzuhal

hheeeeaeeeehheheeeoiohiieehoeeoeddeeeuieruebftfFeeeby
fiieeeeeeeereeetfekeeeeuaeaneauleaste id   ur essa  s
 oaiiiininbnerr   riirr errsssnnlhmihgiigrshgflslebru
geggesssssnstllullssttnsrks haeisSszeaaeeseeatrlrFfm
```

Burrows-Wheeler-Aufgabe

Das Resultat einer BW-Transformation ist

```
( "ootreeesth__bhbuttt__otti_n_oieao__sn_q"
, 36
)
```

- Setzen Sie die `bzip2`-Kompression weiter fort (move-to-front, dynamic huffman).
- Wie lautete die Eingabe?

Überblick

Aspekte der Informations-Übertragung/Speicherung:

- beschränkte Kapazität → Kompression
- fehlerhaftes Medium → Fehler-Erkennung und -Korrektur
- öffentliches/abgehörtes Medium → Verschlüsselung

Fehler-Erkennung und -Korrektur

Prüfsummen

(vgl. Horn, Kerner, Forbrig, Band 2, Kapitel 9)

- man nimmt (etwas) *Redundanz* in Kauf, um fehlerhafte Übertragungen zu bemerken.
- Beispiel: für jedes Byte $b_7 b_6 \dots b_0$ ein zusätzliches *Paritäts-Bit* p, so daß $b_7 + b_6 + \dots + b_0 + p \equiv 0 \mod 2$
- auch komplizierter Funktionen und mehrere Prüf-Bits werden verwendet (Beispiel: `md5sum` — Aufgabe: welcher Algorithmus? siehe `http://www.ietf.org/rfc/rfc1321.txt?number=1321`)
- leicht geänderte Eingabe (in wenigen Bits) → stark geänderte Prüfsumme

Hamming-Abstände

Code-Wörter sind Bit-Folgen.
alle Bit-Folgen gleicher Länge bilden einen *metrischen Raum* durch die Abstandsfunktion (Hamming distance)

$$\operatorname{dist}(u, v) := |\{i : u_i \neq v_i\}|$$

(= Anzahl der Positionen, an denen sich die Werte (Bits) unterscheiden).
(formuliere und beweise die Dreiecks-Ungleichung)
Richard Wesley Hamming (1915 – 1998)
`http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Hamming.html`

Abstände und Codes

die Hamming-Weite einer Menge M von Codewörtern ist

$$\operatorname{dist}(M) = \min\{\operatorname{dist}(u, v) : u \in M, v \in M, u \neq v\}$$

(der kleinste Abstand zwischen zwei Wörtern)
Beispiel: $\operatorname{dist}\{110, 101, 011\} = 2$.
Finde möglichst viele Code-Wörter der Länge l mit Weite $\geq w$! (Beispiel: $l = 6, w = 3$)

Fehler-Erkennung und -Korrektur

- Ein Code M kann k-Bit-Fehler *erkennen*, falls $\operatorname{dist}(M) > k$.
- Ein Code M kann k-Bit-Fehler *korrigieren*, falls $\operatorname{dist}(M) > 2k$.
- Zu empfangenem (möglicherweise verfälschtem) Codewort w' bestimmt man das nächstliegende (bzgl. dist) tatsächliche Codewort.
- Falls $\operatorname{dist}(M) > 2k$ und höchstens k Bit falsch, dann ist dieses eindeutig bestimmt.

Wieviel Bits braucht man für eine Codierung der Dezimalziffern, die 2-Bit-Fehler korrigieren kann?

Perfektion/Redundanz: Kosten/Nutzen

keine Hardware ist vollständig perfekt (durch physikalische, chemische, thermodynamische Schwankungen)
(der technische Aufwand wächst astronomisch mit dem gewünschten Perfektheitsgrad)
Beispiel: CD/DVD-Rohlinge
man verwendet deswegen Formate mit eingebauter Redundanz, damit man Hardwarefehler beim Lesen korrigieren kann.
(der dabei benötigte Extra-Platz ist viel billiger, als die Rohlinge/Brenner physikalisch zu verbessern)
vgl. *bad blocks* auf Festplatten, Fehler auf Halbleiter-Wafern, auf -Chips
Aufgabe: welcher Zusatz-Aufwand (Platz) für Fehlerkorrektur auf CD-ROMs?

Kryptographie

Verschlüsselung

für Datenübertragung über öffentliche Kanäle wird eine verschlüsselte Form der Nachricht gesendet.

- nur der Adressat soll den Inhalt der Nachricht entschlüsseln können
- der Adressat soll sicher sein, daß die empfangene Nachricht wirklich vom behaupteten Absender stammt

Einfache Verfahren

- *Permutation* der Original-Nachricht (z. B. Wörter spiegeln)
- *Substitution* von Zeichen durch andere (z. B. $A \to B \to C \to \ldots$)

Substitutionsverfahren sind anfällig für statistische Analysen (die Häufigkeitsverteilungen der Buchstaben in natürlichen Sprachen sind gut bekannt)

The Gold Bug

Klassisches Kryptogramm aus
Edgar Allan Poe: The Gold Bug (1843)

```
53++!305))6*;4826)4+.)4+);806*
   ;48!8`60))85;]8*:+*8!83(88)5*!;
46(;88*96*?;8)*+(;485);5*!2:*+
   (;4956*2(5*-4)8`8*; 4069285);)6
!8)4++;1(+9;48081;8:8+1;48!85;
   4)485!528806*81(+9;48;(88;4(+?3
4;48)4+;161;:188;+?;
```

welches ist der häufigste Buchstabe (im Englischen)? der zweithäufigste? wie lautet der Text?

Block-Substitution

gegen statistische Analysen:
⇒ große Blöcke (z. B. 512 Bits) benutzen

UNIX-Passwörter

Aufgabe: Nutzer authentifizieren, aber Passwörter *nirgendwo* im Klartext speichern (weil das die Schwachstelle wäre).
Lösung: man wählt eine Funktion f und speichert nicht das Passwort P, sondern $f(P)$.
Bei Einloggen tippt der Nutzer Q, und das System berechnet $f(Q) \stackrel{?}{=} f(P)$.

Nachteil: gleiche Passwörter $\to$ gleiche $f(P)$.
Lösung: das System würfelt S (salt), speichert $(S, f(S,P))$ und vergleicht mit $f(S,Q)$.
Heutzutage ist dieses Verfahen (mit den Parametern aus den 70er Jahren) durch durch brute-force angreifbar $\to$ Shadow-Passwörter
Schwachpunkt ist jedoch immer die Strecke von Tastatur zum Rechner (der ist evtl. weit weg): das Passwort läuft im Klartext durch das Netz $\to$ diese Übertragung muß auch verschlüsselt werden (`ssh, https`)

Kryptographie mit öffentlichen Schlüsseln

Schlüssel-Übertragung

viele Verfahren beruhen auf *geheimen* Schlüsseln:
nur wer den Schlüssel hat, kann entschlüsseln:
Schlüssel muß auf anderem Weg transportiert werden (das ist dann die Schwachstelle)
Ausweg sind Verfahren mit *öffentlichen* Schlüsseln (genauer: Paaren von zueinander passenden öffentlichen und privaten Schlüsseln — die privaten werden aber *nicht* transportiert)

Einweg-Funktionen

f heißt Einweg-Funktion, wenn

- f ist leicht zu berechnen
- f ist umkehrbar
- Umkehrfunktion f^{-1} ist schwer ($=$ praktisch gar nicht) zu berechnen …

  … es sei denn, man kennt ein „Geheimnis"(key) für f.

Öffentliche und private Schlüssel

Einweg-Funktionen für Verschlüsselung: Nachricht N, verschlüsselte Nachricht $f(N)$, Entschlüsselung $f^{-1}(f(N))$.
dazu müßte man aber den f-Schlüssel übertragen $\rightarrow$ müßte selbst verschlüsselt werden usw. ad inf.

- Empfänger wählt (würfelt) eine Einwegfunktion f und veröffentlicht diese ($=$ public key),
- halt aber ihren Schlüssel geheim ($=$ private key).
- D. h. nur er kann f^{-1} berechnen.
- Absender berechnet $f(N)$ und schickt an Empfänger.

Etwas Zahlentheoretie

- Beobachtung: wenn $n = p \cdot q$ für *große* (z. b. $\approx 10^{100}$) Primzahlen p, q, dann kann man p, q aus n allein praktisch nicht bestimmen.
- (Kleiner) Satz von Fermat/Euler: für $\mathrm{ggT}(x, n) = 1$ gilt gilt $x^{\phi(n)} \equiv 1 \bmod n$.
  wobei $\phi(n) = |F(n)|$ mit
  $F(n) = \{y \mid 0 < y < n, \mathrm{ggT}(y, n) = 1\}$
  (alle Zahlen in $\{1, 2, n-1\}$, die zu n teilerfremd sind)
- Folgerung: wenn $a \equiv 1 \bmod \phi(n)$, dann $x^a \equiv x \pmod n$. (Entschlüsselung)
- man kann $\phi(n)$ aber nur bestimmen, wenn man p, q kennt: $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$.

RSA-Verfahren

(Rivest, Shamir, Adleman)

- potentieller Empfänger wählt eine Zahl d mit $\mathrm{ggT}(d, \phi(n)) = 1$ und bestimmt e mit $de \equiv 1 \bmod \phi(n)$ nach erweitertem Euklidischen Algorithmus.
- Veröffentlicht wird (d, n), geheimer Schlüssel ist e.
- Absender der Nachricht x
  sendet Nachricht $y = x^d \bmod n$.
- Empfänger berechnet $y^e = x^{de} \equiv x^{1+k\phi(n)} \equiv x \bmod n$.

Standard:
`http://www.faqs.org/rfcs/rfc2437.html`

Beispiel

- Empfänger wählt Parameter: $p = 5, q = 11, n = 55$.

  $\phi(n) = (p-1)(q-1) = 40$

  $d = 7, e = -17 \equiv 23 \bmod 40$

  öffentlicher Schlüssel $(d, n) = (7, 55)$,
  geheimer Schlüssel $e = 23$
- Nachricht (Klartext) $x = 14$,

  verschlüsselt $y = x^e = 14^7 \equiv 105413504 \equiv 9 \bmod 55$
- Entschlüsselung

  $y^e = 9^{23} = 886293811965250109592 9 \equiv 14 \bmod 55.$

Hilfsmittel: erweiterter Euklid

Satz: wenn $g = \mathrm{ggT}(a, b)$, dann gibt es Zahlen c, d mit $ac + bd = g$.

```
c = 1; d = 0; e = 0; f = 1;
while (b > 0) {
   // c*a0 + d*b0 = a und e*a0 + f*b0 = b
   q = a / b; r = a % b; // resultat, rest
   a = b; b = r;
   e' = c - q*e; c = e; e = e';
   f' = d - q*f; d = f; f = f';
}

40 : 7 = 5 R 5    5 = 40 - 5*7
```

```
5 : 2 = 2 R 1    1 = 5  - 2*2 = 5*40 - 17*
```

Hilfsmittel: schnelles Potenzieren

```
int power (int b, int e) {
  int p = 1;
  while (e > 0) { // inv:  b^e * p = b0 ^ e0
    if (odd (e)) { p = p * b; }
    b = b * b;
    e = e / 2; // abgerundet
  }
  return p;
}
```

```
   9          23     9
  81 => 26    11   234 => 14
 676 => 16     5   224 =>  4
 256 => 36     2     4
1296 => 31     1   124 => 14
```

Sicherheit

Jeder Lauscher kennt y, d, n (aber nicht p, q, e). Nur durch Kenntnis von p, q kann e bestimmt werden.

Verfahren ist so sicher, wie es unmöglich ist, eine große Zahl n in Primfaktoren zu zerlegen.

RSA factoring challenges

`http://www.rsasecurity.com/rsalabs/node.asp?id=2093`

RSA-576 (174 Stellen) gelöst (10.000 USD) J. Franke, Bonn

```
18819881292060796383869723946165043980716356337941
73827007633564229888597152346654853190606065047430
45317388011303396716199692321205734031879550656996
221305168759307650257059
```

RSA-640 (Decimal Digits: 193) offen (20.000 USD)

```
31074182404900437213507500358885679300373460228427
27545720161948823206440518081504556346829671723286
78243791627283803341547107310850191954852900733772
4822783525742386454014691736602477652346609
```

Primzahlgenerierung/-Test

Für RSA werden große Primzahlen p, q benötigt. Wo kommen die her? Würfeln und testen!

eine Primzahl hat keine nichttrivialen Faktoren.

Faktoren einer großen Zahl anzugeben ist schwer.

Feststellen, *ob* sie nichttriviale Faktoren hat, ist verhältnismäßig viel einfacher.

z. B. nach Satz von Fermat/Euler: wenn $x^{n-1} \not\equiv 1 \bmod n$, dann ist n *nicht* prim. — Um Primalität zu *beweisen*, braucht man aber mehr als nur *ein* $x^{n-1} \equiv 1 \bmod n$.

Es gibt dafür schnelle probabilistische Verfahren (und seit ca. 2 Jahren sogar ein relativ schnelles, exaktes, siehe Primes is in P: `http://www.cse.iitk.ac.in/news/primality.html`)

Mehrstufiges Vorgehen

um Geschwindigkeit zu erhöhen, wird RSA nur benutzt, um zu Beginn der Kommunikation einen Schlüssel sicher zu übertragen, für eine schneller zu berechnende Einwegfunktion.

Unterschriften

Wie kann Empfänger B(ob) sich davon überzeugen, daß Nachricht tatsächlich vom behaupteten Absender A(lice) stammt?

Die Rollen von d, e sind austauschbar: für Nachricht x berechnet Alice $y = x^e \pmod n$ und schickt (x, y) an Bob.

Bob entschlüsselt y mit Alices öffentlichem Schlüssel d (berechnet $y^d = x^{ed} \equiv x \pmod n$) und vergleicht mit x.

Nur Alice war in der Lage, so eine Nachricht y herzustellen.

Tatsächlich wird (x, y) natürlich (mit Bobs öffentlichem Schlüssel) verschlüsselt.

Um Zeit zu sparen, kann Alice nicht ganz x verschlüsseln („unterschreiben“), sondern es genügt ein guter Hashwert von x (z. B. md5sum)

Beglaubigte Unterschriften

wie kann der Absender A prüfen, daß er tatsächlich den öffentlichen Schlüssel des Empfängers B sieht?

Ein Angreifer C könnte sich als B „verkleiden“: Natürlich kommt C nicht an den geheimen Schlüssel von B, aber er kann ein neues Schlüsselpaar generieren und (z. B. auf einer gefälschten Webseite) behaupten, das sei von B.

Ansatz: die Schlüssel (von B) werden von einer vertrauenswürdigen dritten Person P unterschrieben. (Der Angreifer C kann keine Unterschrift von P erlangen.)

Zentraler Ansatz: eine Hierachie von Certificate Authorities

Dezentraler Ansatz (web of trust) — auf key signing parties bestätigen sich persönlich bekannte Personen die Richtigkeit ihrer Schlüssel.

Wirtschaft und Politik

Kryptographie ist bedeutsam:

- wirtschaftlich: sichere Transaktionen (e-business)
- (straf)rechtlich: verdeckte Vorbereitung von Straftaten

das Thema ist deswegen politisch etwas heikel
(Widerspruch zw. freier Rede und Strafverfolgung)
die beste Sicherheit wird durch *Offenlegung* der
Algorithmen entstehen (damit sie durch die
wissenschaftliche Öffentlichkeit diskutiert werden können)

Rechnernetze

Geschichte

- Direktverbindungen zwischen einzelnen Rechnern
- ARPA-Net ((defense) advanced research projects agency) 1969
  - für beliebig viele Rechner beliebiger Architektur
  - bei Ausfällen von Teilstrecken automatische Umleitung
- Internet
  - email, news, file transfer, remote login
  - www, applets, grid computing . . .

OSI-Referenzmodell

- 1979: ISO 7498, open systems interconnection
- definiert verschiedene Übertragungs-Schichten
- jede Schicht bietet Dienste für die jeweils nächst höhere (und für keine andere)

OSI-Referenzmodell (I)

- Anwendungsschicht (application layer)
- Darstellungs-Schicht (presentation layer),
- Kommunikations-Steuerungs-S. (session layer),

OSI-Referenzmodell (II)

- Transportschicht (transport layer):

  definiert end-to-end-Verbindungen
- Vermittlungsschicht (network layer):

  Bestimmung des Übertragungsweges (routing)
- Sicherungs-Schicht (data link layer):

  erkennt Bit-Verfälschungen, bestätigt korrekten Empfang bzw. sendet Paket erneut
- Bit-Übertragungs-Schicht (physical layer):

  mechanische, elektrische, optische Festlegungen

Das Internet

- OSI-Vermittlungsschicht realisiert durch IP-Protokoll
- OSI-Transportschicht realisiert durch TCP-Protokoll

darunterliegende Schichten sind hardwarespezifisch:
Ethernet, Tokenring, (Funk-LAN, ISDN)

Ethernet

eigentlich CSMA/CD: carrier sense, multiple access, collision detection

- multiple access: mehrere Geräte (Netzwerkkarten) auf

  einem *Bus*: Koaxialkabel, Twisted Pair, Glasfaser
- senden kann immer nur einer (Sender muß Kollisionen erkennen)
- hören können alle (= Empfänger muß sich „seine“ Pakete selbst heraussuchen)

  → ist sicherheitskritisch

Token Rings

- Geräte sind an Ring angeschlossen
- Informationen fließen auf Ring in *eine* Richtung
- Sende-Berechtigung (token) wird durchgereicht
- Empfänger sendet Quittung

→ bessere Auslastung, garantierte Antwortzeiten
FDDI (fibre distributed data interface)
realisiert 100MBit/s-Tokenring
(bis 1000 Stationen, bis 200 km Länge, oft als Backbone)

Kopplung von Netzen

physikalisch getrennte Teilnetze

werden verbunden durch *Gateways* (= ein Rechner mit mehreren Netzkarten, in jedem Teilnetz eine)

Beispiel: (W)LAN ↔ Rechenzentrum ↔ DFN-Standleitung

Beispiel: ISDN-Router

IP-Adressierung: Routing

Pakete, die nicht an das lokale Netz gehen, müssen an das passende Gateway geschickt werden.

```
/sbin/route

Kernel IP routing table
Destination     Gateway          Genmask          Ifac
lpz2-d5-1.mcbon *                255.255.255.255  ippp
192.168.1.0     afa.local        255.255.255.0    eth1
192.168.0.0     afa.local        255.255.255.0    eth0
default         lpz2-d5-1.mcbon  0.0.0.0          ippp
```

ausprobieren: `traceroute`

Kopplung: Forwarding

(oft bei privaten Netzen, z. B. mehrere Rechner an einem ISDN-Anschluß)

vom Provider her gibt es nur eine IP-Adresse,
aber lokal sind es verschiedene Rechner.

das Gateway muß in den ausgehenden und ankommenden Paketen die Adressen übersetzen.

Das X-Window-System

für geräte-unabhängige, netzwerkfähige Grafik

X-Server: verwaltet

- Tastatur- und Maus-Eingaben des Nutzers
- Grafik-Ausgaben der Programme

Programm kann entfernten X-Server benutzen:

`kmahjongg -display foo.bar.com:0.0`

Beispiel: UNIX-Server (Fb IMN): viele Client-Rechner, auf denen jeweils nur ein X-Server läuft, ein zentraler Server zum „wirklichen Rechnen“.

auch für X-Window Sicherheitsbedenken (Ein- und Ausgaben unverschlüsselt), besser: `ssh -X`

IP-Adressierung: Karten

jede Netzwerkkarte hat

- eine physikalische Adresse (vom Hersteller fest eingetragen)
- eine logische Adresse (4 byte) (je nach Netz-Konfiguration)

```
/sbin/ifconfig   eth0

Link encap:Ethernet  HWaddr 00:40:F4:0C:40:83
inet addr:192.168.0.80  Bcast:192.168.0.255
Mask:255.255.255.0
inet6 addr: fe80::240:f4ff:fe0c:4083/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

Adressierung: Namen

IP-Adressen sind Zahlenfolgen, aber Verwendung von Namen ist besser:

- sind leichter zu merken
- tatsächliche Adresse ist leichter zu ändern

Übersetzung von Namen in Adressen geschieht durch *Nameserver*

(dessen Zahlenfolge muß natürlich bekannt sein, damit man ihn überhaupt erreicht, z. B. HTWK-Rechenzentrum: 141.57.1.1)

siehe `cat /etc/resolv.conf, host, dig`

Netzwerk-Dienste

klassisch:

- `telnet`: liefert Text-Ein/Ausgabefenster auf entferntem Rechner
- `ftp` (file transfer protocol)
- `smtp` (simple mail transfer protocol)
- `http` (hypertext transfer protocol)

heutzutage mit großen Sicherheitsbedenken: Daten werden unverschlüsselt übertragen!

entsprechende verschlüsselte Dienste sind:

`telnet => ssh, ftp => scp, http => https`

Ports

Dienste werden realisiert über Ports

(z. B. ssh benutzt Port 22, http benutzt 80)

- Server hat „offene Ports“ und wartet auf Client-Anfragen
- Client öffnet selbst einen Port zum Senden der Anfrage (und Empfangen der Antwort)
- Portnummern < 1024 sind *reserviert* (nur von `root` lesbar), größere sind *frei*

ausprobieren mit `netcat, netstat`

Firewalls

offene Ports bieten Angriffspunkte:

Angreifer kann „bösartige“ Nachrichten (Anforderungen) schicken.
Dabei werden meist einfach alle bekannten Schwachstellen von Serverprogrammen durchprobiert, z. B.

```
SuSE-FW-DROP-NEW-CONNECT IN=ippp0 OUT= MAC=
SRC=213.7.170.49 DST=213.7.214.65 LEN=48 TOS=0x00 PRE
TTL=123 ID=55650 DF PROTO=TCP SPT=3526 DPT=135 WINDOW
RES=0x00 SYN URGP=0 OPT (020405B401010402)
```

nur *absolut notwendige* Ports öffnen, die anderen sperren (→ firewall)

Privat-Rechner, auf dem keine Dienste angeboten werden, braucht *keine* offenen Ports

Sicherheitslücken

wichtigste Ursachen:

- leichtsinnige oder unwissende Nutzer
  - kein Root-Password,
  - Installation und Ausführen von unbekannten Programmen,
  - ... durch Öffnen (und Ausführen) von Email-Attachments
  - fehlende Rechte-Trennung (Nutzer $\stackrel{?}{=}$ Root)
  - viele Ports offen, ...
- Fehler in Server-Programmen (durch nachlässige Programmierung)

Buffer Overflow Attacks

in Progammiersprachen ohne Bereichsprüfungen bei Arrays und Zeigern

Beispiel: ein Server-Programm speichert eine harmlose Information (die dem Nutzer gehört) direkt neben einer kritischen:

```
char h [10]; char c [10];
```

Das Ende eines Strings wird (in C) durch ein spezielles Zeichen 0 dargestellt

Angreifer überschreibt die 0 am Ende von `h` und liest danach `h` wieder aus —

```
while (0 != h[i]) { put (h[i]); i++; }
```

sieht dabei alle Zeichen von `h` *und* `c`

Double Dot Attack

Server speicher Nachrichten (o. ä.) in Dateien,
Nutzer soll normalerweise darauf auch von außen zugreifen.
z. B. `/var/www/documents/start.html`
Angreifer kann Dateinamen `../../../etc/passwd` eingeben,
um kritische Dateien zu lesen:

```
/var/www/documents/../../../etc/passwd
   = /etc/passwd
```

(SQL) Command Injection

- Nutzer fordert (über Web-Formular) eine Datei an,
- System speichert gewünschten Namen in `String foo`
- und führt dann `cat $foo` aus.

Angreifer gibt `"bar ; rm -rf *"` ein.
Dann wird `cat bar ; rm -rf *` ausgeführt!
Ähnlicher Angriff nicht auf Shell, sonden auf SQL-Shell

Sicherheit (Zusammenfassung)

die gezeigten Beispiele illustrieren nur das Prinzip, die tatsächlichen Angriffe sind komplizierter (aber nicht sehr!)
Siehe aktuelle Nachrichten z. B. auf
`http://www.heise.de/security/`
(Wiederholung:) Schwachstelle ist meist der leichtsinnige Nutzer (/Administrator)!

Zusammenfassung

Informatik — Teilgebiete

Teilgebiete (mit Beispielen/Semester)

- theoretische
  Algorithmen, Datenstrukturen, Komplexität (1.)
- praktische
  Programmier-Konzepte, -Sprachen (z. B. OO) (1.)
  Betriebssysteme (2.)
- technische
  Rechner-Architektur, Netze (2.)
- angewandte
  Datenbanken (4.)

Informatik — Definition

Gegenstand der Informatik:

- (vereinfachend:) Algorithmen
- (genauer:) ... zur symbolischen Verarbeitung von Information

Informatik ist:

- *Wissenschaft:* Erforschen von Grundlagen, Entwickeln und Begründen von Modellen, Methoden, Werkzeugen
- *Ingenieur-Disziplin:* Auswählen und Anwenden von Modellen, Methoden, Werkzeugen

Sprache

- Die *Methoden* der Informatik sind fachspezifisch.
- Die *Sprache* der Informatik ist die *Mathematik*.
- D. h.: Fragen, Antworten, Begründungen werden als mathematische bzw. logische Aussagen formuliert.

Mathematische Modelle

Algebra: betrachtet *Operationen* auf *Mengen*.

- Als Mengen: ganze Zahlen, komplexe Zahlen, Vektoren, Wahrheitswerte, Restklassen, Listen, Prozesse
- Operationen: Addition, Subtraktion, Multiplikation, Verkettung, Nacheinander-, Parallel-Ausführung

Wenn gewisse Voraussetzungen erfüllt sind, entstehen gewisse *Strukturen*.
Beispiel: Operation ist assoziativ $\rightarrow$ beliebig gruppierbar, unter Beachtung der Reihenfolge.

(Software-)System-Design-Regeln

- Aufgabentrennung schafft Strukturen:

  Rechneraufbau, CPU-Aufbau, Betriebssysteme (Prozesse, Speicher, Dateien), OSI-Schichtenmodell
- Hierarchie (= Baumstruktur) verringert Komplexität

  strukturiertes Programmieren (Ausdrücke, Blöcke, Methoden, Klassen, Pakete), Datenstrukturen (Bäume)

Autotool–Highscore-Auswertung

```
20 :    38562
15 :    38507
10 :    38509
```

Buchpreise (Programmierung/Softwaretechnik) gesponsor
von Siemens, Leipzig.
Siehe auch Angebote des Sponsors für Praktika und Diplomarbeiten `http://www.imn.htwk-leipzig.de/~waldmann/edu/diplom/`

Testfragen

1. Was versteht man unter der von-Neumann-Architektur? Welche Rolle spielt sie a) für die Rechner-Hardware, b) für die Software (Bsp: Programmiersprache Java)?
2. Wodurch unterscheiden sich Steuerwerk und Rechenwerk einer CPU *funktionell*? Welche Information aus dem Rechenwerk kann die Reihenfolge der Befehlsabarbeitung beeinflussen?
3. Warum benutzen man im Computer meist eine Zahlendarstellung mit Basis zwei?
4. Wieviele Bit benötigt man wenigstens zur Speicherung eins Datums (mit fixierter Jahreszahl)?
5. Welche Vor- bzw. Nachteile bieten Gleitkommazahlen gegenüber exakten ganzen Zahlen?

6. Wonach ist der Datentyp `Boolean = { false, true }` benannt?

7. Wieviele 3-stellige Boolesche Funktionen gibt es?

8. Welche Gemeinsamkeite/Unterschiede gibt es beim Rechnen mit Wahrheitswerten/ganzen Zahlen?

9. Welche physikalischen Prinzipien/Gesetze kann man zur Speicherung von Information ausnutzen?

0. Wieso verläufen bei einer DVD-9 (zweilagig) beide Spuren in verschiedene Richtungen (von innen nach außen, von außen nach innen)?

1. Eine Speicher-Hierarchie gibt eine Kosten/Nutzen-Beziehung vor. Wie kann man diese durch Cache-Speicher beeinflussen?

2. warum erfolgt die Speicherung von Dateien auf Festplatten weder rein sequentiell noch rein wahlfrei?

3. Ist ein Mehr-Nutzer/mehr-Prozeß-Betriebssystem sinnlos für einen Rechner, der nur von einer Person benutzt wird?

4. Welche Vorteile bietet ein Time-Sharing- gegenüber einem Stapel-Betriebssystem?

5. Warum soll ein BS robust sein? (Beispiel?)

6. Warum sollte man BS und grafische Oberfläche voneinander trennen?

7. Vielen modernen Prozessoren unterstützen relative Adressierung direkt. Was bedeutet das? Welchen Zweck hat das?

8. hängen relative Adressierung und AUslagerung automatisch zusammen?

9. unterscheiden Sie Algorithmus, Prozeß, Sprache, Programm

0. wie können verschiedene Programme (Programmteile) Informationen austauschen?

1. Aufgabe zu 5 Philosophen (Diener läßt nur 4, nur 3 an den Tisch)

2. Warum hat jede UNIX-Datei hat ≥ 9 verschiedene Zugriffsrechte?

3. Welche Vorteile haben Standard-Verzeichnis-Strukturen

(nicht nur unter UNIX)

4. Durch welche sprachlichen Mittel kann der Kontrollfuß eines Java-Programms geändert werden?
5. Für welche Datei-Typen ist verlustbehaftete Kompression nicht akzeptabel?
6. Wieso gibt es keinen Code der Größe ?? mit Lönge ?? und Hamming-Abstand??
7. Wie kann Chiffren mit Einzelbuchstaben-Substitution angreifen?
8. Wieso kann man eine mit RSA veschlüsselte Nachricht knacken, wenn die Faktorisierung des Modules (= öffentlichen Schlüssels) bekannt ist?
9. Wie können Sie einen Geschätfspartner auf

elektronischem Wege davon überzeugen, daß Sie einem Vertrag zustimmen?

0. Warum benutzt man für große, hochverfügbare Netze nicht das Ethernet-Prinzip? Welches dann?